# 目录

[目录 1](#_Toc365976175)

[版本修订 4](#_Toc365976176)

[1 目的 5](#_Toc365976177)

[2 内容 5](#_Toc365976178)

[3 初识JPA 5](#_Toc365976179)

[3.1 持久化的概念 5](#_Toc365976180)

[3.2 ORM 6](#_Toc365976181)

[3.3 POJO类 6](#_Toc365976182)

[3.4 什么是JPA 7](#_Toc365976183)

[3.4.1 提问 7](#_Toc365976184)

[3.5 JPA的优势 7](#_Toc365976185)

[3.6 JPA在哪里？ 8](#_Toc365976186)

[3.7 JPA的主流实现 8](#_Toc365976187)

[3.8 JPA在JAVA应用的分层结构中的位置 9](#_Toc365976188)

[3.9 JPA主要组件 9](#_Toc365976189)

[3.10 常用接口及概念 10](#_Toc365976190)

[3.10.1 EntityManagerFactory 10](#_Toc365976191)

[3.10.2 EntityManager 11](#_Toc365976192)

[3.10.3 Query 13](#_Toc365976193)

[3.10.4 Persistence 14](#_Toc365976194)

[3.10.5 持久化上下文(Persistence Context) 14](#_Toc365976195)

[3.10.6 持久化单元(Persistence Unit) 15](#_Toc365976196)

[3.11 实体的状态 15](#_Toc365976197)

[3.12 实体类及实体类的要求 16](#_Toc365976198)

[3.13 基本配置文件 17](#_Toc365976199)

[3.14 使用JPA的步骤 17](#_Toc365976200)

[3.15 persistence.xml文件配置 18](#_Toc365976201)

[3.15.1 文档声明 18](#_Toc365976202)

[3.15.2 配置持久化单元(Persistence Unit) 18](#_Toc365976203)

[3.15.3 transaction-type 18](#_Toc365976204)

[3.15.4 声明持久化提供者 18](#_Toc365976205)

[3.15.5 实体类的声明 19](#_Toc365976206)

[3.15.6 数据库信息配置 19](#_Toc365976207)

[3.15.7 数据源配置 19](#_Toc365976208)

[3.15.8 完整配置示例 19](#_Toc365976209)

[3.16 JPADemo 20](#_Toc365976210)

[4 JPA注解 21](#_Toc365976211)

[4.1 两种注解方式 21](#_Toc365976212)

[4.2 常用注解 21](#_Toc365976213)

[4.2.1 @Entity 21](#_Toc365976214)

[4.2.2 @Table 21](#_Toc365976215)

[4.2.3 @id 22](#_Toc365976216)

[4.2.4 @GeneratedValue 22](#_Toc365976217)

[4.2.5 @Transient 29](#_Toc365976218)

[4.2.6 @Lob、@Basic 29](#_Toc365976219)

[4.2.7 @Column 31](#_Toc365976220)

[4.2.8 @Enumerated 31](#_Toc365976221)

[4.2.9 @OneToMany 32](#_Toc365976222)

[4.2.10 @ManyToOne 33](#_Toc365976223)

[4.2.11 @OneToOne 33](#_Toc365976224)

[4.2.12 @ManyToMany 34](#_Toc365976225)

[4.2.13 @JoinColumn 35](#_Toc365976226)

[4.2.14 @Temporal 35](#_Toc365976227)

[5 实体关联关系 35](#_Toc365976228)

[5.1 基本概念 36](#_Toc365976229)

[5.1.1 单双向 36](#_Toc365976230)

[5.2 一对一(OneToOne) 36](#_Toc365976231)

[5.2.1 使用JPA来描述一对一关系 37](#_Toc365976232)

[5.2.2 一对一映射关系的分类 37](#_Toc365976233)

[5.2.3 单向一对一 37](#_Toc365976234)

[5.2.4 双向一对一 39](#_Toc365976235)

[5.3 一对多(OneToMany) 40](#_Toc365976236)

[5.3.1 一对多映射关系的分类 41](#_Toc365976237)

[5.3.2 单向一对多 41](#_Toc365976238)

[5.3.3 双向一对多 42](#_Toc365976239)

[5.4 多对一(ManyToOne) 44](#_Toc365976240)

[5.5 多对多(ManyToMany) 44](#_Toc365976241)

[5.5.1 应用场景 44](#_Toc365976242)

[5.5.2 基本概念 44](#_Toc365976243)

[5.5.3 单向多对多 45](#_Toc365976244)

[5.5.4 双向多对多 50](#_Toc365976245)

[5.5.5 注意事项 54](#_Toc365976246)

[5.6 JPA实体继承 55](#_Toc365976247)

[5.6.1 JPA对实体继承的支持 56](#_Toc365976248)

[5.6.2 SINGLE\_TABLE策略 57](#_Toc365976249)

[5.6.3 TABLE\_PER\_CLASS策略 58](#_Toc365976250)

[5.6.4 JOINED策略 59](#_Toc365976251)

[5.6.5 多态查询 60](#_Toc365976252)

[6 实体的延迟加载 60](#_Toc365976253)

[7 级联（cascade）操作 60](#_Toc365976254)

[8 常用实体管理操作 61](#_Toc365976255)

[8.1 增加 61](#_Toc365976256)

[8.2 删除 61](#_Toc365976257)

[8.3 更新 61](#_Toc365976258)

[8.4 JPA查询 62](#_Toc365976259)

[8.4.1 基本查询 62](#_Toc365976260)

[8.4.2 JPQL 62](#_Toc365976261)

[9 缓存 69](#_Toc365976262)

[9.1 一级缓存 69](#_Toc365976263)

[9.2 二级缓存 69](#_Toc365976264)

[10 OpenJPA常用配置属性 69](#_Toc365976265)

[10.1 根据实体生成数据库表 69](#_Toc365976266)

[10.2 配置数据库相关信息 69](#_Toc365976267)

[11 常见问题 69](#_Toc365976268)

[11.1 persistence.xml文件版本问题 70](#_Toc365976269)

[11.2 使用blueprint发布服务时，使用jndi有时候拿不到 70](#_Toc365976270)

# 版本修订

|  |  |  |  |
| --- | --- | --- | --- |
| 版本 | 日期 | 描述 | 修订人 |
| V1.0 | 2013-8-25 | 创建、撰写 | 陈绪绍 |
| V1.1 | 2013-8-26 | 添加多对多关联关系 | 张欢 |
| V1.2 | 2013-9-3 | 添加@Lob、@Basic、  @Enumrated注解描述 | 陈绪绍 |
| V1.3 | 2013-9-6 | 修正一处描述错误：JPQL位置参数查询序号应从1开始 | 陈绪绍 |
| V1.4 | 2013-9-25 | 增加一对一主键关联内容 | 陈绪绍 |
|  |  |  |  |
|  |  |  |  |

# 目的

物流平台持久化技术采用JPA，团队成员掌握JPA是目前最迫切的。

# 内容

持久化、ORM、JPA、OpenJPA…

文档内容技术更新。

# 初识JPA

推荐一个JPA教程：

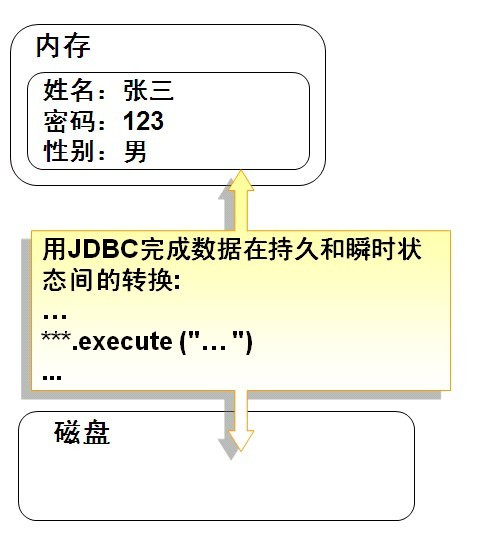
http://docs.oracle.com/cd/E11035\_01/kodo41/full/html/index.html

## 持久化

### 持久化的概念

在系统运行的时候，数据在系统里有二种状态，一是瞬时状态，另一个是持久状态。

将程序数据在瞬时状态和持久状态之间转换的机制叫做持久化。



### 发展中的持久化技术

持久化是企业应用开发的核心需求之一，最近几年以来，它也成为 Java 社区中最热门的话题之一，在 Java 社区努力解决持久化数据管理的过程中，曾经涌现出了非常多的技术方案试图解决这个问题，从最早的序列化，到 JDBC、JDO、ORM、对象数据库、EJB 2.X、EJB3.0，然而这些技术都存在着各种各样的局限，影响他们成为最好的选择。

#### 序列化

序列化是最早出现的、管理持久化数据的实现方案，也是 Java 语言中内置的数据持久化解决方案。它的工作原理是将对象转化为字节流，生成的字节流能够通过网络传输或者保存在文件中。序列化非常易于使用，但是局限性也非常大，由于序列化必须一次将所有对象全部取出，这限制了它在处理大量数据情形下的应用，同时它也无法在更新失败的情况下撤销对对象的修改，这使它无法用于对数据一致性要求严格的应用中。多线程或者多个应用不能同时并发地、互不冲突地读写同一个序列化数据，也不能提供查询功能。

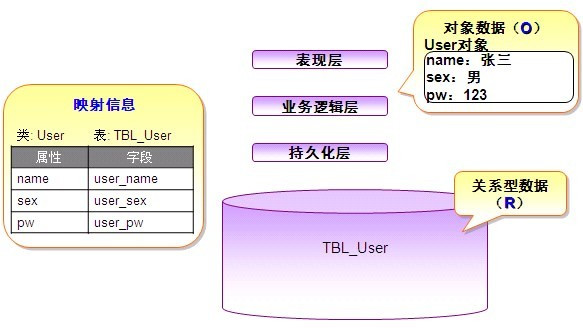
#### JDBC

很多企业应用的开发者选择使用 JDBC 管理关系型数据库中的数据。相对序列化而言，JDBC 克服了很多缺点：它支持处理大量的数据，能够保证数据的一致性，支持信息的并发访问，提供 SQL 查询语言查找数据。不幸的是，JDBC 没有提供序列化所具有的易用性。JDBC 所使用的关系模型不是为保存对象而设计的，因此迫使开发者选择在处理持久数据时放弃面向对象编程，或者自己去开发将面向对象特性（比如：类之间的继承）和关系型数据库进行映射的专有解决方案。

#### ORM

所谓的O，即Object，就是指对象数据，比如图中的User对象，它可以有一些属性。所谓的R，即Relational，就是指关系型数据库，如我们常用的Oracle或Mysql等，而在对象数据和关系型数据库的表之间，我们需要告诉程序是怎么对应的，这就是所谓的映射，即M：Mapping。

简单的说，ORM是通过使用描述对象和数据库之间映射的元数据，将java程序中的对象自动持久化到关系数据库中。本质上就是将数据从一种形式转换到另外一种形式。如下图：



#### 对象数据库（Object DataBase）

已经有一些软件公司选择了开发为保存对象而特别设计的对象数据库，而不是选择将对象映射到关系型数据库上。这种解决方案通常比使用对象/关系映射更加易于使用。和 ORM 相同的问题是，对象数据库的访问接口并没有标准化，因此非常容易形成厂商锁定的局面。与此同时，放弃已经成熟的关系数据库而转向未知的对象数据库让非常多的企业决策者犹豫不决。而且目前为对象数据库而设计的分析工具太少，无法满足企业的需求。而且现实情况下，每一个企业基本上都有大量的已有数据保存在关系数据库中，要从关系数据库转向对象数据库对企业而言也需要大量工作。

#### EJB 2.X

EJB 2.X 实体 Bean 是管理持久化数据的组件框架，和 ORM 解决方案一样，EJB 2.X 实体 Bean 提供持久化数据的面向对象视图。和 ORM 解决方案不一样的是，EJB 2.X 实体 Bean 不仅仅局限于数据库，它展示的信息可能来自于 EIS（Enterprise Information System）或者其他持久化设备。EJB 2.X 实体 Bean 最大的局限是规定了太过于严格的标准，这些标准保证了企业应用能够在不同的 EJB 容器之间可以移植，但是也让 EJB2.X 实体 Bean 规范变得非常复杂并难于使用。而且 EJB 2.X 标准在面向对象特性处理方面的支持非常有限，无法支持继承、多态和复杂关系等面向对象的高级特性。EJB 2.X 实体 Bean 只能在重量级的、价格昂贵的 EJB 容器中运行，这对应用 EJB 2.X 实体 Bean 开发企业应用提出了更高的要求，加重了企业的经济压力。

#### Java 数据对象（Java Data Object，JDO）

JDO 是 Java EE 标准中另外一个支持管理持久化数据的规范，JDO 规范使用和 JPA 非常类似的 API，只是通常是通过 JCA 技术集成到应用服务器上。但是 JDO 是针对轻量级容器而设计的，不能够支持容器级别的声明式安全、事务特性，也无法对远程方法调用提供支持。

#### EJB 3.0 规范

2006 年 5 月 2 日，EJB 3.0 规范最终版由 JCP（Java Community Process） 正式公布，标准号为 JSR（Java Specification Request）220。EJB 3.0 规范的发布为企业应用开发者提供了一种全新的、简化的 API。制定这组 API 的目标是让开发变得更加容易，相对于以前版本的 EJB 规范，这组 API 也更加简单。Java Persistence API 是 EJB 3.0 中负责处理持久化数据管理的部分，目标是为开发者处理持久化数据库管理提供标准支持，也成为 Java EE 容器提供商必须遵守的标准。

EJB 3.0 规范由三部分组成：EJB3.0 Simplified API、EJB 核心规范（EJB Core Contracts and Requirements）和 JPA（Java Persistence API）。

1. **Simplified API**

Simplified API 部分主要规定了基于 EJB 3.0 标准开发企业应用时所需要遵守的 Bean 类和接口要求、这些 API 的使用方式以及容器支持等多方面的内容。还详细的规定了 EJB3.0 中除 Java Persistence API 部分之外的 EJB 实现所支持的注释（Annotation）。规范中还有专门章节讲解 EJB 3.0 和此前的 EJB 规范如何同时工作，以及如何将此前已经开发好的企业应用移植到 EJB 3.0 容器中。其中的 Persistence 的内容放在了 JPA 规范中。

1. **EJB 核心规范**

EJB 核心规范中首先描述了 EJB 在企业应用中的角色、EJB 规范的体系结构，确定了支持 EJB 标准的容器应该遵守的准则和要求。随后从多个角度详细的介绍了 EJB 体系中各部分的功能需求和实现要求，包括 Session Bean、消息驱动 Bean（Message-Driven Bean）、事务、安全管理、部署描述符等。其中的 Persistence 的内容放在了 JPA 规范中。由于 EJB 3.0 规范并不排斥之前的 EJB 规范，因此 EJB 2.X 和 EJB 1.X 中的内容也保留在了 EJB 核心规范中。

1. **Java Persistence API（JPA）**

EJB 2.X 和 EJB 1.X 规范中的实体 Bean（EntityBean）部分都难以使用，使持久化成为 EJB 规范的一个软肋，影响了 EJB 标准发挥更大的作用，自然而然的，JPA 成为了 EJB3.0 规范中被关注最多的部分。JPA 规范部分详细的介绍了 JPA 中实体 Bean 新的定义，并介绍了实体 Bean 支持的注释、全新的查询语言、实体管理接口、容器实现规范等内容。

JPA 标准中引入了新的实体概念，每一个实体都是一个普通的 Java 类，不需要继承任何其他的接口或者扩展某个指定类，这个 Java 类必须使用 javax.persistence.Entity 进行注释。JPA 标准中还提供了包括 javax.persistence.Table、javax.persistence.Id 等在内的多个注释，用于完成实体和数据库之前的映射。JPA 中引入了新的查询语言 JPQL（Java Persistence Query Language），JPQL 允许开发者采用面向对象的查询语言来查找实体，这些实体持久化在关系型的数据库中，”select a from Animal a where a.name=’a’”是一个 JPQL 的例子。其中的 Animal 是一个 Java 类，而不是关系型数据库中的一个表或者视图。除了简单的查询功能之外，JPQL 中还能够支持 Group、Order 等通常只有 SQL 才能提供的高级功能。JPA 标准中还规定了在 Java EE 环境中和非 Java EE 环境中使用 JPA 时的差异，以及 Java EE 环境中容器的职责等。

## 持久化技术的优缺点

下面的这个表格中列出了当前常用持久化技术的优缺点。



## POJO类

简单的Java对象（Plain Ordinary Java Objects）实际就是普通JavaBeans,使用POJO名称是为了避免和EJB混淆起来。一个POJO类应当具有:

* 有一个主键属性,用于唯一标识该对象。
* 有其它的属性
* 有对各个属性操作的get/set方法
* 属性一般是private修饰.
* 一定有一个无参的构造函数(供orm框架反射用)

## 什么是JPA

JPA(JAVA Persistence API)是sun官方提出的Java持久化规范。JPA为Java开发人员提供了一种对象/关系映射工具来管理Java应用中的关系数据库。它的出现主要是为了简化现有的持久化开发工作和整合ORM技术，结束现在Hibernate、TopLink、OpenJPA、JDO等ORM框架各自为营的局面。

JPA 是通过 JDK5.0 注解或 XML 描述对象 - 关系表的映射关系，并将运行期实体对象持久化到数据库中去。JPA 规范小组的领导人就是 Hibernate 的发明者 Gavin King，JPA 规范的制定过程中大量参考了 Hibernate 的内容，所以如果一个对 Hibernate 很熟悉的人，使用起来 JPA 会是轻车熟路，驾轻就熟的，并且会感觉到更简单一些，这主要得益于 JDK5 中引入的注解（annotation）。

