参考：<https://github.com/jcalaz/tip/blob/master/blog/javase/%E4%B8%AA%E4%BA%BAJPA%E5%8F%82%E8%80%83%E6%89%8B%E5%86%8C.md>

个人的JPA参考手册，尚未整理完毕，会更新

githu地址：<https://github.com/jcalaz/tip>

**1. JPA普通注解**

* **@Entity**: 声明为一个实体。**(修饰实体类)**
* **@Table**: 指定实体所映射的表。**(修饰实体类)**

| **属性** | **是否必要** | **说明** |
| --- | --- | --- |
| name | 否 | 设置实体映射的表名。不指定则与实体类的类名相同 |
| catalog | 否 | 设置实体映射的表放入指定的catalog中。不指定则放入默认的catalog中 |
| schema | 否 | 设置实体映射的表放入指定的schema中。 不指定则放入默认的schema中 |
| uniqueConstraints | 否 | 为实体映射的表设置唯一的约束。该属性可以是一个@UniqueConstraint Annotation数组 |

* **@Indexed**：定义索引

@Entity

@Table(name="person\_table", indexes= {

@Index(name="idx\_person", columnList="name")

})

@NamedQuery(

name="simpleByTest",

query="SELECT x FROM SimpleModel x WHERE x.test LIKE :test"

)

publicclassPerson {

@Id

@GeneratedValue

private Long id;

private String name;

//...

}

* **@secondaryTable**: 把实体的部分属性映射到第二个数据表。可通过@secondaryTable指定多个额外的数据表。**(修饰实体类)**

| **属性** | **是否必要** | **说明** |
| --- | --- | --- |
| name | 否 | 指定新数据表的表名 |
| catalog | 否 | 设置实体映射的表放入指定的catalog中。不指定则放入默认的catalog中 |
| schema | 否 | 设置实体映射的表放入指定的schema中。 不指定则放入默认的schema中 |
| uniqueConstraints | 否 | 为实体映射的表设置唯一的约束。 |
| pkJoinColumns | 否 | 指定新数据表的一个或多个外键列，只有通过该外键列才可让新数据表中的记录参照到主表记录。该属性值是一个@PrimaryKeyJoinColumn数组 |

* **@PrimaryKeyJoinColumn**: 用于定义在从表中定义的外键列的映射信息。 | 属性 | 是否必要 | 说明 | | :-------------- | :------------ | :------------ | | name | 否 | 指定从表中的外键列的列名 | | columnDefinition | 否 | 指定JPA使用该属性值指定的SQL片段来创建外键列 | | referencedColumnName| 否 | 指定从表中外键参照的数据列的列名 |
* @Entity
* @Table(name="person\_table")
* @SecondaryTable(name="person\_detail",pkJoinColumns=@PrimaryKeyJoinColumn(name="person\_id"))
* public class Person{
* @Id
* private int id;
* @Column(name="person\_name",length=50)
* private String name;
* @Column(table="person\_detail",name="email")
* private String email;
* @Column(table="person\_detail",name="phone")
* private String phone;
* }
* 指定将实体状态放入第二个person\_detail数据表中，并指定email,phone两个属性放入person\_detail数据表中
* **@Column**: 指定属性映射的列信息，如列名，长度等。**(修饰属性)**

| **属性** | **是否必要** | **说明** |
| --- | --- | --- |
| name | 否 | 指定该列的列名。默认为属性名 |
| length | 否 | 指定该列所能保存的数据的最大长度。默认为255 |
| nullable | 否 | 指定该列是否允许为null。默认为true |
| unique | 否 | 指定该列是否具有唯一约束。默认为false |
| updatable | 否 | 指定该列是否包含在JPA生成的update语句的列列表中。默认为true |
| insertable | 否 | 指定该列是否包含在JPA生成的insert语句的列列表中。默认为true |

