**前言**

    最近在做一个项目的时候因为牵涉到要对数据库的操作，在最开始实现的时候采用了直接的JDBC技术。在实现的时候，发现虽然能够实现所需要的功能，但是具体的实现比较繁琐，而且牵涉到数据库和对象实体间元素的映射，稍微一点点修改就容易带来问题。于是在闲暇时间的时候用JPA重新做了一个实现。和纯JDBC的实现方式不同，JPA本身就是一种ORM（对象关系映射）的实现规范。我们可以通过annotation或者xml的方式来定义属性和对象元素之间的映射关系。这样，我们可以用一种更加面向对象的方式来实现代码，使得开发的效率更高。

**一个简单的JPA示例**

**基本环境准备**

    我们先来看看一个具体的JPA工程示例。要运行这个示例，我们需要如下的类库和软件安装配置好：

    类库： EclipseLink, mysql-connector-j

    数据库： Mysql

    开发环境：eclipse

    因为JPA是一个公开的规范（标准接口），所以有不同的实现。我们需要一个JPA的类库。这里我们引用了如下类库：[EclipseLink](http://www.eclipse.org/eclipselink/)。

    在下载EclipseLink包后（me：gradle项目奖可以到maven上寻找），解压。我们需要将里面jlib目录下的eclipselink.jar以及jlib/jpa目录里的java.persistence.x.jar引入到工程中就可以。后面我们定义代码里的映射关系对应的底层实现就是由这两个包实现（me：maven只需要导入core和jpa包）。

    既然要实现ORM，肯定少不了数据库。这里我们用了比较传统的mysql数据库。所以少不了也需要引用mysql的jdbc驱动：[mysql-connector-javax.jar](http://dev.mysql.com/downloads/connector/j/#downloads)。

    准备好了这些之后，我们创建一个JPAProject的java工程。整体的项目结构如下图：



    这里为了方便工程引用类库，首先将这些库文件拷贝到工程下面的一个lib目录里，再将这些库引入到工程中。

**persistence.xml**

    在前面配置好基本的类库之后我们还有一个需要关心的就是，既然我们是ORM，就有一个要映射到数据库的地方。这里是怎么映射到数据库的呢？具体又映射到哪个数据库呢？这些都是在persistence.xml文件定义的（me：包提供了两个方法：xml和注解的方式）。我们在创建好工程之后的src目录里建立一个META-INF的目录，然后在这个目录里建立一个persistence.xml文件。程序连接数据库和对象映射的配置信息就是通过读取这个目录下的文件来实现的。我们来看一个persistence.xml的示例：

**Xml代码  [收藏代码](javascript:void())**

1. **<?xml** version="1.0" encoding="UTF-8" **?>**
2. **<persistence** xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3. xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
4. http://java.sun.com/xml/ns/persistence/persistence\_2\_0.xsd"
5. version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"**>**
6. **<persistence-unit** name="EmployeeService" transaction-type="RESOURCE\_LOCAL"**>**
7. **<class>**model.PersonInformation**</class>**
8. **<properties>**
9. **<property** name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" **/>**
10. **<property** name="javax.persistence.jdbc.url"
11. value="jdbc:mysql://localhost:3306/simpleDb" **/>**
12. **<property** name="javax.persistence.jdbc.user" value="root" **/>**
13. **<property** name="javax.persistence.jdbc.password" value="root" **/>**
15. <!-- EclipseLink should create the database schema automatically -->
16. **<property** name="eclipselink.ddl-generation" value="create-tables" **/>**
17. **<property** name="eclipselink.ddl-generation.output-mode"
18. value="database" **/>**
19. **</properties>**
20. **</persistence-unit>**
21. **</persistence>**

    这里都是xml的配置项，看起来也很容易懂。在一个<persistence-unit>的单元里定义了这个unit的名字（me：EmployeeService）以及详细的数据库连接驱动，数据库用户名，密码。前面<class>里面定义的是需要映射到数据库的具体实体类。比如model.Employee就对应于我们在package model里定义的Employee类。因为我们要连的数据库是mysql，这里的javax.persistence.jdbc.driver值被设为com.mysql.jdbc.Driver。而我们具体要连接的数据库名字在javax.persistence.jdbc.url对应的值里面定义了，为simpleDb。为了后面实际程序运行的时候能够读写这个库，我们需要事先在数据库里创建simpleDb。

OK，有了这些基本的设置。我们就能连上数据库进行实际的操作了。

注：xml文件需要放在src/main/resources/META-INF/persistence.xml中

**实体对象定义**

    现在，我们需要定义一个可以具体实例化到数据库里的对象。我们定义一个PersonInformation的类如下：

**Java代码  [收藏代码](javascript:void())**

1. **package** model;
3. **import** javax.persistence.Entity;
4. **import** javax.persistence.Id;
6. @Entity
7. **public** **class** PersonInformation {
8. @Id
9. **private** **int** id;
10. **private** String name;
11. **private** **int** age;
13. **public** **int** getId() {
14. **return** id;
15. }
16. **public** **void** setId(**int** id) {
17. **this**.id = id;
18. }
19. **public** String getName() {
20. **return** name;
21. }
22. **public** **void** setName(String name) {
23. **this**.name = name;
24. }
25. **public** **int** getAge() {
26. **return** age;
27. }
28. **public** **void** setAge(**int** age) {
29. **this**.age = age;
30. }
31. }

