JPA

1 JPA概述

1.1 JPA的定义

JPA 全称为 Java Persistence API，即 Java 持久化 API。JPA 吸取了目前 Java 持久化技术的优点，旨在规范、简化 Java 对象的持久化工作。

JPA 是基于 Java 持久化的解决方案，主要是为了解决 ORM 框架的差异，它的出现在某种程度上能够解决目前 ORM 框架之间不能够兼容的问题。

1.2 ORM工作原理

对象关系映射（Object Relational Mapping，ORM）是一种为了解决面向对象与关系数据库存在的互不匹配的现象的技术。

简单地说，ORM 是通过使用描述对象和数据库之间映射的元数据，将面向对象程序中的对象自动持久化到关系数据库中。

在 ORM 中，由于将对象和关系型数据库关联起来，因此操作对象时就自动操作了数据库，从而避免书写大量的 SQL 语句。

在 ORM 中，有复杂的映射类型，比如一对多、多对多映射等。无论是哪种类型的映射，都遵循以下几个基本原则：

1. 类通常映射为表。
2. 类中的属性通常映射为表中的一列。
3. 如果类的属性是集合类，则会涉及到多个表的关联映射。

常用的 ORM 框架工具：TopLink、Hibernate、OpenJPA、iBatis 等。

每种 ORM 框架都有自己的语法，它们之间不能直接通用，因此针对一种 ORM 设计的客户软件很难移植到其他系统上，增加了程序设计人员的负担。

1.3 JPA与ORM

JPA 不是一种新的 ORM 框架，它只是用于规范现有的 ORM 技术，使得目前流行的 ORM 框架可以实现互相移植，而不是对它们的替代。

在采用 JPA 开发时，仍然将使用这些框架，只是对于不同框架的运用可以不修改任何代码，真正做到低耦合、可扩展程序设计。

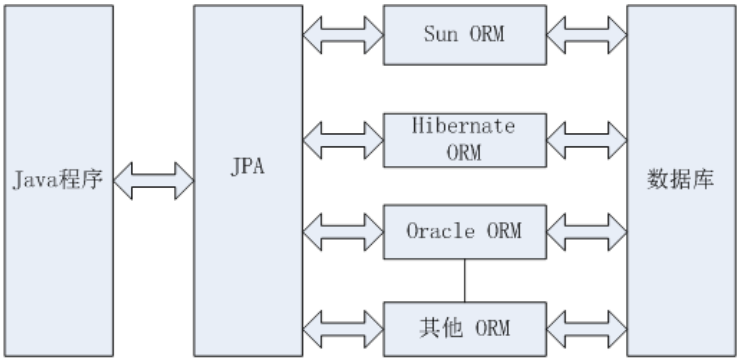

图 1.1　JPA 与 ORM 模块示意图

JPA 包括以下 3 方面的技术：

1. ORM 映射元数据
   JPA 支持 XML 和 JDK 5.0 标注两种元数据的形式，元数据描述对象和表之间的映射关系，框架据此将实体对象持久化到数据库表中。
2. JPA 的 API
   用来操作实体对象，执行 CRUD 操作，框架在后台替我们完成所有的事情，开发者从繁琐的 JDBC 和 SQL 代码中解脱出来。
3. 面向对象查询语言（JPQL）
   这是持久化操作中很重要的一个方面，通过面向对象而非面向数据库的查询语言查询数据，避免程序的 SQL 语句紧密耦合。

JPA 不仅可以使用在 EJB 环境中，也可以在 SE 环境中使用。

Jboss 7.1 中使用 Hibernate 作为 ORM 映射工具。

2 JPA实体映射

简单 Java 对象（Plain Ordinary Java Object，POJO）就是普通的 Java Bean。使用 `@Entity` 标注对一个 POJO 进行标注，就可以使其成为可持久化的实体，也就是**实体类**。只有实体类才能够与数据库中的数据建立映射关系。

将 POJO 映射成为实体类时，需要依赖一些规则。

2.1 `@Entity` 映射实体类

`@Entity` 标注用于实体类声明语句之前，指出该 Java 类为实体类，将此类映射到指定的数据库表。

在 `@Entity` 标注中最常用的属性是 `name`，表示实体的名称。如果没有设置 `name` 属性值，则实体名默认为类名。

使用 `@Entity` 映射实体类时，需要注意的两点：

1. 标注为 `@Entity` 的实体类至少要有一个无参构造方法。因为在使用反射机制的 `class.newInstance()` 方法创建实例时，必须要有一个默认的构造方法，否则抛出 `InstantiationException` 异常。
2. 一个实体类至少要有一个主键。

当实体类被定义在容器中时，服务器将会首先加载所有标注了 `@Entity` 的实体类。非容器中的实体类，会在实体管理器工厂被创建的时候进行加载。

2.2 `@Table` 映射表

当实体类与其映射的数据库表不同名时，需要使用 `@Table` 标注。

`@Table` 标注通常与 `@Entity` 标注并列使用，写在实体类声明语句之前。不能标注在方法或属性前。

`@Table` 标注的属性：

1. `name`：用于指定实体所对应表的名称。默认表名为实体的名称。
2. `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。
3. `uniqueConstraints`：用于设置该实体所关联的唯一约束条件。一个实体可以有多个唯一约束条件，默认没有约束条件。

`@Table` 标注的属性值不区分大小写，因为很多关系型数据库的数据库名和表名不区分大小写。

`@Table` 标注的 `uniqueConstraints` 属性需要配合 `@UniqueConstraint` 标注来使用。这个标注与建表的 SQL 语句中的 `UNIQUE KEY` 对应，表示创建了一个唯一约束。例如，下面的代码指定 student 表的 name 字段和 email 字段为唯一标识。

```
@Entity
@Table(name = "student", uniqueConstrains = { @UniqueConstraint(columnNames = {"name", "email"}) })
public class Student
{
    // ...
}
```

2.3 `@Column` 映射字段

当实体的属性与其映射的数据库表的列不同名时需要使用 `@Column` 标注，通常置于 getter 方法或属性前，也可以与 `@Id` 标注一起使用。

`@Column` 标注的属性：

1. `name`：用于设置映射数据库表的列名。
2. `unique`：表示该字段是否为唯一标识，默认为 `false`。
3. `nullable`：表示该字段是否可以为 null 值，默认为 `true`。
4. `insertable`：表示在使用 insert 语句插入数据时，是否需要插入该字段的值。
5. `updateable`：表示在使用 update 语句修改数据时，是否需要修改该字段的值。

2.4 `@Id` 主键映射

`@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。`@Id` 标注可置于属性声明语句之前或属性的 getter 方法之前。

主键的值可以指定，也可以根据一些特定的规则自动生成。`@GeneratedValue` 标注用于定义主键的自动生成，使用 `strategy` 属性指定主键生成策略。