### 提问

JPA和Hibernate谁出现得更早？

JPA是为了取代Hibernate、TopLink、OpenJPA吗？会导致Hibernate之类的消亡吗？

## JPA的优势

* **标准化**

提供相同的访问 API，这保证了基于JPA开发的企业应用能够经过少量的修改就能够在不同的JPA框架下运行。越来越多的提供商期待在不久的将来提供 JPA 实施。

* **对容器级特性的支持**  
   JPA 框架中支持大数据集、事务、并发等容器级事务
* **简单易用，集成方便**

　 JPA的主要目标之一就是提供更加简单的编程模型，在JPA框架下创建实体和创建Java 类一样简单，只需要使用 javax.persistence.Entity进行注释；JPA的框架和接口也都非常简单。

* **可媲美JDBC的查询能力**

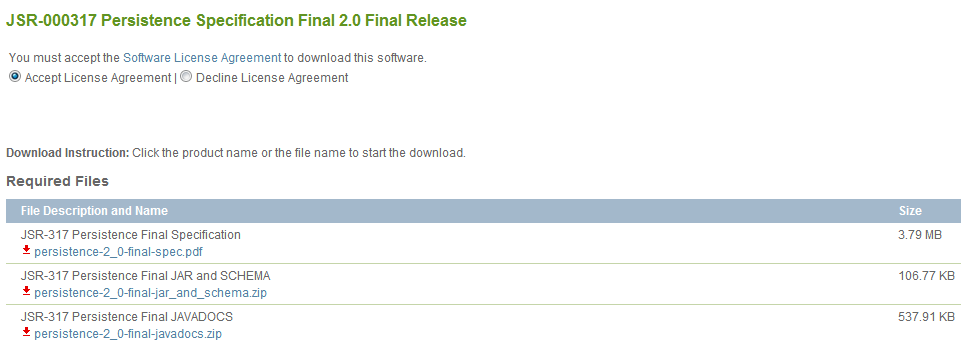
JPA的查询语言是面向对象的，JPA定义了独特的JPQL，而且能够支持批量更新和修改、JOIN、GROUP BY、HAVING 等通常只有 SQL 才能够提供的高级查询特性，甚至还能够支持子查询。

* **支持面向对象的高级特性**

JPA 中能够支持面向对象的高级特性，如类之间的继承、多态和类之间的复杂关系，最大限度的使用面向对象的模型

## JPA在哪里？

<http://download.oracle.com/otndocs/jcp/persistence-2.0-fr-oth-JSpec/>



## JPA的主流实现

JPA 的目标之一是制定一个可以由很多供应商实现的API，目前OpenJPA、EclipseLink、Hibernate、TopLink等都提供了JPA的实现。

#### Apache OpenJPA

OpenJPA是Apache提供的开源Java持久化框架。物流平台使用OpenJPA。

OpenJPA帮助文档：

http://ci.apache.org/projects/openjpa/trunk/docbook/manual.html

#### EclipseLink

EclipseLink是由Oracle TopLink捐献给Eclipse社区的开源JPA框架。

#### Hibernate

Hibernate 从3.2开始兼容JPA。

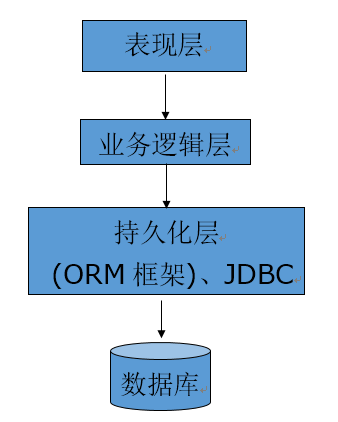
#### Oracle TopLink

由Oracle提供的JPA实现框架。TopLink与EclipseLink不同在于，TopLink在EclipseLink基础上增加与Oracle其他产品整合的代码。

## JPA在JAVA应用的分层结构中的位置

JPA位于JAVA应用的分层中的持久化层。持久化层封装了数据访问的细节，为业务逻辑层提供了面向对象的API。完善的持久化层应该达到的目标：

* 代码重用性高，可完成所有的数据访问操作。
* 如果需要的话，能够支持多种数据库平台。
* 具有相对独立性，当持久化层变化时，不会影响上层实现。



## JPA主要组件

JPA 中定义一套类和接口用于实现持久化管理和对象/关系的映射，下面这张图中显示了 JPA 的主要组件以及它们之间的相互关系。

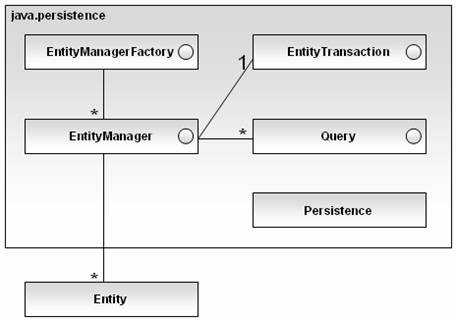


图1 JPA 主要组件和相互关系

## 常用接口及概念

### EntityManagerFactory

 EntityManager 的工厂类，负责创建 EntityManager 对象。类似于Hibernate 中的SessionFactory。线程安全。

Javax.persistence.EntityManagerFactory接口主要用来创建实体管理器。该接口约定了如下4个方法：

createEntityManager()：用于创建实体管理器对象实例。

createEntityManager(Map map)：用于创建实体管理器对象实例的重载方法，Map参数用于提供JDBC驱动属性。

isOpen()：检查实体管理器工厂是否处于打开状态。实体管理器工厂创建后一直处于打开状态，除非调用close()方法将其关闭。

close()：关闭实体管理器工厂。实体管理器工厂关闭后将释放所有资源，isOpen()方法测试将返回false，其它方法将不能调用，否则将导致IllegalStateException异常。

### EntityManager

EntityManager 是 JPA 应用中使用的基本对象，通过它提供的相应方法可以管理持久化对象，也可以新建或者删除持久化对象。EntityManager 还负责创建 Query 实例。类似于Hibernate 中的Session。不是线程安全的。

Javax.persistence.EntityManager接口用于建立持久上下文环境和管理所有实体及其生命周期。

一个EntityManager实例与一个持久上下文环境相关联，代表一组O/R映射的实体集合。

EntityManager接口定义了一系列用于对象持久化的方法，可以创建、删除持久化实体，根据主键查询实体，以及查询关于实体的信息等。

**EntityManager接口的常用方法：**

* persist (Object entity)

持久化一个新的对象实例，并使其处于被管理状态，即将对象实例保存到与其映射的数据库表中，并建立与此实体记录的关联。

* merge (T entity)

更新持久化对象实例，即以新的属性值更新数据库表的原记录。 返回被更新的实体。

* remove (Object entity)

删除实例。如果实例是被管理的，即与数据库实体记录关联，则同时会删除关联的数据库记录。

* find (Class<T> entityClass,Object primaryKey)

根据主键查询实体。找到了则返回找到的实体实例，未找到则返回null。第一个参数为被查询的实体类对应的Class，第二个参数为待查找实体的主键值。

* getReference (Class<T> entityClass,Object primaryKey)

根据主键获取实例，常用于延迟加载的情况。

* flush ()

同步持久上下文环境，即将持久上下文环境的所有未保存实体的状态信息保存到数据库中。

* setFlushMode (FlushModeType flushMode)

设置持久上下文环境的Flush模式。参数可以取2个枚举值：FlushModeType.AUTO为自动更新数据库实体，FlushModeType.COMMIT为直到提交事务时才更新数据库记录。

* getFlushMode ()

获取持久上下文环境的Flush模式。返回FlushModeType类的枚举值。

* lock (Object entity, LockModeType lockMode)

给实体加读/写锁。第一个参数为需要加锁的实体，第二个参数为锁的类型。读锁为枚举值LockModeType.READ，写锁为LockModeType.WRITE。读锁只会阻止其它用户更新或删除该实体记录；写锁则完全阻止其它用户访问该实体记录。

* refresh (Object entity)

与merge相反，是用数据库实体记录的值更新实体对象的状态，即更新实例的属性值。

* clear ()

清除持久上下文环境，断开所有关联的实体。如果这时还有未提交的更新则会被撤消。

* contains (Object entity)

判断一个实例是否属于当前持久上下文环境管理的实体。

* createQuery (String qlString)

创建一个查询对象。

* close ()

关闭实体管理器。之后若调用实体管理器实例的方法或其派生的查询对象的方法都将抛出IllegalstateException异常，除了getTransaction和isOpen方法(返回false)。

不过，当与实体管理器关联的事务处于活动状态时，调用close方法后持久上下文将仍处于被管理状态，直到事务完成。

* isOpen ()

判断当前的实体管理器是否是已被关闭。

* getTransaction ()

返回资源层的事务对象。EntityTransaction实例可以用于顺序的开始和提交多个事务。

### Query

Query 是查询实体的接口，Query 对象可以从 EntityManager 中获得。根据 EJB 3.0 规范中的描述，Query 接口需要同时支持 JPQL 和原生态 SQL 两种语法。

Query接口封装了执行数据库查询的相关方法。调用实体管理器的createQuery、create NamedQuery及createNative Query方法可以获得查询对象，进而可调用Query接口的相关方法来执行查询操作。

**Query接口的主要方法：**

* int executeUpdate()

用于执行update或delete语句。如果Query对象的查询语句为select将抛出IllegalStateException异常，如果事务不存在将抛出TransactionRequiredException异常。

* List getResultList()

用于执行select语句并返回结果集实体列表。如果Query对象的查询语句为update或delete将抛出IllegalStateException异常。

* Object getSingleResult()

用于执行只返回单个结果实体的select语句。如果查询语句为update或delete则抛出IllsegalStateException异常；如果没有得到返回结果将抛出EntityNotFoundException异常；如果返回多于一个结果将抛出NonUniqueResultException异常；

* Query setFirstResult(int startPosition)

用于设置从哪个实体记录开始返回查询结果。如果参数为负数将抛出IllegalArgumentException异常。

* Query setMaxResults(int maxResult)

用于设置返回结果实体的最大数。如果参数为负数将抛出IllegalArgumentException异常。与setFirstResult结合使用可实现分页查询。

* Query setFlushMode(FlushModeType flushMode)

设置查询对象的Flush模式。参数可以取2个枚举值：FlushModeType.AUTO为自动更新数据库记录，FlushMode Type.COMMIT为直到提交事务时才更新数据库记录。

* setHint(String hintName, Object value)

设置与查询对象相关的特定供应商参数或提示信息。参数名及其取值需要参考特定JPA实现库提供商的文档。如果第二个参数无效将抛出IllegalArgumentException异常。

* setParameter(int position, Object value)

为查询语句的指定位置参数赋值。position指定参数序号，value为赋给参数的值。

* setParameter(int position, Date d, TemporalType type)

为查询语句的指定位置参数赋Date值。position指定参数序号，value为赋给参数的值，temporalType取TemporalType的枚举常量，包括DATE、TIME及TIMESTAMP三个，，用于将Java的Date型值临时转换为数据库支持的日期时间类型（java.sql.Date、java.sql.Time及java.sql.Timestamp）。

### Persistence

Persistence 是一个工具类，负责根据配置文件提供的参数创建 EntityManagerFactory 对象。

### 持久化上下文(Persistence Context)

持久化上下文是一级处于托管状态下实体所组成的集合，它处于EntityManager的管理之下，因此应用程序可通过EntityManager与持久化上下文进行交互。如果持久化上下文被关闭，该上下文中的所有实体都将脱离EntityManager的管理，进入脱管状态，此时对实体所做的修改将不会被自动同步至底层数据库。

EntityManager负责跟踪持久化上下文中所有实体的状态，当应用程序改变了持久化上下文中的实体状态后，EntityManager将会根据指定的flush模式将实体的状态写入底层数据库。

### 持久化单元(Persistence Unit)

持久化单元由persistence.xml文件定义，该文件中包含一个<persistence-unit>元素，该元素用于定义持久化单元，该元素的name属性的值就是该持久化单元的名称，应用程序可以通过该名称来访问该持久化单元。

## 实体的状态

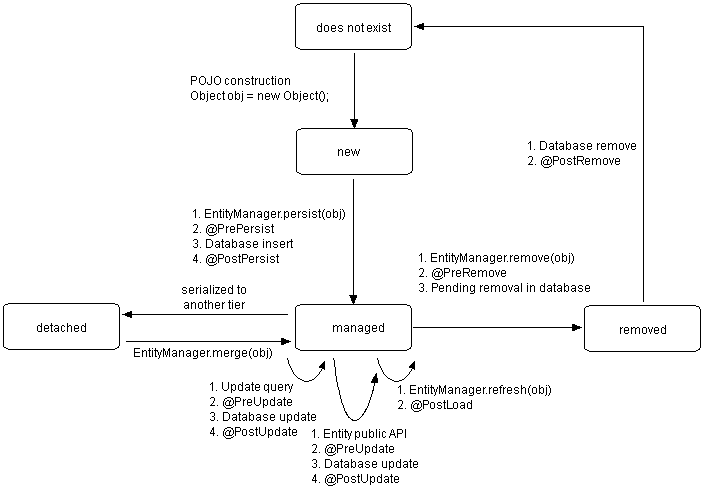
JPA中，对象的状态有多个状态：

* 新建（New）状态：当对象用new操作符创建时，对象处于该状态，此时，对象不受任何管理。
* 被管理（Managed）状态：当新建（New）或已被删除（Removed）的对象被持久化(persist)后，即处于被管理的状态，此外，经由EntityManager从数据库中获取（retrieve,find ,query……）的对象也处于被管理状态。调用EntityManager对象的contains(entity)方法来判别对象处于被管理（Managed）状态，非Managed状态的对象均返回false,代码如下：

Boolean isManaged = em.[contains](http://www.objectdb.com/api/java/jpa/EntityManager/contains_Object)(employee);

* 已删除(Removed)状态：当被管理的（Managed）的对象被删除(remove)后，即处于此状态。
* 脱管（Detached）状态：当EntityManager关闭（close）或者清空（clear）时，已删除(Removed)和被管理（Managed）对象的状态变为脱管状态（Detached）。

对所有管理（Managed）状态或者已删除(Removed)状态执行flush或commit命令，则变化会反映到数据库中。如下图：



## 实体类及实体类的要求

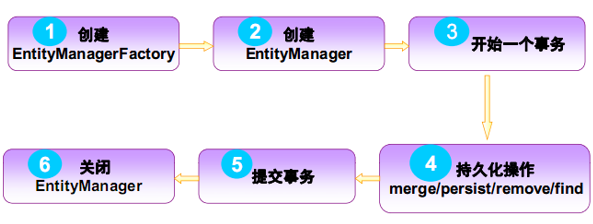
1. 实体类就是普通的JavaBean，即POJO。
2. 实体类必须用 @javax.persistence.Entity 进行注解；
3. 必须使用 @javax.persistence.Id 来注解一个主键；
4. 实体类必须拥有一个 public 或者 protected 的无参构造函数，之外实体类还可以拥有其他的构造函数；
5. 实体类必须是一个顶级类（top-level class）。一个枚举（enum）或者一个接口（interface）不能被注解为一个实体；
6. 实体类不能是 final 类型的，也不能有 final 类型的方法；
7. 如果实体类的一个实例需要用传值的方式调用（例如，远程调用），则这个实体类必须实现（implements）java.io.Serializable 接口。
8. 为了表示实体信息，需要定义与数据库表的列对应的属性，Userinfo表包含userid、username、userpass和usertype 4个属性，并且需要提供对属性操作的set方法和get方法

## 基本配置文件

基本配置文件为persistence.xml，persistence.xml配置必须位于META-INF目录下。

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <persistence xmlns="http://java.sun.com/xml/ns/persistence"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence  http://java.sun.com/xml/ns/persistence/persistence\_2\_0.xsd"  version="2.0">  <persistence-unit name="OpenBooks">  <jta-data-source>java:comp/env/jdbc/OpenBooks</jta-data-source>  <non-jta-data-source>java:comp/env/jdbc/NonTxOpenBooks</non-jta-data-source>  <class>openbook.domain.Book</class>  <class>openbook.domain.Customer</class>  <class>openbook.domain.Author</class>  <properties>  <property name="openjpa.DataCache" value="true"/>  <property name="openjpa.RemoteCommitProvider" value="sjvm"/>  <property name="openjpa.DynamicEnhancementAgent" value="false"/>  <property name="openjpa.RuntimeUnenhancedClasses" value="unsupported"/>  <property name=  "openjpa.jdbc.SynchronizeMappings" value="buildSchema(ForeignKeys=true)"/>  <property name=  "openjpa.jdbc.QuerySQLCache" value="true(EnableStatistics=true)"/>  </properties>  </persistence-unit>  </persistence> |

## 使用JPA的步骤



## persistence.xml文件配置

### 文档声明

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <persistence xmlns=*"http://java.sun.com/xml/ns/persistence"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://java.sun.com/xml/ns/persistence*  *http://java.sun.com/xml/ns/persistence/persistence\_2\_0.xsd"*  version=*"2.0"*>  ……  </persistence> |

