


Hibernate4 之 JPA 规范配置详解

1 @Table

Table 用来定义 entity 主表的 name, catalog, schema 等属性。

属性说明：

- name: 表名
- catalog: 对应关系数据库中的 catalog
- schema: 对应关系数据库中的 schema
- UniqueConstraints: 定义一个 UniqueConstraint 数组，指定需要建唯一约束的列

Java 代码 

```
1. @Entity
2. @Table(name="CUST")
3. public class Customer { ... }
```

2 @SecondaryTable

一个 entity class 可以映射到多表，SecondaryTable 用来定义单个从表的名字，主键名字等属性。

属性说明：

- name: 表名
- catalog: 对应关系数据库中的 catalog
- pkJoin: 定义一个 PrimaryKeyJoinColumn 数组，指定从表的主键列
- UniqueConstraints: 定义一个 UniqueConstraint 数组，指定需要建唯一约束的列

下面的代码说明 Customer 类映射到两个表，主表名是 CUSTOMER，从表名是 CUST_DETAIL，从表的主键列和主表的主键列类型相同，列名为 CUST_ID。

Java 代码 


```
1. @Entity
2. @Table(name="CUSTOMER")
3. @SecondaryTable(name="CUST_DETAIL",pkJoin=@PrimaryKeyJoinColumn(name="CUST_ID"))
4. public class Customer { ... }
```

3 @SecondaryTables

当一个 entity class 映射到一个主表和多个从表时，用 **SecondaryTables** 来定义各个从表的属性。

属性说明：

- **value**: 定义一个 **SecondaryTable** 数组，指定每个从表的属性。

Java 代码 


```
1. @Table(name = "CUSTOMER")
2. @SecondaryTables( value = {
3.     @SecondaryTable(name = "CUST_NAME", pkJoin = { @PrimaryKeyJoinColumn(name
        e = "STMO_ID", referencedColumnName = "id") }),
4.     @SecondaryTable(name = "CUST_ADDRESS", pkJoin = { @PrimaryKeyJoinColumn(name
        e = "STMO_ID", referencedColumnName = "id") }) })
5. public class Customer {}
```

4 @UniqueConstraint

UniqueConstraint 定义在 **Table** 或 **SecondaryTable** 元数据里，用来指定建表时需要建唯一约束的列。

属性说明：

- **columnNames**: 定义一个字符串数组，指定要建唯一约束的列名。

Java 代码 

```
1. @Entity
2. @Table(name="EMPLOYEE",uniqueConstraints={@UniqueConstraint(columnNames={"EMP_ID", "EMP_NAME"})})
3. public class Employee { ... }
```


5 @Column

Column 元数据定义了映射到数据库的列的所有属性：列名，是否唯一，是否允许为空，是否允许更新等。

属性说明：

- **unique**: 是否唯一
- **nullable**: 是否允许为空

- insertable: 是否允许插入
- updatable: 是否允许更新
- columnDefinition: 定义建表时创建此列的 DDL
- secondaryTable: 从表名。如果此列不建在主表上（默认建在主表），该属性定义该列所在从表的名字。

Java 代码 

```
1. public class Person {
2.     @Column(name = "PERSONNAME", unique = true, nullable = false, updatable = true)
3.     private String name;
4.     @Column(name = "PHOTO", columnDefinition = "BLOB NOT NULL", secondaryTable="PER_PHOTO")
5.     private byte[] picture;
```

6 @JoinColumn

如果在 entity class 的 field 上定义了关系(one2one 或 one2many 等), 我们通过 JoinColumn 来定义关系的属性。JoinColumn 的大部分属性和 Column 类似。

属性说明:

- unique: 是否唯一
- referencedColumnName: 该列指向列的列名（建表时该列作为外键列指向关系另一端的指定列）
- nullable: 是否允许为空
- insertable: 是否允许插入
- updatable: 是否允许更新
- columnDefinition: 定义建表时创建此列的 DDL
- secondaryTable: 从表名。如果此列不建在主表上（默认建在主表），该属性定义该列所在从表的名字。