JPA 默认提供了 4 种主键生成策略：

1. `GeneratorType.AUTO`：容器自动生成，自动选择一个最适合底层数据库的主键生成策略。默认值。
2. `GeneratorType.IDENTITY`：使用数据库的自动增长字段生成，JPA 容器将使用数据库的自增长字段为新增的实体对象赋唯一值。这种情况下需要数据库本身提供自增长字段属性。支持这一策略的数据库有 SQL Server、DB2、MySQL、Derby 等。

3. `GeneratorType.SEQUENCE`：使用数据库的序列号为新增加的实体对象赋唯一值，这种情况下需要数据库提供对序列号的支持。支持这一策略的数据库有 Oracle。
4. `GeneratorType.TABLE`：使用数据库中指定表的某个字段记录实体对象的标识。

配置复合主键的步骤：

1. 编写一个复合主键类。复合主键类要满足以下要求：
   （1）必须实现 `Serializable` 接口。
   （2）必须有默认的 `public` 无参数的构造方法。
   （3）必须重写 `hashCode()` 、 `equals()` 方法。
2. 通过 `@IdClass` 标注在实体类之前标注复合主键，同时在实体类中使用 `@Id` 标注主键的属性。

3 实体管理器

为了实现与后台数据库的交互，构建实体类之后，还需要创建

1. persistence unit：定义了实体和数据存储直接的映射；
2. persistence context：维护了一组受托管的实体对象实例所构成的集合，会追踪实体的状态变化；
3. `EntityManager`：管理实体和数据存储之间的交互，当一个 `EntityManager` 实例创建后，会与 persistence context 关联。

3.1 实体管理器概述

实体管理器（Entity Manager）是 Java 实体对象与数据库交互的中介，它负责管理一组对应的实体，包括这组实体的 CRUD 操作等。同时，实体管理器也负责与持久化上下文（Persistence Context）进行交互，可以实现对实体不同状态进行转换操作。

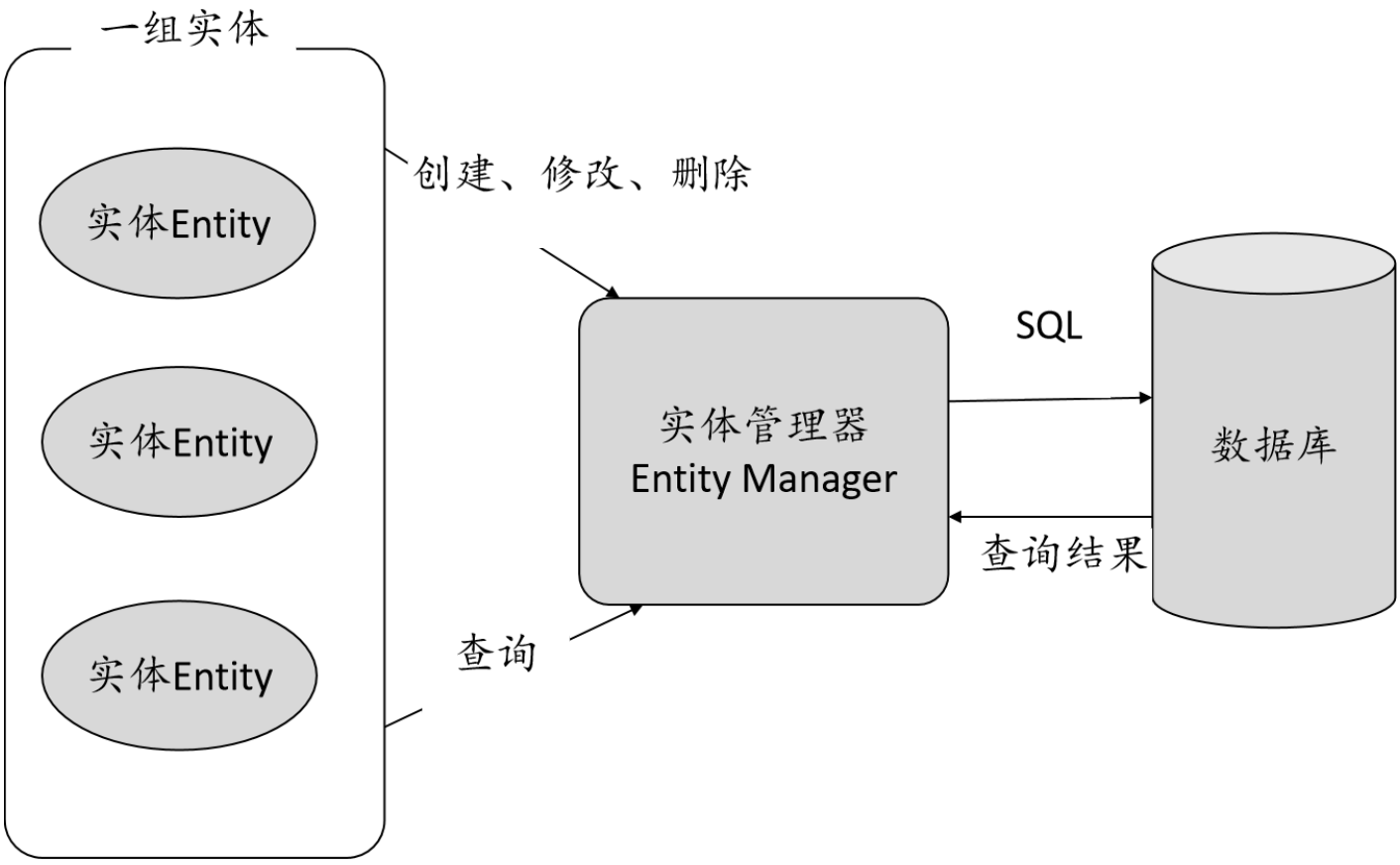

图 3.1　实体管理器

如图 3.1 所示，实体管理器与数据库交互的作用主要体现在以下两个方面：

1. 负责将 Java 中的实体对象操作转化成数据库识别的 SQL 脚本，以便实现实体的持久化。
2. 负责将执行的面向实体查询的 JPQL 转化成 SQL 脚本，并将返回的查询结果组装成实体。

也就是说，凡是涉及对实体的操作，以及对数据库的操作，都需要通过实体管理器来管理。

3.2 实体管理器的分类

根据`EntityManager`对象的管理方式，可以有以下两种类型：

1. 容器托管的`EntityManager`对象
   容器托管的`EntityManager`对象最简单，程序员不需要考虑`EntityManager`连接的释放，以及事务等复杂的问题，所有这些都交给容器去管理。
   容器托管的`EntityManager`对象必须在 EJB 容器中运行，而不能在 Web 容器和 Java SE 环境中运行。
2. 应用托管的`EntityManager`对象
   应用托管的`EntityManager`对象，程序员需要手动地控制它的释放和连接、手动地控制事务等。
   这种获得应用托管的`EntityManager`对象的方式，不仅可以在 EJB 容器中应用，也可以使 JPA 脱离 EJB 容器，而与任何的 Java 环境集成，比如 Web 容器、Java SE 环境等。所以从某种角度上来说，这种方式是使得 JPA 能够独立于 EJB 环境运行的基础。