### 配置持久化单元(Persistence Unit)

包含的元素为<persistence-unit>，主要属性包括name和transaction-type，name指出持久单元的名字，transaction-type指出事务的类型。

|  |
| --- |
| <persistence-unit name="default“ transaction-type="JTA ">  </persistence-unit> |

name="default"声明持久单元的名字为default，transaction-type="JTA"声明当JTA事务。

### transaction-type

　配置使用的事务类型。有JTA和RESOURCE\_LOCAL两种类型可以选择。在OSGi和JavaEE环境中默认为JTA,在JavaSE环境中默认为RESOURCE\_LOCAL。当在persistent.xml文件使用<jta-data-source>,默认就是JTA事务，使用<non-jta-data-source>，默认就是使用RESOURCE\_LOCAL事务。

更多有关JTA和RESOURCE\_LOCAL的内容描述，参考如下文档：

OSGi技术预研 - Apache Aries应用(JPA,JNDI,Transaction,etc).docx

### 声明持久化提供者

使用<provider>标签

<provider>持久提供者的名字</provider>

例如，使用OpenJPA作为持久提供者，配置方式如下：

|  |
| --- |
| <provider>org.apache.openjpa.persistence.PersistenceProviderImpl</provider> |

### 实体类的声明

使用<class>标签。如：

|  |
| --- |
| <class>net.chenxs.samples.jpa.domain.Classes</class>  <class>net.chenxs.samples.jpa.domain.Student</class> |

### 数据库信息配置

通过<properties>标签配置数据库信息。

|  |
| --- |
| <properties>  <property name=*"openjpa.ConnectionURL"* value=*"jdbc:mysql://localhost:3306/jpa"* />  <property name=*"openjpa.ConnectionDriverName"* value=*"com.mysql.jdbc.Driver"* />  <property name=*"openjpa.ConnectionUserName"* value=*"root"* />  <property name=*"openjpa.ConnectionPassword"* value=*"123456"* />  </properties> |

### 数据源配置

使用<jta-data-source>标签进行配置。支持一个jndi名称。如：

<jta-data-source>osgi:service/jdbc/mysqlds</jta-data-source>

### 完整配置示例

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <persistence xmlns="http://java.sun.com/xml/ns/persistence"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence  http://java.sun.com/xml/ns/persistence/persistence\_2\_0.xsd"  version="2.0">  <persistence-unit name=*"default"* transaction-type=*"RESOURCE\_LOCAL"*>  <provider>org.apache.openjpa.persistence.PersistenceProviderImpl</provider>  <class>net.chenxs.samples.jpa.domain.Classes</class>  <class>net.chenxs.samples.jpa.domain.Student</class>  <exclude-unlisted-classes>true</exclude-unlisted-classes>  <properties>  <property name=*"openjpa.jdbc.SynchronizeMappings"* value=*"buildSchema(ForeignKeys=true)"* />  <property name=*"openjpa.Log"* value=*"DefaultLevel=TRACE, Tool=TRACE, SQL=TRACE"* />  <property name=*"openjpa.ConnectionURL"* value=*"jdbc:mysql://localhost:3306/jpa"* />  <property name=*"openjpa.ConnectionDriverName"* value=*"com.mysql.jdbc.Driver"* />  <property name=*"openjpa.ConnectionUserName"* value=*"root"* />  <property name=*"openjpa.ConnectionPassword"* value=*"123456"* />  </properties>  </persistence-unit>  </persistence> |

## JPADemo

|  |
| --- |
| package net.chenxs.samples.jpa;  import java.util.List;  import javax.persistence.EntityManager;  import javax.persistence.EntityManagerFactory;  import javax.persistence.Persistence;  import javax.persistence.Query;  import net.chenxs.samples.jpa.domain.User;  public class JPADemo {  public static void main(String[] args) {  EntityManagerFactory factory = Persistence  .createEntityManagerFactory("default");  EntityManager em = factory.createEntityManager();  em.getTransaction().begin();//begin  　　User user = new User();  user.setId(1);  user.setName("cxs");  user.setAge(100);  em. persist (user);  em.getTransaction().commit();//commit  em.close();  factory.close();  }  } |

# JPA注解

## 两种注解方式

JPA 中将一个类注解成实体类（entity class）有两种不同的注解方式：基于属性（property-based）和基于字段（field-based）的注解。

1，基于字段的注解，就是直接将注解放置在实体类的字段的前面。前面的 Employee 实体类就是使用的这种注解方式；

2，基于属性的注解，就是直接将注解放置在实体类相应的 getter 方法前面，而不是 setter 方法前面。

## 常用注解

常用注解位于javax.persistence.\*包下。

Oracle官方页面

<http://www.oracle.com/technetwork/cn/testcontent/toplink-jpa-annotations-096251.html>

### @Entity

@Entity注解用于实体类声明语句之前，指出该Java类为实体类，将映射到指定的数据库表。如下面的语句声明一个实体类Customer，它将映射到数据库中的customer表上。

**@Entity(name=”t\_customer”)**

**public class Customer { … }**

name为可选，对应数据库中的一个表。

### @Table

当实体类与其映射的数据库表名不同名时需要使用@Table标注说明，该标注与@Entity标注同时使用。建议时刻使用该属性。

@Table标注的常用选项是name，用于指明数据库的表名。如下面的实体类User的声明中使用了@Table指明其映射的数据库表名为t\_user：

@Table(name = "t\_user")

@Entity

public class User { … }

@Table标注还有一个两个选项catalog和schema用于设置表所属的数据库目录或模式，通常为数据库名。uniqueConstraints选项用于设置约束条件，通常不须在此设置。

### @id

@Id标注用于声明一个实体类的属性映射为数据库的主键列。该属性通常置于属性声明语句之前，@Id标注也可置于属性的getter方法之前。如：

@Id

private String bid;

### @GeneratedValue

@GeneratedValue设置实体标识的生成策略，常与@Id一起使用。

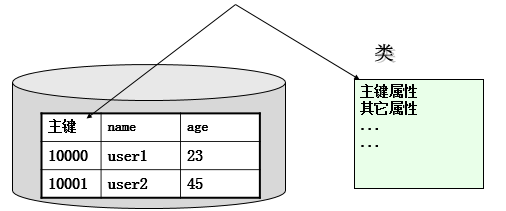
语法：

@Id

@Generatedvalue(strategy=GenerationType.SEQUENCE,generator=”seq”)

#### 什么是对象标识符(OID)

即Object ID，JPA中的持久化对象对应数据库中的一张数据表，因此区分不同的持久化对象，在Hibernate中是通过OID(对象标识符)来完成的，从表的角度看，OID对应表的主键。从类的角度看OID对应类的主键属性。如下图：



#### 主键生成策略

数据的唯一性是所有应用程序非常基本的要求，由开发者或者用户来维护这种唯一性存在着较大的风险，因此，由系统自动产生唯一标识是一种常见的做法。OpenJPA 中支持四种不同的实体标识自动生成策略：

* 容器自动生成的实体标识；
* 使用数据库的自动增长字段生成实体标识；
* 根据数据库序列号（Sequence）技术生成实体标识；
* 使用数据库表的字段生成实体标识；

这四种方式各有优缺点，开发者可以根据实际情况进行选择。

每一个需要自动生成实体标识的实体都需要为它的实体标识字段提供 GeneratedValue注释和相应的参数，OpenJPA 框架会根据注释和参数来处理实体标识的自动生成。

使用 GeneratedValue注释自动生成的实体标识可以是数值类型字段如 byte、short、int、long等，或者它们对应的包装器类型Byte、Short、Integer、Long等，也可以是字符串类型。

GeneratedValue注解可以支持两个属性 strategy和 generator。

* **strategy**

strategy是 GenerationType类型的枚举值，它的内容将指定 OpenJPA 容器自动生成实体标识的方式。strategy属性可以是下列枚举值：

1. GeneratorType.AUTO

表示实体标识由 OpenJPA 容器自动生成，这也是 Strategy 属性的默认值。

OpenJPA 容器默认的实体标识自动生成策略是由容器管理实体标识的自动生成，容器管理的实体标识可以支持数值型和字符型两种。当容器管理的实体标识是数值型时，OpenJPA 容器自动创建一个数据库表 OPENJPA\_SEQUENCE\_TABLE，用其中的SEQUENCE\_VALUE字段来记录实体的实体标识的增长。

当容器管理的实体标识是字符串类型时，OpenJPA 支持使用 uuid-string 和 uuid-hex 两种方式生成相应的实体标识。如果我们选择使用 uuid-string 方式生成实体标识时，OpenJPA 框架会自动为实体生成一个 128 位的 UUID，并且将这个 UUID 转化为使用 16 位字符表示的字符串。如果我们选择使用 uuid-hex 方式生成实体标识时，OpenJPA 框架会自动为实体生成一个 128 位的 UUID，并且将这个 UUID 转化为使用 32 位字符表示的 16 进制的字符串。

**数值标识**

容器管理的实体标识可以是数值型的，OpenJPA 框架管理的实体标识借助于数据库的表来实现，在运行时 OpenJPA 框架会自动在数据库中创建表 OPENJPA\_SEQUENCE\_TABLE。它有两个字段：ID和 SEQUENCE\_VALUE，这两个字段都是数值类型，其中 ID是表的主键字段，它的内容是查询当前实体标识时所使用的关键词，默认值是 0。而 SEQUENCE\_VALUE记录了当前 OpenJPA 框架中当前实体标识的历史数据，内容是已经被获取实体标识的最大数值加 1。

我们要使用注释描述 Animal实体的标识由容器自动生成，只需要为它的标识字段提供 GeneratedValue注释，并且把它的 strategy属性设置为 GenerationType.AUTO，如：

|  |
| --- |
| import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  @Entity  public class Animal {  @Id  **@GeneratedValue(strategy=GenerationType.AUTO)**  private long id;  private String name;  …….  } |

保存 实体的第一个实例时，OpenJPA 框架自动调用 SQL 语句 SELECT SEQUENCE\_VALUE FROM OPENJPA\_SEQUENCE\_TABLE WHERE ID=0，从默认保存实体标识的 OPENJPA\_SEQUENCE\_TABLE表中获取实体的标识，如果不存在 ID为 0 的记录，OpenJPA 框架自动将实体的标识设置为 1。

容器管理实体标识的情况下，为了获得实体标识，应用程序将不得不频繁地和数据库交互，这会影响应用程序的运行效率。OpenJPA 中使用实体标识缓存机制解决这个问题。默认情况下，当应用程序第一次获取实体标识时，OpenJPA 框架从数据库中一次性获取 50 个连续的实体标识缓存起来，当下一次应用程序需要获取实体标识时，OpenJPA 将首先检测缓存中是否存在实体标识，如果存在，OpenJPA 将直接使用缓存中的实体标识，如果不存在，OpenJPA 框架将会从数据库中再次获取 50 个连续的实体标识缓存起来，如此类推。这样的处理方式可以大大减少由于获取实体标识而产生的数据库交互，提升应用程序的运行效率。

当实体标识成功获取之后，OpenJPA 框架会把当前实体标识的最大值 +1 后持久化到数据库中。由于实体标识缓存的原因，当我们第一次获取实体标识后，OpenJPA 会将 OPENJPA\_SEQUENCE\_TABLE表的 SEQUENCE\_VALUE的值设置为 51，当 OpenJPA 多次从数据库中获取实体标识后，SEQUENCE\_VALUE的值会以 50 为单位递增，变为 101、151、201 …。

OpenJPA 缓存的实体标识不是永久存在的，只能在同一个 EntityManagerFactory管理范围内起作用，也就是说，当获取实体标识的EntityManagerFactory对象被关闭后，这些被获取的实体标识中没有用掉的那一部分标识就丢失了，这会造成实体标识的不连续。由同一个 EntityManagerFactory对象创建的 EntityManager上下文之间则能够共享 OpenJPA 框架获取的实体标识，这意味着，我们可以使用同一个 EntityManagerFactory对象创建多个 EntityManager对象，用它来持久化实体，然后关闭它，在持久化过程中所需要的实体表示将会使用同一个实体标识的缓存区，因此不会引起实体标识的丢失。

容器管理的实体标识还有一个非常重要的特性：所有被容器管理的实体标识都是共享的。不管 OpenJPA 容器中存在多少个不同的被容器管理的实体标识，它们都会从同一个实体标识缓存中获取实体标识。我们可以用下面的例子说明这种情况：假设 OpenJPA 容器中存在两个实体类 Dog和 Fish，它们的实体标识字段都是数值型，并且都由 OpenJPA 管理。当我们首先持久化一个 Dog对象时，它的实体标识将会是 1，紧接着我们持久化一个 Fish对象，它的实体标识就是 2，依次类推。

**uuid-string(16位)**

要使用 uuid-string 机制自动生成实体标识，我们需要将实体主键字段的 GeneratedValue注释的 strategy属性设置为GenarationType.AUTO，然后将 GeneratedValue注释的 generator属性设置为 uuid-string。如：

|  |
| --- |
| import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  @Entity  public class Animal {  @Id  @GeneratedValue(strategy=GenerationType.AUTO, generator = "uuid-string")  private String id;  private String name;  …….  } |

**uuid-hex(32位)**

要使用 uuid-hex 机制自动生成实体标识，我们必须将实体主键字段的 GeneratedValue注释的 strategy属性设置为GenarationType.AUTO，然后将 GeneratedValue注释的 generator属性设置为 uuid-hex。如：

|  |
| --- |
| import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  @Entity  public class Animal {  @Id  @GeneratedValue(strategy=GenerationType.AUTO, generator = "uuid-hex")  private String id;  private String name;  …….  } |

1. GenerationType.IDENTITY

OpenJPA容器将使用数据库的自增长字段为新增加的实体对象赋唯一值，作为实体的标识。这种情况下需要数据库提供对自增长字段的支持，常用的数据库中，HSQL、SQL Server、MySQL、DB2、Derby 等数据库都能够提供这种支持。Oracle不支持。

要将实体标识的自动生成委托给数据库的自增长字段特性，需要数据库和实体定义的双方配合才能够达到：首先，必须将实体标识字段对应的数据库列修改为自动增长列，另外还需要将实体类中实体标识字段的 GeneratedValue注释的 stragety属性的值设置为GenerationType.IDENTITY。如：

|  |
| --- |
| import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  @Entity  public class Animal {  @Id  **@GeneratedValue(strategy=GenerationType.IDENTITY)**  private long id;  private String name;  …….  } |

1. GenerationType.SEQUENCE

表示使用数据库的序列号为新增加的实体对象赋唯一值，作为实体的标识。这种情况下需要数据库提供对序列号的支持，常用的数据库中，Oracle、PostgreSQL 等数据库都能够提供这种支持，MySQL不支持。

**序列号（Sequence）**

序列号是 Oracle、PostgreSQL 等数据库提供的一种特性，用于为数据库的记录提供自动增长的编号，使用 Oracle、PostgreSQL 等数据库应用程序的设计者通常期望将实体标识的自动生成委托给数据库的这种特性，OpenJPA 框架中的实体标识能够满足应用程序设计者的要求，使用数据库的序列号为实体自动生成标识。

要将实体标识的自动生成委托给数据库的序列号特性，需要数据库和实体定义的双方配合才能够达到：首先，必须在数据库中创建合适的序列号，另外还需要为实体标识字段提供 SequenceGenerator注解，设置它的参数，为实体类提供关于序列号的信息，同时将实体类中实体标识字段的 GeneratedValue注释的 stragety属性的值设置为 GenerationType.SEQUENCE，将 generator属性的值设置为SequenceGenerator注释的 name属性的值。

我们以 Animal 实体在 Oracle 数据库中的持久化来说明如何使用自增长字段自动生成实体标识所需要采取的步骤：

首先，在Oracle 数据库中运行下面的 SQL 语句创建名为 HelloWorldSequence的序列号，序列号支持 cache，大小为 50：

|  |
| --- |
| CREATE SEQUENCE HELLOWORLDSEQUENCE  START WITH 0  INCREMENT BY 1  MINVALUE 1  CACHE 50  NOCYCLE  NOORDER |

然后，在 Oracle 数据库中，我们使用下面的 SQL 语句创建 ANIMAL表：

|  |
| --- |
| CREATE TABLE EOS52.ANIMAL  (  ID CHAR(10),  NAME VARCHAR2(100) NOT NULL,  CONSTRAINT PK\_ANIMAL PRIMARY KEY (ID )  ) |

在数据库部分创建合适的序列号和相应的数据库表后，在实体 Animal的定义中，我们需要将 id字段 GeneratedValue注释的stragety属性的值设置为 GenerationType.SEQUENCE，设置它的 generator属性的值为 SeqGenerator。我们还需要为 id字段提供另外一个相关的注释 SequenceGenerator，设置它的 name属性为 SeqGenerator，设置它 sequenceName属性为 HelloWorldSequence。如：

|  |
| --- |
| @Entity  public class Animal {  @Id  **@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "SeqGenerator")**  **@SequenceGenerator(name = "SeqGenerator", sequenceName = " HelloWorldSequence")**  private long id;  private String name;  ……..  } |

1. GenerationType.TABLE

表示使用数据库中指定表的某个字段记录实体对象的标识，通过该字段的增长为新增加的实体对象赋唯一值，作为实体的标识。此文档不详述。

### @Transient

标识该字段不被持久化。

如：

|  |
| --- |
| @Entity  @Table(name="t\_person\_onetoone1")  **public** **class** Person {  @Id  @GeneratedValue(strategy = GenerationType.*IDENTITY*)  **private** Integer id;  **private** String name;  @Transient  private String pswd;  **private** **int** sex;  **private** **int** age;  } |