下面的代码说明 Custom 和 Order 是一对一关系。在 Order 对应的映射表建一个名为 CUST_ID 的列，该列作为外键指向 Custom 对应表中名为 ID 的列。

Java 代码 

```

1. public class Custom {
2.     @OneToOne
3.     @JoinColumn(name="CUST_ID", referencedColumnName="ID", unique=true, nullable=true, updatable=true)
4.     public Order getOrder() {
5.         return order;
6.     }

```


7 @JoinColumns

如果在 entity class 的 field 上定义了关系（one2one 或 one2many 等），并且关系存在多个 JoinColumn，用 JoinColumns 定义多个 JoinColumn 的属性。

属性说明：

- value: 定义 JoinColumn 数组，指定每个 JoinColumn 的属性。

下面的代码说明 Custom 和 Order 是一对一关系。在 Order 对应的映射表建两列，一列名为 CUST_ID, 该列作为外键指向 Custom 对应表中名为 ID 的列, 另一列名为 CUST_NAME, 该列作为外键指向 Custom 对应表中名为 NAME 的列。

Java 代码 

```

1. public class Custom {
2.     @OneToOne
3.     @JoinColumns({
4.         @JoinColumn(name="CUST_ID", referencedColumnName="ID"),
5.         @JoinColumn(name="CUST_NAME", referencedColumnName="NAME")
6.     })
7.     public Order getOrder() {
8.         return order;
9.     }

```


8 @Id

声明当前 field 为映射表中的主键列。id 值的获取方式有五种：TABLE, SEQUENCE, IDENTITY, AUTO, NONE。Oracle 和 DB2 支持 SEQUENCE, SQL Server 和 Sybase 支持 IDENTITY, mysql 支持 AUTO。所有的数据库都可以指定为 AUTO, 我们会根据不同数据库做转换。NONE(默认)需要用户自己指定 Id 的值。

属性说明：

- **generate**: 主键值的获取类型
- **generator**: TableGenerator 的名字（当 **generate=GeneratorType.TABLE** 才需要指定该属性）

下面的代码声明 Task 的主键列 id 是自动增长的。(Oracle 和 DB2 从默认的 SEQUENCE 取值，SQL Server 和 Sybase 该列建成 IDENTITY，mysql 该列建成 auto increment。)

Java 代码 

```
1. @Entity
2. @Table(name = "OTASK")
3. public class Task {
4.     @Id(generate = GeneratorType.AUTO)
5.     public Integer getId() {
6.         return id;
7.     }
8. }
```


9 @IdClass

当 entity class 使用复合主键时，需要定义一个类作为 id class。id class 必须符合以下要求：类必须声明为 public，并提供一个声明为 public 的空构造函数。必须实现 Serializable 接口，覆盖 equals() 和 hashCode() 方法。entity class 的所有 id field 在 id class 都要定义，且类型一样。

属性说明：

- **value**: id class 的类名

下面的代码声明 Task 的主键列 id 是自动增长的。(Oracle 和 DB2 从默认的 SEQUENCE 取值，SQL Server 和 Sybase 该列建成 IDENTITY，mysql 该列建成 auto increment。)

Java 代码 

```
1. public class EmployeePK implements java.io.Serializable{
2.     String empName;
3.     Integer empAge;
4.     public EmployeePK(){ }
5.     public boolean equals(Object obj){ .....}
6.     public int hashCode(){.....}
7. }
```

```
8.
9. @IdClass(value=com.acme.EmployeePK.class)
10. @Entity(access=FIELD)
11. public class Employee {
12.     @Id String empName;
13.     @Id Integer empAge;
14. }
```


10 @MapKey

在一对多，多对多关系中，我们可以用 **Map** 来保存集合对象。默认用主键值做 **key**，如果使用复合主键，则用 **id class** 的实例做 **key**，如果指定了 **name** 属性，就用指定的 **field** 的值做 **key**。

属性说明：

- **name**: 用来做 **key** 的 **field** 名字

下面的代码说明 **Person** 和 **Book** 之间是一对多关系。**Person** 的 **books** 字段是 **Map** 类型，用 **Book** 的 **isbn** 字段的值作为 **Map** 的 **key**。