3.3 实体管理器的获取

3.3.1 容器托管的`EntityManager`对象的获取

在 EJB 容器中获得`EntityManager`对象主要有两种方式：

1. `@PersistenceContext`标注注入
2. JNDI 方式

通过`@PersistenceContext`标注注入获得`EntityManager`对象的方法最为常用，代码如下所示：

```
@Stateless
public class StudentService implements IStudentService
{
    @PersistenceContext(unitName = "jpaUnit")
    private EntityManager entityManager;

    public List<Student> findAllStudents()
    {
        Query query = entityManager.createQuery("SELECT c FROM Student C");
        List<Student> result = query.getResultList();
        for (Student c : result)
        {
            System.out.println(c.getId() + ", " + c.getName());
        }
        return result;
    }
}
```

在`@PersistenceContext`标注中，`unitName`为 persistence.xml 文件中`<persistence-unit>`元素中的属性`name`的值，表示要初始化哪个持久化单元。persistence.xml 文件的内容如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
             http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
    <persistence-unit name="jpaUnit" transaction-type="JTA">
        <jta-data-source>java:jboss/datasources/MySqlDS</jta-data-source>
        <class>javaee.jpa.simpleJPA.Student</class>
    </persistence-unit>
</persistence>
```

通过 JNDI 的方式获得`EntityManager`对象的代码如下：

```java
@Stateless
@PersistenceContext(name = "persistence/jpaUnit", unitName = "jpaUnit")
public class StudentService implements IStudentService
{
    @Resource
    SessionContext ctx;

    public List<Student> findAllStudents()
    {
        EntityManager entityManager = (EntityManager) ctx.lookup("persistence/jpaUnit");
        Query query = entityManager.createQuery("SELECT c FROM Student c");
        List<Student> result = query.getResultList();
        for (Student c : result)
        {
            System.out.println(c.getId() + ", " + c.getName();
        }
        return result;
    }
}
```

通过 JNDI 查找的 jndi-ref 可以通过标注或通过 web.xml 配置，两种方法都可以。上面的例子使用的是标注方法。使用 web.xml 配置的例子如下所示：

```xml
<persistence-context-ref>
    <persistence-context-ref-name>persistence/jpaUnit</persistence-context-ref-name>
    <persistence-unit-name>jpaUnit</persistence-unit-name>
</persistence-context-ref>
```

3.3.2 应用托管的 `EntityManager` 对象的获取

应用托管的 `EntityManager` 对象，都是通过实体管理器工厂（`EntityManagerFactory`）对象创建的。

在 EJB 中，`EntityManagerFactory` 对象可以通过 `@PersistenceUnit` 标注获得，代码如下：

```java
@Stateless
public class StudentService implements IStudentService
{
    @PersistenceUnit(unitName = "jpaUnit")
    private EntityManagerFactory emf;

    public List<Student> findAllStudents()
    {
        // 创建 EntityManager 对象
        EntityManager em = emf.createEntityManager();

        // 使用 EntityManager 对象执行持久化操作

        // 关闭 EntityManager
        em.close();
    }
}
```

在 Web 容器中，`EntityManagerFactory` 对象也可以通过 `@PersistenceUnit` 标注获得，代码如下：

```java
public class TestServlet extends HttpServlet
{
    @PersistenceUnit(unitName = "jpaUnit")
    private EntityManagerFactory emf;

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        // ...
        if (emf != null)
        {
            // 创建 EntityManager 对象
            EntityManager em= emf.createEntityManager();
            try
            {
                // ...
            }
            finally
            {
                // 关闭 EntityManager
                em.close();
            }
            // ...
        }
    }
}
```

在 Java SE 环境中，获得应用托管的`EntityManager`对象只能通过手动的方式创建，而不能使用标注的方式。代码如下：