### @Lob、@Basic

通常，可以在数据库中保存诸如图片、长文本类型的数据。这些类型的数据一般是保存为Blog和Clob类型，在JDBC中会采用java.sql.Blob、java.sql.Clob来表示这些大数据类型的值。

JPA也为这种大数据类型的值提供了支持，JPA使用@Lob来修饰这种大数据类型，当实体的Field为byte[]、Byte[]或java.io.Serializable类型时，@Lob修饰的Field将映射为底层的Blog列；当实体的Field为Char[]、Character[]或java.lang.String类型时，@Lob修饰的Field将映射为底层的Clob列。如：

|  |
| --- |
| @Entity  @Table(name="t\_user")  public class Person extends AbstractBaseModel{  @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  private Integer id;  private String name;  //登记照  @Lob  private byte[] pic;  …..  } |

上面Person实体的pic是一个byte[]类型的字段，我们采用了@Lob来修饰该字段，那么pic会被映射为Blob类型。

对于使用@Lob修饰的在数据类型，底层数据库往往会采用Blog或Clob类型的列来保存这种大数据类型的值。数据库在加载这种大数据类型的值也是需要较大开销的。如上的Person实体，在默认情况下，JPA会在加载该Person实体时也加载它的pic属性――这是一个大数据类型的值，将需要较大的系统开销。有一种可能的情况是：程序只需要该访问该Person实体的name，根本不关心Person实体的pic属性，那么JPA就白白浪费时间来加载pic属性性了，这显然不是一个好的做法。

为了改变这种情况，我们希望有一种机制可以做到：JPA加载Person实体时并不加载它的pic属性，而是加载一个“虚拟”代理，等到程序真正需要pic属性时才从底层数据库中加载数据。JPA为这种机制提供了支持，并将这种机制称为延迟加载，只需要在开发实体时使用@Basic修饰该Field即可。

使用@Basic可以指定如下属性：

* fetch：指定是否延迟加载该Field。该属性值可接受FetchType.EAGER、FetchType.LAZY两个值之一，其中前者指定不使用延迟加载；后者指定使用延迟加载。
* Optional：指定该Field映射的数据列是否允许使用null值。

如：

|  |
| --- |
| @Entity  @Table(name="t\_user")  **public** **class** Person **extends** AbstractBaseModel{  @Id  @GeneratedValue(strategy = GenerationType.*IDENTITY*)  **private** Integer id;  **private** String name;  //登记照  @Lob  @Basic(fetch=FetchType.*LAZY*)  **private** **byte**[] pic; |

### @Column

@javax.persistence.Column 注解，定义了列的属性，可以用这个注解改变数据库中表的列名（缺省情况下表对应的列名和类的字段名同名）；指定列的长度；或者指定某列是否可以为空，或者是否唯一，或者能否更新或插入。

@Column标注的常用选项是name，用于设置映射数据库表的列名。此外，该标注还包含其它多个属性(详见Java EE API对于javax.persistence.Column的说明)，如：unique用于设置唯一性，为true时表示不能取重复值；nullable用于设置是否允许空值，为false时表示不能为空；length可用于设置可接受的最大长度，等等。例如：

@Column(name = "f\_bname", nullable = false, length = 30)

private String bname;

### @Enumerated

JPA使用通过@Enumerated标注枚举类型。使用@Enumerated时，需要指定EnumType属性。

EnumType定义如下：

|  |
| --- |
| public enum EnumType {  ORDINAL,  STRING  } |

ORDINAL表示持久化的为枚举常量的序数值，STRING表示持久化的为枚举类型的名称。默认为ORDINAL，持久化值。例如以上示例中标注的为STRING，这样持久化实体后，数据库中保存的是枚举类型的名称。如：

|  |
| --- |
| @Enumerated(EnumType.STRING)  Private Season season; |

说明这个属性是个枚举类型，括号内的表示存入数据库的字符串而不是枚举索引。

#### 如何选择STRING和ORDINAL？

如果使用STRING保存，虽然从数据库中查询数据时非常直观，能够清楚的看出该类型代表意义，但这样也会带来其他的问题。若此时枚举类型的定义改变，例如上例中的枚举类型名称改为：

|  |
| --- |
| public enum CustomerType {  CUST\_COMPETITOR, INVESTOR, PARTNER, VENDER  } |

则此时数据库中保存的“COMPETITOR”的值将不能转化为枚举类型CustomerType中的“CUST\_COMPETITOR”的值。但若使用ORDINAL则不会带来这种问题。所以建议使用ORDINAL类型来持久化枚举类型。

### @OneToMany

@OneToMany标注用于声明一对多关联关系。如订单(Order) 和订单项目(OrderItem)两个实体之间即为一对多关联，一个订单对应多个订单项。反之则为多对一关联。

在Order实体中可添加一个订单项目(OrderItem)的集合类属性，以便获取该订单的明细项目。具体设置如下：

|  |
| --- |
| @Entity  public class Order {  　…  　@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY, mappedBy = "order")  　private Set<OrderItem> orderItems = new HashSet <OrderItem>();  …  } |

@OneToMany标注可设置如下5个选项：

fetch —— 用于设置关联表加载的时机，有两种枚举值：FetchType.EAGERLY表示立即加载，FetchType.LAZY为延迟加载。

cascade —— 用于设置级联操作的方式，可以取CascadeType类的多个枚举值：MERGE为级联修改，PERSIST为级联保存，REFRESH为级联刷新，REMOVE为级联删除，ALL为所有级联操作。

targetEntity —— 用于设置关联的目标实体类。

optional —— 用于设置关联是否是可选的。默认为true，如果被设置为false则必须存在非空的关联关系。

mappedBy ——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个属性可引用到当前实体。

对于mappedBy属性而言，它可在@OneToONe、@OneToMany、@ManyToMany这三个Annotation中使用，它不在@ManyToOne中使用。它指定了该属性出现的当前实体不再控制关联关系―――类似于Hibernate框架中设置了inverse=”true”。一旦为@OneToMany、@ManyToMany指定了mappedBy属性，这两个Annotation将不能和@JoinColumn、@JoinTable一起使用。

### @ManyToOne

@ManyToOne标注用于声明多对一关联关系。如订单项目(OrderItem)和订单(Order)两个实体之间即为多对一关联，多个订单项目对应一个订单。反之则为一对多关联。

在OrderItem实体中可声明一个Order类属性，设置如下：

|  |
| --- |
| @ManyToOne(fetch=FetchType.LAZY)  @JoinColumn(name="OID", nullable=false, insertable=false, updatable=false)  pivate Order order; |

@ManyToOne标注可设置fetch、cascade、targetEntity、optional等4个选项，其含义及设置方法与@OneToMany相同

### @OneToOne

@OneToOne标注用于声明一对一关联关系。这种情况通常可将两个表合并为一个，因而该标注很少使用，只有当表的字段太多，需要拆分为两个表时才会用到。

一对一关联的两个表通常具有相同的主健。如存在一个雇员基本信息表employee和一个雇员其它信息表employee\_info，它们具有相同的主键empid。两实体类声明如下：

|  |
| --- |
| @Entity  public class Employee {  @Id  Int empid;  @OneToOne  EmployeeInfo info;  ...  } |

|  |
| --- |
| @Entity  public class EmployeeInfo {  @Id  int empid;  ...  } |

如果两表主键列名不同而以外键建立关联，则@PrimaryKey JoinColumn标注应改为：@JoinColumn(name="einfoid")

其中的einfoid为employeeinfo表的主键列名。

@OneToOne标注也可设置fetch、cascade、targetEntity、optional及mappedBy 5个选项，其含义及设置方法与@OneToMany相同。

### @ManyToMany

@ManyToMany标注通常很少使用，因为多对多关联通常可通过引入第三方实体而分解为两个一对多关联。

也有极个别情况需要使用@ManyToMany标注。如客户与电话号簿，一个客户可能有多个联系电话或手机号，一个办公室电话号码则可能关联多个同室的客户，它们之间很难引入第三方实体来转化为一对多关联，对应的数据库表通常也没有必要设置外键关联。这时可作如下映射设置：

|  |
| --- |
| @Entity  public class Customer {  …  @ManyToMany(targetEntity=jxust.jpa.entity.Phone.class)  public Set<Phone> phone = new HashSet<Phone>();  …  }  @Entity  public class Phone {  …  @ManyToMany(targetEntity=jxust.jpa.entity.Customer.class)  public Set<Customer> customers = new HashSet<Customer>();  …  } |

### @JoinColumn

|  |
| --- |
| @Entity  @Table(name = "people")  public class People (){       @Id       @GeneratedValue(strategy = GenerationType.AUTO)       private Long id;       private String name;       //由于，people 是这个一对一的关系的主控方，所以，在people表中添加了一个 pet 的外键。       //通过这个外键来维护 people和pet的一对一关系，而不是用第三张码表。这个是通过@JoinColumn注释实现的。       @OneToOne //JPA注释： 一对一关系       @JoinColumn(name="pet\_fk" )// 在pepole中，添加一个外键 "pet\_fk"       private Pet pet;       //省略 get / set  方法...  } |

### @Temporal

@Temporal用来指定的时间类型。支持Date，Time，TimeStamp三个枚举类型。如：

@Temporal(TemporalType.Date)

### @[SecondaryTable](http://www.objectdb.com/api/java/jpa/SecondaryTable)

能否将一个实体类映射到数据库两张或更多表中呢表中呢。在有些情况下如数据库中已经存在原始数据类型，并且要求不能更改，这个时候如果能实现一个实体类对应两张或多张表的话，将是很方便的。JPA2.0 中提供了一个 @SecondaryTablez 注解（annotation）就可以实现这种情况。下面用一个例子说明一下这个注解的使用方法：

清单 6. @SecondaryTable 的使用

@Entity

@SecondaryTables({

@SecondaryTable(name = "Address"),

@SecondaryTable(name = "Comments")

})

public class Forum implements Serializable {

private static final long serialVersionUID = 1L;

@Id

@GeneratedValue

private Long id;

private String username;

private String password;

@Column(table = "Address", length = 100)

private String street;

@Column(table = "Address", nullable = false)

private String city;

@Column(table = "Address")

private String conutry;

@Column(table = "Comments")

private String title;

@Column(table = "Comments")

private String Comments;

@Column(table = "Comments")

private Integer comments\_length;

// Getters and Setters

}

[清单 5](http://www.ibm.com/developerworks/cn/java/j-lo-jpasimpemap/#listing5) 中定义了两个 Secondary 表，分别为 Address 和 Comments，同时在 Forum 实体类中也通过 @Column 注解将某些子段分别分配给了这两张表，那些 table 属性得值是 Adress 的就会存在于 Address 表中，同理 table 属性的值是 Comments 的就会存在于 Comments 表中。那些没有用 @Column 注解改变属性默认的字段将会存在于 Forum 表中。[图 4](http://www.ibm.com/developerworks/cn/java/j-lo-jpasimpemap/#fig4) 就是持久化后在数据库中对应的表的 ER 图，从图中可看出来，这些字段如我们预料的一样被映射到了不同的表中。

图 4. @SecondaryTable 持久化后对赢得 ER 图



Example 1: Single secondary table with a single primary key column

|  |
| --- |
| @[Entity](http://www.objectdb.com/api/java/jpa/Entity)  @[Table](http://www.objectdb.com/api/java/jpa/Table)(name="CUSTOMER")  @[SecondaryTable](http://www.objectdb.com/api/java/jpa/SecondaryTable)(name="CUST\_DETAIL",  pkJoinColumns=@[PrimaryKeyJoinColumn](http://www.objectdb.com/api/java/jpa/PrimaryKeyJoinColumn)(name="CUST\_ID"))  public class Customer { ... } |

Example 2: Single secondary table with multiple primary key columns.

|  |
| --- |
| @[Entity](http://www.objectdb.com/api/java/jpa/Entity)  @[Table](http://www.objectdb.com/api/java/jpa/Table)(name="CUSTOMER")  @[SecondaryTable](http://www.objectdb.com/api/java/jpa/SecondaryTable)(name="CUST\_DETAIL",  pkJoinColumns={  @[PrimaryKeyJoinColumn](http://www.objectdb.com/api/java/jpa/PrimaryKeyJoinColumn)(name="CUST\_ID"),  @[PrimaryKeyJoinColumn](http://www.objectdb.com/api/java/jpa/PrimaryKeyJoinColumn)(name="CUST\_TYPE")})  public class Customer { ... } |

示例：

|  |
| --- |
| package org.mag;  @Entity  @Table(name="ART")  @SecondaryTable(name="ART\_DATA",  pkJoinColumns=@PrimaryKeyJoinColumn(name="ART\_ID", referencedColumnName="ID"))  public class Article {  @Id private long id;  @Column(table="ART\_DATA")  private byte[] content;  ...  } |

### @Embeddable

### @ Inheritance

在面向对象的类之间存在继承关系，@inheritance 的出现就将继承关系能够映射到关系数据库模型中。在实体基类中定义继承策略

InheritanceType stragegy：用以声明继承策略的枚举类型，可取的值包括 SINGLE\_TABLE,JOINED 和 TABLE\_PER\_CLASS：

* InheritanceType.SINGLE\_TABLE：把所有的子类及其父类都映射到同一张表中。
* InheritanceType.JOINED： 类图中每个类都映射到各自的表项，但是每张表中仅仅包含在各个层次的类中定义的列。
* Inheritance.TABLE\_PER\_CLASS：类图中每个类都映射到各自的表现，和 JOINED 不同之处在于每张表包含了对应的实体类所有的列。

# 映射实体关联关系

对象和对象之间除了继承关系之外，还存在着关联关系：包括一对一、一对多、多对一和多对多关系。

JPA定义了one-to-one、one-to-many、many-to-one、many-to-many 4种关系。

对于数据库来说，通常在一个表中记录对另一个表的外键关联；对应到实体对象，持有关联数据的一方称为owning-side，另一方称为inverse-side。

为了编程的方便，我们经常会希望在inverse-side也能引用到owning-side的对象，此时就构建了双向关联关系。在双向关联中，需要在inverse-side定义mappedBy属性，以指明在owning-side是哪一个属性持有的关联数据。

## 基本概念

### 单双向

实体的关联关系具有方向：单向和双向。

在面向对象的世界里，类 A 和类 B 之间形成系必须是通过如下途径来达成的：对象 A1 引用了对象 B1或对象B1引用了对象A1。引用又叫持有。在代码上通常表现如下：

|  |  |
| --- | --- |
| **public** **class** A {  String name;  B b;  } | **public** **class** B {  String name;  A a;  } |

* **单向关联**

在一个单向关联中，两个类是相关的，但是只有一个类知道这种联系的存在。代码上体现如下，只在A 类有 B 类型的属性，而 B 并没有 A 类型的属性，那么说明 A 和 B 实体之间是单向关系

|  |  |
| --- | --- |
| **public** **class** A {  String name;  B b;  } | **public** **class** B {  String name;  } |

* **双向关联**

在双向关联中，两个类彼此知道它们间的联系。

如在A 中定义了 B 属性，在B中也定义了A属性，那么A和B实体之间就构成了双向关系。

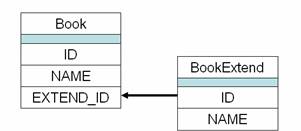
|  |  |
| --- | --- |
| **public** **class** A {  String name;  B b;  } | **public** **class** B {  String name;  A a;  } |

## 一对一关系

在面向对象的世界里，类 A 和类 B 之间形成一对一关系必须满足如下条件：

1. 对象 A1 引用了对象 B1；
2. 类 A 的其它对象 An 不能引用同样的对象 B1。

在关系数据库中，我们通常使用唯一外键的方式来实现一对一关系，下面这个图说明了这种的情况。



单向一对一是关联关系映射中最简单的一种，简单地说就是可以从关联的一方去查询另一方，却不能反向查询。

生活中的一对一关系，举例：人(Person)和身份证(IdCard)。

一对一关联，支持如下三种形式：关联表(有第三张表，大量冗余，很少使用)，外键关联，主键关联。不一定每个JPA产品都提供支持。

### 使用JPA来描述一对一关系

在JPA中，使用@OneToOne 和@JoinColumn两个注解来描述一对一关系。其中@OneToOne 注解是必须使用的，它被用来声明类和类之间存在着一对一关系，@JoinColumn 注解是可选的，开发者使用 JoinColumn 注解来声明两个类在数据库中对应的表之间关联时的细节，包括主表中关联字段的名称、从表中使用什么字段来进行关联等。

### 一对一映射关系的分类

一对一关系映射分为单向一对一和双向一对一。在配置关系时必须确立控制方和被控制方。单向和双向的区别为看主控方和被控方两边是否都配置了@OneToOne，如果都有为双向一对一，反之为单向。

### 单向一对一

 为什么这个一对一关系是单向的？如果，只能通过人找到身份证，而不能通过身份证找到人，那么这种关系就是单向的。

类似于上述这种情况，或者业务关系。实体间的关系是一对一，并且，我们只需要通过一个实体得到其对应的实体，而且并不需要反向执行这个操作的时候。我们就需要使用单向一对一关系。

Person类：

|  |
| --- |
| @Entity  @Table(name="t\_person\_one2one\_1")  **public** **class** Person {  @Id  @GeneratedValue(strategy = GenerationType.*IDENTITY*)  **private** Integer id;    **private** String name;    **private** **int** sex;    **private** **int** age;  //在Person表中添加一个外键  @OneToOne(cascade=CascadeType.*PERSIST*)  @JoinColumn(name="idcard")  **private** IdCard idCard;  ……..  } |