Java 代码 

```
1. @Table(name = "PERSON")
2. public class Person {
3.     @OneToMany(targetEntity = Book.class, cascade = CascadeType.ALL, mappedBy = "person")
4.     @MapKey(name = "isbn")
5.     private Map books = new HashMap();
6. }
```


11 @MappedSuperclass

使用 **@MappedSuperclass** 指定一个实体类从中继承持久字段的超类。当多个实体类共享通用的持久字段或属性时，这将是一个方便的模式。

您可以像对实体那样使用任何直接和关系映射批注（如 **@Basic** 和 **@ManyToMany**）对该超类的字段和属性进行批注，但由于没有针对该超类本身的表存在，因此这些映射只适用于它的子类。继承的持久字段或属性属于子类的表。

可以在子类中使用 `@AttributeOverride` 或 `@AssociationOverride` 来覆盖超类的映射配置。

`@MappedSuperclass` 没有属性。

Java 代码 

```
1. //如何将 Employee 指定为映射超类
2. @MappedSuperclass
3. public class Employee {
4.     @Id
5.     protected Integer empId;
6.
7.     @Version
8.     protected Integer version;
9.
10.    @ManyToOne
11.    @JoinColumn(name="ADDR")
12.    protected Address address;
13. }
14.
15. //如何在实体类中使用@AttributeOverride 以覆盖超类中设置的配置。
16. @Entity
17. @AttributeOverride(name="address", column=@Column(name="ADDR_ID"))
18. public class PartTimeEmployee extends Employee {
19.     @Column(name="WAGE")
20.     protected Float hourlyWage;
21. }
```

12 @PrimaryKeyJoinColumn

在三种情况下会用到 `@PrimaryKeyJoinColumn`


- 继承。
- entity class 映射到一个或多个从表。从表根据主表的主键列（列名为 `referencedColumnName` 值的列），建立一个类型一样的主键列，列名由 `name` 属性定义。
- one2one 关系，关系维护端的主键作为外键指向关系被维护端的主键，不再新建一个外键列。

属性说明：

- `name`: 列名。


- **referencedColumnName:** 该列引用列的列名
- **columnDefinition:** 定义建表时创建此列的 DDL

下面的代码说明 **Customer** 映射到两个表，主表 **CUSTOMER**,从表 **CUST_DETAIL**，从表需要建立主键列 **CUST_ID**，该列和主表的主键列 **id** 除了列名不同，其他定义一样。

Java 代码 

```
1. @Entity
2. @Table(name="CUSTOMER")
3. @SecondaryTable(name="CUST_DETAIL",pkJoin=@PrimaryKeyJoinColumn(name="CUST_ID", referencedColumnName="id"))
4. public class Customer {
5.     @Id(generate = GeneratorType.AUTO)
6.     public Integer getId() {
7.         return id;
8.     }
9. }
```

下面的代码说明 **Employee** 和 **EmployeeInfo** 是一对一关系，**Employee** 的主键列 **id** 作为外键指向 **EmployeeInfo** 的主键列 **INFO_ID**。

Java 代码 

```
1. @Table(name = "Employee")
2. public class Employee {
3.     @OneToOne
4.     @PrimaryKeyJoinColumn(name = "id", referencedColumnName="INFO_ID")
5.     EmployeeInfo info;
6. }
```


13 @PrimaryKeyJoinColumns

如果 **entity class** 使用了复合主键，指定单个 **PrimaryKeyJoinColumn** 不能满足要求时，可以用 **PrimaryKeyJoinColumns** 来定义多个 **PrimaryKeyJoinColumn**

属性说明：

- **value:** 一个 **PrimaryKeyJoinColumn** 数组，包含所有 **PrimaryKeyJoinColumn**


下面的代码说明了 **Employee** 和 **EmployeeInfo** 是一对一关系。他们都使用复合主键，建表时需要在 **Employee** 表建立一个外键，从 **Employee** 的主键列 **id,name** 指向 **EmployeeInfo** 的主键列 **INFO_ID** 和 **INFO_NAME**