    这里只是很简单的一个entity bean对象定义。我们针对每个可以操作的对象定义了get, set方法。这里比较有意思的地方是我们用了两个annotation，一个是@Entity，一个是@Id。这里的@Entity表示一个可以序列化映射的对象。如果我们希望这个对象被映射到数据库中的某个表，则必须要加上这个annotation。而@Id则表示对应表的主键。我们建一个表要求有对应的主键。这里指定id为主键。如果我们不指定主键的话则运行的时候会出错。有兴趣的可以尝试一下。

    有了这些定义之后，我们就可以来使用他们了。这里是通过他们访问数据库的代码：

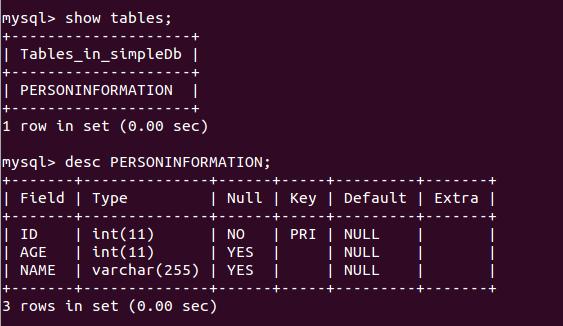
**Java代码  [收藏代码](javascript:void())**

1. **package** main;
3. **import** javax.persistence.EntityManager;
4. **import** javax.persistence.EntityManagerFactory;
5. **import** javax.persistence.Persistence;
7. **import** model.PersonInformation;
9. **public** **class** Main {
10. **private** **static** **final** String PERSISTENCE\_UNIT\_NAME = "EmployeeService";
11. **private** **static** EntityManagerFactory factory;
13. **public** **static** **void** main(String[] args) {
14. factory = Persistence.createEntityManagerFactory(
15. PERSISTENCE\_UNIT\_NAME);
16. EntityManager em = factory.createEntityManager();
17. em.getTransaction().begin();
18. PersonInformation person = **new** PersonInformation();
19. person.setId(1);
20. person.setAge(30);
21. person.setName("fred");
22. em.persist(person);
23. em.getTransaction().commit();
24. em.close();
25. }
26. }

    第一个需要创建的对象是EntityManagerFactory，这里通过Persistence.createEntityManagerFactory方法。而这个"EmployeeService"是哪里来的呢？我们看前面的persistence.xml文件，那里有<persistence-unit name="EmployeeService" transaction-type="RESOURCE\_LOCAL">这个定义。EmployeeService就是对应到这里一个persistence-unit的名字。他们必须一致。实际上，如果我们需要访问多个库的话，在配置文件里也可以定义多个persistence-unit。有了这个factory之后我们再创建一个EntityManager对象。

    为了使得对象的创建成为一个事务来提交，我们通过em.getTransaction().begin(); em.getTransaction().commit();这两个方法来完成整个数据插入的过程。

    现在运行程序，我们再去查看数据库的话，则会发现如下的信息：



    回顾一下我们前面定义的PersonInformation对象的信息，我们定义的字段都是小写格式的。映射到数据库里包括表名和表里面字段名都是大写格式的。前面我们指定了id字段为主键之后，对应的表里这个字段就有了primary key not null的限制。

**修改**

    从前面的运行结果，我们可以看到一些实体对象和数据库表之间的映射关系。可是在实际情况中，有时候我们希望数据库表名或者里面的字段名能够更加可以定制化一点。不是简单的将对象里的字段变成大写的。这样可以增加一点可读性，那我们该怎么修改呢？

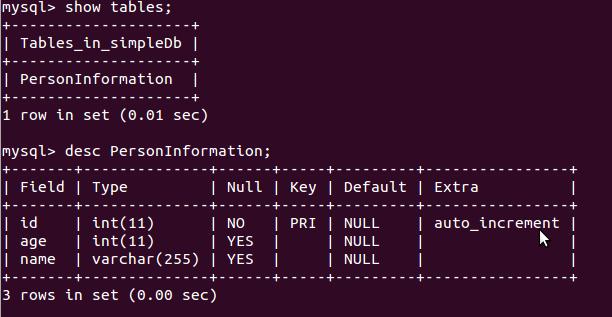
    很显然，我们需要指定对象应该对应哪个表名以及每个属性应该对应什么名字。这是修改后的代码：

**Java代码  [收藏代码](javascript:void())**

1. **package** model;
3. **import** javax.persistence.Column;
4. **import** javax.persistence.Entity;
5. **import** javax.persistence.GeneratedValue;
6. **import** javax.persistence.GenerationType;
7. **import** javax.persistence.Id;
8. **import** javax.persistence.Table;
10. @Entity
11. @Table(name="PersonInformation")
12. **public** **class** PersonInformation {
13. @Id
14. @GeneratedValue(strategy = GenerationType.IDENTITY)
15. @Column(name="id")
16. **private** **int** id;
18. @Column(name="name")
19. **private** String name;
21. @Column(name="age")
22. **private** **int** age;
24. **public** **int** getId() {
25. **return** id;
26. }
27. **public** **void** setId(**int** id) {
28. **this**.id = id;
29. }
30. **public** String getName() {
31. **return** name;
32. }
33. **public** **void** setName(String name) {
34. **this**.name = name;
35. }
36. **public** **int** getAge() {
37. **return** age;
38. }
39. **public** **void** setAge(**int** age) {
40. **this**.age = age;
41. }
42. }

