



传智播客JPA学习笔记

作者: mzhj <http://mzhj.javaeye.com>

我的博客文章精选

目 录

1. 传智播客JPA学习笔记

1.1 01、全面阐释和精彩总结JPA 3

1.2 02、JPA开发环境和思想介绍 5

1.3 03、搭建JPA开发环境和全局事务介绍 10

1.4 04、第一个JPA实例与JPA主键生成策略 14

1.5 05、日期_枚举等字段类型的JPA映射 20

1.6 06、大数据字段映射与字段延迟加载 24

1.7 07、使用JPA加载_更新_删除对象 28

1.8 08、分析JPA与持久化实现产品对接的源代码 33

1.9 09、使用JPQL语句进行查询 38

1.10 10、JPA中的一对多双向关联与级联操作（一对多关系：一） 42

1.11 11、JPA中的一对多延迟加载与关系维护（一对多关系：二） 47

1.12 12、JPA中的一对一双向关联 53

1.13 13、JPA中的多对多双向关联实体定义与注解设置 59

1.14 14、JPA中的多对多双向关联的各项关系操作 64

1.15 15、JPA中的联合主键 74

1.1 01、全面阐释和精彩总结JPA

发表时间: 2010-07-09

什么是JPA

JPA(Java Persistence API)是Sun官方提出的Java持久化规范。它为Java开发人员提供了一种对象/关联映射工具来管理Java应用中的关系数据。他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，JDO等ORM框架各自为营的局面。值得注意的是，JPA是在充分吸收了现有Hibernate，TopLink，JDO等ORM框架的基础上发展而来的，具有易于使用，伸缩性强等优点。从目前的开发社区的反应上看，JPA受到了极大的支持和赞扬，其中就包括了Spring与EJB3.0的开发团队。着眼未来几年的技术走向，JPA作为ORM领域标准化整合者的目标应该不难实现。

JPA的总体思想和现有Hibernate,TopLink,JDO等ORM框架大体一致。总的来说，JPA包括以下3方面的技术：

- **ORM映射元数据**

JPA支持XML和JDK5.0注释(也可译作注解)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。

- **Java持久化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，开发者可以从繁琐的JDBC和SQL代码中解脱出来。

- **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。

提示：JPA不是一种新的ORM框架，他的出现只是用于规范现有的ORM技术，他不能取代现有的Hibernate，TopLink等ORM框架。相反，在采用JPA开发时，我们仍将使用到这些ORM框架，只是此时开发出来的应用不再依赖于某个持久化提供商。应用可以在不修改代码的情况下在任何JPA环境下运行，真正做到低耦合，可扩展的程序设计。

简单说，JPA干的东西就是Hibernate干的东西，他们的作用是一样的。但要注意的是：JPA只是一套规

范，不是一套产品，Hibernate已经是一套产品了。

他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，JDO等ORM框架各自为营的局面。之前学的Hibernate，实际上我们面对的是Hibernate的API进行开发，那么面对Hibernate的API开发有哪些不好的地方呢？不好的地方是我们跟Hibernate这个产品就会紧密的耦合在一块，如果离开了Hibernate我们是无法在别的ORM框架中使用我们的应用的。那么JPA的出现就是为了结束现在Hibernate，TopLink，JDO等ORM框架各自为营的局面。简单的说就是：你采用JPA开发应用，那么你的应用可以运用在实现了JPA规范的持久化产品中（好比说Hibernate，TopLink,JDO）

JPA这门技术是未来发展的必然趋势，以后我们要采用ORM技术呢，我们不会在面對Hibernate编程，不会在面對TopLink编程，而是面对JPA规范编程。就是说，过了几年之后，你们的应用就会很少面对Hibernate API进行编程，这是为什么呢？这就好比以前我们访问数据库一样，假设以前我们没有JDBC这门技术的话，我们跟各个数据库链接只能使用各个数据库厂商给我们提供的API,去访问他们的数据库，那么自从有了JDBC之后，我们就不再需要面对数据库厂商给我们提供的API进行跟数据库链接了，而是直接使用JDBC这套规范，我们就可以跟各个数据库进行对接。目前，JPA跟Hibernate,TopLink的关系也是一样的，JPA就和JDBC一样，提供一种通用的，访问各个ORM实现产品的桥梁工具。通过JPA技术，我们只需要面对它的规范编程，编出来的应用就可以应用在各个持久化产品中（包括Hibernate,TopLink），就是说你底层用的产品对我来说，已经不再重要了。

总结一下:JPA是一套规范，不是一套产品，那么像Hibernate,TopLink,JDO他们是一套产品，如果说这些产品实现了这个JPA规范，那么我们就可以叫他们为JPA的实现产品。

JPA的主要设计者是Hibernate的设计者。JPA是一种规范不是产品，而Hibernate是一种ORM技术的产品。JPA有点像JDBC，为各种不同的ORM技术提供一个统一的接口，方便把应用移植到不同的ORM技术上。

低耦合一直是我们在软件设计上追求的目标，使用JPA，就可以把我们的应用完全从Hibernate中解脱出来了。

1.2 02、JPA开发环境和思想介绍

发表时间: 2010-07-09

开发JPA依赖的jar文件

注意jar文件不能放在含有中文或是含有空格的路径下，否则可能会出现找不到类或是编译失败的错误。

Hibernate核心包(8个文件) : hibernate-distribution-3.3.1.GA.ZIP

hibernate3.jar

lib\bytecode\cglib\hibernate-cglib-repack-2.1_3.jar (CGLIB库，Hibernate用它来实现PO字节码的动态生成，非常核心的库，必须使用的jar包)

lib\required*.jar

Hibernate注解包(3个文件) : hibernate-annotations-3.4.0.GA.ZIP

hibernate-annotations.jar

lib\ejb3-persistence.jar, hibernate-commons-annotations.jar

Hibernate针对JPA的实现包(3个文件) : hibernate-entitymanager-3.4.0.GA.ZIP

hibernate-entitymanager.jar

lib\test\log4j.jar, slf4j-log4j12.jar

Hiberante封装了JDBC，连接具体的数据库还需要具体数据库的JDBC驱动包，这里使用的是MySQL，需要**MySQL数据库驱动包**：

mysql-connector-java-3.1.10-bin.jar

JPA的配置文件

JPA规范要求在类路径（Eclipse工程的src目录）的META-INF目录下放置

[persistence.xml](#)，文件的名称是固定的，配置模板（**此处是针对Hibernate**）如下：

```
<?xml version="1.0"?>

<persistence xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">

<persistence-unit name="itcast" transaction-type="RESOURCE_LOCAL">

    <properties>
        <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5Dialect" />
        <property name="hibernate.connection.driver_class" value="org.gjt.mm.mysql.Driver" />
        <property name="hibernate.connection.username" value="root" />
        <property name="hibernate.connection.password" value="123456" />
        <property name="hibernate.connection.url" value="jdbc:mysql://localhost:3306/itcast?useUni
        <property name="hibernate.max_fetch_depth" value="3" />
        <property name="hibernate.show_sql" value="true" />
        <property name="hibernate.hbm2ddl.auto" value="update"/>
    </properties>

</persistence-unit>

</persistence>
```

hibernate.hbm2ddl.auto

```
<PROPERTIES>
    <property name="hibernate.show_sql" value="true"></property>
    <property name="hibernate.hbm2ddl.auto" value="create"></property>
</PROPERTIES>
```

其实这个hibernate.hbm2ddl.auto参数的作用主要用于：自动创建|更新|验证数据库表结构。如果不是此方面的需求建议set value="none"。里面可以设置的几个参数：

- **validate** 每次加载hibernate时，验证创建数据库表结构，只会和数据库中的表进行比较，不会创建新表，但是会插入新值。

- **create** 每次加载hibernate时都会删除上一次的生成的表，然后根据你的model类再重新来生成新表，哪怕两次没有任何改变也要这样执行，这就是导致数据库表数据丢失的一个重要原因。
- **create-drop** 每次加载hibernate时根据model类生成表，但是sessionFactory一关闭,表就自动删除。
- **update** 最常用的属性，第一次加载hibernate时根据model类会自动建立起表的结构（前提是先建立好数据库），以后加载hibernate时根据 model类自动更新表结构，即使表结构改变了但表中的行仍然存在不会删除以前的行。要注意的是当部署到服务器后，表结构是不会被马上建立起来的，是要等应用第一次运行起来后才会。

总结：

- 请慎重使用此参数，没必要就不要随便用。
- 如果发现数据库表丢失，请检查hibernate.hbm2ddl.auto的配置

再说点“废话”：

当我们把hibernate.hbm2ddl.auto=create时hibernate先用hbm2ddl来生成数据库schema。当我们把hibernate.cfg.xml文件中hbm2ddl属性注释掉，这样我们就取消了在启动时用hbm2ddl来生成数据库schema。通常只有在不断重复进行单元测试的时候才需要打开它，但再次运行hbm2ddl会把你保存的一切都删除掉（drop）---- create配置的含义是：“在创建SessionFactory的时候，从schema中drop掉所有的表，再重新创建它们”。注意，很多Hibernate新手在这一步会失败，我们不时看到关于Table not found错误信息的提问。但是，只要你根据上面描述的步骤来执行，就不会有这个问题，因为hbm2ddl会在第一次运行的时候创建数据库schema，后续的应用程序重启后还能继续使用这个schema。假若你修改了映射，或者修改了数据库schema,你必须把hbm2ddl重新打开一次。

示例代码：

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence version="1.0"
xmlns:persistence="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 persistence_1_0.xsd ">

    <!--      -->

<!--
```

Name属性用于定义持久化单元的名字（name必选,空值也合法）;

transaction-type 指定事务类型(可选)

ÃÄµµÓÉ Foxit Reader ±à¼-
°æÈ'À»ÓÐ ..°æ²»¾¿
½ö¹©ÆÀ¹Àj£

```
-->
<persistence-unit name="unitName" transaction-type="JTA">

<!-- 描述信息.(可选) -->
<description> </description>

<!-- javax.persistence.PersistenceProvider接口实现类(可选) -->
<provider> </provider>

<!-- Jta-data-source和 non-jta-data-source用于分别指定持久化提供商使用的JTA和/或non-JTA数据源的全局JNDI名称(可选) -->
<jta-data-source>java:/MySqlDS</jta-data-source>
<non-jta-data-source> </non-jta-data-source>

<!-- 声明orm.xml所在位置.(可选) -->
<mapping-file>product.xml</mapping-file>

<!-- 以包含persistence.xml的jar文件为基准的相对路径,添加额外的jar文件.(可选) -->
<jar-file>../lib/model.jar</jar-file>

<!-- 显式列出实体类,在Java SE 环境中应该显式列出.(可选) -->
<class>com.domain.User</class>
<class>com.domain.Product</class>

<!-- 声明是否扫描jar文件中标注了@Entity类加入到上下文.若不扫描,则如下:(可选) -->
<exclude-unlisted-classes/>

<!-- 厂商专有属性(可选) -->
<properties>
  <!-- hibernate.hbm2ddl.auto= create-drop / create / update -->
  <property name="hibernate.hbm2ddl.auto" value="update" />
  <property name="hibernate.show_sql" value="true" />
</properties>

</persistence-unit>

</persistence>
```


通常在企业开发中，有两种做法：

- 1.先建表，后再根据表来编写配置文件和实体bean。使用这种方案的开发人员受到了传统数据库建模的影响。
- 2.先编写配置文件和实体bean，然后再生成表，使用这种方案的开发人员采用的是领域建模思想，这种思想相对前一种思想更加OOP。

建议使用第二种（领域建模思想），从软件开发来想，这种思想比第一种思想更加面向对象。领域建模思想也是目前比较新的一门建模思想，第一种是传统的建模思想，已经有10来年的发展历程了，而领域建模思想是近几年才兴起的，这种思想更加的面向对象。

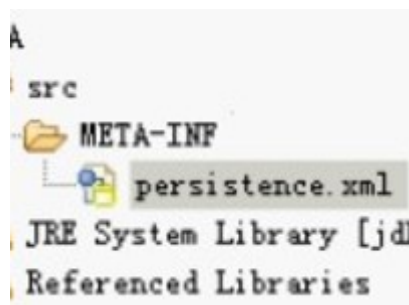
附件下载:

- mysql-connector-java-3.1.10-bin.jar (408.9 KB)
- dl.javaeye.com/topics/download/ece22346-3240-3847-93fe-ac518d43cc35
- JPA开发所用到的所有jar包.rar (5.4 MB)
- dl.javaeye.com/topics/download/906a1117-6a65-32f9-af38-afd1556e874c

1.3 03、搭建JPA开发环境和全局事务介绍

发表时间: 2010-07-09

persistence.xml(JPA规范要求要在类路径的META-INF目录下) , 如下图 :