Java 代码 

```
1. @Entity
2. @IdClass(EmpPK.class)
3. @Table(name = "EMPLOYEE")
4. public class Employee {
5.     private int id;
6.     private String name;
7.     private String address;
8.     @OneToOne(cascade = CascadeType.ALL)
9.     @PrimaryKeyJoinColumns({
10.         @PrimaryKeyJoinColumn(name="id", referencedColumnName="INFO_ID"),
11.         @PrimaryKeyJoinColumn(name="name", referencedColumnName="INFO_NAME")})
12.     EmployeeInfo info;
13. }
14.
15. @Entity
16. @IdClass(EmpPK.class)
17. @Table(name = "EMPLOYEE_INFO")
18. public class EmployeeInfo {
19.     @Id
20.     @Column(name = "INFO_ID")
21.     private int id;
22.     @Id
23.     @Column(name = "INFO_NAME")
24.     private String name;
25. }
```

14 @Transient

Transient 用来注释 entity 的属性，指定的这些属性不会被持久化，也不会为这些属性建表

Java 代码 

```
1. @Transient
2. private String name;
```

15 @Version

Version 指定实体类在乐观事务中的 **version** 属性。在实体类重新由 **EntityManager** 管理并且加入到乐观事务中时，保证完整性。每一个类只能有一个属性被指定为 **version**，**version** 属性应该映射到实体类的主表上。

属性说明：

value: 一个 **PrimaryKeyJoinColumn** 数组，包含所有 **PrimaryKeyJoinColumn**

下面的代码说明 **versionNum** 属性作为这个类的 **version**，映射到数据库中主表的列名是 **OPTLOCK**

Java 代码 

```
1. @Version
2. @Column("OPTLOCK")
3. protected int getVersionNum() { return versionNum; }
```

16 @Lob

Lob 指定一个属性作为数据库支持的大对象类型在数据库中存储。使用 **LobType** 这个枚举来定义 **Lob** 是二进制类型还是字符类型。

LobType 枚举类型说明：

BLOB 二进制大对象，**Byte[]**或者 **Serializable** 的类型可以指定为 **BLOB**。


CLOB 字符型大对象，**char[]**、**Character[]**或 **String** 类型可以指定为 **CLOB**。

属性说明：

fetch: 定义这个字段是 **lazy loaded** 还是 **eagerly fetched**。数据类型是 **FetchType** 枚举，默认为 **LAZY**，即 **lazy loaded**。

type: 定义这个字段在数据库中的 **JDBC** 数据类型。数据类型是 **LobType** 枚举，默认为 **BLOB**。

下面的代码定义了一个 **BLOB** 类型的属性和一个 **CLOB** 类型的属性

Java 代码 

```
1. @Lob
2. @Column(name="PHOTO" columnDefinition="BLOB NOT NULL")
```

```
3. protected JPEGImage picture;  
4.  
5. @Lob(fetch=EAGER, type=CLOB)  
6. @Column(name="REPORT")  
7. protected String report;
```


17 @JoinTable

JoinTable 在 many-to-many 关系的所有者一边定义。如果没有定义 JoinTable，使用 JoinTable 的默认值。

属性说明：

- **table**: 这个 join table 的 Table 定义。
- **joinColumns**: 定义指向所有者主表的外键列，数据类型是 JoinColumn 数组。
- **inverseJoinColumns**: 定义指向非所有者主表的外键列，数据类型是 JoinColumn 数组。

下面的代码定义了一个连接表 CUST 和 PHONE 的 join table。join table 的表名是 CUST_PHONE，包含两个外键，一个外键是 CUST_ID，指向表 CUST 的主键 ID，另一个外键是 PHONE_ID，指向表 PHONE 的主键 ID。

Java 代码 

```
1. @JoinTable(  
2. table=@Table(name=CUST_PHONE),  
3. joinColumns=@JoinColumn(name="CUST_ID", referencedColumnName="ID"),  
4. inverseJoinColumns=@JoinColumn(name="PHONE_ID", referencedColumnName="ID")  
5. )
```

18 @TableGenerator


TableGenerator 定义一个主键值生成器，在 Id 这个元数据的 generate=TABLE 时，generator 属性中可以使用生成器的名字。生成器可以在类、方法或者属性上定义。生成器是为多个实体类提供连续的 ID 值的表，每一行为一个类提供 ID 值，ID 值通常是整数。