    和前面的代码比起来，我们这里增加了几个描述属性。一个是@Table，这里通过它来设定对应的数据库表名字是什么。@Column这个用来设定对应的数据库字段名。还有一个比较有意思的地方就是我们在id字段增加了@GeneratedValue(strategy = GenerationType.IDENTITY)这个描述属性。它表示什么意思呢？它表示这个主键的值可以自动来生成，而后面的GenerationType.IDENTITY表明它的生成方式是自动增长，类似于auto increment.

    针对前面修改之后我们再来看数据库的运行结果：



    这样，经过前面这些步骤的描述，我们已经能够完成一个简单的JPA示例了。也因此知道了将对象如何映射到数据库表中。

**各种关系映射**

    前面的示例是简单的将一个实体对象映射到数据库中。在很多实际的情况下，我们并不是简单的一个对象映射，往往对象和对象之间还存在着一定的关联关系。我们在做ORM的时候，也需要针对不同的情况进行考虑。这个就牵涉到数据库中间表或者实体对象之间的关系。总的来说，实体对象之间的关系无非为以下几种：一对一，一对多，多对多。而对于多对一的情况来说，从另外一个角度来看也是一种一对多的关系。

**一对一**

    我们都知道，对于不管是哪种对应的关系，实际上他们在对应到数据库的实现里，我们是可以有几种不同的映射实现方式的。我们以一对一的关系开始。假设我们有一个Employee的实体对象。它同时有一个属性是Address。这个Address和它是一对一的映射关系。那么在数据库的实现里，我们该怎么考虑呢？

    我们实际上有几种实现方式。一种很简单，既然他们反正是一对一的关系，我们完全可以把他们当成一个表来使。也就是说这个Address虽然是Employee的属性，但是他们所有的属性放到一个表里是完全没问题的。还有一种就是我们可以定义两个分离的表，他们之间通过外键关联。当然，除了这种，我们也可以定义一个单独的映射表来保存他们之间的映射关系。

    现在我们来看看以上各种方式的实现：

**合并成一个表**

    我们定义的Employee对象如下：

**Java代码  [收藏代码](javascript:void())**

1. **package** model;
3. **import** javax.persistence.Embedded;
4. **import** javax.persistence.Entity;
5. **import** javax.persistence.Id;
6. **import** javax.persistence.Table;
8. @Entity
9. @Table(name="Employee")
10. **public** **class** Employee {
11. @Id
12. **private** **int** id;
13. **private** String name;
14. **private** **long** salary;
16. @Embedded
17. **private** Address address;
19. **public** Address getAddress() {
20. **return** address;
21. }
23. **public** **void** setAddress(Address address) {
24. **this**.address = address;
25. }
27. **public** Employee() {}
29. **public** Employee(**int** id) { **this**.id = id; }
30. // ...
31. }

    这里出于篇幅的限制，省略了那些属性的get, set方法。

Address类的定义实现如下：

**Java代码  [收藏代码](javascript:void())**

1. **package** model;
3. **import** javax.persistence.Access;
4. **import** javax.persistence.AccessType;
5. **import** javax.persistence.Column;
6. **import** javax.persistence.Embeddable;
8. @Embeddable
9. **public** **class** Address {
10. **private** String street;
11. **private** String city;
12. **private** String state;
14. @Column(name="Zip\_Code")
15. **private** String zip;
16. //...
17. }

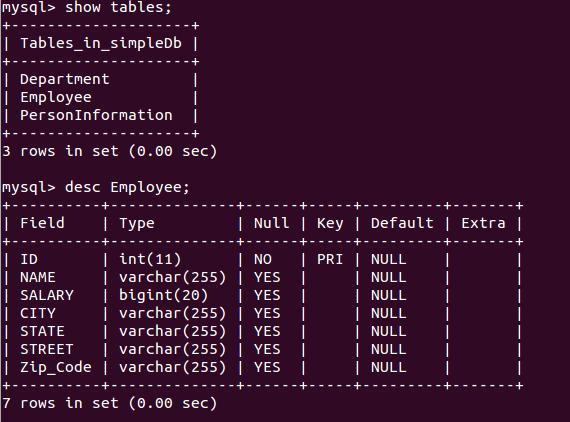
    我们来看代码里加入的标注。这里Employee有一个Address的属性。我们在Employee里对这个属性增加了@Embedded标注。而定义Address的类里增加了@Embeddable标注。其中@Embedded表示Address元素被嵌入到Employee表中间。为了能够嵌入到Employee表中，我们还需要将Address属性增加@Embeddable，表示它是能够被嵌入到其他对象里面的。

