Hibernate笔记

声明实体@Entity

@Entity

声明主键@Id

@Id

定义主键生成策略@GeneratedValue

@GeneratedValue(generator **=** "generator")  
@GenericGenerator(name **=** "generator", strategy **=** "increment")

@Id 映射主键属性  
 @GeneratedValue —— 注解声明了主键的生成策略。该注解有如下属性  
 strategy 指定生成的策略,默认是GenerationType. AUTO  
 GenerationType.*AUTO* 主键由程序控制  
 GenerationType.*TABLE* 使用一个特定的数据库表格来保存主键  
 GenerationType.*IDENTITY* 主键由数据库自动生成,主要是自动增长类型  
 GenerationType.*SEQUENCE* 根据底层数据库的序列来生成主键，条件是数据库支持序列  
 generator 指定生成主键使用的生成器

@GeneratedValue(strategy **=** GenerationType.*AUTO*)

指定表名@Table

@Table(name **=** "user")

映射大对象@Lob（属性将被持久化为Blob或者Clob类型, 具体取决于属性的类型, java.sql.Clob, Character[], char[] 和 java.lang.String这些类型的属性都被持久化为Clob类型, 而java.sql.Blob, Byte[], byte[] 和 serializable类型则被持久化为Blob类型。 Mysql 不支持标准 SQL 的 CLOB 类型, 在 Mysql 中, 用 TEXT, MEDIUMTEXT 及 LONGTEXT 类型来表示长度操作 255 的长文本数据）

@Lob

注解可将属性映射到列@Column

@Column(length **=** 32,nullable **=** false)

嵌入式对象(组件)@Embedded（

组件类必须在类一级定义@Embeddable注解，在特定的实体的关联属性上使用@Embedded和 @AttributeOverride注解可以覆盖该属性对应的嵌入式对象的列映射

）

@Embedded  
@AttributeOverrides({  
 @AttributeOverride(name **=** "province",column **=** @Column(name **=** "address\_province")),  
 @AttributeOverride(name**=**"city",column **=** @Column(name **=** "address\_city")),  
 @AttributeOverride(name **=** "region",column**=**@Column(name **=** "address\_region"))  
})  
private Address address;

被嵌入的实体@Embeddable

@Embeddable  
public class Address {  
 private String province;  
 private String city;  
 private String region;

}

一对一@OneToOne

**\*  
 \*** @OneToOne：一对一关联  
 **\*** cascade：级联,它可以有有五个值可选,分别是：  
 **\*** CascadeType.*PERSIST*：级联新建  
 **\*** CascadeType.*REMOVE* **:** 级联删除  
 **\*** CascadeType.*REFRESH*：级联刷新  
 **\*** CascadeType.*MERGE* ： 级联更新  
 **\*** CascadeType.*ALL* ： 以上全部四项  
 **\*** @JoinColumn**:**主表外键字段

实例{

student和studentCard是一对一关系，studentCard所对应表中的外键是cardId。 mappedBy标签一定是定义在the owned side(被拥有方的)，他指向the owning side(拥有方)。student里面card是注释是optional=true,也就是说一个学生是可以没有学生证的,但是一个学生证不可以没有人,所以在card里面注释student的时候,optional就为false了,这样就可以防止一个空的身份证记录进数据库。

@PrimaryKeyJoinColumn:共享主键方式

@JoinColumn：外键方式

@JoinTable：通过关联表实现

}

@Id  
@GeneratedValue(generator **=** "generator")  
@GenericGenerator(name **=** "generator", strategy **=** "increment")  
private Integer cardId;

@OneToOne(cascade**=**CascadeType.*ALL*,mappedBy**=**"card",optional**=**false)  
private Student student;

@OneToOne(cascade **=** CascadeType.*ALL*,optional**=**true)  
@JoinColumn(name **=** "cardId")//@PrimaryKeyJoinColumn指定从表的主键列  
private StudentCard card;

一对多@ OneToMany

@OneToMany(mappedBy **=** "classRoom")  
@LazyCollection(LazyCollectionOption.*EXTRA*)//LazyCollection属性设置成EXTRA指定了当如果查询数据的个数时候，只会发出一条 count(\*)的语句，提高性能  
private Set**<**Student**>** students;

多对一@ManyToOne

@ManyToOne(fetch**=**FetchType.*LAZY*)//ManyToOne指定了多对一的关系，fetch=FetchType.LAZY属性表示在多的那一方通过延迟加载的方式加载对象  
@JoinColumn(name**=**"classRoomId")  
private ClassRoom classRoom;

通过hibernate来进行插入操作的时候，不管是一对多、一对一还是多对多，都只需要记住一点，在哪个实体类声明了外键，就由哪个类来维护关系，在保存数据时，总是先保存的是没有维护关联关系的那一方的数据，后保存维护了关联关系的那一方的数据