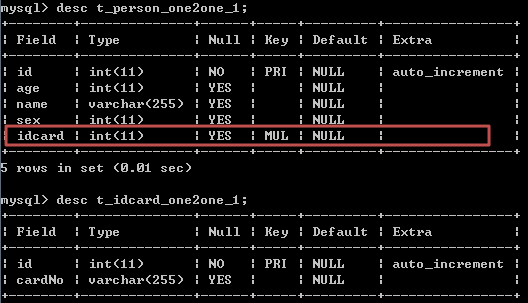
由于是单向关系，IdCard类不需要额外使用@OneToOne进行修饰。IdCard类也不持有Perosn类的引用。

|  |
| --- |
| @Entity(name="t\_idcard\_one2one\_1")  **public** **class** IdCard {  @Id  @GeneratedValue(strategy=GenerationType.*IDENTITY*)  **private** **int** id;  @Column  **private** String cardNo;  …..  } |

数据库中生成2张表：

t\_person\_one2one\_1和t\_idcard\_one2one\_1，在Person对应的表中增加了一个外键idCard。

表结构如下：



### 双向一对一

仍然以人(Person)和身份证(IdCard)为例。

通过人可以找到身份证，通过身份证可以找到人。

对于这种数据模型来说，双向一对一关系已经很明显了。双向一对一，表结构不会发生任何改变，只是一对一双方均可以找到对方，即持对方的引用。

Person类：

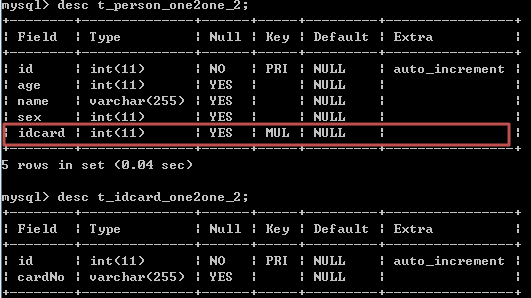
|  |
| --- |
| @Entity  @Table(name="t\_person\_one2one\_2")  **public** **class** Person {  @Id  @GeneratedValue(strategy = GenerationType.*IDENTITY*)  **private** Integer id;    **private** String name;  **private** **int** sex;  **private** **int** age;  //由于，Person是这个一对一的关系的主控方，所以，在Person表中添加了一个IdCard的外键。  @OneToOne(cascade=CascadeType.*PERSIST*)  @JoinColumn(name="idcard")  **private** IdCard idCard;  ……  } |

IdCard也需要使用@OneToOne进行修饰，注意mappedBy属性。

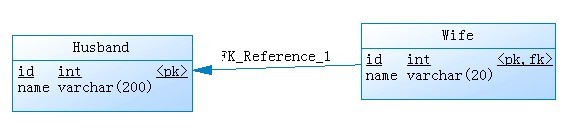
|  |
| --- |
| @Entity(name="t\_idcard\_one2one\_2")  **public** **class** IdCard {  @Id  @GeneratedValue(strategy=GenerationType.*IDENTITY*)  **private** **int** id;  @Column  **private** String cardNo;  //注意：由于是双向的一对一关系。作为关系的被维护端。需要在IdCard实体中添加Person关联的属性  //mappedBy指定了，这个实体是被关系维护端的那个属性所维护。  @OneToOne(mappedBy="idCard")  **private** Person pserson;  ……  } |

数据库中生成2张表：

t\_person\_one2one\_1和t\_idcard\_one2one\_1，在Person对应的表中增加了一个外键idCard。表结构如下：



### 主键关联



create table Husband  
(  
   id                   **int** not null auto\_increment,  
   name                 **varchar**(**200**),  
   primary key (id)  
);

create table Wife  
(  
   id                   **int** not null,  
   name                 **varchar**(**20**),  
   primary key (id)  
);

alter table Wife add constraint FK\_Reference\_1 foreign key (id)  
      references Husband (id) on delete restrict on update restrict;

Wife.java

package com.jpa.one2one.bean;  
  
import java.io.Serializable;  
  
import javax.persistence.Column;  
import javax.persistence.Entity;  
import javax.persistence.Id;  
import javax.persistence.OneToOne;  
import javax.persistence.PrimaryKeyJoinColumn;  
import javax.persistence.Table;  
  
@SuppressWarnings("serial")  
@Entity  
@Table  
public class Wife implements Serializable{  
    private int id;  
    private String name;  
    private Husband husband;  
    @Id  
    @Column  
    public int getId() {  
        return id;  
    }  
    public void setId(int id) {  
        this.id = id;  
    }  
    @Column(name="name")  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    @OneToOne  
    @PrimaryKeyJoinColumn  
    public Husband getHusband() {  
        return husband;  
    }  
    public void setHusband(Husband husband) {  
        this.husband = husband;  
    }  
      
}

Husband.java

package com.jpa.one2one.bean;  
  
import javax.persistence.CascadeType;  
import javax.persistence.Column;  
import javax.persistence.Entity;  
import javax.persistence.GeneratedValue;  
import javax.persistence.GenerationType;  
import javax.persistence.Id;  
import javax.persistence.OneToOne;  
import javax.persistence.PrimaryKeyJoinColumn;  
import javax.persistence.Table;  
  
@Entity  
@Table  
public class Husband {  
      
    private int id;  
    private String name;  
    private Wife wife;  
    @Id  
    @Column  
    @GeneratedValue(strategy=GenerationType.AUTO)  
    public int getId() {  
        return id;  
    }  
    public void setId(int id) {  
        this.id = id;  
    }  
    @Column(name="name")  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    @OneToOne(cascade=CascadeType.ALL)  
    @PrimaryKeyJoinColumn  
    public Wife getWife() {  
        return wife;  
    }  
    public void setWife(Wife wife) {  
        this.wife = wife;  
    }  
      
}

HusbandDAO

package com.jpa.one2one.dao;  
  
import javax.persistence.EntityManager;  
import javax.persistence.EntityTransaction;  
  
import org.junit.Test;  
  
import com.jpa.one2one.bean.Husband;  
import com.jpa.one2one.bean.Wife;  
import com.jpa.one2one.util.JPAUtil;  
  
public class HusbandDAO {  
      
    @Test  
    public void insert(){  
        EntityManager entityManager=JPAUtil.getInstance();  
        EntityTransaction transaction=entityManager.getTransaction();  
        try {  
            transaction.begin();  
            Husband husband=new Husband();  
            husband.setName("张三");  
            entityManager.persist(husband);  
            Wife wife=new Wife();  
            //wife.setHusband(husband);  
            wife.setName("如花");  
            wife.setId(husband.getId());  
            entityManager.persist(wife);  
            transaction.commit();  
        } catch (Exception e) {  
            e.printStackTrace();  
            transaction.rollback();  
        }  
    }  
}

JPAUtil

package com.jpa.one2one.util;  
  
import javax.persistence.EntityManager;  
import javax.persistence.EntityManagerFactory;  
import javax.persistence.Persistence;  
  
public class JPAUtil {  
      
    private static EntityManager entityManager;  
    public static EntityManager getInstance(){  
        if(entityManager!=null){  
            return entityManager;  
        }else{  
            return makeInstance();  
        }  
    }  
    private static synchronized EntityManager makeInstance() {  
        if(entityManager==null){  
            EntityManagerFactory entityManagerFactory=Persistence.createEntityManagerFactory("JPAPU");  
            return entityManagerFactory.createEntityManager();  
        }  
        return null;  
    }  
}

persistence.xml

<?xml version="1.0" encoding="UTF-8"?>  
<persistence xmlns="http://java.sun.com/xml/ns/persistence"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence  
    http://java.sun.com/xml/ns/persistence/persistence\_1\_0.xsd" version="1.0">  
      
    <persistence-unit name="JPAPU" transaction-type="RESOURCE\_LOCAL">  
        <provider>org.hibernate.ejb.HibernatePersistence</provider>  
        <class>com.jpa.one2one.bean.Wife</class>  
        <class>com.jpa.one2one.bean.Husband</class>  
          <properties>  
            <property name = "hibernate.connection.driver\_class" value = "com.mysql.jdbc.Driver"/>  
            <property name = "hibernate.connection.url" value = "jdbc:mysql://localhost:3306/JPA"/>  
            <property name = "hibernate.connection.username" value = "root"/>  
            <property name = "hibernate.connection.password" value = "root"/>  
            <property name="hibernate.show\_sql" value="true"/>  
            <property name="hibernate.format\_sql" value="true"/>  
          </properties>  
    </persistence-unit>  
    
</persistence>

另一种形式：

## 一对多(OneToMany)

在对象模型中，一对多的关联关系，使用集合来表示。比如Classes（班级）和Student（学生）之间是一对多的关系：

|  |  |
| --- | --- |
| public class Classes **{**  private int id**;**  private String name**;**  private Set**<**Student**>** students **=** **new** HashSet**<**Student**>();**  **......**  **}** | public class Student **{**  private int id**;**  private String name**;**  **......**  **}** |

### 一对多映射关系的分类

一对多关系映射分为单向一对多和双向一对多。在配置关系时必须确立控制方和被控制方。单向一对多只需要使用@OneToMany注解，双向一对多需要同时使用@OneToMany和@ManyToOne注解

### 单向一对多

通过在多方的表中增加一个外键列实现一对多的单向关联。同时使用了@OneToMany和@JoinColumn注解。

Classes类：

|  |
| --- |
| @Entity  @Table(name = "t\_classes\_one2many\_1")  **public** **class** Classes {  @Id  @GeneratedValue(strategy = GenerationType.*IDENTITY*)  **private** **int** id;  **private** String name;  @OneToMany(cascade = CascadeType.*ALL*)  @JoinColumn(name="cid")//在student表增加一个外键列来实现一对多的单向关联  **private** Set<Student> students = **new** HashSet<Student>();  ……….  } |

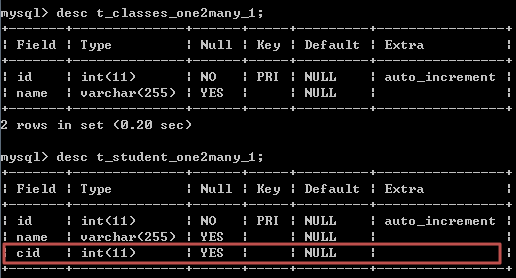
由于是单向一对多关系，Student类不持有Classes的引用，不需要配置关联关系注解。

|  |
| --- |
| @Entity  @Table(name="t\_student\_one2many\_1")  **public** **class** Student {  @Id  @GeneratedValue(strategy=GenerationType.*IDENTITY*)  **private** **int** id;  **private** String name;  …….  } |

在数据库中只生成了两张表：

t\_classes\_one2many\_1和t\_student\_one2many\_1

表结构如下图所示，在t\_student\_one2many\_1表中增加了一个外键cid。



### 双向一对多

仍然以Classes（班级）和Student（学生）为例。

双向一对多关系中，一方使用OneToMany，一方使用ManyToOne。一般来说，在双向一对多关系中，"多"这一端，一般都是关系的维护端。

双向一对多，表的结构不会做任何改变。只是在Student一端持有了对Classes的引用。

Classes类：

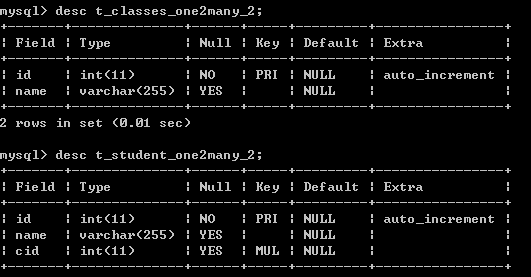
|  |
| --- |
| @Entity  @Table(name = "t\_classes\_one2many\_2")  **public** **class** Classes {  @Id  @GeneratedValue(strategy = GenerationType.*IDENTITY*)  **private** **int** id;  **private** String name;  //配置了mappedBy，说明Classes不负责维护关系  @OneToMany(mappedBy="classes",cascade=CascadeType.*ALL*)  **private** Set<Student> students = **new** HashSet<Student>();  ……  } |

Student类：

|  |
| --- |
| @Entity  @Table(name="t\_student\_one2many\_2")  **public** **class** Student {  @Id  @GeneratedValue(strategy=GenerationType.*IDENTITY*)  **private** **int** id;  **private** String name;  //在Student一端使用了@ManyToOne注解  @ManyToOne(cascade=CascadeType.*PERSIST*)  //在Student表中增加外键cid  @JoinColumn(name="cid",referencedColumnName="id")  **private** Classes classes;  ………  } |

同样，数据库中生成了2张表：

t\_classes\_one2many\_2和t\_student\_one2many\_2，表结构和单向一对多是一样的。



## 多对一(ManyToOne)

多对一，在一对多的双向关联中已有描述。没有双向多对一，因为双向多对一和双向一对多是一样的。

## 多对多(ManyToMany)

所谓多对多关系，就是指一个A实体可以对应N个B实体，而一个B实体也可以对应N个A实体。

### 应用场景

例如生活中，作家是一种实体，书籍也是一种实体。考虑映射关系方面：一个作家可以写很多本书（对应N本书）；而一本书也可能有多个作者（对应N个作家）。因而作家和书之间是多对多（N-N）关系。

### 基本概念

#### 关系维护

通常情况下，映射关系需要有专门的措施进行存储维护。在单向一对多、双向一对多、单向多对一关系中，这种映射关系的维护策略为：**在“多”的一方建立维护列**，这是最简单的方案。

假设一种1-N的关系为，一名学生属于一个班级，而一个班级包含很多学生。那么此时一个班级就是一条数据库表记录，在这一条班级记录中不需要记录所有学生的ID集合——这在数据库表中是不推荐的做法，也不易实现。而如果让每一条学生记录中都包含“学生所属班级”的唯一ID，那么表结构将非常符合数据库规范。因而无论单双向的1-N关系中，维护列都应当放在多的一方。

比较特殊的是多对多关系。按照上述原理，若干条A记录和若干条B记录：每一条A记录都对应多条B记录；而每一条B记录都对应多条A记录。那么这种互补的映射关系维护信息放在A或者B中都不合适。这就好像是简单数学的乘法多项式：

(a + b) \* (c + d) = ac + ad + bc + bd

类似这种运算的方式和结果在数学中称之为“**笛卡尔积**”。我们的多对多映射关系中，最好**借助第三张表**来维护笛卡尔积映射关系。

### 单向多对多

#### 应用场景

有很多作家（Author）和很多书（Book）。作家知道自己写了那几本书，即Author中含有Book的引用；但书中没有作家的署名，即Book中不含有Author的引用。因此Author是知情方，Book是不知情方。“作家-书”为“单向多对多”关系映射。

#### 实体类配置

* Author（作家）

|  |
| --- |
| @Entity  @Table(name = "tb\_author\_1")  **public** **class** Author {  @Id  @GeneratedValue(strategy = GenerationType.*AUTO*)  **private** Integer aid;  **private** String name;  @ManyToMany(fetch = FetchType.*EAGER*,  cascade = {CascadeType.*PERSIST*,  CascadeType.*MERGE*})  @JoinTable(  name = "tb\_book\_author\_1",  joinColumns = @JoinColumn(  name = "author\_id",  referencedColumnName = "aid"),  inverseJoinColumns = @JoinColumn(  name = "book\_id",  referencedColumnName = "bid"))  **private** Set<Book> books = **new** HashSet<Book>();  // 省略setter getter  } |

**@Entity**告诉JPA这是一个实体类，可以在OpenJPA的配置文件persistence.xml中进行罗列。

**@Table**用于指定该实体类对应的表名称。

**@Id**用于指定类（数据库表）中的一个属性为主键。

**@GeneratedValue**用于指定主键生成策略，例如使用数据库默认的策略（MySQL默认生成数字主键）或者UUID等。

**@ManyToMany**主键用于指定多对多关系映射，可选设置加载策略为“早加载”或“懒加载”；可选设置级联关系，即在当前类数据发生何种操作时，将同时作用于目标属性的类数据。

**@JoinTable**即为第三张表，用于维护笛卡尔积关系。其中name为表名称，joinColumns用于指定维护当前表（Author）的外键列；inverseJoinTable用于指定维护目标表（Book）的外键列。在**@JoinColumn**注解中，name用于指定第三张表中的列名称，而referencedColumnName用于指定原始数据表中被引用的列名称。

例如：Author和Book通过第三张表tb\_book\_author\_1进行笛卡尔积关系维护；当前表tb\_author\_1需要在笛卡尔积表中注册一个列，名称为author\_id；目标表tb\_book\_1需要在笛卡尔积表中注册一个列，名称为book\_id；而author\_id的引用来自于当前表中的aid字段；目标表中的book\_id引用自目标表中的bid字段。

另外，由于作家和书两种实体均为不可重复的对象，因而使用Set集合。

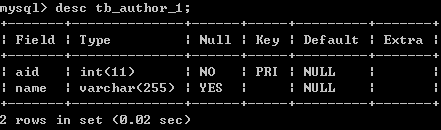
* Book（书籍）

由于是单向多对多关系，因而Book将作为不知情的一方，不需要进行特殊关系映射配置：

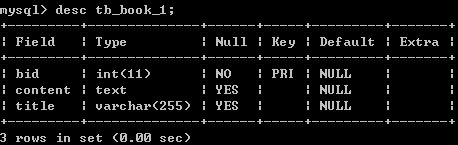
|  |
| --- |
| @Entity  @Table(name = "tb\_book\_1")  **public** **class** Book {  @Id  @GeneratedValue(strategy = GenerationType.*AUTO*)  **private** Integer bid;  **private** String title;  @Column(columnDefinition = "text")  **private** String content;  // 省略setter getter  } |