    我们运行如下的代码：

**Java代码  [收藏代码](javascript:void())**

1. **package** main;
3. **import** javax.persistence.EntityManager;
4. **import** javax.persistence.EntityManagerFactory;
5. **import** javax.persistence.EntityTransaction;
6. **import** javax.persistence.Persistence;
8. **import** model.Address;
9. **import** model.Employee;
11. **public** **class** EmployeeTest {
13. **public** **static** **void** main(String[] args) {
15. EntityManagerFactory entityManagerFactory =
16. Persistence.createEntityManagerFactory("EmployeeService");
17. EntityManager em = entityManagerFactory.createEntityManager();
18. EntityTransaction userTransaction = em.getTransaction();
20. userTransaction.begin();
22. Employee employee = **new** Employee();
23. employee.setName("frank");
24. employee.setSalary(2000);
26. Address address = **new** Address();
27. address.setCity("Beijing");
28. address.setState("BJ");
29. address.setStreet("Shuangying");
30. address.setZip("100000");
31. employee.setAddress(address);
32. em.persist(employee);
33. userTransaction.commit();
34. em.close();
35. entityManagerFactory.close();
36. }
37. }

    我们现在来看数据库的结果：



**外键约束**

    在采用这种外键约束的时候我们需要首先考虑一下。根据我们定义的逻辑关系，可以认为是Employee里有Address这么一个字段。那么就相当于Employee里要引用到Address的信息。而按照外键的定义，是一个表引用另外一个表的主键。那么，我们就需要给Address定义一个主键，同时我们也要标注一下在Employee里引用的Address字段该是什么名字。我们定义的Employee如下：

**Java代码  [收藏代码](javascript:void())**

1. **package** model;
3. **import** javax.persistence.CascadeType;
4. **import** javax.persistence.Entity;
5. **import** javax.persistence.GeneratedValue;
6. **import** javax.persistence.GenerationType;
7. **import** javax.persistence.Id;
8. **import** javax.persistence.JoinColumn;
9. **import** javax.persistence.OneToOne;
10. **import** javax.persistence.Table;
12. @Entity
13. @Table(name="Employee")
14. **public** **class** Employee {
15. @Id
16. @GeneratedValue(strategy = GenerationType.IDENTITY)
17. **private** **int** id;
18. **private** String name;
19. **private** **long** salary;

22. @OneToOne(cascade=CascadeType.ALL)
23. @JoinColumn(name="address\_id")
24. **private** Address address;
26. **public** Address getAddress() {
27. **return** address;
28. }
30. **public** **void** setAddress(Address address) {
31. **this**.address = address;
32. }
34. **public** Employee() {}
36. **public** Employee(**int** id) { **this**.id = id; }
38. // ...
39. }

    这里，我们增加了一个标注@OneToOne(cascade=CascadeType.ALL)和@JoinColumn(name="address\_id")。@OneToOne表示他们是一对一的关系，同时cascade表示他们的级联关系。如果我们仔细观察一下的话会发现前面应用代码里有一个比较有趣的地方，我们em.persist()只是保存了employee对象。而对应的Address对象只是设定为employee对象的一个属性。我们希望是employee对象被保存到数据库里的时候address对象也自动保存进去。那么我们就需要设定这个cascade的级联访问属性。否则我们就需要显式的利用em.persist()来保存address对象。这也就是为什么我们要用一个cascade的属性。

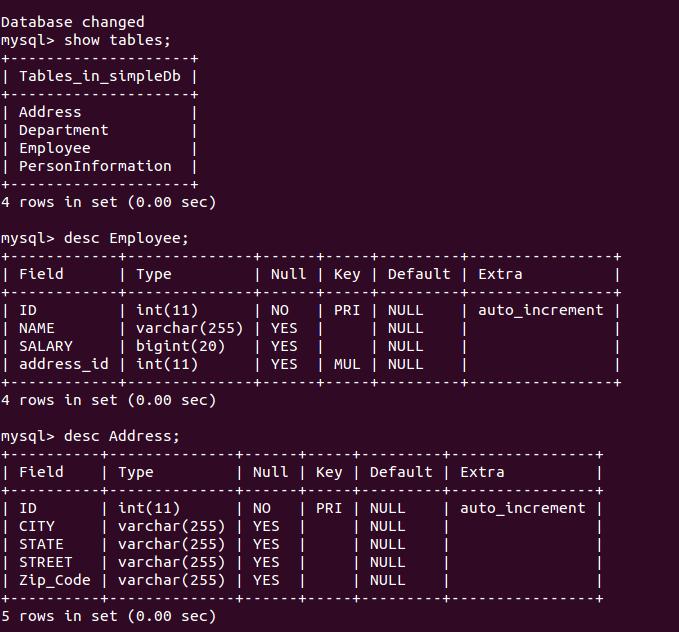
  @JoinColumn里面指定了Employee表里引用到Address时关联的名字是什么。

    和前面的定义比较起来，Address的定义如要增加了一个id：

**Java代码  [收藏代码](javascript:void())**

1. @Id
2. @GeneratedValue(strategy = GenerationType.IDENTITY)
3. **private** **int** id;

