声明实体Bean

@Entity

public class Flight implements Serializable {

Long id;

@Id

public Long getId() { return id; }

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

}

@Entity 注解将一个类声明为实体 Bean, @Id 注解声明了该实体Bean的标识属性。

Hibernate 可以对类的属性或者方法进行注解。属性对应field类别，方法的 getXxx()对应property类别。

定义表

通过 @Table 为实体Bean指定对应数据库表，目录和schema的名字。

@Entity

@Table(name="tbl\_sky")

public class Sky implements Serializable {

...

@Table 注解包含一个schema和一个catelog 属性，使用@UniqueConstraints 可以定义表的唯一约束。

@Table(name="tbl\_sky",

uniqueConstraints = {@UniqueConstraint(columnNames={"month", "day"})}

)

上述代码在 "month" 和 "day" 两个 field 上加上 unique constrainst.

@Version 注解用于支持乐观锁版本控制。

@Entity

public class Flight implements Serializable {

...

@Version

@Column(name="OPTLOCK")

public Integer getVersion() { ... }

}

version属性映射到 "OPTLOCK" 列，entity manager 使用这个字段来检测冲突。 一般可以用 数字 或者 timestamp 类型来支持 version.

实体Bean中所有非static 非 transient 属性都可以被持久化，除非用@Transient注解。

默认情况下，所有属性都用 @Basic 注解。

public transient int counter; //transient property

private String firstname; //persistent property

@Transient

String getLengthInMeter() { ... } //transient property

String getName() {... } // persistent property

@Basic

int getLength() { ... } // persistent property

@Basic(fetch = FetchType.LAZY)

String getDetailedComment() { ... } // persistent property

@Temporal(TemporalType.TIME)

java.util.Date getDepartureTime() { ... } // persistent property

@Enumerated(EnumType.STRING)

Starred getNote() { ... } //enum persisted as String in database

上述代码中 counter, lengthInMeter 属性将忽略不被持久化，而 firstname, name, length 被定义为可持久化和可获取的。

@TemporalType.(DATE,TIME,TIMESTAMP) 分别Map java.sql.(Date, Time, Timestamp).

@Lob 注解属性将被持久化为 Blog 或 Clob 类型。具体的java.sql.Clob, Character[], char[] 和 java.lang.String 将被持久化为 Clob 类型. java.sql.Blob, Byte[], byte[] 和 serializable type 将被持久化为 Blob 类型。

@Lob

public String getFullText() {

return fullText; // clob type

}

@Lob

public byte[] getFullCode() {

return fullCode; // blog type

}

@Column 注解将属性映射到列。

@Entity

public class Flight implements Serializable {

...

@Column(updatable = false, name = "flight\_name", nullable = false, length=50)

public String getName() { ... }

定义 name 属性映射到 flight\_name column, not null, can't update, length equal 50

@Column(

name="columnName"; (1) 列名)

boolean unique() default false; (2) 是否在该列上设置唯一约束

boolean nullable() default true; (3) 列可空？

boolean insertable() default true; (4) 该列是否作为生成 insert语句的一个列

boolean updatable() default true; (5) 该列是否作为生成 update语句的一个列

String columnDefinition() default ""; (6) 默认值

String table() default ""; (7) 定义对应的表（deault 是主表）

int length() default 255; (8) 列长度

int precision() default 0; // decimal precision (9) decimal精度

int scale() default 0; // decimal scale (10) decimal长度

嵌入式对象（又称组件）也就是别的对象定义的属性

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

@Entity

public class Person implements Serializable {

// Persistent component using defaults

Address homeAddress;

@Embedded

@AttributeOverrides( {

@AttributeOverride(name="iso2", column = @Column(name="bornIso2") ),

@AttributeOverride(name="name", column = @Column(name="bornCountryName") )

} )

Country bornIn;

...

}

@Embeddable

public class Address implements Serializable {

String city;

Country nationality; //no overriding here

}

@Embeddable

public class Country implements Serializable {

private String iso2;

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

public String getIso2() { return iso2; }

public void setIso2(String iso2) { this.iso2 = iso2; }

public String getName() { return name; }

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

...