* **@Id**和\*\*@GeneratedValue\*\*：映射实体类的主键。**(修饰属性)** @Id定义主键，可以是基本类型，基本类型的包装类，String，Date等类型。 @GeneratedValue：设置自动生成属性值，属性如下：

| **属性** | **是否必要** | **说明** |
| --- | --- | --- |
| strategy | 否 | 使用怎样的主键生成策略。GenerationType.AUTO：JPA自动选择最适合底层数据库的主键生成策略; GenerationType.IDENTITY: 对于mysql，sqlserver这样的数据库选择自增长的主键生成策略；GenerationType.SEQUENCE: 对于oracle这样的数据库，选择使用基于sequence的主键生成策略，应与@SequenceGenderator一起使用；GenerationType.TABLE：使用一个辅助表来生成主键，应与@TableGenderator一起使用 |
| generator | 否 | 当使用GenerationType.SEQUENCE,GenerationType.TABLE主键生成策略时，该属性指定sequence,辅助表的名称 |

* **@Transient**: 修饰不想持久保存的属性。**(修饰属性)**
* **@Enumerated**：修饰枚举类型。**(修饰属性)**

当@Enumerated的value属性为EnumType.STRING时，底层数据库保存的是枚举值的名称；

当@Enumerated的value属性为EnumType.ORDINAL时，保存枚举值的序号。

如@Enumerated(EnumType.ORDINAL).

* **@Lob**:修饰大数据类型，对应JDBC的java.sql.Clob类型或者java.sql.Blob类型。**(修饰属性)**

当修饰的属性为byte[],Byte[],java.io.Serializable类型时，将映射为数据库底层的Blob列；

当修饰的属性为char[],Character[]或java.lang.String类型时，映射为底层的Clob列。

* **@Basic**：用于延迟加载操作。**(修饰属性)**

比如JPA加载Person实体时并不需要立即加载它的pic属性，而只加载一个"虚拟的"代理，真正需要pic属性再从数据库加载。

@Basic可以指定的属性：

fetch:指定是否需要延迟加载该属性。FetchType.EAGER不使用延迟加载，Fetch.LAZY使用延迟加载。

optional：指定映射的数据列是否允许使用null值。

例如：

@Lob

@Basic(fetch=FetchType.LAZY)

private byte[] pic;

* **@Temporal**: 修饰日期类型。**(修饰属性)**

@Temporal可以指定一个value属性，

该属性支持Temporal.DATE,Temporal.TIME,Temporal.TIMESTAMP，

分别对应于数据库date,time,timestamp类型的数据列。

* **@Embedded**和\*\*@Embeddable\*\*: 映射复合类型。@Embeadded修饰这个复合类型属性，@Embeaddable修饰这个复合类。 @AttributeOverride用来指定复合类型的成员属性的映射配置，它支持的属性：

| **属性** | **是否必要** | **说明** |
| --- | --- | --- |
| name | 是 | 指定对复合类型中哪个属性进行配置 |
| column | 是 | 指定该属性所映射的数据列的列名 |

@Entity

@Table(name="person\_table")

public class Person{

@Id

private int id;

@Column(name="person\_name",length=50)

private String name;

@Embedded

@AttributeOverrides({

@AttributeOverride(name="name",column=@Column(name="cat\_name",length=35)),

@AttributeOverride(name="color",column=@Column("cat\_color"))

})

private Cat cat;

}

@Embeddable

public class Cat{

private String name;

private String color;

}

* **@IdClass**和\*\*@EmbeddedId\*\*: 定义复合类型的主键。**(修饰属性)** 定义复合类型的主键有两种方式： (1). 使用@IdClass和多个@Id; (2). 使用一个@EmbeddedId即可。

方式一：使用@IdClass和多个@Id

@Entity

@Table(name="person\_table")

@IdClass(Cat.class)

public class Person{

//两个@Id定义联合主键

@Id

private int id;

@Id

private String name;

private Cat cat;

}

方式二：用一个@EmbeddedId

@Entity

@Table(name="person\_table")

public class Person{

@Id

private int id;

@Column(name="person\_name",length=50)

private String name;

@EmbeddedId

@Embedded

@AttributeOverrides({

@AttributeOverride(name="name",column=@Column(name="cat\_name",length=35)),

@AttributeOverride(name="color",column=@Column("cat\_color"))

})

private Cat cat;

}

@Embeddable

public class Cat{

private String name;

private String color;

}

* **@OrderBy**: 对关联实体进行排序

//Example 1:

@Entity

publicclassCourse {

@ManyToMany

@OrderBy("lastname ASC")

publicList<Student>getStudents() {};

}

//Example 2:

@Entity

publicclassStudent {

@ManyToMany(mappedBy="students")