#### 映射表结构

如上的JPA配置映射到数据库表中，Author类的表结构如下图所示：



Book类的表结构如下图所示：



而笛卡尔积关系表的结构如下图所示：



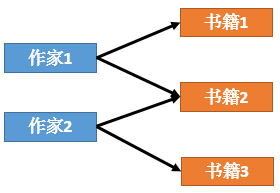
#### CRUD操作

本小节提供一些增删改查操作，用于验证单向多对多关系映射的维护。

* 增加（Create）

|  |
| --- |
| @Test  **public** **void** create() {  // 两个作家，三本书  Author author1 = **new** Author();  author1.setName("作家1");    Author author2 = **new** Author();  author2.setName("作家2");    Book book1 = **new** Book();  book1.setTitle("书籍标题1");  book1.setContent("书籍内容1");    Book book2 = **new** Book();  book2.setTitle("书籍标题2");  book2.setContent("书籍内容2");    Book book3 = **new** Book();  book3.setTitle("书籍标题3");  book3.setContent("书籍内容3");    // 设置关系映射  author1.getBooks().add(book1);  author1.getBooks().add(book2);    author2.getBooks().add(book2);  author2.getBooks().add(book3);    // 持久化到数据库  EntityManagerFactory factory = Persistence.*createEntityManagerFactory*("jpaunit");;  EntityManager em = factory.createEntityManager();    em.getTransaction().begin();  em.persist(author1);  em.persist(author2);  em.getTransaction().commit();  em.close();  } |

实体类映射关系如图所示：



* 查询（Read）

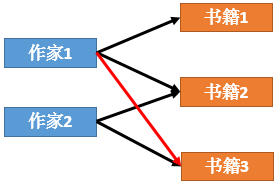
|  |
| --- |
| @Test  **public** **void** read() {  EntityManagerFactory factory = Persistence.*createEntityManagerFactory*("jpaunit");;  EntityManager em = factory.createEntityManager();    String sql = " select a from Author a ";  @SuppressWarnings("unchecked")  List<Author> list = em.createQuery(sql).getResultList();  **for** (Author author : list) {  System.*out*.println("ID为" + author.getAid() + "的" + author.getName() + "所写的书有：");  Set<Book> set = author.getBooks();  **for** (Book book : set) {  System.*out*.println("\tID为" + book.getBid() + "的《" + book.getTitle() + "》——" + book.getContent());  }  }  } |

查询每一个作家写的书有哪些，但无法查询每一本书的作者有那些人。

* 修改（Update）

|  |
| --- |
| @Test  **public** **void** update() {  EntityManagerFactory factory = Persistence.*createEntityManagerFactory*("jpaunit");;  EntityManager em = factory.createEntityManager();  em.getTransaction().begin();    String sql1 = " select a from Author a where a.name='作家1' ";  Author author1 = (Author) em.createQuery(sql1).getResultList().get(0);  String sql2 = " select b from Book b where b.title='书籍标题3' ";  Book book3 = (Book) em.createQuery(sql2).getResultList().get(0);  author1.getBooks().add(book3);    em.merge(author1);  em.getTransaction().commit();  em.close();  } |

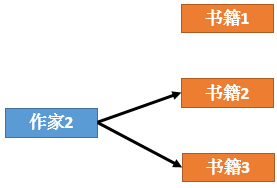
将映射关系修改为：



* 删除（Delete）

|  |
| --- |
| @Test  **public** **void** delete() {  EntityManagerFactory factory = Persistence.*createEntityManagerFactory*("jpaunit");;  EntityManager em = factory.createEntityManager();  em.getTransaction().begin();    String sql = " select a from Author a where a.name='作家1' ";  Author author = (Author) em.createQuery(sql).getResultList().get(0);    em.remove(author);  em.getTransaction().commit();  em.close();  } |

删除之后，映射关系为：



### 双向多对多

#### 应用场景

作家（Author）知道自己写了那几本书（Book），即Author中含有Book的引用；书中也有作家的署名，即Book中也含有Author的引用。因此Author和Book都是知情方，都有对方的引用。“作家-书”为“双向多对多”关系映射。

#### 实体类配置

* Author（作家）

Author仍然为知情方，除了表名称之外，配置与此前丝毫没有区别，请参考前文第1.3.2.1小节内容。

* Book（书籍）

Book由此前的不知情方变为知情方，内部需要增加一个Author的集合。而映射关系已经在Author类当中配置好了，Book中仅仅需要在Author集合属性上使用一个**mappedBy**即可：

|  |
| --- |
| @Entity  @Table(name = "tb\_book\_2")  **public** **class** Book {  @Id  @GeneratedValue(strategy = GenerationType.*AUTO*)  **private** Integer bid;  **private** String title;  @Column(columnDefinition = "text")  **private** String content;  @ManyToMany(mappedBy = "books")  **private** Set<Author> authors = **new** HashSet<Author>();  // setter getter |

@ManyToMany是应用在Set<Author>类型属性上的，mappedBy属性指定的名称是在目标集合泛型（Author）类中对应的books属性。表示映射关系到Author类当中的books属性中去找，那里已经配置好了，没有必要再做冗余配置。

#### 映射表结构

知情与不知情，只是体现在Java类的属性中是否含有对方的引用上的，而数据库表结构丝毫没有变化。

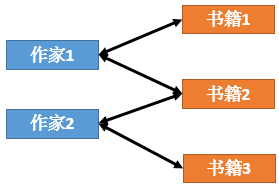
#### CRUD操作

本节提供增删改查方法，用于验证双向多对多关系映射中的维护操作。

* 增加（Create）

|  |
| --- |
| @Test  **public** **void** create() {  // 两个作家，三本书  Author author1 = **new** Author();  author1.setName("作家1");    Author author2 = **new** Author();  author2.setName("作家2");    Book book1 = **new** Book();  book1.setTitle("书籍标题1");  book1.setContent("书籍内容1");    Book book2 = **new** Book();  book2.setTitle("书籍标题2");  book2.setContent("书籍内容2");    Book book3 = **new** Book();  book3.setTitle("书籍标题3");  book3.setContent("书籍内容3");    // 设置关系映射  author1.getBooks().add(book1);  author1.getBooks().add(book2);    author2.getBooks().add(book2);  author2.getBooks().add(book3);    book1.getAuthors().add(author1);  book2.getAuthors().add(author1);  book2.getAuthors().add(author2);  book3.getAuthors().add(author2);    // 持久化到数据库  EntityManagerFactory factory = Persistence.*createEntityManagerFactory*("jpaunit");;  EntityManager em = factory.createEntityManager();    em.getTransaction().begin();  em.persist(author1);  em.persist(author2);  em.persist(book1);  em.persist(book2);  em.persist(book3);  em.getTransaction().commit();  em.close();  } |

映射关系如下图所示：



* 查询（Read）

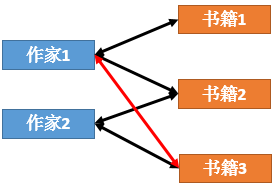
由于是双向关系，不仅可以列出每一个作家写了哪些书籍，而且还可以列出每一本书籍都有哪些作者：

|  |
| --- |
| @Test  **public** **void** read() {  EntityManagerFactory factory = Persistence.*createEntityManagerFactory*("jpaunit");;  EntityManager em = factory.createEntityManager();    String sql1 = " select a from Author a ";  @SuppressWarnings("unchecked")  List<Author> aList = em.createQuery(sql1).getResultList();  **for** (Author author : aList) {  System.*out*.println("ID为" + author.getAid() + "的" + author.getName() + "所写的书有：");  Set<Book> set = author.getBooks();  **for** (Book book : set) {  System.*out*.println("\tID为" + book.getBid() + "的《" + book.getTitle() + "》——" + book.getContent());  }  }    String sql2 = " select b from Book b ";  @SuppressWarnings("unchecked")  List<Book> bList = em.createQuery(sql2).getResultList();  **for** (Book book : bList) {  System.*out*.println("ID为" + book.getBid() + "的《" + book.getTitle() + "》——" + book.getContent() + "的作者有：");  **for** (Author author: book.getAuthors()) {  System.*out*.println("\tID为" + author.getAid() + "的" + author.getName());  }  }  } |

* 修改（Update）

|  |
| --- |
| @Test  **public** **void** update() {  EntityManagerFactory factory = Persistence.*createEntityManagerFactory*("jpaunit");;  EntityManager em = factory.createEntityManager();  em.getTransaction().begin();    String sql1 = " select a from Author a where a.name='作家1' ";  Author author1 = (Author) em.createQuery(sql1).getResultList().get(0);  String sql2 = " select b from Book b where b.title='书籍标题3' ";  Book book3 = (Book) em.createQuery(sql2).getResultList().get(0);    author1.getBooks().add(book3);  book3.getAuthors().add(author1);    em.merge(author1);  em.merge(book3);  em.getTransaction().commit();  em.close();  } |

将映射关系修改为：

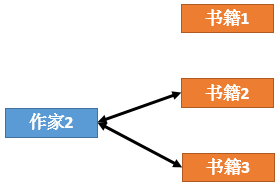


由于是双向关系，需要同时更新作家1和书籍3。

* 删除（Delete）

|  |
| --- |
| @Test  **public** **void** delete() {  EntityManagerFactory factory = Persistence.*createEntityManagerFactory*("jpaunit");;  EntityManager em = factory.createEntityManager();  em.getTransaction().begin();    String sql = " select a from Author a where a.name='作家1' ";  Author author = (Author) em.createQuery(sql).getResultList().get(0);    em.remove(author);  em.getTransaction().commit();  em.close();  } |

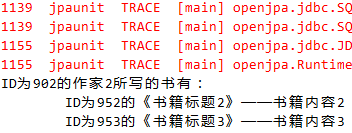
仍然只删除作家1而保留书籍1：



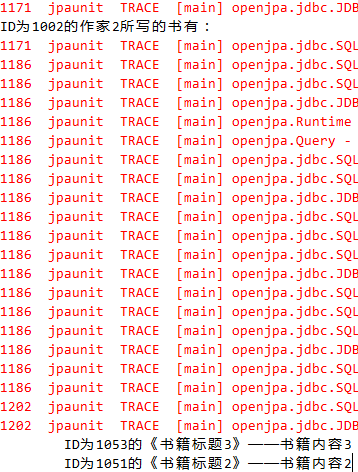
### 注意事项

#### 加载策略

本章在单向多对多关系中，使用了早加载（EAGER）方式，因而在读取数据的时候，从控制台中将可以看到信息的输出是连续的，中间没有穿插数据库操作语句。如下图所示：



然而本章在讨论双向多对多关系中，使用了懒加载（LAZY）方式，因而可以从控制台中看到读取数据的输出信息里，穿插了数据库操作语句。如下图所示：



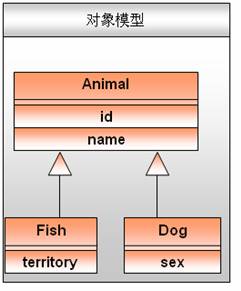
有关于详尽的JPA加载策略及级联配置，请参考其他相关文档。

## 映射继承关系

在 Java 世界中，对象和对象之间存在着继承、引用关系，这样的关系无法通过简单的方式直接映射到关系型数据库中。因此在关系型数据库与面向对象之间存在着阻抗失谐。

Java 是一种面向对象的语言，面向对象的语言特的性之一就是继承。而关系型数据库中的表与表之间的关系，并没有这种继承关系，不能说一张表继承另一张表，它们之间的关系只是关联关系。那么现在 JPA 要将 POJO 的 Java 类映射到数据库中，对于 Java 类中的这种继承关系是如何映射到数据库中的呢。JPA 规范中现在提供了三种不同的策略来实现继承与关系之间的映射。

假设有如下三个 Java 类：Animal、Fish 和 Dog。Animal 仅仅具备两个属性：id 和 name。Fish 是一种 Animal，但是人们比较关注它的生活区域是在海里还是在河里，因此它除了继承自 Animal 之外，还有自己独特的属性territory。Dog 也是一种 Animal，人们比较关注它的性别，因此它除了继承自 Animal 之外，还有自己独特的属性 sex。我们可以用下面这个图描述三者之间的关系模型。



在 Java 应用中，由于动态绑定的支持，Fish、Dog 都可以被作为 Animal 对象处理。但是如果我们换到关系型数据库的视角，情况发生了改变： 通常情况下，Animal、Fish、Dog 毫无关联，它们都保存在各自对应的表中，假设分别对应 Animal 表、Fish 表和 Dog 表，如果要维护 Animal 和 Fish 的继承关系，我们就需要使用 SQL 的联合查询语句查出 Animal 的所有属性和 Fish 的所有属性，这样就必须使用某种外键进行关联：

|  |
| --- |
| Select animal.\*,fish.\* form animal,fish where animal.id = fish.id |

从这个简单的例子中我们就可以看出，开发者需要同时掌握面向对象和关系型数据库的两种思想，而且还必须保证它们之间的映射是正确的，否则无法保证应用的正确性。

在JPA中，开发者仅需要处理对象模型，而不需要处理和关系型数据库有关的内容，极大地降低了对象继承关系持久化的难度。

我们从关系数据库角度看对象继承关系的持久化这个问题域：对象继承通常意味着子类比父类提供更多的属性，持久化对象继承关系的实质就是如何根据对象的类型动态的处理这些多出来的属性。OpenJPA 框架支持使用三种不同的策略处理对象继承关系：

### JPA对实体继承的支持

JPA使用Inheritance 注解对JPA的实体继承提供支持。

javax.persistence.Inheritance 注解用来指定对象继承关系持久化的方式。它的 strategy 属性用于指定持久化对象继承关系在关系数据库中的表现形式，可选择项包括 SINGLE\_TABLE、JOINED 和 TABLE\_PER\_CLASS。它们三个都是 javax.persistence.InheritanceType 中定义的常量。

* **SINGLE\_TABLE**

strategy 设置为 SINGLE\_TABLE 选项表示所有类及其子类保存在同一个数据库表中，对象的类型使用表中的特殊字段 DTYPE 进行识别。

* **TABLE\_PER\_CLASS**

strategy 设置为该选项表示每个类使用一个表。

* **JOINED**

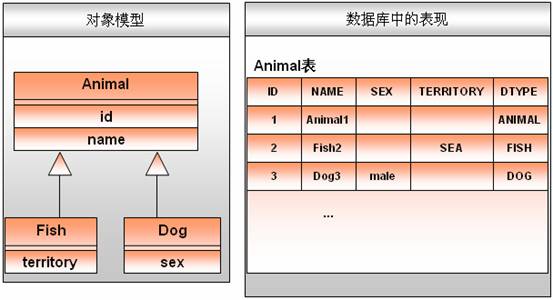
strategy 设置为该选项表示父类和子类分别保存在不同的数据库表中，子类中不保存父类对应数据库表中已有的属性，仅通过主键进行关联。

javax.persistence.Inheritance 注释是类级别的注释。需要为每一个成为父类的实体类提供 javax.persistence.Inheritance 注释并且指定 strategy 属性。在同一个企业应用中，开发者可以根据实际情况选择这三种策略中的一种，或者是几种同时使用。

### ****SINGLE\_TABLE策略****

在这种情况下，类及其子类都保存在同一张数据表中，该表提供足够的字段保存类及其子类的所有属性，同时提供一个特别字段保存当前记录对应类的实际类名（默认名 DTYPE，也可以在开发时指定其它名称）。

Animal、Fish、Dog 三个类的所有对象实例都被保存在 Animal 数据表中，该表将会有 5 个属性，其中 ID，NAME 字段对应 ANIMAL 类的两个属性，ID、NAME、SEX 对应 Dog 类的属性，ID、NAME、STERRITORY 对应 Fish 类的属性。DTYPE 是 OpenJPA 加入的字段，用于确定当前记录的实际类类型，在这里例子中，它的内容是“ANIMAL”、“FISH”或者是“DOG”。 第一种策略的数据库表现如下：



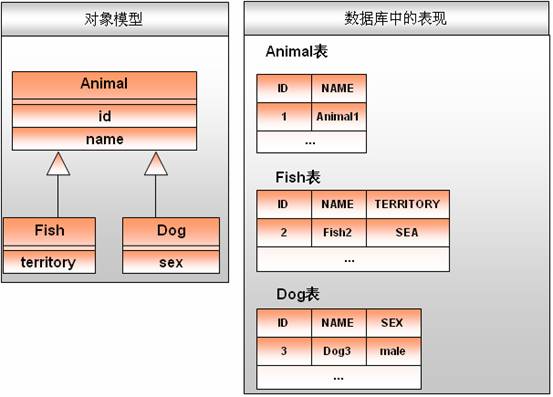
单表映射策略是缺省的映射策略，这种策略对实体之间的多种关联关系能提供很好的支持，同时在查询方面也有很好的效率。但是这种映射策略在数据库表中会有很多的空字段的存在，如图 3 和图 4 所示。这样势必会造成数据库资源的大量浪费，同时这个映射策略也要求子类中的所有属性也必须是可空 (null able) 的。

### ****TABLE\_PER\_CLASS策略****

这种情况下，开发者不理会类之间是否存在继承关系，为每一个类的持久化使用唯一的表，类和子类分别保存在不同的数据库表中，互相之间没有关联，父类对象保存在父类对应的表中，子类对象的信息保存在子类对应的表中。这种策略在 JPA2.0 中仍然是自由选取得，也就是说实现 JPA2.0 规范的持久化引擎，如 Toplink，Hibernate 等 , 仍然可以自由选取是否实现这种策略。