    在我们运行程序前需要将Address加入到persistence.xml文件里。因为我们将它映射到一个单独的表里。我们再运行前面的程序，发现生成如下的表结构：



**单独的映射表（me:即中间表）**

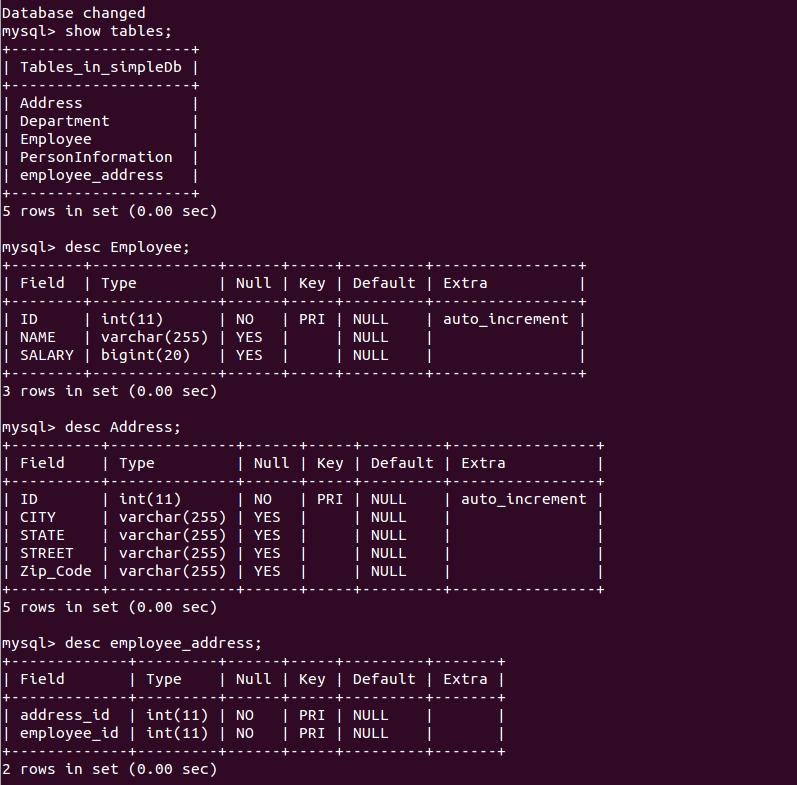
    采用这种映射方式的实现很简单，只需要在Employee里面做一点如下的修改：

**Java代码  [收藏代码](javascript:void())**

1. @OneToOne(cascade=CascadeType.ALL)
2. @JoinTable(name="employee\_address",joinColumns=@JoinColumn(name="address\_id"),inverseJoinColumns=@JoinColumn(name="employee\_id"))
3. **private** Address address;

 其他地方都不需要改变。这里的@JoinTable描述了关联表的名字，他们相互关联的时候里面包含的字段。一个为address\_id，一个为employee\_id。

    运行结果如下图：



**一对多**

    讨论完了前面的一对一映射，我们再来看看一对多的关系。其实和前面一对一的关系很类似，我们可以猜想得到，既然前面描述一对一的关系有@OneToOne，我们这里应该有@OneToMany的映射关系。而且比较有意思的是，在JPA里，除了一对多的关系，也存在着一个多对一的关系描述。那么，在哪些情况下该使用一对多来描述，哪些情况下用多对一来描述呢？我觉得这主要还是取决于我们设计的对象逻辑关系。通过对象的逻辑关系来设置不同的选择。

    我们来看一个示例。

**Many to one**

     假定我们有一个Employee类，然后还有一个Department类。对于每个Employee来说，他属于且仅属于一个Department。这样对于Employee和Department来说，他们就构成一个多对一的关系。而从Department的角度来说，他们则是一个一对多的关系。假定我们要求Employee对象有一个方法来取得他所在的Department。这样从面向对象的角度来说，我们可能希望Employee有一个指向Department的引用。于是按照这种思路，我们设计Employee的代码如下：

**Java代码  [收藏代码](javascript:void())**

1. **package** model;
3. **import** javax.persistence.CascadeType;
4. **import** javax.persistence.Entity;
5. **import** javax.persistence.GeneratedValue;
6. **import** javax.persistence.GenerationType;
7. **import** javax.persistence.Id;
8. **import** javax.persistence.JoinColumn;
9. **import** javax.persistence.ManyToOne;
10. **import** javax.persistence.Table;
12. @Entity
13. @Table(name="Employee")
14. **public** **class** Employee {
15. @Id
16. @GeneratedValue(strategy = GenerationType.IDENTITY)
17. **private** **int** id;
18. **private** String name;
19. **private** **long** salary;
21. @ManyToOne(cascade=CascadeType.ALL)
22. @JoinColumn(name="Dept\_Id")
23. **private** Department department;
25. **public** Employee() {}
27. **public** Employee(**int** id) { **this**.id = id; }
28. // ...
29. }