```
<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/persistence_1_0.xsd"
  version="1.0">

  <persistence-unit name="itcast" transaction-type="RESOURCE_LOCAL">
    <properties>
      <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5Dialect" />
      <property name="hibernate.hbm2ddl.auto" value="update"/>
      <property name="hibernate.connection.driver_class" value="org.gjt.mm.mysql.Driver" />
      <property name="hibernate.connection.username" value="root" />
      <property name="hibernate.connection.password" value="123456" />
      <property name="hibernate.connection.url" value="jdbc:mysql://localhost:3306/itcast?useUnicode=true&characterEncoding=UTF-8" />
    </properties>
  </persistence-unit>

</persistence>
```

persistence.xml这个配置文件的模板可以从哪里找到呢？

因为JPA是一规范，所以你即可以从JPA的规范文档里找到，也可以从任何一个实现了JPA规范的实现产品中
找到。好比Hibernate，可以从hibernate-

entitymanager-3.4.0.GA\doc\reference\en\html_single\index.html中找到。

全局事务 本地事务

全局事务：资源管理器管理和协调的事务，可以跨越多个数据库和进程。资源管理器一般使用 XA 二阶段提交协议与“企业信息系统”(EIS) 或数据库进行交互。

本地事务：在单个 EIS 或数据库的本地并且限制在单个进程内的事务。本地事务不涉及多个数据来源。

<persistence-unit> <persistence-unit/> 标签还有个属性，是 transaction-type (事务的类型)，这属性有两个值，分别是 JTA(全局事务) 和 RESOURCE_LOCAL(本地事务)。

这里我们配置为 transaction-type="RESOURCE_LOCAL"，因为我们只针对一个数据库进行操作，也说只针对一个事务性资源进行操作。

以前我们学习的事务类型都属于本地事务。JTA(全局事务) 和 RESOURCE_LOCAL(本地事务) 有什么区别呢？在某些应用场合，只能使用全局事务，比如：

有两个数据库：

1.mysql 2.oracle 现在有个业务需求--转账

step 1> update mysql_table set amount=amount-xx where id=aaa 发生扣钱,假设是在mysql数据库扣钱的。

step 2> update oracle_table set amount=amount+xx where id=bbb 加钱,假设是在oracle数据库扣钱的。

现在怎么确保两个语句在同一个事务里执行呢？

以前在JDBC里是这样做

connection = mysql 连接mysql

connection.setAutoCommit(false); 不自动提交

1> update mysql_table set amount=amount-xx where id=aaa 发生扣钱,假设是在mysql数据库扣钱的。

2> update oracle_table set amount=amount+xx where id=bbb 发生在oracle数据库

connection.commit();

执行这两条语句，然后通过connection对象提交事务。我们这样子做只能确保这两个语句在同一个数据库mysql里面实现在同一个事务里执行。但是问题是我们现在是要连接到oracle数据库，是不是需要connection2啊？

```
connection = mysql 连接mysql
```

```
connection2 = oracle 连接oracle
```

```
connection.setAutoCommit(false); 不自动提交
```

```
1> update mysql_table set amount=amount-xx where id=aaa 发生扣钱,假设是在mysql数据库扣钱的。
```

```
2> update oracle_table set amount=amount+xx where id=bbb 发生在oracle数据库
```

```
connection.commit();
```

```
connection2.setAutoCommit(false);
```

```
connection2.commit();
```

事务只能在一个connection里打开，并且确保两条语句都在该connection里执行，这样才能让两条语句在同一事务里执行，现在问题就在于connection2是连接到oracle数据库的，那么connection2再开事务有意义吗？它能确保吗？不能，所以在这种情况下就只能使用全局事务了。

这种情况下用普通JDBC操作是满足不了这个业务需求的，这种业务需求只能使用全局事务，本地事务是无法支持我们的操作的，因为这时候，事务的生命周期不应该局限于connection对象的生命周期范围

全局事务怎么做呢？

JPA.getUserTransaction().begin(); 首先要全局事务的API,不需要我们编写，通常容器已经提供给我们了，我们只需要begin一下

```
connection = mysql 连接mysql
```

```
connection2 = oracle 连接oracle
```

```
connection--> update mysql_table set amount=amount-xx where id=aaa 发生扣钱,假设是在mysql数据库扣钱的。
```

```
connection2--> update oracle_table set amount=amount+xx where id=bbb 发生在oracle数据库
```

```
JPA.getUserTransaction().commit();
```

那么它是怎么知道事务该提交还是回滚呢？

这时候它使用了二次提交协议。二次提交协议简单说就这样：如果你先执行第一条语句，执行的结果先预提交到数据库，预提交到数据库了，数据库会执行这条语句，然后返回一个执行的结果，这个结果假如我们用布尔值表示的话，成功就是true，失败就是false.然后把执行的结果放入一个（假设是List）对象里面去，接下来再执行第二条语句，执行完第二条语句之后（也是预处理，数据库不会真正实现数据的提交，只是说这条语句送到数据库里面，它模拟下执行，给你返回个执行的结果），假如这两条语句的执行结果在List里面都是true的话，那么这个事务就认为语句是成功的，这时候全局事务就会提交。二次提交协议，数据库在第一次提交这个语句时，只会做预处理，不会发生真正的数据改变，当我们在全局事务提交的时候，这时候发生了第二次提交，那么第二次提交的时候才会真正的发生数据的改动。

如果说在执行这两条语句中，有一个出错了，那么List集合里就有个元素为false，那么全局事务就认为你这个事务是失败的，它就会进行回滚，回滚的时候，哪怕你的第二条语句在第一次提交的时候是成功的，它在第二次提交的时候也会回滚，那么第一次的更改也会恢复到之前的状态，这就是二次提交协议。（可以查看一下数据库方面的文档来了解二次提交协议）

回到persistence.xml的配置里面去，事务类型有两种，什么时候该用全局事务(JTA)?什么时候改用本地事务(RESOURCE_LOCAL)?应有你的业务应用需求来定，我们的大部分应用只是需要本地事务。全局事务通常是在应用服务器里使用，比如weblogic,JBoss,学习EJB3后给大家介绍。企业面试时被问到就要注意了（事务类型有哪几种？分别用在什么场景下？）

1.4 04、第一个JPA实例与JPA主键生成策略

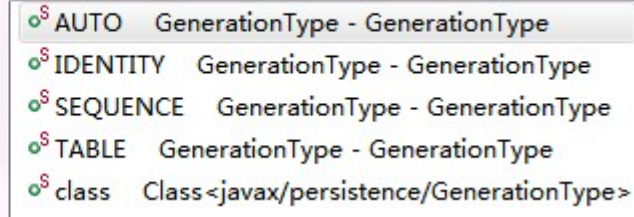
发表时间: 2010-07-13

写实体bean，映射的数据可以采用XML配置方式，也可以采用注解方式，在JPA中推荐大家用注解的方式，因为注解的方式开发应用效率是挺高的。

~~每个实体bean都要有个实体标识属性，这个实体标识属性主要用于在内存里面判断对象。~~通过@Id就可以定义实体标识。可以标识在属性的get方法前面，也可以标识在字段上面，通常我们更倾向于标识在属性的get方法上面。

如果我们希望采用数据库的id自增长的方式来生成主键值的话，这时候我们要用到一个注解@GeneratedValue，这注解里面有一些属性，其中一个策略strategy，生成主键值的方案，JPA里没有Hibernate提供的那么多方案，它提供的方案有如下图：

```
@Id @GeneratedValue(strategy=GenerationType.  
public Integer getId() {  
    return id;  
}  
  
public void setId(Integer id) {  
    this.id = id;  
}
```



1.

- AUTO：JPA自动选择合适的策略，是默认选项；
- IDENTITY：采用数据库ID自增长的方式来生成主键值，Oracle不支持这种方式；
- SEQUENCE：通过序列产生主键，通过@SequenceGenerator注解指定序列名，MySQL不支持这种方式；
- TABLE：采用表生成方式来生成主键值，那怎么样生成呢？很简单，表里面通常有两个字段，第一个字段是给它一个名称（就是个列名而已），第二个字段专门用来累加用的，就是说每访问一次这个表，第二个字段就会累加1，不断累加。就是说你们要得到这个主键值的话，访问这个表，然后update这个表的这个字段，把它累加1之后，然后再把这个值取出来作为主键，再给他赋进去，表生成就是这样。
- Oracle数据库默认情况下，不能支持用id自增长方式来生成主键值；

- mysql在默认情况下不能支持SEQUENCE序列的方式来生成主键值，所以我们一定要注意我们使用的数据库。
- TABLE表生成方式才是通用的，但是这种方式效率并不高。
- 如果我们开发的应用，我们不可以预测用户到底使用哪种数据库，那么这个时候应该设为哪个值呢？答案是AUTO，就是说由持久化实现产品，来根据你使用的方言来决定它采用的主键值的生成方式，到底是IDENTITY？还是SEQUENCE？还是TABLE？如果用的是Hibernate,那么它会用IDENTITY这种生成方式来生成主键值。

IDENTITY和SEQUENCE这两种生成方案通不通用啊？对所有数据库：

注意：如果我们把策略strategy设置成@GeneratedValue(strategy=GenerationType.AUTO)的话，AUTO本身就是策略的默认值，我们可以省略掉，就是说简单写成这样@GeneratedValue

摘自CSDN：

@GeneratedValue：主键的产生策略，通过strategy属性指定。默认情况下，JPA自动选择一个最适合底层数据库的主键生成策略，如SqlServer对应identity，MySql对应auto increment。在javax.persistence.GenerationType中定义了以下几种可供选择的策略：

- 1) IDENTITY：采用数据库ID自增长的方式来自增主键字段，Oracle不支持这种方式；
- 2) AUTO：JPA自动选择合适的策略，是默认选项；
- 3) SEQUENCE：通过序列产生主键，通过@SequenceGenerator注解指定序列名，MySql不支持这种方式；
- 4) TABLE：通过表产生主键，框架借由表模拟序列产生主键，使用该策略可以使应用更易于数据库移植。不同的JPA实现商生成的表名是不同的，如OpenJPA生成openjpa_sequence_table表，Hibernate生成一个hibernate_sequences表，而TopLink则生成sequence表。这些表都具有一个序列名和对应值两个字段，如SEQ_NAME和SEQ_COUNT。

可参考<http://blog.csdn.net/lzxvip/archive/2009/06/19/4282484.aspx>

实体bean开发完后，就要用持久化API对实体bean进行添删改查的操作，我们学习持久化API的时候，可以对照Hibernate来学习，接下来建立个单元测试，在开发的过程中，建议大家一定要用单元测试（junit可以用来进行单元测试）。

第一步写：persistence.xml（要求放在类路径的META-INF目录下）

```
<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/persistence_1_0.xsd" version="1.0">
```



```
<persistence-unit name="itcast" transaction-type="RESOURCE_LOCAL">

    <properties>
        <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5Dialect" />
        <property name="hibernate.hbm2ddl.auto" value="update"/>
        <property name="hibernate.connection.driver_class" value="org.gjt.mm.mysql.Driver" />
        <property name="hibernate.connection.username" value="root" />
        <property name="hibernate.connection.password" value="456" />
        <property name="hibernate.connection.url" value="jdbc:mysql://localhost:3306/itcast?useUnicode=true&characterEncoding=utf8" />
    </properties>