}

Person 类定义了 Address 和 Country 对象，具体两个类实现见上。

无注解属性默认值：

? 属性为简单类型，则映射为 @Basic

? 属性对应的类型定义了 @Embeddable 注解，则映射为 @Embedded

? 属性对应的类型实现了Serializable,则属性被映射为@Basic并在一个列中保存该对象的serialized版本。

? 属性的类型为 java.sql.Clob or java.sql.Blob, 则映射到 @Lob 对应的类型。

映射主键属性

@Id 注解可将实体Bean中某个属性定义为主键，使用@GenerateValue注解可以定义该标识符的生成策略。

? AUTO - 可以是 identity column, sequence 或者 table 类型，取决于不同底层的数据库

? TABLE - 使用table保存id值

? IDENTITY - identity column

? SEQUENCE - seque

nce

@Id @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="SEQ\_STORE")

public Integer getId() { ... }

@Id @GeneratedValue(strategy=GenerationType.IDENTITY)

public Long getId() { ... }

AUTO 生成器，适用与可移值的应用，多个@Id可以共享同一个 identifier生成器，只要把generator属性设成相同的值就可以。通过@SequenceGenerator 和 @TableGenerator 可以配置不同的 identifier 生成器。

<table-generator name="EMP\_GEN"

table="GENERATOR\_TABLE"

pk-column-name="key"

value-column-name="hi"

pk-column-value="EMP"

allocation-size="20"/>

//and the annotation equivalent

@javax.persistence.TableGenerator(

name="EMP\_GEN",

table="GENERATOR\_TABLE",

pkColumnName = "key",

valueColumnName = "hi"

pkColumnValue="EMP",

allocationSize=20

)

<sequence-generator name="SEQ\_GEN"

sequence-name="my\_sequence"

allocation-size="20"/>

//and the annotation equivalent

@javax.persistence.SequenceGenerator(

name="SEQ\_GEN",

sequenceName="my\_sequence",

allocationSize=20

)

The next example shows the definition of a sequence generator in a class scope:

@Entity

@javax.persistence.SequenceGenerator(

name="SEQ\_STORE",

sequenceName="my\_sequence"

)

public class Store implements Serializable {

private Long id;

@Id @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="SEQ\_STORE")

public Long getId() { return id; }

}

Store类使用名为my\_sequence的sequence，并且SEQ\_STORE生成器对于其他类是不可见的。

通过下面语法，你可以定义组合键。

? 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，并将该实体中所有主键的属性都注解为 @Id

@Entity

@IdClass(FootballerPk.class)

public class Footballer {

//part of the id key

@Id public String getFirstname() {

return firstname;

}

public void setFirstname(String firstname) {

this.firstname = firstname;

}

//part of the id key

@Id public String getLastname() {

return lastname;

}

public void setLastname(String lastname) {

this.lastname = lastname;

}

public String getClub() {

return club;

}

public void setClub(String club) {

this.club = club;

}

//appropriate equals() and hashCode() implementation

}

@Embeddable

public class FootballerPk implements Serializable {

//same name and type as in Footballer

public String getFirstname() {

return firstname;

}

public void setFirstname(String firstname) {

this.firstname = firstname;

}

//same name and type as in Footballer

public String getLastname() {

return lastname;

}

public void setLastname(String lastname) {

this.lastname = lastname;

}

//appropriate equals() and hashCode() implementation

}

@Entity

@AssociationOverride( name="id.channel", joinColumns = @JoinColumn(name="chan\_id") )

public class TvMagazin {

@EmbeddedId public TvMagazinPk id;

@Temporal(TemporalType.TIME) Date time;

}

@Embeddable

public class TvMagazinPk implements Serializable {

@ManyToOne

public Channel channel;

public String name;

@ManyToOne

public Presenter presenter;

}

映射继承关系

EJB支持3种类型的继承。

? Table per Class Strategy: the <union-class> element in Hibernate 每个类一张表

? Single Table per Class Hierarchy Strategy: the <subclass> element in Hibernate 每个类层次结构一张表

? Joined Subclass Strategy: the <joined-subclass> element in Hibernate 连接的子类策略

@Inheritance 注解来定义所选的之类策略。

每个类一张表

@Entity

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

public class Flight implements Serializable {

有缺点，如多态查询或关联。Hibernate 使用 SQL Union 查询来实现这种策略。 这种策略支持双向的一对多关联，但不支持 IDENTIFY 生成器策略，因为ID必须在多个表间共享。一旦使用就不能使用AUTO和IDENTIFY生成器。

每个类层次结构一张表

@Entity

@Inheritance(strategy=InheritanceType.SINGLE\_TABLE)

@DiscriminatorColumn(

name="planetype",

discriminatorType=DiscriminatorType.STRING

)

