

博客

登录 | 注册

Q

tanyit的专栏

目录视图

摘要视图

RSS 订阅

个人资料



笑一笑

+ 加关注

发私信

访问：484511次

积分：4972

等级：BLOG > 5

排名：第4964名

原创：75篇

转载：102篇

译文：0篇

评论：58条

文章搜索

Q

文章分类

corejava (19)

strust 2 (16)

spring (6)

ssh (4)

ajax (2)

ejb (0)

oracle/mysql (13)

JavaScript (1)

Web容器之Tomcat (4)

Java Web编程Servlet (5)

Java Web编程JSP (23)

J2EE容器之JBoss (0)

JDBC (1)

Web Services (0)

J2EE容器之WebLogic (0)

SVN/CVS及编程规范 (0)

软件测试 (0)

java 软件官网下载地址 (0)

corejava 编程问题 (2)

web 编程问题 (9)

ssh 编程问题 (3)

设计模式 (2)

编程工具优化 (11)

【评论送书】机器学习、Spring MVC、Android

CSDN日报20170507 ——《技能终将过时，而能力与时俱进》

CSDN技术直播：php实战微信公众号开发！

原

Hibernate -annotation 学习笔记 1 马士兵

标签：hibernate session string 数据库 class generator

2011-11-18 09:59 50032人阅读 评论(1) 快速回复

分类：

hibernate (6)

我要收藏

返回顶部

版权声明：本文为博主原创文章，未经博主允许不得转载。

目录(?)

[+]

第1课 课程内容. 6

第2课Hibernate UML图. 6

第3课 风格. 7

第4课 资源. 7

第5课 环境准备. 7

第6课 第一个示例HibernateHelloWorld 7

第7课 建立Annotation版本的HellWorld 9

第8课 什么是O/RMapping 11

一、 定义：. 11

二、 Hibernate的创始人：. 11

三、 Hibernate做什么：. 12

四、 Hibernate存在的原因：. 12

五、 Hibernate的优缺点：. 12

六、 Hibernate使用范围：. 12

第9课Hibernate的重点学习：Hibernate的对象关系映射. 12

一、 对象---关系映射模式. 12

二、 常用的O/R映射框架：. 13

第10课 模拟Hibernate原理(OR模拟) 13

一、 项目名称. 13

二、 源代码. 13

第11课Hibernate基础配置. 15

一、 提纲. 15

二、 介绍MYSQL的图形化客户端. 16

三、 Hibernate.cfg.xml：hbm2ddl.auto 16

四、 搭建日志环境并配置显示DDL语句. 16

五、 搭建JUnit环境. 16

六、 ehibernate.cfg.xml：show_sql 17

七、 hibernate.cfg.xml：format_sql 17

八、 表名和类名不同，对表名进行配置. 17

九、 字段名和属性相同. 17

十、 字段名和属性名不同. 17

关闭

1/94

文件上传与下载 (10)

OA (4)

休闲圣地 (6)

ext (1)

其他 (8)

hibernate (7)

linux (3)

操作系统 (1)

高级程序员 (1)

面试 (2)

算法 (1)

电子商务网站开发 (4)

资料总结 (1)

开源项目研究 (2)

java项目常用的公共代码 (1)

java架构师之路 (0)

文件管理 (1)

文章存档

2015年04月 (4)

2015年02月 (1)

2013年11月 (2)

2013年08月 (1)

2013年07月 (2)

展开

阅读排行

The word is not correctly (55780)

Hibernate -annotation 学 (50031)

java 数字格式化：小数点 (19861)

getRemoteAddr()和getR (19689)

java.lang.ClassCastExc (14150)

javax.servlet.ServletExc (14001)

oracle 分析函数 (11293)

java 如何把别人的项目引 (9382)

org.hibernate.NonUniqu (9084)

<c:import>,<c:redirect>,< (8326)

评论排行

Hibernate -annotation 学 (15)

The word is not correctly (4)

SmartUpload综合 (3)

java 数字格式化：小数点 (3)

According to TLD, tag js (3)

oracle 分析函数 (3)

JSP标签编程 1 (2)

研究生毕业3年，年薪从 (2)

AOP IOC (2)

getRemoteAddr()和getR (2)

推荐文章

* CSDN日报20170505 ——《创业时该不该用新手程序员》

* 程序员要拥抱变化，聊聊Android即将支持的Java 8

* 彻底弄懂prepack与webpack的关系

* 用 TensorFlow 做个聊天机器人

十一、 不需要(持久化)persistence的字段. 18

十二、 映射日期与时间类型，指定时间精度. 18

十三、 映射枚举类型. 19

第12课 使用hibernate工具类将对象模型生成关系模型. 19

第13课ID主键生成策略. 20

一、 Xml方式. 20

<generator>元素(主键生成策略) 20

二、 annotateon方式. 21

1、 AUTO默认. 21

2、 IDENTITY 22

3、 SEQUENCE 22

4、 为Oracle指定定义的Sequence 22

5、 TABLE - 使用表保存id值. 23

三、 联合主键. 24

1、 xml方式. 24

2、 annotation方式. 27

第14课Hibernate核心开发接口(重点) 29

一、 Configuration(AnnotationConfiguration) 29

二、 SessionFactory 29

三、 Session 29

1、 管理一个数据库的任务单元. 29

2、 save(); 29

3、 delete() 29

4、 load() 29

5、 Get() 30

6、 load()与get()区别. 31

7、 update() 31

8、 saveOrUpdate() 32

9、 clear() 32

10、 flush() 33

11、 evict() 33

第15课 持久化对象的三种状态. 35

一、 瞬时对象(TransientObject)：. 35

二、 持久化对象(PersistentObject)：. 35

三、 离线对象(DetachedObject)：. 35

四、 三种状态的区分：. 35

五、 总结：. 35

第16课 关系映射(重点) 36

一、 一对一 关联映射. 36

(一) 唯一-外键关联-单向(unilateralism) 37

(二) 唯一-外键关联-双向. 40

(三) 主键关联-单向(不重要) 41

(四) 主键关联-双向(不重要) 44

(五) 联合主键关联(Annotation方式) 44

二、 component (组件) 关联映射. 45

(一) Component关联映射：. 45

(二) User实体类：. 45

(三) Contact值对象：. 46

(四) xml--User映射文件(组件映射)：. 46

(五) annotateon注解. 46

(六) 导出数据库输出SQL语句：. 47

(七) 数据表结构：. 47

(八) 组件映射数据保存：. 47

三、 多对一— 单向. 48

快速回复

我要收藏

返回顶部

关闭

http://blog.csdn.net/tanyit/article/details/6987279#_Toc251597219

2/94

* 分布式机器学习的集群方案介绍之HPC实现

* Android 音频系统：从AudioTrack 到 AudioFlinger

最新评论

java.lang.ClassCastException: c十二-李晓洁: 应该是类型转换错误

Vector和ArrayList,LinkedList,Ha丁国华: mark

java 数字格式化：小数点、百分比陈晓婵: 感谢楼主分享，学习了！

作为软件工程师，你必须知道的2eson_15: 不错

tomcat session 持久化yebai: xuexile

The word is not correctly spelled 纳兰白帝: 谢谢

java 显示图片预览DIV 蜉蝣若梦: 至少也要告诉我们这个怎么用吧？？？

java.lang.NoClassDefFoundError独爱cyjs: hibernate-core的版本跟hibernate-entitymanager一致就行

研究生毕业3年，年薪从5万到20Geory王: 挺厉害的

java 数字格式化：小数点、百分比清澈@Cherry: 多谢楼主

(一) 对象模型图：. 48

(二) 关系模型：. 48

(三) 关联映射的本质：. 48

(四) 实体类. 48

(五) xml方式：映射文件：. 49

(六) annotation 50

(七) 多对一 存储(先存储group(对象持久化状态后，再保存user))：. 50

(八) 重要属性-cascade(级联)：. 51

(九) 多对一 加载数据. 51

四、 一对多- 单向. 51

(一) 对象模型：. 52

(二) 关系模型：. 52

(三) 多对一、一对多的区别：. 52

(四) 实体类. 52

(五) xml方式：映射. 52

(六) annotateon注解. 53

(七) 导出至数据库(hbm2ddl)生成的SQL语句：. 53

(八) 一对多 单向存储实例：. 53

(九) 生成的SQL语句：. 54

(十) 一对多，在一的一端维护关系的缺点：. 54

(十一) 一对多 单向数据加载：. 54

(十二) 加载生成SQL语句：. 54

五、 一对多- 双向. 54

(一) xml方式：映射. 55

(二) annotateon方式注解. 55

(三) 数据保存：. 56

(四) 关于inverse属性：. 56

(五) Inverse和cascade区别：. 56

(六) 一对多双向关联映射总结：. 57

六、 多对多- 单向. 57

(一) 实例场景：. 57

(二) 对象模型：. 57

(三) 关系模型：. 57

(四) 实体类. 57

(五) xml方式：映射. 58

(六) annotation注解方式. 58

(七) 生成SQL语句. 59

(八) 数据库表及结构：. 59

(九) 多对多关联映射 单向数据存储：. 59

(十) 多对多关联映射 单向数据加载：. 61

七、 多对多- 双向. 61

(一) xml方式：映射. 61

(二) annotation注解方式. 62

八、 关联关系中的CRUD_Cascade_Fetch 63

九、 集合映射. 63

十、 继承关联映射. 64

(一) 继承关联映射的分类：. 64

(二) 对象模型：. 64

(三) 单表继承SINGLE_TABLE：. 64

(四) 具体表继承JOINED：. 70

(五) 类表继承TABLE_PER_CLASS 72

(六) 三种继承关联映射的区别：. 74

第17课hibernate树形结构(重点) 75

一、 节点实体类：. 75

 快速回复

 我要收藏


 返回顶部

http://blog.csdn.net/tanyit/article/details/6987279#_Toc251597219

关闭

3/94

- 二、 xml方式：映射文件：. 75
- 三、 annotation注解. 76
- 四、 测试代码：. 76
- 五、 相应的类代码：. 76
- 第18课 作业-学生、课程、分数的映射关系. 79
 - 一、 设计. 79
 - 二、 代码：. 79
 - 三、 注意. 80
- 第19课Hibernate查询(Query Language) 80
 - 一、 Hibernate可以使用的查询语言. 80
 - 二、 实例一. 80
 - 三、 实体一测试代码：. 82
 - 四、 实例二. 86
 - 五、 实例二测试代码. 87
- 第20课Query by Criteria(QBC) 89
 - 一、 实体代码：. 89
 - 二、 Restrictions用法. 90
 - 三、 工具类Order提供设置排序方式. 91
 - 四、 工具类Projections提供对查询结果进行统计与分组操作. 91
 - 五、 QBC分页查询. 92
 - 六、 QBC复合查询. 92
 - 七、 QBC离线查询. 92
- 第21课Query By Example(QBE) 92
 - 一、 实例代码. 92
- 第22课Query.list与query.iterate(不太重要) 93
 - 一、 query.iterate查询数据. 93
 - 二、 query.list()和query.iterate()的区别. 94
 - 三、 两次query.list() 94
- 第23课 性能优化策略. 95
- 第24课hibernate缓存. 95
 - 一、 Session级缓存(一级缓存) 95
 - 二、 二级缓存. 95
 - 1、 二级缓存的配置和使用：. 96
 - 2、 二级缓存的开启：. 96
 - 3、 指定二级缓存产品提供商：. 96
 - 4、 使用二级缓存. 97
 - 5、 应用范围. 99
 - 6、 二级缓存的管理：. 99
 - 7、 二级缓存的交互. 100
 - 8、 总结. 102
 - 三、 查询缓存. 102
 - 四、 缓存算法. 103
- 第25课 事务并发处理. 104
 - 一、 数据库的隔离级别：并发性作用。 . 104
 - 1、 Mysql查看数据库隔离级别：. 104
 - 2、 Mysql数据库修改隔离级别：. 104
 - 二、 事务概念(ACID) 104
 - 三、 事务并发时可能出现问题. 104
- 第26课hibernate悲观锁、乐观锁. 105
 - 一、 悲观锁. 105
 - 1、 悲观锁的实现. 105
 - 2、 悲观锁的适用场景：. 105
 - 3、 实例：. 105
 - 4、 悲观锁的使用. 106

 快速回复 我要收藏 返回顶部

- 5、 执行输出SQL语句：. 106
- 二、 乐观锁. 107

第1课 课程内容

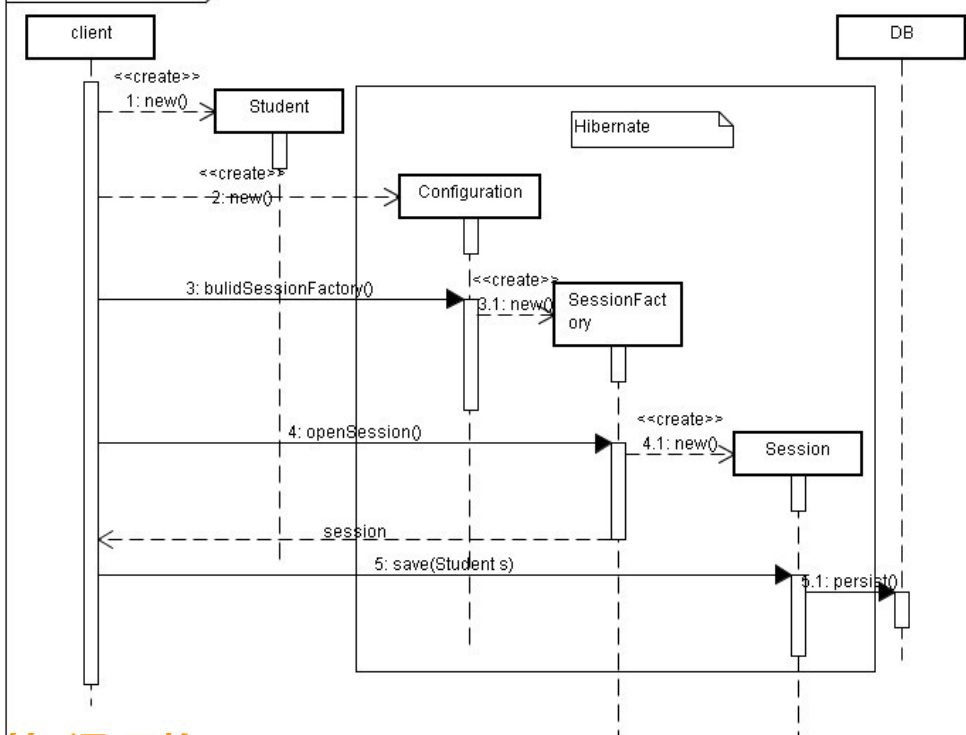
- 1、 HelloWorld
 - a) Xml
 - b) Annotation
- 2、 **hibernate**原理模拟-什么是O/RMapping以及为什么要有O/RMapping
- 3、 常风的O/R框架
- 4、 Hibernate基础配置
- 5、 Hibernate核心接口介绍
- 6、 对象的三种状态
- 7、 ID生成策略
- 8、 关系映射
- 9、 Hibernate查询(HQL)
- 10、 在Struts基础上继续完美BBS2009
- 11、 性能优化
- 12、 补充话题

快速回复

我要收藏

返回顶部

第2课 Hibernate UML图



第3课 风格

- 1、 先脉络，后细节
- 2、 先操作、后原理
- 3、 重Annotation，轻xml配置文件
 - a) JPA (可以认为EJB3的一部分)
 - b) Hibernate- extension

第4课 资源

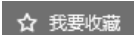
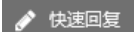
- 1、 <http://www.hibernate.org>
 - a) hibernate-distribution-3.3.2.GA-dist.zip
 - b) hibernate-annotations-3.4.0.GA.zip
 - c) slf4j-1.5.10.zip (hibernate内部日志用)
- 2、 hibernatezh_CN文档
- 3、 hibernateannotation references

第5课 环境准备

- 1、 下载hibernate3.3.2
- 2、 下载hibernate-annotations-3.4.0
- 3、 注意阅读hibernate compatibility matrix
- 4、 下载slf4j 1.5.8

第6课 第一个示例Hibernate HelloWorld

- 1、 建立新的**Java**项目, 名为hibernate_0100_HelloWorld
- 2、 学习建立User-library-hibernate, 并加入相应的jar包
 - a) 项目右键-build path-configure build path-add library
 - b) 选择User-library, 在其中新建library, 命名为hibernate
 - c) 在该library中加入hibernate所需的jar名
 - i. Hibernatecore
 - ii. /lib/required
 - iii. Slf-nopjar
- 3、 引入**MySQL**的JDBC驱动名
- 4、 在mysql中建立对应的**数据库**以及表
 - a) Create databasehibernate;
 - b) Usehibernate;
 - c) Createtable Student (id int primary key, name varchar(20),age int);
- 5、 建立hibernate配置文件hibernate.cfg.xml
 - a) 从参考文档中copy
 - b) 修改对应的数据库连接
 - c) 注释提暂时不需要的内容
- 6、 建立Student类
- 7、 建立Student映射文件Student.hbm.xml
 - a) 参考文档
- 8、 将映射文件加入到hibernate.cfg.xml
 - a) 参考文档
- 9、 写**测试**类Main, 在Main中对Student对象进行直接的存储测试



```
[java] view plain copy print ?

01. public static void main(String[] args) {
02.
03.     Configuration cfg = null;
04.     SessionFactory sf = null;
05.     Session session = null;
06.
07.     Student s = new Student();
08.     s.setId(2);
09.     s.setName("s1");
10.     s.setAge(1);
11.
12.     /*
13.      * org.hibernate.cfg.Configuration类的作用:
14.      * 读取hibernate配置文件(hibernate.cfg.xml或hibernate.properties)的.
15.      * new Configuration()默认是读取hibernate.properties
16.      * 所以使用new Configuration().configure();来读取hibernate.cfg.xml配置文件
17.      */
18.     cfg = new Configuration().configure();
19.
20.     /*
21.      * 创建SessionFactory
22.      * 一个数据库对应一个SessionFactory
23.      * SessionFactory是线程安全的。
24.      */
25.     sf = cfg.buildSessionFactory();
26.
27.     try {
28.         //创建session
29.         //此处的session并不是web中的session
30.         //session只有在用时,才建立connection,session还管理缓存。
31.         //session用完后,必须关闭。
32.         //session是非线程安全,一般是一个请求一个session.
```

关闭

```
33.         session = sf.openSession();
34.
35.         //手动开启事务(可以在hibernate.cfg.xml配置文件中配置自动开启事务)
36.         session.beginTransaction();
37.         /*
38.          * 保存数据,此处的数据是保存对象,这就是hibernate操作对象的好处,
39.          * 我们不用写那么多的JDBC代码,只要利用session操作对象,至于hibernat如何存在对象,这不
           需要我们去关心它,
40.          * 这些都有hibernate来完成。我们只要将对象创建完后,交给hibernate就可以了。
41.          */
42.         session.save(s);
43.         session.getTransaction().commit();
44.     } catch (HibernateException e) {
45.         e.printStackTrace();
46.         //回滚事务
47.         session.getTransaction().rollback();
48.     } finally {
49.         //关闭session
50.         session.close();
51.         sf.close();
52.     }
53. }
```

[快速回复](#)[我要收藏](#)[返回顶部](#)

1、FAQ：

- 要调用new Configuration().configure().buildSessionFactory(),而不是省略
- org.hibernate.cfg.Configuration类的作用：
 - *读取hibernate配置文件(hibernate.cfg.xml或hiberante.properties)的。
 - *new Configuration()默认是读取hibernate.properties
 - *所以使用new Configuration().configure();来读取hibernate.cfg.xml配置文件

注意：在hibernate里的操作都应该放在事务里

第7课 建立Annotation版本的HellWorld

注意：要求hibernate3.0版本以后支持

- 创建teacher表, create table teacher(id int primary key,namevarchar(20),title varchar(10));
- 创建Teacher类

```
[java] view plain copy print ?
01. public class Teacher {
02.     private int id;
03.     private String name;
04.     private String title;
05.     //设置主键使用@Id
06.     public int getId() {
07.         return id;
08.     }
09.     public void setId(int id) {
10.         this.id = id;
11.     }
12.     public String getName() {
13.         return name;
14.     }
15.     public void setName(String name) {
16.         this.name = name;
17.     }
18.     public String getTitle() {
19.         return title;
20.     }
21.     public void setTitle(String title) {
22.         this.title = title;
23.     }
24. }
```

1、在hibernate library中加入annotation的jar包

- Hibernateannotations jar
- Ejb3persistence jar

[关闭](#)

- c) Hibernatecommon annotations jar
d) 注意文档中没有提到hibernate-common-annotations.jar文件
- 2、 参考Annotation文档建立对应的注解

```
[java] view plain copy print ?
01. import javax.persistence.Entity;
02. import javax.persistence.Id;
03. /** @Entity 表示下面的这个Teacher是一个实体类
04.  * @Id 表示主键Id*/
05. @Entity /**
06. public class Teacher {
07.     private int id;
08.     private String name;
09.     private String title;
10.     //设置主键使用@Id
11.     @Id /**
12.     public int getId() {
13.         return id;
14.     }
15.     public void setId(int id) {
16.         this.id = id;
17.     }
18.     public String getName() {
19.         return name;
20.     }
21.     public void setName(String name) {
22.         this.name = name;
23.     }
24.     public String getTitle() {
25.         return title;
26.     }
27.     public void setTitle(String title) {
28.         this.title = title;
29.     }
}}
```

[快速回复](#)[我要收藏](#)[返回顶部](#)

- 1、 在hibernate.cfg.xml中建立映射<mapping class=.../>

```
<mapping class="com.wjt276.hibernate.model.Teacher"/>
```

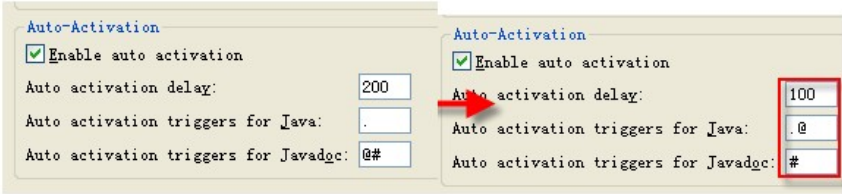
注意：<mapping>标签中使用的是class属性，而不是resource属性，并且使用小数点(.)导航，而不是"/"

- 2、 参考文档进行测试

```
[java] view plain copy print ?
01. public static void main(String[] args) {
02.     Teacher t = new Teacher();
03.     t.setId(1);
04.     t.setName("s1");
05.     t.setTitle("中级");
06.     //注此处并不是使用org.hibernate.cfg.Configuration来创建Configuration
07.     //而使用org.hibernate.cfg.AnnotationConfiguration来创建Configuration，这样就可以使用Annotation功能
08.     Configuration cfg = new AnnotationConfiguration();
09.
10.     SessionFactory sf = cfg.configure().buildSessionFactory();
11.     Session session = sf.openSession();
12.
13.     session.beginTransaction();
14.     session.save(t);
15.     session.getTransaction().commit();
16.
17.     session.close();
18.     sf.close();
19. }
```

- 1、 FAQ：@ 后不给提示

解决方法：windows→Preferences→search "Content Assist"设置Auto-Activation如下：



第8课 什么是O/R Mapping

一、定义：

ORM(ObjectRelational Mapping)---是一种为了解决面向对象与关系型数据库存在的互不匹配的现象的技术。简单说：ORM是通过使用描述对象和数据库之间映射的元数据，将Java程序中的对象自动持久化到关系数据库中。本质上就是将数据从一种形式转换到另外一种形式。



分层后，上层不需要知道下层是如何做了。
分层后，不可以循环依赖，一般是单向依赖。

一、Hibernate的创始人：

Gavin King

二、Hibernate做什么：

- 1、就是将对象模型(实体类)的东西存入关系模型中，
 - 2、实体中类对应关系型库中的一个表，
 - 3、实体类中的一个属性会对应关系型数据库表中的一个列
 - 4、实体类的一个实例会对应关系型数据库表中的一条记录。
- %%将对象数据保存到数据库、将数据库数据读入到对象中%%

OOA---面向对象的分析、面向对象的设计

OOD---设计对象化

OOP---面向对象的开发

阻抗不匹配---例JAVA类中有继承关系，但关系型数据库中不存在这个概念这就是阻抗不匹配。Hibernate可以解决这个问题

三、Hibernate存在的原因：

- 1、解决阻抗不匹配的问题；
- 2、目前不存在完整的面向对象的数据库(目前都是关系型数据库);
- 3、JDBC操作数据库很繁琐
- 4、SQL语句编写并不是面向对象
- 5、可以在对象和关系表之间建立关联来简化编程
- 6、O/RMapping简化编程
- 7、O/RMapping跨越数据库平台

8、hibernate_0200_OR_Mapping_Simulation

四、Hibernate的优缺点：

- 1、不需要编写的SQL语句(不需要编辑JDBC)，只需要操作相应的对象就可以了，就可以能够存储、更新、删除、加载对象，可以提高生产效；
- 2、因为使用Hibernate只需要操作对象就可以了，所以我们的开发更对象化了；
- 3、使用Hibernate，移植性好(只要使用Hibernate标准开发，更换数据库时，只需要配置相应的配置文件就可以了，不需要做其它任务的操作)；
- 4、Hibernate实现了透明持久化：当保存一个对象时，这个对象不需要继承Hibernate中的任何类、实现任何接口，只是个纯粹的单纯对象—称为POJO对象(最纯粹的对象—这个对象没有继承第三方框架的任何类和实现它的任何接口)
- 5、Hibernate是一个没有侵入性的框架，没有侵入性的框架我们一般称为轻量级框架
- 6、Hibernate代码测试方便。

五、Hibernate使用范围：

1. 针对某一个对象，简单的将它加载、编辑、修改，且修改只是对单个对象(而不是批量的进行修改)，这种情况比较适用；
2. 对象之间有着很清晰的关系(例：多个用户属于一个组(多对一)、一个组有多个用户(一对多))
3. 聚集性操作：批量性添加、修改时，不适合使用Hibernate(O/R映射框架都不适合使用)；
4. 要求使用数据库中特定的功能时不适合使用,因为Hibernate不使用SQL语句；

[快速回复](#)[我要收藏](#)[返回顶部](#)