</persistence-unit>
</persistence>
```

第二步写：Person.java（实体bean）

```
package cn.itcast.bean;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity //以javax开头的包，都是Sun公司制定的一些规范
public class Person {
    private Integer id;
    private String name;

    public Person() {
        //对象是由Hibernate为我们创建的，当我们通过ID来获取某个实体的时候，这个实体给我们返回了这个对象的创建是由Hibernate内部通过反射技术来创建的，反射的时候用到了默认的构造函数，所以这时候必须给它提供一个public的无参构造函数。*/
    }
}
```



```
public Person(String name) {
    this.name = name;
}

@Id
@GeneratedValue(strategy=GenerationType.AUTO) //auto默认，可不写，直接写@GeneratedValue
public Integer getId() {
    return id;
}

public void setId(Integer id) {
    this.id = id;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}
}
```

第三步：PersonTest.java（junit单元测试）

```
package junit.test;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import org.junit.BeforeClass;
```

```
import org.junit.Test;

import cn.itcast.bean.Person;

public class PersonTest {

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
    }

    @Test public void save(){
        //对实体bean进行操作，第一步应该获取什么对象啊？      SessionFactory对象
        //这里用获取的EntityManagerFactory对象，这可以把它看成跟Hibernate的SessionFactory对象差不多的东西
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("itcast")
        //参数"itcast"是persistence.xml文件中<persistence-unit  name="itcast">name的属性值。
        EntityManager em = factory.createEntityManager();
        em.getTransaction().begin();    //开启事务
        em.persist(new Person("传智播客")); //持久化对象
        em.getTransaction().commit();    //提交事务
        em.close();
        factory.close();

        //SessionFactory --> Session --> begin事务
    }
}
```

/*

session.save(obj);

persist这方法在Hibernate里也存在，Hibernate的作者已经不太推荐大家用save方法，而是推荐大家用persist方法。

why? 首先并不是代码上的问题，主要是这个名字上的问题，因为我们把这个ORM技术叫做持久化产品，那么我们对某个对象持久化,应该叫持久化，而不应该叫保存，所以后来Hibernate的作者推荐用persist方法，这并不是功能的问题，主要是取名的问题，所以用persist方法也可以。

*/

目前数据库表是不存在的，我们采取实体（领域）建模的思想，让它根据实体bean来生成数据库表，在persistence.xml里，<property name="hibernate.hbm2ddl.auto" value="update"/>，生成策略是update,就是说表不存在的时候，它会创建数据库表。

问题，它什么时候创建表啊？创建表的时机是在什么时候创建的啊？答案是得到SessionFactory的时候，在JPA里也一样，是我们**得到EntityManagerFactory的时候创建表，也就是说我们只要执行下面的那段代码就生成表了。**

```
public class PersonTest {

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
    }

    @Test public void save(){
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("itcast")
        factory.close();
    }

}
```

通过这个特性，可以在开发的时候，用来验证我们编写的实体映射元数据是否是正确的，通过这个就可以判断。如果生成不了表，就说明是编写的实体映射出问题了（比如实体bean），以后要学会怎样排错。

1.5 05、日期 枚举等字段类型的JPA映射

发表时间: 2010-07-13

映射元数据是什么样的？不设置默认的情况下：实体类Person生成表是Person表；**字段id,name,采用bean中getXXX、setXXX的XXX名称作为字段的名称，而不是采用属性的名称作为字段名称；**

Person.java

```
package cn.itcast.bean;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.EnumType;
import javax.persistence.Enumerated;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

import java.util.Date;

@Entity          //以javax开发的包，都是Sun公司制定的一些规范
@Table(name = "PersonTable")      //改变数据库中映射表名
public class Person {
    private Integer id;
    private String name;
    private Date birthday; //1987-12-10
    private Gender gender = Gender.MAN; //这里可以设置默认值， Gender是一个枚举类型。

    @Enumerated(EnumType.STRING) //说明这个属性是个枚举类型，括号内的表示存入数据库的枚举字符串而不是枚举索引
    @Column(length = 5, nullable = false) //Eclipse代码助手快捷键为ALT+/
    public Gender getGender() {
        return gender;
    }
}
```

```
    }

    public void setGender(Gender gender) {
        this.gender = gender;
    }

    @Temporal(TemporalType.DATE) //说明这个属性映射到数据库中是一个日期类型，括号中的是日期格式
    public Date getBirthday() {
        return birthday;
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }

    public Person() {
        /* 对象是由Hibernate为我们创建的，当我们通过ID来获取某个实体的时候，这个实体给我们返回了这个对象的创建是由Hibernate内部通过反射技术来创建的，反射的时候用到了默认的构造函数，所以这时候必须给它提供一个public的无参构造函数。 */
    }

    public Person(String name) {
        this.name = name;
    }

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    // auto是默认值，可不写，直接写@GeneratedValue
    public Integer getId() {
        return id;
    }

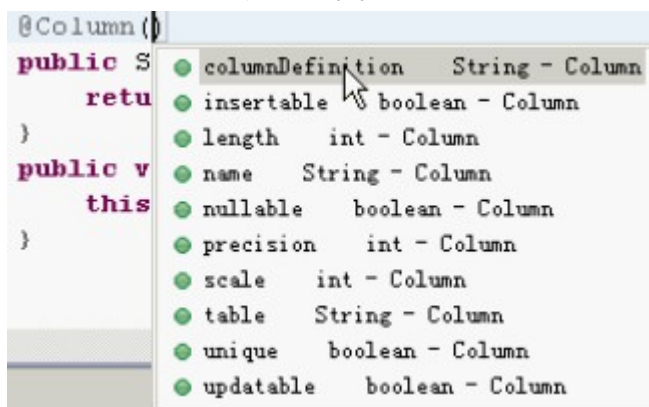
    public void setId(Integer id) {
        this.id = id;
    }

    @Column(length = 10, nullable = false, name = "personName")
    public String getName() {
        return name;
    }
}
```

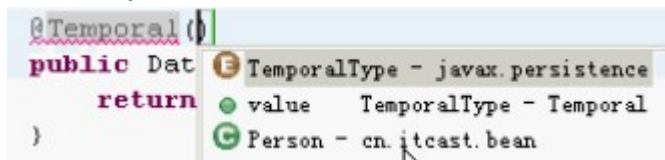
```
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

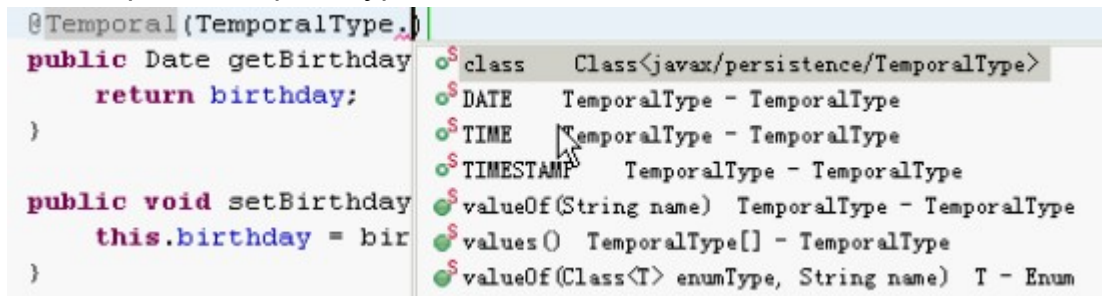
@Column 的选项。 看图：



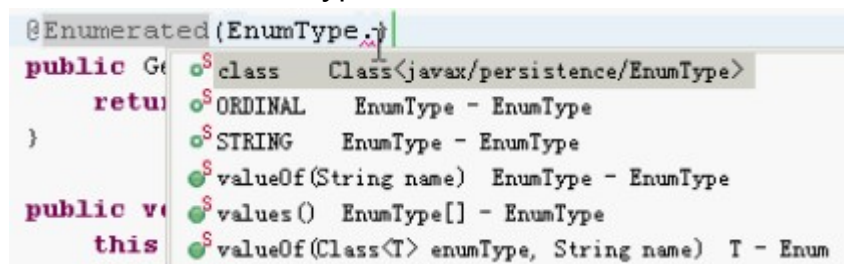
@Temporal 的选项。 看图：



@Temporal(TemporalType.)的选项。 看图：



@Enumerated(EnumType.) 的选项。 看图：



The screenshot shows a code editor with a tooltip for the `@Enumerated` annotation. The tooltip lists the following options for the `enumType` parameter:

- `class` `Class<java.lang.Enum>`
- `ORDINAL` `EnumType - EnumType`
- `STRING` `EnumType - EnumType`
- `valueOf(String name)` `EnumType - EnumType`
- `values()` `EnumType[] - EnumType`
- `valueOf(Class<T> enumType, String name)` `T - Enum`

1.6 06、大数据字段映射与字段延迟加载

发表时间: 2010-07-13

Person.java

```
package cn.itcast.bean;

import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.EnumType;
import javax.persistence.Enumerated;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Lob;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import javax.persistence.Transient;

import java.util.Date;

@Entity
//以javax开发的包，都是Sun公司制定的一些规范
@Table(name = "PersonTable")
public class Person {
    private Integer id;
    private String name;
    private Date birthday;
    private Gender gender = Gender.MAN; // 这里可以设置默认值
    private String info;
    private Byte[] file;
    private String imagePath; //该属性不希望成为可持久化字段
```



```
@Transient    //这个注解用来标注imagePath这个属性不作为可持久化字段，就是说不跟数据库的字段映射
public String getImagePath() {
    return imagePath;
}

public void setImagePath(String imagePath) {
    this.imagePath = imagePath;
}

@Lob    //申明属性对应的数据库字段为一个大文本类，文件属性也是用这个声明映射。
public String getInfo() {
    return info;
}

public void setInfo(String info) {
    this.info = info;
}

@Lob    //声明属性对应的是一个文件数据字段。
@Basic(fetch = FetchType.LAZY)    //设置为延迟加载，当我们在数据库中取这条记录的时候，不会去取文件
public Byte[] getFile() {
    return file;
}

public void setFile(Byte[] file) {
    this.file = file;
}

@Enumerated(EnumType.STRING)
@Column(length = 5, nullable = false)
public Gender getGender() {
    return gender;
}

public void setGender(Gender gender) {
    this.gender = gender;
}
```

```
}

@Temporal(TemporalType.DATE)
public Date getBirthday() {
    return birthday;
}

public void setBirthday(Date birthday) {
    this.birthday = birthday;
}

public Person() {
    /* 对象是由Hibernate为我们创建的，当我们通过ID来获取某个实体的时候，这个实体给我们返回了这个对象的创建是由Hibernate内部通过反射技术来创建的，反射的时候用到了默认的构造函数，所以这时候必须给它提供一个public的无参构造函数。*/
}

public Person(String name) {
    this.name = name;
}

@Id
@GeneratedValue(strategy = GenerationType.AUTO)
// auto是默认值，可不写
public Integer getId() {
    return id;
}

public void setId(Integer id) {
    this.id = id;
}

@Column(length = 10, nullable = false, name = "personName")
public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}
```

```
}  
  
}
```

生成的数据库字段类型如下：

Properties	Fields	Indices	Foreign Keys	Triggers	Data	Dependencies	DDL
Field Name		Field Type	Size	Precision	Not Null	Default	
id		INTEGER	11	0	<input checked="" type="checkbox"/>	Null	
birthday		DATE	0	0	<input type="checkbox"/>	Null	
file		LONGBLOB	0	0	<input type="checkbox"/>	Null	
gender		VARCHAR	5	0	<input checked="" type="checkbox"/>		
info		LONGTEXT	0	0	<input type="checkbox"/>	Null	
personName		VARCHAR	10	0	<input checked="" type="checkbox"/>		