多对多@ManyToMany 中间表

@ManyToMany(mappedBy**=**"teachers")//表示由Course那一方来进行维护  
private Set**<**Course**>** courses;

@ManyToMany//ManyToMany指定多对多的关联关系  
@JoinTable(name**=**"teacher\_course", joinColumns**=**{ @JoinColumn(name**=**"courseId")},  
 inverseJoinColumns**=**{ @JoinColumn(name **=** "teacherId") })

//因为多对多之间会通过一张中间表来维护两表直接的关系，  
// 所以通过 JoinTable 这个注解来声明，name就是指定了中间表的名字，JoinColumns是一个 @JoinColumn类型的数组，  
// 表示的是我这方在对方中的外键名称，我方是Course，所以在对方外键的名称就是 rid，inverseJoinColumns也是一个 @JoinColumn类型的数组，]  
// 表示的是对方在我这方中的外键名称，对方是Teacher，所以在我方外键的名称就是 teacherid  
private Set**<**Teacher**>** teachers;

@AssociationOverride( name**=**"name", joinColumns **=** @JoinColumn(name**=**"subEntityName") )//在@\*ToOne和@\*ToMany中覆写相关列

映射继承关系

每个类一张表(TPC)

@Inheritance(strategy **=** InheritanceType.*TABLE\_PER\_CLASS*)

每个层次结构一张表(TPH)

父类：

@Inheritance(strategy**=**InheritanceType.*SINGLE\_TABLE*)  
@DiscriminatorColumn(  
 name**=**"区别类型列的值",  
 discriminatorType**=** DiscriminatorType.*STRING*)  
@DiscriminatorValue("区别类型的列名")

子类：

@Entity  
@DiscriminatorValue("区别类型列的值")

独立子类特有的字段(TPT)

@Inheritance(strategy**=** InheritanceType.*JOINED*)

从父类继承父类的属性

父类标注

@MappedSuperclass//标注为@MappedSuperclass的类将不是一个完整的实体类，他将不会映射到数据库表，但是他的属性都将映射到其子类的数据库字段中。  
标注为@MappedSuperclass的类不能再标注@Entity或@Table注解

父类将不会建立一张单独的表。且不能使用session.save(父类)

子类可以覆写父类列名

@AttributeOverride( name**=**"id", column **=** @Column(name**=**"subEntityID") )  
@AssociationOverride( name**=**"name", joinColumns **=** @JoinColumn(name**=**"subEntityName") ) //父类字段有@\*ToOne和@\*ToMany的话，可覆写相关列

启用二级缓存（ehcache）

1. 使用 Hibernate 二级缓存的步骤:

1). 加入二级缓存插件的 jar 包及配置文件:

//ehcache  
"net.sf.ehcache:ehcache-core:2.6.11",  
"org.hibernate:hibernate-ehcache:5.1.0.Final",

2). 配置 hibernate.cfg.xml

I. 配置启用 hibernate 的二级缓存

<property name="cache.use\_second\_level\_cache">true</property>

II. 配置hibernate二级缓存使用的产品

<property name="hibernate.cache.region.factory\_class">org.hibernate.cache.ehcache.EhCacheRegionFactory</property>

III. 配置对哪些类使用 hibernate 的二级缓存

不推荐下面的xml配置，应当使用

@Cache(usage **=** CacheConcurrencyStrategy.*READ\_WRITE*)

标到需要的model上面！

<class-cache usage="read-write" class=" "/>

实际上也可以在 .hbm.xml 文件中配置对哪些类使用二级缓存, 及二级缓存的策略是什么.

2). 集合级别的二级缓存的配置

I. 配置对集合使用二级缓存

<collection-cache usage="read-write" collection=" "/>

也可以在 .hbm.xml 文件中进行配置

<set name="emps" table="table " inverse="true" lazy="true">

<cache usage="read-write"/>

<key>

<column name="DEPT\_ID" />

</key>

<one-to-many class=" " />

</set>

II. 注意: 还需要配置集合中的元素对应的持久化类也使用二级缓存! 否则将会多出 n 条 SQL 语句.

3). ehcache 的 配置文件: ehcache.xml

4). 查询缓存: 默认情况下, 设置的缓存对 HQL 及 QBC 查询时无效的, 但可以通过以下方式使其是有效的

I. 在 hibernate 配置文件中声明开启查询缓存

<property name="cache.use\_query\_cache">true</property>

II. 调用 Query 或 Criteria 的 setCacheable(true) 方法

III. 查询缓存依赖于二级缓存

在criteria接口中提供了一个setResultTransformer(ResultTransformer resultTransformer)，这个ResultTransformer就是结果集转换策略接口，在criteria的父接口CriteriaSpecification中定义了几个ResultTransformer的常用实现。