```java
public class StudentClient
{
    public static void main(String[] args)
    {
        // 创建 EntityManagerFactory 对象
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpaUnit");
        // 创建 EntityManager 对象
        EntityManager em = emf.createEntityManager();
        // ...
        // 关闭 EntityManager 对象
        em.close();
        // 关闭 EntityManagerFactory 对象
        emf.close();
    }
}
```

3.4 实体管理器API

1. `public void persist(Object entity)`：创建实体，将实体保存到数据库中，也可以将一个 Java 的实体对象持久化到数据库中。
2. `public <T> T merge(T entity)`：更新持久化上下文中的实体，并将更新保存到数据库中，返回更新后的实体。
3. `public void remove(Object entity)`：删除实体，将实体从数据库中删除。
4. `public void flush()`：将持久化上下文中的实体保存到数据库中，实现与数据库同步。
5. `public void refresh(Object entity)`：将数据库中的数据重新读取到持久化上下文的实体中，实现与数据库同步。
6. `public void clear()`：将持久化上下文中的实体全部转变成游离状态，此时还没有通过`flush()`方法与数据库同步的实体将不会被持久化到数据库中。

查询方法：

1. `public <T> T find(Class<T> entityClass, Object primaryKey)`
   作用：通过实体主键查找实体对象。
   参数：`entityClass`为实体的类型，`primaryKey`为实体主键对象。
   返回值：如果查询到则返回所查询的实体对象，否则返回`null`。
2. `public <T> T getReference(Class<T> entityClass, Object primaryKey)`
   与`find()`方法类似，不同的是：如果缓存中不存在指定的实体，实体管理器会创建一个实体类的代理，但是不会立即加载数据库中的信息，只有第一次使用此实体类的属性时才会加载。因此，如果此实体类在数据库中不存在，此方法不会返回`null`值，而是抛出`EntityNotFoundException`异常。

3. `public Query createQuery(String ejbqlString)`
   作用：执行 JPQL 查询
   参数：EQL 查询语句
   返回值：新创建的`Query`对象
4. `public Query createNativeQuery(String sqlString)`
   作用：执行本地查询
   参数：本地数据库的 SQL 脚本
   返回值：新创建的`Query`对象

3.5 实体的生命周期

3.5.1 实体的状态

一个实体从创建到销毁要经历多个状态，通常有这样几个状态：瞬时状态（New/Transient）、托管状态（Attatched/Managed）、游离状态（Detached）、销毁状态（Removed）。

这些状态的改变也是通过实体管理器来操作的。

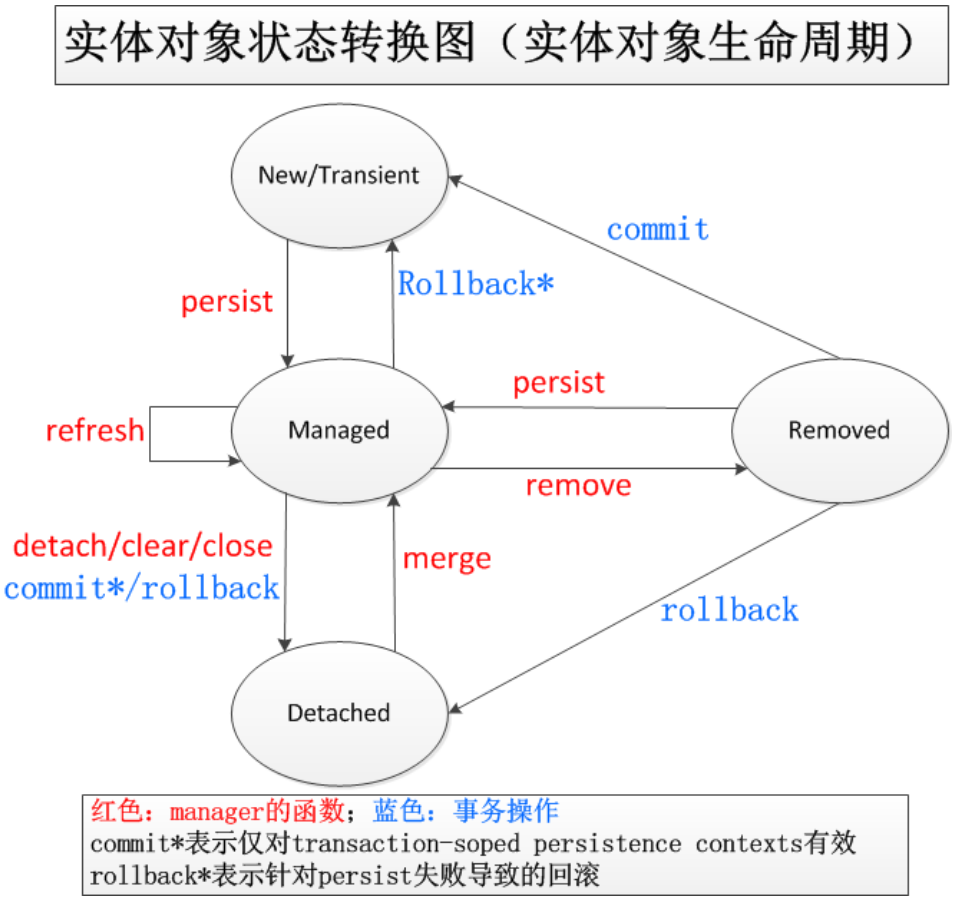

图 3.2　实体对象状态转换图

瞬时状态（New/Transient）：对象尚未保存到数据库中对应记录时的状态。通过`new`创建一个对象后，该对象保存在内存当中，一旦停止应用服务器，该对象就消失了，所以瞬时状态的对象是未持久化的对象。

托管状态（Attatched）：也可以叫做被管理状态（Managed），表示实体处于持久化上下文中，并被其所管理。处于托管状态的实体对象会在事务提交或者`flush()`方法调用后，被同步到数据库中。

游离状态（Detached）：相对于托管状态而言，当实体不在持久化上下文中时，实体将处在游离状态。处在游离状态的实体最大的特点是，它的属性与数据库中持久化的实体是不同步的。

销毁状态（Removed）：实体从持久化上下文中删除后的状态。在事务提交时，实体对应数据将从数据库中删除。

3.5.2 实体操作及实体状态变化

图 3.3　实体状态转换事件及监听

- `public void persist(Object entity)`
  `persist()` 方法可以将实例转换为托管状态。在调用 `flush()` 方法或提交事务后，实例将会被插入到数据库中。
  对不同状态下的实体对象 `x`， `persist()` 方法会产生以下操作：
  1. 如果 `x` 是一个 New 状态的实体，它将会转为 Managed 状态；实体对象 `x` 会在事务提交后或者调用 `flush()` 方法后被插入到数据库中。
  2. 如果 `x` 是一个 Managed 状态的实体， `persist()` 操作会被忽略。
  3. 如果 `x` 是一个 Removed 状态的实体，它将会转换为 Managed 状态。
  4. 如果 `x` 是一个 Detached 状态的实体，该方法会抛出 `EntityExistsException` 异常，具体异常与不同的 JPA 实现有关。
  5. 对所有与 `x` 相关联的实体对象 `y`，如果关联关系中的 `cascade` 属性被设置为 `PERSIST` 或者 `ALL`，则 `persist()` 操作会作用于 `y`。
- `public void remove(Object entity)`
  从持久化上下文中移除实体对象，实体状态变为 Removed 状态。在事务提交时，实体对应数据从数据库中删除。
  对不同状态下的实例 `A`， `remove()` 方法会产生以下操作：
  1. 如果 `A` 是一个 New 状态的实例， `remove()` 操作会被忽略。
  2. 如果 `A` 是一个 Managed 状态的实例，它的状态会转换为 Removed。实体对象 `A` 会在事务提交后从数据库中删除。
  3. 如果 `A` 是一个 Detached 状态的实体，该方法将会抛出 `IllegalArgumentException` 异常，或者事务提交会失败。
  4. 如果 `A` 是一个 Removed 状态的实例， `remove()` 操作会被忽略。
  5. 如果存在与 `A` 实体对象相关联的实体 `y`，并且 `cascade` 属性被设置为 `REMOVE` 或者 `ALL` 时， `remove()` 操作会级联到关联对象 `y` 上。
- `public void refresh(Object entity)`
  让当前处于托管状态的实体对象与数据库中的内容一致。
  对不同状态下的实例 `A`， `refresh()` 方法会产生以下操作：
  1. 如果 `A` 是一个 Managed 状态的实例，它的属性将会和数据库中的数据同步；如果存在与该实体对象相关联的实体 `y`，并且 `cascade` 属性被设置为 `REFRESH` 或者 `ALL` 时， `refresh()` 操作会级联到关联对象 `y` 上。
  2. 如果 `A` 是一个 New、Removed、Detached 状态的实例，抛出 `IllegalArgumentException` 异常，具体情况与不同 JPA 实现有关。
- `public <T> T merge(T entity)`
  将一个新建状态或游离状态实体对象的属性值更新到一个对应的托管状态的实体对象中。
  对不同状态下的实例 `A`， `merge()` 方法会产生以下操作：
  1. 如果 `A` 是一个 Detached 状态的实体对象，找到或新建一个托管状态的实体对象 `A1`，并将 `A` 的属性值赋值给 `A1` 的属性。 `A1` 的属性值会在事务提交或者 `flush()` 方法被调用时，同步到数据库中。
  2. 如果 `A` 是一个 New 状态的实体，新建一个托管状态的实体对象 `A1`，并将 `A` 的属性值赋值给 `A1` 的属性。 `A1` 的属性值会在事务提交或者 `flush()` 方法被调用时，同步到数据库中。
  3. 如果 `A` 是一个 Managed 状态的实体，该操作会被忽略。
  4. 如果 `A` 是一个 Removed 状态的实体，该方法会抛出 `IllegalArgumentException` 异常。
  5. 如果存在与该实体对象相关联的实体 `y`，并且 `cascade` 属性被设置为 `MERGE` 或者 `ALL` 时， `refresh()` 操作会级联到关联对象 `y` 上。

- `public void detach(Object entity)`
  将一个托管状态的实体对象从持久化上下文中移除，使其变成一个游离状态实体对象，终止与数据库的同步。
  对不同状态下的实例`A`，`detach()`方法会产生以下操作：
  1. 如果`A`是一个 Managed 状态的实体，实体对象`A`会从持久化上下文中移除，实体状态变为游离状态。如果存在与该实体对象相关联的实体`y`，并且`cascade`属性被设置为`DETACH`或者`ALL`时，`detach()`操作会级联到关联对象`y`上。
  2. 如果`A`是一个 New 状态的实体，该操作会被忽略。

4 简单的JPA例子

4.1 Java SE环境下JPA程序的开发方法

1. 创建 JPA 工程
   （1）在 Eclipse 中选择 File -> New -> JPA Project。
   （2）在弹出的对话框中设置工程名，并将运行时环境设置为 JDK，JPA 版本保持默认，点击 Next。
   （3）在路径选择界面不用做任何设置，直接下一步。
   （4）在 JPA Facet 设置界面中，Platform 处选择“EclipseLink 2.5.x”，Type 处选择“User Library”，并勾选相应的实现。首次配置时需要下载类库，下载完成后会返回到 JPA Facet 配置页。在Connection 处新建一个数据库连接并选中，然后点击 Finish。
2. 创建实体类
   （1）在 Eclipse 中选择 File -> New -> JPA Entities from Tables。
   （2）在弹出的对话框中指定数据库连接和数据库名，并选择要创建实体类的表，点击 Next。
   （3）在 Customize Defaults 界面中，选择 Key generator，并设置 Package。
   （4）其他设置全部保持默认，点击 Finish。
3. 配置 persistence.xml 文件