假如 `private Byte[] file;`保存的是一个文件，如果我们要获取一个Person对象的话，会把file这个字段保存的内容找回来，并且放在内存里面。（如果我们保存的文件有50M，那每次获取Person bean的时候，都会获取file这个文件，在内存中可能是50.1M，这样太占资源了，怎么办？）可以给file加`@Basic(fetch = FetchType.LAZY)`这个注解，如果我们设置了延迟加载，那么当我们调用Hibernate的get方法得到Person这个记录的时候，如果没有访问file这个属性的get方法的话，那么它就不会从数据库里帮我们把这个file得到；如果说你要访问这个file属性，那么它才会从数据库里面把这个file数据装载上来。也就是说，只要我们不访问它，那么它就不会从数据库里面把数据装载进内存里面。如果不装载文件的话，那么得到的Person记录可能就是0.1M左右，当然，如果你访问了file这个属性的话，那么它会从数据库里面把数据再装载一次上来，在内存里可能就有50.1M了。所以，`@Basic`这个标签一般用在大数据，也就是说你存放的数据大小比较大的话，大概数据如果超过1M的话，就应该使用`@Basic`标签，把属性做延迟初始化，那么当初次得到Person对象的时候，就不会立刻去装载数据，而是在第一次访问的时候才去装载file数据。当然在第一次访问file的时候，必须要确保EntityManager这个对象要处于打开状态（就好比session对象要处于打开状态一样），假如EntityManager对象被close了的话，我们再访问它的延迟属性会出现延迟加载例外，这个在Hibernate的教程里也经常遇到这问题。

1.7 07、使用JPA加载_更新_删除对象

发表时间: 2010-07-13

PersonTest.java

```
package junit.test;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import org.junit.BeforeClass;
import org.junit.Test;

import cn.itcast.bean.Person;

public class PersonTest {

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
    }

    @Test public void save(){
        //对实体bean进行操作，第一步应该获取什么对象啊？      SessionFactory对象。
        //这里用获取的EntityManagerFactory对象，这可以把它看成跟Hibernate的SessionFactory对象差不多的东西。
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("itcast");
        EntityManager em = factory.createEntityManager();
        em.getTransaction().begin();//开启事务
        em.persist(new Person("传智播客"));
        em.getTransaction().commit();
        em.close();
        factory.close();

        //SessionFactory --> Session --> begin事务
    }
}
```




```
        EntityManager em = factory.createEntityManager();
        em.getTransaction().begin(); //开启事务，只读取数据不需要开事务，更改数据的时候需要开事务。
        Person person=em.find(Person.class,1); //相当于Hibernate的get方法,第一个参数是实体Bean类（get方法采用泛型）（传什么类的class对象进去就返回什么类的对象回来），第二个参数是实体bean标识符的值（即ID的值，传入哪个id值返回的就是id为那个值的那条记录）。
        person.setName("老张"); //这个person是处于一个托管状态的bean对象，用set方法改变的属性并不会马上更新到数据库里面去
        //而是把这个操作放在一个jdbc的批处理里面，等到事务提交的时候才会更新到数据库里面去。
        em.getTransaction().commit();
        em.close();
        factory.close();
    }


    @Test public void updatePerson2(){
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("itcast");
        EntityManager em = factory.createEntityManager();
        em.getTransaction().begin(); //开启事务。
        Person person=em.find(Person.class,1);
        em.clear(); //把实体管理器中的所有实体变成游离状态。
        person.setName("老黎");
        em.merge(person); //更新处于游离状态的bean对象。
        em.getTransaction().commit();
        em.close();
        factory.close();
    }

    @Test public void delete(){
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("itcast");
        EntityManager em = factory.createEntityManager();
        em.getTransaction().begin(); //开启事务。
        Person person=em.find(Person.class,1);
        em.remove(person); //删除的bean对象也必须是处于托管状态的对象才能被删除成功。否则，如果person不是托管状态的bean，也不会报错但数据库内的数据也不会删除。
        em.getTransaction().commit();
        em.close();
        factory.close();
    }
}
```

我们目前使用的是Hibernate，实际上我们操纵EntityManager对象时，它内部是操纵了Hibernate里面的session对象。它内部只是对session对象做了个封装而已。

```
@Test public void getPerson(){
    EntityManagerFactory factory = Persistence.createEntityManagerFactory("itcast")
    EntityManager em = factory.createEntityManager();
    Person person = em. (Person.class,2); //相当于Hibernate的get方法。
    System.out.println(person);
    em.close();
    factory.close();
}
```

如果不存在id为2的person的话，那么返回的是null值。

```
@Test public void getPerson2(){
    EntityManagerFactory factory = Persistence.createEntityManagerFactory("itcast")
    EntityManager em = factory.createEntityManager();
    Person person = em. (Person.class,2); //1
    System.out.println(person); //2
    em.close();
    factory.close();
}
```

如果不存在id为2的person的话，那么返回的是异常（`javax.persistence.EntityNotFoundException`）。

异常是在什么时候触发的呢？是在1？还是2呢？

答案是2。这说明并不是`em.getReference()`这个方法执行时就发生异常，而是在你访问这个对象或是它属性的时候才出现异常。

数据库里没有相应的记录，EntityManagerd对象的get方法获取不到记录会返回null，而EntityManagerd对象的getReference方法获取不到记录会在下一次访问这个返回值的时候抛出异常。

```
@Test public void updatePerson2(){
    EntityManagerFactory factory = Persistence.createEntityManagerFactory("itcast")
    EntityManager em = factory.createEntityManager();
    em.getTransaction().begin();//开启事务。
    Person person=em.find(Person.class,1);
    em.clear(); //把实体管理器中的所有实体变成游离状态。
    person.setName("老黎");
    em.getTransaction().commit();
    em.close();
    factory.close();
}
```

在clear之后，person变成了游离状态，这时候对游离状态的实体进行更新的话（person.setName("老黎");），更新的数据是不能同步到数据库的。可以采用方法em.merge(person);这方法是用于把在游离状态时候的更新同步到数据库。

1.8 08、分析JPA与持久化实现产品对接的源代码

发表时间: 2010-07-13

EntityManagerFactory factory = Persistence.createEntityManagerFactory("itcast");

讲解下这个方法内部的一些原理（了解下就OK）

打开源代码Persistence.java (用DJ Java Decompiler 3.7反编译的代码)

```
// Decompiled by DJ v3.7.7.81 Copyright 2004 Atanas Neshkov   Date: 2010-7-12 20:30:06
// Home Page : http://members.fortunecity.com/neshkov/dj.html   - Check often for new version!
// Decompiler options: packimports(3)
// Source File Name:   Persistence.java

package javax.persistence;

import java.io.*;
import java.net.URL;
import java.util.*;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import javax.persistence.spi.PersistenceProvider;

// Referenced classes of package javax.persistence:
//      PersistenceException, EntityManagerFactory

public class Persistence
{
    public Persistence()
    {

    }

    public static EntityManagerFactory createEntityManagerFactory(String persistenceUnitName)
    {
        return createEntityManagerFactory(persistenceUnitName, null);
    }
}
```

```
}

public static EntityManagerFactory createEntityManagerFactory(String persistenceUnitName, Map properties)
{
    EntityManagerFactory emf = null;
    if(providers.size() == 0)
        findAllProviders();
    Iterator i$ = providers.iterator();
    do
    {
        if(!i$.hasNext())
            break;
        PersistenceProvider provider = (PersistenceProvider)i$.next();
        emf = provider.createEntityManagerFactory(persistenceUnitName, properties);
    } while(emf == null);
    if(emf == null)
        throw new PersistenceException((new StringBuilder()).append("No Persistence provider for EntityManager named ").append(persistenceUnitName).toString());
    else
        return emf;
}

private static void findAllProviders()
{
    ClassLoader loader;
    Enumeration resources;
    Set names;
    loader = Thread.currentThread().getContextClassLoader();
    resources = loader.getResources((new StringBuilder()).append("META-INF/services/").append(javax.persistence.spi.PersistenceProvider.getName()).toString());
    names = new HashSet();
_L2:
    InputStream is;
    if(!resources.hasMoreElements())
        break; /* Loop/switch isn't completed */
    URL url = (URL)resources.nextElement();
    is = url.openStream();
    names.addAll(providerNamesFromReader(new BufferedReader(new InputStreamReader(is))));
    is.close();
}
```

```
        if(true) goto _L2; else goto _L1
        Exception exception;
        exception;
        is.close();
        throw exception;
_L1:
        Class providerClass;
for(Iterator i$ = names.iterator(); i$.hasNext(); providers.add((PersistenceProvider)providerClass.newInstance()))
    {
        String s = (String)i$.next();
        providerClass = loader.loadClass(s);
    }

        break MISSING_BLOCK_LABEL_214;
        IOException e;
        e;
        throw new PersistenceException(e);
        e;
        throw new PersistenceException(e);
        e;
        throw new PersistenceException(e);
        e;
        throw new PersistenceException(e);
    }

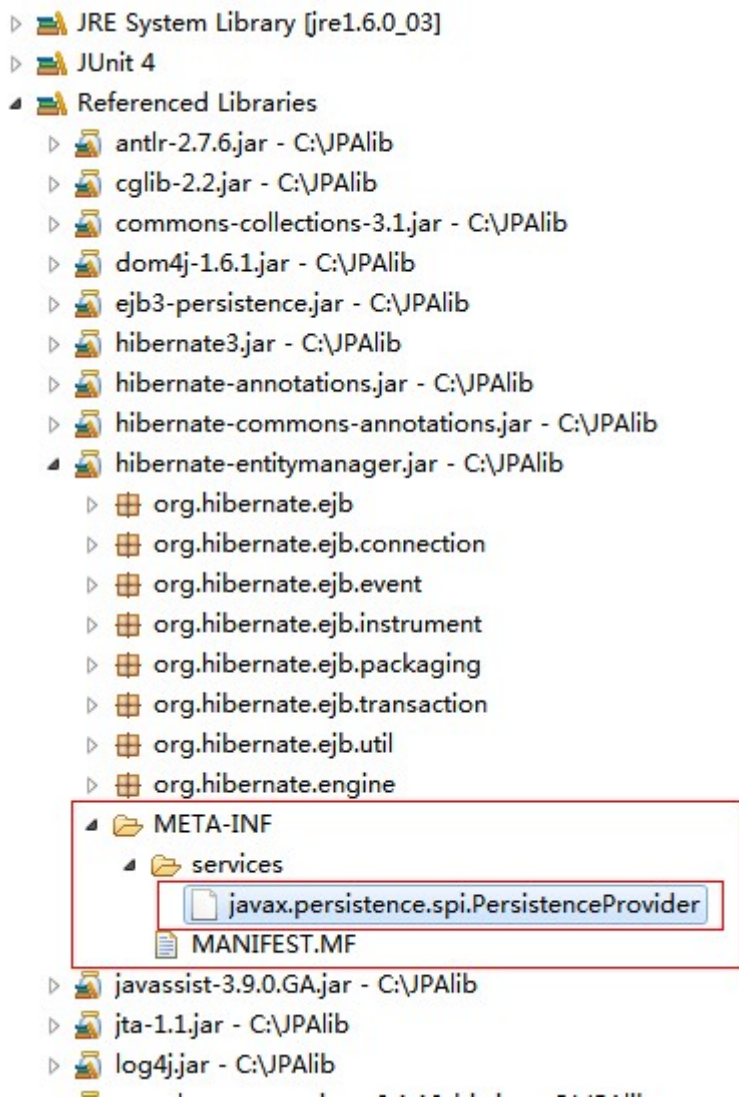
private static Set providerNamesFromReader(BufferedReader reader)
    throws IOException
{
    Set names = new HashSet();
    do
    {
        String line;
        if((line = reader.readLine()) == null)
            break;
        line = line.trim();
        Matcher m = nonCommentPattern.matcher(line);
        if(m.find())
```

```
        names.add(m.group().trim());
    } while(true);
    return names;
}

public static final String PERSISTENCE_PROVIDER = "javax.persistence.spi.PersistenceProvider";
protected static final Set providers = new HashSet();
private static final Pattern nonCommentPattern = Pattern.compile("^(^#)+");

}
```

这个资源在哪里呢？看图：



打开，内容为 `org.hibernate.ejb.HibernatePersistence`

程序会在类路径下寻找到这个文件，并读取这个配置文件里面指定的可持久化驱动。

Hibernate提供的可持久化驱动就是`org.hibernate.ejb.HibernatePersistence`这个类，这个类是Hibernate的入口类，类似JDBC里面的驱动类。