这种映射策略和连接映射策略很类似，不同的是子类对应的表中要继承根实体 (root entity) 中的属性，根实体 (root entity) 对应的表中也不需要区分子类的列，表之间没有共享的表，也没有共享的列。

Animal、Fish、Dog 三个类的对象实例都被保存在各自对应的数据表中。下面这个图显示了这种情况下对象继承关系数据库中的表现。第二种策略的数据库表现如下：



#### **重写根实体的属性**

因为每个子实体对应的表中都继承了根实体中的属性，为了区分可以使用 @AttributeOverride 注解来修改根实体中的属性在不同的子实体中的对应的列名。

Dog实体 (entity)

@Entity

@AttributeOverrides({

@AttributeOverride(name="id",

column=@Column(name="Magazine\_id")),

@AttributeOverride(name="title",

column=@Column(name="Magazine\_title")),

@AttributeOverride(name="description",

column=@Column(name="Magazine\_desc")),

})

public class Dog extends Animal {

private String isbn;

private String publisher;

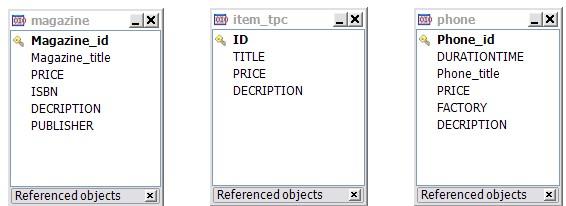
// Getters and Setters

}

Dob 实体将从 Animal 根实体中继承而来的 id、title、description 属性分别重写为 Magazine\_id、Magazine\_title 和 Magazine\_desc。

从图 7 就可以看出，根实体中的属性，在子实体中的列名已经被相应地修改了。

图 7. 映射的 ER 图

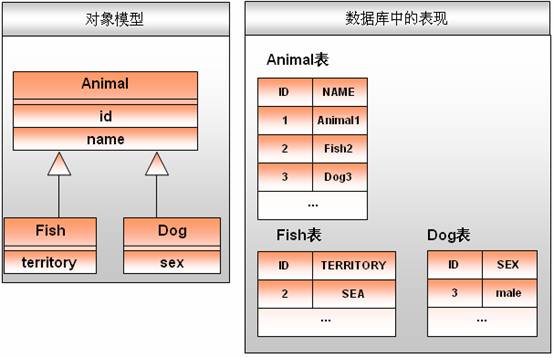


### JOINED策略

这种情况下，父类和子类对应不同的表，但是子类对应的表中不再保存父类对应表中已经存在的字段信息，两个表之间通过关键字段关联起来，也就是数据库技术中通常所说的外健。

JOINED策略是 OpenJPA 中持久化对象继承的最佳实践，这种实现方式是最理想化的一种，既符合 Java 开发者面向对象的习惯，也能够符合关系数据库设计的范式要求，而且数据库中的数据冗余最小。

Animal、Fish、Dog 三个类的对象实例都被在 Animal 表中有记录；而 Fish 对象的 TERRITORY 属性者被 FISH 表所保存，FISH 表通过 ID 和 Animal 表中的数据进行关联；而 Dog 对象的 SEX 属性者被 Dog 表所保存，Dog 表通过 ID 和 Animal 表中的数据进行关联。下面这个图显示了这种情况下对象继承关系数据库中的表现。JOINED策略的数据库表现如下：



这种映射策略，在继承比较多时，查寻起来效率就会差一些，因为在查询的过程中需要多表的连接，连接的表数越多，查询效率越低下。

### 继承映射中的其它类型

上面的例子中的所使用的根实体类都是具体 (concrete) 实体类，下面将介绍一下当根实体是其他类型类的情况下的映射规则。

**抽象实体类 (Abstract entity)** ：

上面的例子中所使用的 Item 是个具体类 (concrete class)，并且使用 @Entity 注解 (annotation) 进行了注释。那么当 Item 类不是具体类 (concrete class)，而是一个抽象类 (abstract class) 的时候，也就是当 Item 类的声明中使用了 abstract 关键字的时候，是如何影射的呢？事实上根实体是否是抽象实体类，在数据库中映射成的表没有任何区别。也就是说上面的例子中如果根实体类 Item 是个抽象实体类，使用了 abstract 关键字的话，在数据库中生成的表和上面的例子是相同的。唯一的区别就是，如果根实体是抽象实体类的话，就不能使用 new 关键字来生成这个实体类的对象了。他们的区别只是在 Java 语言语法上的区别，在持久化上没有任何区别。

**非实体类 (Nonentity):**

非实体类 (Nonentity) 也叫瞬态类 (transient class)，就是普通的 POJO 类，没有使用 @Entity 注解 (annotation) 注释，这种类在持久化的时候不会被映射到数据库中，因为根据之前介绍过的持久化的原则，一个类如果想被持久化到数据库中，必须使用 @Entity 注解。那么当一个实体类继承了一个这样的非实体类的情况下，该如何影射呢。

清单 15.Item 类是一个没有使用 @Entity 普通的 POJO 的 Java 类

public class Item{

private String title;

private Float price;

private String decription;

// Getters and Setters

}

清单 16 中的 Magazine 实体类继承了 Item 非实体类，根据 Java 的语法规则，Magazine 实体类的对象将可以访问 Item 类中的属性。并且根据上面介绍过，Item 类可以是具体类 (concrete class) 也可以是抽象类 (abstract class)，这对持久化后的结果没有影响。

清单 16.Magazine 实体继承了 Item 非实体类

@Entity

public class Magazine extends Item {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private Long id;

private String isbn;

private String publisher;

// Getters and Setters

}

Magazine 实体类继承了 Item 类，但是在持久化的时候，只有 Magazine 类中的属性才能持久化到数据库中去。从 Item 类中继承下来的属性不会持久化数据库中。清单 17 就是将 Magazine 实体类持久化到数据库中的表的结构，从中可以看出并没有 Item 中的属性。

清单 17.Magazine 实体继承了 Item 非实体类

CREATE TABLE `magazine` (

`ID` bigint(20) NOT NULL,

`ISBN` varchar(255) default NULL,

`PUBLISHER` varchar(255) default NULL,

PRIMARY KEY (`ID`)

) ENGINE=InnoDB DEFAULT CHARSET=latin1;

**Mapped SupperClass：**

JPA 有一种特殊的类叫做 Mapped Supper class，这种类不是实体类，他与实体类的区别就是用 @MappedSuperclass 注解来替代 @Entity 注解，其他方面没有变化。如清单 18 中的例子所示，Employee 类被注解成了一个 Mapped Superclass，并且使用了继承映射中的 Joined 映射策略。

清单 18.Employee 是一个 Mapped superclass

@MappedSuperclass

@Inheritance(strategy=InheritanceType.JOINED)

public class Employee {

@Id

@GeneratedValue

private Long id;

private String name;

private String depart;

// Getters and Setters

}

这种类不能被映射到数据库中，持久化后数据库中没有与之对应的表，因此就不能使用 @Table 注解，也就不能对他进行查询等操作。但是任何继承他的子类可以将继承而来的属性持久化到数据库中。

清单 19.Manager 继承了 Employee

@Entity

public class Manager extends Employee {

private String office;

private String car;

// Getters and Setters

}

清单 19 中的 Manager 类继承了 Employee 类，清单 20 就是 Manager 类映射到数据库中后对应的表结构。从表结构中可以看出，Manager 从 Employee 继承而来的属性也被映射到了数据库中。

清单 20.Manager 表的结构

CREATE TABLE `manager` (

`ID` bigint(20) NOT NULL,

`OFFICE` varchar(255) default NULL,

`CAR` varchar(255) default NULL,

`DEPART` varchar(255) default NULL,

`NAME` varchar(255) default NULL,

PRIMARY KEY (`ID`)

) ENGINE=InnoDB DEFAULT CHARSET=latin1;

### 继承映射示例

以JOINED策略为例：

继承关系的父类Animal：

|  |
| --- |
| import javax.persistence.Entity;  import javax.persistence.Id;  import javax.persistence.Inheritance;  import javax.persistence.InheritanceType;    @Entity  @Inheritance(strategy = InheritanceType.JOINED)  public class Animal {  @Id  private int id;    private String name;    public Animal() {  }    public Animal(int id, String name) {  this.id = id;  this.name = name;  }    public int getId() {  return id;  }    public void setId(int id) {  this.id = id;  }    public String getName() {  return name;  }    public void setName(String name) {  this.name = name;  }    } |

实体类Fish：

|  |
| --- |
| import javax.persistence.Entity;  @Entity  public class Fish extends Animal {  /\* 鱼的活动范围,比如江、河、湖、海 \*/  private String territory;  public Fish() {  }  public Fish(int id, String name, String territory) {  super(id, name);  this.territory = territory;  }  public String getTerritory() {  return territory;  }  public void setTerritory(String territory) {  this.territory = territory;  }  } |

实体Dog：

|  |
| --- |
| import javax.persistence.Entity;  @Entity  public class Dog extends Animal {  /\* 性别 \*/  private String sex;  public Dog() {  }    public Dog(int id, String name, String sex) {  super(id, name);  this.sex = sex;  }    public String getSex() {  return sex;  }    public void setSex(String sex) {  this.sex = sex;  }    } |

创建合适的数据库表

我们可以使用下面的语句创建数据库表：

|  |
| --- |
| CREATE TABLE ANIMAL(ID INTEGER NOT NULL PRIMARY KEY,NAME VARCHAR(255))  CREATE TABLE DOG(ID INTEGER NOT NULL PRIMARY KEY,SEX VARCHAR(255))  CREATE TABLE FISH(ID INTEGER NOT NULL PRIMARY KEY,TERRITORY VARCHAR(255)) |

持久化实体

使用 OpenJPA 持久化实体的继承关系时，开发者只需要按照面向对象的思想操纵实体即可，无需为实体的继承关系作多余的工作。下面的章节中我们将了解持久化实体 Animal、Fish、Dog 时开发者需要完成的工作以及 OpenJPA 转化后在关系数据库中的实现细节。

持久化Animal

我们可以使用下面的代码段来持久化一个新的 Animal 对象：

|  |
| --- |
| // 通过 Persistence 创建 EntityManagerFactory  EntityManagerFactory factory = Persistence.createEntityManagerFactory(  "jpa-unit", System.getProperties());  // 从 EntityManagerFactory 中创建 EntityManager  EntityManager em = factory.createEntityManager();  // 开始持久化实体的事务  em.getTransaction().begin();    // 使用相同的方式持久化实体  em.persist(new Animal(1,"honey"));    // 提交持久化实体的事务  em.getTransaction().commit();    // 关闭EntityManager  em.close(); |

当我们执行这段代码时，OpenJPA 会将它转化为关系数据库对应的 SQL 语句：

|  |
| --- |
| INSERT INTO Animal (id, name) VALUES (1, 'honey') |

持久化Fish

Fish 对象的持久化和 Animal 实体的持久化过程没有任何的不同，只不过 persist 方法的参数变成了 Fish 对象。我们可以使用下面的代码段来持久化一个新的 Fish 对象。

|  |
| --- |
| //通过 Persistence 创建 EntityManagerFactory  EntityManagerFactory factory = Persistence.createEntityManagerFactory(  "jpa-unit", System.getProperties());  // 从 EntityManagerFactory 中创建 EntityManager  EntityManager em = factory.createEntityManager();  // 开始持久化实体的事务  em.getTransaction().begin();  // 使用相同的方式持久化实体  em.persist(new Fish(2,"mermaid","SEA"));  // 提交持久化实体的事务  em.getTransaction().commit();  // 关闭EntityManager  em.close(); |

由于 Fish 对象的属性保存在两个表中，因此当我们执行这段代码时，OpenJPA 会将它转化为对应的两条 SQL 语句：

|  |
| --- |
| INSERT INTO Animal (id, name) VALUES (2, 'mermaid')  INSERT INTO Fish (id, territory) VALUES (2, 'SEA') |

持久化Dog

持久化 Dog 对象和持久化 Fish 对象的过程几乎一样，区别是 persist 方法的参数变成了 Dog 对象。

|  |
| --- |
| em.persist(new Dog(3,"ba guai","MALE")); |

和持久化 Fish 对象时一样，Dog 对象的属性也保存在两个表中，因此当我们执行这段代码时，OpenJPA 会将它转化为对应的两条 SQL 语句：

|  |
| --- |
| INSERT INTO Animal (id, name) VALUES (3, 'ba guai')  INSERT INTO Dog (id, sex) VALUES (3, 'MALE') |

从数据库中查询实体对象

获取所有的Animal对象

我们通过 OpenJPA 中的 Query 接口和 JPQL（Java Persistence Query Language）语言来获取数据库中的记录并且转换为相应的 Java 对象，因此开发者只需要处理 Java 对象模型即可。下面的代码可以从数据库中获取所有的 Animal 对象，请注意其中粗体的部分。

|  |
| --- |
| //通过 Persistence 创建 EntityManagerFactory  EntityManagerFactory factory = Persistence.createEntityManagerFactory(  "jpa-unit", System.getProperties());  // 创建新的 EntityManager  EntityManager em2 = factory.createEntityManager();  // 查询所有 Animal 对象  Query q = em2.createQuery("select m from Animal m");  // 直接处理 Animal 对象，打印 Animal 对象的信息  for (Animal m : (List<Animal>) q.getResultList()) {  System.out.println("Animal Object:");  System.out.println(" id:" + m.getId());  System.out.println(" name:" + m.getName());  }    // 关闭 EntityManager 和 EntityManagerFactory  em2.close();  factory.close(); |

当我们执行这段代码时，OpenJPA 会将它转化为关系数据库对应的 SQL 查询语句：

|  |
| --- |
| SELECT t0.id, t1.id, t2.id, t0.name, t1.sex, t2.territory FROM Animal t0  LEFT OUTER JOIN Dog t1 ON t0.id = t1.id  LEFT OUTER JOIN Fish t2 ON t0.id = t2.id |

在查询结果返回后，OpenJPA 会将查询结果影射到相关的 Animal 对象上，整个过程是透明的，开发者只需要处理对象模型即可。

获取所有的Fish对象

Fish 对象的获取和 Animal 对象的获取在 OpenJPA 中大同小异，唯一的区别是使用 JPQL 不相同，查询 Fish 对象时使用“select fish from Fish fish”。下面的代码可以从数据库中获取所有的 Fish 对象，请注意其中粗体的部分。

|  |
| --- |
| //通过 Persistence 创建 EntityManagerFactory  EntityManagerFactory factory = Persistence.createEntityManagerFactory(  "jpa-unit", System.getProperties());  // 创建新的 EntityManager  EntityManager em2 = factory.createEntityManager();    // 查询所有 Fish 对象  Query q1 = em2.createQuery("select fish from Fish fish");    // 打印 Fish 对象的信息  for (Fish fish : (List<Fish>) q1.getResultList()) {  System.out.println("Fish Object:");  System.out.println(" id:" + fish.getId());  System.out.println(" name:" + fish.getName());  System.out.println(" territory:" + fish.getTerritory());  }    // 关闭 EntityManager 和 EntityManagerFactory  em2.close();  factory.close(); |

当我们执行这段代码时，OpenJPA 会将它转化为关系数据库对应的 SQL 查询语句：

|  |
| --- |
| SELECT t1.id, t0.id, t1.name, t0.territory  FROM Fish t0  INNER JOIN Animal t1 ON t0.id = t1.id |

在查询结果返回后，OpenJPA 会将查询结果影射到相关的Fish对象上，整个过程是透明的，开发者只需要处理对象模型即可。

获取所有的Dog对象

获取 Dog 对象的过程和获取 Fish 对象的过程一致，开发者只需要将 Query 接口使用的 JPQL 语句改为“select dog from Dog dog”。

|  |
| --- |
| Query q1 = em2.createQuery("select dog from Dog dog "); |

当我们执行这段代码时，OpenJPA 会将它转化为关系数据库对应的 SQL 查询语句：

|  |
| --- |
| SELECT t1.id, t0.id, t1.name, t0.sex  FROM Dog t0  INNER JOIN Animal t1 ON t0.id = t1.id |

在查询结果返回后，OpenJPA 会将查询结果影射到相关的Fish对象上，整个过程是透明的，开发者只需要处理对象模型即可。

### 总结

对象继承关系在关系数据库中的表现是对象持久化中难于实现的部分，OpenJPA 为开发者提供了一种透明的实现。在 OpenJPA 中提供了 SINGLE\_TABLE、JOINED 和 TABLE\_PER\_CLASS 三种实现方式处理实体继承关系，开发者需要做的仅仅是为实体类提供 javax.persistence.Inheritance 注释，同时设置它的 strategy 属性，确定使用哪种对象继承关系即可，和关系数据库交互的部分由 OpenJPA 框架完成。

这三种方式的处理对于开发者而言是透明的，无论选择哪一种，仅仅影响数据在关系数据库中的保存方式，对于开发者而言，只需要按照面向对象的方式操作对象既可，OpenJPA 框架在处理持久化操作的时候，会动态地判断当前对象的实际类类型（后期绑定），从而确定持久化到哪个表中。开发者可以根据需要选择这三种方式的一种或者几种来处理对象之间的继承关系。