@OrderBy// ordering by primary key is assumed

publicList<Course>getCourses() {};

}

//Example 3:

@Entity

publicclassPerson {

@ElementCollection

@OrderBy("zipcode.zip, zipcode.plusFour")

publicSet<Address>getResidences() {};

}

@Embeddable

publicclassAddress {

protected String street;

protected String city;

protected String state;

@Embeddedprotected Zipcode zipcode;

}

@Embeddable

publicclassZipcode {

protected String zip;

protected String plusFour;

}

* **@JoinColumn**: 定义外键。**(修饰属性)**

| **属性** | **是否必要** | **说明** |
| --- | --- | --- |
| columnDefinition | 否 | 指定JPA使用该属性值指定的SQL片段来创建外键列 |
| name | 否 | 指定该外键列的列名 |
| insertable | 否 | 指定该列是否包含在JPA生成的insert语句的列列表中。默认为true |
| updatable | 否 | 指定该列是否包含在JPA生成的update语句的列列表中。默认为true |
| nullable | 否 | 指定该列是否允许为null。默认为true |
| table | 否 | 指定该列所在的数据表的表名 |
| unique | 否 | 指定是否为该列增加唯一约束 |
| referenceColumnName | 否 | 指定该列所参照的主键列的列名 |

* **@ManyToOne**: 映射多对一关系。**(修饰属性)**

| **属性** | **是否必要** | **说明** |
| --- | --- | --- |
| cascade | 否 | 指定JPA对关联实体采用怎样的级联策略，该级联策略支持四个属性值。CascadeType.ALL：指定JPA将所有的持久化操作都级联到关联实体；CascadeType.MERGE: 指定JPA将merge操作都级联到关联实体；CascadeType.PERSIST：指定JPA将persist操作级联到关联实体；CascadeType.REFRESH: 指定JPA将refresh操作级联到关联实体；CascadeType.REMOVE: 指定JPA将remove操作关联到关联实体 |
| fetch | 否 | 指定抓取关联实体时抓取策略，该属性支持两个值。FetchType.EAGER: 抓取实体时，立即抓取关联实体，默认值；FetchType.LAZY：抓取实体时延迟抓取关联实体，等到真到用到时再去抓取。 |
| optional | 否 | 该属性指定关联关系是否可选。 |
| targetEntity | 否 | 该属性指定关联实体的类名。 在默认情况下，JPA通过反射判断 |

* **@OneToOne**: 映射一对一关系。**(修饰属性)**

| **属性** | **是否必要** | **说明** |
| --- | --- | --- |
| cascade | 否 | 指定JPA对关联实体采用怎样的级联策略，该级联策略支持四个属性值。CascadeType.ALL：指定JPA将所有的持久化操作都级联到关联实体；CascadeType.MERGE: 指定JPA将merge操作都级联到关联实体；CascadeType.PERSIST：指定JPA将persist操作级联到关联实体；CascadeType.REFRESH: 指定JPA将refresh操作级联到关联实体；CascadeType.REMOVE: 指定JPA将remove操作关联到关联实体 |
| fetch | 否 | 指定抓取关联实体时抓取策略，该属性支持两个值。FetchType.EAGER: 抓取实体时，立即抓取关联实体，默认值；FetchType.LAZY：抓取实体时延迟抓取关联实体，等到真到用到时再去抓取。 |
| optional | 否 | 该属性指定关联关系是否可选。 |
| targetEntity | 否 | 该属性指定关联实体的类名。 在默认情况下，JPA通过反射判断 |
| mappedBy | 否 | 该属性合法的属性值为关联实体的属性名，该属性指定关联实体中哪一个属性可引用到关联实体时采取抓取。 |