    这里我们省略了对所有属性get, set方法。这里用了一个@ManyToOne的标注。并设定了映射的名字为“Dept\_Id”。

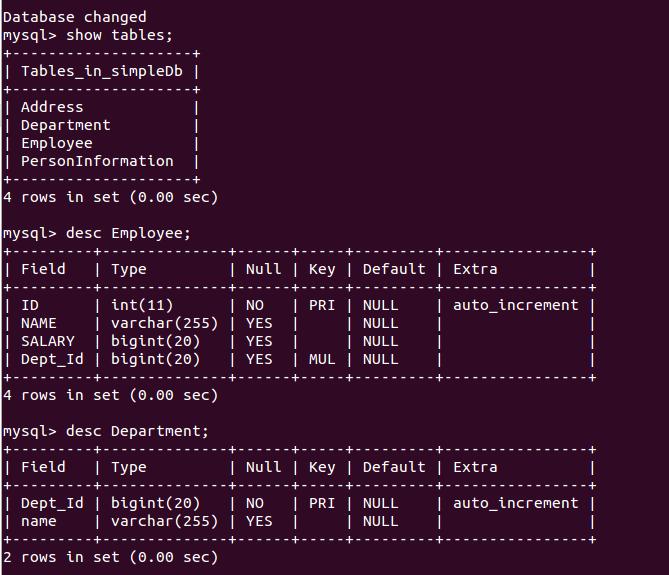
    而Department的定义则如下：

**Java代码  [收藏代码](javascript:void())**

1. **package** model;
3. **import** javax.persistence.Column;
4. **import** javax.persistence.Entity;
5. **import** javax.persistence.GeneratedValue;
6. **import** javax.persistence.GenerationType;
7. **import** javax.persistence.Id;
8. **import** javax.persistence.Table;
10. @Entity
11. @Table(name="Department")
12. **public** **class** Department {
13. @Id
14. @Column(name="Dept\_Id")
15. @GeneratedValue(strategy = GenerationType.IDENTITY)
16. **private** **long** id;
18. @Column(name="name")
19. **private** String name;
20. // ...
21. }

    这个类的定义则没有什么特殊的。和前面定义的差不多。

    运行完之后的数据库情况如下：



**One to many**

    还是前面那个示例，不过这次我们换一个角度来考虑一下。前面是我们希望Employee有一个取得所在Department的属性的方式。这里我们希望从Department对象得到里面所有的员工。从面向对象的角度来说，我们可以在Department里面定义一个集合类来保存一系列的Employee对象。那么，这里两个类的定义形式则如下：

Employee:

**Java代码  [收藏代码](javascript:void())**

1. **package** model;
3. **import** javax.persistence.Entity;
4. **import** javax.persistence.GeneratedValue;
5. **import** javax.persistence.GenerationType;
6. **import** javax.persistence.Id;
7. **import** javax.persistence.Table;
9. @Entity
10. @Table(name="Employee")
11. **public** **class** Employee {
12. @Id
13. @GeneratedValue(strategy = GenerationType.IDENTITY)
14. **private** **int** id;
15. **private** String name;
16. **private** **long** salary;
18. **public** Employee() {}
20. **public** Employee(**int** id) { **this**.id = id; }
21. // ...
22. }

    Employee的定义基本上不需要做什么特殊的关系映射定义。

Department:

**Java代码  [收藏代码](javascript:void())**

1. **package** model;
3. **import** java.util.List;
5. **import** javax.persistence.CascadeType;
6. **import** javax.persistence.Column;
7. **import** javax.persistence.Entity;
8. **import** javax.persistence.FetchType;
9. **import** javax.persistence.GeneratedValue;
10. **import** javax.persistence.GenerationType;
11. **import** javax.persistence.Id;
12. **import** javax.persistence.JoinColumn;
13. **import** javax.persistence.OneToMany;
14. **import** javax.persistence.Table;
16. @Entity
17. @Table(name="Department")
18. **public** **class** Department {
19. @Id
20. @Column(name="Dept\_Id")
21. @GeneratedValue(strategy = GenerationType.IDENTITY)
22. **private** **long** id;
24. @Column(name="name")
25. **private** String name;
27. @OneToMany(cascade=CascadeType.ALL,fetch=FetchType.LAZY)
28. @JoinColumn(name="Dept\_Id")
29. **private** List<Employee> employees;
30. // ...
31. }

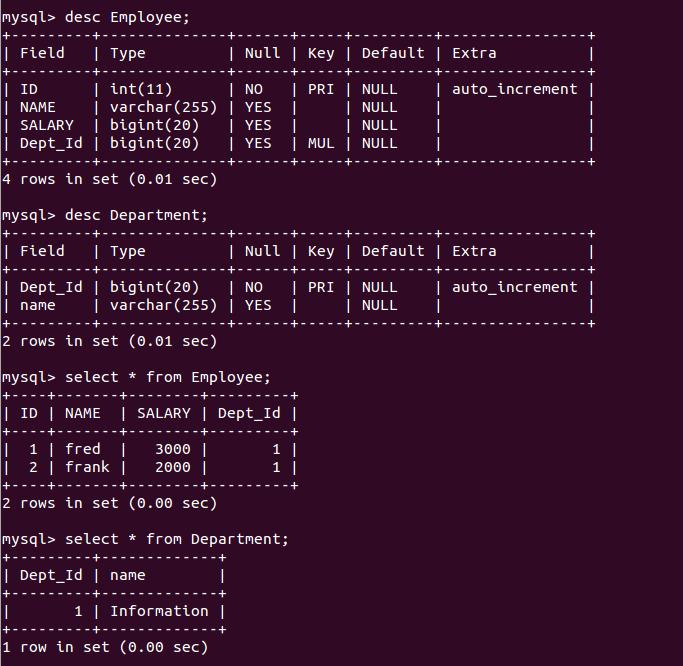
    这里一个比较有意思的地方是在定义了employees的地方加了一个@OneToMany的标注。它同时也通过@JoinColumn说明了关联的项。