@DiscriminatorValue("Plane")

public class Plane { ... }

@Entity

@DiscriminatorValue("A320")

public class A320 extends Plane { ... }

整个层次结构中的所有父类和子类属性都映射到同一个表中，他们的实例通过一个辨别符列（discriminator）来区分。

Plane 是父类。@DiscriminatorColumn 注解定义了辨别符列。对于继承层次结构中的每个类, @DiscriminatorValue 注解指定了用来辨别该类的值。 辨别符列名字默认为 DTYPE，其默认值为实体名。其类型为DiscriminatorType.STRING。

连接的子类

@Entity

@Inheritance(strategy=InheritanceType.JOINED)

public class Boat implements Serializable { ... }

@Entity

public class Ferry extends Boat { ... }

@Entity

@PrimaryKeyJoinColumn(name="BOAT\_ID")

public class AmericaCupClass extends Boat { ... }

以上所有实体使用 JOINED 策略 Ferry和Boat class使用同名的主键关联(eg: Boat.id = Ferry.id)， AmericaCupClass 和 Boat 关联的条件为 Boat.id = AmericaCupClass.BOAT\_ID.

从父类继承的属性

@MappedSuperclass

public class BaseEntity {

@Basic

@Temporal(TemporalType.TIMESTAMP)

public Date getLastUpdate() { ... }

public String getLastUpdater() { ... }

...

}

@Entity class Order extends BaseEntity {

@Id public Integer getId() { ... }

...

}

继承父类的一些属性，但不用父类作为映射实体，这时候需要 @MappedSuperclass 注解。 上述实体映射到数据库中的时候对应 Order 实体Bean, 其具有 id, lastUpdate, lastUpdater 三个属性。如果没有@MappedSuperclass 注解,则父类中属性忽略，这是 Order 实体 Bean 只有 id 一个属性。

映射实体Bean的关联关系

一对一

使用 @OneToOne 注解可以建立实体Bean之间的一对一关系。一对一关系有3种情况。

? 关联的实体都共享同样的主键。

@Entity

public class Body {

@Id

public Long getId() { return id; }

@OneToOne(cascade = CascadeType.ALL)

@PrimaryKeyJoinColumn

public Heart getHeart() {

return heart;

}

...

}

@Entity

public class Heart {

@Id

public Long getId() { ...}

}

通过@PrimaryKeyJoinColumn 注解定义了一对一的关联关系。

多对一

使用 @ManyToOne 注解定义多对一关系。

@Entity()

public class Flight implements Serializable {

@ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE} )

@JoinColumn(name="COMP\_ID")

public Company getCompany() {

return company;

}

...

}

其中@JoinColumn 注解是可选的，关键字段默认值和一对一关联的情况相似。列名为：主题的关联属性名 + 下划线 + 被关联端的主键列名。本例中为company\_id,因为关联的属性是company, Company的主键为 id.

@ManyToOne 注解有个targetEntity属性，该参数定义了目标实体名。通常不需要定义，大部分情况为默认值。但下面这种情况则需要 targetEntity 定义（使用接口作为返回值，而不是常用的实体）。

@Entity()

public class Flight implements Serializable {

@ManyToOne(cascade= {CascadeType.PERSIST,CascadeType.MERGE},targetEntity= CompanyImpl.class)

@JoinColumn(name="COMP\_ID")

public Company getCompany() {

return company;

}

...

}

public interface Company {

...

多对一也可以通过关联表的方式来映射，通过 @JoinTable 注解可定义关联表。该关联表包含指回实体的外键（通过@JoinTable.joinColumns）以及指向目标实体表的外键（通过@JoinTable.inverseJoinColumns）.

@Entity()

public class Flight implements Serializable {

@ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE} )

@JoinTable(name="Flight\_Company",

joinColumns = @JoinColumn(name="FLIGHT\_ID"),

inverseJoinColumns = @JoinColumn(name="COMP\_ID")

)

public Company getCompany() {

return company;

}

...

}

集合类型

一对多

@OneToMany 注解可定义一对多关联。一对多关联可以是双向的。

双向

规范中多对一端几乎总是双向关联中的主体（owner）端，而一对多的关联注解为 @OneToMany(mappedBy=)

@Entity

public class Troop {

@OneToMany(mappedBy="troop")

public Set<Soldier> getSoldiers() {

...

}

@Entity

public class Soldier {

@ManyToOne

@JoinColumn(name="troop\_fk")

public Troop getTroop() {

...