当然，不同的可持久化产品的入口类是不同的，

调用JPA应用，它能使用Hibernate，是因为有这样一个驱动类，它起到了一个桥梁的作用，过渡到Hibernate的产品上，这就是调用`EntityManagerFactory factory = Persistence.createEntityManagerFactory("itcast");`创建实体管理器方法的一些执行细节

`factory` 是由Hibernate的可持久化驱动类创建出来的，如果观察Hibernate的实现类的话，会发现实际上`EntityManagerFactory` 是对`SessionFactory`这个类进行了一层封装。

包括`EntityManager`类也是对`Session`对象进行了一层封装而已。

只要研究下Hibernate的JPA实现代码就可以观察出来

1.9 09、使用JPQL语句进行查询

发表时间: 2010-07-13

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。

PersonTest.java

```
package junit.test;

import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;
import org.junit.BeforeClass;
import org.junit.Test;
import cn.itcast.bean.Person;

public class PersonTest {

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {

    }

    @Test
    public void save() {
        // 对实体bean进行操作，第一步应该获取什么对象啊？ SessionFactory对象。
        // 这里用获取的EntityManagerFactory对象，这可以把它看成跟Hibernate的SessionFactory对象差不多的东西。
        EntityManagerFactory factory = Persistence
            .createEntityManagerFactory("itcast");
        EntityManager em = factory.createEntityManager();
        em.getTransaction().begin();    // 开启事务。
        em.persist(new Person("传智播客"));
```

```

        em.getTransaction().commit();

        em.close();

        factory.close();

        // sessionFactory --> Session --> begin事务
    }

```

```

/*

```

```

    session.save(obj);

```

persist这方法在Hibernate里也存在，Hibernate的作者已经不太推荐大家用save方法，而是推荐大家用persist方法。why? 首先并不是代码上的问题，主要是这个名字上的问题，因为我们把这个ORM技术叫做持久化产品，那么我们对某个对象持久化，应该叫持久化而不应该叫保存，所以后来Hibernate的作者推荐用persist方法，这并不是功能的问题，主要是取名的问题，所以用推荐用persist方法。

```

*/

```

```

@Test

```

```

public void query1() {

```

```

    EntityManagerFactory factory = Persistence

```

```

        .createEntityManagerFactory("itcast");

```

```

    EntityManager em = factory.createEntityManager();

```

```

    // 只是想获取数据，那么创建的查询语句可以不在事务里创建，不需要开启事务。

```

```

    // 但是如果通过语句去更新数据库的话，就必须打开事务了，否则不会保存成功。

```

```

    Query query = em.createQuery("select o from Person o where o.id = ?1"); //?1采用位置参数查询，?1表示一个参数。

```

```

    // 和Hibernate一样，都是面向对象的语句，不是sql语句。 里面出现的都是实体的名称和实体的属性。

```

```

    //?1表示第一个参数，后面可以用Query对象的setParameter方法设置这个参数的内容。参数可以不用从1开始，可以从2等其他数字开始。

```

```

    // JPA规范的写法前面是要加"select o"的(o是起的别名，名字可以任意)，而Hibernate是可以省略的，但是如果你选用的JPA实现产品是Hibernate的话，不写也不会出错。但是可能移植到别的可持久化实现产品中就有可能出错，所以我们应该严格按照JPA规范来编写代码。

```

```

    query.setParameter(1, 1); //设置第一个参数的值为1。

```

不要直接上面的语句里写值，因为如果你用JDBC的话就会存在一个问题：注入sql工具，注入sql工具在ASP年代（00-03年），sql工具是个高峰期，就因为我们的开发人员把从请求参数里面得到参数值后，直接赋进去，那么这时候就可能有问题，

如果别人输入的是正确的，那当然好咯，如果输入错误的，

比如 String name = request.getParameter("name"); 假如name="";delete from Person"的话，那么sql语句会变成select o from Person o where o.id =;delete from Person, 这样在sql server数据库里面，必然会把数据表里面的数据全部删除掉，所以我们一定要注意sql工具，在开发应用时也要注意，所以在写参数的时候，不要把参数直接写进去，也不要采用字符串拼接的方式来使用，而是应该采用命名参数查询，或者位置参数查询，JPA里面也提供了两种，命名参数查询(:id)和位置参数查询(?)；JPA里面提供了更方便的实现，可以给?编号，就是?后面跟数字。

```

    //防sql注入其实就是参数设置的地方只能允许设置为一个参数，并且参数设置的方法还会去检查参数设置的合理性。如果是字符串拼接等方法就存在安全问题了。

```

```

    Person person = (Person) query.getSingleResult();

```

```

    // getSingleResult方法相当于Hibernate里的Session.createQuery("").uniqueResult();

```

```

    // 使用getSingleResult方法的前提是必须保证记录存在，如果记录不存在，那么会出错的。这个方法还需要query对象里的记录是唯一的，有多个记录也会出错。

```

```

    System.out.println(person.getName());

```

```

    em.close();

```

```

    factory.close();

```

```

}

```

```
@Test
public void query2() {
    EntityManagerFactory factory = Persistence
        .createEntityManagerFactory("itcast");
    EntityManager em = factory.createEntityManager();
    Query query = em.createQuery("select o from Person o where o.id = ?1");
    query.setParameter(1, 1);
    List<Person> persons = query.getResultList(); //返回的是一个list，这里指定了list的泛型是Person类型的。

    for (Person person : persons) {
        System.out.println(person.getName());
    }
    em.close();
    factory.close();
}
```

```
@Test
public void deleteQuery() {
    EntityManagerFactory factory = Persistence
        .createEntityManagerFactory("itcast");
    EntityManager em = factory.createEntityManager();
    em.getTransaction().begin(); // 进行数据的更改操作，必须开启事务。
    Query query = em.createQuery("delete from Person o where o.id = ?1");
    query.setParameter(1, 1);
    query.executeUpdate();
    em.getTransaction().commit();
    em.close();
    factory.close();
}
```

```
@Test
public void updateQuery() {
    EntityManagerFactory factory = Persistence
        .createEntityManagerFactory("itcast");
    EntityManager em = factory.createEntityManager();
    em.getTransaction().begin(); // 开启事务。
    Query query = em.createQuery("update Person o set o.name = :name where o.id = :id"); // 采用命名参数查询。
```



```
        query.setParameter("name", "xxx");  
        query.setParameter("id", 2);  
        query.executeUpdate();  
        em.getTransaction().commit();  
        em.close();  
        factory.close();  
    }  
}
```

1.10 10、JPA中的一对多双向关联与级联操作（一对多关系：一）

发表时间: 2010-07-13

Order.java

```
package cn.itcast.bean;

import java.util.HashSet;
import java.util.Set;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToMany;

@Entity
public class Order {
    private String orderId;
    private Float amount = 0f;
    private Set<OrderItem> items = new HashSet<OrderItem>();

    @Id //要注意：目前JPA规范并没有提供UUID这种生成策略，目前主键值只提供了整型的生成方式，所
    @Column(length = 12)
    public String getOrderId() {
        return orderId;
    }

    public void setOrderId(String orderId) {
        this.orderId = orderId;
    }

    @Column(nullable = false)
    public Float getAmount() {
```

AIAppOE Foxit Reader ±a%
°æE Å»OD ·-°æ²»³/4¿
½ö¹©ÆÄ¹Àj£

```
        return amount;
    }

    public void setAmount(Float amount) {
        this.amount = amount;
    }

    @OneToMany(cascade = { CascadeType.REFRESH, CascadeType.PERSIST,
        CascadeType.MERGE, CascadeType.REMOVE }) //设置成一对多的关系。
    public Set<OrderItem> getItems() {
        return items;
    }

    public void setItems(Set<OrderItem> items) {
        this.items = items;
    }
}
```



OrderItem.java

```
package cn.itcast.bean;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class OrderItem {
    private Integer id;
    private String productName;
    private Float sellPrice = 0f; //默认值为0。
    private Order order;

    @Id
```

  长方式生成主键。

```
@GeneratedValue //id
public Integer getId() {
    return id;
}

public void setId(Integer id) {
    this.id = id;
}

@Column(length = 40, nullable = false)
public String getProductName() {
    return productName;
}

public void setProductName(String productName) {
    this.productName = productName;
}

@Column(nullable = false)
public Float getSellPrice() {
    return sellPrice;
}

public void setSellPrice(Float sellPrice) {
    this.sellPrice = sellPrice;
}

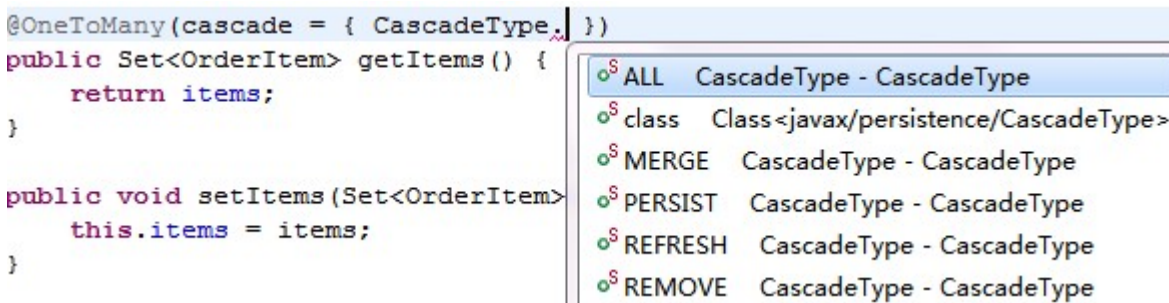
public Order getOrder() {
    return order;
}

public void setOrder(Order order) {
    this.order = order;
}
}
```

在JPA里面，一对多关系(1-n)：

多的一方为关系维护端，关系维护端负责外键记录的更新（如果是条字段就负责字段的更新，如果是多对多关系中的中间表就负责中间表记录的更新），关系被维护端是没有任何权力更新外键记录（外键字段）的。

CascadeType的选项有，看图：



CascadeType.REFRESH：级联刷新，也就是说，当你刚开始获取到了这条记录，那么在你处理业务过程中，这条记录被另一个业务程序修改了（数据库这条记录被修改了），那么你获取的这条数据就不是最新的数据，那你就需要调用实体管理器里面的refresh方法来刷新实体，所谓刷新，大家一定要记住方向，它是获取数据，相当于执行select语句的（但不能用select，select方法返回的是EntityManager缓存中的数据，不是数据库里面最新的数据），也就是重新获取数据。

CascadeType.PERSIST：级联持久化，也就是级联保存。保存order的时候也保存orderItem，如果在数据库里已经存在与需要保存的orderItem相同的id记录，则级联保存出错。

CascadeType.MERGE：级联更新，也可以叫级联合并；当对象Order处于游离状态时，对对象Order里面的属性作修改，也修改了Order里面的orderItems,当要更新对象Order时，是否也要把对orderItems的修改同步到数据库呢？这就是由CascadeType.MERGE来决定的，如果设了这个值，那么Order处于游离状态时，会先update order,然后for循环update orderItem,如果没设CascadeType.MERGE这个值，就不会出现for循环update orderItem语句。

所以说，级联更新是控制对Order的更新是否会波及到orderItems对象。也就是说对Order进行update操作的时候，orderItems是否也要做update操作呢？完全是由CascadeType.MERGE控制的。

CascadeType.REMOVE：当对Order进行删除操作的时候，是否也要对orderItems对象进行级联删除操作呢？是的则写，不是的则不写。

如果在应用中，要同时使用这四项的话，可以改成cascade = CascadeType.ALL

应用场合问题：这四种级联操作，并不是对所有的操作都起作用，只有当我们调用实体管理器的persist方法的时候，CascadeType.PERSIST才会起作用；同样道理，只有当我们调用实体管理器的merge方法的时候，CascadeType.MERGE才会起作用,其他方法不起作用。 同样道理，只有当我们调用实体管理器的remove方

法的时候，`CascadeType.REMOVE`才会起作用。

注意：`Query query = em.createQuery("delete from Person o where o.id=?1");`这种删除会不会起作用呢？是不会起作用的，因为配置里那四项都是针对实体管理器的对应的方法。