    这里我们运行的程序稍微修改了一下，代码如下：

**Java代码  [收藏代码](javascript:void())**

1. **package** main;
3. **import** java.util.ArrayList;
4. **import** java.util.List;
6. **import** javax.persistence.EntityManager;
7. **import** javax.persistence.EntityManagerFactory;
8. **import** javax.persistence.EntityTransaction;
9. **import** javax.persistence.Persistence;
11. **import** model.Department;
12. **import** model.Employee;
14. **public** **class** EmployeeTest {
16. **public** **static** **void** main(String[] args) {
18. EntityManagerFactory entityManagerFactory =
19. Persistence.createEntityManagerFactory("EmployeeService");
20. EntityManager em = entityManagerFactory.createEntityManager();
21. EntityTransaction userTransaction = em.getTransaction();
23. userTransaction.begin();
25. Employee employee = **new** Employee();
26. Department dept = **new** Department();
27. dept.setName("Information");
29. List<Employee> employees = **new** ArrayList<Employee>();
31. employee.setName("frank");
32. employee.setSalary(2000);
33. employees.add(employee);
35. employee = **new** Employee();
36. employee.setName("fred");
37. employee.setSalary(3000);
38. employees.add(employee);
39. dept.setEmployees(employees);
41. em.persist(dept);
42. userTransaction.commit();
43. em.close();
44. entityManagerFactory.close();
45. }
46. }

    我们写入了一个Employee的集合。所以最后运行结果生成的表和结果如下：



    和我们前面一对一映射的关系类似。我们也可以将一对多或者多对一的关系用一个中间关系表来保存。在这里实现就很简单，之需要将Department里面的@JoinColumn去掉就可以了。因为在JPA里，如果我们设定了两个对象之间的一对多属性关系，它会默认生成一个中间表，并且表名以两个表的名字加一个下划线拼接起来。

**多对多**

    对于多对多的场景，看起来它会显得更复杂一些。实际上则未必。因为当我们典型的针对这种关系来设计数据库表的时候，肯定会考虑将他们拆分出一个中间的映射表来。这样的话，基本上要实现这样关系的映射，肯定就只有拆分映射表这么一条路。这里定义了另外一个示例。假定Student和Department是一个多对多的关系。

这里，我们定义了Student:

**Java代码  [收藏代码](javascript:void())**

1. **package** model;
3. **import** java.util.ArrayList;
4. **import** java.util.Collection;
6. **import** javax.persistence.Entity;
7. **import** javax.persistence.GeneratedValue;
8. **import** javax.persistence.GenerationType;
9. **import** javax.persistence.Id;
10. **import** javax.persistence.JoinColumn;
11. **import** javax.persistence.JoinTable;
12. **import** javax.persistence.ManyToMany;
14. @Entity
15. **public** **class** Student {
16. @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
17. **private** **int** id;
18. **private** String name;
20. @ManyToMany
21. @JoinTable(name="Student\_Dept",
22. joinColumns=@JoinColumn(name="Stut\_ID"),
23. inverseJoinColumns=@JoinColumn(name="DEPT\_ID"))
24. **private** Collection<Department> departments;
25. **public** Student() {
26. departments = **new** ArrayList<Department>();
27. }
29. **public** **void** addDepartment(Department department) {
30. **if** (!getDepartments().contains(department)) {
31. getDepartments().add(department);
32. }
33. **if** (!department.getStudents().contains(**this**)) {
34. department.getStudents().add(**this**);
35. }
36. }
38. **public** String toString() {
39. **return** "\n\nID:" + id + "\nName:" + name ;
40. }
41. // ...
42. }

    这里省略了元素的get, set方法。为了实现对象的双向关联，这里定义的addDepartment需要将自己加入到目标Department对象的列表里。另外，我们在对多的关系部分添加了一个@ManyToMany的标注，同时也指定@JoinTable的属性。

    而对于Department：

**Java代码  [收藏代码](javascript:void())**

1. **package** model;
3. **import** java.util.ArrayList;
4. **import** java.util.Collection;
6. **import** javax.persistence.Entity;
7. **import** javax.persistence.GeneratedValue;
8. **import** javax.persistence.GenerationType;
9. **import** javax.persistence.Id;
10. **import** javax.persistence.ManyToMany;
12. @Entity
13. **public** **class** Department {
14. @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
15. **private** **int** id;
16. **private** String name;
18. @ManyToMany(mappedBy="departments")
19. **private** Collection<Student> students;
21. **public** Department(){
22. students = **new** ArrayList<Student>();
23. }
25. **public** **void** addStudent(Student student) {
26. **if** (!getStudents().contains(student)) {
27. getStudents().add(student);
28. }
29. **if** (!student.getDepartments().contains(**this**)) {
30. student.getDepartments().add(**this**);
31. }
32. }
34. **public** String toString() {
35. **return** "Department id: " + getId() +
36. ", name: " + getName();
37. }
38. // ...
39. }