```
<!-- 采用本地事务 -->
<persistence-unit name="simpleJPA" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>simpleJPA.Student</class>
    <properties>
        <property name="eclipselink.logging.level" value="FINE" />
        <!-- JDBC驱动 -->
        <property name="eclipselink.jdbc.driver" value="com.mysql.jdbc.Driver" />
        <!-- 数据库URL -->
        <property name="eclipselink.jdbc.url" value="jdbc:mysql://localhost:3306/jpa" />
        <!-- 数据库用户名 -->
        <property name="eclipselink.jdbc.user" value="root" />
        <!-- 密码 -->
        <property name="eclipselink.jdbc.password" value="123456" />
        <property name="eclipselink.allow-zero-id" value="true" />
    </properties>
</persistence-unit>
```

4. 客户端调用 JPA

```java
public class Client
{
    public static void main(String[] args) throws Exception
    {
        // 获得实体管理器工厂
        // 参数 "simpleJPA" 与配置文件中的 <persistence-unit name="simpleJPA"> 保持一致
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("simpleJPA", null);
        // 获得实体管理器
        EntityManager manager = factory.createEntityManager();
        try
        {
            testsave1(manager);
            showAll(manager);
        }
        finally
        {
            manager.close();
            factory.close();
        }
    }

    public static void testsave1(EntityManager manager)
    {
        Student p = new Student();
        p.setName("jase");
        p.setGender("female");
        p.setmajor("计算机");

        EntityTransaction transaction = manager.getTransaction();
        transaction.begin();
        manager.persist(p);
        transaction.commit();
    }

    public static void showAll(EntityManager manager)
    {
        EntityTransaction transaction = manager.getTransaction();

        transaction.begin();
        Query q = manager.createQuery("select c from Student c");
        List results = q.getResultList();
        transaction.commit();

        Iterator it = results.iterator();
        while (it.hasNext())
        {
            Student p = (Student) it.next();
            System.out.print(p.getId() + "\t");
            System.out.print(p.getName() + "\t");
            System.out.println(p.getGender() + "\t");
        }
    }
}
```

4.2 Java EE环境下JPA程序的开发方法

1. 创建 Java EE 项目
2. 设置工程
   （1）右键点击项目，选择 Properties
   （2）在配置对话框中选择 Project Facets，在 Project Facets 页面中将 JPA 选中。
3. 创建实体类
4. 配置 persistence.xml 文件