第9课 Hibernate的重点学习：Hibernate的对象关系映射

一、对象---关系映射模式

- l 属性映射；
- l 类映射；
- l 关联映射：
- n 一对一；
- n 一对多；
- n 多对多。

二、常用的O/R映射框架：

- 1、Hibernate
 - 2、ApacheOJB
 - 3、JDO(是SUN提出的一套标准—Java数据对象)
 - 4、Toplink(Orocle公司的)
 - 5、EJB(2.0X中有CMP;3.0X提出了一套“Java持久化API”---JPA)
 - 6、IBatis(非常的轻量级，对JDBC做了一个非常非常轻量级的包装，严格说不是O/R映射框架，而是基于SQL的映射(提供了一套配置文件，把SQL语句配置到文件中，再配置一个对象进去，只要访问配置文件时，就可得到对象))
 - 7、JAP(是SUN公司的一套标准)
- a) 意愿统一天下

第10课 模拟Hibernate原理(OR模拟)

我们使用一个项目来完成

功能：有一个配置文件，文件中完成表名与类名对象，字段与类属性对应起来。

测试驱动开发

一、项目名称

hibernate_0200_OR_Mapping_Simulation

二、源代码

[java] view plain copy print ?

```
01. Test类:
02. public static void main(String[] args) throws Exception{
03.
```

[关闭](#)

```

04.         Student s = new Student();
05.         s.setId(10);
06.         s.setName("s1");
07.         s.setAge(1);
08.
09.         Session session = new Session();//此Session是我们自己定义的Session
10.
11.         session.save(s);
12.     }

```

[java] view plain copy print ?

```

01. Session类
02. import java.lang.reflect.Method;
03. import java.sql.Connection;
04. import java.sql.DriverManager;
05. import java.sql.PreparedStatement;
06. import java.util.HashMap;
07. import java.util.Map;
08. import com.wjt276.hibernate.model.Student;
09. public class Session {
10.     String tableName = "_Student";
11.     Map<String,String> cfs = new HashMap<String,String>();
12.     String[] methodNames;//用于存入实体类中的get方法数组
13.     public Session(){
14.         cfs.put("_id", "id");
15.         cfs.put("_name", "name");
16.         cfs.put("_age", "age");
17.         methodNames = new String[cfs.size()];
18.     }
19.     public void save(Student s) throws Exception{
20.         String sql = createSQL();//创建SQL串
21.         Class.forName("com.mysql.jdbc.Driver");
22.         Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/hibernate",'
23.         PreparedStatement ps = conn.prepareStatement(sql);
24.         //
25.         for(int i = 0; i < methodNames.length; i++){
26.             Method m = s.getClass().getMethod(methodNames[i]);//返回一个 Method 对象，它反映
此 Class 对象所表示的类或接口的指定公共成员方法
27.             Class r = m.getReturnType();//返回一个 Class 对象，该对象描述了此 Method 对象所表示的
方法的正式返回类型
28.             if(r.getName().equals("java.lang.String")) {
29.                 //对带有指定参数的指定对象调用由此 Method 对象表示的底层方法。
30.                 //个别参数被自动解包，以便与基本形参相匹配，基本参数和引用参数都需服从方法调用转
换
31.                 String returnValue = (String)m.invoke(s);
32.                 ps.setString(i + 1, returnValue);
33.             }
34.             if(r.getName().equals("int")) {
35.                 Integer returnValue = (Integer)m.invoke(s);
36.                 ps.setInt(i + 1, returnValue);
37.             }
38.             if(r.getName().equals("java.lang.String")) {
39.                 String returnValue = (String)m.invoke(s);
40.                 ps.setString(i + 1, returnValue);
41.             }
42.             System.out.println(m.getName() + "|" + r.getName());
43.         }
44.         ps.executeUpdate();
45.         ps.close();
46.         conn.close();
47.     }
48.     private String createSQL() {
49.         String str1 = "";
50.         int index = 0;
51.
52.         for(String s : cfs.keySet()){
53.             String v = cfs.get(s);//取出实体类成员属性
54.             v = Character.toUpperCase(v.charAt(0)) + v.substring(1);//将成员属性第一个字符大
写
55.             methodNames[index] = "get" + v;//拼实体类成员属性的getter方法
56.             str1 += s + ",";//根据表中字段名拼成字段串
57.             index ++;
58.         }
59.         str1 = str1.substring(0,str1.length() -1);
60.         String str2 = "";

```

快速回复

我要收藏

返回顶部

```
61.         //根据表中字段数, 拼成?串
62.         for (int i = 0; i < cfs.size(); i++){      str2 += "?,"; }
63.         str2 = str2.substring(0, str2.length() - 1);
64.         String sql = "insert into " + tableName + "
        (" + str1 + ") " + " values (" + str2 + ")";
65.         System.out.println(sql);
66.         return sql;
67.     }}
```

第11课 Hibernate基础配置

一、提纲

- 1、 对应项目：hibernate_0300_BasicConfiguration
- 2、 介绍MYSQL的图形化客户端
- 3、 Hibernate.cfg.xml：hbm2ddl.auto
 - a) 先建表还是先建实体类
- 4、 搭建日志环境并配置显示DDL语句
- 5、 搭建Junit环境
 - a) 需要注意Junit的Bug
- 6、 ehibernate.cfg.xml: show_sql
- 7、 hibernate.cfg.xml:format_sql
- 8、 表名和类名不同，对表名进行配置
 - a) Annotation:@Table
 - b) Xml:自己查询
- 9、 字段名和属性相同
 - a) 默认为@Basic
 - b) Xml中不用写column
- 10、 字段名和属性名不同
 - a) Annotation:@Column
 - b) Xml:自己查询
- 11、 不需要psersistence的字段
 - a) Annotation:@Transient
 - b) Xml:不写
- 12、 映射日期与时间类型，指定时间精度
 - a) Annotation:@Temporal
 - b) Xml:指定type
- 13、 映射枚举类型
 - a) Annotation:@Enumerated
 - b) Xml:麻烦
- 14、 字段映射的位置(field或者get方法)
 - a) Best practice:保持field和get/set方法的一致
- 15、 @Lob
- 16、 课外：CLOB BLOB类型的数据存取
- 17、 课外：Hibernate自定义数据类型
- 18、 Hibernate类型

二、介绍MYSQL的图形化客户端

这样的软件网络很多，主要自己动手做

三、Hibernate.cfg.xml：hbm2ddl.auto

在SessionFactory创建时，自动检查数据库结构，或者将数据库schema的DDL导出到数据库. 使用 create-drop 时,在显式关闭SessionFactory时，将drop掉数据库schema.取值 validate | update | create | create-drop

四、搭建日志环境并配置显示DDL语句

我们使用slf接口，然后使用log4j的实现。

- 1、 首先引入log4j的jar包(log4j-1.2.14.jar)，
- 2、 然后再引入slf4j实现LOG4J和适配器jar包(slf4j-log4j12-1.5.8.jar)
- 3、 最后创建log4j的配置文件(log4j.properties)，并加以修改，只要保留
log4j.logger.org.hibernate.tool.hbm2ddl=debug

五、搭建Junit环境

[快速回复](#)[我要收藏](#)[返回顶部](#)

- 1、首先引入Junit 类库 jar包 (junit-4.8.1.jar)
- 2、在项目名上右键→new→Source Folder→输入名称→finish
- 3、注意，你对哪个包进行测试，你就在测试下建立和那个包相同的包
- 4、建立测试类，需要在测试的方法前面加入"@Test"

```
[java] view plain copy print ?
01. public class TeacherTest {
02.
03.     private static SessionFactory sf = null;
04.
05.     @BeforeClass//表示Junit此类被加载到内存中就执行这个方法
06.     public static void beforClass(){
07.         sf = new AnnotationConfiguration().configure().buildSessionFactory();
08.     }
09.     @Test//表示下面的方法是测试用的。
10.     public void testTeacherSave(){
11.         Teacher t = new Teacher();
12.         t.setId(6);
13.         t.setName("s1");
14.         t.setTitle("中級");
15.         Session session = sf.openSession();
16.         session.beginTransaction();
17.         session.save(t);
18.         session.getTransaction().commit();
19.
20.         session.close();
21.     }
22.
23.     @AfterClass//Junit在类结果时，自动关闭
24.     public static void afterClass(){
25.         sf.close();
26.     }}
```

快速回复

☆ 我要收藏

^ 返回顶部

六、ehibernate.cfg.xml : show_sql

输出所有SQL语句到控制台. 有一个另外的选择是把org.hibernate.SQL这个log category设为debug。

取值：true | false

七、hibernate.cfg.xml :format_sql

在log和console中打印出更漂亮的SQL。

取值：true | false

True样式：

```
[java] view plain copy print ?
01. 16:32:39,750 DEBUG SchemaExport:377 -
02.     create table Teacher (
03.         id integer not null,
04.         name varchar(255),
05.         title varchar(255),
06.         primary key (id)
07.     )
```

False样式：

```
[java] view plain copy print ?
01. 16:33:40,484 DEBUG SchemaExport:377 - create table Teacher (id integer not null, name varchar(
```

八、表名和类名不同，对表名进行配置

Annotation：使用 @Table(name="tableName") 进行注解

例如：

```
[java] view plain copy print ?
01. /**
```

关闭

```
02.  * @Entity 表示下面的这个Teacher是一个实体类
03.  * @Table 表示映射到数据表中的表名, 其中的name参数表示"表名称"
04.  * @Id 表示主键Id, 一般放在getXXX前面
05.  */
06.  @Entity
07.  @Table(name="_teacher")
08.  public class Teacher {
09.  [.....]
10.  }
```

Xml :

```
[java] view plain copy print ?
01.  <class name="Student" table="_student">
```

九、 字段名和属性相同

Annotation : 默认为@Basic

注意 : 如果在成员属性没有加入任何注解, 则默认在前面加入了@Basic

Xml中不用写column

十、 字段名和属性名不同

Annotation : 使用@Column(name="columnName")进行注解

例如 :

```
[java] view plain copy print ?
01.  /**
02.  * @Entity 表示下面的这个Teacher是一个实体类
03.  * @Table 表示映射到数据表中的表名, 其中的name参数表示"表名称"
04.  * @Column 表示实体类成员属性映射数据表中的字段名, 其中name参数指定一个新的字段名
05.  * @Id 表示主键Id
06.  */
07.  @Entity
08.  @Table(name="_teacher")
09.  public class Teacher {
10.
11.      private int id;
12.      private String name;
13.      private String title;
14.
15.      //设置主键使用@Id
16.      @Id
17.      public int getId() {
18.          return id;
19.      }
20.
21.      @Column(name="_name")//字段名与属性不同时
22.      public String getName() {
23.          return name;
24.      }
25.      .....
```


Xml :


```
[java] view plain copy print ?
01.  <property name="name" column="_name"/>
```

十一、 不需要(持久化)persistence的字段

就是不实体的某个成员属性不需要存入数据库中

Annotation : 使用@Transient 进行注解就可以了。

 快速回复

 我要收藏

 返回顶部

例如：

```
[java] view plain copy print ?
01. @Transient
02. public String getTitle() {
03.     return title;
04. }
```

Xml：不写(就是不需要对这个成员属性进行映射)

十二、映射日期与时间类型，指定时间精度

Annotation：使用@Temporal(value=TemporalType)来注解表示日期和时间的注解

其中TemporalType有三个值：TemporalType.TIMESTAMP 表示yyyy-MM-dd HH:mm:ss

TemporalType.DATE 表示yyyy-MM-dd

TemporalType.TIME 表示HH:mm:ss

```
[java] view plain copy print ?
01. @Temporal(value=TemporalType.DATE)
02. public Date getBirthDate() {
03.     return birthDate;
04. }
```

快速回复

我要收藏

返回顶部

注意：当使用注解时，属性为value时，则这个属性名可以省略，例如：@Temporal(TemporalType)

Xml：使用type属性指定hibernate类型

```
[java] view plain copy print ?
01. <property name="birthDate" type="date"/>
```

注意：hibernate日期时间类型有：date, time, timestamp，当然您也可以使用Java包装类

十二、映射枚举类型

Annotation：使用@Enumerated(value=EnumType)来注解表示此成员属性为枚举映射到数据库

其中EnumType有二个值：①EnumType.STRING 表示直接将枚举名称存入数据库

②EnumType.ORDINAL 表示将枚举所对应的数值存入数据库

Xml：映射非常的麻烦，先要定义自定义类型，然后再使用这个定义的类型.....

第12课 使用hibernate工具类将对象模型生成关系模型

(也就是实体类生成数据库中的表),完整代码如下：

```
package com.wjt276.hibernate;
```

```
import org.hibernate.cfg.AnnotationConfiguration;
```

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

```
import org.hibernate.tool.hbm2ddl.SchemaExport;
```

```
/**
```

```
 * Hibernate工具<br/>
```

```
 * 将对象模型生成关系模型(将对象生成数据库中的表)
```

```
 * 把hbm映射文件(或Annotation注解)转换成DDL
```

```
 * 生成数据表之前要求已经存在数据库
```

```
 * 注：这个工具类建立好后，以后就不用建立了。以后直接Copy来用。
```

```
 * @author wjt276
```

```
 * @version 1.0 2009/10/16
```

```
 */
```


关闭


```

public class ExportDB {
    public static void main(String[] args){
        /*org.hibernate.cfg.Configuration类的作用：
        *读取hibernate配置文件(hibernate.cfg.xml或hibernate.properties)的。
        *new Configuration()默认是读取hibernate.properties
        *所以使用new Configuration().configure();来读取hibernate.cfg.xml配置文件
        */
        Configuration cfg = new AnnotationConfiguration().configure();
        /*org.hibernate.tool.hbm2ddl.SchemaExport工具类：
        *需要传入Configuration参数
        *此工具类可以将类导出生成数据库表
        */
        SchemaExport export = new SchemaExport(cfg);
        /** 开始导出
        *第一个参数：script是否打印DDL信息
        *第二个参数：export是否导出到数据库中生成表
        */
        export.create(true, true);
    }
}

```

运行刚刚建立的ExportDB类中的main()方法，进行实际的导出类。

 快速回复

 我要收藏

 返回顶部

第13课 ID主键生成策略

一、Xml方式

<id>标签必须配置在<class>标签内第一个位置。由一个字段构成主键，如果是复杂主键<composite-id>标签被映射的类必须定义对应数据库表主键字段。大多数类有一个JavaBeans风格的属性，为每一个实例包含唯一的标识。<id>元素定义了该属性到数据库表主键字段的映射。

```

<id
    name="propertyName"                (1)
    type="typename"                    (2)
    column="column_name"                (3)
    unsaved-value="null|any|none|undefined|id_value" (4)
    access="field|property|ClassName"    (5)
    node="element-name|@attribute-name|element/@attribute.">

    <generatorclass="generatorClass"/>
</id>

```

(1) name (可选): 标识属性的名字(实体类的属性)。

(2) type (可选): 标识Hibernate类型的名字(省略则使用hibernate默认类型)，也可以自己配置其它hibernate类型(integer, long, short, float,double, character, byte, boolean, yes_no, true_false)

(2) length(可选):当type为varchar时，设置字段长度

(3) column (可选 - 默认为属性名): 主键字段的名字(省略则取name为字段名)。

(4) unsaved-value (可选 - 默认为一个切合实际 (sensible) 的值): 一个特定的标识属性值，用来标志该实例是刚刚创建的，尚未保存。这可以把这种实例和从以前的session中装载过 (可能又做过修改--译者注) 但未再次持久化的实例区分开来。

(5) access (可选 - 默认为property): Hibernate用来访问属性值的策略。

如果 name属性不存在，会认为这个类没有标识属性。

unsaved-value 属性在Hibernate3中几乎不再需要。

还有一个另外的<composite-id>定义可以访问旧式的多主键数据。我们强烈不建议使用这种方式。

<generator>元素(主键生成策略)

主键生成策略是必须配置

用来为该持久化类的实例生成唯一的标识。如果这个生成器实例需要某些配置值或者初始化参数，用<param>元素来传递。


```
<id name="id" type="long" column="cat_id">
    <generator class="org.hibernate.id.TableHiLoGenerator">
        <param name="table">uid_table</param>
        <param name="column">next_hi_value_column</param>
    </generator>
</id>
```

所有的生成器都实现org.hibernate.id.IdentifierGenerator接口。这是一个非常简单的接口；某些应用程序可以选择提供他们自己特定的实现。当然，Hibernate提供了很多内置的实现。下面是一些内置生成器的快捷名字：

increment

用于为long, short或者int类型生成 唯一标识。只有在没有其他进程往同一张表中插入数据时才能使用。在集群下不要使用。

identity

对DB2,MySQL, MS SQL Server,Sybase和HypersonicSQL的内置标识字段提供支持。返回的标识符是long, short 或者int类型的。(数据库自增)

sequence

在DB2,PostgreSQL, **Oracle**, SAPDB, McKoi中使用序列 (sequence), 而在Interbase中使用生成器(generator)。返回的标识符是long, short或者 int类型的。(数据库自增)

hilo

使用一个高/低位算法高效的生成long, short 或者 int类型的标识符。给定一个表和字段 (默认分别hibernate_unique_key和next_hi) 作为高位值的来源。高/低位算法生成的标识符只在一个特定的数据库中是唯一的。

seqhilo

使用一个高/低位**算法**来高效的生成long, short 或者 int类型的标识符，给定一个数据库序列 (sequence)的名字。

uuid

用一个128-bit的UUID算法生成字符串类型的标识符，这在一个网络中是唯一的 (使用了IP地址)。UUID被编码为一个32位16进制数字的字符串，它的生成是由hibernate生成，一般不会重复。

UUID包含：IP地址，JVM的启动时间 (精确到1/4秒)，系统时间和一个计数器值 (在JVM中唯一)。在Java代码中不可能获得MAC地址或者内存地址，所以这已经是我们在不使用JNI的前提下的能做的最好实现了

guid

在MS SQL Server 和 MySQL 中使用数据库生成的GUID字符串。

native

根据底层数据库的能力选择identity,sequence 或者hilo中的一个。(数据库自增)

assigned

让应用程序在save()之前为对象分配一个标示符。这是 <generator>元素没有指定时的默认生成策略。(如果是手动分配，则需要设置此配置)

select

通过数据库触发器选择一些唯一主键的行并返回主键值来分配一个主键。

foreign

使用另外一个相关联的对象的标识符。通常和<one-to-one>联合起来使用。

二、annotate方式

使用@GeneratedValue (strategy=GenerationType) 注解可以定义该标识符的生成策略

Strategy有四个值：

- ① 、AUTO- 可以是identity column类型,或者sequence类型或者table类型,取决于不同的底层数据库。相当于native
- ② 、TABLE- 使用表保存id值
- ③ 、IDENTITY- identity column
- ④ 、SEQUENCE- sequence

注意：auto是默认值，也就是说没有后的参数则表示为auto

1、AUTO默认

```
@Id
@GeneratedValue
public int getId() {
```

快速回复

☆ 我要收藏

返回顶部

```

        return id;
    }
    或
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    public int getId() {
        return id;
    }

```

- 1、对于mysql, 使用auto_increment
- 2、对于oracle使用hibernate_sequence(名称固定)

2、IDENTITY

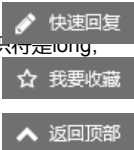
```

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    public int getId() {
        return id;
    }

```

对DB2,MySQL, MS SQL Server,Sybase和HypersonicSQL的内置标识字段提供支持。返回的标识符是long, short 或者int类型的。(数据库自增)

注意：此生成策略不支持Oracle



3、SEQUENCE

```

    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE)
    public int getId() {
        return id;
    }

```

在DB2,PostgreSQL,Oracle, SAP DB, McKoi中使用序列 (sequence), 而在Interbase中使用生成器 (generator)。返回的标识符是long, short或者 int类型的。(数据库自增)

注意：此生成策略不支持MySQL

4、为Oracle指定定义的Sequence

- a)、首先需要在实体类前面申明一个Sequence如下：

方法：@SequenceGenerator(name="SEQ_Name",sequenceName="SEQ_DB_Name")

参数注意：SEQ_Name:表示为申明的这个Sequence指定一个名称，以便使用

SEQ_DB_Name:表示为数据库中的Sequence指定一个名称。

两个参数的名称可以一样。

```

@Entity
@SequenceGenerator(name="teacherSEQ",sequenceName="teacherSEQ_DB")
public class Teacher {
    .....
}

```

- b)、然后使用@GeneratedValue注解

方法：@GeneratedValue(strategy=GenerationType.SEQUENCE,generator="SEQ_Name")

参数：strategy：固定为GenerationType.SEQUENCE

Generator:在实体类前面申明的sequence的名称

```

@Entity
@SequenceGenerator(name="teacherSEQ",sequenceName="teacherSEQ_DB")
public class Teacher {
    private int id;

    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE,generator="teacherSEQ")
    public int getId() {
        return id;
    }
}

```

```
}}
```

5、TABLE - 使用表保存id值

原理：就是在数据库中建立一个表，这个表包含两个字段，一个字段表示名称，另一个字段表示值。每次在添加数据时，使用第一个字段的名称，来取值作为添加数据的ID，然后再给这个值累加一个值再次存入数据库，以便下次取出使用。

Table主键生成策略的定义：

```
@javax.persistence.TableGenerator(
    name="Teacher_GEN",          //生成策略的名称
    table="GENERATOR_TABLE",     //在数据库生成表的名称
    pkColumnName = "pk_key",     //表中第一个字段的字段名 类型为varchar,key
    valueColumnName = "pk_value", //表中第二个字段的字段名 int ,value
    pkColumnValue="teacher",     //这个策略中使用该记录的第一个字段的值(key值)
    initialValue = 1,           //这个策略中使用该记录的第二个字段的值(value值)初始化值
    allocationSize=1            //每次使用数据后累加的数值
)
```

这样执行后，会在数据库建立一个表，语句如下：

```
create tableGENERATOR_TABLE (pk_key varchar(255),pk_value integer )
```

结构：

Field	Type	Null	Key	Default	Extra
pk_key	varchar(255)	YES		NULL	
pk_value	int(11)	YES		NULL	

并且表建立好后，就插入了一个记录，如下：

pk_key	pk_value
teacher	2

注：这条记录的pk_value值为2，是因为刚刚做例程序时，已经插入一条记录了。初始化为1。

使用TABLE主键生成策略：

@Entity

```
@javax.persistence.TableGenerator(
    name="Teacher_GEN",          //生成策略的名称
    table="GENERATOR_TABLE",     //在数据库生成表的名称
    pkColumnName = "pk_key",     //表中第一个字段的字段名 类型为varchar,key
    valueColumnName = "pk_value", //表中第二个字段的字段名 int ,value
    pkColumnValue="teacher",     //这个策略中使用该记录的第一个字段的值(key值)
    initialValue = 1,           //这个策略中使用该记录的第二个字段的值(value值)初始化值
    allocationSize=1            //每次使用数据后累加的数值
)
```

```
public class Teacher {
    private int id;
    @Id
    @GeneratedValue(strategy=GenerationType.TABLE,generator="Teacher_GEN")
    public int getId() {
        return id;}}

```

注意：这样每次在添加Teacher记录时，都会先到GENERATOR_TABLE表取pk_key=teacher的记录后，使用pk_value值作为记录的主键。然后再给这个pk_value字段累加1，再存入到GENERATOR_TABLE表中，以便下次使用。

这个表可以给无数的表作为主键表，只是添加一条记录而以(需要保证table、pkColumnName、valueColumnName三个属性值一样就可以了。)，这个主键生成策略可以跨数据库平台。

三、联合主键

复合主键(联合主键)：多个字段构成唯一性。

1、xml方式

a) 实例场景：核算期间

// 核算期间

```
public class FiscalYearPeriod {
```

快速回复

我要收藏

返回顶部

```
private int fiscalYear; //核算年
private int fiscalPeriod; //核算月
private Date beginDate; //开始日期
private Date endDate; //结束日期
private String periodSts; //状态
public int getFiscalYear() {
    return fiscalYear;
}
public void setFiscalYear(int fiscalYear) {
    this.fiscalYear = fiscalYear;
}
public int getFiscalPeriod(){ return fiscalPeriod;}
public void setFiscalPeriod(int fiscalPeriod) {
    this.fiscalPeriod =fiscalPeriod;
}
public DategetBeginDate() {return beginDate;}
public void setBeginDate(DatebeginDate) { this.beginDate = beginDate; }
public Date getEndDate(){return endDate;}
public void setEndDate(DateendDate) { this.endDate = endDate; }
public StringgetPeriodSts() { return periodSts;}
public voidsetPeriodSts(String periodSts) {this.periodSts = periodSts;}
}
```

[快速回复](#)[我要收藏](#)[返回顶部](#)

复合主键的映射，一般情况把主键相关的属性抽取出来单独放入一个类中。而这个类是有要求的：必需实现序列化接口(java.io.Serializable)(可以保存到磁盘上)，为了确定这个复合主键类所对应对象的唯一性就会产生比较，对象比较就需要复写对象的hashCode()、equals()方法(复写方法如下图片)，然后在类中引用这个复合主键类



b) 复合主键类：

复合主键必需实现java.io.Serializable接口

```
public class FiscalYearPeriodPKimplements java.io.Serializable {
    private int fiscalYear;//核算年
    private int fiscalPeriod;//核算月
    public int getFiscalYear() {
        return fiscalYear;
    }
    public void setFiscalYear(int fiscalYear) {
        this.fiscalYear = fiscalYear;
    }
    public int getFiscalPeriod(){
        return fiscalPeriod;
    }
    public void setFiscalPeriod(int fiscalPeriod) {
```

```

    this.fiscalPeriod =fiscalPeriod;
}
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime* result + fiscalPeriod;
    result = prime* result + fiscalYear;
    return result;
}
@Override
public boolean equals(Object obj){
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() !=obj.getClass())
        return false;
    FiscalYearPeriodPKother = (FiscalYearPeriodPK) obj;
    if (fiscalPeriod != other.fiscalPeriod)
        return false;
    if (fiscalYear != other.fiscalYear)
        return false;
    return true;
}
}

```

c) 实体类：(中引用了复合主键类)

```

public class FiscalYearPeriod{
    private FiscalYearPeriodPK fiscalYearPeriodPK;//引用 复合主键类
    private Date beginDate;//开始日期
    private Date endDate;//结束日期
    private String periodSts;//状态
    public FiscalYearPeriodPK getFiscalYearPeriodPK() {
        return fiscalYearPeriodPK;
    }
    public void setFiscalYearPeriodPK(FiscalYearPeriodPKfiscalYearPeriodPK) {
        this.fiscalYearPeriodPK = fiscalYearPeriodPK;
    }
}

```

.....

d) FiscalYearPeriod.hbm.xml映射文件

```

<hibernate-mapping>
    <class name="com.bjsxt.hibernate.FiscalYearPeriod"table="t_fiscal_year_period">
        <composite-id name="fiscalYearPeriodPK">
            <key-property name="fiscalYear"/>
            <key-property name="fiscalPeriod"/>
        </composite-id>
        <property name="beginDate"/>
        <property name="endDate"/>
        <property name="periodSts"/>
    </class>
</hibernate-mapping>