    这里引用Student集合的地方也用了一个@ManyToMany的标注，同时，里面还设置了一个mappedBy属性。这个属性有什么用呢？它是指定我们这个students的集合是映射到Student类里面的departments列表。这样后面生成的对象才找到对应的映射一方。还有一个就是，既然是多对多的映射，在Student里面定义了jointable的属性，这和我们在Department里面定义有什么差别呢？

    这个问题在于我们想定义这个关系的拥有方。我们把jointable的定义放在哪个类这个类就成为了拥有方。那么这个拥有方来绑定和解除他们的关系。另外一方则不能用来绑定和解除他们的关系。

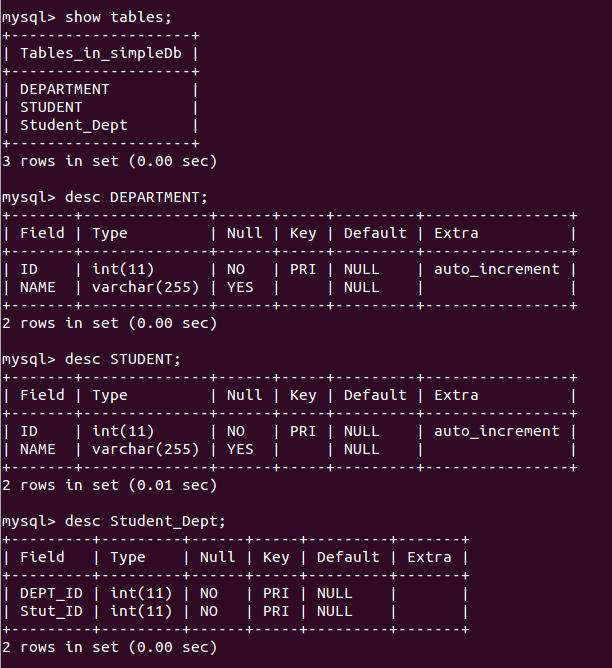
    我们调用的代码实现如下：

**Java代码  [收藏代码](javascript:void())**

1. **package** main;
3. **import** java.util.List;
5. **import** javax.persistence.EntityManager;
6. **import** javax.persistence.EntityManagerFactory;
7. **import** javax.persistence.Persistence;
8. **import** javax.persistence.Query;
10. **import** model.Department;
11. **import** model.Student;
13. **public** **class** Main {
14. **static** EntityManagerFactory emf = Persistence.createEntityManagerFactory("EmployeeService");
15. **static** EntityManager em = emf.createEntityManager();
17. **public** **static** **void** main(String[] a) **throws** Exception {
18. em.getTransaction().begin();

21. Student student = **new** Student();
22. student.setName("Joe");
23. em.persist(student);
25. Student student1 = **new** Student();
26. student1.setName("Joe");
27. em.persist(student1);
29. Department dept = **new** Department();
30. dept.setName("dept name");
31. dept.addStudent(student);
33. dept.addStudent(student1);
34. em.persist(dept);
36. em.flush();
38. Query query = em.createQuery("SELECT e FROM Student e");
39. List<Student> list = (List<Student>) query.getResultList();
40. System.out.println(list);
42. query = em.createQuery("SELECT d FROM Department d");
43. List<Department> dList = (List<Department>) query.getResultList();
44. System.out.println(dList);
46. em.getTransaction().commit();
47. em.close();
48. emf.close();
50. }
51. }

    这里，我们添加了两个Student对象到一个Department对象中。他们运行后的数据表结构如下：



    多对多关系对应的对象关系还有一个比较麻烦的地方在于。一旦建立了双方的对象关系之后，就基本上形成了一个对象之间的循环引用。从面向对象的角度来说这不是一个合适的设计。另外，从内存管理的角度来看，这样也容易导致内存泄漏。因为我们一不小心就容易导致一些引用的对象没有被释放。在将这些关系映射到不同对象的时候，一定要非常的慎重。

**总结**

    以前使用一些语言和开发框架的时候用到ORM。这里针对一个最基本的JPA示例一并探讨了几种典型的对象映射关系在JPA中的实现。有了这些对象关系的定义作为基础我们可以定义很多面向对象的方法而不用采取直接拼接sql字符串的方式与数据库交互。由于要针对这些关系一一讨论，本文的篇幅就会显得长一些。

**参考材料**

[pro jpa2](http://www.amazon.com/gp/product/1430249269/ref=s9_simh_gw_p14_d7_i3?pf_rd_m=ATVPDKIKX0DER&pf_rd_s=center-2&pf_rd_r=05AE0E3K11W0V15XCX1Z&pf_rd_t=101&pf_rd_p=1630083502&pf_rd_i=507846)

<http://www.java2s.com/Tutorial/Java/0355__JPA/ManyToManyJoinTableJoinInverseJoinColumn.htm>