JPA规范

## 概述

这份文档是关于在Java EE和Java SE中管理持久化和对象/关系映射的Java API的规范。这

项工作的技术目标是提供一个对象/关系映射机制，通过它，Java应用开发者可以使用域模

型来管理关系数据库。

这个持久化API——连同在本文中定义的查询语言和对象/关系映射元数据——是Enterprise

JavaBeans 3.0必须要支持的。它也被定位于在Java SE中独立地使用。

## 实体

### 概述

实体（entity）是一个轻量的持久化的域对象（domain object）。

实体类是编程工作的主要制品。实体类可以使用辅助类，它们起助手类的作用或者被用于表示实体的状态。

### 实体类的需求

实体类必须用Entity注解进行标注或者在XML描述符中将它声明为实体。

实体类必须有一个无参数的构造方法，而且必须为public或protected。在这之外，实体类

可以有其它的构造方法。

实体类必须是一个顶层类（top-level class）。枚举和接口不能作为实体。

实体类不能用final修饰。实体类的所有方法和持久实例变量都不能是final的。

如果一个实体的实例作为脱管对象（detached object）被传值（passed by value，如，通

过远程接口），那么这个实体类必须实现Serializable接口。

实体支持继承，多态关联和多态查询。

抽象类和具体类都可以作为实体。实体类可以扩展非实体类和实体类，非实体类也可以扩展

实体类。

### 主键和实体标识

每个实体都必须要有一个主键。

主键必须定义在实体层次（entity hierarchy）的根（root）实体，或实体层次的映射超类

（mapped superclass）上。一个实体层次只能定义一个主键。

简单（即，非组合的）主键必须对应实体的一个持久化字段或属性。Id注解被用来指示简

单主键。见9.1.8节。

复合主键要么对应单个持久化字段/属性，要么对应一组这样的字段或属性（见下文说明）。

要表示一个复合主键，必须定义一个主键类。复合主键通常出现在从遗留数据库进行的映射

中，这些遗留数据库的主键由多个列构成。EmbeddedId和IdClass注解用于指示复合主键。

见9.1.14和9.1.15节。

主键（或者复合主键中的字段或属性）应为下列类型之一：Java基本类型；基本类型的包

装类；java.lang.String；java.util.Date；java.sql.Date。然而，一般说来，不应该为

主键使用近似数类型（例如，浮点数）。使用其它类型作为实体主键会失去可移植性。如果

使用（自动）生成的主键，则只有整形是可移植的。如果用java.util.Date作为主键字段

或属性，则时态类型（temporal type）应该指定为DATE。

主键类的访问类型（基于字段还是属性）由包含该主键类的实体的访问类型决定。

### 实体关系

实体之间的关系有:一对一,一对多，多对一，多对多，关系是多态的。

如果两个实体之间存在关联，参照实体(referencing entity)上对应的持久化属性或实例变

量上必须应用下列关系建模注解之一：OneToOne, OneToMany, ManyToOne, ManyToMany。

对于未指定目标类型的关联（例如，对集合没有使用Java泛型），则有必要指定作为关系

目标的实体

（译注：如果实体A参照/引用实体B，则实体A为“参照实体”，实体B为“被参照实

体”）

这些Java注解“镜像”了关系数据库建模中的通常做法，用这些注解进

行对象/关系映射完全可以采用缺省的方式，以便快速开发。

关系可以是双向或单向的，双向关系同时具有持有端（owning side）和反向端（inverse side），单向关系只有一个持有端，关系的持有端觉得对数据库的更新。

以下规则应用于双向关系：

双向关系中的反向端必须通过oneToOne，OneToMany或ManyToMany注解的mappedBy元素指向他的持有端，mappedBy元素指出实体中作为关系的主控端的属性/字段