```

e) 导出数据库输出SQL语句：

```

create table t_fiscalYearPeriod (fiscalYear integer not null, fiscalPeriodinteger not null, beginDate datetime,
endDate datetime, periodSts varchar(255),primary key (fiscalYear, fiscalPeriod))//实体映射到数据就是两个

```



字段构成复合主键

f) 数据库表结构：

```
mysql> show tables;
+-----+
| Tables_in_hibernate_composite_mapping |
+-----+
| t_fiscalyearperiod |
+-----+
1 row in set (0.00 sec)

mysql> desc t_fiscalyearperiod;
+-----+-----+-----+-----+-----+-----+
| Field      | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| fiscalYear | int(11)   | NO   | PRI |          |       |
| fiscalPeriod | int(11)   | NO   | PRI |          |       |
| beginDate  | datetime  | YES  |     | NULL    |       |
| endDate    | datetime  | YES  |     | NULL    |       |
| periodSts  | varchar(255) | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)
```

g) 复合主键关联映射数据存储：

```
session =HibernateUtils.getSession();
```

```
tx =session.beginTransaction();
```

```
FiscalYearPeriod fiscalYearPeriod = new FiscalYearPeriod();
```

```
//构造复合主键
```

```
FiscalYearPeriodPK pk = new FiscalYearPeriodPK();
```

```
pk.setFiscalYear(2009);
```

```
pk.setFiscalPeriod(11);
```

```
fiscalYearPeriod.setFiscalYearPeriodPK(pk);//为对象设置复合主键
```

```
fiscalYearPeriod.setEndDate(new Date());
```

```
fiscalYearPeriod.setBeginDate(new Date());
```

```
fiscalYearPeriod.setPeriodSts("Y");
```

```
session.save(fiscalYearPeriod);
```

h) 执行输出SQL语句：

```
Hibernate: insert into t_fiscalYearPeriod (beginDate, endDate, periodSts,fiscalYear, fiscalPeriod) values (?, ?, ?, ?, ?)
```

注：如果再存入相同复合主键的记录，就会出错。

i) 数据的加载：

数据加载非常简单，只是主键是一个对象而以，不是一个普通属性。

2、annotation方式

下面是定义组合主键的几种语法:

将组件类注解为@Embeddable,并将组件的属性注解为@Id

将组件的属性注解为@EmbeddedId

将类注解为@IdClass,并将该实体中所有属于主键的属性都注解为@Id

a) 将组件类注解为@Embeddable,并将组件的属性注解为@Id

组件类：

@Embeddable

```
public class TeacherPK implements java.io.Serializable{
    private int id;
    private String name;
    public int getId() {return id; }
    public void setId(int id) {this.id = id;}
    public String getName() { return name;}
    public void setName(Stringname) { this.name = name;}
```



```

@Override
public boolean equals(Object o) { .....}

@Override
public int hashCode() { return this.name.hashCode(); }
}

```

将组件类的属性注解为@Id,实体类中组件的引用

```

@Entity
public class Teacher {
    private TeacherPK pk;
    private String title;

    @Id
    public TeacherPK getPk(){
        return pk;
    }
}

```

b) 将组件的属性注解为@EmbeddedId

注意：只需要在实体类中表示复合主键属性前注解为@Entity，表示此主键是一个复合主键注意了，复合主键类不需要任何的注意。

```

@Entity
public class Teacher {
    private TeacherPK pk;
    private String title;

    @EmbeddedId
    public TeacherPK getPk(){
        return pk;
    }
}

```

c) 类注解为@IdClass，主键的属性都注解为@Id

需要将复合主键类建立好，不需要进行任何注解

在实体类中不需要进行复合主键类的引用

需要在实体类前面注解为@IdClass,并且指定一个value属性,值为复合主键类的class

需要在实体类中进行复合主键成员属性前面注解为@Id

如下：

```

@Entity
@IdClass(TeacherPK.class)
public class Teacher {
    //private TeacherPK pk;//不再需要
    private int id;
    private String name;

    @Id
    public int getId() {return id; }
    public void setId(int id) { this.id = id; }

    @Id
    public String getName() {return name;}
    public void setName(Stringname) {this.name = name;}
}

```

第14课 Hibernate核心开发接口(重点)

一、Configuration(AnnotationConfiguration)

作用：进行配置信息的管理

 快速回复

 我要收藏

 返回顶部

目标：用来产生SessionFactory

可以在configure方法中指定hibernate配置文件，默认(不指定)时在classpath下加载hibernate.cfg.xml文件

加载默认的hibernate的配置文件

```
sessionFactory = new AnnotationConfiguration().configure().buildSessionFactory();
```

加载指定hibernate的配置文件

```
sessionFactory=new AnnotationConfiguration().configure("hibernate.xml").buildSessionFactory();
```

只需要关注一个方法:buildSessionFactory();

二、SessionFactory

作用：主要用于产生Session的工厂(数据库连接池)

当它产生一个Session时，会从数据库连接池取出一个连接，交给这个Session

```
Session session = sessionFactory.getCurrentSession();
```

并且可以通过这个Session取出这个连接

关注两个方法：

getCurrentSession():表示当前环境没有Session时，则创建一个，否则不用创建

openSession()：表示创建一个Session(3.0以后不常用)，使用后需要关闭这个Session

两方法的区别：

①、openSession永远是每次都打开一个新的Session,而getCurrentSession不是，是从上下文找，当前没有Session时，才创建一个新的Session

②、OpenSession需要手动close,getCurrentSession不需要手动close，事务提交自动close

③、getCurrentSession界定事务边界

上下文：

所指的上下文是指hibernate配置文件(hibernate.cfg.xml)中的“current_session_context_class”所指的值：(可取值：jta|thread|managed|custom.Class)

```
<property name="current_session_context_class">thread</property>
```

常用的是：①、thread：是从上下文找、只有当前没有Session时，才创建一个新的Session，主要从数据库界定事务

②、jta：主要从分布式界定事务，运行时需要Application Server来支持(Tomcat不支持)

③、managed:不常用

④、custom.Class:不常用

三、Session

1、管理一个数据库的任务单元

2、save();

```
session.save(Object)
```

session的save方法是向数据库中保存一个对象，这个方法产生对象的三种状态

3、delete()

```
session.delete(Object)
```

Object对象需要有ID

对象删除后，对象状态为Transient状态

4、load()

格式：Session.load(Class arg0,Serializable arg1) throws HibernateException

*arg0:需要加载对象的类，例如：User.class

*arg1:查询条件(实现了序列化接口的对象)：例"4028818a245fdd0301245fdd06380001"字符串已经实现了序列化接口。如果是数值类类型，则hibernate会自动使用包装类，例如 1

* 此方法返回类型为Object，但返回的是代理对象。

* 执行此方法时不会立即发出查询SQL语句。只有在使用对象时，它才发出查询SQL语句，加载对象。

* 因为load方法实现了lazy(称为延迟加载、懒加载)

* 延迟加载：只有真正使用这个对象的时候，才加载(才发出SQL语句)

*hibernate延迟加载实现原理是代理方式。

* 采用load()方法加载数据，如果数据库中没有相应的记录，则会抛出异常对象不找到(org.hibernate.ObjectNotFoundException)

```
try {  
    session =sf.openSession();  
    session.beginTransaction();
```




```

User user =(User)session.load(User.class,1);

//只有在使用对象时，它才发出查询SQL语句，加载对象。
System.out.println("user.name=" + user.getName());

//因为此的user为persistent状态，所以数据库进行同步为龙哥。
user.setName("发哥");

session.getTransaction().commit();
} catch (HibernateException e) {
    e.printStackTrace();
    session.getTransaction().rollback();
} finally{
    if (session != null){
        if(session.isOpen()){
            session.close();
        }
    }
}
}

```

快速回复

我要收藏

返回顶部

5、 Get()

格式：Session.get(Class arg0,Serializable arg1)方法

* arg0:需要加载对象的类，例如：User.class

* arg1:查询条件(实现了序列化接口的对象)：

例"4028818a245fdd0301245fdd06380001"字符串已经实现了序列化接口。如果是基数类型，则hibernate会自动转换成包装类，如 1

返回值：此方法返回类型为Object，也就是对象，然后我们再强行转换为需要加载的对象就可以了。

如果数据不存在，则返回null;

注：执行此方法时立即发出查询SQL语句。加载User对象

加载数据库中存在的数据库，代码如下：

```

try {
    session =sf.openSession();
    session.beginTransaction();

    * 此方法返回类型为Object，也就是对象，然后我们再强行转换为需要加载的对象就可以了。
    如果数据不存在，则返回null
    * 执行此方法时立即发出查询SQL语句。加载User对象。
    */

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

    //数据加载完后的状态为persistent状态。数据将与数据库同步。
    System.out.println("user.name=" + user.getName());

    //因为此的user为persistent状态，所以数据库进行同步为龙哥。
    user.setName("龙哥");

    session.getTransaction().commit();
} catch (HibernateException e) {
    e.printStackTrace();
    session.getTransaction().rollback();
} finally{
    if (session != null){
        if(session.isOpen()){
            session.close();
        }
    }
}
}

```

```

    }
}

```

6、load()与get()区别

- ①、不存在对应记录时表现不一样；
- ②、load返回的是代理对象，等到真正使用对象的内容时才发出sql语句，这样就要求在第一次使用对象时，要求session处于open状态，否则出错
- ③、get直接从数据库加载，不会延迟加载

get()和load()只根据主键查询，不能根据其它字段查询，如果想根据非主键查询，可以使用HQL

7、update()

- ①、用来更新detached对象，更新完成后转为为persistent状态(默认更新全部字段)
- ②、更新transient对象会报错(没有ID)
- ③、更新自己设定ID的transient对象可以(默认更新全部字段)
- ④、persistent状态的对象，只要设定字段不同的值，在session提交时，会自动更新(默认更新全部字段)
- ⑤、更新部分更新的字段(更改了哪个字段就更新哪个字段的内容)

a) 方法1：update/updatable属性

xml：设定<property>标签的update属性，设置在更新时是否参数更新

```
<property name="name" update="false"/>
```

注意：update可取值为true(默认)：参与更新；false：更新时不参与更新

annotation:设定@Column的updatable属性值，true参与更新，false：不参与更新

```
@Column(updatable=false)
```

```
public String getTitle(){return title;}
```

注意：此种方法很少用，因为它不灵活

b) 方法二：dynamic-update属性

注意：此方法目前只适合xml方式，JAP1.0annotation没有对应的

在实体类的映射文件中的<class>标签中，使用dynamic-update属性，true：表示修改了哪个字段就更新哪个字段，其它字段不更新，但要求是同一个session(不能跨session),如果跨了session同样会更新所有的字段内容。

```
<class name="com.bjst.Student" dynamic-update="true">
```

代码：

```
@Test
```

```
public void testUpdate5() {
    Session session = sessionFactory.getCurrentSession();
    session.beginTransaction();
    Student s =(Student)session.get(Student.class, 1);
    s.setName("zhangsan5");
    //提交时，会只更新name字段，因为此时的s为persistent状态
    session.getTransaction().commit();
    s.setName("z4");
    Session session2 = sessionFactory.getCurrentSession();
    session2.beginTransaction();
    //更新时，会更新所有的字段，因为此时的s不是persistent状态
    session2.update(s);
    session2.getTransaction().commit(); }

```

如果需要跨session实现更新修改的部分字段，需要使用session.merge()方法,合并字段内容

```
@Test
```

```
public void testUpdate6() {
    Session session = sessionFactory.getCurrentSession();
    session.beginTransaction();
    Student s =(Student)session.get(Student.class, 1);
    s.setName("zhangsan6");
    session.getTransaction().commit();
    s.setName("z4");
    Session session2 = sessionFactory.getCurrentSession();
    session2.beginTransaction();
    session2.merge(s);

```



```
session2.getTransaction().commit();
```

这样虽然可以实现部分字段更新，但这样会多出一条select语句，因为在字段数据合并时，需要比较字段内容是否已变化，就需要从数据库中取出这条记录进行比较

c) 使用HQL(EJBQL)面向对象的查询语言(建议)

```
@Test
```

```
public void testUpdate7() {
```

```
    Session session = sessionFactory.getCurrentSession();
```

```
    session.beginTransaction();
```

```
    Query q = session.createQuery(
```

```
"update Student s set s.name='z5' where s.id = 1");
```

```
    q.executeUpdate();
```

```
    session.getTransaction().commit();
```

```
}
```

8、 saveOrUpdate()

在执行的时候hibernate会检查，如果对象在数据库中已经有对应的记录(是指主键)，则会更新update，否则会添加数据save

9、 clear()

清除session缓存

无论是load还是get，都会首先查找缓存(一级缓存，也叫session级缓存)，如果没有，才会去数据库

clear()方法可以强制清除session缓存

```
Session session= sessionFactory.getCurrentSession();
```

```
session.beginTransaction();
```

```
Teacher t =(Teacher)session.load(Teacher.class, 1);
```

```
System.out.println(t.getName());
```

```
session.clear();
```

```
Teacher t2 =(Teacher)session.load(Teacher.class, 1);
```

```
System.out.println(t2.getName());
```

```
session.getTransaction().commit();
```

注意：这样就会发出两条SELECT语句，如果把session.clear()去除，则只会发出一条SELECT语句，因为第二次load时，是使用session缓存中ID为1的对象，而这个对象已经在第一次load到缓存中了。

10、 flush()

在hibernate中也存在flush这个功能，在默认的情况下session.commit()之前时，其实执行了一个flush命令。

Session.flush功能：

n 清理缓存；

n 执行sql(确定是执行SQL语句(确定生成update、insert、delete语句等),然后执行SQL语句。)

Session在什么情况下执行flush:

① 默认在事务提交时执行；

注意：flush时，可以自己设定,使用session.setFlushMode(FlushMode)来指定。

```
session.setFlushMode(FlushMode);
```

FlushMode的枚举值：

l FlushMode.ALWAYS：任务一条SQL语句，都会flush一次

l FlushMode.AUTO：自动flush(默认)

l FlushMode.COMMIT: 只有在commit时才flush

l FlushMode.MANUAL：手动flush。

l FlushMode.NEVER：永远不flush 此选项在性能优化时可能用，比如session取数据为只读时用，这样就不需要与数据库同步了

注意：设置flush模式时，需要在session开启事务之前设置。


② 可以显示的调用flush；

③ 在执行查询前，如:iterate.

注：如果主键生成策略是uuid等不是由数据库生成的，则session.save()时并不会发出SQL语句，只有flush时才会发出SQL语句，但如果主键生成策略是native由数据库生成的，则session.save的同时就发出SQL语句。

11、 evict()

 快速回复

 我要收藏

 返回顶部

例如：session.evict(user)

作用：从session缓存(EntityEntries属性)中逐出该对象

但是与commit同时使用，会抛出异常

```
session = HibernateUtils.getSession();
```

```
tx = session.beginTransaction();
```

```
User1 user = new User1();
```

```
user.setName("李四");
```

```
user.setPassword("123");
```

```
user.setCreateTime(new Date());
```

```
user.setExpireTime(new Date());
```

//利用Hibernate将实体类对象保存到数据库中，因为user主键生成策略采用的是uuid，所以调用完成save后，只是将user纳入session的管理，不会发出insert语句，但是id已经生成，session中的existsInDatabase状态为false

```
session.save(user);
```

```
session.evict(user);//从session缓存(EntityEntries属性)中逐出该对象
```

//无法成功提交，因为hibernate在清理缓存时，在session的临时集合(insertions)中取出user对象进行更新，更新后需要更新entityEntries属性中的existsInDatabase为true,而我们采用evict已经将user从session中逐出了，所以找不到相关数据,无法更新，抛出异常。

```
tx.commit();
```

解决在逐出session缓存中的对象不抛出异常的方法：

在session.evict()之前进行显示的调用session.flush()方法就可以了。

```
session.save(user);
```

//flush后hibernate会清理缓存，会将user对象保存到数据库中，将session中的insertions中的user对象清除，并且会设置session中的existsInDatabase状态为false

```
session.flush();
```

```
session.evict(user);//从session缓存(EntityEntries属性)中逐出该对象
```

//可以成功提交，因为hibernate在清理缓存时，在Session的insertions中集合中无法找到user对象所以不会发出insert语句，也不会更新session中existsInDatabase的状态。

```
tx.commit();
```

第15课 持久化对象的三种状态

一、瞬时对象(Transient Object)：

使用new操作符初始化的对象不是立刻就持久的。它们的状态是瞬时的，也就是说它们没有任何跟数据库表相关联的行为，只要应用不再引用这些对象(不再被任何其它对象所引用)，它们的状态将会丢失，并由垃圾回收机制回收

二、持久化对象(Persistent Object)：

持久实例是任何具有数据库标识的实例，它有持久化管理器Session统一管理，持久实例是在事务中进行操作的---它们的状态在事务结束时同数据库进行同步。当事务提交时，通过执行SQL的INSERT、UPDATE和DELETE语句把内存中的状态同步到数据库中。

三、离线对象(Detached Object)：

Session关闭之后，持久化对象就变为离线对象。离线表示这个对象不能再与数据库保持同步，它们不再受hibernate管理。

 快速回复

 我要收藏

 返回顶部

- u 一张主表、多张子表
- 7、 组件映射
- u @Embeddable
- u @Embedded

一、 一对一关联映射

- 2 两个对象之间是一一对应的关系，如Person-IdCard(人—身份证号)
- 2 有两种策略可以实现一对一的关联映射
 - Ø 主键关联：即让两个对象具有相同的主键值，以表明它们之间的——对应的关系；数据库表不会有额外的字段来维护它们之间的关系，仅通过表的主键来关联。
 - Ø 唯一外键关联：外键关联，本来是用于多对一的配置，但是如果加上唯一的限制之后，也可以用来表示一对一关联关系。

对象模型

实体类：

```
/** 人 - 实体类 */
public class Person {
    private int id;
    private String name;
    public int getId() {return id; }
    public void setId(int id) {this.id = id;}
    public String getName() {return name;}
    public void setName(Stringname) {this.name = name;}
}

/**身份证-实体类*/
public class IdCard {
    private int id;
    private String cardNo;
    public int getId() {return id;}
    public void setId(int id) {this.id = id;}
    public String getCardNo(){ return cardNo;}
    public void setCardNo(StringcardNo) {this.cardNo = cardNo;}
}
```

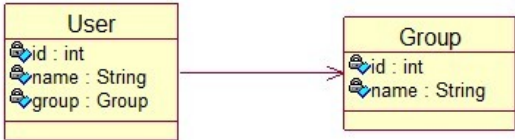
快速回复

我要收藏

返回顶部

(一) 唯一外键关联-单向(unilateralism)

- 1、 说明：人—> 身份证号(PersonàIdCard)，从IdCard看不到Person对象
- 2、 对象模型需要在Person类中持有IdCard的一个引用idCard，则IdCard中没有Person的引用



3、 关系模型

关系模型目的：是实体类映射到关系模型(数据库中)，是要求person中添加一个外键指向idcard

t_person			t_idcard	
id	name	idcard(唯一)	id	cardNo
5	菜 10	100	100	888888888888888888
6	容祖儿	200	200	999999999999999999

4、 实体类：

注：IdCard是被引用对象，没有变化。

```
/** 人 - 实体类 */
```

```

public class Person {
    private int id;
    private String name;
    private IdCard idCard;//引用IdCard对象
    public int getId() {return id; }
    public void setId(int id) {this.id = id;}
    public String getName() {return name;}
    public void setName(String name){this.name = name;}
    public IdCard getIdCard() { return idCard;}
    public void setIdCard(IdCardidCard) {this.idCard = idCard;}
}

```

5、xml映射

IdCard实体类的映射文件：

因为IdCard是被引用的，所以没有什么特殊的映射

```

<hibernate-mapping>
  <class name="com.wjt276.hibernate.IdCard" table="t_idcard">
    <id name="id" column="id">
      <generator class="native"/>
    </id>
    <property name="cardNo"/>
  </class>
</hibernate-mapping>

```

Person实体类的映射文件

在映射时需要添加一个外键的映射，就是指定IdCard的引用的映射。这样映射到数据库时，就会自动添加一个字段并作用外键指向被引用的表

```

<hibernate-mapping>
  <class name="com.wjt276.hibernate.Person" table="t_person">
    <id name="id" column="id">
      <generator class="native"/>
    </id>
    <property name="name"/>
    <!-- <many-to-one>:在多的一端(当前Person一端)，加入一个外键(当前为idCard)指向一的一端(当前IdCard),但
    多对一 关联映射字段是可以重复的，所以需要加入一个唯一条件unique="true",这样就可以此字段唯一了。-->
    <many-to-one name="idCard" unique="true"/>
  </class>
</hibernate-mapping>

```

注意：这里的<many-to-one>标签中的name属性值并不是数据库中的字段名，而是Person实体类中引用IdCard对象成员属性的getxxx方法后面的xxx(此处是getIdCard，所以是idCard)，要求第一个字段小写。如果不指定column属性，则数据库中的字段名同name值

6、annotation注解映射

注意IdCard是被引用对象，除正常注解，无需要其它注解

```

/**身份证*/
@Entity
public class IdCard {
    private int id;
    private String cardNo;
    @Id
    @GeneratedValue
    public int getId() {return id;}
    public void setId(int id) { this.id = id;}
    public String getCardNo(){return cardNo;}
    public void setCardNo(StringcardNo) {this.cardNo = cardNo;}
}

```



而引用对象的实体类需要使用@ManyToOne进行注解，来表面是一对一的关系
再使用@JoinColumn注解来为数据库表中这个外键指定个字段名称就可以了。如果省略@JoinColumn注解，则hibernate会自动为其生成一个字段名(好像是：被引用对象名称_被引用对象的主键ID)

```
/** 人 - 实体类 */
@Entity
public class Person {
    private int id;
    private IdCard idCard;//引用IdCard对象
    private String name;
    @Id
    @GeneratedValue
    public int getId() {return id;}
    @ManyToOne//表示一对一的关系
    @JoinColumn(name="idCard")//为数据中的外键指定个名称
    public IdCard getIdCard(){ return idCard;}
    public String getName() {return name;}
    public void setId(int id) {this.id = id;}
    public void setIdCard(IdCardidCard) {this.idCard = idCard;}
    public void setName(Stringname) {this.name = name;}
}
```

快速回复

我要收藏

返回顶部

7、生成的SQL语句：

```
create tableIdCard (
    id integernot null auto_increment,
    cardNo varchar(255),
    primary key(id)
)
create tablePerson (
    id integernot null auto_increment,
    namevarchar(255),
    idCardinteger,//新添加的外键
    primary key(id)
)
alter tablePerson
    add indexFK8E488775BE010483 (idCard),
    addconstraint FK8E488775BE010483
    foreign key(idCard) //外键
    referencesIdCard (id)//引用IdCard的id字段
```

Field	Type	Null	Key	Default	Extra
id	int<11>	NO	PRI	NULL	auto_increment
cardNo	varchar<255>	YES		NULL	

2 rows in set (0.00 sec)

mysql> desc person
-> ;

Field	Type	Null	Key	Default	Extra
id	int<11>	NO	PRI	NULL	auto_increment
name	varchar<255>	YES		NULL	
idCard	int<11>	YES	MUL	NULL	

8、存储测试

```
Session session = sf.getCurrentSession();
IdCard idCard = new IdCard();
idCard.setCardNo("88888888888888888888888888888888");
session.beginTransaction();
```



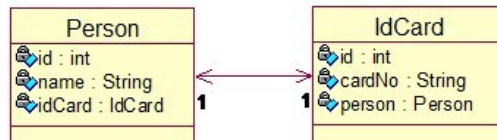
```
// 如果先不保存idCard, 则抛出Transient异常, 因为idCard不是持久化状态。
session.save(idCard);
Person person = new Person();
person.setName("菜10");
person.setIdCard(idCard);
session.save(person);
session.getTransaction().commit();
```

(二) 唯一-外键关联-双向

1、说明：

人<—> 身份证号(Person<->IdCard)双向：互相持有对方的引用

2、对象模型：



3、关系模型：

关系模型没有任务变化，同单向

4、实体类：

实体类，只是相互持有对象的引用，并且要求getter和setter方法

5、xml映射

Person实体类映射文件：同单向的没有变化

IdCard实体类映射文件：如果使用同样的方法映射，这样就会在表中也添加一个外键指向对象，但对象已经有一个外键指向自己了，这样就造成了冗余字段，因为不需要在表另外添加字段，而是让hibernate在加载这个对象时，会根据对象的ID到对方的表中查询外键等于这个ID的记录，这样就把对象加载上来了。也同样需要使用<one-to-one>标签来映射，但是需要使用property-ref属性来指定对象持有你自己的引用的成员属性名称(是gettxxxx后面的名称)，这样在生成数据库表时，就不会再添加一个多余的字段了。数据加载时hibernate会根据这些配置自己加载数据

```
<class name="com.wjt276.hibernate.IdCard" table="idcard">
    <id name="id" column="id">
        <generator class="native"/></id>
    <property name="cardNo"/>
    <!--<one-to-one>标签：告诉hibernate如何加载其关联对象
        property-ref属性：是根据哪个字段进行比较加载数据 -->
    <one-to-one name="person" property-ref="idCard"/>
</class>
```

一对一 唯一-外键 关联映射 双向 需要在另一端(当前IdCard)，添加<one-to-one>标签，指示hibernate如何加载其关联对象(或引用对象)，默认根据主键加载(加载person),外键关联映射中，因为两个实体采用的是person的外键来维护的关系，所以不能指定主键加载person,而要根据person的外键加载，所以采用如下映射方式：

```
<!--<one-to-one>标签：告诉hibernate如何加载其关联对象
        property-ref属性：是根据哪个字段进行比较加载数据 -->
    <one-to-one name="person" property-ref="idCard"/>
```

6、annotation注解映射

Person注解映射同单向一样

IdCard注解映射如下：使用@OneToOne注解来一对一，但这样会在表中多加一个字段，因为需要使用对象的外键来加载数据，所以使用属性mappedBy属性在实现这个功能