}

Troop 通过troop属性和Soldier建立了一对多的双向关联。在 mappedBy 端不必也不能定义任何物理映射。

单向

@Entity

public class Customer implements Serializable {

@OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)

@JoinColumn(name="CUST\_ID")

public Set<Ticket> getTickets() {

...

}

@Entity

public class Ticket implements Serializable {

... //no bidir

}

一般通过连接表来实现这种关联，可以通过@JoinColumn注解来描述这种单向关联关系。上例 Customer 通过 CUST\_ID 列和 Ticket 建立了单向关联关系。

通过关联表来处理单向关联

@Entity

public class Trainer {

@OneToMany

@JoinTable(

name="TrainedMonkeys",

joinColumns = @JoinColumn( name="trainer\_id"),

inverseJoinColumns = @JoinColumn( name="monkey\_id")

)

public Set<Monkey> getTrainedMonkeys() {

...

}

@Entity

public class Monkey {

... //no bidir

}

通过关联表来处理单向一对多关系是首选，这种关联通过 @JoinTable 注解来进行描述。上例子中 Trainer 通过TrainedMonkeys表和Monkey建立了单向关联关系。其中外键trainer\_id关联到Trainer(joinColumns)而外键monkey\_id关联到Monkey(inverseJoinColumns).

默认处理机制

通过连接表来建立单向一对多关联不需要描述任何物理映射，表名由一下3个部分组成，主表(owner table)表名 + 下划线 + 从表(the other side table)表名。指向主表的外键名:主表表名+下划线+主表主键列名 指向从表的外键定义为唯一约束，用来表示一对多的关联关系。

@Entity

public class Trainer {

@OneToMany

public Set<Tiger> getTrainedTigers() {

...

}

@Entity

public class Tiger {

... //no bidir

}

上述例子中 Trainer 和 Tiger 通过 Trainer\_Tiger 连接表建立单向关联关系。其中外键 trainer\_id 关联到 Trainer表，而外键 trainedTigers\_id 关联到 Tiger 表。

多对多

通过 @ManyToMany 注解定义多对多关系，同时通过 @JoinTable 注解描述关联表和关联条件。其中一端定义为 owner, 另一段定义为 inverse(对关联表进行更新操作，这段被忽略)。

@Entity

public class Employer implements Serializable {

@ManyToMany(

targetEntity=org.hibernate.test.metadata.manytomany.Employee.class,

cascade={CascadeType.PERSIST, CascadeType.MERGE}

)

@JoinTable(

name="EMPLOYER\_EMPLOYEE",

joinColumns=@JoinColumn(name="EMPER\_ID"),

inverseJoinColumns=@JoinColumn(name="EMPEE\_ID")

)

public Collection getEmployees() {

return employees;

}

...

}

@Entity

public class Employee implements Serializable {

@ManyToMany(

cascade = {CascadeType.PERSIST, CascadeType.MERGE},

mappedBy = "employees",

targetEntity = Employer.class

)

public Collection getEmployers() {

return employers;

}

}

默认值：

关联表名：主表表名 + 下划线 + 从表表名；关联表到主表的外键：主表表名 + 下划线 + 主表中主键列名；关联表到从表的外键名：主表中用于关联的属性名 + 下划线 + 从表的主键列名。

用 cascading 实现传播持久化（Transitive persistence）

cascade 属性接受值为 CascadeType 数组，其类型如下：

? CascadeType.PERSIST: cascades the persist (create) operation to associated entities persist() is called or if the entity is managed 如果一个实体是受管状态，或者当 persist() 函数被调用时，触发级联创建(create)操作。

? CascadeType.MERGE: cascades the merge operation to associated entities if merge() is called or if the entity is managed 如果一个实体是受管状态，或者当 merge() 函数被调用时，触发级联合并(merge)操作。

? CascadeType.REMOVE: cascades the remove operation to associated entities if delete() is called 当 delete() 函数被调用时，触发级联删除(remove)操作。

? CascadeType.REFRESH: cascades the refresh operation to associated entities if refresh() is called 当 refresh() 函数被调用时，出发级联更新(refresh)操作。

? CascadeType.ALL: all of the above 以上全部

映射二级列表

使用类一级的 @SecondaryTable 和 @SecondaryTables 注解可以实现单个实体到多个表的映射。使用 @Column 或者 @JoinColumn 注解中的 table 参数可以指定某个列所属的特定表。

@Entity

@Table(name="MainCat")

@SecondaryTables({

@SecondaryTable(name="Cat1", pkJoinColumns={

@PrimaryKeyJoinColumn(name="cat\_id", referencedColumnName="id")}),

@SecondaryTable(name="Cat2", uniqueConstraints={

@UniqueConstraint(columnNames={"storyPart2"})})