一对多/多对一双向关系中的‘多’端必须是持有端,因此不能再ManyToOne注解中使用mappedBy注解

对于一对一双向关系，包含对应的外间（foreign key）的那一端是持有端。

对于多对多双向关系，任一端都可以作为持有端

关系模型注解限制对cascade=REMOVE的使用。cascade=REMOVE只应该用在OneToOne或

OneToMany关系上，对其它类型的关系应用cascade=REMOVE是不可移植的。

其它映射注解（如，数据库表和列的注解）可以用来覆盖或进一步明确缺省的映射（2.1.8

节中描述缺省映射）。例如，可以对单向一对多映射使用外键映射。这种模式级别（

schema-level）的映射注解必须在关系的持有端指定。任何此类覆盖都必须和已指定的关系

模型注解保持一致。例如，如果指定了多对一的关系，就不允许在关系的外键上指定唯一键

约束。

持久化提供者负责处理关系的对象——关系映射，包括从数据库中加载和写回到数据库（根

据实体类的元数据），以及数据库中指定的关系的参照完整性（如，外键约束）。

（TODO: 原文是斜体、段落缩进）维护运行期关系的完整性是应用程序的责任，例如，当应

用程序在运行期间更新关系时，要确保双向关系的“一”和“多”端相互保持一致。

对于实体的多值关系，如果在数据库中没有获取到相关联的实体。持久化提供者应返回一个

空集合作为关系的值。

### 关系映射

#### 双向OneToOne关系

假设:

实体A引用实体B的单个实例

实体B引用实体A的单个实例

实体A被指定为关系的持有者。

应用以下映射缺省:

实体a被映射到表A

实体B被映射到表B

表A含有一个参照表B的外键，外键的列名是按如下顺序拼接的字符串：实体A的关系属性或字段的名称，‘\_’，表b的主键名，外键的类型和表b的主键类型相同，并且在外键上有个唯一键约束。

例如:

@Entity

public class Employee {

private Cubicle assignedCubicle;

@OneToOne

public Cubicle getAssignedCubicle() {

return assignedCubicle;

}

public void setAssignedCubicle(Cubicle cubicle) {

this.assignedCubicle = cubicle;

}

...

} @

Entity

public class Cubicle {

private Employee residentEmployee;

@OneToOne(mappedBy="assignedCubicle")

public Employee getResidentEmployee() {

return residentEmployee;

}

public void setResidentEmployee(Employee employee) {

this.residentEmployee = employee;

}

...

}

在这个例子中：

实体Employee引用实体Cubicle的单个实例。

实体Cubicle引用实体Employee的单个实例。

实体Employee是关系的持有端。

将应用以下映射缺省：

实体Employee被映射到表EMPLOYEE。

实体Cubicle被映射到表CUBICLE。

表EMPLOYEE中有一个参照CUBICLE表的外键。外键的列名为：ASSIGNEDCUBICLE\_<PK

of CUBICLE>，这里<PK of CUBICLE>是CUBICLE表的主键的名称。外键的类型与CUBICLE表

的主键的类型相同，并且外键上具有唯一约束。

#### 双向ManyToOne/OneToMany关系

假设：

实体a引用实体b的单个实例

实体b引用一个实体a的集合

则实体a必须是关系的持有者

将应用一下映射缺省：

实体a映射到表A

实体b映射到表b

表a含有一个参照表b的外键，外键的列名为按如下次序拼接的字符串：实体a的关系属性/字段的名称,”\_”,表b的主键名，外键的类型和表b的主键类型相同

例子：

@Entity

public class Employee {

private Department department;

@ManyToOne

public Department getDepartment() {

return department;

}

public void setDepartment(Department department) {

this.department = department;

}

...

} @

Entity

public class Department {

private Collection<Employee> employees = new HashSet();

@OneToMany(mappedBy="department")

public Collection<Employee> getEmployees() {

return employees;

}

public void setEmployees(Collection<Employee> employees) {

this.employees = employees;

}

...