* **@OneToMany**：映射一对多关系。**(修饰属性)**

| **属性** | **是否必要** | **说明** |
| --- | --- | --- |
| cascade | 否 | 指定JPA对关联实体采用怎样的级联策略，该级联策略支持四个属性值。CascadeType.ALL：指定JPA将所有的持久化操作都级联到关联实体；CascadeType.MERGE: 指定JPA将merge操作都级联到关联实体；CascadeType.PERSIST：指定JPA将persist操作级联到关联实体；CascadeType.REFRESH: 指定JPA将refresh操作级联到关联实体；CascadeType.REMOVE: 指定JPA将remove操作关联到关联实体 |
| fetch | 否 | 指定抓取关联实体时抓取策略，该属性支持两个值。FetchType.EAGER: 抓取实体时，立即抓取关联实体，默认值；FetchType.LAZY：抓取实体时延迟抓取关联实体，等到真到用到时再去抓取。 |
| targetEntity | 否 | 该属性指定关联实体的类名。 在默认情况下，JPA通过反射判断 |
| mappedBy | 否 | 该属性合法的属性值为关联实体的属性名，该属性指定关联实体中哪一个属性可引用到关联实体时采取抓取。 |

* **@ManyToMany**：映射多对多关系。**(修饰属性)**

| **属性** | **是否必要** | **说明** |
| --- | --- | --- |
| cascade | 否 | 指定JPA对关联实体采用怎样的级联策略，该级联策略支持四个属性值。CascadeType.ALL：指定JPA将所有的持久化操作都级联到关联实体；CascadeType.MERGE: 指定JPA将merge操作都级联到关联实体；CascadeType.PERSIST：指定JPA将persist操作级联到关联实体；CascadeType.REFRESH: 指定JPA将refresh操作级联到关联实体；CascadeType.REMOVE: 指定JPA将remove操作关联到关联实体 |
| fetch | 否 | 指定抓取关联实体时抓取策略，该属性支持两个值。FetchType.EAGER: 抓取实体时，立即抓取关联实体，默认值；FetchType.LAZY：抓取实体时延迟抓取关联实体，等到真到用到时再去抓取。 |
| targetEntity | 否 | 该属性指定关联实体的类名。 在默认情况下，JPA通过反射判断 |
| mappedBy | 否 | 该属性合法的属性值为关联实体的属性名，该属性指定关联实体中哪一个属性可引用到关联实体时采取抓取。 |

* **@JoinTable**：专门用于多对多关联关系指定连接表的配置信息。

| **属性** | **是否必要** | **说明** |
| --- | --- | --- |
| name | 否 | 指定该连接表的表名 |
| catalog | 否 | 设置将该连接表放入指定的catalog内。如果没有指定该属性，连接表放入默认的catalog中。 |
| schema | 否 | 设置将该连接表放入指定的schema内。 如果没有指定该属性，连接表放入默认的schema中。 |
| joinColumns | 否 | 该属性值可接受多个@JoinColumn，用于配置连接表中外键列的列信息，这些列参照当前实体对应表的主键列 |
| inverseJoinColumns | 否 | 该属性值可接受多个@JoinColumn，用于配置连接表中外键列的列信息，这些列参照当前实体的关联实体对应表的主键列 |
| uniqueConstraints | 否 | 该属性为连接表增加唯一约束。 |

* **@MapKey**: 使用Map集合记录关联实体。
* **MappedSuperClass**: 映射为非实体父类，该实体父类不会生成对应的数据表

@MappedSuperclass

publicclassEmployee {

@Idprotected Integer empId;

@Versionprotected Integer version;

@ManyToOne@JoinColumn(name="ADDR")

protected Address address;

public Integer getEmpId() { }

publicvoidsetEmpId(Integer id) { }

public Address getAddress() { }

publicvoidsetAddress(Address addr) { }

}

// Default table is FTEMPLOYEE table

@Entity

publicclassFTEmployeeextendsEmployee {

// Inherited empId field mapped to FTEMPLOYEE.EMPID

// Inherited version field mapped to FTEMPLOYEE.VERSION

// Inherited address field mapped to FTEMPLOYEE.ADDR fk

// Defaults to FTEMPLOYEE.SALARY

protected Integer salary;

publicFTEmployee() {}

public Integer getSalary() { }

publicvoidsetSalary(Integer salary) { }

}

@Entity@Table(name="PT\_EMP")