　　ALIAS\_TO\_ENTITY\_MAP 将结果集封装到Map对象；

　　ROOT\_ENTITY 将结果集封装到根实体对象；

　　DISTINCT\_ROOT\_ENTITY 将结果集封装到不重复的根实体对象；

　　PROJECTION 根据投影的结果类型自动封装；

　　当进行投影查询时，结果的封装策略由ROOT\_ENTITY 变为了PROJECTION， 所以是根据查询数据进行封装，而不是封装到根对象。

使用AliasToBeanResultTransformer 修改结果封装策略，AliasToBeanResultTransformer 会根据返回列自动匹配类中属性名，完成封装。

List result **=** *session*.createCriteria(Student.class)  
 .setProjection(  
 Projections.projectionList()  
 .add(Property.forName("name").as("name"))  
 .add(Property.forName("age").as("age"))  
 )  
 .setResultTransformer(new AliasToBeanResultTransformer(Student.class))  
 .list();

@LazyToOne(LazyToOneOption.*PROXY*)

定义了 @ManyToOne 和 @OneToOne 关联的延迟选项. LazyToOneOption 可以是 PROXY (例如:基于代理的延迟加载), NO\_PROXY (例如:基于字节码增强的延迟加载 - 注意需要在构建期处理字节码) 和 FALSE (非延迟加载的关联)

@LazyCollection(*TRUE*)

定义了 @ManyToMany和 @OneToMany 关联的延迟选项. LazyCollectionOption 可以是TRUE (集合具有延迟性,只有在访问的时候才加载), EXTRA (集合具有延迟性,并且所有的操作都会尽量避免加载集合, 对于一个巨大的集合特别有用,因为这样的集合中的元素没有必要全部加载)和 FALSE (非延迟加载的关联)

@Fetch(FetchMode.*SELECT*)

定义了加载关联关系的获取策略. FetchMode 可以是 SELECT (在需要加载关联的时候触发select操作), SUBSELECT (只对集合有效,使用了子查询策略,详情参考Hibernate参考文档) or JOIN (在加载主实体(owner entity)的时候使用SQL JOIN来加载关联关系). JOIN 将覆写任何延迟属性 (通过JOIN策略加载的关联将不再具有延迟性)

**延迟和获取选项的等效注解**

| **Annotations** | **Lazy** | **Fetch** |
| --- | --- | --- |
| @[One|Many]ToOne](fetch=FetchType.LAZY) | @LazyToOne(PROXY) | @Fetch(SELECT) |
| @[One|Many]ToOne](fetch=FetchType.EAGER) | @LazyToOne(FALSE) | @Fetch(JOIN) |
| @ManyTo[One|Many](fetch=FetchType.LAZY) | @LazyCollection(TRUE) | @Fetch(SELECT) |
| @ManyTo[One|Many](fetch=FetchType.EAGER) | @LazyCollection(FALSE) | @Fetch(JOIN) |

.

**内建约束**

| **注解** | **应用目标** | **运行时检查** | **Hibernate元数据影响** |
| --- | --- | --- | --- |
| @Length(min=, max=) | 属性(String) | 检查字符串长度是否符合范围 | 列长度会被设到最大值 |
| @Max(value=) | 属性 (以numeric或者string类型来表示一个数字) | 检查值是否小于或等于最大值 | 对列增加一个检查约束 |
| @Min(value=) | 属性(以numeric或者string类型来表示一个数字) | 检查值是否大于或等于最小值 | 对列增加一个检查约束 |
| @NotNull | 属性 | 检查值是否非空(not null) | 列不为空 |
| @Past | 属性(date或calendar) | 检查日期是否是过去时 | 对列增加一个检查约束 |
| @Future | 属性 (date 或 calendar) | 检查日期是否是将来时 | 无 |
| @Pattern(regex="regexp", flag=) | 属性 (string) | 检查属性是否与给定匹配标志的正则表达式相匹配(见 java.util.regex.Pattern ) | 无 |
| @Range(min=, max=) | 属性(以numeric或者string类型来表示一个数字) | 检查值是否在最小和最大值之间(包括临界值) | 对列增加一个检查约束 |
| @Size(min=, max=) | 属性 (array, collection, map) | 检查元素大小是否在最小和最大值之间(包括临界值) | 无 |
| @AssertFalse | 属性 | 检查方法的演算结果是否为false(对以代码方式而不是注解表示的约束很有用) | 无 |
| @AssertTrue | 属性 | 检查方法的演算结果是否为true(对以代码方式而不是注解表示的约束很有用) | 无 |
| @Valid | 属性 (object) | 对关联对象递归的进行验证。如果对象是集合或数组，就递归地验证其元素。如果对象是Map，则递归验证其值元素。 | 无 |
| @Email | 属性（String） | 检查字符串是否符合有效的email地址规范。 | 无 |

[自定义约束](http://docs.jboss.org/hibernate/annotations/3.4/reference/zh_cn/html_single/#d0e3208)

@Temporal(TemporalType.*TIMESTAMP*)//TIMESTAMP/TIME

对时间类型进行控制

@Transient

打上这个注解的属性会被orm框架忽略