1.11 11、JPA中的一对多延迟加载与关系维护（一对多关系：二）

发表时间: 2010-07-13

order.java

```
package cn.itcast.bean;

import java.util.HashSet;
import java.util.Set;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;

@Entity
@Table(name="orders") //把表名改成orders（默认表名是order），防止默认表名order与数据库的关键字"order by"中的order冲突。不改的话测试不成功，出现异常，orderitem表建立成功，order表建不了。
public class Order {
    private String orderId;
    private Float amount = 0f;
    private Set<OrderItem> items = new HashSet<OrderItem>();

    @Id //要注意：目前JPA规范并没有提供UUID这种生成策略，目前主键值只提供了整型的生成方式，所以@GeneratedValue这个注解就不能在这里用上，不能对字符串进行id自增长。
    @Column(length = 12)
    public String getOrderId() {
        return orderId;
    }

    public void setOrderId(String orderId) {
        this.orderId = orderId;
    }
}
```

```
@Column(nullable = false)
public Float getAmount() {
    return amount;
}
```

```
public void setAmount(Float amount) {
    this.amount = amount;
}
```

```
@OneToMany(cascade = { CascadeType.REFRESH, CascadeType.PERSIST,
    CascadeType.MERGE, CascadeType.REMOVE }, fetch=FetchType.LAZY, mappedBy="order")
//mappedBy="order", 中的order是关系维护端的order属性, 这个order属性的类型是这个bean。
```

```
public Set<OrderItem> getItems() {
    return items;
}
/*
```

@OneToMany(fetch=FetchType.)的选项有, 如下图:

FetchType.EAGER:代表立即加载;

FetchType.LAZY:代表延迟加载。

当我们把fetch设置为FetchType.LAZY的时候, 什么时候初始化items里面的数据呢? 当我们第一次访问这个属性, 并对这个属性进行操作的时候, 这个集合的数据才会从数据库里面load出来。但要注意: 当我们访问这个延迟属性的时候, 我们的前提要EntityManager这个对象没有被关闭, 如果被关闭了我们再去访问延迟属性的话, 就访问不到, 并抛出延迟加载意外。

如果没有设置fetch这属性的话, 会怎么样呢? 是立即加载? 还是延迟加载呢?

记住@OneToMany这个标签最后的英文单词, 如果是要得到Many的一方, 我不管你前面是什么, 只要后面的单词是Many, 也就是说要得到多的一方, 你们就给我记住, 默认的加载策略就是延迟加载(Many记录可能上几万条, 立即加载的话可能对效率影响大, 所以延迟加载)。

反过来, 如果后面是One呢? 因为它是加载一的一方, 这对性能影响不是很大, 所以它的默认加载策略是立即加载。mappedBy: 我们怎么知道关系的维护端和被维护端呢? 当然JPA规范规定多的一端应该为是维护端(关系维护端增加一个字段为外键, 里面保存的是一端的主键), 一端为关系被维护端, 那么我们总要在程序里给他们打上标志吧? 虽然规范是这么规定, 但总要申明一下吧? 就是通过mappedBy属性, 只要哪个类出现了mappedBy, 那么这个类就是关系的被维护端。里面的值指定的是关系维护端。

orderItem这边由哪一个属性去维护关系呢? 是OrderItem类的order属性。

```
mappedBy属性对应Hibernate里面的inverse属性: <SET name="items" inverse="true"></SET>
*/
```

```
public void setItems(Set<OrderItem> items) {
    this.items = items;
}
```

//用这个方法会方便很多

```
public void addOrderItem(OrderItem orderItem){
```


orderItem.setOrder(this); //关系维护方orderItem加入关系被维护方(this)后,才能维护更新关系(orderItem表中的外键字段order_id),维护关系其实就是更新外键。只有为关系维护端设置了关系被维护端,关系才能建立起来。

```
        this.items.add(orderItem);  
    }  
}
```

fetch=FetchType.)

- EAGER FetchType - FetchType
- LAZY FetchType - FetchType
- class Class<javax/persistence/FetchType>

OrderItem.java

```
package cn.itcast.bean;  
  
import javax.persistence.CascadeType;  
import javax.persistence.Column;  
import javax.persistence.Entity;  
import javax.persistence.GeneratedValue;  
import javax.persistence.Id;  
import javax.persistence.JoinColumn;  
import javax.persistence.ManyToOne;  
  
@Entity  
public class OrderItem {  
    private Integer id;  
    private String productName;  
    private Float sellPrice = 0f; //默认值为0。  
    private Order order;  
  
    @Id  
    @GeneratedValue //id自增长方式生成主键。  
    public Integer getId() {  
        return id;  
    }  
}
```

```
}

public void setId(Integer id) {
    this.id = id;
}

@Column(length = 40, nullable = false)
public String getProductName() {
    return productName;
}

public void setProductName(String productName) {
    this.productName = productName;
}

@Column(nullable = false)
public Float getSellPrice() {
    return sellPrice;
}

public void setSellPrice(Float sellPrice) {
    this.sellPrice = sellPrice;
}

@ManyToOne(cascade={CascadeType.MERGE,CascadeType.REFRESH},optional=false)
@JoinColumn(name="order_id")    //设置外键的名称。
```

```
public Order getOrder() {    //OrderItem是关系维护端，负责关系更新，它是根据它的order属性值维护关系的。当它保存的时候（主动保存或是被级联保存），他会根据order属性的值更新关系，
    return order;
}
```

当order为null时，就不会更新关系了。级联操作也是根据双方对象中的映射属性值进行的，当映射属性没值的时候就不会对对方进行级联操作了。

```
/*
```

@ManyToOne的级联保存（CascadeType.PERSIST）是不需要的，不可能说你保存某个订单项OrderItem的时候，也保存订单Order的。通常都是保存订单Order的时候，保存订单项OrderItem的。

CascadeType.MERGE：如果我们更新了订单项orderItem产品的价钱，那么整个订单Order的总金额是会发生改变的，所以可以定义这个级联更新。

CascadeType.REFRESH：如果我们想得到目前数据库里orderItem最新的数据的话，我们也希望得到订单order的最新数据，我们可以定义这个级联刷新，就是说把数据库里最新的数据重新得到。

CascadeType.REMOVE：这个属性这里肯定不设。就好比现在有一个订单，一个订单里面有3个购物项orderI

```
*/
```

//optional：说明order这个是否是可选的？是否可以没有的？false表示必须的，true表示是可选的。

```
    public void setOrder(Order order) {  
        this.order = order;  
    }  
}
```

OneToManyTest.java

```
package junit.test;  
  
import javax.persistence.EntityManager;  
import javax.persistence.EntityManagerFactory;  
import javax.persistence.Persistence;  
  
import org.junit.BeforeClass;  
import org.junit.Test;  
  
import cn.itcast.bean.Order;  
import cn.itcast.bean.OrderItem;  
  
public class OneToManyTest {  
  
    @BeforeClass  
    public static void setUpBeforeClass() throws Exception {  
    }  
  
    @Test  
    public void save() {  
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("itcast")  
        EntityManager em = factory.createEntityManager();  
        em.getTransaction().begin();  
  
        Order order = new Order();  
        order.setAmount(34f);
```

order.setOrderId("992"); //orderId是数据库里面的主键，同时也是实体的标识。这里并没有使用Id自增长的方式来生成主键值，而是自己设值，所以可以随便写。如果想用UUID，可以这样写UUID.randomUUID().toString();这个类JDK5提供了。

```
        OrderItem orderItem1 = new OrderItem();
        orderItem1.setProductName("足球");
        orderItem1.setSellPrice(90f);
        OrderItem orderItem2=new OrderItem();
        orderItem2.setProductName("瑜伽球");
        orderItem2.setSellPrice(30f);
        order.addOrderItem(orderItem1);
        order.addOrderItem(orderItem2);

        em.persist(order);
        em.getTransaction().commit();
        em.close();
        factory.close();
    }
}
```

在junit test里面，可以通过下面代码来生成数据库表，根据数据库表来确定元数据是否定义成功：

```
@Test
public void save() {
    EntityManagerFactory factory = Persistence.createEntityManagerFactory("itcast");
    factory.close();
}
```

运行junit测试，发现保存订单Order的时候，也保存了订单项OrderItem.为什么呢？是因为订单Order和订单项OrderItem定义了级联保存（CascadeType.PERSIST）关系，这个级联关系在我们调用em.persist(order);的persist方法时就会起作用。

1.12 12、JPA中的一对一双向关联

发表时间: 2010-07-13

IDCard.java

```
package cn.itcast.bean;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToOne;

@Entity
public class IDCard {
    private Integer id;
    private String cardNo;
    private Person person;

    public IDCard() {
    }

    public IDCard(String cardNo) {
        this.cardNo = cardNo;
    }

    @Id
    @GeneratedValue
    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }
}
```

```
}

@Column(length = 18,nullable=false)
public String getCardNo() {
    return cardNo;
}

public void setCardNo(String cardNo) {
    this.cardNo = cardNo;
}

@OneToOne(mappedBy = "idCard", cascade = { CascadeType.PERSIST,
        CascadeType.MERGE, CascadeType.REFRESH } /*,optional = false*/)
public Person getPerson() {
    return person;
}
/*
```

mappedBy：如何把IDCard指定为关系被维护端？就是通过这属性。使用了这属性的类，就是关系被维护端。

cascade：

CascadeType.REMOVE：删除身份证，需要把这个人干掉吗？不用，所以这个属性不设。

CascadeType.PERSIST：一出生就有身份证号。

CascadeType.MERGE：在游离状态的时候，修改了身份证号码，需要对人员的信息进行修改么？如果有这种业务需求，就设上去。

CascadeType.REFRESH：重新获取idCard的数据的时候，需不需要获取person的数据呢？

这些级联的定义，一定是根据你们的业务需求来定的。用不用是根据你的业务来决定的，业务需要就用，业务不需要就不用。

optional：是否可选，是否允许为null？反映在业务上，就是有身份证，是否一定要有这个人呢？

因为在Person里已经指定了idCard是必须要存在的，外键由person表维护，那么这里这个属性就是可选的。设不设置这个person属性都行。那么在这里option这属性就可以不再进行设置了，不设置不对我们的数据构成任何影响，person可有可无不对关系（外键）存在影响。

外键由Person的option属性决定，就算你设置了这属性，其实它也不看你这属性。在设外键字段是否允许为空的时候，也不看这属性，而是看关系维护端的设定。

fetch：加载行为默认为立刻记载，凭one。

*/

```
public void setPerson(Person person) {
    this.person = person;
}

}
```

```
}
```

Person.java

```
package cn.itcast.bean;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;

@Entity
public class Person {
    private Integer id;
    private String name;
    private IDCard idCard;

    public Person() {
    }

    @Id
    @GeneratedValue
    // 采用数据库Id自增长方式来生成主键值。
    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    @Column(length = 10, nullable = false)
    public String getName() {
        return name;
    }
}
```

```
    public void setName(String name) {
        this.name = name;
    }

    @OneToOne(optional = false, cascade = CascadeType.ALL)
    @JoinColumn(name = "idCard_id")
    public IDCard getIdCard() {
        return idCard;
    }

    public void setIdCard(IDCard idCard) {
        this.idCard = idCard;
    }

    public Person(String name) {
        this.name = name;
    }
}
```