```xml
<!-- 采用 JTA 事务类型 -->
<persistence-unit name="simpleJPA" transaction-type="JTA">
    <!-- 使用 JDNI 数据源 -->
    <jta-data-source>java:jboss/datasources/MySqlDS</jta-data-source>
    <class>javaee.jpa.simpleJPA.Student</class>
</persistence-unit>
```

5. 客户端调用 JPA

5 实体关系映射

5.1 实体关系

实体关系是指实体与实体之间的关系。从方向上分为单向关联和双向关联，从实体数量上分为一对一、一对多和多对多。

单向关联是一个实体中引用了另外一个实体。简单地说，就是通过一个实体可以获得另一实体的对象。

双向关联是指两个实体之间可以相互获得对方对象的引用。

一对一关系表示实体类 A 中只能找到实体类 B 的一个对象，反之亦然。

一对多关系表示实体类 A 中可以找到实体类 B 的多个对象，而实体类 B 中的多个对象对应实体类 A 的一个对象。

多对多关系表示实体类 A 中可以找到实体类 B 的多个对象，反之亦然。

JPA 中 7 种实体关系：

1. 一对一单向
2. 一对一双向
3. 一对多单向
4. 一对多双向
5. 多对一单向
6. 多对多单向
7. 多对多双向

在关系型数据库中，有两种方式描述关联：外键、表关联。

1. 外键：指向另一个表的主键的字段，称为外键字段。
2. 表关联：两个表的关系单独定义一个表中，通过一个中间表来关联。

5.2 一对一单向关联

在数据库中，通常使用外键关联表示一对一单向关联。

在 JPA 中，一对一单向关联需要使用 2 个标注：`@OneToOne`、`@JoinColumn`。

`@OneToOne`标注表示一对一关系。`@OneToOne`标注具有如下属性：

属性	含义
`targetEntity`	表示关联的实体类型，默认为当前标注的实体类
`cascade`	表示与此实体一对一关联的实体的级联操作类型，默认情况下不关联任何操作。 取值包括：`PERSIST`（级联新建）、`REMOVE`（级联删除）、`REFRESH`（级联刷新）、`MERGE`（级联更新）、`ALL`（全部四项）
`fetch`	该实体的加载方式。取值包括：`EAGER`（关系类在主类加载的时候同时加载， 默认值）、`LAZY`（关系类在被访问时才加载）
`optional`	关联的该实体是否能够存在`null`值
`mappedBy`	用于双向关联实体时，标注在不保存关系的实体中

其中，`cascade`属性所表示的 JPA 级联的各种类型如下所示：

取值	说明
`PERSIST`	当对象被`EntityManager`通过`persist()`方法持久化时，和该对象存在关联关系的其他实体对象也会被执行`persist()`操作
`MERGE`	当对象被`EntityManager`通过`merge()`方法持久化时，和该对象存在关联关系的其他实体对象也会被执行`merge()`操作
`REMOVE`	当对象被`EntityManager`通过`remove()`方法从数据库中删除时， 和该对象存在关联关系的其他实体对象也会被执行`remove()`操作
`REFRESH`	当对象被`EntityManager`通过`refresh()`方法将数据库中的最新状态刷新到实体对象中时， 和该对象存在关联关系的其他实体对象也会被执行`refresh()`操作
`ALL`	等价于上面所有选项的逻辑或

`@OneToOne`标注只能确定实体之间是一对一关系，不能指定数据库表中用来表示关联关系的字段。所以此时要结合`@JoinColumn`标注来进行说明。

`@JoinColumn`标注用来指定实体关系的保存情况，其属性如下：

1. `name`：指定外键字段的名字。在默认情况下，`name = 关联表的名称 + "_" + 关联表主键的字段名`。
2. `referencedColumnName`：指定了这个外键的引用字段。在默认情况下，为外键所指向的表的主键字段名称。

以学生实体（Student）与地址实体（Address）为例来说明一对一单向关系。一个学生对应一个地址，通过学生可以获得该学生的地址信息，因此学生和地址是一对一的关系；但通过地址不能获得学生信息，因此学生和地址是单向关联。

```java
@Entity
public class Student implements Serializable
{
    private static final long serialVersionUID = 1L;

    // 主键
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String name;
    private String gender;
    private String major;

    @OneToOne(cascade = {CascadeType.ALL}) // 指定一对一关系
    @JoinColumn(name = "address_id") // 指定关联字段
    private Address address;

    // .......
}
```

```java
@Entity
public class Address implements Serializable
{
    private static final long serialVersionUID = 1L;

    // 主键
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String detail;

    public Address() {}

    // ........
}
```

5.3 一对一双向关联

在数据库中，仍然可以使用外键关联表示一对一双向关联。

在 JPA 中，一对一双向关联也使用 `@OneToOne` 和 `@JoinColumn` 标注。

```java
@Entity
public class Student implements Serializable
{
    private static final long serialVersionUID = 1L;

    // 主键
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String name;
    private String gender;
    private String major;

    @OneToOne(cascade = {CascadeType.ALL})  // 指定一对一关系
    @JoinColumn(name = "address_id")  // 指定关联字段
    private Address address;

    //.......
}
```

```java
@Entity
public class Address implements Serializable
{
    private static final long serialVersionUID = 1L;

    // 主键
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String detail;

    // 指定一对一关系，并使用 mappedBy 属性指定所关联的实体
    @OneToOne(cascade = {CascadeType.ALL}, mappedBy = "address")
    private Student student;

    public Address() {}

    //........
}
```

5.4 一对多单向关联

一对多单向关联用到的标注：`@OneToMany`、`@JoinColumn`、`@JoinTable`。

`@OneToMany` 标注指明实体类之间是一对多的关系，但它不能指定数据库表中保存的关联字段，所以还要结合 `@JoinColumn` 或 `@JoinTable` 标注来指定保存实体关系的配置。

`@OneToMany` 标注的属性与 `@OneToOne` 标注相似，如下所示：

属性	含义
`targetEntity`	表示关联的实体类型。因为一对多的实体映射保存在集合类中，所以必须指明集合类中保存的具体类型。
`cascade`	表示与此实体一对一关联的实体的级联操作类型，默认情况下不关联任何操作。 取值包括：`PERSIST`、`REMOVE`、`REFRESH`、`MERGE`、`ALL`
`fetch`	该实体的加载方式。取值包括：`EAGER`（关系类在主类加载的时候同时加载）、`LAZY`（关系类在被访问时才加载，默认值）
`mappedBy`	用于双向关联实体时，标注在不保存关系的实体中

指定关联的实体类型时，有两种方案：

1. 指定集合泛型的具体类型，此时不需要设置`targetEntity`属性。例如：

```
@OneToMany(cascade = {CascadeType.ALL})
@JoinColumn(name = "student_id")
private Collection<Student> students = new ArrayList<Student>();
```

2. 指定`targetEntity`属性类型。例如：

```
@OneToMany(targetEntity = Student.class, cascade = {CascadeType.ALL})
@JoinColumn(name = "student_id")
private Collection students = new ArrayList();
```

对于一对多的实体关系，表结构有两种设计策略：外键关联、表关联。

- 外键关联：不管是一对多还是多对一，都是在“多”方添加指向“一”方的外键。
- 表关联：两个表的关系单独定义在一个表中，通过一个中间表来关联。

以老师和学生为例，假设一个老师可以有多个学生，而一个学生只能有一个老师，则老师和学生之间具有一对多关系。采用外键关联建立一对多单向关联的示例如下：

```
@Entity
public class Teacher implements Serializable
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private int age;
    private String gender;
    private String teacherName;

    @OneToMany(cascade = {CascadeType.ALL})
    @JoinColumn(name = "teacher_id")
    private Collection<Student> students = new ArrayList<Student>();

    public Teacher() { }

    // ...
}
```

```
@Entity
public class Student implements Serializable
{
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String gender;
    private String name;
    private String major;

    public Student() { }

    // ...
}
```

注意：

1. 在创建`Student`实体时，student 表中的外键字段（teacher_id）被排除在外。因为关系型数据库间的外键字段用于实现两张表间的关联，对应于 JPA 实体对象间的关联关系。
2. 在数据库中，无论是一对多单向还是一对多双向，映射关系的维护端都是在多的那一方，也就是`Student`类。
3. 在这里`@JoinColumn`标注中`name`属性的值是 student 表中的外键的列名，它并不像在一对一里面的`name`属性值是自己表中的外键的列名。

通过表关联的方式来映射一对多关系时，要使用`@JoinTable`标注，它的属性如下：

属性	说明
`name`	将两个表关联起来的关系表的名称。若不指定，则使用默认的表名称为`表名1_表名2`
`joinColumns`	关系表中与当前实体对应的外键名称，配合`@JoinColumn`标记使用
`inverseJoinColumns`	关系表中与当前实体所关联的对象对应的外键名称

采用表关联建立一对多单向关联的示例如下：

```
@Entity
public class Teacher implements Serializable
{
    // ...