}

在这个例子中：

实体Employee引用实体Department的单个实例。

实体Department引用一个实体Employee的集合。

实体Employee是关系的持有者。

应用以下映射缺省：

实体Employee映射到表EMPLOYEE。

实体Department映射到表DEPARTMENT。

表EMPLOYEE包含参照表DEPARTMENT的外键。外键的列名为：DEPARTMENT\_<PK of

DEPARTMENT>，这里<PK of DEPARTMENT>是DEPARTMENT 表的主键的名称。外键的类型与

DEPARTMENT 表的主键的类型相同

#### 单向单值关系

假设：

实体a引用实体b的单个实例

实体b没有引用实体a

单向关系只有一个持有端（没有反向端），在这里必须是实体a，可用单向OneToOne或单向ManyToOne关系来表示单向单值关系模型

##### 单向OneToOne关系

应用以下映射缺省：

实体a映射到表a

实体b映射到表b

表a包含一个到表b的外键

外键的列名为按如下次序拼接的字符串：实体A的关系

属性/字段的名称；“\_”；表B的主键名。外键的类型和表B的主键类型相同，并且在外键

上有唯一键约束。

例子：

@Entity

public class Employee {

private TravelProfile profile;

@OneToOne

public TravelProfile getProfile() {

return profile;

}

public void setProfile(TravelProfile profile) {

this.profile = profile;

}

...

} @

Entity

public class TravelProfile {

...

}

在这个例子中：

实体Employee引用实体TravelProfile的单个实例。

实体TravelProfile没有引用实体Employee。

实体Employee是关系的持有者。

应用以下映射缺省：

实体Employee映射到表EMPLOYEE。

实体TravelProfile映射到表TRAVELPROFILE。

表EMPLOYEE包含一个到表TRAVELPROFILE的外键。外键的列名是PROFILE\_<PK of

TRAVELPROFILE>，这里的<PK of TRAVELPROFILE>是 TRAVELPROFILE表的主键名。外键的类

型与 TRAVELPROFILE表的主键的类型相同，并且在外键上有唯一键约束。

##### 单向ManyToOne关系

应用以下映射缺省： 表A包含一个到表B的外键。 外键的列名为按如下次序拼接的字符串：实体A的关系

属性或字段的名称；“\_”；表B的主键名。外键的类型和表B的主键类型相同。

实体A映射到表A。

实体B映射到表B。

例子：

@Entity

public class Employee {

private Address address;

@ManyToOne

public Address getAddress() {

return address;

}

public void setAddress(Address address) {

this.address = address;

}

...

} @

Entity

public class Address {

...

}

在这个例子中：

实体Employee引用实体Address的单个实例。

实体Address没有引用实体Employee。

实体Employee是关系的持有者。

应用以下的映射缺省：

实体Employee映射到表EMPLOYEE。

实体Address映射到表ADDRESS。

表EMPLOYEE包含一个到表ADDRESS的外键。外键的列名是ADDRESS\_<PK of

ADDRESS>，这里的<PK of ADDRESS>是 ADDRESS表的主键名。外键的类型与 TRAVELPROFILE

表的主键的类型相同。

#### 双向ManyToMany关系

假设：

实体a引用实体b的一个集合

实体b引用实体a的一个集合

实体a是关系的持有者

应用以下映射缺省：

实体A映射到表A。

实体B映射到表B。

有一个连接表(join table),名为A\_B(持有端的名字在前)，这个连接表有俩个外键列，一个外键列指向表a并且与表a的主键类型相同，该外键列的名称是按如下次序拼接的字符串：

实体B的关系属性/字段的名称；“\_”；表A的主键列的名称。另一个外键列

指向表B并且与表B的主键类型相同。该外键列的名称是按如下次序拼接的字符串：实体A

的关系属性或字段的名称；“\_”；表B的主键列的名称。

例子：

@Entity

public class Project {

private Collection<Employee> employees;

@ManyToMany

public Collection<Employee> getEmployees() {

return employees;

}

public void setEmployees(Collection<Employee> employees) {

this.employees = employees;

}

...

} @

Entity

public class Employee {

private Collection<Project> projects;

@ManyToMany(mappedBy="employees")

public Collection<Project> getProjects() {

return projects;

}

public void setProjects(Collection<Project> projects) {

this.projects = projects;

}

...