OneToOneTest.java

```
package junit.test;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import org.junit.BeforeClass;
import org.junit.Test;

import cn.itcast.bean.IDCard;
import cn.itcast.bean.Person;
```



```
public class OneToOneTest {

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
    }

    @Test
    public void save() {
        EntityManagerFactory factory = Persistence
            .createEntityManagerFactory("itcast");
        EntityManager em = factory.createEntityManager();
        em.getTransaction().begin();

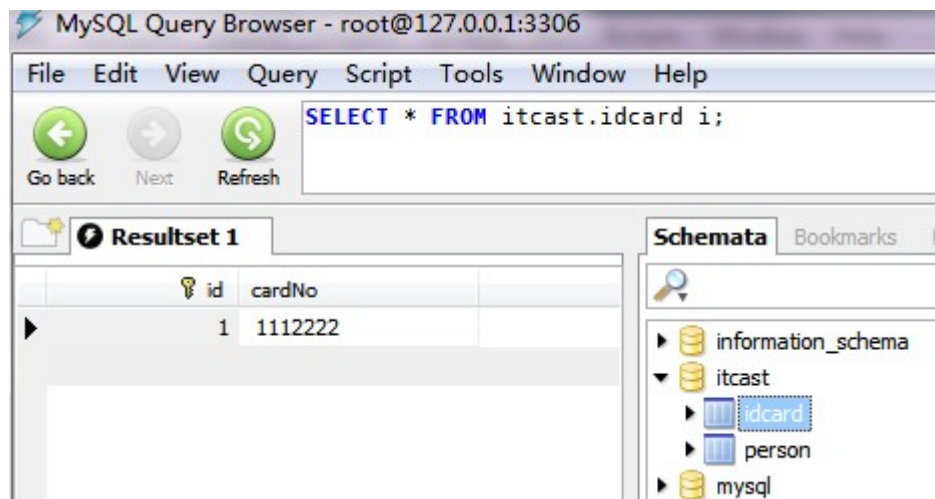
        Person person = new Person("老张");        // person是关系维护端。
        person.setIdCard(new IDCard("1112222"));    // 通过person把idCard放进去，这关系就由person来维护了。
        em.persist(person);    // 先保存idCard,得到保存记录的id,用id作为外键的值,再保存person。因为person表里的外键值是idcard表里面的主键，只有先生成主键值才有外键值。

        em.getTransaction().commit();
        em.close();
        factory.close();
    }
}
```

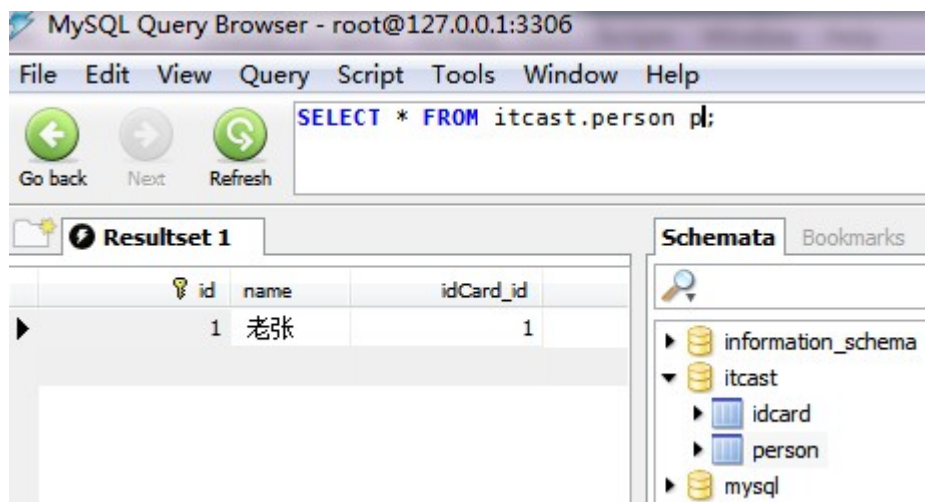
谁是关系维护端，谁就负责外键字段的更新。

Person是关系维护端，IDCard是关系被维护端，怎么维护更新呢？往Person里面设置idCard,这样就相当于把关系建立起来了；如果通过IDCard设置person的话，那么这种关系是建立不起来的，因为IDCard是关系被维护端

idcard表结构，看图：



person表结构，看图：



1.13 13、JPA中的多对多双向关联实体定义与注解设置

发表时间: 2010-07-13

Student.java

```
package cn.itcast.bean;

import java.util.HashSet;
import java.util.Set;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;

@Entity
public class Student {
    private Integer id;
    private String name;
    private Set<Teacher> teachers = new HashSet<Teacher>();

    @Id      @GeneratedValue //id作为实体标识符，并且采用数据库id自增长的方式生成主键值。
    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    @Column(length = 10, nullable = false)
```

```
public String getName() {  
    return name;  
}  
  
public void setName(String name) {  
    this.name = name;  
}  
  
@ManyToMany(cascade = CascadeType.REFRESH)  
@JoinTable(name = "student_teacher", inverseJoinColumns = @JoinColumn(name = "teacher_id"), joinColumns = @JoinColumn(name = "student_id"))  
public Set<Teacher> getTeachers() {  
    return teachers;  
}  
/*  
    假如不对关联表里的字段做任何设定，那么表里面的字段默认由JPA的实现产品来帮我们自动生成。  
    inverseJoinColumns：inverse中文是反转的意思，但是觉得太恶心了，在JPA里，可以理解为被维护端。  
    inverseJoinColumns：被维护端外键的定义。  
    @JoinColumn：外键，设置中间表跟teacher表的主键关联的那个外键的名称。  
    joinColumns：关系维护端的定义。  
*/  
  
public void setTeachers(Set<Teacher> teachers) {  
    this.teachers = teachers;  
}  
}
```

Teacher.java

```
package cn.itcast.bean;  
  
import java.util.HashSet;  
import java.util.Set;  
  
import javax.persistence.CascadeType;
```

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToMany;

@Entity
public class Teacher {
    private Integer id;
    private String name;
    private Set<Student> students=new HashSet<Student>();

    @Id      @GeneratedValue //id作为实体标识符，并且采用数据库id自增长的方式生成主键值。
    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    @Column(length = 10, nullable = false)
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @ManyToMany(cascade=CascadeType.REFRESH,mappedBy="teachers")
    //mappedBy="teachers",表示关系由Student对象维护。"teachers"与Student对象的teachers属性对应
    public Set<Student> getStudents() {
        return students;
    }
    /*
    cascade:
        CascadeType.PERSIST:级联保存不要，学生没来之前，老师就已经在了。
        CascadeType.MERGE：级联更新不要，把学生的信息改了，没必要修改相应的老师的信息，压根就没这业务需求。
```

CascadeType.REMOVE：级联删除更不要，如果双方都设了级联删除，加入删除学生，会删除相应的老师，被删除的老师又跟学生发生千丝万缕的关系，又把一批学生删掉.....没完没了...最终的结果可能是数据里面所有的记录都被清掉。所以在多对多关系中，级联删除通常是用不上的。

这里只需设置级联刷新CascadeType.PERSIST就可以了，事实上refresh方法也很少使用。

mappedBy：通过这个属性来说明老师是关系被维护端。

fetch：加载行为默认是延迟加载（懒加载），凭Many。这里不需要设置。

```
*/  
  
    public void setStudents(Set<Student> students) {  
        this.students = students;  
    }  
  
}
```

ManyToManyTest.java

```
package junit.test;  
  
  
import javax.persistence.EntityManager;  
import javax.persistence.EntityManagerFactory;  
import javax.persistence.Persistence;  
  
import org.junit.BeforeClass;  
import org.junit.Test;  
  
public class ManyToManyTest {  
  
    @BeforeClass  
    public static void setUpBeforeClass() throws Exception {  
    }  
  
    @Test  
    public void save() {  
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("itcast")  
        factory.close();  
    }  
  
}
```

```
}
```

双向多对多关系是一种对等关系，既然是对等关系，也就是说我们可以人为决定谁是关系维护端，谁是关系被维护端，这里选学生做关系维护端。那么以后就只能通过学生来维护老师跟学生的关系。

假设：

学生A id是1

老师B id是1

那通过什么东西把他们的关系确立起来呢？采用什么来存放他们的关联关系呢？是中间表（关联表）。

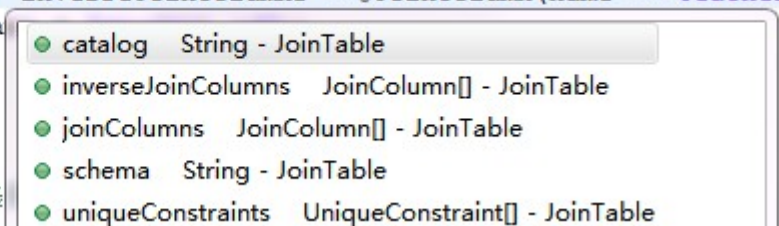
学生A和老师B建立起关系，首先要找到关系维护端，是学生，就要通过学生这个关系维护端，学生A.getTeachers().add(Teacher); 这样就能把老师跟学生的关系确立起来了。确立起来后，反应在中间表里面就是insert into...一条语句

如果学生A要把老师B开掉，那就要解除关系，也是通过关系维护端学生A，反映在面向对象的操作就是 学生A.getTeachers().remove(Teacher); 执行这句代码的时候，在底层JDBC它会对中间表做delete from...语句的操作。

我们都是通过关系维护端来进行操作的，以后在双向关系中一定要找准谁是关系维护端，谁是关系被维护端

@JoinTable的注解有：看图，

```
@JoinTable(name = "student_teacher", inverseJoinColumns = @JoinColumn(name = "teacher_id"),
    joinColumns = @JoinColumn(name = "student_id"))
public Set<Teacher> getTeachers() {
    return teachers;
}
/*
假如不对关联表里的字段做任何设定，那么表
```

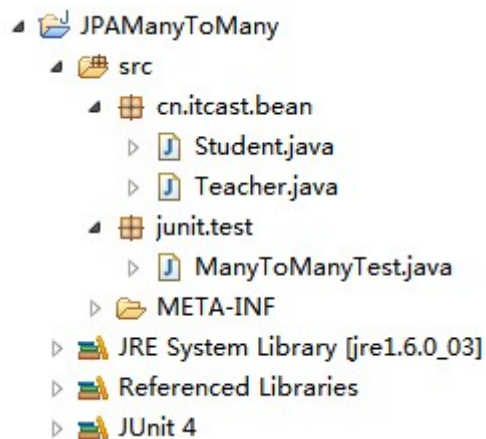


- catalog String - JoinTable
- inverseJoinColumns JoinColumn[] - JoinTable
- joinColumns JoinColumn[] - JoinTable
- schema String - JoinTable
- uniqueConstraints UniqueConstraint[] - JoinTable

1.14 14、JPA中的多对多双向关联的各项关系操作

发表时间: 2010-07-13

目录结构，看图：



Student.java

```
package cn.itcast.bean;

import java.util.HashSet;
import java.util.Set;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;

@Entity
public class Student {
    private Integer id;
    private String name;
    private Set<Teacher> teachers = new HashSet<Teacher>();
}
```



```
public Student() {  
}  
  
public Student(String name) {  
    this.name = name;  
}  
  
@Id  
@GeneratedValue  
// id作为实体标识符，并且采用数据库的id自增长方式生成主键值。  
public Integer getId() {  
    return id;  
}  
  
public void setId(Integer id) {  
    this.id = id;  
}  
  
@Column(length = 10, nullable = false)  
public String getName() {  
    return name;  
}  
  
public void setName(String name) {  
    this.name = name;  
}  
  
@ManyToMany(cascade = CascadeType.REFRESH)  
@JoinTable(name = "student_teacher", inverseJoinColumns = @JoinColumn(name = "teacher_id"), joinColumns = @JoinColumn(name = "student_id"))  
public Set<Teacher> getTeachers() {  
    return teachers;  
}  
  
/*
```

假如不对关联表里的字段做任何设定，那么表里面的字段默认由JPA的实现产品来帮我们自动生成。

inverseJoinColumns：inverse中文是反转的意思，但是觉得太恶心了，在JPA里，可以理解为被维护端

```
inverseJoinColumn：被维护端外键的定义。  
@JoinColumn：外键名称（中间表跟teacher表的主键关联的那个外键名称）。  
joinColumns：关系维护端的定义。  
*/  
  
public void setTeachers(Set<Teacher> teachers) {  
    this.teachers = teachers;  
}  
  
public void addTeacher(Teacher teacher) {  
    this.teachers.add(teacher);  
}  
  
public void removeTeacher(Teacher teacher) {  
    if(this.teachers.contains(teacher)){ //凭什么判断teacher在集合teachers中呢？是根据teacher的id。这就要求必要重写Teacher.java的hashCode和equals方法，通过这两个方法来判断对象是否相等。  
        this.teachers.remove(teacher);  
    }  
}  
}
```

Teacher.java