属性说明：

- **name**: 生成器的唯一名字，可以被 Id 元数据使用。
- **table**: 生成器用来存储 id 值的 Table 定义。

- **pkColumnName**: 生成器表的主键名称。
- **valueColumnName**: 生成器表的 ID 值的列名称。
- **pkColumnValue**: 生成器表中的一行数据的主键值。
- **initialValue**: id 值的初始值。
- **allocationSize**: id 值的增量。

下面的代码定义了两个生成器 **empGen** 和 **addressGen**，生成器的表是 **ID_GEN**

Java 代码 

```

1. @Entity
2. public class Employee {
3.     ...
4.     @TableGenerator(name="empGen",table=@Table(name="ID_GEN"),pkColumnName="G
        EN_KEY", valueColumnName="GEN_VALUE",pkColumnValue="EMP_ID",allocationSize=
        1)
5.     @Id(generate=TABLE, generator="empGen")
6.     public int id;
7.     ...
8. }
9.
10. @Entity
11. public class Address {
12.     ...
13.     @TableGenerator(name="addressGen",table=@Table(name="ID_GEN"),pkColumnVal
        ue="ADDR_ID")
14.     @Id(generate=TABLE, generator="addressGen")
15.     public int id;
16.     ...
17. }

```

19 @SequenceGenerator

SequenceGenerator 定义一个主键值生成器，在 **Id** 这个元数据的 **generator** 属性中可以使用生成器的名字。生成器可以在类、方法或者属性上定义。生成器是数据库支持的 **sequence** 对象。

属性说明：

- **name**: 生成器的唯一名字，可以被 **Id** 元数据使用。

- **sequenceName**: 数据库中, **sequence** 对象的名称。如果不指定, 会使用提供商指定的默认名称。
- **initialValue**: **id** 值的初始值。
- **allocationSize**: **id** 值的增量。

下面的代码定义了一个使用提供商默认名称的 **sequence** 生成器

Java 代码 

```
1. @SequenceGenerator(name="EMP_SEQ", allocationSize=25)
```

20 @DiscriminatorColumn

DiscriminatorColumn 定义在使用 **SINGLE_TABLE** 或 **JOINED** 继承策略的表中区别不继承层次的列

属性说明:

- **name**: **column** 的名字。默认值为 **TYPE**。
- **columnDefinition**: 生成 DDL 的 sql 片断。
- **length**: **String** 类型的 **column** 的长度, 其他类型使用默认值 10。

下面的代码定义了一个列名为 **DISC**, 长度为 20 的 **String** 类型的区别列

Java 代码 

```
1. @Entity
2. @Table(name="CUST")
3. @Inheritance(strategy=SINGLE_TABLE,discriminatorType=STRING,discriminatorValue="CUSTOMER")
4. @DiscriminatorColumn(name="DISC", length=20)
5. public class Customer { ... }
```

21 @NamedQuery

在使用 **JPA** 持久化规范的应用程序中, 可以使用实体管理器动态创建和执行查询, 也可以预定义查询并在运行时按名称执行。


使用 **@NamedQuery** 创建与 **@Entity** 或 **@MappedSuperclass** 关联的预定义查询, 这些查询:

- 使用 JPA 查询语言进行基于任何基础数据库的可移植执行
- 经常被使用
- 比较复杂并且难于创建
- 可以在不同实体之间共享
- 只返回实体（从不返回标量值），并只返回一个类型的实体

属性说明：

- **query:** (必须属性)要指定查询，请将 **query** 设置为 JPA 查询语言（作为 **String**）
- **hints:** 默认值：空 **QueryHint** 数组。默认情况下，JPA 持续性提供程序假设 SQL 查询应完全按照 **query** 属性提供的方式执行。要微调查询的执行，可以选择将 **hints** 设置为一个 **QueryHint** 数组（请参阅 **@QueryHint**）。在执行时，**EntityManager** 将向基础数据库传递提示。
- **name:**(必须属性)要指定查询名称，请将 **name** 设置为所需的 **String** 名称