})

public class Cat implements Serializable {

private Integer id;

private String name;

private String storyPart1;

private String storyPart2;

@Id @GeneratedValue

public Integer getId() {

return id;

}

public String getName() {

return name;

}

@Column(table="Cat1")

public String getStoryPart1() {

return storyPart1;

}

@Column(table="Cat2")

public String getStoryPart2() {

return storyPart2;

}

上述例子中， name 保存在 MainCat 表中，storyPart1保存在 Cat1 表中，storyPart2 保存在 Cat2 表中。 Cat1 表通过外键 cat\_id 和 MainCat 表关联， Cat2 表通过 id 列和 MainCat 表关联。对storyPart2 列还定义了唯一约束。

映射查询

使用注解可以映射 EJBQL/HQL 查询，@NamedQuery 和 @NamedQueries 是可以使用在类级别或者JPA的XML文件中的注解。

<entity-mappings>

<named-query name="plane.getAll">

<query>select p from Plane p</query>

</named-query>

...

</entity-mappings>

...

@Entity

@NamedQuery(name="night.moreRecentThan", query="select n from Night n where n.date >= :date")

public class Night {

...

}

public class MyDao {

doStuff() {

Query q = s.getNamedQuery("night.moreRecentThan");

q.setDate( "date", aMonthAgo );

List results = q.list();

...

}

...

}

可以通过定义 QueryHint 数组的 hints 属性为查询提供一些 hint 信息。下图是一些 Hibernate hints:

映射本地化查询

通过@SqlResultSetMapping 注解来描述 SQL 的 resultset 结构。如果定义多个结果集映射，则用 @SqlResultSetMappings。

@NamedNativeQuery(name="night&area", query="select night.id nid, night.night\_duration, "

+ " night.night\_date, area.id aid, night.area\_id, area.name "

+ "from Night night, Area area where night.area\_id = area.id", resultSetMapping="joinMapping")

@SqlResultSetMapping( name="joinMapping", entities={

@EntityResult(entityClass=org.hibernate.test.annotations.query.Night.class, fields = {

@FieldResult(name="id", column="nid"),

@FieldResult(name="duration", column="night\_duration"),

@FieldResult(name="date", column="night\_date"),

@FieldResult(name="area", column="area\_id"),

discriminatorColumn="disc"

}),

@EntityResult(entityClass=org.hibernate.test.annotations.query.Area.class, fields = {

@FieldResult(name="id", column="aid"),

@FieldResult(name="name", column="name")

})

}

)

上面的例子，名为“night&area”的查询和 "joinMapping"结果集映射对应，该映射返回两个实体，分别为 Night 和 Area, 其中每个属性都和一个列关联，列名通过查询获取。

@Entity

@SqlResultSetMapping(name="implicit",

entities=@EntityResult(

entityClass=org.hibernate.test.annotations.@NamedNativeQuery(

name="implicitSample", query="select \* from SpaceShip",

resultSetMapping="implicit")

public class SpaceShip {

private String name;

private String model;

private double speed;

@Id

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

@Column(name="model\_txt")

public String getModel() {

return model;

}

public void setModel(String model) {

this.model = model;

}

public double getSpeed() {

return speed;

}

public void setSpeed(double speed) {

this.speed = speed;

}

}

上例中 model1 属性绑定到 model\_txt 列，如果和相关实体关联设计到组合主键，那么应该使用 @FieldResult 注解来定义每个外键列。@FieldResult的名字组成：定义这种关系的属性名字 + "." + 主键名或主键列或主键属性。

@Entity

@SqlResultSetMapping(name="compositekey",

entities=@EntityResult(entityClass=org.hibernate.test.annotations.query.SpaceShip.class,

fields = {

@FieldResult(name="name", column = "name"),

@FieldResult(name="model", column = "model"),

@FieldResult(name="speed", column = "speed"),

@FieldResult(name="captain.firstname", column = "firstn"),

@FieldResult(name="captain.lastname", column = "lastn"),

@FieldResult(name="dimensions.length", column = "length"),

@FieldResult(name="dimensions.width", column = "width")

}),

columns = { @ColumnResult(name = "surface"),

@ColumnResult(name = "volume") } )

@NamedNativeQuery(name="compositekey",

query="select name, model, speed, lname as lastn, fname as firstn, length, width, length \* width as resultSetMapping="compositekey")