@Entity

```
public class IdCard {
    private int id;
    private String cardNo;
    private Person person;

    //mappedBy：在指定当前对象在被Person对象的idCard做了映射了
    //此值：当前对象持有引用对象中引用当前对象的成员属性名称(getXXX后的名称)
```



```
//因为Person对象的持有IdCard对象的方法是getIdCard()因为需要小写,所以为idCard
@OneToOne(mappedBy="idCard")
public Person getPerson(){return person;}
public void setPerson(Person person){this.person = person;}
@Id
@GeneratedValue
public int getId() {return id;}
public void setId(int id) { this.id = id;}
public String getCardNo(){return cardNo;}
public void setCardNo(StringcardNo) {this.cardNo = cardNo;}
}
```

7、生成SQL语句

因为关系模型没有变化，也就是数据库的结构没有变化，只是在数据加载时需要相互加载对方，这由hibernate来完成。因为生成的sql语句同单向一样。

8、存储测试

存储同单向一样

9、总结：

规律：凡是双向关联，必设mappedBy

快速回复

我要收藏

返回顶部

(三) 主键关联-单向(不重要)

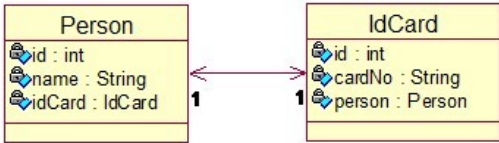
主键关联：即让两个对象具有相同的主键值，以表明它们之间的——对应的关系；数据库表不会有额外的字段来维护它们之间的关系，仅通过表的主键来关联。

1、说明：

人——> 身份证号(PersonàIdCard)，从IdCard看不到Person对象

2、对象模型

站在人的角度来看，对象模型与唯一外键关联一个，只是关系模型不同



3、关系模型

因为是person引用idcard，所以idcard要求先有值。而person的主键值不是自己生成的。而是参考idcard的值，person表中即是主键，同时也是外键

t_person			t_idcard	
id	name	idcard(唯一)	id	cardNo
5	菜 10	100	100	888888888888888888
6	容祖儿	200	200	999999999999999999

4、实体类：

实体类同 一对一 唯一外键关联的实体类一个，在person对象中持有idcard对象的引用(代码见唯一外键关系)

5、xml映射

IdCard映射文件，先生成ID

```
<class name="com.wjt276.hibernate.IdCard" table="t_idcard">
    <id name="id"column="id">
        <generator class="native"/>
    </id>
    <property name="cardNo"/>
</class>
```

Person实体类映射文件，ID是根据IdCard主键值

```
<class name="com.wjt276.hibernate.Person"table="t_person">
    <id name="id"column="id">
```

<!--因为主键不是自己生成的，而是作为一个外键(来源于其它值)，所以使用foreign生成策略
foreign:使用另外一个相关联的对象的标识符，通常和<one-to-one>联合起来使用。再使用元素<param>的属性
值指定相关联对象(这里Person相关联的对象为idCard,则标识符为idCard的id)为了能够在加载person数据同时
加载idCard数据，所以需要使用一个标签<one-to-one>来设置这个功能。 -->

```
<generator class="foreign">
    <!-- 元素<param>属性name的值是固定为property -->
    <param name="property">idCard</param>
</generator>
</id>
<property name="name"/>
<!-- <one-to-one>标签
```

表示如何加载它的引用对象(这里引用对象就指idCard这里的name值是idCard)，同时也说是一对一的关系。默认方式是根据主键加载(把person中的主键取出再到IdCard中来取相关IdCard数据。)我们也说过此主键也作为一个外键引用了IdCard，所以需要加一个数据库限制(外键约束)constrained="true" -->

```
<one-to-one name="idCard"constrained="true"/>
```

6、annotateton注解映射

Person实体类注解

方法：只需要使用@OneToOne注解一对一关系，再使用@PrimaryKeyJoinColumn来注解主键关联

@Entity

```
public class Person {
    private int id;
    private IdCard idCard;//引用IdCard对象
    private String name;
    @Id
    public int getId() {return id;}
    @OneToOne//表示一对一的关系
    @PrimaryKeyJoinColumn//注解主键关联映射
    public IdCard getIdCard(){ return idCard;}
    public String getName() {return name;}
    public void setId(int id) {this.id = id;}
    public void setIdCard(IdCard idCard){this.idCard = idCard;}
    public void setName(Stringname) {this.name = name;}
}
```

IdCard实体类，不需要持有对象的引用，正常注解就可以了。

7、生成SQL语句

生成的两个表并没有多余的字段，是因为通过主键在关键的

```
create tableIdCard (
    id integernot null auto_increment,
    cardNovarchar(255),
    primary key (id)
)
create tablePerson (
    id integernot null,
    namevarchar(255),
    primary key(id)
)
alter table person
add index FK785BED805248EF3 (id),
add constraint FK785BED805248EF3
foreign key (id) references idcard (id)
```

注意：annotation注解后，并没有映射出外键关键的关联，而xml可以映射，是主键关联不重要

8、存储测试

```
session = HibernateUtils.getSession();
```

快速回复

我要收藏

返回顶部

```

tx = session.beginTransaction();
IdCard idCard = new IdCard();
idCard.setCardNo("88888888888888888888888888888888");

Person person = new Person();
person.setName("菜10");
person.setIdCard(idCard);

//不会出现TransientObjectException异常
//因为一对一主键映射中，默认了cascade属性。
session.save(person);
tx.commit();

```

9、总结

让两个实体对象的ID保持相同，这样可以避免多余的字段被创建

```

<id name="id" column="id">
    <!--person的主键来源idcard，也就是共享idCard的主键-->
    <generator class="foreign">
        <param name="property">idCard</param>
    </generator>
</id>
<property name="name"/>
<!--one-to-one标签的含义：指示hibernate怎么加载它的关联对象，默认根据主键加载
constrained="true"，表面当前主键上存在一个约束：person的主键作为外键参照了idCard-->
<one-to-one name="idCard" constrained="true"/>

```

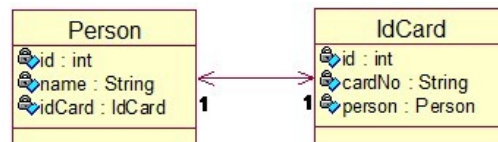
[快速回复](#)
[我要收藏](#)
[返回顶部](#)

(四) 主键关联-双向(不重要)

主键关联：即让两个对象具有相同的主键值，以表明它们之间的——对应的关系；数据库表不会有额外的字段来维护它们之间的关系，仅通过表的主键来关联。

主键关联映射，实际是数据库的存储结构并没有变化，只是要求双方都可以持有对象引用，也就是说实体模型变化，实体类都相互持有对方引用。

另外映射文件也变化了。



1、xml映射

Person实体类映射文件不变，

IdCard如下：

```

<class name="com.wjt276.hibernate.IdCard" table="t_idcard">
    <id name="id" column="id">
        <generator class="native"/> </id>
    <property name="cardNo"/>
<!--one-to-one标签的含义：指示hibernate怎么加载它的关联对象(这里的关联对象为person)，默认根据主键加载-->
    <one-to-one name="person"/>
</class>

```

2、annotation注解映射：

Person的注解不变，同主键单向注解

IdCard注解，只需要在持有对象引用的getXXX前加上

@OneToOne(mappedBy="idCard") 如下：

@Entity

```

public class IdCard {
    private int id;

```

```

private String cardNo;
private Person person;
@OneToOne(mappedBy="idCard")
public Person getPerson(){
    return person;
}
}

```

(五) 联合主键关联(Annotation方式)

实现上联合主键的原理同唯一外键关联-单向一样，只是使用的是@JoinColumns，而不是@JoinColumn，实体类注解如下：

```

@OneToOne
@JoinColumns(
{
    @JoinColumn(name="wifeId", referencedColumnName="id"),
    @JoinColumn(name="wifeName", referencedColumnName="name")
}
)
public WifegetWife() {
    return wife;
}
}

```

注意：@oinColumns注解联合主键一对一联系，然后再使用@JoinColumn来注解当前表中的外键字段名，并指定关联哪个字段，使用referencedColumnName指定哪个字段的名称

快速回复

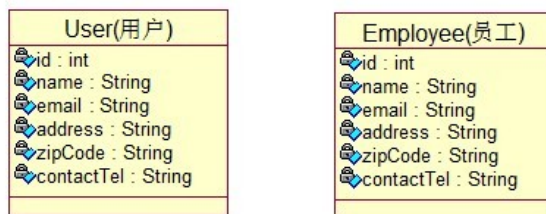
☆ 我要收藏

^ 返回顶部

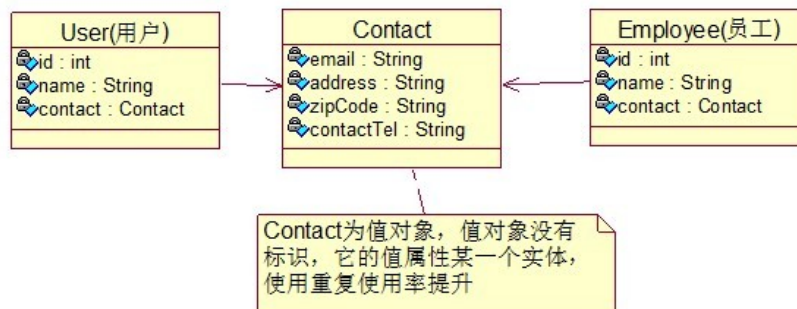
二、 component (组件) 关联映射

(一) Component关联映射：

目前有两个类如下：



大家发现用户与员工存在很多相同的字段，但是两者有不可以是同一个类中，这样在实体类中每次都要输入很多信息，现在把联系信息抽取出来成为一个类，然后在用户、员工对象中引用就可以，如下：



值对象没有标识，而实体对象具有标识，值对象属于某一个实体，使用它重复使用率提升，而且更清晰。

以上关系的映射称为component (组件) 关联映射

在hibernate中，component是某个实体的逻辑组成部分，它与实体的根本区别是没有oid.component可以成为是值对象 (DDD)。

采用component映射的好处:它实现了对象模型的细粒度划分，层次会更加分明，复用率会更高。

(二) User实体类：

```

public class User {
private int id;
private String name;
private Contact contact;//值对象的引用
public int getId() {return id;}
}

```

```

public void setId(int id) { this.id = id;}
public String getName() { return name;}
public void setName(Stringname) { this.name = name;}
public ContactgetContact() { return contact;}
public void setContact(Contactcontact) { this.contact = contact;}
}

```

(三) Contact值对象：

```

public class Contact {
    private String email;
    private String address;
    private String zipCode;
    private String contactTel;
    public String getEmail(){ return email;}
    public void setEmail(Stringemail) { this.email = email; }
    public StringgetAddress() {return address;}
    public void setAddress(Stringaddress) {this.address = address;}
    public StringgetZipCode() {return zipCode;}
    public void setZipCode(StringzipCode) {this.zipCode = zipCode;}
    public StringgetContactTel() { return contactTel;}
    public voidsetContactTel(String contactTel){this.contactTel = contactTel;}
}

```

[快速回复](#)
[我要收藏](#)
[返回顶部](#)

(四) xml--User映射文件(组件映射)：

```

<hibernate-mapping>
  <class name="com.wjt276.hibernate.User" table="t_user">
    <id name="id" column="id">
      <generator class="native"/>
    </id>
    <property name="name" column="name">
      <!-- <component>标签用于映射Component(组件)关系
      其内部属性正常映射。
      -->
      <component name="contact">
        <property name="email"/>
        <property name="address"/>
        <property name="zipCode"/>
        <property name="contactTel"/>
      </component>
    </property>
  </class>
</hibernate-mapping>

```

(五) annotateon注解

使用@Embedded用于注解组件映射，表示嵌入对象的映射

@Entity

```

public class User {
    private int id;
    private String name;
    private Contact contact;//值对象的引用
    @Id
    @GeneratedValue
    public int getId() { return id;}

    @Embedded//用于注解组件映射，表示嵌入对象的映射
    public ContactgetContact() {return contact;}
    public void setContact(Contactcontact) {this.contact = contact;}
}

```

Contact类是值对象，不是实体对象，是属于实体类的某一部分，因此没有映射文件

(六) 导出数据库输出SQL语句：

```
create table User (
    id integer not null auto_increment,
    address varchar(255),
    contactTel varchar(255),
    email varchar(255),
    zipCode varchar(255),
    name varchar(255),
    primary key (id)
)
```

(七) 数据表结构：

```
mysql> show tables;
+-----+
| Tables_in_hibernate_component_mapping |
+-----+
| t_user                                |
+-----+
1 row in set (0.00 sec)

mysql> desc t_user;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id    | int(11) | NO | PRI | NULL | auto_increment |
| name  | varchar(255) | YES | | NULL | |
| email | varchar(255) | YES | | NULL | |
| address | varchar(255) | YES | | NULL | |
| zipCode | varchar(255) | YES | | NULL | |
| contactTel | varchar(255) | YES | | NULL | |
+-----+-----+-----+-----+-----+-----+
6 rows in set (0.01 sec)
```

注：虽然实体类没有基本联系信息，只是有一个引用，但在映射数据库时全部都映射进来了。以后值对象可以重复使用，只要在相应的实体类中加入一个引用即可。

(八) 组件映射数据保存：

```
session =HibernateUtils.getSession();
tx =session.beginTransaction();
```

```
User user= new User();
user.setName("10");
```

```
Contactcontact = new Contact();
contact.setEmail("wjt276");
contact.setAddress("aksdfj");
contact.setZipCode("230051");
contact.setContactTel("3464661");
```

```
user.setContact(contact);
session.save(user);
```

```
tx.commit();
```

实体类中引用值对象时，不用先保存值对象，因为它不是实体类，它只是一个附属类，而session.save()中保存的对象是实体类。

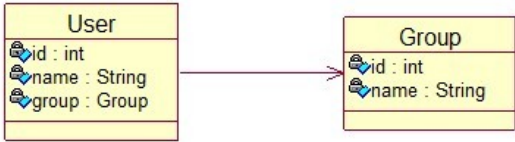
三、 多对一 -单向

场景：用户和组；从用户角度来，多个用户属于一个组(多对一 关联)

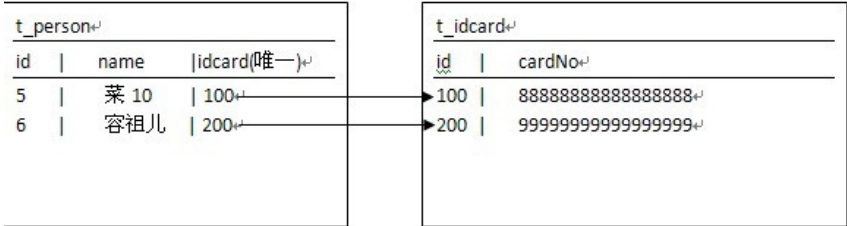
使用hibernate开发的思路：先建立对象模型(领域模型)，把实体抽取出来。

目前两个实体：用户和组两个实体，多个用户属于一个组，那么一个用户都会对应于一个组，所以用户实体中应该有一个持有组的引用。

(一) 对象模型图：



(二) 关系模型：



(三) 关联映射的本质：

将关联关系映射到数据库，所谓的关联关系是对象模型在内存中一个或多个引用。

(四) 实体类

User实体类：

```
public class User {
    private int id;
    private String name;
    private Group group;
    public Group getGroup() {return group; }
    public void setGroup(Group group) {this.group = group;}
    public int getId() {return id; }
    public void setId(int id) { this.id = id;}
    public String getName() {return name;}
    public void setName(Stringname) { this.name = name;}}
```

Group实体类：

```
public class Group {
    private int id;
    private String name;
    public int getId() {return id;}
    public void setId(int id) { this.id = id;}
    public String getName() {return name;}
    public void setName(Stringname) {this.name = name;}
}
```

实体类建立完后，开始创建映射文件，先建立简单的映射文件：

(五) xml方式：映射文件：

1、 Group实体类的映射文件：

```
<hibernate-mapping>
  <class name="com.wjt276.hibernate.Group" table="t_group">
    <id name="id" column="id">
      <generator class="native"/>
    </id>
    <property name="name"/>
  </class>
</hibernate-mapping>
```

2、 User实体类的映射文件：

```
<hibernate-mapping>
  <class name="com.wjt276.hibernate.User" table="t_user">
    <id name="id" column="id">
      <generator class="native"/>
    </id>
```

快速回复

我要收藏

返回顶部


```
<property name="name"/>
```

```
<!--<many-to-one> 关联映射 多对一的关系
```

name:是维护的属性(User.group),这样表示在多的的一端表里加入一个字段名称为group,但group与SQL中的关键字重复,所以需要重新命名字段(column="groupid").这样这个字段(groupid)会作为外键参照数据库中group表(t_group也叫一的一端),也就是就在多的一端加入一个外键指向一的一端。 -->

```
<many-to-one name="group" column="groupid"/>
```

```
</class>
```

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

3、※<many-to-one>标签※：

例如：<many-to-one name="group" column="groupid"/>

<many-to-one> 关联映射 多对一的关系

name:是维护的属性(User.group),这样表示在多的的一端表里加入一个字段名称为group,但group与SQL中的关键字重复,所以需要重新命名字段(column="groupid").这样这个字段(groupid)会作为外键参照数据库中group表(t_group也叫一的一端),也就是就在多的一端加入一个外键指向一的一端。

这样导出至数据库会生成下列语句：

```
alter table t_user drop foreign keyFKCB63CCB695B3B5AC
```

```
drop table if exists t_group
```

```
drop table if exists t_user
```

```
create table t_group (id integer not nullauto_increment, name varchar(255), primary key (id))
```

```
create table t_user (id integer not nullauto_increment, name varchar(255), groupid integer,primary key (id))
```

```
alter table t_user add index FKCB63CCB695B3B5AC (groupid), add constraint FKCB63CCB695B3B5AC
```

```
foreign key (groupid) referencet_group (id)
```

[快速回复](#)
[我要收藏](#)
[返回顶部](#)

(六) annotation

Group(一的一端)注解只需要正常的注解就可以了，因为在实体类中它是被引用的。

User*(多的一端)：@ManyToOne来注解多一对的关键，并且用@JoinColumn来指定外键的字段名

@Entity

```
public class User {
    private int id;
    private String name;
    private Group group;
```

```
@ManyToOne
@JoinColumn(name="groupId")
public Group getGroup() {
    return group;
}
```

(七) 多对一 存储(先存储group(对象持久化状态后，再保存user))：

```
session = HibernateUtils.getSession();
```

```
tx =session.beginTransaction();
```

```
Groupgroup = new Group();
group.setName("wj1276");
session.save(group); //存储Group对象。
```

```
Useruser1 = new User();
user1.setName("菜10");
user1.setGroup(group);//设置用户所属的组
```

```
Useruser2 = new User();
user2.setName("容祖儿");
user2.setGroup(group);//设置用户所属的组
```

```
//开始存储
session.save(user1);//存储用户
session.save(user2);
```

```
tx.commit();//提交事务
```

执行后hibernate执行以下SQL语句：

Hibernate: insert into t_group (name) values (?)

Hibernate: insert into t_user (name, groupid) values (?, ?)

Hibernate: insert into t_user (name, groupid) values (?, ?)

注意：如果上面的session.save(group)不执行，则存储不存储不成功。则抛出TransientObjectException异常。因为Group为Transient状态，Object的id没有分配值。2610644

结果：persistent状态的对象是不能引用Transient状态的对象

以上代码操作，必须首先保存group对象，再保存user对象。我们可以利用cascade(级联)方式，不需要先保存group对象。而是直接保存user对象，这样就可以在存储user之前先把group存储了。

利用cascade属性是解决TransientObjectException异常的一种手段。

(八) 重要属性-cascade(级联)：

级联的意思是指定两个对象之间的操作联运关系，对一个对象执行了操作之后，对其指定的级联对象

执行相同的操作，取值：all、none、save_update、delete

- 1、 all：代码在所有的情况下都执行级联操作
- 2、 none:在所有情况下都不执行级联操作
- 3、 save-update：在保存和更新的时候执行级联操作
- 4、 delete：在删除的时候执行级联操作。

1、 xml

例如：<many-to-one name="group"column="groupid" cascade="save-update"/>

2、 annotation

cascade属性：其值： CascadeType.ALL 所有

CascadeType.MERGE save+ update

CascadeType.PERSIST

CascadeType.REFRESH

CascadeType.REMOVE

例如：

@ManyToMany(cascade={CascadeType.ALL})

注意：cascade只是帮我们省了编程的麻烦而已，不要把它的作用看的太大

(九) 多对一 加载数据

代码如下：

```
session = HibernateUtils.getSession();
tx =session.beginTransaction();

User user = (User)session.load(User.class, 3);
System.out.println("user.name=" + user.getName());
System.out.println("user.group.name=" + user.getGroup().getName());

//提交事务
tx.commit();
```

执行后向SQL发出以下语句：

Hibernate: select user0_.id as id0_0_,user0_.name as name0_0_, user0_.groupid as groupid0_0_ from t_user user0_ where user0_.id=?

Hibernate: select group0_.id as id1_0_,group0_.name as name1_0_ from t_group group0_ where group0_.id=?



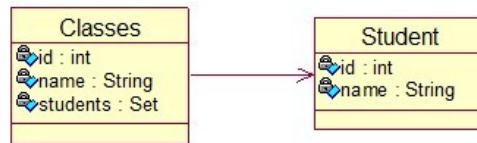
可以加载Group信息：因为采用了<many-to-one>这个标签，这个标签会在多的一端(User)加一个外键，指向一的一端(Group)，也就是它维护了从多到一的这种关系，多指向一的关系。当你加载多一端的数据时，它就能把一的这一端数据加载上来。当加载User对象后hibernate会根据User对象中的groupid再来加载Group信息给用户对象中的group属性。

四、 一对多 - 单向

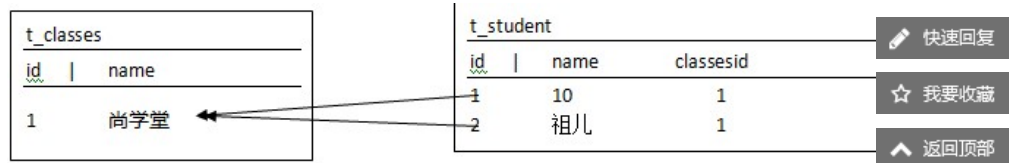
在对象模型中，一对多的关联关系，使用集合来表示。

实例场景：班级对学生；Classes(班级)和Student(学生)之间是一对多的关系。

(一) 对象模型：



(二) 关系模型：



一对多关联映射利用了对一关联映射原理。

(三) 多对一、一对多的区别：

多对一关联映射：在多的的一端加入一个外键指向一的一端，它维护的关系是多指向一的。

一对多关联映射：在多的的一端加入一个外键指向一的一端，它维护的关系是一指向多的。

两者使用的策略是一样的，只是各自所站的角度不同。

(四) 实体类

Classes实体类：

```

public class Classes {
    private int id;
    private String name;
    //一对多通常使用Set来映射，Set是不可重复内容。
    //注意使用Set这个接口，不要使用HashSet,因为hibernate有延迟加载，
    private Set<Student>students = new HashSet<Student>();
    public int getId() {return id; }
    public void setId(int id) {this.id = id;}
    public String getName() {return name;}
    public void setName(Stringname) {this.name = name;}
}
  
```

Student实体类：

```

public class Student {
    private int id;
    private String name;
    public int getId() {return id;}
    public void setId(int id) { this.id = id;}
    public String getName() {return name;}
    public void setName(Stringname) { this.name = name;}
}
  
```

(五) xml方式：映射

1、 Student映射文件：

```

<hibernate-mapping>
    <class name="com.wjt276.hibernate.Student" table="t_student">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
    </class>
</hibernate-mapping>
  
```

```
<property name="name" column="name"/>
</class>
```

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

2、Classes映射文件：

```
<hibernate-mapping>
```

```
<class name="com.wjt276.hibernate.Classes" table="t_classes">
```

```
<id name="id" column="id">
```

```
<generator class="native"/>
```

```
</id>
```

```
<property name="name" column="name"/>
```

<!--<set>标签 映射一对多(映射set集合),name="属性集合名称",然后在用<key>标签,在多的的一端加入一个外键(column属性指定列名称)指向一的一端,再采用<one-to-many>标签说明一对多,还指定<set>标签中name="students"这个集合中的类型要使用完整的类路径(例如: class="com.wjt276.hibernate.Student") -->

```
<set name="students">
```

```
<key column="classesid"/>
```

```
<one-to-many class="com.wjt276.hibernate.Student"/>
```

```
</set>
```

```
</class>
```

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

(六) annotation注解

一对多 多的一端只需要正常注解就可以了。

需要在一的一端进行注解一对多的关系。

使用@OneToMany

@Entity

```
public class Classes {
```

```
    private int id;
```

```
    private String name;
```

```
// 一对多通常使用Set来映射, Set是不可重复内容。
```

```
// 注意使用Set这个接口, 不要使用HashSet,因为hibernate有延迟加载,
```

```
private Set<Student>students = new HashSet<Student>();
```

```
@OneToMany//进行注解为一对多的关系
```

```
@JoinColumn(name="classesId")//在多的的一端注解一个字段(名为classesid)
```

```
public Set<Student>getStudents() {
```

```
    return students;
```

```
}
```

(七) 导出至数据库(hbm2ddl)生成的SQL语句：

```
create table t_classes (id integer not null auto_increment, namevarchar(255), primary key (id))
```

```
create table t_student (id integer not null auto_increment, namevarchar(255), classesid integer, primary key (id))
```

```
alter table t_student add index FK4B90757070CFE27A (classesid), add constraint FK4B90757070CFE27A foreign key (classesid) reference t_classes (id)
```

(八) 一对多 单向存储实例：

```
session = HibernateUtils.getSession();
```

```
tx =session.beginTransaction();
```

```
Studentstudent1 = new Student();
```

```
student1.setName("10");
```


```
session.save(student1);//必需先存储, 否则在保存classes时出错.
```

```
Studentstudent2 = new Student();
```

```
student2.setName("祖儿");
```

```
session.save(student2);//必需先存储, 否则在保存classes时出错.
```

 快速回复

 我要收藏

 返回顶部

```

Set<Student>students = new HashSet<Student>();
students.add(student1);
students.add(student2);

Classesclasses = new Classes();
classes.setName("wj276");
classes.setStudents(students);

session.save(classes);
tx.commit();

```

(九) 生成的SQL语句：


Hibernate: insert into t_student (name) values (?)


Hibernate: insert into t_student (name) values (?)

Hibernate: insert into t_classes (name) values (?)

Hibernate: update t_student set classesid=? where id=?

Hibernate: update t_student set classesid=? where id=?