}

在这个例子中：

实体Project引用实体Employee的一个集合。

实体Employee引用实体Project的一个集合。

实体Project是关系的持有者。

应用以下映射缺省：

实体Project映射到表PROJECT。

实体Employee映射到表EMPLOYEE。

有一个名为PROJECT\_EMPLOYEE的连接表（持有者的名称在前），它有两个外键列。

一个外键列指向表PROJECT并且与PROJECT表的主键类型相同。该外键列的名称是

PROJECT\_<PK of PROJECT>，这里的<PK of PROJECT>是PROJECT表的主键列名。另一个外键

列指向表EMPLOYEE并且与EMPLOYEE表的主键类型相同。该外键列的名称是EMPLOYEE\_<PK

of EMPLOYEE>，这里的<PK of EMPLOYEE>是EMPLOYEE 表的主键列名。

#### 单向多值关系

假设：

实体a引用实体b的一个集合

实体b没有引用实体a

单向关系只有一个持有端（没有反向端），这里必须是a

单向多值关系模型可以用单向OneToMany或单向ManyToMany关系来表示。

单向OneToMany关系

应用以下映射缺省

实体A映射到表A。

实体B映射到表B。

有一个连接表，表名为A\_B（关系的持有者在前）。该连接表有两个外键列。一个外

键列指向表A并且与表A的主键类型相同。该外键列的名称是按如下次序拼接的字符串：实

体A的名称；“\_”；表A的主键列的名称。另一个外键列指向表B并且与表B的主键类型相

同，并且在该外键上有唯一键约束。该外键列的名称是按如下次序拼接的字符串：实体A的

关系属性或字段的名称；“\_”；表B的主键列的名称。

例子：

@Entity

public class Employee {

private Collection<AnnualReview> annualReviews;

@OneToMany

public Collection<AnnualReview> getAnnualReviews() {

return annualReviews;

}

public void setAnnualReviews(Collection<AnnualReview> annualReviews)

{

this.annualReviews = annualReviews;

}

...

} @Entity

public class AnnualReview {

...

}

在这个例子中：

实体Employee引用实体AnnualReview的一个集合。

实体AnnualReview没有引用实体Employee。

实体Employee是关系的持有者。

应用以下映射缺省：

实体Employee映射到表EMPLOYEE。

实体AnnualReview映射到表ANNUALREVIEW。

有一个名为EMPLOYEE\_ANNUALREVIEW（关系持有者在前）的连接表。它有两个外键。

一个外键指向EMPLOYEE表并且与EMPLOYEE表的主键类型相同，该外键的名称为

EMPLOYEE\_<PK of EMPLOYEE>，这里<PK of EMPLOYEE>是EMPLOYEE表的主键列名。另一个外

键指向表ANNUALREVIEW并且与ANNUALREVIEW表的主键类型相同，这个外键的名称为

ANNUALREVIEWS\_<PK of ANNUALREVIEW>，这里<PK of ANNUALREVIEW>是ANNUALREVIEW表的

主键列名。在指向ANNUALREVIEW表的外键上有唯一键约束。

##### 单向ManyToMany关系

映射缺省：

在这个例子中：

实体Employee引用实体AnnualReview的一个集合。

实体AnnualReview没有引用实体Employee。

实体Employee是关系的持有者。

应用以下映射缺省：

实体Employee映射到表EMPLOYEE。

实体AnnualReview映射到表ANNUALREVIEW。

有一个名为EMPLOYEE\_ANNUALREVIEW（关系持有者在前）的连接表。它有两个外键。

一个外键指向EMPLOYEE表并且与EMPLOYEE表的主键类型相同，该外键的名称为

EMPLOYEE\_<PK of EMPLOYEE>，这里<PK of EMPLOYEE>是EMPLOYEE表的主键列名。另一个外

键指向表ANNUALREVIEW并且与ANNUALREVIEW表的主键类型相同，这个外键的名称为

ANNUALREVIEWS\_<PK of ANNUALREVIEW>，这里<PK of ANNUALREVIEW>是ANNUALREVIEW表的

主键列名。在指向ANNUALREVIEW表的外键上有唯一键约束。

例子：

@Entity

public class Employee {

private Collection<Patent> patents;

@ManyToMany

public Collection<Patent> getPatents() {

return patents;

}

public void setPatents(Collection<Patent> patents) {

this.patents = patents;

}

...

} @

Entity

public class Patent {

...