@AssociationOverride(

name="address",

joincolumns=@JoinColumn(name="ADDR\_ID"))

publicclassPartTimeEmployeeextendsEmployee {

// Inherited empId field mapped to PT\_EMP.EMPID

// Inherited version field mapped to PT\_EMP.VERSION

// address field mapping overridden to PT\_EMP.ADDR\_ID fk

@Column(name="WAGE")

protected Float hourlyWage;

publicPartTimeEmployee() {}

public Float getHourlyWage() {}

publicvoidsetHourlyWage(Float wage) {}

}

* **@Inheritance**：指定映射策略 InheritanceType.SINGLE\_TABLE：整个类层次对应一张表策略,这是继承映射的默认策略。 InheritanceType.JOINED：连接子类策略。父亲的放在一张表，儿子只是保存和父亲不一样的，增加的属性。 InheritanceType.TABLE\_PER\_CLASS：每个具体的类一个表的策略。
* **@DiscriminatorColumn**:在整个类层次对应一张表策略的映射策略中配置辨别列。

| **属性** | **是否必要** | **说明** |
| --- | --- | --- |
| columnDefinition | 否 | 指定JPA使用该属性值指定的SQL片段来创建外键列 |
| name | 否 | 指定辨别列的名称，默认值为"DTYPE" |
| discriminatorType | 否 | 指定该辨别者列的数据类型。 DiscriminatorType.CHAR: 辨别者列的类型是字符类型，即该列只接受单个字符；DiscriminatorType.INTEGER：辨别者列的类型是整数类型，即该列只接受整数值；DiscriminatorType.STRING：辨别者列的类型是字符串类型，即该列只接受字符串值，为默认值 |
| length | 否 | 该属性指定辨别者的字符长度 |

**2. JPA生命周期注解**

* **@PerPersist**：保存实体之前回调它修饰的方法。
* **@PostPersist**：保存实体之后回调它修饰的方法。
* **@PreRemove**：删除实体之前回调它修饰的方法。
* **@PostRemove**：删除实体之后回调它修饰的方法。
* **@PreUpdate**：更新实体之前回调它修饰的方法。
* **@PostUpdate**：更新实体之后回调它修饰的方法。
* **@PostLoad**：记载实体之后回调它修饰的方法。
* **@EntityListeners**: 自定义专门的监听器

@Entity

@EntityListeners(PersonListener.class)

publicclassPersonimplementsSerializable{}

* **@ExcludeDefaultListeners和@ExcludeSuperclassListeners**：排除监听器。

**3. 关联**

* **单向N-1关联**：使用\*\*@ManyToOne\*\*注解。比如一个人对应多个手机号,仅通过手机号获取用户，无需获取用户的手机号的场景。 **当使用@JoinColumn通过外键实现，否则通过第三方表实现。**

@Entity(name="Person")

publicstaticclassPerson {

@Id

@GeneratedValue

private Long id;

publicPerson() {

}

}

@Entity(name="Phone")

publicstaticclassPhone {

@Id

@GeneratedValue

private Long id;

@Column(name="`number`")

private String number;

@ManyToOne(optional=false,cascade=CascadeType.ALL,fetch=FetchType.LAZY,targetEntity=Person.class)

@JoinColumn(name="person\_id",

foreignKey=@ForeignKey(name="PERSON\_ID\_FK")

)

private Person person;

publicPhone() {

}

publicPhone(String number) {

this.number = number;

}

public Long getId() {

return id;

}

public String getNumber() {

return number;

}

public Person getPerson() {

return person;

}

publicvoidsetPerson(Person person) {

this.person = person;

}

}

对应的sql语句：

CREATETABLEPerson (

id BIGINTNOT NULL ,

PRIMARY KEY ( id )

)

CREATETABLEPhone (

id BIGINTNOT NULL ,

numberVARCHAR(255) ,

person\_id BIGINT ,

PRIMARY KEY ( id )

)

ALTERTABLE Phone

ADD CONSTRAINT PERSON\_ID\_FK

FOREIGN KEY (person\_id) REFERENCES Person

* **单向1-1关联**: 使用\*\*@OneToOne\*\*注解。比如一个人对应一个身份证Id,只需获取一个人的身份证号，而无需通过身份证号获取用户的情况。 **当使用@JoinColumn通过外键实现，否则通过第三方表实现。**

@Entity

@Table(name="person\_table")

publicclassPerson{

@Id

privateint personid;

private String name;

@OneToOne(optional=false,cascade=CascadeType.ALL,fetch=FetchType.LAZY,targetEntity=IdCard.class)

@JoinColumn(name="id\_card\_id",nullable=false,updatable=false)//映射外键列

private IdCard idCard;