 快速回复

 我要收藏

 返回顶部

(十) 一对多，在一的一端维护关系的缺点：

因为是在一的一端维护关系，这样会发出多余的更新语句，这样在批量数据时，效率不高。

还有一个，当在多的的一端的那个外键设置为非空时，则在添加多的一端数据时会发生错误，数据存储不成功。

(十一) 一对多 单向数据加载：

```

session = HibernateUtils.getSession();
tx =session.beginTransaction();
Classesclasses = (Classes)session.load(Classes.class, 2);
System.out.println("classes.name=" + classes.getName());
Set<Student> students = classes.getStudents();
for(Iterator<Student> iter = students.iterator();iter.hasNext();){
    Studentstudent = iter.next();
    System.out.println(student.getName());
}
tx.commit();

```

(十二) 加载生成SQL语句：

Hibernate: select classes0_.id as id0_0_, classes0_.name as name0_0_ fromt_classes classes0_ where classes0_.id=?

Hibernate: select students0_.classesid as classesid1_, students0_.id asid1_, students0_.id as id1_0_, students0_.name as name1_0_ from t_studentstudents0_ where students0_.classesid=?

五、 一对多 - 双向

是加载学生时，能够把班级加载上来。当然加载班级也可以把学生加载上来

- 1、 在学生对象模型中，要持有班级的引用,并修改学生映射文件就可以了。。
- 2、 存储没有变化
- 3、 关系模型也没有变化

(一) xml方式：映射

学生映射文件修改后的：

```

<class name="com.wjt276.hibernate.Student" table="t_student">
    <id name="id" column="id">
        <generator class="native"/>
    </id>
    <property name="name" column="name"/>
<!--使用多对一标签映射 一对多双向，下列的column值必需与多的一端的key字段值一样。-->
    <many-to-one name="classes" column="classesid"/>
</class>

```

如果在一对多的映射关系中采用一的一端来维护关系的话会存在以下两个缺点：①如果多的一端那个外键设置为非空时，则多的一端就存不进数据；②会发出多于的Update语句，这样会影响效率。所以常用对于一对多的映射关系我们在多的一端维护关系，并让多的一端维护关系失效(见下面属性)。

代码：

```
<class name="com.wjt276.hibernate.Classes" table="t_classes">
    <id name="id"column="id">
        <generator class="native"/>
    </id>
    <property name="name"column="name"/>

    <!--
        <set>标签 映射一对多(映射set集合),name="属性集合名称"
        然后在用<key>标签，在多的一端加入一个外键(column属性指定列名称)指向一的一端
        再采用<one-to-many>标签说明一对多，还指定<set>标签中name="students"这个集合中的类型
        要使用完整的类路径(例如：class="com.wjt276.hibernate.Student")
inverse="false":一的一端维护关系失效(反转)：false：可以从一的一端维护关系(默认)；true：从多的一端维护关系失效，这样如果在一的一端维护关系则不会发出Update语句。 -->
        <set name="students"inverse="true">
            <key column="classesid"/>
            <one-to-many class="com.wjt276.hibernate.Student"/>
        </set>
    </class>
```

快速回复

我要收藏

返回顶部

(二) annotateon方式注解

首先需要在多的一端持有一一端的引用

因为一对多的双向关联，就是多对一的关系，我们一般在多的一端来维护关系，这样我们需要在多的一端实体类进行映射多对一，并且设置一个字段，而在一的一端只需要进行映射一对多的关系就可以了，如下：

多的一端

@Entity

```
public class Student {
    private int id;
    private String name;
    private Classes classes;

    @ManyToOne
    @JoinColumn(name="classesId")
    public ClassesgetClasses() {
        return classes;
    }
}
```

一的一端

@Entity

```
public class Classes {
    private int id;
    private String name;

    // 一对多通常使用Set来映射，Set是不可重复内容。
    // 注意使用Set这个接口，不要使用HashSet,因为hibernate有延迟加载，
    private Set<Student>students = new HashSet<Student>();

    @OneToMany(mappedBy="classes")//进行注解为一对多的关系
    public Set<Student>getStudents() {
        return students;}
}
```

(三) 数据保存：

一对多 数据保存，从多的一端进行保存：

```
session = HibernateUtils.getSession();
tx =session.beginTransaction();

Classesclasses = new Classes();
classes.setName("wjt168");
session.save(classes);

Studentstudent1 = new Student();
student1.setName("10");
student1.setClasses(classes);
session.save(student1);

Studentstudent2 = new Student();
student2.setName("祖儿");
student2.setClasses(classes);
session.save(student2);

//提交事务
tx.commit();
```

注意：一对多，从多的一端保存数据比从一的一端保存数据要快，因为从一的一端保存数据时，会多更新多的一端的一个外键(是指定一的一端的。)

(四) 关于inverse属性：

inverse主要用在一对多和多对多双向关联上，inverse可以被设置到集合标签<set>上，默认inverse为false，所以我们可以从一的一端和多的一端维护关联关系，如果设置inverse为true，则我们只能从多的一端维护关联关系。

注意：inverse属性，只影响数据的存储，也就是持久化

(五) Inverse和cascade区别：

Inverse是关联关系的控制方向

Cascade操作上的连锁反应

(六) 一对多双向关联映射总结：

在一的一端的集合上使用<key>，在对方表中加入一个外键指向一的一端

在多的的一端采用<many-to-one>

注意：<key>标签指定的外键字段必须和<many-to-one>指定的外键字段一致，否则引用字段的错误

如果在一的一端维护一对多的关系，hibernate会发出多余的update语句，所以我们一般在多的一端来维护关系。

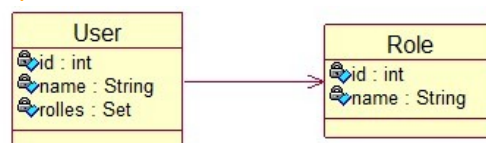
六、 多对多 - 单向

- Ø 一般的设计中，多对多关联映射，需要一个中间表
- Ø Hibernate会自动生成中间表
- Ø Hibernate使用many-to-many标签来表示多对多的关联
- Ø 多对多的关联映射，在实体类中，跟一对多一样，也是用集合来表示的。

(一) 实例场景：

用户与他的角色(一个用户拥有多个角色，一个角色还可以属于多个用户)

(二) 对象模型：



(三) 关系模型：

t_user		t_user_role(第三方表, 表示 user 和 role 关系)		t_role	
id	name	id	name	id	name
1	10	1	1	1	数据录入人员
2	祖儿	1	2	2	商务主管
3	成龙	2	2	3	大区经理
		2	3		
		3	1		
		3	2		
		3	3		

(四) 实体类

Role实体类：

```
public class Role {
    private int id;
    private String name;
    public int getId() {return id;}
    public void setId(int id) {this.id = id;}
    public String getName() {return name;}
    public void setName(String name) { this.name = name;}}
```

User实体类：

```
public class User {
    private int id;
    private String name;
    private Set<User> roles = new HashSet<User>();// Role对象的集合
    public int getId() {return id;}
    public void setId(int id) {this.id = id;}
    public String getName() {return name;}
    public void setName(String name) {this.name = name;}
    public Set<User> getRoles() {return roles;}
    public void setRoles(Set<User> roles) {this.roles = roles; }
```

(五) xml方式：映射

Role映射文件：

```
<class name="com.wjt276.hibernate.Role" table="t_role">
    <id name="id">
        <generator class="native"/>
    </id>
    <property name="name" column="name"/>
</class>
```

User映射文件：

```
<class name="com.wjt276.hibernate.User" table="t_user">
    <id name="id" column="id">
        <generator class="native"/>
    </id>
    <property name="name"/>
```

<!--使用<set>标签映射集合(set)，标签中的name值为对象属性名(集合roles)，而使用table属性是用于生成第三方表名称，例：table="t_user_role"，但是第三方面中的字段是自动加入的，作为外键分别指向其它表。

所以表<key>标签设置，例：<key column="userid"/>，意思是：在第三方表(t_user_role)中加入一个外键并且指向当前的映射实体类所对应的表(t_user)。使用<many-to-many>来指定此映射集合所对象的类(实例类)，并且使用column属性加入一个外键指向Role实体类所对应的表(t_role) -->

```
<set name="roles" table="t_user_role">
    <key column="userid"/>
    <many-to-many class="com.wjt276.hibernate.Role" column="roleid"/>
</set>
</class>
```

(六) annotation注解方式

快速回复

我要收藏

返回顶部

注意：因为是多对多单向(当然用户拥有多个角色，一个角色也可属性多个用户，但这里角色看不到用户)，所以角色只需要正常注解就可以了，

现在要使用@ManyToMany来注解多对多的关系，并使用@JoinTable来注解第三方表的名称，再使用joinColumns属性来指定当前对象在第三方表中的字段名，并且这个字段会指向当前类相对应的表，最后再用inverseJoinColumns来指定当前类持有引用的实体类在第三方表中的字段名，并且指向被引用对象相对应的表，如下：

@Entity

```
public class User {
    private int id;
    private String name;
    private Set<User> roles = new HashSet<User>();// Role对象的集合 @Id
    @GeneratedValue
    public int getId() {return id;}
```

@ManyToMany

@JoinTable(name="u_r",//使用@JoinTable标签的name属性注解第三方表名称

joinColumns={@JoinColumn(name="userId")},

//使用joinColumns属性来注解当前实体类在第三方表中的字段名称并指向该对象

inverseJoinColumns={@JoinColumn(name="roleId")}

//使用inverseJoinColumns属性来注解当前实体类持有引用对象在第三方表中的字段名称并指向被引用对象表

)

```
public Set<User> getRoles() {return roles;}
```

(七) 生成SQL语句

```
create table t_role (id integer not null auto_increment, name varchar(255), primary key (id))
```

```
create table t_user (id integer not null auto_increment, name varchar(255), primary key (id))
```


```
create table t_user_role (userid integer not null, roleid integer not null, primary key (userid, roleid))
```


```
alter table t_user_role add index FK331DEE5F1FB4B2D4 (roleid), add constraint FK331DEE5F1FB4B2D4
foreign key (roleid) reference t_role (id)
```

```
alter table t_user_role add index FK331DEE5F250A083E(userid), add constraint FK331DEE5F250A083E
foreign key (userid) reference t_user (id)
```

注：根据DDL语句可以看出第三方表的主键是一个复合主键(primary key (userid, roleid))，也就是说记录不可以有相同的数据。

(八) 数据库表及结构：

 快速回复

 我要收藏

 返回顶部

```
mysql> show tables;
+-----+
| Tables_in_hibernate_session |
+-----+
| t_role                       |
| t_user                       |
| t_user_role                  |
| user                         |
+-----+
4 rows in set (0.01 sec)

mysql> desc t_role;
+-----+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id    | int(11)   | NO   | PRI | NULL    | auto_increment |
| name  | varchar(255) | YES  |     | NULL    |               |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> desc t_user;
+-----+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id    | int(11)   | NO   | PRI | NULL    | auto_increment |
| name  | varchar(255) | YES  |     | NULL    |               |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> desc t_user_role;
+-----+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| userid | int(11)   | NO   | PRI |         |         |
| roleid | int(11)   | NO   | PRI |         |         |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

(九) 多对多关联映射 单向数据存储：

```
session = HibernateUtils.getSession();
    tx =session.beginTransaction();

    Role r1 = new Role();
    r1.setName("数据录入人员");
    session.save(r1);

    Role r2 = new Role();
    r2.setName("商务主管");
    session.save(r2);

    Role r3 = new Role();
    r3.setName("大区经理");
    session.save(r3);

    User u1 = new User();
    u1.setName("10");
    Set<Role>u1Roles = new HashSet<Role>();
    u1Roles.add(r1);
    u1Roles.add(r2);
    u1.setRoles(u1Roles);

    User u2 = new User();
    u2.setName("祖儿");
    Set<Role>u2Roles = new HashSet<Role>();
    u2Roles.add(r2);
    u2Roles.add(r3);
    u2.setRoles(u2Roles);

    User u3 = new User();
    u3.setName("成龙");
```

快速回复

☆ 我要收藏

^ 返回顶部

```

Set<Role>u3Roles = new HashSet<Role>();
u3Roles.add(r1);
u3Roles.add(r2);
u3Roles.add(r3);
u3.setRoles(u3Roles);

```

```

session.save(u1);
session.save(u2);
session.save(u3);

```

```
tx.commit();
```

发出SQL语句：

Hibernate: insert into t_role (name) values (?)

Hibernate: insert into t_role (name) values (?)

Hibernate: insert into t_role (name) values (?)

Hibernate: insert into t_user (name) values (?)

Hibernate: insert into t_user (name) values (?)

Hibernate: insert into t_user (name) values (?)

Hibernate: insert into t_user_role (userid, roleid)values (?, ?)

Hibernate: insert into t_user_role (userid, roleid)values (?, ?)

Hibernate: insert into t_user_role (userid, roleid)values (?, ?)

Hibernate: insert into t_user_role (userid, roleid)values (?, ?)


Hibernate: insert into t_user_role (userid, roleid)values (?, ?)

Hibernate: insert into t_user_role (userid, roleid)values (?, ?)

Hibernate: insert into t_user_role (userid, roleid)values (?, ?)

注：前三条SQL语句，添加Role记录，第三条到第六条添加User，最后7条SQL语句是在向第三方表(t_user_role)中添加多对多关系(User与Role关系)

 快速回复

 我要收藏

 返回顶部

(十) 多对多关联映射 单向数据加载：

```

session =HibernateUtils.getSession();
tx =session.beginTransaction();

```

```

User user =(User)session.load(User.class, 1);
System.out.println("user.name=" + user.getName());
for(Iterator<Role> iter = user.getRoles().iterator();iter.hasNext();){
    Role role = (Role) iter.next();
    System.out.println(role.getName());
}
tx.commit();

```

生成SQL语句：

Hibernate: select user0_.id as id0_0_, user0_.name as name0_0_ from t_user user0_ where user0_.id=?
user.name=10

Hibernate: select roles0_.userid as userid1_, roles0_.roleid as roleid1_,role1_.id as id2_0_, role1_.name as name2_0_ from t_user_role roles0_ leftouter join t_role role1_ on roles0_.roleid=role1_.id where roles0_.userid=?

商务主管

数据录入人员

七、 多对多 - 双向

多对多关联映射 双向 两方都持有对象引用，修改对象模型,但数据的存储没有变化。

(一) xml方式：映射

再修改映射文件：

```

<class name="com.wjt276.hibernate.Role" table="t_role">
    <id name="id">
        <generator class="native"/>

```

```

</id>
<property name="name" column="name"/>
<!--order-by 属性是第三方表哪个字段进行排序-->
<set name="users" table="t_user_role"order-by="userid">
    <key column="roleid"/>
    <many-to-many class="com.wjt276.hibernate.User" column="userid"/>
</set> </class>

```

注：数据的存储与单向一样。但一般维护这个多对多关系，只需要使用一方，而使另一方维护关系失效。

总结：

<!-- order-by 属性是第三方表哪个字段进行排序-->

```


<set name="users" table="t_user_role"order-by="userid">
    <key column="roleid"/>
    <many-to-many class="com.wjt276.hibernate.User" column="userid"/>
</set>


```

Ø **table属性值必须和单向关联中的table属性值一致**

Ø **<key>中column属性值要与单向关联中的<many-to-many>标签中的column属性值一致**

Ø **在<many-to-many>中的column属性值要与单向关联中<key>标签的column属性值一致。**

 快速回复

 我要收藏

 返回顶部

(二) annotation注解方式

多对多关联映射 双向 两方都持有对象引用，修改对象模型,但数据的存储没有变化

只需要修改注解映射就可以了。

User实体类注解没有变化和单向一样：

@Entity

public class User {

private int id;

private Set<User> roles = new HashSet<User>();// Role对象的集合 @Id

@GeneratedValue

public int getId() {return id;}

@ManyToMany

@JoinTable(name="u_r",//使用@JoinTable标签的name属性注解第三方表名称

joinColumns=@JoinColumn(name="userId"),//使用joinColumns属性来注解当前实体类在第三方表中的
字段名称并指向该对象

inverseJoinColumns=@JoinColumn(name="roleId"))

//使用inverseJoinColumns属性来注解当前实体类持有引用对象在第三方表中的字段名称并指向被引用对象表
)

public Set<User> getRoles() {return roles;}

Role实体类注解也非常的简单:使用@ManyToMany注解，并使用mappedBy属性指定引用对象持有自己的属性名

@Entity

public class Role {

private int id;

private String name;

private Set<User> users = new HashSet<User>();

@Id

@GeneratedValue

public int getId() {return id; }

@ManyToMany(mappedBy="roles")

public Set<User> getUsers() {return users;}

public void setUsers(Set<User> users) {this.users = users; }

多对多关联映射 双向 数据加载

session =HibernateUtils.getSession();

tx =session.beginTransaction();

```

Role role= (Role)session.load(Role.class, 1);
System.out.println("role.name=" + role.getName());
for(Iterator<User> iter = role.getUsers().iterator();iter.hasNext();){
    User user = iter.next();
    System.out.println("user.name=" + user.getName());
}
tx.commit();

```

生成SQL语句：

Hibernate: select role0_.id as id2_0_, role0_.name as name2_0_ from t_role role0_ where role0_.id=?

role.name=数据录入人员

Hibernate: select users0_.roleid as roleid1_, users0_.userid as userid1_, user1_.id as id0_0_, user1_.name as name0_0_ from t_user_role users0_ leftouter join t_user user1_ on users0_.userid=user1_.id where users0_.roleid=? order by users0_.userid

user.name=10

user.name=成龙

八、 关联关系中的CRUD Cascade Fetch

- 1、 设定cascade可以设定在持久化时对于关联对象的操作(CUD, R归Fetch管)
- 2、 cascade仅仅是帮助我们省了编程的麻烦而已，不要把它的作用看的太大
 - a) cascade的属性指明做什么操作的时候关系对象是绑在一起的
 - b) refresh=A里面需要读B改过之后的数据
- 3、 铁律：双向关系在程序中要设定双向关联
- 4、 铁律：双向一定需要设置mappedBy
- 5、 fetch
 - a) 铁律：双向不要两边设置Eager(会有多余的查询语句发出)
 - b) 对多方设置fetch的时候要谨慎，结合具体应用，一般用Lazy不用eager，特殊情况(多方数量不多的可以考虑，提高效率的时候可以考)
- 6、 O/R Mapping编程模型
 - a) 映射模型
 - i. jpa annotation
 - ii. hibernate annotation extension
 - b) 编程接口
 - i. jpa
 - ii. hibernate
 - c) 数据查询语言
 - i. HQL
 - ii. EJBQL(JPQL)
- 7、 要想删除或者更新，先做load，除了精确知道ID之外
- 8、 如果想消除关联关系，先设定关系为null，再删除对应记录，如果不删记录，该记录就变成垃圾数据

九、 集合映射

1、 Set

2、 List

- a) @OrderBy

注意: List与Set注解是一样的，就是把Set更改为List就可以了

```

private List<User> users = new ArrayList<User>();
@OneToMany(mappedBy="group",
    cascade={CascadeType.ALL}
)

```

@OrderBy("name ASC")//使用@OrderBy注解List中使用哪个字段进行排序，可以组合排序，中间使用逗号分开

```

public List<User> getUsers() { return users;}
public void setUsers(List<User> users) {this.users = users;}

```

3、 Map

- a) @Mapkey



注解：关联模型中并没有变化，只是注解发生了变化，而这个变化是给hibernate看的

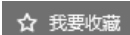
Map在映射时，因为Key是唯一的，所以一般可以使用主键表示，然后在映射时使用@MapKey来注解使用哪个字段为key,如下：

```
private Map<Integer,User> users = new HashMap<Integer, User>();
@OneToMany(mappedBy="group", cascade=CascadeType.ALL)
@MapKey(name="id")//注解使用哪个字段为key
public Map<Integer,User> getUsers() { return users;}
public voidsetUsers(Map<Integer, User> users) {this.users = users;}
```

数据加载：

```
Session s = sessionFactory.getCurrentSession();
s.beginTransaction();
Group g =(Group)s.load(Group.class, 1);
for(Map.Entry<Integer,User> entry : g.getUsers().entrySet()) {
    System.out.println(entry.getValue().getName());
}
s.getTransaction().commit();
```

 快速回复

 我要收藏

 返回顶部

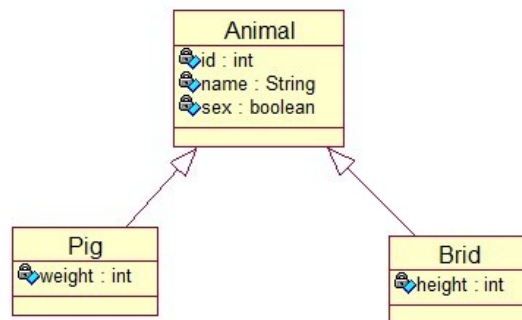
十、 继承关联映射

继承映射：就是把类的继承关系映射到数据库里(首先正确的存储，再正确的加载数据)

(一) 继承关联映射的分类：

- Ø 单表继承：每棵类继承树使用一个表(table per class hierarchy)
- Ø 具体表继承：每个子类一个表(table per subclass)
- Ø 类表继承：每个具体类一个表(table per concrete class)(有一些限制)
- Ø **实例环境：**动物Animal有三个基本属性，然后有一个Pig继承了它并扩展了一个属性，还有一个Brid也继承了并且扩展了一个属性

(二) 对象模型：



(三) 单表继承SINGLE_TABLE：

每棵类继承树使用一个表

把所有的属性都要存储表中，目前至少需要5个字段，另外需要加入一个标识字段(表示哪个具体的子类)

t_animal

Id	Name	Sex	Weight	Height	Type
1	猪猪	true	100		P
2	鸟鸟	false		50	B

其中：

- ①、id:表主键
- ②、name:动物的姓名，所有的动物都有
- ③、sex:动物的性别，所有的动物都有
- ④、weight:猪(Pig)的重量，只有猪才有，所以鸟鸟就没有重量数据
- ⑤、height：鸟(height)的调试，只有鸟才有，所以猪猪就没有高度数据
- ⑥、type:表示动物的类型；P表示猪；B表示鸟

1、 实体类

Animal实体类：

```
public class Animal {
    private int id;
```

```

private String name;
private boolean sex;
public int getId() {return id; }
public void setId(int id) { this.id = id;}
public String getName() {return name;}
public void setName(Stringname) {this.name = name;}
public boolean isSex() {return sex;}
public void setSex(boolean sex) {this.sex = sex;}
}

```

Pig实体类：

```

public class Pig extends Animal {
    private int weight;
    public int getWeight() {return weight;}
    public void setWeight(int weight) {this.weight = weight;}
}


```


Bird实体类：

```

public class Bird extends Animal {
    private int height;
    public int getHeight() {return height;}
    public void setHeight(int height) {this.height = height;}
}

```

 快速回复

 我要收藏

 返回顶部

2、xml方式：映射

```

<class name="Animal" table="t_animal" lazy="false">
    <id name="id">
        <generator class="native"/>
    </id>
    <discriminator column="type" type="string"/>
    <property name="name"/>
    <property name="sex"/>
    <subclass name="Pig" discriminator-value="P">
        <property name="weight"/>
    </subclass>
    <subclass name="Bird" discriminator-value="B">
        <property name="height"/>
    </subclass>
</class>

```

1、理解如何映射

因为类继承树肯定是对应多个类，要把多个类的信息存放在一张表中，必须有某种机制来区分哪些记录是属于哪个类的。

这种机制就是，在表中添加一个字段，用这个字段的值来进行区分。用hibernate实现这种策略的时候，有如下步骤：

父类用普通的<class>标签定义

在父类中定义一个discriminator，即指定这个区分的字段的名称和类型

如：<discriminator column="XXX" type="string"/>

子类使用<subclass>标签定义，在定义subclass的时候，需要注意如下几点：

Subclass标签的name属性是子类的全路径名

在Subclass标签中，用discriminator-value属性来标明本子类的discriminator字段（用来区分不同类的字段）的值Subclass标签，既可以被class标签所包含（这种包含关系正是表明了类之间的继承关系），也可以与class标

签平行。当subclass标签的定义与class标签平行的时候，需要在subclass标签中，添加extends属性，里面的值

是父类的全路径名称。子类的其它属性，像普通类一样，定义在subclass标签的内部。

2、理解如何存储

存储的时候hibernate会自动将鉴别字段值插入到数据库中，在加载数据的时候，hibernate能根据这个鉴别值正确的加载对象

多态查询：在hibernate加载数据的时候能鉴别出真正的类型（instanceOf）

get支持多态查询

load只有在lazy=false，才支持多态查询

hql支持多态查询

3、annotation注解

父类中注解如下：

使用@Inheritance注解为继承映射，再使用strategy属性来指定继承映射的方式

strategy有三个值：InheritanceType.SINGLE_TABLE 单表继承

InheritanceType.TABLE_PER_CLASS 类表继承

InheritanceType.JOINED 具体表继承

再使用@DiscriminatorColumn注意标识字段的字段名，及字段类型

在类中使用@DiscriminatorValue来注解标识字段的值

@Entity

@Inheritance(strategy=InheritanceType.SINGLE_TABLE)

@DiscriminatorColumn(

name="discriminator",

discriminatorType=DiscriminatorType.STRING)

@DiscriminatorValue("person")

public class Person {private int id;private String name;

@Id

@GeneratedValue

public int getId() {return id;}

继承类中注解如下：

只需要使用@DiscriminatorValue来注解标识字段的值

4、导出后生成SQL语句：

create table t_animal (id integer not null auto_increment, type varchar(255) not null, name varchar(255), sex bit, weight integer, height integer, primary key (id))

5、数据库中表结构：

```
mysql> desc t_animal;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id    | int(11) | NO | PRI | NULL | auto_increment |
| type  | varchar(255) | NO | | | |
| name  | varchar(255) | YES | | NULL | |
| sex   | bit(1) | YES | | NULL | |
| weight | int(11) | YES | | NULL | |
| height | int(11) | YES | | NULL | |
+-----+-----+-----+-----+-----+-----+
6 rows in set (0.08 sec)
```

6、单表继承数据存储：

session = HibernateUtils.getSession();

tx=session.beginTransaction();

Pig pig = new Pig();


pig.setName("猪猪");


pig.setSex(true);

pig.setWeight(100);

session.save(pig);

Bird bird = new Bird();

 快速回复

 我要收藏

 返回顶部

```
bird.setName("鸟鸟");
bird.setSex(false);
bird.setHeight(50);
session.save(bird);
```