### 多态查询

继承结构支持多态查询。

## JPA高级映射

### 嵌套映射

在使用嵌套映射的时候首先要有一个被嵌套的类，[清单 5](http://www.ibm.com/developerworks/cn/java/j-lo-jpasimpemap/#listing5) 中 Address 实体类使用 @Embeddable 注解，说明这个就是一个可被嵌套的类，与 @EmbeddedId 复合主键策略中的主键类（primary key class）稍有不同的是，这个被嵌套类不用重写 hashCode() 和 equals() 方法，复合主键将在后面进行介绍。

##### 清单 7. 被嵌套类

@Embeddable

public class Address implements Serializable {

private String street;

private String city;

private String province;

private String country;

// Getters and Setters

}

清单 6 中 Employee 实体类是嵌套类的拥有者，其中使用了 @Embedded 注解将 Address 类嵌套进来了。

##### 清单 8. 嵌套类的使用者

@Entity

public class Employee implements Serializable {

private static final long serialVersionUID = 1L;

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private Long id;

private String name;

private String email;

private String cellPhone;

@Embedded

private Address address;

// Getters and Setters

}

[清单 7](http://www.ibm.com/developerworks/cn/java/j-lo-jpasimpemap/#listing7) 是持久化后生成的数据库表，可以看出被嵌套类的属性，也被持久化到了数据库中，默认的表名就是嵌套类的拥有者的类名。

##### 清单 9. 使用嵌套类生成的表结构

CREATE TABLE `employee` (

`ID` bigint(20) NOT NULL,

`EMAIL` varchar(255) default NULL,

`NAME` varchar(255) default NULL,

`CELLPHONE` varchar(255) default NULL,

`STREET` varchar(255) default NULL,

`PROVINCE` varchar(255) default NULL,

`CITY` varchar(255) default NULL,

`COUNTRY` varchar(255) default NULL,

PRIMARY KEY (`ID`)

) ENGINE=InnoDB DEFAULT CHARSET=latin1;

被嵌套类的注解方式，field 方式或者 property 方式，依赖于嵌套类的拥有者。上面例子中的 Employee 实体类采用的是 field 注解方式，那么在持久化的过程中，被嵌套类 Address 也是按照 field 注解方式就行映射的。

我们也可以通过 @Access 注解改变被嵌套类映射方式，[清单 8](http://www.ibm.com/developerworks/cn/java/j-lo-jpasimpemap/#listing8) 通过使用 @Access 注解将 Address 被嵌套类的注解方式设定成了 property 方式。[清单 9](http://www.ibm.com/developerworks/cn/java/j-lo-jpasimpemap/#listing9) Employee 仍然采用 filed 注解方式。这种情况下，持久化的时候，被嵌套类就会按照自己设定的注解方式映射，而不会再依赖于嵌套类的拥有者的注解方式。但这并不会映射的结果。

##### 清单 10. 基于 property 方式注解的被嵌套类

@Embeddable

@Access(AccessType.PROPERTY)

public class Address implements Serializable {

private String street;

private String city;

private String province;

private String country;

@Column(nullable=false)

public String getCity() {

return city;

}

public void setCity(String city) {

this.city = city;

}

@Column(nullable=false,length=50)

public String getCountry() {

return country;

}

public void setCountry(String country) {

this.country = country;

}

@Column(nullable=false,length=20)

public String getProvince() {

return province;

}

public void setProvince(String province) {

this.province = province;

}

public String getStreet() {

return street;

}

public void setStreet(String street) {

this.street = street;

}

}

##### 清单 11. 基于 field 方式注解

@Entity

@Access(AccessType. FIELD)

public class Employee implements Serializable {

private static final long serialVersionUID = 1L;

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private Long id;

private String name;

private String email;

private String cellPhone;

@Embedded

private Address address;

// Getters and Setters

}

事先设定被嵌套类的注解方式，是一种应该大力提倡的做法，因为当同一个类被不同的注解方式的类嵌套时，可能会出现一些错误。

# 实体的延迟加载

FetchType定义延迟类型。可选值为EAGER，LAZY。

如在实体关联中（如一对一等）fetch= FetchType.LAZY表示只有实际访问关联的时候才会被加载。默认为EAGER。

延迟加载仅在实体被持久化上下文托管时才生效。

# 级联（cascade）操作

级联的意思是指定两个对象之间的操作联动关系，对一个对象执行了操作之后，对其指定的级联对象也需要执行相同的操作。

级联可以在映射关系中通过cascade= CascadeType.ALL属性配置。CascadeType枚举的属性为ALL,PERSIST,MERGE,REMOVE,PEFRESH。

这些值都定义在javax.persistence. CascadeType中：

|  |
| --- |
| **public** **static** **final** CascadeType *ALL*;  **public** **static** **final** CascadeType *PERSIST*;  **public** **static** **final** CascadeType *MERGE*;  **public** **static** **final** CascadeType *REMOVE*;  **public** **static** **final** CascadeType *REFRESH*;  **public** **static** **final** CascadeType *DETACH*; |

级联仅仅是一种减少EntityManager API调用次数的便捷工具。在使用级联配置时，一定确定级联应该如何使用。如果不确定实体之间怎么配置级联，应该关闭级联功能。默认为关闭。

一般对many-to-one,many-to-many不设置级联，在<one-to-one>和<one-to-many>中设置级联。

# 常用实体管理操作

主要涉及的类为EntityManager。

|  |
| --- |
| private EntityManager　; |

## 增加

|  |
| --- |
| em.persist(t); |

## 删除

|  |
| --- |
| T rt = (T) em.find(t.getClass(), t.getId());  em.remove(rt); |

## 更新

|  |
| --- |
| T ut = (T) em.find(t.getClass(), t.getId());  **if** (ut != **null**) {  em.merge(t);  } |

## JPA查询

### 基本查询

|  |
| --- |
| @Override  **public**TfindEntityById(Class<T>clazz, Object id) {  **return**em.find(clazz, id);  } |

|  |
| --- |
| @Override  public List<T>findAll(Class<T>clazz) {  String entityName = clazz.getSimpleName();  String jpql = "select " + entityName.toLowerCase() + " from "  + entityName + " " + entityName.toLowerCase();  Query query = em.createQuery(jpql);  return query.getResultList();  } |

### JPQL

Java Persistence Query Language（Java持久化查询语言），是一种可移植的查询语言，可以被编译成所有主流数据库服务器上的SQL。扩展了EJB2.1规范的EJB QL语言。

JPQL是面向对象的，通过面向对象而非面向数据库的查询语言查询数据，在Java空间对类和对象进行操作，避免程序的SQL语句紧密耦合

* 支持单个或多个值的结果类型。
* 支持聚合函数，带排序和分组子句。
* 支持实体的连接查询（inner join left join right join join fetch）
* 支持所有类似于数据库查询的条件表达式，并支持子查询。
* 支持批量更新和删除。
* 支持结果投影到非持久化类上。
* 支持原生数据库查询
* 支持动态查询（Query API）
* 支持命名查询
* 支持多种返回结果组装

JPQL支持三种语句类型，可以在查询中使用JPQL执行选择、更新、删除操作，即select、update、delete。不支持insert操作。

JPQL的更多信息，请查看：

http://docs.oracle.com/cd/E11035\_01/kodo41/full/html/ejb3\_langref.html

#### 基本的查询

　　“SELECT u FROM User u”检索所有的User。u是User的别名。注意：关键字不区分大小写，例如：SELECT和select是一样的，但是实体的名称和实体的字段是区分大小写的，例如：User和user是不一样的。

|  |
| --- |
| String jpql = "SELECT u FROM User u";//u是User的别名  List<User> users = em.createQuery(jpql).getResultList();  //用User.class指定返回的类  List<User> users = em.createQuery(jpql, User.class).getResultList(); |

#### 查询所有实体

查询所有实体的JPQL查询字串很简单，例如：

select o from Orders o 或 select o from Orders as o

该JPQL语句执行后将返回所有订单实体，它们属于实体类Orders，由from子句声明并指定标识符变量o，相当于SQL的别名。Orders与其后标识符变量之间的关键字as可以省去。

标识符变量的命名规范与Java标识符相同，且区分大小写。from子句可以包含多个声明，以逗号分隔。如：

|  |
| --- |
| Query query = entityManager.createQuery( "select o from Orders o");  List orders = query.getResultList();  Iterator iterator = orders.iterator();  while( iterator.hasNext() ) {  // 处理Orders  } |

#### 查询部分属性

在实际应用中，常常只须查询实体的部分属性而不需要返回整个实体。尤其在实体的属性特别多时，只查询部分属性还可显著提高查询性能。例如：

select o.id, o.customerName, o.address.streetNumber from Orders o order by o.id

执行该查询返回的不再是Orders实体集合，而是一个对象数组的集合(Object[])，集合的每个成员为一个对象数组，可通过数组元素访问各个属性。

例如：

|  |
| --- |
| Query query = entityManager.createQuery(  "select o.id, o.customerName, o.address.streetNumber  from Orders o order by o.id" );  List result = query.getResultList();  Object[] row = result.get( 0 );  int id = Integer.parseInt( row[0].toString() );  String customerName = row[1].toString();  int streetNumber = Integer.parseInt( row[2].toString() );  … … |

#### where子句

where子句用于指定查询条件，where跟条件表达式。例如：

select o from Orders o where o.id = 1

select o from Orders o where o.id > 3 and o.confirm = 'true'

select o from Orders o where o.address.streetNumber >= 123

JPQL也支持包含位置参数查询和名字参数查询

1. 使用位置参数查询

“SELECT u FROM User u WHERE u.name = ?1”检索参数指定name的User。位置参数格式是：?+位置编号，例如：?1和?2。

注意：?和位置编号之间不能有空格，不能写成 ? 1，位置编号是大于等于1的正整数。问号?不能写成中文输入法状态下的？。

|  |
| --- |
| String jpql = "SELECT u FROM User u WHERE u.name = ?1";  Query query = em.createQuery(jpql);  query.setParameter(1, "叶开");//给编号为1的参数设值  List<User> users = query.getResultList(); |

1. 使用名字参数查询

“SELECT u FROM User u WHERE u.name = :name”，检索参数指定name的User。名字参数格式是：:+参数名称，例如：:name和:id。

注意：:和参数名称之间不能有空格，不能写成 : name。冒号:不能写成中文输入法状态下的：。

|  |
| --- |
| String jpql = "SELECT u FROM User u WHERE u.name = :name";  Query query = em.createQuery(jpql);  query.setParameter("name", "叶开");//给参数"name"设值  List<User> users = query.getResultList(); |

where条件表达式中可用的运算符基本上与SQL一致，包括：

* 导航运算符：.
* 算术运算符：+　-　\*　/　+(正)　-(负)
* 关系运算符：==　<>　>　>=　<　<=　[not]between　[not]like　[not]in　is [not] null　 is [not]empty　[not] member [of]
* 逻辑运算符： and　or 　not

#### 子查询

JPQL也支持子查询，在where或having子句中可以包含另一个查询。当子查询返回多于1个结果集时，它常出现在any、all、exists表达式中用于集合匹配查询。它们的用法与SQL语句基本相同。

下面是一些常见查询表达式示例：

// 以下语句查询Id介于100至200之间的订单。

select o from Orders o where o.id between 100 and 200

// 以下语句查询国籍为的 'US'、'CN'或'JP' 的客户。

select c from Customers c where c.county in ('US','CN','JP')

// 以下语句查询手机号以139开头的客户。%表示任意多个字符序列，包括0个。

select c from Customers c where c.phone like '139%'

// 以下语句查询名字包含4个字符，且234位为ose的客户。\_表示任意单个字符。

select c from Customers c where c.lname like '\_ose'

// 以下语句查询电话号码未知的客户。null用于测试单值是否为空。

select c from Customers c where c.phone is null

// 以下语句查询尚未输入订单项的订单。empty用于测试集合是否为空。

select o from Orders o where o.orderItems is empty

#### order by子句

order by子句用于对查询结果集进行排序。和SQL的用法类似，可以用"asc"和"desc"指定升降序。如果不显式注明，JPQL默认为升序。例如：

select o from Orders o order by o.id

select o from Orders o order by o.address.streetNumber desc

select o from Orders o order by o.customer asc, o.id desc

#### group by子句与聚合查询

group by子句用于对查询结果分组统计，通常需要使用聚合函数。常用的聚合函数主要有AVG、SUM、COUNT、MAX、MIN等，它们的含义与SQL相同。例如：

select max(o.id) from Orders o

没有group by子句的查询是基于整个实体类的，使用聚合函数将返回单个结果值，可以使用Query.getSingleResult()得到查询结果。例如：

|  |
| --- |
| Query query = entityManager.createQuery("select max(o.id) from Orders o");  Object result = query.getSingleResult();  Long max = (Long)result; |

对于某些数据库，max()方法返回的结果类型不一定与id对应列的类型相符，更安全的方式是采用String来转型。例如：

long max = Long.parseLong(result.toString());

使用group by子句可以实现分组统计的聚合查询。这时聚合函数的值将被作为查询的一个属性与其它属性列一起返回。这时，group by子句必须包含select语句中除聚合函数外的所有属性列。例如，以下语句基于订单视图分组查询统计各商家销售的各产品数量：

select o.seller, o.goodId, sum(o.amount) from V\_Orders o group by o.seller, o.goodId

注意：group by后面必须包含o.seller和o.goodId

#### having子句

having子句用于对group by分组设置约束条件，用法与where子句基本相同，不同是where子句作用于基表或视图，以便从中选择满足条件的记录；having子句则作用于分组，用于选择满足条件的组，其条件表达式中通常会使用聚合函数。

例如，以下语句用于查询订购总数大于100的商家所售商品及数量：

|  |
| --- |
| select o.seller, o.goodId, sum(o.amount) from V\_Orders o group by o.seller, o.goodId having sum(o.amount) > 100 |

#### 关联查询

JPQL也支持和SQL中类似的关联语法。如：

left out join / left join

inner join

left join / inner join fetch

例如，以下外关联查询可以找出所有客户实体记录，即使它未曾订货：

select c from Customers c left join c.orders o

以下内关联查询只找出所有曾订过商品的客户实体记录：

select c from Customers c inner join c.orders o

如果001号客户下过5次订单的话，以下fetch关联查询将得到5个客户实体的引用，并且执行了5个订单的查询：

select c from Customers c left join fetch c.orders o where c.id=001

#### JPQL函数

JPQL提供了以下一些内置函数，包括字符串处理函数、算术函数和日期函数。

* **字符串处理函数**

concat(String s1, String s2)：字符串合并/连接函数。

substring(String s, int start, int length)：取字串函数。

trim([leading|trailing|both,] [char c,] String s)：从字符串中去掉首/尾指定的字符或空格。

lower(String s)：将字符串转换成小写形式。

upper(String s)：将字符串转换成大写形式。

length(String s)：求字符串的长度。

locate(String s1, String s2[, int start])：从第一个字符串中查找第二个字符串(子串)出现的位置。若未找到则返回0。

* **算术函数**

主要有abs、mod、sqrt、size等。size用于求集合的元素个数。

* **日期函数**

主要为三个，即current\_date、current\_time、current\_timestamp，它们不需要参数，返回服务器上的当前日期、时间和时戳。

#### update语句

update语句用于执行数据更新操作。

例如，以下语句将帐户余额不足万元的客户状态设置为未偿付：

update Customers c set c.status = '未偿付' where c.balance < 10000

值得注意的是，update语句会直接映射到数据库的更新操作，不进行乐观锁检查，并可能导致数据库与当前持久化环境不一致。

建议update语句最好作为单独事务提交，或者放在事务的开始处执行。

#### delete语句

delete语句用于执行数据更新操作

例如，以下语句删除不活跃的、没有订单的客户：

delete from Customers c where c.status = 'inactive' and c.orders is empty

delete语句同样可能导致数据库与当前持久化环境不一致。delete语句通常也是作为单独事务提交，或者放在事务的开始处执行。

# 缓存

性能是企业应用重点关注的内容之一，缓存是提升企业系统性能的重要手段之一。OpenJPA 针对数据持久化提供多种层次、多方面的缓存支持，包括数据、查询、汇编查询的缓存等。这些缓存的应用可以大幅度的提高企业应用的运行效率。

## 一级缓存

## 二级缓存

暂时不涉及。待后续。

# OpenJPA常用配置属性

## 根据实体生成数据库表

|  |
| --- |
| <property name=*"openjpa.jdbc.SynchronizeMappings"* value=*"buildSchema(ForeignKeys=true)"* /> |

## 配置数据库相关信息

|  |
| --- |
| <property name=*"openjpa.ConnectionDriverName"* value=*"com.mysql.jdbc.Driver"* />  <property name=*"openjpa.ConnectionUserName"* value=*"root"* />  <property name=*"openjpa.ConnectionPassword"* value=*"123456"* /> |

# 常见问题

## persistence.xml文件版本问题

|  |
| --- |
| <openjpa-2.2.2-r422266:1468616 fatal user error> org.apache.openjpa.persistence.ArgumentException: You have supplied columns for "net.chenxs.samples.jpa.domain.Classes.students", but this mapping cannot have columns in this context. at org.apache.openjpa.jdbc.meta.MappingInfo.assertNoSchemaComponents(MappingInfo.java:382) at org.apache.openjpa.jdbc.meta.strats.RelationToManyTableFieldStrategy.map(RelationToManyTableFieldStrategy.java:97) at |

解决：需要换成2.0版本。建立直接从OpenJAP安装包的samples示例中拿至persistence.xml模板

## 使用blueprint发布服务时，使用jndi有时候拿不到

启动级别，启动的时候，服务还没有被注册，也没有注册至jndi，控制一下启动级别。