}

@Entity

@Table(name="id\_card\_table")

publicclassIdCard{

@Id

privateint idCardId;

private String cardNumber;

}

* **单向1-N关联**：使用\*\*@OneToMany\*\*注解。 -**对于1-N关联，应尽量设计为双向关联，而不是单向** 比如一个人有多个手机号，仅需要获取一个人的手机号，而无需通过手机号获取用户的场景。 **当使**
* **用@JoinColumn通过外键实现，否则通过第三方表实现。**

**//=======20170731**

@Entity(name="Person")

publicstaticclassPerson {

@Id

@GeneratedValue

private Long id;

@OneToMany(cascade= CascadeType.ALL, orphanRemoval=true)

privateList<Phone> phones =newArrayList<>();

publicPerson() {

}

publicList<Phone>getPhones() {

return phones;

}

}

@Entity(name="Phone")

publicstaticclassPhone {

@Id

@GeneratedValue

private Long id;

@Column(name="`number`")

private String number;

publicPhone() {

}

publicPhone(String number) {

this.number = number;

}

public Long getId() {

return id;

}

public String getNumber() {

return number;

}

}

对应的sql：

CREATETABLEPerson (

id BIGINTNOT NULL ,

PRIMARY KEY ( id )

)

CREATETABLEPerson\_Phone (

Person\_id BIGINTNOT NULL ,

phones\_id BIGINTNOT NULL

)

CREATETABLEPhone (

id BIGINTNOT NULL ,

numberVARCHAR(255) ,

PRIMARY KEY ( id )

)

ALTERTABLE Person\_Phone

ADD CONSTRAINT UK\_9uhc5itwc9h5gcng944pcaslf

UNIQUE (phones\_id);

ALTERTABLE Person\_Phone

ADD CONSTRAINT FKr38us2n8g5p9rj0b494sd3391

FOREIGN KEY (phones\_id) REFERENCES Phone;

ALTERTABLE Person\_Phone

ADD CONSTRAINT FK2ex4e4p7w1cj310kg2woisjl2

FOREIGN KEY (Person\_id) REFERENCES Person

* **单向N-N关联**：使用\*\*@ManyToMany\*\*注解。比如一个人有多个住址，一个住址又对应多个用户，仅需通过用户获取住址列表的场景。 **对于多对多关系，数据库底层只能通过关联表实现。** 方式一：使用默认

@Entity(name="Person")

publicstaticclassPerson {

@Id

@GeneratedValue

private Long id;

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

privateList<Address> addresses =newArrayList<>();

publicPerson() {

}

publicList<Address>getAddresses() {

return addresses;

}

}

@Entity(name="Address")

publicstaticclassAddress {

@Id

@GeneratedValue

private Long id;

private String street;

@Column(name="`number`")

private String number;

publicAddress() {

}

publicAddress(String street, String number) {

this.street = street;

this.number = number;

}

public Long getId() {

return id;

}

public String getStreet() {

return street;

}

public String getNumber() {

return number;

}

}

对应的sql：

CREATETABLEAddress (

id BIGINTNOT NULL ,

numberVARCHAR(255) ,

street VARCHAR(255) ,

PRIMARY KEY ( id )

)

CREATETABLEPerson (

id BIGINTNOT NULL ,

PRIMARY KEY ( id )

)

CREATETABLEPerson\_Address (

Person\_id BIGINTNOT NULL ,

addresses\_id BIGINTNOT NULL

)

ALTERTABLE Person\_Address

ADD CONSTRAINT FKm7j0bnabh2yr0pe99il1d066u

FOREIGN KEY (addresses\_id) REFERENCES Address;

ALTERTABLE Person\_Address

ADD CONSTRAINT FKba7rc9qe2vh44u93u0p2auwti

FOREIGN KEY (Person\_id) REFERENCES Person

方式二：**通过@JoinTable配置关联表**

@Entity(name="Person")

publicstaticclassPerson {

@Id

@GeneratedValue

private Long id;

@ManyToMany(cascade= {CascadeType.PERSIST, CascadeType.MERGE},targetEntity=Address.class)

@JoinTable(

name="person\_address",

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

inverseJoinTableColumns=@JoinColumn(name="address\_id")

)

privateList<Address> addresses =newArrayList<>();

publicPerson() {

}

publicList<Address>getAddresses() {

return addresses;