```
tx.commit();
```

存储执行输出SQL语句：

Hibernate: insert into t_animal (name, sex, weight, type) values (?, ?, ?, 'P') //自动确定无height字段,并且设置标识符为P

Hibernate: insert into t_animal (name, sex, height, type) values (?, ?, ?, 'B') //自动确定无weight字段,并且设置标识符为B

解释：hibernate会根据单表继承映射文件的配置内容，自动在插入数据时哪个子类需要插入哪些字段，并且自动插入标识符字符值（在映射文件中配置了）

7、单表继承映射数据加载(指定加载子类)：

```
session = HibernateUtils.getSession();
tx = session.beginTransaction();
```

```
Pig pig = (Pig) session.load(Pig.class, 1);
System.out.println("pig.name=" + pig.getName());
System.out.println("pig.weight=" + pig.getWeight());
```

```
Bird bird = (Bird) session.load(Bird.class, 2);
System.out.println("bird.name=" + bird.getName());
System.out.println("bird.height=" + bird.getHeight());
```

```
tx.commit();
```

8、加载执行输出SQL语句：

Hibernate: select pig0_.id as id0_0_, pig0_.name as name0_0_, pig0_.sex as sex0_0_, pig0_.weight as weight0_0_ from t_animal pig0_ where pig0_.id=? and pig0_.type='P'

//hibernate会根据映射文件自动确定哪些字段(虽然表中有height,但hibernate并不会加载)属于这个子类，并且确定标识符为B（已经在配置文件中设置了）

pig.name=猪猪

pig.weight=100

Hibernate: select bird0_.id as id0_0_, bird0_.name as name0_0_, bird0_.sex as sex0_0_, bird0_.height as height0_0_ from t_animal bird0_ where bird0_.id=? and bird0_.type='B'

//hibernate会根据映射文件自动确定哪些字段虽然表中有weight,但hibernate并不会加载属于这个子类，并且确定标识符为B（已经在配置文件中设置了）

bird.name=鸟鸟

bird.height=50

9、单表继承映射数据加载(指定加载父类)：

```
session = HibernateUtils.getSession();
tx = session.beginTransaction();
```

//不会发出SQL，返回一个代理类

```
Animal animal = (Animal) session.load(Animal.class, 1);
```

//发出SQL语句，并且加载所有字段的数据(因为使用父类加载对象数据)

```
System.out.println("animal.name=" + animal.getName());
```


```
System.out.println("animal.sex=" + animal.isSex());
```

```
tx.commit();
```

加载执行生成SQL语句：

Hibernate: select animal0_.id as id0_0_, animal0_.name as name0_0_, animal0_.sex as sex0_0_, animal0_.weight as weight0_0_, animal0_.height as height0_0_, animal0_.type as type0_0_ from t_animal

 快速回复

 我要收藏

 返回顶部

```
animal0_ where animal0_.id=?
```

//注：因为使用父类加载数据，所以hibernate会将所有字段(height、weight、type)的数据全部加载，并且条件中没有识别字段type(也就不区分什么子类，把所有子类全部加载上来。)

10、单表继承映射数据加载(指定加载父类，看能否鉴别真实对象)：

```
session = HibernateUtils.getSession();
tx = session.beginTransaction();
```

//不会发出SQL语句(load默认支持延迟加载(lazy))，返回一个animal的代理对象(此代理类是继承Animal生成的，也就是说Animal一个子类)

```
Animal animal =(Animal)session.load(Animal.class, 1);
```

//因为在上面返回的是一个代理类(父类的一个子类)，所以animal不是Pig

//通过instanceof是反应不出正直的对象类型的，因此load在默认情况下是不支持多态查询的。

```
if (animal instanceof Pig) {
    System.out.println("是猪");
} else {
    System.out.println("不是猪");//这就是结果
}

System.out.println("animal.name=" + animal.getName());
System.out.println("animal.sex=" + animal.isSex());
```

```
tx.commit();
```

11、多态查询：

在hibernate加载数据的时候能鉴别出正直的类型(通过instanceof)

get支持多态查询；load只有在lazy=false，才支持多态查询；HQL支持多态查询

12、采用load，通过Animal查询，将<class>标签上的lazy设置为false

```
session = HibernateUtils.getSession();
tx = session.beginTransaction();
```

//会发出SQL语句，因为设置lazy为false，不支持延迟加载

```
Animal animal =(Animal)session.load(Animal.class, 1);
```

//可以正确的判断出Pig的类型，因为lazy=false，返回具体的Pid类型

//此时load支持多态查询

```
if (animal instanceof Pig) {
    System.out.println("是猪");//结果
} else {
    System.out.println("不是猪");
}

System.out.println("animal.name=" + animal.getName());
System.out.println("animal.sex=" + animal.isSex());
```

```
tx.commit();
```

13、采用get，通过Animal查询,可以判断出正直的类型

```
session = HibernateUtils.getSession();
tx = session.beginTransaction();
```

//会发出SQL语句，因为get不支持延迟加载，返回的是正直的类型，

```
Animal animal =(Animal)session.load(Animal.class, 1);
```

//可以判断出正直的类型

//get是支持多态查询

```
if (animal instanceof Pig) {
    System.out.println("是猪");//结果
} else {
```



```
        System.out.println("不是猪");
    }
    System.out.println("animal.name=" + animal.getName());
    System.out.println("animal.sex=" + animal.isSex());

    tx.commit();
```

14、采用HQL查询，HQL是否支持多态查询

```
List animalList = session.createQuery("from Animal").list();
for (Iterator iter = animalList.iterator(); iter.hasNext();) {
    Animal a = (Animal)iter.next();
    //能够正确鉴别出正直的类型，HQL是支持多态查询的。
    if (a instanceof Pig) {
        System.out.println("是Pig");
    } else if (a instanceof Bird) {
        System.out.println("是Bird");
    }
}
```

快速回复

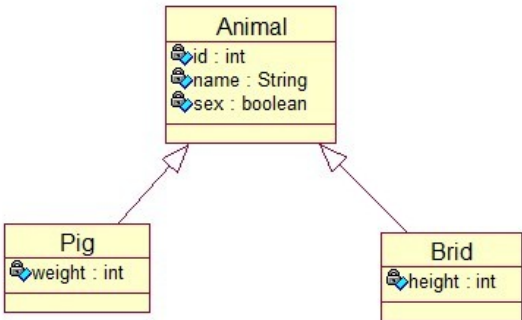
我要收藏

返回顶部

15、通过HQL查询表中所有的实体对象

```
* HQL语句：session.createQuery("from java.lang.Object").list();
* 因为所有对象都是继承Object类
List list = session.createQuery("from java.lang.Object").list();
for (Iterator iter = list.iterator(); iter.hasNext();){
    Object o =iter.next();
    if (o instanceof Pig) {
        System.out.println("是Pig");
    } else {
        System.out.println("是Bird");
    }
}
```

(四) 具体表继承JOINED：
每个类映射成一个表(table per subclass)
对象模型不用变化，存储模型需要变化



1、关系模型：

每个类映射成一个表(table per subclass)

t_animal		
Id	Name	Sex
1	猪猪	True
2	鸟鸟	False
t_pig		
Pigid	Weight	
1	100	
t_bird		
Birdid	Height	
2	50	

注：因为需要每个类都映射成一张表，所以Animal也映射成一张表(t_animal),表中字段为实体类属性而pig子类也需要映射成一张表(t_pid)，但为了与父类联系需要加入一个外键(pidid)指向父类映射成的表(t_animal),字段为子类的扩展属性。Bird子类同样也映射成一张表(t_bird),也加入一个外键(birdid)指向父类映射成的表(t_animal),字段为子类的扩展属性。

2、xml方式（每个类映射成一个表）：

```
<class name="com.wjt276.hibernate.Animal" table="t_animal">
    <id name="id"column="id"><!-- 映射主键 -->
        <generator class="native"/>
    </id>
    <property name="name"/><!-- 映射普通属性 -->
    <property name="sex"/>
    <!--<joined-subclass>标签：继承映射 每个类映射成一个表 -->
    <joined-subclass name="com.wjt276.hibernate.Pig" table="t_pig">
<!-- <key>标签：会在相应的表（当前映射的表）里，加入一个外键，参照指向当前类的父类(当前Class标签对
象的表(t_animal))-->
        <key column="pidid"/>
        <property name="weight"/>
    </joined-subclass>
    <joined-subclass name="com.wjt276.hibernate.Bird" table="t_bird">
        <key column="birdid"/>
        <property name="height"/>
    </joined-subclass>
</class>
```

[快速回复](#)
[我要收藏](#)
[返回顶部](#)

1、理解如何映射

这种策略是使用joined-subclass标签来定义子类的。父类、子类，每个类都对应一张数据库表。在父类对应的数据库表中，实际上会存储所有的记录，包括父类和子类的记录；在子类对应的数据库表中，这个表只定义了子类中所特有的属性映射的字段。子类与父类，通过相同的主键值来关联。实现这种策略的时候，

有如下步骤：

父类用普通的<class>标签定义即可

父类不再需要定义discriminator字段

子类用<joined-subclass>标签定义，在定义joined-subclass的时候，需要注意如下几点：

Joined-subclass标签的name属性是子类的全路径名

Joined-subclass标签需要包含一个key标签，这个标签指定了子类和父类之间是通过哪个字段来关联的。

如：<key column="PARENT_KEY_ID"/>，这里的column，实际上就是父类的主键对应的映射字段名称。

Joined-subclass标签，既可以被class标签所包含（这种包含关系正是表明了类之间的继承关系），

也可以与class标签平行。当Joined-subclass标签的定义与class标签平行的时候，需要在Joined-subclass标签中，添加extends属性，里面的值是父类的全路径名称。子类的其它属性，像普通类一样，定义在joined-subclass标签的内部。

3、annotation注解

因为，子类生成的表需要引用父类生成的表，所以只需要在父类设置具体表继承映射就可以了，其它子类只需要使用@Entity注解就可以了

@Entity

@Inheritance(strategy=InheritanceType.JOINED)

```
public class Person {
    private int id;
    private String name;
    @Id
    @GeneratedValue
    public int getId() {return id;}
}
```

4、导出输出SQL语句：

```
create table t_animal (id integer notnull auto_increment, name varchar(255), sex bit, primary key (id))
```

[关闭](#)

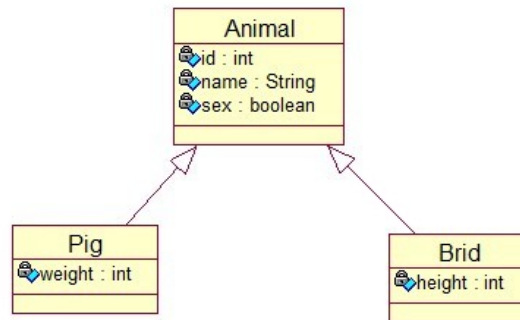
```
create table t_bird (birdid integernot null, height integer, primary key (birdid))
create table t_pig (pigid integer notnull, weight integer, primary key (pigid))
altertable t_bird add index FKCB5B05A4A554009D(birdid), add constraint FKCB5B05A4A554009D foreign
key (birdid) references t_animal (id)
alter table t_pig add index FK68F8743FE77AC32 (pigid), add constraint FK68F8743FE77AC32 foreign key
(pigid) references t_animal (id)
//共生成三个表，并在子类表中各有一个外键参照指向父类表
```

数据的存储，不需要其它的任务变化，直接使用单表继承存储就可以了，加载也是一样。
具体表继承效率没有单表继承高，但是单表继承会出现多余的冗余字段，具体表层次分明

(五) 类表继承 TABLE_PER_CLASS

每个具体类映射成一个表(table per concrete class)(有一些限制)

-
对象模型不用变化，存储模型需要变化



快速回复

我要收藏

返回顶部

1、关系模型：

每个具体类(Pig、Bird)映射成一个表(table per concrete class)(有一些限制)

t_pig

Id	Name	Sex	Weight
1	猪猪	True	100

t_bird

Id	Name	Sex	Height
2	鸟鸟	False	50

2、xml方式：映射文件：

```
<class name="com.wjt276.hibernate.Animal" table="t_animal">
  <id name="id"column="id"><!-- 映射主键 -->
    <generator class="assigned"/><!-- 每个具体类映射一个表主键生成策略不可使用native -->    </id>
  <property name="name"/><!-- 映射普通属性 -->
  <property name="sex"/>
  <!--使用<union-subclass>标签来映射"每个具体类映射成一张表"的映射关系
    , 实现上上面的表t_animal虽然映射到数据库中，但它没有任何作用。 -->
  <union-subclass name="com.wjt276.hibernate.Pig" table="t_pig">
    <property name="weight"/>
  </union-subclass>
  <union-subclass name="com.wjt276.hibernate.Bird" table="t_bird">
    <property name="height"/>
  </union-subclass>
</class>
```

理解如何映射

这种策略是使用union-subclass标签来定义子类的。每个子类对应一张表，而且这个表的信息是完备的，即包含了所有从父类继承下来的属性映射的字段（这就是它跟joined-subclass的不同之处，joined-subclass定义的子类的表，只包含子类特有属性映射的字段）。实现这种策略的时候，有如下步骤：
父类用普通<class>标签定义即可
子类用<union-subclass>标签定义，在定义union-subclass的时候，需要注意如下几点：
Union-subclass标签不再需要包含key标签（与joined-subclass不同）

关闭

Union-subclass标签，既可以被class标签所包含（这种包含关系正是表明了类之间的继承关系），也可以与class标签平行。当Union-subclass标签的定义与class标签平行的时候，需要在Union-subclass标签中，添加extends属性，里面的值是父类的全路径名称。子类的其它属性，像普通类一样，定义在Union-subclass标签的内部。这个时候，虽然在union-subclass里面定义的只有子类的属性，但是因为它继承了父类，所以，不需要定义其它的属性，在映射到数据库表的时候，依然包含了父类的所有属性的映射字段。

注意：在保存对象的时候id是不能重复的（不能使用自增生成主键）

3、annotation注解

只需要对父类进行注解就可以了，

因为子类表的ID是不可以重复，所以一般的主键生成策略已经不适应了，只有表主键生成策略。

首先使用@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)来注解继承映射，并且使用具体表继承方式，使用@TableGenerator来申明一个表主键生成策略

再在主键上@GeneratedValue(generator="t_gen", strategy=GenerationType.TABLE)来注解生成策略为表生成策略，并且指定表生成策略的名称

继承类只需要使用@Entity进行注解就可以了

@Entity

@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)

@TableGenerator(

name="t_gen",

table="t_gen_table",

pkColumnName="t_pk",

valueColumnName="t_value",

pkColumnValue="person_pk",

initialValue=1,

allocationSize=1

)

public class Person {

private int id;

private String name;

@Id

@GeneratedValue(generator="t_gen", strategy=GenerationType.TABLE)

public int getId() {return id;}

4、导出输出SQL语句：

create table t_animal (id integer not null, name varchar(255), sex bit, primary key (id))

create table t_bird (id integer not null, name varchar(255), sex bit, height integer, primary key(id))

create table t_pig (id integer not null, name varchar(255), sex bit, weight integer, primary key(id))

注：表t_animal、t_bird、t_pig并不是自增的，是因为bird、pig都是animal类，也就是说animal不可以有相同的ID号(Bird、Pig是类型，只是存储的位置不同而以)

5、数据库表结构如下：




```
mysql> show tables;
+-----+
| Tables_in_hibernate_extends_3 |
+-----+
| t_animal                       |
| t_bird                        |
| t_pig                         |
+-----+
3 rows in set (0.00 sec)
```

```
mysql> desc t_animal;
+-----+
| Field | Type          | Null | Key | Default | Extra |
+-----+
| id     | int(11)       | NO   | PRI |         |       |
| name   | varchar(255)  | YES  |     | NULL    |       |
| sex    | bit(1)        | YES  |     | NULL    |       |
+-----+
3 rows in set (0.00 sec)
```

```
mysql> desc t_pig;
+-----+
| Field | Type          | Null | Key | Default | Extra |
+-----+
| id     | int(11)       | NO   | PRI |         |       |
| name   | varchar(255)  | YES  |     | NULL    |       |
| sex    | bit(1)        | YES  |     | NULL    |       |
| weight | int(11)       | YES  |     | NULL    |       |
+-----+
4 rows in set (0.00 sec)
```

```
mysql> desc t_bird;
+-----+
| Field | Type          | Null | Key | Default | Extra |
+-----+
| id     | int(11)       | NO   | PRI |         |       |
| name   | varchar(255)  | YES  |     | NULL    |       |
| sex    | bit(1)        | YES  |     | NULL    |       |
| height | int(11)       | YES  |     | NULL    |       |
+-----+
4 rows in set (0.00 sec)
```

t_animal没有实际作用

注：如果不想t_animal存在(因为它没有实际的作用)，可以设置<class>标签中的abstract="true"(抽象表)，这样在导出至数据库时，就不会生成t_animal表了。

```
<class name="com.wjt276.hibernate.Animal" table="t_animal" abstract="true">
    <id name="id" column="id"><!-- 映射主键 -->
```

(六) 三种继承关联映射的区别：

- 1、 第一种：它把所有数据都存入一个表中，优点：效率好（操作的就是一个表）；缺点：存在冗余字段，如果将冗余字段设置为非空，则就无法存入数据；
- 2、 第二种：层次分明，缺点：效率不好（表间存在关联表）
- 3、 第三种：主键字段不可以设置为自增主键生成策略。

一般使用第一种

第17课 hibernate树形结构(重点)

树形结构：也就是目录结构，有父目录、子目录、文件等信息，而在程序中树形结构只是称为节点。

一棵树有一个根节点，而根节点也有一个或多个子节点，而一个子节点有且仅有一个父节点(当前除根节点外)，而且也存在一个或多个子节点。

也就是说树形结构，重点就是节点，也就是我们需要关心的节点对象。

节点：一个节点有一个ID、一个名称、它所属的父节点(根节点无父节点或为null)，有一个或多的子节点等其它信息。

Hibernate将节点抽取成实体类，节点相对于父节点是“多对一”映射关系，节点相对于子节点是“一对多”映射关系。

一、 节点实体类：

```
/** * 节点*/
public class Node {
    private int id; //标识符
    private String name; //节点名称
    private int level; //层次，为了输出设计
```

快速回复

我要收藏

返回顶部

```

private boolean leaf; //是否为叶子节点，这是为了效率设计，可有可无
//父节点：因为多个节点属于一个父节点，因此用hibernate映射关系说是“多对一”
private Node parent;
//子节点：因为一个节点有多个子节点，因此用hibernate映射关系说是“一对多”
private Set children;

public int getId() {return id;}
public void setId(int id) {this.id = id;}
public String getName() {return name;}
public void setName(String name) { this.name = name;}
public int getLevel() { return level;}
public void setLevel(int level) {this.level = level;}
public boolean isLeaf() {return leaf;}
public void setLeaf(boolean leaf) {this.leaf = leaf;}
public Node getParent() {return parent;}
public void setParent(Node parent) {this.parent = parent;}
public Set getChildren() {return children;}
public void setChildren(Set children) {this.children = children;}

```

二、xml方式：映射文件：

```

<class name="com.wjt276.hibernate.Node" table="t_node">
    <id name="id" column="id">
        <generator class="native"/>
    </id>
    <property name="name"/>
    <property name="level"/>
    <property name="leaf"/>

```

!— 一对多：加入一个外键，参照当前表t_node主键，而属性parent类型为Node，也就是当前类，则会在同一个表中加入这个字段，参照这个表的主键-->

```

<many-to-one name="parent" column="pid"/>
<!--<set>标签是映射一对多的方式，加入一个外键，参照主键。-->
<set name="children" lazy="extra" inverse="true">
    <key column="pid"/>
    <one-to-many class="com.wjt276.hibernate.Node"/>
</set>
</class>

```

三、annotation注解

因为树型节点所有的数据，在数据库中只是存储在一个表中，而对于实体类来说，节点对子节点来说是一对多的关系，而对于父节点来说是多对一的关系。因此可以在一个实体类中注解。如下

@Entity

```

public class Node {
    private int id; // 标识符
    private String name; // 节点名称
    private int level; // 层次，为了输出设计
    private boolean leaf; // 是否为叶子节点，这是为了效率设计，可有可无
    // 父节点：因为多个节点属于一个父节点，因此用hibernate映射关系说是“多对一”
    private Node parent;
    // 子节点：因为一个节点有多个子节点，因此用hibernate映射关系说是“一对多”
    private Set<Node> children = new HashSet<Node>();
    @Id
    @GeneratedValue
    public int getId() {return id;}
    @OneToMany(mappedBy="parent")
    public Set<Node> getChildren() {return children;}
    @ManyToOne

```



```
@JoinColumn(name="pid")
public Node getParent() {return parent;}
```

四、 测试代码：

```
public class NodeTest extends TestCase {
    //测试节点的存在
    public void testSave1(){
        NodeManage.getInstance().createNode("F:\\hibernate\\hibernate_training_tree");
    }
    //测试节点的加载
    public void testPrintById(){
        NodeManage.getInstance().printNodeById(1);
    }
}
```

五、 相应的类代码：

```
public class NodeManage {
    private static NodeManage nodeManage= new NodeManage();
    private NodeManage(){}//因为要使用单例，所以将其构造方法私有化
    //向外提供一个接口
    public static NodeManage getInstance(){
        return nodeManage;
    }
    /**
     * 创建树
     * @param filePath 需要创建树目录的根目录
     */
    public void createNode(String dir) {
        Session session = null;
        try {
            session =HibernateUtils.getSession();
            session.beginTransaction();

            File root = new File(dir);
            //因为第一个节点无父节点，因为是null
            this.saveNode(root, session, null,0);

            session.getTransaction().commit();
        } catch (HibernateException e) {
            e.printStackTrace();
            session.getTransaction().rollback();
        } finally {
            HibernateUtils.closeSession(session);
        }
    }
    /**
     * 保存节点对象至数据库
     * @param file 节点所对应的文件
     * @param session session
     * @param parent 父节点
     * @param level 级别
     */
    public void saveNode(File file, Session session, Node parent, int level) {
        if (file == null ||!file.exists()){return;}
        //如果是文件则返回true,则表示是叶子节点，否则为目录，非叶子节点
        boolean isLeaf = file.isFile();
        Node node = new Node();
```

[快速回复](#)
[我要收藏](#)
[返回顶部](#)

```
node.setName(file.getName());
node.setLeaf(isLeaf);
node.setLevel(level);
node.setParent(parent);

session.save(node);

//进行循环迭代子目录
File[] subFiles = file.listFiles();
if (subFiles != null &&subFiles.length > 0){
    for (int i = 0; i <subFiles.length ; i++){
        this.saveNode(subFiles[i],session, node, level + 1);
    }
}

/**
 * 输出树结构
 * @param id
 */
public void printNodeById(int id) {

    Session session = null;



    try {
        session =HibernateUtils.getSession();
        session.beginTransaction();

        Node node =(Node)session.get(Node.class, 1);

        printNode(node);

        session.getTransaction().commit();
    } catch (HibernateException e) {
        e.printStackTrace();
        session.getTransaction().rollback();
    } finally {
        HibernateUtils.closeSession(session);
    }
}

private void printNode(Node node) {
    if (node == null){ return; }
    int level = node.getLevel();
    if (level > 0){
        for (int i = 0; i < level; i++){
            System.out.print("  ");
        }
        System.out.print("--");
    }
    System.out.println(node.getName() +(node.isLeaf() ? "" : "[" + node.getChildren().size() +"]"));
    Set children = node.getChildren();
    for (Iterator iter =children.iterator(); iter.hasNext(); ){
        Node child = (Node)iter.next();
        printNode(child);
    }
}
```

 快速回复 我要收藏 返回顶部

第18课 作业-学生、课程、分数的映射关系

关闭

一、设计

- 1、 实体类(表)
- 2、 导航(编程方便)
 - a) 通过学生 取出 学生所先的课程
 - b) 但是通过课程 取出 学该课程的学生不好。学的学生太多
 - c) 确定编程的方式
- 3、 可以利用联合主键映射可以，
 - a) 学生生成一个表
 - b) 课程生成一个表
 - c) 再生成一个表，主键是联合主键(学生ID、课程ID) + 学生共生成一个表
- 4、 也可以利用一对多，多对多 都可以(推荐)
 - a) 学生生成一个表
 - b) 课程生成一个表
 - c) 分数生成一个表，并且有两个外键，分别指向学生、课程表

二、代码：

* 课程

@Entity

```
public class Course {
    private int id;
    private String name;
    @Id
    @GeneratedValue
    public int getId() {return id;}
    public void setId(int id) { this.id = id;}
    public String getName() {return name;}
    public void setName(Stringname) {this.name = name;}}
```

* 分数

@Entity

@Table(name = "score")


```
public class Score {
    private int id;
    private int score;
    private Student student;
    private Course course;
    @Id
    @GeneratedValue
    public int getId() {return id;}
    @ManyToOne
    @JoinColumn(name = "student_id")
    public StudentgetStudent() {return student;}
    @ManyToOne
    @JoinColumn(name = "score_id")
    public Course getCourse(){ return course;}
    public int getScore() { return score;}
    public void setScore(int score) {this.score = score;}
    public void setStudent(Studentstudent) {this.student = student;}
    public void setCourse(Coursecourse) {this.course = course;}
    public void setId(int id) { this.id = id;}}
```

* 学生通过课程可以导航到分数

@Entity

```
public class Student {
    private int id;
```

 快速回复

 我要收藏

 返回顶部

```

private String name;
private Set<Course> courses = new HashSet<Course>();
@Id
@GeneratedValue
public int getId() {return id;}
@ManyToOne
@JoinTable(name = "score", //此表就是Score实体类在数据库生成的表叫score
    joinColumns= @JoinColumn(name = "student_id"),
    inverseJoinColumns= @JoinColumn(name = "course_id")
)
public Set<Course>getCourses() {return courses;}
public void setCourses(Set<Course> courses) {this.courses = courses;}
public void setId(int id) { this.id = id;}
public String getName() {return name;}
public void setName(String name) {this.name = name;}}

```

三、 注意

在Student实体类中的使用的第三方表使用了两个字段，而hibernate会使这两个字段生成联合主键，这并非我们需要的结果，因为我们需要手动到数据库中修改。这样才可以存储数据，否则数据存储不进去。hibernate的一个小bug

快速回复

☆ 我要收藏

^ 返回顶部

第19课 Hibernate查询(Query Language)

HQL VS EJBQL

一、 Hibernate可以使用的查询语言

- 1、 NativeSQL：本地语言(数据库自己的SQL语句)
- 2、 HQL：Hibernate自带的查询语句，可以使用HQL语言，转换成具体的方言
- 3、 EJBQL：JPQL 1.0，可以认为是HQL的一个子节(重点)
- 4、 QBC：Query By Criteria
- 5、 QBE：Query By Example

注意：上面的功能是从1至5的比较，1的功能最大，5的功能最小

二、 实例一

- 1、 版块

/** 版块*/

@Entity