}

在这个例子中：

实体Employee引用实体Patent的一个集合。

实体Patent没有引用实体Employee。

实体Employee是关系的持有者。

应用以下映射缺省：

实体Employee映射到表EMPLOYEE。

实体Patent映射到表PATENT。

有一个名为EMPLOYEE\_PATENT（关系持有者在前）的连接表。它有两个外键。一个外

键指向EMPLOYEE表并且与EMPLOYEE表的主键类型相同，该外键的名称为EMPLOYEE\_<PK of

EMPLOYEE>，这里<PK of EMPLOYEE>是EMPLOYEE表的主键列名。另一个外键指向表PATENT

并且与PATENT表的主键类型相同，这个外键的名称为PATENTS\_<PK of PATENT>，这里<PK

## 对象/关系映射元数据

### 概述

对象/关系映射元数据是应用程序域模型的一部分。它表达了将应用程序的实体和关系映射

到数据库的需求和期望。与应用程序域模型相对应的针对数据库模式编写的查询（以及，特

别是SQL查询），依赖映射元数据所表达出来的映射关系。本规范的实现必须支持应用程序

对这种对象/关系映射元数据的依赖，并遵守映射所表达的需求和语义。

本规范的实现可以支持DDL的生成，但这不是必须的，可移植应用程序不应该依赖DDL生成。

### 对象/关系映射注解

这些注解及类型在javax.persistence中。

XML元数据是注解之外的替选，也可以用来覆盖和补充这些注解

### @column

Column注解指出属性或字段要映射到哪个列。

表7指定可用于Column注解的元素及它们的缺省值。

如果没有Column注解，则应用表7中的缺省值。

#### 源码

@Target({METHOD, FIELD}) @Retention(RUNTIME)

public @interface Column {

String name() default "";

boolean unique() default false;

boolean nullable() default true;

boolean insertable() default true;

boolean updatable() default true;

String columnDefinition() default "";

String table() default "";

int length() default 255;

int precision() default 0; // decimal precision

int scale() default 0; // decimal scale

}

#### 属性列表

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 名称 | 说明 | 缺省 |
| String | Name | 列名 | 属性或字段名称 |
| boolean | unique | （可选）该属性是否  为唯一键。这是表级  的UniqueConstraint  注解的便捷方式，当  唯一键约束只是单个  字段时这种指定方式  很有用。这里的约束  是对主键映射所需的  约束和在表级指定的  约束的补充。 | False |
| boolean | nullable | （可选）数据库列是  否可以为空 | True |
| boolean | insertable | （可选）是否在由持  久化提供者生成的  SQL INSERT语句中包  含此列。 | True |
| boolean | updatable | （可选）是否在由持  久化提供者生成的  SQL UPDATE语句中包  含此列。 | True |
| int | length | （可选）列的长度  （仅用于字符串值的  列） | 255 |

### @ManyToOne注解

ManyToOne 注解定义一个与其它实体类的单值（single-valued）关联，这个关联具有多重

性。一般不需要明确指定目标实体，因为可以从被引用的对象的类型中推测出来。

Cascade元素指定传播到被关联实体上的可级联操作集，可级联的操作由CascadeType枚举定义：

Punlic enum CascadeType{All,Persist,Merge,Remove,Refresh}

值cascade = all 等同于cascade = {p,m.re,ref}

#### 源码

@Target({METHOD, FIELD}) @Retention(RUNTIME)

public @interface ManyToOne {

Class targetEntity() default void.class;

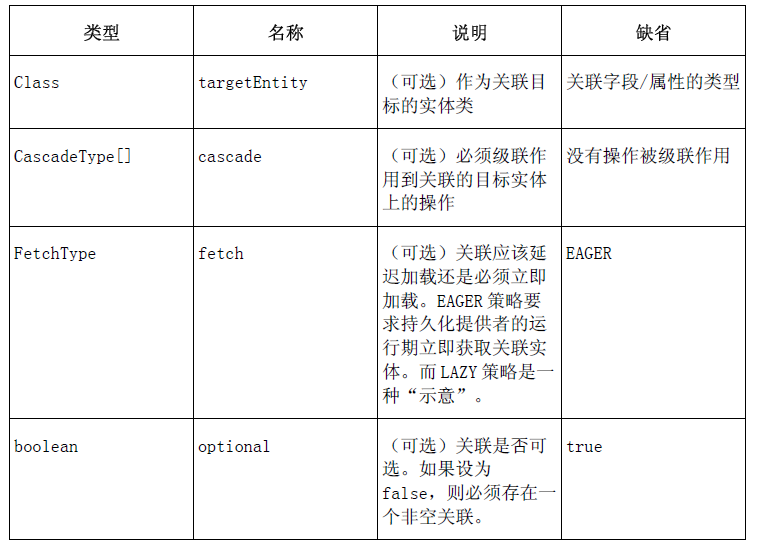
CascadeType[] cascade() default {};