}

}

* **双向1-1关联**: 使用**两边@OneToOne**注解和**mappedBy**属性双向需要两边实体类都增加@OneToOne，可在一边的实体类增加mappedBy属性。当使用mappedBy属性后表示当前实体不再控制关联联系，因此不可使用@JoinColumn。比如一个人一个精确住址，既可以通过用户获取住址，又可以通过住址获取该住户的场景。

@Entity

@Table(name="person\_table")

publicclassPerson{

@Id

@GeneratedValue(strategy=GenerationType.IDENTITY)

privateint personId;

private String name;

privateint age;

@OneToOne(mappedBy="person",cascade=CascadeType.ALL)

private Address address;

//...

}

@Entity

@Table(name="address\_table")

publicclassAddress{

@Id

privateint addressId;

private String detail;

@OneToOne(optional=false,cascade=CascadeType.ALL)

@JoinColumn(name="person\_id",nullable=false,updatable=false)

private Person person;

//...

}

* **双向1-N关联**：使用\*\*@OneToMany**和**@ManyToOne**注解和**mappedBy\*\*属性 **对于1-N关联，应尽量设计为双向关联，而不是单向，并且尽量使用N的一端来控制关联。** 1的一端使用@OneToMany注解和mappedBy属性，N的一端使用@ManyToOne和@JoinColumn。比如一个人有多个住址，既可以通过用户获取住址，又可以通过住址获取用户的场景。

@Entity

@Table(name="person\_table")

publicclassPerson{

@Id

@GeneratedValue(strategy=GenerationType.IDENTITY)

privateint personId;

private String name;

privateint age;

@OneToMany(mappedBy="person",cascade=CascadeType.ALL)

privateSet<Address> addresses=newHashSet<Address>();

//...

}

@Entity

@Table(name="address\_table")

publicclassAddress{

@Id

privateint addressId;

private String detail;

@ManyToOne(optional=false,cascade=CascadeType.ALL)

@JoinColumn(name="person\_id",nullable=true)

private Person person;

//...

}

* **双向N-N关联**: 使用**两边@ManyToMany**注解，**一边mapperBy**属性和。对于N-N关联，底层数据库必须通过关联表来关联实体之间的关系。对于双向N-N关联，两边实体对等，一边通过mappedBy不再控制关系，另一边通过@JoinTable控制关系即可。比如多个人住在同一个地址，但一个人也可有多个住址，既可以通过用户找到住址列表，又可以通过住址找到用户列表的场景。

@Entity

@Table(name="person\_table")

publicclassPerson{

@Id

@GeneratedValue(strategy=GenerationType.IDENTITY)

privateint personId;

private String name;

privateint age;

@ManyToMany(mappedBy="persons",cascade=CascadeType.ALL)

privateSet<Address> addresses=newHashSet<Address>();

//...

}

@Entity

@Table(name="address\_table")

publicclassAddress{

@Id

privateint addressId;

private String detail;

@ManyToMany(optional=false,cascade=CascadeType.ALL)

@JoinColumn(name="person\_id",nullable=true)

@JoinTable(

name="person\_address",

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

inverseJoinTableColumns=@JoinColumn(name="person\_id")

)

privateSet<Person> persons=newHashSet<Person>();

//...

}

* 使用Map集合记录关联实体：使用\*\*@MapKey\*\*注解：比如一个人有多个住址，既可以通过用户获取住址，又可以通过住址获取用户的场景。使用@MapKey时必须指定一个name属性，name属性的属性值为当前实体的关联实体中标识属性的属性名。

@Entity

@Table(name="person\_table")

publicclassPerson{

@Id

@GeneratedValue(strategy=GenerationType.IDENTITY)

privateint personId;

private String name;

privateint age;

@OneToMany(mappedBy="person",cascade=CascadeType.ALL)

@MapKey(name="pk")

privateMap<AddressPk,Address> addresses=newHashMap<AddressPk,Address>();

//...