```
package cn.itcast.bean;  
  
import java.util.HashSet;  
import java.util.Set;  
  
import javax.persistence.CascadeType;  
import javax.persistence.Column;  
import javax.persistence.Entity;  
import javax.persistence.GeneratedValue;  
import javax.persistence.Id;  
import javax.persistence.ManyToMany;
```

```
@Entity
public class Teacher {
    private Integer id;
    private String name;
    private Set<Student> students = new HashSet<Student>();

    public Teacher() {
    }

    public Teacher(String name) {
        this.name = name;
    }

    @Id
    @GeneratedValue
    // id作为实体标识符，并且采用数据库的id自增长方式生成主键值。
    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    @Column(length = 10, nullable = false)
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @ManyToMany(cascade = CascadeType.REFRESH, mappedBy = "teachers")
    public Set<Student> getStudents() {
        return students;
    }
}
```

```
/*
```

cascade: CascadeType.PERSIST:级联保存不要，学生没来之前，老师就已经在了。

CascadeType.MERGE：级联更新不要，把学生的信息改了，没必要修改相应的老师的信息，压根就没这业

CascadeType.REMOVE：级联删除更不要，如果双方都设了级联删除，加入删除学生，会删除相应的老师，被删除的老师又跟学生发生千丝万缕的关系，又把一批学生删掉.....没完没了...最终的结果可能是数据里面所有的记录都被清掉。

所以在多对多关系中，级联删除通常是用不上的 这里只需设置级联刷新CascadeType.PERSIST就可以了，事实上refresh方法也很少使用。

mappedBy：通过这个属性来说明老师是关系被维护端 fetch：加载行为默认是延迟加载（懒加载），凭Many。这里不需要设置。

```
*/
```

```
public void setStudents(Set<Student> students) {  
    this.students = students;  
}
```

```
@Override
```

```
public int hashCode() {  
    final int prime = 31;  
    int result = 1;  
    result = prime * result + ((id == null) ? 0 : id.hashCode());  
    //判断的依据是，如果id不为null的话，就返回id的哈希码。  
    return result;  
}
```

```
@Override
```

```
public boolean equals(Object obj) {  
    if (this == obj)  
        return true;  
    if (obj == null)  
        return false;  
    if (getClass() != obj.getClass())  
        return false;  
    final Teacher other = (Teacher) obj;  
    if (id == null) {  
        if (other.id != null)  
            return false;  
    } else if (!id.equals(other.id))  
        return false;  
    return true;  
}
```

```
    }  
}
```

ManyToManyTest.java

```
package junit.test;  
  
import javax.persistence.EntityManager;  
import javax.persistence.EntityManagerFactory;  
import javax.persistence.Persistence;  
  
import org.junit.BeforeClass;  
import org.junit.Test;  
  
import cn.itcast.bean.Student;  
import cn.itcast.bean.Teacher;  
  
public class ManyToManyTest {  
  
    @BeforeClass  
    public static void setUpBeforeClass() throws Exception {  
    }  
  
    @Test  
    public void save() {  
        EntityManagerFactory factory = Persistence  
            .createEntityManagerFactory("itcast");  
        EntityManager em = factory.createEntityManager();  
        em.getTransaction().begin();  
  
        em.persist(new Student("小张同学"));  
        em.persist(new Teacher("李勇老师"));  
    }  
}
```

```
        em.getTransaction().commit();
        em.close();
        factory.close();
    }

    /**
     * 建立学生跟老师的关系
     */
    @Test
    public void buildTS() {
        EntityManagerFactory factory = Persistence
            .createEntityManagerFactory("itcast");
        EntityManager em = factory.createEntityManager();
        em.getTransaction().begin();
```

Student student = em.find(Student.class, 1); // 首先要得到学生，因为学生是关系维护端，通过关系维护端来建立关系。

student.addTeacher(em.getReference(Teacher.class, 1)); //这方法在业务意义上，就代表建立跟老师的关系。

//所谓建立跟老师的关系，无非就是把老师加进集合里面去。

//建立关系，体现在JDBC上面，就是添加一条记录进中间表。

```
        em.getTransaction().commit();
        em.close();
        factory.close();
    }

    /**
     * 解除学生跟老师的关系
     */
    @Test
    public void deleteTS() {
        EntityManagerFactory factory = Persistence
            .createEntityManagerFactory("itcast");
        EntityManager em = factory.createEntityManager();
        em.getTransaction().begin();
```

Student student = em.find(Student.class, 1); // 首先要得到学生，因为学生是关系维护端，通过关系维护端来建立关系。

student.removeTeacher(em.getReference(Teacher.class, 1)); //这方法在业务意义上，就代表解除跟老师的关系。

//所谓解除跟老师的关系，无非就是把老师从集合里面删去。
//解除关系，体现在JDBC上面，就是在中间表删除一条记录。

```
em.getTransaction().commit();
em.close();
factory.close();
}

/*
 * 删除老师,老师已经跟学生建立起了关系（错误写法）
 */
@Test
public void deleteTeacher1() {
    EntityManagerFactory factory = Persistence
        .createEntityManagerFactory("itcast");
    EntityManager em = factory.createEntityManager();
    em.getTransaction().begin();
```

```
em.remove(em.getReference(Teacher.class, 1));
```

//并不需要发生数据装载行，只需要一个托管状态的实体，所以用getReference可以提供性能。

```
em.getTransaction().commit();
em.close();
factory.close();
}
```

/*

该方法会出错，因为中间表中已经有记录了，会抛出以下错误：

Caused by: java.sql.BatchUpdateException:

```
Cannot delete or update a parent row: a foreign key constraint fails
('itcast/student_teacher', CONSTRAINT 'FKD4E389DE1D49449D' FOREIGN KEY ('teacher_
REFERENCES 'teacher' ('id'))
```

关系被维护端没有权力更新外键，所以不会删除中间表的记录。

*/

/*

* 删除老师,老师已经跟学生建立起了关系（正确写法）

```
*/
@Test
public void deleteTeacher2() {
    EntityManagerFactory factory = Persistence
        .createEntityManagerFactory("itcast");
    EntityManager em = factory.createEntityManager();
    em.getTransaction().begin();

    Student student = em.find(Student.class, 1);
    Teacher teacher = em.getReference(Teacher.class, 1);
    //并不需要发生数据装载行为，只需要一个托管状态的实体，所以用getReference可以提供性能。
    student.removeTeacher(teacher);
    //student是关系维护端，有权利删除外键，只要在对象中删除了teacher，那么中间表中相关外键记录也就被删除了。
    //想要删除teacher记录，必须先通过student解除关系才行。
    em.remove(teacher);

    em.getTransaction().commit();
    em.close();
    factory.close();
}

/*
 * 删除学生,老师已经跟学生建立起了关系
 */
@Test
public void deleteStudent() {
    EntityManagerFactory factory = Persistence
        .createEntityManagerFactory("itcast");
    EntityManager em = factory.createEntityManager();
    em.getTransaction().begin();

    Student student = em.getReference(Student.class, 1);
    em.remove(student);    //这样是可以删除学生的，尽管目前是有关系，中间表有关联记录，

    em.getTransaction().commit();
    em.close();
    factory.close();
}
```



```
        }  
    }  
}
```

1.15 15、JPA中的联合主键

发表时间: 2010-07-13

两个或多个字段组成的主键，我们叫联合主键。在面向对象中，我们用JPA怎么定义这种情况呢？

怎么定义联合主键？用面向对象的思想来思考的话，联合主键里的复合主键（字段），可以把它看成一个整体，然后采用一个主键类来描述这个复合主键的字段。

关于联合主键类，大家一定要遵守以下几点JPA规范：

1. 必须提供一个public的无参数构造函数。
2. 必须实现序列化接口。
3. 必须重写hashCode()和equals()这两个方法。这两个方法应该采用复合主键的字段作为判断这个对象是否相等的。
4. 联合主键类的类名结尾一般要加上PK两个字母代表一个主键类，不是要求而是一种命名风格。

ArtLinePK.java

```
package cn.itcast.bean;

import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Embeddable;

@Embeddable
//这个注解代表ArtLinePK这个类是用在实体里面，告诉JPA的实现产品：在实体类里面只是使用这个类定义的属性。
//简单的理解为：ArtLinePK里的属性可以看成是ArtLine类里的属性，好比ArtLinePK的属性就是在ArtLine里定义的
public class ArtLinePK implements Serializable{
    private String startCity;
    private String endCity;

    public ArtLinePK() {
    }
}
```

```
public ArtLinePK(String startCity, String endCity) {
    this.startCity = startCity;
    this.endCity = endCity;
}

@Column(length=3)
public String getStartCity() {
    return startCity;
}

public void setStartCity(String startCity) {
    this.startCity = startCity;
}

@Column(length=3)
public String getEndCity() {
    return endCity;
}

public void setEndCity(String endCity) {
    this.endCity = endCity;
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((endCity == null) ? 0 : endCity.hashCode());
    result = prime * result
        + ((startCity == null) ? 0 : startCity.hashCode());
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
```

```
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    final ArtLinePK other = (ArtLinePK) obj;
    if (endCity == null) {
        if (other.endCity != null)
            return false;
    } else if (!endCity.equals(other.endCity))
        return false;
    if (startCity == null) {
        if (other.startCity != null)
            return false;
    } else if (!startCity.equals(other.startCity))
        return false;
    return true;
}

}
```

这个联合主键类，应该作为实体类的一个主键（实体标识符，id）。

ArtLine.java

```
package cn.itcast.bean;

import javax.persistence.Column;
import javax.persistence.EmbeddedId;
import javax.persistence.Entity;

@Entity
public class ArtLine {
    private ArtLinePK id;    //用面向对象的思想去思考的话，这个复合主键看成一个整体，由复合主键类ArtLinePK来描述。
```

```
private String name;

public ArtLine() {
}

public ArtLine(ArtLinePK id) {
    this.id = id;
}

public ArtLine(String startCity, String endCity, String name) {
    this.id = new ArtLinePK(startCity, endCity);
    this.name = name;
}
```

`@EmbeddedId` //按照JPA规范要求，我们并不是用`@Id`来标注它。

//`@EmbeddedId` 这个注解用于标注`id`这个属性为实体的标识符，因为我们使用的是复合主键类，所以我们要用`@EmbeddedId`这个专门针对复合主键类的标志实体标识符的注解。

```
public ArtLinePK getId() {
    return id;
}

public void setId(ArtLinePK id) {
    this.id = id;
}

@Column(length=20)
public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}
}
```

我们的复合主键类，目前为止，已经定义好了。定义好了之后，接着我们就看它生成的数据库表是否满足复合主键这么一种情况。

ArtLineTest.java

```
package junit.test;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import org.junit.BeforeClass;
import org.junit.Test;

import cn.itcast.bean.ArtLine;

public class ArtLineTest {

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
    }

    @Test public void save(){
        EntityManagerFactory factory = Persistence
            .createEntityManagerFactory("itcast");
        EntityManager em = factory.createEntityManager();
        em.getTransaction().begin();

        em.persist(new ArtLine("PEK","SHA","北京飞上海"));

        em.getTransaction().commit();
        em.close();
        factory.close();
    }
}
```

字段生成了，看图：

Table Name: Database: Comment:

Columns and Indices | Table Options | Advanced Options

Column Name	Datatype	NOT NULL	AUTO INC	Flags	Default Value	Comment
endCity	VARCHAR(3)	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/> BINARY		
startCity	VARCHAR(3)	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/> BINARY		
name	VARCHAR(20)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/> BINARY	<input type="text" value="NULL"/>	

看主键，两个字段，看图：符合主键的定义就OK:

Indices | Foreign Keys | Column Details

PRIMARY

Index Settings

Index Name:

Index Kind:

Index Type:

Index Columns (Use Drag'n'Drop)

endCity
startCity

数据也添加进去了，看图：

File Edit View Query Script Tools Window

Go back Next Refresh

`SELECT * FROM artline a;`

Resultset 1

endCity	startCity	name
▶ SHA	PEK	北京飞上海