下面的代码使用 **@NamedQuery** 批注定义一个 JPA 查询语言查询，该查询使用名为 **firstname** 的参数

Java 代码 

```

1. //使用 @NamedQuery 实现一个带参数的查询
2. @Entity
3. @NamedQuery(name="findAllEmployeesByFirstName",
4.             query="SELECT OBJECT(emp) FROM Employee emp WHERE emp.firstName = :firstname")
5. public class Employee implements Serializable {
6.     ...
7. }
8.
9. //执行命名查询
10. Query queryEmployeesByFirstName = em.createNamedQuery("findAllEmployeesByFirstName");
11. queryEmployeeByFirstName.setParameter("firstName", "John");
12. Collection employees = queryEmployeeByFirstName.getResultList();


```

22 @NamedQueries

如果需要指定多个 **@NamedQuery**，则必须使用一个 **@NamedQueries** 指定所有命名查询

属性说明：

- **value:** 要指定两个或更多属性覆盖，请将 **value** 设置为 **NamedQuery** 实例数组

Java 代码 

```
1. @Entity
2. @NamedQueries({@NamedQuery(name="findAllEmployeesByFirstName",
3.                             query="SELECT OBJECT(emp) FROM Employee emp WHERE emp.firstName = :firstname"),
4.               @NamedQuery(name="findAllEmployeesByLastName",
5.                             query="SELECT OBJECT(emp) FROM Employee emp WHERE emp.lastName = :lastname")})
6. public class PartTimeEmployee extends Employee {
7.
8. }
```

23 @NamedNativeQuery


使用 **@NamedNativeQuery** 创建与 **@Entity** 或 **@MappedSuperclass** 关联的预定义查询，这些查询：

- 使用基础数据库的原生 SQL
- 经常被使用
- 比较复杂并且难于创建
- 可以在不同实体之间共享
- 返回实体、标量值或两者的组合（另请参阅 **@ColumnResult**、**@EntityResult**、**@FieldResult** 和 **@SqlResultSetMapping**）

属性说明：

- **query:** (必须属性)要指定查询，请将 **query** 设置为 SQL 查询（作为 **String**）
- **hints:** 默认值：空 **QueryHint** 数组。默认情况下，JPA 持续性提供程序假设 SQL 查询应完全按照 **query** 属性提供的方式执行。要微调查询的执行，可以选择将 **hints** 设置为一个 **QueryHint** 数组（请参阅 **@QueryHint**）。在执行时，**EntityManager** 将向基础数据库传递提示。
- **name:** (必须属性)要指定查询名称，请将 **name** 设置为所需的 **String** 名称
- **resultClass:** 默认值：JPA 持续性提供程序假设结果类是关联实体的 **Class**。要指定结果类，请将 **resultClass** 设置为所需的 **Class**

- **resultSetMapping**: 默认值: JPA 持续性提供程序假设原生 SQL 查询中的 SELECT 语句: 返回一个类型的实体; 包括与返回的实体的所有字段或属性相对应的所有列; 并使用与字段或属性名称 (未使用 AS 语句) 相对应的列名。要控制 JPA 持续性提供程序如何将 JDBC 结果集映射到实体字段或属性以及标量, 请通过将 **resultSetMapping** 设置为所需的 **@SqlResultSetMapping** 的 String 名称来指定结果集映射

Java 代码 

```
1. //定义一个使用基础数据库的原生 SQL 的查询
2. @Entity
3. @NamedNativeQuery(name="findAllEmployees",query="SELECT * FROM EMPLOYEE")
4. public class Employee implements Serializable {
5.
6. }
7.
8. //Hibernate 如何使用 EntityManager 获取此查询以及如何通过 Query 方法 getResultList
   执行该查询
9. Query queryEmployees = em.createNamedQuery("findAllEmployees");
10. Collection employees = queryEmployees.getResultList();
```


24 @NamedNativeQueries

如果需要指定多个 **@NamedNativeQuery**, 则必须使用一个 **@NamedNativeQueries** 指定所有命名查询

属性说明:

- **value**: 要指定两个或更多属性覆盖, 请将 **value** 设置为 **NamedNativeQuery** 实例数组

下面代码显示了如何使用此批注指定两个命名原生查询

Java 代码 

```
1. @Entity
2. @NamedNativeQueries({@NamedNativeQuery(name="findAllPartTimeEmployees",
3.                                     query="SELECT * FROM EMPLOYEE WHERE PRT_TIME=1"),
4.                     @NamedNativeQuery(name="findAllSeasonalEmployees",
5.                                     query="SELECT * FROM EMPLOYEE WHERE SEASON=1")})
6.
7. public class PartTimeEmployee extends Employee {
8.
9. }
```


25 @OneToMany 和 @ManyToOne

属性说明：

cascade	默认值 CascadeType 的空数组。默认情况下，JPA 不会将任何持久化操作层叠到关联的目标。如果希望某些或所有持久化操作层叠到关联的目标，应将 cascade 设置为一个或多个 CascadeType 类型的枚举值，其中包括：● ALL - 针对拥有实体执行的任何持久化操作均层叠到关联的目标。● MERGE - 如果合并了拥有实体，则将 merge 层叠到关联的目标。● PERSIST - 如果持久保存拥有实体，则将 persist 层叠到关联的目标。● REFRESH - 如果刷新了拥有实体，则 refresh 为关联的层叠目标。● REMOVE - 如果删除了拥有实体，则还删除关联的目标。
fetch	在 Hibernate 里用时默认值：FetchType.LAZY，它要求程序运行时延迟加载所有的集合和实体。如果这不适合于应用程序或特定的持久字段，将 fetch 设置为 FetchType.EAGER，它提示程序在首次访问数据时应马上加载所有的集合和实体
mappedBy	默认值：如果关系是单向的，则该关联提供程序确定拥有该关系的字段。如果关系是双向的，则将关联相反（非拥有）方上的 mappedBy 元素设置为拥有此关系的字段或属性的名称
targetEntity	默认值：使用一般参数定义的 Collection 的参数化类型。默认情况下，如果使用通过一般参数定义的 Collection，则程序将从被引用的对象类型推断出关联的目标实体。如果 Collection 不使用一般参数，则必须指定作为关联目标的实体类：将关联拥有方上的 targetEntity 元素设置为作为关系目标的实体的 Class

26 @OneToOne

属性比 @OneToMany 多一个：

optionalde	默认值：true。默认情况下，JPA 持久化程序假设所有（非基元）字段和属性的值可以为空。如果这并不适合于您的应用程序，请将 optional 设置为 false
------------	---

27 @OrderBy

一般将 @OrderBy 与 @OneToMany 和 @ManyToMany 一起使用

在一对多，多对多关系中，有时我们希望从数据库加载出来的集合对象是按一定方式排序的，这可以通过 OrderBy 来实现，默认是按对象的主键升序排列。


属性说明：

- **value:** 字符串类型，指定排序方式。

格式为"fieldName1 [ASC|DESC],fieldName2 [ASC|DESC],....."

排序类型可以不指定，默认是 ASC 升序。

下面的代码说明 **Person** 和 **Book** 之间是一对多关系。集合 **books** 按照 **Book** 的 **isbn** 升序，**name** 降序排列。

Java 代码 

```
1. @Table(name = "MAPKEY_PERSON")
2. public class Person {
3.     @OneToMany(targetEntity = Book.class, cascade = CascadeType.ALL, mapped
        By = "person")
4.     @OrderBy(name = "isbn ASC, name DESC")
5.     private List books = new ArrayList();
6. }
7.
8. @Entity
9. public class Project {
10.     @ManyToMany
11.     @OrderBy("lastname ASC", "seniority DESC")
12.     public List<Employee> getEmployees() {
13.         ...
14.     }
15. }
16.
17. @Entity
18. public class Employee {
19.     @Id
20.     private int empId;
21.
22.     private String lastname;
23.
24.     private int seniority;
25.
26.     @ManyToMany(mappedBy="employees")
27.     // By default, returns a List in ascending order by empId
28.     public List<Project> getProjects() {
29.         ...
30.     }
31. }
```