```

public class Category {
    private int id;
    private String name;
    @Id
    @GeneratedValue
    public int getId() {return id;}
    public void setId(int id) {this.id = id;}
    public String getName() {return name;}
    public void setName(String name) {this.name = name;}}

```

- 2、 主题

/**主题*/

@Entity

```

public class Topic {
    private int id;
    private String title;
    private Category category;
    //private Category category2;

```

```

private Date createDate;
public DategetCreateDate() {return createDate;}
public void setCreateDate(DatecreateDate) {this.createDate = createDate;}
@ManyToOne(fetch=FetchType.LAZY)
public CategorygetCategory() { return category;}
public voidsetCategory(Category category) {this.category = category; }
@Id
@GeneratedValue
public int getId() {return id;}
public void setId(int id) {this.id = id;}
public String getTitle(){return title;}
public void setTitle(Stringtitle) {this.title = title;}}

```

3、 主题回复

/**主题回复*/

@Entity

```

public class Msg {
    private int id;
    private String cont;
    private Topic topic;
    @ManyToOne
    public Topic getTopic() {return topic;}
    public void setTopic(Topictopic) {this.topic = topic;}
    @Id
    @GeneratedValue
    public int getId() {return id;}
    public void setId(int id) {this.id = id;}
    public String getCont() {return cont;}
    public void setCont(Stringcont) {this.cont = cont;}}

```

4、 临时类


/**临时类 */

```

public class MsgInfo { //VO DTO Value Object username p1 p2UserInfo->User->DB
    private int id;
    private String cont;
    private String topicName;
    private String categoryName;
    public MsgInfo(int id, String cont,String topicName, String categoryName) {
        super();
        this.id = id;
        this.cont = cont;
        this.topicName = topicName;
        this.categoryName =categoryName;
    }
    public StringgetTopicName() {return topicName;}
    public voidsetTopicName(String topicName) {this.topicName = topicName;}
    public StringgetCategoryName() {return categoryName;}
    public voidsetCategoryName(String categoryName) {
        this.categoryName =categoryName;
    }
    public int getId() {return id;}
    public void setId(int id) {this.id = id;}
    public String getCont() {return cont;}
    public void setCont(Stringcont) {this.cont = cont;}}

```

三、 实体一测试代码：

 快速回复

 我要收藏

 返回顶部

//初始化数据

```
@Test
public void testSave() {
    Session session = sf.openSession();
    session.beginTransaction();
    for(int i=0; i<10; i++){
        Category c = new Category();
        c.setName("c" + i);
        session.save(c);
    }
    for(int i=0; i<10; i++){
        Category c = new Category();
        c.setId(1);
        Topic t = new Topic();
        t.setCategory(c);
        t.setTitle("t" + i);
        t.setCreateDate(new Date());
        session.save(t);
    }
    for(int i=0; i<10; i++){
        Topic t = new Topic();
        t.setId(1);
        Msg m = new Msg();
        m.setCont("m" + i);
        m.setTopic(t);
        session.save(m);
    }
    session.getTransaction().commit();
    session.close();
}

/** QL:from + 实体类名称 */
Query q = session.createQuery("from Category");
List<Category> categories = (List<Category>)q.list();
for(Category c : categories) {
    System.out.println(c.getName());
}

/* 可以为实体类起个别名，然后使用它 */
Query q = session.createQuery("from Category c where c.name > 'c5'");
List<Category> categories = (List<Category>)q.list();
for(Category c : categories) {
    System.out.println(c.getName());
}

//排序
Query q = session.createQuery("from Category c order by c.name desc");
List<Category> categories = (List<Category>)q.list();
for(Category c : categories) {
    System.out.println(c.getName());
}
```

* 为加载上来的对象属性起别名，还可以使用


[快速回复](#)[我要收藏](#)[返回顶部](#)


```
Query q =session.createQuery("select distinct c fromCategory c order by c.name desc");
List<Category>categories = (List<Category>)q.list();
for(Category c : categories) {
    System.out.println(c.getName());
}
```

```
/*Query q = session.createQuery("from Category c where c.id > :minand c.id < :max");
//q.setParameter("min",2);
//q.setParameter("max",8);
q.setInteger("min",2);
q.setInteger("max",8);*/
```

* 可以使用冒号(:),作为占位符, 来接受参数使用。如下(链式编程)

```
Query q =session.createQuery("from Category c wherec.id > :min and c.id < :max")
    .setInteger("min", 2)
    .setInteger("max", 8);
List<Category>categories = (List<Category>)q.list();
for(Category c : categories) {
    System.out.println(c.getId()+ "-" + c.getName());
}
```

 快速回复

 我要收藏

 返回顶部

```
Query q =session.createQuery("from Category c wherec.id > ? and c.id < ?");
q.setParameter(0, 2)
    .setParameter(1, 8);
// q.setParameter(1, 8);
List<Category>categories = (List<Category>)q.list();
for(Category c : categories) {
    System.out.println(c.getId()+ "-" + c.getName());
}
```

//分页

```
Query q =session.createQuery("from Category c orderby c.name desc");
q.setMaxResults(4);//每页显示的最大记录数
q.setFirstResult(2);//从第几条开始显示, 从0开始
List<Category>categories = (List<Category>)q.list();
for(Category c : categories) {
    System.out.println(c.getId()+ "-" + c.getName());
}
```

```
Query q =session.createQuery("select c.id, c.name from Category c order by c.namedesc");
List<Object[]>categories = (List<Object[]>)q.list();
for(Object[] o : categories) {
    System.out.println(o[0] + "-" + o[1]);
}
```

//设定fetch type 为lazy后将不会有第二条sql语句

```
Query q =session.createQuery("from Topic t wheret.category.id = 1");
List<Topic>topics = (List<Topic>)q.list();
for(Topic t : topics) {
    System.out.println(t.getTitle());
    //System.out.println(t.getCategory().getName());
}
```

//设定fetch type 为lazy后将不会有第二条sql语句

```
Query q =session.createQuery("from Topic t wheret.category.id = 1");
```

```
List<Topic>topics = (List<Topic>)q.list();
for(Topic t : topics) {
    System.out.println(t.getTitle());
}


Query q =session.createQuery("from Msg m wherem.topic.category.id = 1");
for(Object o : q.list()) {
    Msg m = (Msg)o;
    System.out.println(m.getCont());
}
```


//了解即可

//VO Value Object

//DTO data transfer object

```
Query q =session.createQuery("select newcom.bjsxt.hibernate.MsgInfo(m.id, m.cont, m.topic.title,
m.topic.category.name)from Msg");
for(Object o : q.list()) {
    MsgInfo m = (MsgInfo)o;
    System.out.println(m.getCont());
}
```

 快速回复

 我要收藏

 返回顶部

//动手测试left right join

//为什么不能直接写Category名，而必须写t.category

//因为有可能存在多个成员变量（同一个类），需要指明用哪一个成员变量的连接条件来做连接

```
Query q =session.createQuery("select t.title, c.namefrom Topic t join t.category c "); //join Category c
for(Object o : q.list()) {
    Object[] m = (Object[])o;
    System.out.println(m[0] + "-" + m[1]);
}
```

//学习使用uniqueResult

```
Query q = session.createQuery("from Msg m where m = :MsgToSearch "); //不重要
Msg m = new Msg();
m.setId(1);
q.setParameter("MsgToSearch", m);
Msg mResult =(Msg)q.uniqueResult();
System.out.println(mResult.getCont());
```

```
Query q =session.createQuery("select count(*) fromMsg m");
long count = (Long)q.uniqueResult();
System.out.println(count);
```

```
Query q = session.createQuery("select max(m.id), min(m.id), avg(m.id), sum(m.id)from Msg m");
Object[] o =(Object[])q.uniqueResult();
System.out.println(o[0] + "-" + o[1] + "-" + o[2] + "-" + o[3]);
```

```
Query q =session.createQuery("from Msg m where m.idbetween 3 and 5");
for(Object o : q.list()) {
    Msg m = (Msg)o;
    System.out.println(m.getId()+ "-" + m.getCont());
}
```

```
Query q =session.createQuery("from Msg m where m.idin (3,4, 5)");
for(Object o : q.list()) {
```

```

        Msg m = (Msg)o;
        System.out.println(m.getId()+"-" + m.getCont());
    }

```

//is null 与 is notnull

```

        Query q =session.createQuery("from Msg m where m.contis not null");
        for(Object o : q.list()) {
            Msg m = (Msg)o;
            System.out.println(m.getId()+"-" + m.getCont());
        }

```

四、 实例二

注意：实体二，实体类，只是在实体一的基础上修改了Topic类，添加了多对一的关联关系

@Entity

```

@NamedQueries({
    @NamedQuery(name="topic.selectCertainTopic", query="from Topic t where t.id = :id")
})
/*@NamedNativeQueries(
{
    @NamedNativeQuery(name="topic.select2_5Topic",query="select * from topic limit 2, 5")
})*

```

```

public class Topic {
    private int id;
    private String title;
    private Category category;
    private Date createDate;
    private List<Msg> msgs = new ArrayList<Msg>();
    @OneToMany(mappedBy="topic")
    public List<Msg> getMsgs() {return msgs;}
    public void setMsgs(List<Msg> msgs) {this.msgs = msgs;}
    public Date getCreateDate() {return createDate;}
    public void setCreateDate(Date createDate) {this.createDate = createDate; }
    @ManyToOne(fetch=FetchType.LAZY)
    public Category getCategory() { return category;}
    public void setCategory(Category category) {this.category = category;}
    @Id
    @GeneratedValue
    public int getId() {return id;}
    public void setId(int id) {this.id = id;}
    public String getTitle() { return title;}
    public void setTitle(String title) {this.title = title;}}

```

五、 实例二测试代码

注意：测试数据是实例一的测试数据

```

//is empty and is not empty
        Query q =session.createQuery("from Topic t wheret.msgs is empty");
        for(Object o : q.list()) {
            Topic t = (Topic)o;
            System.out.println(t.getId()+"-" + t.getTitle());
        }

        Query q =session.createQuery("from Topic t wheret.title like '%5'");

```



```

for(Object o : q.list()) {
    Topic t = (Topic)o;
    System.out.println(t.getId()+ "-" + t.getTitle());
}

Query q =session.createQuery("from Topic t wheret.title like '_5'");
for(Object o : q.list()) {
    Topic t = (Topic)o;
    System.out.println(t.getId()+ "-" + t.getTitle());
}

//不重要
Query q = session.createQuery("select lower(t.title)," +
    "upper(t.title)," +
    "trim(t.title)," +
    "concat(t.title, '***')," +
    "length(t.title)" +
    " from Topict ");

for(Object o : q.list()) {
    Object[] arr = (Object[])o;
    System.out.println(arr[0] + "-" + arr[1] + "-" + arr[2] + "-" + arr[3] + "-" + arr[4] + "-");
}

Query q =session.createQuery("select abs(t.id)," +
    "sqrt(t.id)," +
    "mod(t.id,2)" +
    " from Topict ");

for(Object o : q.list()) {
    Object[] arr =(Object[])o;
    System.out.println(arr[0] + "-" + arr[1] + "-" + arr[2] );
}

Query q = session.createQuery("selectcurrent_date, current_time, current_timestamp, t.id from Topic t");
for(Object o : q.list()) {
    Object[] arr =(Object[])o;
    System.out.println(arr[0] + " | " + arr[1] + " | " + arr[2] + " | " + arr[3]);
}

Query q =session.createQuery("from Topic t wheret.createDate < :date");
q.setParameter("date", new Date());
for(Object o : q.list()) {
    Topic t = (Topic)o;
    System.out.println(t.getTitle());
}

Query q =session.createQuery("select t.title,count(*) from Topic t group by t.title" );
for(Object o : q.list()) {
    Object[] arr =(Object[])o;
    System.out.println(arr[0] + "|" + arr[1]);
}

Query q = session.createQuery("selectt.title, count(*) from Topic t group by t.title having count(*) >= 1" );
for(Object o : q.list()) {
    Object[] arr =(Object[])o;

```

 快速回复 我要收藏 返回顶部

```

        System.out.println(arr[0] + "|" + arr[1]);
    }

    Query q =session.createQuery("from Topic t where t.id< (select avg(t.id) from Topic t)" );
    for(Object o : q.list()) {
        Topic t = (Topic)o;
        System.out.println(t.getTitle());
    }

    Query q =session.createQuery("from Topic t where t.id< ALL (select t.id from Topic t where mod(t.id,
2)= 0) " );
    for(Object o : q.list()) {
        Topic t = (Topic)o;
        System.out.println(t.getTitle());
    }

    //用in 可以实现exists的功能
    //但是exists执行效率高
    // t.id not in (1)
    Query q =session.createQuery("from Topic t where notexists (select m.id from Msg m where
m.topic.id=t.id)" );
    // Query q =session.createQuery("from Topic t where exists (select m.id from Msg mwhere
m.topic.id=t.id)" );
    for(Object o : q.list()) {
        Topic t = (Topic)o;
        System.out.println(t.getTitle());
    }

    //update and delete
    //规范并没有说明是不是要更新persistent object , 所以如果要使用 , 建议在单独的trasaction中执行
    Query q =session.createQuery("update Topic t sett.title = upper(t.title)" );
    q.executeUpdate();
    q = session.createQuery("from Topic");
    for(Object o : q.list()) {
        Topic t = (Topic)o;
        System.out.println(t.getTitle());
    }
    session.createQuery("update Topic t set t.title = lower(t.title)")
        .executeUpdate();

    //不重要
    Query q =session.getNamedQuery("topic.selectCertainTopic");
    q.setParameter("id", 5);
    Topic t =(Topic)q.uniqueResult();
    System.out.println(t.getTitle());

    //Native ( 了解 )
    SQLQuery q =session.createSQLQuery("select *from category limit 2,4").addEntity(Category.class);
    List<Category>categories = (List<Category>)q.list();
    for(Category c : categories) {
        System.out.println(c.getName());
    }

    public void testHQL_35() {

```

快速回复

我要收藏

返回顶部

```
//尚未实现JPA命名的NativeSQL
}
```

第20课 Query by Criteria(QBC)

QBC(Query By Criteria)查询方式是Hibernate提供的“更加面向对象”的一种检索方式。QBC在条件查询上比HQL查询更为灵活，而且支持运行时动态生成查询语句。

在Hibernate应用中使用QBC查询通常经过3个步骤

- (1)使用Session实例的createCriteria()方法创建Criteria对象
- (2)使用工具类Restrictions的相关方法为Criteria对象设置查询对象
- (3)使用Criteria对象的list()方法执行查询，返回查询结果

一、 实体代码：

注意：数据是使用Hibernate查询章节的数据

```
//criterion 标准/准则/约束
Criteria c =session.createCriteria(Topic.class) //from Topic
    .add(Restrictions.gt("id", 2)) //greater than = id > 2
    .add(Restrictions.lt("id", 8)) //little than = id < 8
    .add(Restrictions.like("title", "t_"))
    .createCriteria("category")
    .add(Restrictions.between("id", 3, 5)) //category.id >= 3 and category.id <=5
;

//DetachedCriterea
for(Object o : c.list()) {
    Topic t = (Topic)o;
    System.out.println(t.getId()+ "-" + t.getTitle());
}
```

快速回复

我要收藏

返回顶部

二、 Restrictions用法

Hibernate中Restrictions的方法	说明
Restrictions.eq	=
Restrictions.allEq	利用Map来进行多个等于的限制
Restrictions.gt	>
Restrictions.ge	> =
Restrictions.lt	<
Restrictions.le	< =
Restrictions.between	BETWEEN
Restrictions.like	LIKE
Restrictions.in	in
Restrictions.and	and
Restrictions.or	or
Restrictions.sqlRestriction	用SQL限定查询

```
=====
QBE (QueryBy Example)
Criteria cri = session.createCriteria(Student.class);
cri.add(Example.create(s)); //s是一个Student对象
list cri.list();
实质：创建一个模版，比如我有一个表serial有一个 giftortoy字段，我设置serial.setgifttoy("2"),
则这个表中的所有的giftortoy为2的数据都会出来
```

2: QBC (Query By Criteria) 主要有Criteria,Criterion,Oder,Restrictions类组成

```
session = this.getSession();
Criteria cri = session.createCriteria(JdItemSerialnumber.class);
Criterion cron = Restrictions.like("customer",name);
cri.add(cron);
```

```
list = cri.list();
=====
```

Hibernate中 Restrictions.or()和Restrictions.disjunction()的区别是什么？

比较运算符

HQL运算符	QBC运算符	含义
=	Restrictions.eq()	等于
<>	Restrictions.not(Exprission.eq())	不等于
>	Restrictions.gt()	大于
>=	Restrictions.ge()	大于等于
<	Restrictions.lt()	小于
<=	Restrictions.le()	小于等于
is null	Restrictions.isNull()	等于空值
is not null	Restrictions.isNotNull()	非空值
like	Restrictions.like()	字符串模式匹配
and	Restrictions.and()	逻辑与
and	Restrictions.conjunction()	逻辑与
or	Restrictions.or()	逻辑或
or	Restrictions.disjunction()	逻辑或
not	Restrictions.not()	逻辑非
in(列表)	Restrictions.in()	等于列表中的某一个值
ont in(列表)	Restrictions.not(Restrictions.in())	不等于列表中任意一个值
between x and y	Restrictions.between()	闭区间xy中的任意值
not between x and y	Restrictions.not(Restrictions..between())	小于值X或者大于值y

快速回复

我要收藏

返回顶部

3: HQL

```
String hql = "select s.name ,avg(s.age) from Student s group bys.name";
Query query = session.createQuery(hql);
list = query.list();
....
```

4: 本地SQL查询

```
session = sessionFactory.openSession();
tran = session.beginTransaction();
SQLQuery sq = session.createSQLQuery(sql);
sq.addEntity(Student.class);
list = sq.list();
tran.commit();
```

5: QID

Session的get()和load()方法提供了根据对象ID来检索对象的方式。该方式被用于事先知道了要检索对象ID的情况。

三、 工具类Order提供设置排序方式

```
Order.asc(String propertyName)
升序排序
Order.desc(String propertyName)
降序排序
```

四、 工具类Projections提供对查询结果进行统计与分组操作

```
Porjections.avg(String propertyName)
求某属性的平均值
Projections.count(String propertyName)
统计某属性的数量
Projections.countDistinct(String propertyName)
统计某属性的不同值的数量
```



```
Projections.groupProperty(String propertyName)
```

指定一组属性值

```
Projections.max(String propertyName)
```

某属性的最大值

```
Projections.min(String propertyName)
```

某属性的最小值

```
Projections.projectionList()
```

创建一个新的projectionList对象

```
Projections.rowCount()
```

查询结果集中记录的条数

```
Projections.sum(String propertyName)
```

返回某属性值的合计

五、QBC分页查询

Criteria为我们提供了两个有用的方法：setFirstResult(int firstResult)和setMaxResults(int maxResults)。

setFirstResult(int firstResult)方法用于指定从哪一个对象开始检索（序号从0开始），默认为第一个对象（序号为0）；setMaxResults(int maxResults)方法用于指定一次最多检索出的对象数目，默认为所有对象。

```
Session session = HibernateSessionFactory.getSessionFactory().openSession();
```

```
Transaction ts = null;
```

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

```
int pageSize = 15;
```

```
int pageNo = 1;
```

```
criteria.setFirstResult((pageNo-1)*pageSize);
```

```
criteria.setMaxResults(pageSize);
```

```
Iterator it = criteria.list().iterator();
```

```
ts.commit();
```

```
HibernateSessionFactory.closeSession();
```

六、QBC复合查询

复合查询就是在原有的查询基础上再进行查询。例如在顾客对定单的一对多关系中，在查询出所有的顾客对象后，希望在查询定单中money大于1000的定单对象。

```
Session session = HibernateSessionFactory.getSessionFactory().openSession();
```

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

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

```
Criteria ordCriteria = cusCriteria.createCriteria("orders");
```

```
ordCriteria.add(Restrictions.gt("money", new Double(1000)));
```

```
Iterator it = cusCriteria.list().iterator();
```

```
ts.commit();
```

```
HibernateSessionFactory.closeSession();
```

七、QBC离线查询

离线查询又叫DetachedCriteria查询，它可以在Session之外进行构造，只有在需要执行查询时才与Session绑定。Session session = HibernateSessionFactory.getSessionFactory().openSession();

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

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

```
Criteria ordCriteria = cusCriteria.createCriteria("orders");
```

```
ordCriteria.add(Restrictions.gt("money", new Double(1000)));
```

```
Iterator it = cusCriteria.list().iterator();
```

```
ts.commit();
```

```
HibernateSessionFactory.closeSession();
```

第21课 Query By Example(QBE)

QBE查询就是检索与指定样本对象具有相同属性值的对象。因此QBE查询的关键就是样本对象的创建，样本对象中的所有非空属性均将作为查询条件。QBE查询的功能子集，虽然QBE没有QBC功能大，但是有些场合QBE





使用起来更为方便。

工具类Example为Criteria对象指定样本对象作为查询条件

一、实例代码

```
Session session = sf.openSession();
session.beginTransaction();
Topic tExample = new Topic();
tExample.setTitle("T_");
Example e = Example.create(tExample)
    .ignoreCase().enableLike();
Criteria c = session.createCriteria(Topic.class)
    .add(Restrictions.gt("id", 2))
    .add(Restrictions.lt("id", 8))
    .add(e);
for(Object o : c.list()) {
    Topic t = (Topic)o;
    System.out.println(t.getId() + "-" + t.getTitle());
}
session.getTransaction().commit();
session.close();
```

 快速回复

 我要收藏

 返回顶部

```
Session session = HibernateSessionFactory.getSessionFactory().openSession();
Transaction ts = session.beginTransaction();
Customer c = new Customer();
c.setCName("Hibernate");
Criteria criteria = session.createCriteria(Customer.class);
Criteria.add(Example.create(c));
Iterator it = criteria.list().iterator();
ts.commit();
HibernateSessionFactory.closeSession();
```

第22课 Query.list与query.iterate(不太重要)

一、query.iterate查询数据

* query.iterate()方式返回迭代查询

* 会开始发出一条语句：查询所有记录ID语句

* Hibernate: select student0_.id as col_0_0_ from t_student student0_

* 然后有多少条记录，会发出多少条查询语句。

* n + 1问题：n:有n条记录，发出n条查询语句；1：发出一条查询所有记录ID语句。

* 出现n+1的原因：因为iterate(迭代查询)是使用缓存的，

第一次查询数据时发出查询语句加载数据并加入到缓存，以后再查询时hibernate会先到ession缓存(一级缓存)中查看数据是否存在，如果存在则直接取出使用，否则发出查询语句进行查询。

```
session= HibernateUtils.getSession();
tx = session.beginTransaction();

/**
 * 出现N+1问题
 * 发出查询id列表的sql语句
 * Hibernate: select student0_.id as col_0_0_ from t_student student0_
 *
 * 再依次发出根据id查询Student对象的sql语句
 * Hibernate: select student0_.id as id1_0_, student0_.name as name1_0_,
 * student0_.createTime as createTime1_0_, student0_.classesid as classesid1_0_
 * from t_student student0_ where student0_.id=?
 */

Iterator students = session.createQuery("from Student").iterate();
```

关闭

```

        while (students.hasNext()){
            Student student =(Student)students.next();
            System.out.println(student.getName());
        }
tx.commit();

```

二、 query.list()和query.iterate()的区别

先执行query.list(), 再执行query.iterate, 这样不会出现N+1问题,

- * 因为list操作已经将Student对象放到了一级缓存中, 所以再次使用iterate操作的时候
- * 它首先发出一条查询id列表的sql, 再根据id到缓存中取数据, 只有在缓存中找不到相应的
- * 数据时, 才会发出sql到数据库中查询

```
Liststudents = session.createQuery("from Student").list();
```

```

for (Iterator iter = students.iterator();iter.hasNext();){
    Student student =(Student)iter.next();
    System.out.println(student.getName());
}
System.out.println("-----");
// 不会出现N+1问题, 因为list操作已经将数据加入到一级缓存。
Iterator iters =session.createQuery("from Student").iterate();

while (iters.hasNext()){
    Student student =(Student)iters.next();
    System.out.println(student.getName());
}

```

[快速回复](#)
[我要收藏](#)
[返回顶部](#)

三、 两次query.list()

- * 会再次发出查询sql
- * 在默认情况下list每次都会向数据库发出查询对象的sql, 除非配置了查询缓存
- * 所以: 虽然list操作已经将数据放到一级缓存, 但list默认情况下不会利用缓存, 而再次发出sql
- * 默认情况下, list会向缓存中放入数据, 但不会使用数据。

```
Liststudents = session.createQuery("from Student").list();
```

```

for (Iterator iter = students.iterator();iter.hasNext();){
    Student student =(Student)iter.next();
    System.out.println(student.getName());
}

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

//会再次发现SQL语句进行查询, 因为默认情况list只向缓存中放入数据, 不会使用缓存中数据
students = session.createQuery("fromStudent").list();

for (Iterator iter = students.iterator();iter.hasNext();){
    Student student =(Student)iter.next();
    System.out.println(student.getName());
}

```

第23课 性能优化策略

- 1、 注意session.clear()的动用, 尤其在不断分页循环的时候
 - a) 在一个大集合中进行遍历, 遍历msg.取出其中的含有敏感字样的对象
 - b) 另外一种形式的内存泄露 //面试题是: Java有内存泄漏吗?
- 2、 1 + N问题 //典型的面试题
 - a) Lazy

[关闭](#)

- b) BatchSize 设置在实体类的前面
- c) joinfetch
- 3、list 和 iterate不同之处
- a) list取所有
- b) Iterate先取ID，等用到的时候再根据ID来取对象
- c) session中list第二次发出，仍会到数据库查询
- d) iterate第二次，首先找session级缓存

第24课 hibernate缓存

一、Session级缓存(一级缓存)

一级缓存很短和session的生命周期一致，因此也叫session级缓存或事务级缓存

hibernate一级缓存

那些方法支持一级缓存：

- * get()
- * load()
- * iterate (查询实体对象)

如何管理一级缓存：

- *session.clear(),session.evict()

如何避免一次性大量的实体数据入库导致内存溢出

- * 先flush，再clear

如果数据量特别大，考虑采用jdbc实现，如果jdbc也不能满足要求可以考虑采用数据本身的特定导入工具

二、二级缓存

Hibernate默认的二级缓存是开启的。

二级缓存也称为进程级的缓存，也可称为SessionFactory级的缓存(因为SessionFactory可以管理二级缓存)，它与session级缓存不一样，一级缓存只要session关闭缓存就不存在了。而二级缓存则只要进程在二级缓存就可用。

二级缓存可以被所有的session共享

二级缓存的生命周期和SessionFactory的生命周期一样，SessionFactory可以管理二级缓存

二级缓存同session级缓存一样，只缓存实体对象，普通属性的查询不会缓存

二级缓存一般使用第三方的产品，如EHCache

1、二级缓存的配置和使用：

配置二级缓存的配置文件：模板文件位于hibernate\etc目录下(如ehcache.xml)，将模板存放在ClassPath目录中，一般放在根目录下(src目录下)

<ehcache>

<!-- 设置当缓存对象溢出时，对象保存到磁盘时的保存路径。

如 d:\xxxx

The following properties are translated:

user.home - User's home directory

user.dir - User's current working directory

java.io.tmpdir - windows的临时目录 -->

<diskStore path="java.io.tmpdir"/>

<!-- 默认配置/或对某一个类进行管理

maxInMemory - 缓存中可以存入的最多个对象数

eternal - true:表示永不失效，false：不是永久有效的。

timeToIdleSeconds - 空闲时间，当第一次访问后在空闲时间内没有访问，则对象失效，单位为秒

timeToLiveSeconds - 被缓存的对象有效的生命时间，单位为秒

overflowToDisk 当缓存中对象数超过核定数(溢出时)时，对象是否保存到磁盘上。true:保存；false：不保存



如果保存，则保存路径在标签<diskStore>中属性path指定