    @OneToMany(cascade = {CascadeType.ALL})
    @JoinTable(name = "ref_student_teacher", // 关系表的名称
               joinColumns = {
                   // 当前实体类对应的外键
                   @JoinColumn(name="teacher_id", referencedColumnName="id")
               },
               inverseJoinColumns = {
                   // 与当前实体相关联的实体类对应的外键
                   @JoinColumn(name="student_id", referencedColumnName="id")
               }
    )
    private Collection<Student> students = new ArrayList<Student>();

    // ...
}
```

```
@Entity
public class Student implements Serializable
{
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String gender;
    private String name;
    private String major;

    public Student() { }

    // ...
}
```

5.5 多对一单向关联

多对一单向关联用到的标注：`@ManyToOne`、`@JoinColumn`、`@JoinTable`。

`@ManyToOne`标注指明实体类之间是多对一的关系，但它不能指定数据库表中保存的关联字段，所以还要结合`@JoinColumn`或`@JoinTable`标注来指定保存实体关系的配置。

在多对一的实体关系中，表结构的设计方法与一对多是一样的，可以使用外键关联和表关联。

在老师与学生的例子中，老师与学生具有一对多关系，相反地，学生与老师之间具有多对一关系。

采用外键关联建立多对一单向关联的示例如下：

```java
@Entity
public class Teacher implements Serializable
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String teacherName;
    private int age;
    private String gender;

    public Teacher() { }

    // ...
}
```

```java
@Entity
public class Student implements Serializable
{
    @Id
    private int id;

    private String name;
    private int age;
    private String gender;

    @ManyToOne(cascade = {CascadeType.ALL}) // 多对一关系
    @JoinColumn(name = "teacher_id") // 指定外键
    private Teacher teacher;

    // ...
}
```

采用表关联建立多对一单向关联的示例如下：

```java
@Entity
public class Teacher implements Serializable
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String teacherName;
    private int age;
    private String gender;

    public Teacher() { }

    // ...
}
```

```java
@Entity
public class Student implements Serializable
{
    @Id
    private int id;

    private String name;
    private int age;
    private String gender;

    @ManyToOne(cascade = {CascadeType.ALL}) // 多对一关系
    @JoinTable(name = "ref_teacher_student", // 关系表的名称
               joinColumns = {
                   // 当前实体类的对应的外键
                   @JoinColumn(name="student_id", referencedColumnName="id")
               },
               inverseJoinColumns = {
                   // 与当前实体相关联的实体类对应的外键
                   @JoinColumn(name="teacher_id", referencedColumnName="id")
               }
    )
    private Teacher teacher;

    // ...
}
```

5.6 一对多双向关联

一对多双向关联用到的标注：`@OneToMany`、`@ManyToOne`、`@JoinColumn`、`@JoinTable`。

在 JPA 规范中，“多”方实体类为关系主动方，负责外键记录的更新；“一”方为被动方，没有权力更新外键记录。

仍旧使用学生与教师为例来说明一对多双向关系。由于一名教师可以有多个学生，所以“一”方是教师，而“多”方为学生。因此本例子只需要在多对一单向关联的基础上，在“一”端定义`mappedBy`属性，即在教师实体类中使用`mappedBy`属性，它的值是关系主动方中维护的被动方的属性名。

采用外键关联建立一对多双向关联的示例如下：

```java
@Entity
public class Teacher implements Serializable
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private int age;
    private String gender;
    private String teacherName;

    // "一"方作为被动方，使用 @OneToMany 标注，并设置 mappedBy 属性
    @OneToMany(cascade = {CascadeType.ALL}, mappedBy = "teacher")
    private Collection<Student> students = new ArrayList<Student>();

    // ...
}
```

```java
@Entity
public class Student implements Serializable
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String major;
    private String gender;
    private String name;

    @ManyToOne(cascade = {CascadeType.ALL})
    @JoinColumn(name = "teacher_id")
    private Teacher teacher;

    // ...
}
```

采用表关联建立一对多双向关联的示例如下：

```java
@Entity
public class Teacher implements Serializable
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private int age;
    private String gender;
    private String teacherName;

    // "一"方作为被动方，使用 @OneToMany 标注，并设置 mappedBy 属性
    @OneToMany(cascade = {CascadeType.ALL}, mappedBy = "teacher")
    private Collection<Student> students = new ArrayList<Student>();

    // ...
}
```

```java
@Entity
public class Student implements Serializable
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String major;
    private String gender;
    private String name;

    @ManyToOne(cascade = {CascadeType.ALL})
    @JoinTable(name = "ref_teacher_student", // 关系表的名称
               joinColumns = {
                   // 当前实体类的外键
                   @JoinColumn(name="student_id", referencedColumnName="id")
               },
               inverseJoinColumns = {
                   // 与当前实体相关联的实体类对应的外键
                   @JoinColumn(name="teacher_id", referencedColumnName="id")
               }
    )
    private Teacher teacher;

    // ...
}
```

5.7 多对多单向关联

对于多对多的实体关系，在设计表结构时只能采用表关联的方式。

多对多单向关联用到的标注：`@ManyToMany`、`@JoinTable`。

`@ManyToMany`用于标注实体关系为多对多，它定义的属性与`@OneToMany`相同。

使用学生实体（Student）与教师实体（Teacher）为例来说明如何映射多对多关系。一个学生可以有多个老师，一个老师又可以有多个学生，因此学生和老师是多对多的关系。

```
@Entity
public class Teacher implements Serializable
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private int age;
    private String gender;
    private String teacherName;

    public Teacher() {}

    // ...
}
```

```
@Entity
public class Student implements Serializable
{
    // ...

    @ManyToMany
    @JoinTable(name = "ref_teacher_student",
               joinColumns = {@JoinColumn(name="student_id", referencedColumnName="id")},
               inverseJoinColumns = {@JoinColumn(name="teacher_id", referencedColumnName="id")})
    private Collection<Teacher> teachers = new ArrayList<Teacher>();

    // ...
}
```

5.8 多对多双向关联

多对多双向可以分解为两个多对多单向，仍然需要使用关联表来维护其关系。

配置多对多双向关联的方法是，在关联关系的两端都增加`@ManyToMany`标注，并在关联关系拥有者一端（主动端）配置`@JoinTable`标注，而在另一端（被动端）配置`mappedBy`属性。

本节仍旧使用学生实体（Student）与教师实体（Teacher）为例来说明如何映射多对多双向关系。我们将`Student`实体类作为主动方，需要使用`@ManyToMany`标注和`@JoinTable`标注；`Teacher`实体类作为被动方，需要使用`@ManyToMany`标注并添加`mappedBy`属性。

```
@Entity
public class Teacher implements Serializable
{
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

    private String teacherName;
    private int age;
    private String gender;

    @ManyToMany(mappedBy = "teachers")
    private Collection<Student> students = new ArrayList<Student>();

    public Teacher() {}

    // ...
}
```

```java
@Entity
public class Student implements Serializable
{
    // ...