})

如果查询返回的是单个实体，或者打算用系统默认的映射，这种情况下可以不使用 resultSetMapping，而使用resultClass属性，例如：

@NamedNativeQuery(name="implicitSample", query="select \* from SpaceShip",

resultClass=SpaceShip.class)

public class SpaceShip {

Hibernate 独有的注解扩展

Hibernate 提供了与其自身特性想吻合的注解，org.hibernate.annotations package包含了这些注解。

实体

org.hibernate.annotations.Entity 定义了 Hibernate 实体需要的信息。

? mutable: whether this entity is mutable or not 此实体是否可变

? dynamicInsert: allow dynamic SQL for inserts 用动态SQL新增

? dynamicUpdate: allow dynamic SQL for updates 用动态SQL更新

? selectBeforeUpdate: Specifies that Hibernate should never perform an SQL UPDATE unless it is certain that an object is actually modified.指明Hibernate从不运行SQL Update，除非能确定对象已经被修改

? polymorphism: whether the entity polymorphism is of PolymorphismType.IMPLICIT (default) or PolymorphismType.EXPLICIT 指出实体多态是 PolymorphismType.IMPLICIT(默认)还是PolymorphismType.EXPLICIT

? optimisticLock: optimistic locking strategy (OptimisticLockType.VERSION, OptimisticLockType.NONE, OptimisticLockType.DIRTY or OptimisticLockType.ALL) 乐观锁策略

标识符

@org.hibernate.annotations.GenericGenerator和@org.hibernate.annotations.GenericGenerators允许你定义hibernate特有的标识符。

@Id @GeneratedValue(generator="system-uuid")

@GenericGenerator(name="system-uuid", strategy = "uuid")

public String getId() {

@Id @GeneratedValue(generator="hibseq")

@GenericGenerator(name="hibseq", strategy = "seqhilo",

parameters = {

@Parameter(name="max\_lo", value = "5"),

@Parameter(name="sequence", value="heybabyhey")

}

)

public Integer getId() {

新例子

@GenericGenerators(

{

@GenericGenerator(

name="hibseq",

strategy = "seqhilo",

parameters = {

@Parameter(name="max\_lo", value = "5"),

@Parameter(name="sequence", value="heybabyhey")

}

),

@GenericGenerator(...)

}

)

自然ID

用 @NaturalId 注解标识

公式

让数据库而不是JVM进行计算。

@Formula("obj\_length \* obj\_height \* obj\_width")

public long getObjectVolume()

索引

通过在列属性(property)上使用@Index注解，可以指定特定列的索引，columnNames属性(attribute)将随之被忽略。

@Column(secondaryTable="Cat1")

@Index(name="story1index")

public String getStoryPart1() {

return storyPart1;

}

辨别符

@Entity

@DiscriminatorFormula("case when forest\_type is null then 0 else forest\_type end")

public class Forest { ... }

过滤 查询 ...

? 其中一个实体通过外键关联到另一个实体的主键。注：一对一，则外键必须为唯一约束。

@Entity

public class Customer implements Serializable {

@OneToOne(cascade = CascadeType.ALL)

@JoinColumn(name="passport\_fk")

public Passport getPassport() {

...

}

@Entity

public class Passport implements Serializable {

@OneToOne(mappedBy = "passport")

public Customer getOwner() {

...

}

通过@JoinColumn注解定义一对一的关联关系。如果没有@JoinColumn注解，则系统自动处理，在主表中将创建连接列，列名为：主题的关联属性名 + 下划线 + 被关联端的主键列名。上例为 passport\_id, 因为Customer 中关联属性为 passport, Passport 的主键为 id.

? 通过关联表来保存两个实体之间的关联关系。注：一对一，则关联表每个外键都必须是唯一约束。

@Entity

public class Customer implements Serializable {

@OneToOne(cascade = CascadeType.ALL)

@JoinTable(name = "CustomerPassports",

joinColumns = @JoinColumn(name="customer\_fk"),

inverseJoinColumns = @JoinColumn(name="passport\_fk")

)

public Passport getPassport() {

...

}

@Entity public class Passport implements Serializable {

@OneToOne(mappedBy = "passport")

public Customer getOwner() {

...

}

Customer 通过 CustomerPassports 关联表和 Passport 关联。该关联表通过 passport\_fk 外键指向 Passport 表，该信心定义为 inverseJoinColumns 的属性值。 通过 customer\_fk 外键指向 Customer 表，该信息定义为 joinColumns 属性值。