FetchType fetch() default EAGER;

boolean optional() default true;

}

Eager策略要求持久化提供者的运行期必须立即获取关联的实体，lazy策略示意持久化提供者的运行期关联实体应该在首次被访问时获取（延时获取），制订了lazy策略，但是对关联的获取是即时的，这对本规范的一个实现来说是允许的。



### @OneToMany注解

该注解定义了一个一对多值关联，其多重性为一对多

如果定义集合时使用泛型指明了元素类型，则无需指定关联的目标实体的类型；否则必须指

定目标实体类型。

#### 源码

@Target({METHOD, FIELD}) @Retention(RUNTIME)

public @interface OneToMany {

Class targetEntity() default void.class;

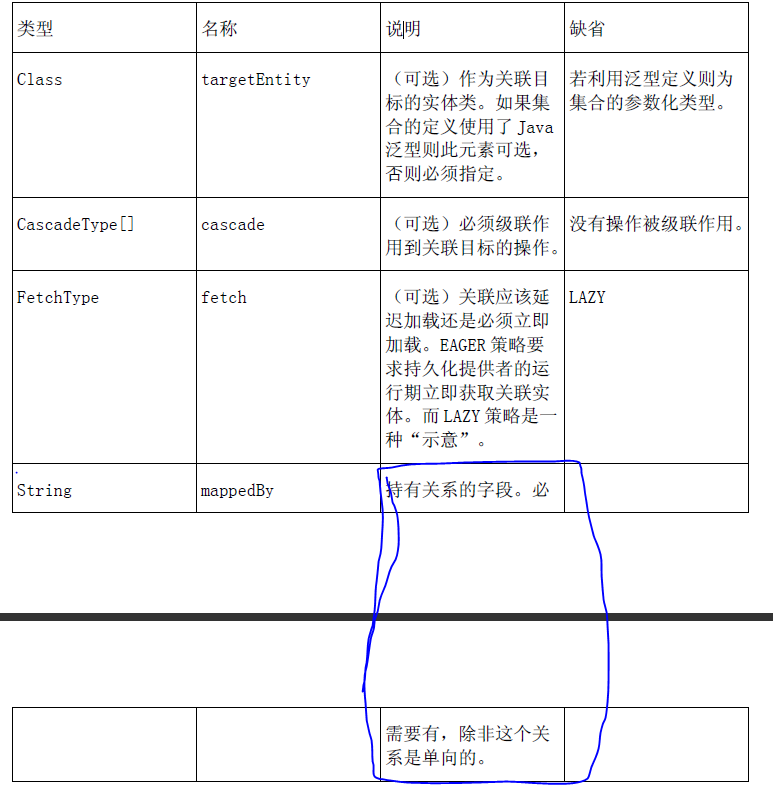
CascadeType[] cascade() default {};

FetchType fetch() default LAZY;

String mappedBy() default "";

}

#### 属性表



#### 实例

##### 使用泛型的一对多关联

Customer 与 Order

在Customer类这边:

@OneToMany(cascade=ALL, mappedBy=”customer”)

public Set<Order> getOrders() { return orders; }

在Order 类这边:

@ManyToOne

@JoinColumn(name="CUST\_ID", nullable=false)

public Customer getCustomer() { return customer; }

##### 未使用泛型的一对多关联

在 Customer 类这边:

@OneToMany(targetEntity=com.acme.Order.class, cascade=ALL,

mappedBy=”customer”)

public Set getOrders() { return orders; }

在 Order 类这边:

@ManyToOne

@JoinColumn(name="CUST\_ID", nullable=false)

public Customer getCustomer() { return customer; }