    @ManyToMany
    @JoinTable(name = "ref_teacher_student",
               joinColumns = {@JoinColumn(name="student_id", referencedColumnName="id")},
               inverseJoinColumns = {@JoinColumn(name="teacher_id", referencedColumnName="id")})
    private Collection<Teacher> teachers = new ArrayList<Teacher>();

    // ...
}
```

注意：

1. 可以随意指定一方为关系主动方，在这个例子中指定`Student`为关系维护端。
2. 多对多关系的维护（建立、删除和更新）由关系主动方来完成，关系被动方无法完成关联关系的维护。
3. 多对多关系中一般不设置级联保存、级联删除、级联更新等操作。

在上面的例子中，关系表 ref_teacher_student 中不包含业务字段。关系表除了在主动实体对象中通过`@JoinTable`进行声明外，在代码中没有任何其他体现，因此关系表中的业务字段在程序中无法进行操作。

如果关联表中有其他业务字段，就需要进行修改，将`@ManyToMany`类型的关联分解成两个相互独立的双向一对多关联。

例如，在学生和老师的关联表中添加 course 字段，就需要定义`Course`实体类，分别建立`Student`与`Course`、`Teacher`与`Course`的一对多双向关联，并且把`Course`类作为每个关联关系的主动方。

```java
@Entity
@Table(name = "course")
public class Course
{
    // ...

    @ManyToOne(cascade = {CascadeType.ALL}, optional = false)
    @JoinColumn(name = "TEACHER_ID", referencedColumnName = "ID")
    private Teacher teacher;

    @ManyToOne(cascade={CascadeType.ALL}, optional = false)
    @JoinColumn(name = "STUDENT_ID", referencedColumnName = "ID")
    private Student student;

    private String course;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    // ...
}
```

```java
@Entity
public class Student implements Serializable
{
    // ...

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String gender;
    private String name;
    private String major;

    @OneToMany(cascade = {CascadeType.ALL}, mappedBy = "student")
    private Collection<Course> courses = new ArrayList<Course>();

    // ...
}
```

```java
@Entity
public class Teacher implements Serializable
{
    // ...

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private int age;
    private String gender;
    private String teacherName;

    @OneToMany(cascade = {CascadeType.ALL}, mappedBy= "teacher")
    private Collection<Course> courses = new ArrayList<Course>();

    // ...
}
```

6 JPQL

6.1 什么是JPQL

JPQL 的全称是 Java Persistence Query Language，即 Java 持久化查询语言。JPQL 是一种简单的类似于 SQL 的基于字符串的语言，用于查询实体，以及实体间的关系。

JPQL 简单、可读性强，允许编写可移植的查询，不必考虑底层数据存储。JPQL 语法与 SQL 很相似。

JPQL 是一种与数据库无关的，基于实体的查询语言。

JPQL 是完全面向对象的，具备继承、多态和关联等特性。

JPQL 操作的是实体类，不是数据库定义的物理模型。

JPQL 是 EJB QL 的一个扩展，加入了很多 EJB QL 中没有的新特性。

JPQL 支持映射（projection）、批量操作（update 和 delete）、子查询、join、group by、having 等操作。

JPQL 支持与 SQL 类似的函数功能（如 max 和 min），但是并不能取代 SQL。

6.2 基本语句

JPQL 基本语句定义：`JPQL_statement ::= select_statement | update_statement | delete_statement`

`select_statement` 定义：`select_statement ::= select_clause from_clause [where_clause] [group by_clause] [having_clause] [order by_clause]`

select 子句返回的类型可以是以下 4 种：

- 一个实体
- 一个实体的属性
- 一个计算结果
- 一个新对象

`update_statement` 定义：`update_statement ::= update_clause [where_clause]`

`delete_statement` 定义：`delete_statement ::= delete_clause [where_clause]`

6.3 标识变量

例如，在语句 `select c from Customer c` 中，`c` 是一个变量，它的类型是 `Customer`，这种变量称为标识变量。标识变量在 from 子句中定义。

标识变量的命名规则与 Java 标识符的命名规则一致。

JPQL 语句中，标识变量后面可以跟一个 `.` 操作符，再跟一个属性名，这种表达方式称为路径表达式，`.` 操作符称为导航操作符。

6.4 在Java中使用JPQL

JPA 使用 `javax.persistence.Query` 接口执行查询实例，`Query` 实例的查询交由 `EntityManager` 构建相应的查询语句。该接口拥有多个执行数据查询的接口方法：

- `Object getSingleResult()`：执行 SELECT 查询语句，并返回一个结果。
- `List getResultList()`：执行 SELECT 查询语句，并返回多个结果。
- `Query setParameter(int position, Object value)`：通过参数位置号绑定查询语句中的参数，如果查询语句使用了命名参数，则可以使用此方法绑定命名参数。
- `Query setMaxResults(int maxResult)`：设置返回的最大结果数。
- `int executeUpdate()`：执行新增、删除或更新操作。

执行 JPQL 查询的步骤：

1. 使用注入或通过 `EntityManagerFactory` 实例获取一个 `EntityManager` 实例。
2. 通过调用相应 `EntityManager` 的 `createQuery()` 方法，创建一个 `Query` 实例。
3. 如果有查询参数，使用相应 `Query` 的 `setParameter()` 方法进行设置。
4. 如果需要，使用 `Query` 的 `setMaxResults()` 和/或 `setFirstResult()` 方法设置要检索的实例的最大数量和/或指定检索的起始实例位置。
5. 如果需要，使用 `Query` 的 `setHint()` 方法设置供应商特定的提示。
6. 如果需要，使用 `Query` 的 `setFlushMode()` 方法设置查询执行的刷新模式，覆盖实体管理器的刷新模式。
7. 使用相应 `Query` 的方法 `getSingleResult()` 或 `getResultList()` 执行查询。如果进行更新或删除操作，必须使用 `executeUpdate()` 方法，它返回已更新或删除的实体实例的数量。

6.5 查询参数

在 JPQL 中，查询参数可以分为位置参数和命名参数。位置参数的定义是通过问号（`?`）加位置来定义。命名参数是通过冒号（`:`）加参数名来定义。

位置参数示例：

```
Query query = em.createQuery("select p from Person p where p.id=?1");
query.setParameter(1, 5311);
```

命名参数示例：

```
Query query = em.createQuery("select p from Person p where p.id=:id");
query.setParameter("id", 5311);
```