```
-->
<defaultCache
  maxElementsInMemory="10000"
  eternal="false"
  timeToIdleSeconds="120"
  timeToLiveSeconds="120"
  overflowToDisk="true"
/>
</ehcache>
```

2、 二级缓存的开启：

Hibernate中二级缓存默认就是开启的，也可以显示的开启

二级缓存是hibernate的配置文件设置如下：

<!--开启二级缓存，hibernate默认的二级缓存就是开启的 -->

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

快速回复

我要收藏

返回顶部

3、 指定二级缓存产品提供商：

修改hibernate的 配置文件，指定二级缓存提供商，如下：

<!--指定二级缓存提供商-->

```
<property name="hibernate.cache.provider_class">
org.hibernate.cache.EhCacheProvider
</property>
```

以下为常见缓存提供商：

Cache	Provider class	Type	Cluster Safe	Query Cache Supported
Hashtable (not intended for production use)	org.hibernate.cache.HashtableCacheProvider	memory		yes
EHCache	org.hibernate.cache.EhCacheProvider	memory, disk		yes
OSCache	org.hibernate.cache.OSCacheProvider	memory, disk		yes
SwarmCache	org.hibernate.cache.SwarmCacheProvider	clustered (ip multicast)	yes (clustered invalidation)	
JBoss TreeCache	org.hibernate.cache.TreeCacheProvider	clustered (ip multicast), transactional	yes (replication)	yes (clock sync req.)

4、 使用二级缓存

a) xml方式：指定哪些实体类使用二级缓存：

方法一：在实体类映射文件中，使用<cache>来指定那个实体类使用二级缓存，如下：

```
<cache
  usage="transactional|read-write|nonstrict-read-write|read-only" (1)

  region="RegionName"
  (2)

  include="all|non-lazy"
  (3)
/>
```

(1) usage(必须)说明了缓存的策略: transactional、 read-write、 nonstrict-read-write或 read-only。

(2) `region` (可选, 默认为类或者集合的名字(class or collection role name)) 指定第二级缓存的区域名(name of the second level cache region)

(3) `include` (可选, 默认为 `all`) `non-lazy` 当属性级延迟抓取打开时, 标记为 `lazy="true"` 的实体的属性可能无法被缓存

另外(首选?), 你可以在 `hibernate.cfg.xml` 中指定 `<class-cache>` 和 `<collection-cache>` 元素。

这里的 `usage` 属性指明了缓存并发策略 (*cache concurrency strategy*)。

策略: 只读缓存 (*Strategy: read only*)

如果你的应用程序只需读取一个持久化类的实例, 而无需对其修改, 那么就可以对其进行只读缓存。这是最简单, 也是实用性最好的方法。甚至在集群中, 它也能完美地运作。

```
<class name="eg.Immutable" mutable="false">

    <cache usage="read-only"/>

    ....

</class>
```

策略: 读/写缓存 (*Strategy: read/write*)

如果应用程序需要更新数据, 那么使用读/写缓存比较合适。如果应用程序要求“序列化事务”的隔离 (*serializable transaction isolation level*), 那么就决不能使用这种缓存策略。如果在 JTA 环境中使用缓存, 你必须指定 `hibernate.transaction.manager_lookup_class` 属性的值, 通过它, Hibernate 才能知道 JTA 的 `TransactionManager` 的具体策略。在其它环境中, 你必须保证在 `Session.close()`、或 `Session.disconnect()` 调用前, 整个事务已经结束。如果你想在集群环境中使用此策略, 你必须保证底层的缓存实现支持锁定 (locking)。Hibernate 内置的缓存策略并不支持锁定功能。

```
<class name="eg.Cat" .... >

    <cache usage="read-write"/>

    ....

    <set name="kittens" ... >

        <cache usage="read-write"/>

        ....

    </set>

</class>
```

策略: 非严格读/写缓存 (*Strategy: nonstrictread/write*)

如果应用程序只偶尔需要更新数据 (也就是说, 两个事务同时更新同一记录的情况很不常见), 也不需要十分严格的事务隔离, 那么比较适合使用非严格读/写缓存策略。如果在 JTA 环境中使用该策略, 你必须为其指定 `hibernate.transaction.manager_lookup_class` 属性的值, 在其它环境中, 你必须保证在 `Session.close()`、或 `Session.disconnect()` 调用前, 整个事务已经结束。

策略: 事务缓存 (*transactional*)

Hibernate 的事务缓存策略提供了全事务的缓存支持, 例如对 JBoss TreeCache 的支持。这样的缓存只能用于 JTA 环境中, 你必须指定为其 `hibernate.transaction.manager_lookup_class` 属性。

没有一种缓存提供商能够支持上列的所有缓存并发策略。下表中列出了各种提供者、及其各自适用的并发策略。

表 19.2. 各种缓存提供商对缓存并发策略的支持情况 (*Cache Concurrency Strategy Support*)

Cache	read-only	nonstrict-read-write	read-write	transactional
Hashtable (not intended for production use)	yes	yes	yes	
EHCache	yes	yes	yes	
OSCache	yes	yes	yes	
SwarmCache	yes	yes		
JBoss TreeCache	yes			yes

注: 此方法要求: 必须要标签 `<cache>` 放在 `<id>` 标签之前

```
-
<class name="com.wjt276.hibernate.Student"table="t_student">
```

```
<!-- 指定实体类使用二级缓存 -->
```

```
<cache usage="read-only"/>
```

1、 清除指定实体类的所有数据

```
SessionFactory.evict(Student.class);
```

2、 清除指定实体类的指定对象

SessionFactory.evict(Student.class, 1);//第二个参数是指定对象的ID，就可以清除指定ID的对象

使用SessionFactory清除二级缓存

```
Session session = null;

try {
    session = HibernateUtils.getSession();
    session.beginTransaction();

    Student student = (Student)session.load(Student.class, 1);
    System.out.println("student.name=" + student.getName());

    session.getTransaction().commit();
} catch (Exception e) {
    e.printStackTrace();
    session.getTransaction().rollback();
} finally {
    HibernateUtils.closeSession(session);
}
```

//管理二级缓存

```
SessionFactory factory = HibernateUtils.getSessionFactory();
//factory.evict(Student.class);
factory.evict(Student.class, 1);
```

```
try {
    session = HibernateUtils.getSession();
    session.beginTransaction();

    //会发出查询sql，因为二级缓存中的数据被清除了
    Student student = (Student)session.load(Student.class, 1);
    System.out.println("student.name=" + student.getName());

    session.getTransaction().commit();
} catch (Exception e) {
    e.printStackTrace();
    session.getTransaction().rollback();
} finally {
    HibernateUtils.closeSession(session);
}
```

7、 二级缓存的交互

```
Session session = null;

try {
    session = HibernateUtils.getSession();
    session.beginTransaction();

    //仅向二级缓存读数据，而不向二级缓存写数据
    session.setCacheMode(CacheMode.GET);
    Student student = (Student)session.load(Student.class, 1);
    System.out.println("student.name=" + student.getName());

    session.getTransaction().commit();
}
```

[快速回复](#)[我要收藏](#)[返回顶部](#)


```

}catch(Exception e) {
    e.printStackTrace();
    session.getTransaction().rollback();
}finally {
    HibernateUtils.closeSession(session);
}

try {
    session = HibernateUtils.getSession();
    session.beginTransaction();

    //发出sql语句，因为session设置了CacheMode为GET，所以二级缓存中没有数据
    Student student =(Student)session.load(Student.class, 1);
    System.out.println("student.name=" +student.getName());

    session.getTransaction().commit();
}catch(Exception e) {
    e.printStackTrace();
    session.getTransaction().rollback();
}finally {
    HibernateUtils.closeSession(session);
}

try {
    session = HibernateUtils.getSession();
    session.beginTransaction();

    //只向二级缓存写数据，而不从二级缓存读数据
    session.setCacheMode(CacheMode.PUT);

    //会发出查询sql，因为session将CacheMode设置成了PUT
    Student student =(Student)session.load(Student.class, 1);
    System.out.println("student.name=" +student.getName());

    session.getTransaction().commit();
}catch(Exception e) {
    e.printStackTrace();
    session.getTransaction().rollback();
}finally {
    HibernateUtils.closeSession(session);
}

```

CacheMode参数用于控制具体的Session如何与二级缓存进行交互。

- CacheMode.NORMAL - 从二级缓存中读、写数据。
 - CacheMode.GET - 从二级缓存中读取数据，仅在数据更新时对二级缓存写数据。
 - CacheMode.PUT - 仅向二级缓存写数据，但不从二级缓存中读数据。
 - CacheMode.REFRESH - 仅向二级缓存写数据，但不从二级缓存中读数据。通过hibernate.cache.use_minimal_puts的设置，强制二级缓存从数据库中读取数据，刷新缓存内容。
- 如若需要查看二级缓存或查询缓存区域的内容，你可以使用统计（Statistics）API。

```

Map cacheEntries = sessionFactory.getStatistics()

    .getSecondLevelCacheStatistics(regionName)

    .getEntries();

```

此时，你必须手工打开统计选项。可选的，你可以让Hibernate更人工可读的方式维护缓存内容。



```
hibernate.generate_statistics true
```

```
hibernate.cache.use_structured_entries true
```

8、 总结

load默认使用二级缓存，iterate默认使用二级缓存

list默认向二级缓存中加数据，但是查询时候不使用

三、 查询缓存

查询缓存，是用于缓存普通属性查询的，当查询实体时缓存实体ID。

默认情况下关闭，需要打开。查询缓存，对list/iterator这样的操作会起作用。

可以使用<property name="hibernate.cache.use_query_cache">true</property>来打开查询缓存，默认为关闭。

所谓查询缓存：即让hibernate缓存list、iterator、createQuery等方法的查询结果集。如果没有打开查询缓存，hibernate将只缓存load方法获得的单个持久化对象。

在打开了查询缓存之后，需要注意，调用query.list()操作之前，必须显式调用query.setCacheable()来标识某个查询使用缓存。

查询缓存的生命周期：当前关联的表发生修改，那么查询缓存生命周期结束

注意查询缓存依赖于二级缓存，因为使用查询缓存需要打开二级缓存

查询缓存的配置和使用：

* 在hibernate.cfg.xml文件中启用查询缓存，如：

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

* 在程序中必须手动启用查询缓存，如：

```
query.setCacheable(true);
```

例如：

```
session= HibernateUtils.getSession();
session.beginTransaction();
Query query = session.createQuery("selects.name from Student s");
//启用查询缓存
query.setCacheable(true);

List names = query.list();
for (Iterator iter=names.iterator();iter.hasNext();) {
    String name =(String)iter.next();
    System.out.println(name);
}
System.out.println("-----");
query = session.createQuery("selects.name from Student s");
//启用查询缓存
query.setCacheable(true);

//没有发出查询sql，因为启用了查询缓存
names = query.list();
for (Iterator iter=names.iterator();iter.hasNext();) {
    String name =(String)iter.next();
    System.out.println(name);
}

session.getTransaction().commit();
```

 快速回复

 我要收藏

 返回顶部

```
Session session = sf.openSession();
    session.beginTransaction();
    List<Category>categories = (List<Category>)session.createQuery("from Category")
        .setCacheable(true).list();
    session.getTransaction().commit();
    session.close();

    Session session2 = sf.openSession();
    session2.beginTransaction();
    List<Category>categories2 = (List<Category>)session2.createQuery("from Category")
        .setCacheable(true).list();

    session2.getTransaction().commit();

    session2.close();
```

快速回复

我要收藏

返回顶部

注：查询缓存的生命周期与session无关。

查询缓存只对query.list()起作用，query.iterate不起作用，也就是query.iterate不使用

四、缓存算法

- 1、 LRU、LFU、FIFO

第25课 事务并发处理

一、数据库的隔离级别：并发性作用。

- 1、 ReadUncommitted(未提交读):没有提交就可以读取到数据（发出了Insert，但没有commit就可以读取到。）很少用
- 2、 ReadCommitted(提交读):只有提交后可以读，常用，
- 3、 RepeatableRead(可重复读):mysql默认级别, 必需提交才能见到，读取数据时数据被锁住。
- 4、 Serialiazble(序列化读):最高隔离级别，串型的，你操作完了，我才可以操作，并发性特别不好，

隔离级别	是否存在脏读	是否存在不可重复读	是否存在幻读
Read Uncommitted(未提交读)	Y	Y	Y
Read Committed(提交读)	N	Y(可采用悲观锁解决)	Y
Repeatable Read(可重复读)	N	N	Y
Serialiazble(序列化读)			

脏读：没有提交就可以读取到数据称为脏读

不可重复读：再重复读一次，数据与你上的不一样。称不可重复读。

幻读：在查询某一条件的数据，开始查询的后，别人又加入或删除些数据，再读取时与原来的数据不一样了。

1、 Mysql查看数据库隔离级别：

```
方法：select @@tx_isolation;

mysql> select @@tx_isolation;
+-----+
| @@tx_isolation |
+-----+
| REPEATABLE-READ |
+-----+
1 row in set <0.00 sec>
```

2、 Mysql数据库修改隔离级别：

方法：set transaction isolation level 隔离级别名称;

例如：修改为未提交读：set transaction isolation level read uncommitted;

```
mysql> set transaction isolation level read uncommitted;
Query OK, 0 rows affected (0.00 sec)

mysql> select @@tx_isolation;
+-----+
| @@tx_isolation |
+-----+
| READ-UNCOMMITTED |
+-----+
1 row in set (0.00 sec)
```

二、事务概念(ACID)

ACID即：事务的原子性、一致性、独立性及持久性

事务的原子性:是指一个事务要么全部执行,要么不执行.也就是说一个事务不可能只执行了一半就停止了.比如你从取款机取钱,这个事务可以分成两个步骤:1划卡,2出钱.不可能划了卡,而钱却没出来.这两步必须同时完成.要么就不完成.

事务的一致性:是指事务的运行并不改变数据库中数据的一致性.例如,完整性约束了a+b=10,一个事务改变了a,那么b也应该随之改变.

事务的独立性:是指两个以上的事务不会出现交错执行的状态.因为这样可能会导致数据不一致.

事务的持久性:是指事务运行成功以后,就系统的更新是永久的.不会无缘无故的回滚.

三、事务并发时可能出现问题

1、第一类丢失更新(Lost Update)

时间		

快速回复

我要收藏

返回顶部

第26课 hibernate悲观锁、乐观锁

Hibernate谈到悲观锁、乐观锁，就要谈到数据库的并发问题，数据库的隔离级别越高它的并发性就越差

并发性：当前系统进行了序列化后，当前读取数据后，别人查询不了，看不了。称为并发性不好

数据库隔离级别：见前面章级

一、悲观锁

悲观锁：具有排他性(我锁住当前数据后，别人看到不此数据)

悲观锁一般由数据机制来做到的。

1、悲观锁的实现

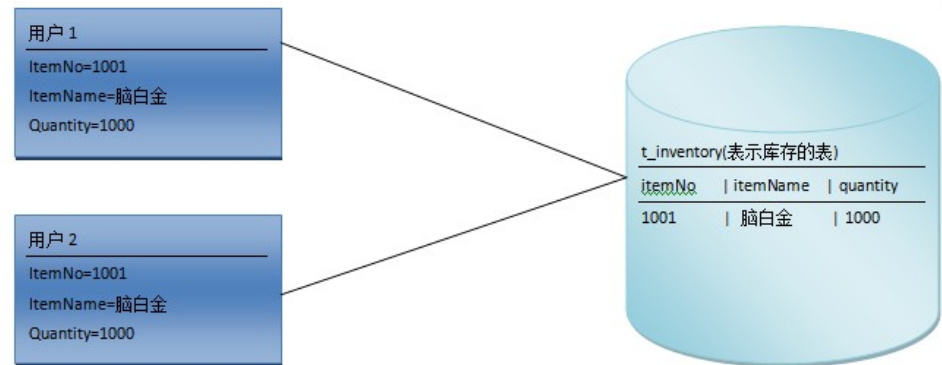
通常依赖于数据库机制，在整修过程中将数据锁定，其它任何用户都不能读取或修改(如：必需我修改完之后，别人才可以修改)

2、悲观锁的运用场景：

悲观锁一般适合短事务比较多(如某一数据取出后加1，立即释放)

长事务占有时间(如果占有1个小时，那么这个1小时别人就不可以使用这些数据)，不常用。

3、实例：



用户1、用户2 同时读取到数据，但是用户2先 - 200，这时数据库里的是800，现在用户1也开始 - 200，可是用户1刚才读取到的数据是1000，现在用户用刚刚开始读取的数据1000 - 200为800，而用户1在更新时数据库里的是用更新的数据800，按理说用户1应该是800 - 200 = 600，而现在是800，这样就造成的更新丢失。这种情况该如何处理呢，可采用两种方法：悲观锁、乐观锁。先看看悲观锁：用户1读取数据后，用锁将其

读取的数据锁上，这时用户2是读取不到数据的，只有用户1释放锁后用户2才可以读取，同样用户2读取数据也锁上。这样就可以解决更新丢失的问题了。

实体类：

```
public class Inventory {
    private int itemNo;
    private String itemName;
    private int quantity;
    public int getItemNo() {
        return itemNo;
    }
    public void setItemNo(int itemNo) {
        this.itemNo = itemNo;
    }
    public String getItemName() {
        return itemName;
    }
    public void setItemName(String itemName) {
        this.itemName = itemName;
    }
    public int getQuantity() {
        return quantity;
    }
    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
}
```

映射文件：

```
<hibernate-mapping>
    <class name="com.wjt276.hibernate.Inventory" table="t_inventory">
        <id name="itemNo">
            <generator class="native"/>
        </id>
        <property name="itemName"/>
        <property name="quantity"/>
    </class>
</hibernate-mapping>
```

4、悲观锁的使用

如果需要使用悲观锁，肯定在加载数据时就要锁住，通常采用数据库的for update语句。

Hibernate使用Load进行悲观锁加载。

Session.load(Class arg0, Serializable arg1, LockMode arg2) throws HibernateException

LockMode：悲观锁模式（一般使用LockMode.UPGRADE）

```
session = HibernateUtils.getSession();
tx = session.beginTransaction();
Inventory inv = (Inventory) session.load(Inventory.class, 1, LockMode.UPGRADE);
System.out.println(inv.getItemName());
inv.setQuantity(inv.getQuantity()-200);

session.update(inv);
tx.commit();
```

5、执行输出SQL语句：

Hibernate:select inventory0_.itemNo as itemNo0_0_, inventory0_.itemName as itemName0_0_, inventory0_.quantity as quantity0_0_ from t_inventory inventory0_ where inventory0_.itemNo=? for update //在select语句中加入for update进行使用悲观锁。

脑白金



Hibernate:update t_inventory set itemName=?, quantity=? where itemNo=?

注：只有用户释放锁后，别的用户才可以读取

注：如果使用悲观锁，那么lazy(懒加载无效)

二、乐观锁

乐观锁：不是锁，是一种冲突检测机制。

乐观锁的并发性较好，因为我改的时候，别人随边修改。

乐观锁的实现方式：常用的是版本的方式（每个数据表中有一个版本字段version，某一个用户更新数据后，版本号+1，另一个用户修改后再+1，当用户更新发现数据库当前版本号与读取数据时版本号不一致(等于小于数据库当前版本号)，则更新不了。


Hibernate使用乐观锁需要在映射文件中配置项才可生效。


实体类：

```
public class Inventory {
    private int itemNo;
    private String itemName;
    private int quantity;
    private int version; //Hibernate用户实现版本方式乐观锁，但需要在映射文件中配置
    public int getItemNo() {
        return itemNo;
    }
    public void setItemNo(int itemNo) {
        this.itemNo = itemNo;
    }
    public String getItemName() {
        return itemName;
    }
    public void setItemName(String itemName) {
        this.itemName = itemName;
    }
    public int getQuantity() {
        return quantity;
    }
    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
    public int getVersion() {
        return version;
    }
    public void setVersion(int version) {
        this.version = version;
    }
}
```

映射文件

```
<hibernate-mapping>
<!-- 映射实体类时，需要加入一个开启乐观锁的属性
optimistic-lock="version" 共有好几种方式：
    - none -version - dirty - all
同时需要在主键映射后面映射版本号字段
-->
<class name="com.wjt276.hibernate.Inventory" table="t_inventory" optimistic-lock="version">
    <id name="itemNo">
        <generator class="native"/>
    </id>
```

 快速回复

 我要收藏

 返回顶部

```
<version name="version"/><!--必需配置在主键映射后面 -->
<property name="itemName"/>
<property name="quantity"/>
</class>
</hibernate-mapping>
```

导出输出SQL语句：

createtable t_inventory (itemNo integer not null auto_increment, versioninteger not null, itemName
varchar(255), quantity integer,primary key (itemNo))

注：添加的版本字段version，还是我们来维护的，是由hibernate来维护的。

乐观锁在存储数据时不用关心



顶3

踩0

- 快速回复
- 我要收藏
- 返回顶部

- 上一篇 struts2.1.6 interceptor 09
- 下一篇 hibernate 1

我的同类文章

hibernate (6)			
• Hibernate的merge()方法	2012-07-23 阅读 835	• jta 事物	2011-12-22 阅读 571
• hibernate 4	2011-11-26 阅读 1706	• hibernate 3	2011-11-24 阅读 906
• hibernate 2	2011-11-22 阅读 1096	• hibernate 1	2011-11-18 阅读 2210

参考知识库

MySQL知识库
22207 关注 | 1449 收录

Oracle知识库
4984 关注 | 252 收录

软件测试知识库
4611 关注 | 318 收录

Java SE知识库
26006 关注 | 479 收录

Java EE知识库
18039 关注 | 1324 收录

Java 知识库
26347 关注 | 1457 收录

算法与数据结构知识库
16043 关注 | 2320 收录

猜你在找

- hibernate4_基于xml和annotation两种配置方式
- 尚学堂马士兵Oracle学习笔记之二select子句和常用
- SQLServer数据库高级实战视频课程
- 马士兵 struts 学习笔记
- 企业级Oracle数据库实战开发应用视频课程
- Struts2 学习笔记马士兵教程 struts216版本 第五天
- Android核心技术——Android数据存储
- 马士兵J2SE学习笔记第七章 容器
- 360度解析亚马逊AWS数据存储服务
- hibernate马士兵笔记

关闭

查看评论

15楼 叮叮猫儿 2015-01-26 13:54发表



楼主，你这么叼，你家人知道吗

14楼 guanghui9097 2014-12-30 10:42发表



值得看~~

13楼 点点滴滴_IT_man 2013-10-21 16:05发表



很好很强大

12楼 Cary_诚 2013-06-25 16:19发表



谢谢楼主整理，马老师上课的要点都有了。看完视频，以后复习很有用。

快速回复

我要收藏

返回顶部

11楼 笈姓 2013-06-22 18:10发表



马士兵的Hibernate内容

10楼 luo_8073 2013-05-20 16:33发表



很喜欢。。。

9楼 Java_远 2013-04-22 13:57发表



很棒啊，最近在学这个

8楼 傲雪kimi 2013-01-16 14:40发表



不得不支持一下，笔记是好东西

7楼 wenhuaweiping 2012-11-16 13:13发表



很好，很喜欢

6楼 omi8888 2012-08-07 21:58发表



怎么不能转载。。太棒了

5楼 xwqfudimo 2012-08-04 14:05发表



好同学啊

4楼 xiazhiwuyu321 2012-07-12 11:17发表



很好很强大

3楼 eyeooo 2012-07-12 11:16发表



这篇文章真的很逆天！佩服！

2楼 mcgre 2012-02-28 14:10发表



很好很强大

1楼 三月神 2012-02-24 21:35发表



很好。。。。。。。。。。


您还没有登录,请[登录](#)或[注册](#)

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题	Hadoop	AWS	移动游戏	Java	Android	iOS	Swift	智能硬件	Docker			
OpenStack	VPN	Spark	ERP	IE10	Eclipse	CRM	JavaScript	数据库	Ubuntu	NFC		
WAP	jQuery	BI	HTML5	Spring	Apache	.NET	API	HTML	SDK	IIS	Fedora	XML
LBS	Unity	Splashtop	UML	components	Windows Mobile	Rails	QEMU	KDE	Cassandra			
CloudStack	FTC	coremail	OPhone	CouchBase	云计算	iOS6	Rackspace	Web App				
SpringSide	Maemo	Compuware	大数据	aptech	Perl	Tornado	Ruby	Hibernate	ThinkPHP			
HBase	Pure	Solr	Angular	Cloud Foundry	Redis	Scala	Django	Bootstrap				

公司简介 | 招贤纳士 | 广告服务 | 联系方式 | 版权声明 | 法律顾问 | 问题报告 | 合作伙伴 | 论坛反馈

 网站客服  杂志客服  微博客服  webmaster@csdn.net  400-600-2320 | 北京创新乐知信息技术有限公司 版权所有 | 江苏知之为计算机有限公司 |

江苏乐知网络技术有限公司

京 ICP 证 09002463 号 | Copyright © 1999-2016, CSDN.NET, All Rights Reserved 

-  快速回复
-  我要收藏
-  返回顶部