}

**4. JPA映射策略**

JPA提供了3种映射策略：

(1)、整个类层次对应一张表策略,这是继承映射的默认策略。

即如果实体类B继承实体类A，实体类C也继承自实体A，那么只会映射成一个表，

这个表中包括了实体类A、B、C中所有的字段，JPA使用一个叫做“discriminator列”来区分某一行数据是应该映射成哪个实体。

注解为：@Inheritance(strategy = InheritanceType.SINGLE\_TABLE)

(2)、连接子类策略。父亲的放在一张表，儿子只是保存和父亲不一样的，增加的属性。

这种情况下子类的字段被映射到各自的表中，这些字段包括父类中的字段，并执行一个join操作来实例化子类。

注解为：@Inheritance(strategy = InheritanceType.JOINED)

(3)、每个具体的类一个表的策略。

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

可使用@Inheritance指定映射策略

InheritanceType.SINGLE\_TABLE：第一种

InheritanceType.JOINED：第二种

InheritanceType.TABLE\_PER\_CLASS：第三种

* **整个类层次对应一张表策略**。这种策略下，整个类层次所有的实体都存放在一张数据表中，系统通过在该表增加额外的一个辨别列，用来区分每行记录到底是哪一个类的实例。使用@DiscriminatorColumn来配置辨别列。

@Entity

@Inheritance(strategy=InheritanceType.SINGLE\_TABLE)

// 定义辨别者列的列名为person\_type，列类型为字符串

@DiscriminatorColumn(name="person\_type" ,

discriminatorType=DiscriminatorType.STRING)

// 指定Person实体对应的记录在辨别者列的值为"普通人"

@DiscriminatorValue("普通人")

@Table(name="person\_inf")

publicclassPerson{}

// 顾客类继承了Person类

@Entity

// 指定Customer实体对应的记录在辨别者列的值为"顾客"

@DiscriminatorValue("顾客")

@Table(name="customer\_inf")

publicclassCustomerextendsPerson{}

// 员工类继承了Person类

@Entity

// 指定Employee实体对应的记录在辨别者列的值为"员工"

@DiscriminatorValue("员工")

@Table(name="employee\_inf")

publicclassEmployeeextendsPerson{}

* **连接子类的映射策略** 这种策略中父类实体保存在父类表中，而子类实体由父亲表和子类表共同存储，父类和子类共有部分存储在父类表，子类单独存在属性存储在子类表中。无需使用辨别者，只需要在继承树的根实体类上使用@Inheritance,指定strategy=InheritanceType.JOINED即可

@Entity

@Inheritance(strategy=InheritanceType.JOINED)

@Table(name="person\_inf")

publicclassPerson{}

@Entity

@Table(name="customer\_inf")

publicclassCustomerextendsPerson{}

@Entity

@Table(name="employee\_inf")

publicclassEmployeeextendsPerson{}

* **每个具体的类一个表的策略** 子类实例仅保存在子类表中，在父类表中没有任何记录。单从数据库来看，几乎难以看出继承关系，只是多个实体之间主键存在某种连续性，因此不能让数据库自动生成主键，因此**不能使用GenerationType.IDENTITY和GenerationType.AUTO这两种主键生成策略**。无需使用辨别者，只需要在继承树的根实体类上使用@Inheritance,指定strategy=InheritanceType.TABLE\_PER\_CLASS即可

@Entity

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

@Table(name="person\_inf")

publicclassPerson{}

**5. spring data注解**

* **@CreatedBy**：Declares a field as the one representing the principal that created the entity containing the field.
* **@CreatedDate**：Declares a field as the one representing the date the entity containing the field was created.
* **@Id**：Demarcates an identifier.
* **@LastModifiedBy**：Declares a field as the one representing the principal that recently modified the entity containing the field.
* **@LastModifiedDate**：Declares a field as the one representing the date the entity containing the field was recently modified.
* **@ReadOnlyProperty**：Marks a field to be read-only for the mapping framework and therefore will not be persisted.
* **@Reference**：Meta-annotation to be used to annotate annotations that mark references to other objects.
* **@Transient**：Marks a field to be transient for the mapping framework. Thus the property will not be persisted and not further inspected by the mapping framework.
* **@TypeAlias**：Annotation to allow String based type aliases to be used when writing type information for PersistentEntitys.
* **@version**: 定义一个属性为版本字段用于实现乐观锁

**参考**

* 《经典JAVAEE企业应用实战》
* JPA API
* spring data commons源码
* <http://blog.csdn.net/u012881904/article/details/51059156>