

第十三章 开发 JPA 应用

第十三章 开发JPA应用	1
13.1 介绍	2
13.1.1 JPA 简介	2
13.1.2 MyEclipse提供的JPA开发功能	4
13.1.3 JPA的代码结构和相关理论知识	4
13.1.3.1 JPA代码结构	4
13.1.3.2 配置文件persistence.xml	4
13.1.3.3 实体类及标注	6
13.1.3.4 使用EntityManager来管理实体	13
13.1.3.5 Query对象	18
13.1.3.6 JPA 查询语言（JPA QL）简介	19
13.1.3.7 回调方法	21
13.2 准备工作	22
13.3 创建JPAHello项目	22
13.3.1 创建表格	22
13.3.2 创建 HelloJPA Java Project	23
13.3.3 添加 JPA Capabilities 到现有项目	23
13.3.4 使用JPA配置文件编辑器修改文件	25
13.3.5 使用反向工程快速生成JPA实体类和DAO	28
13.3.6 调整生成的实体类标注	42
13.3.7 编写测试代码	43
13.4 JPA 工具高级部分	46
13.4.1 MyEclipse Java Persistence Perspective透视图	46
13.4.2 JPA Details 视图	47
13.4.3 JPA 代码编辑辅助	49
13.4.4 生成一对多等复杂映射代码	49
13.5 Spring整合JPA开发	58
13.5.1 添加Spring开发功能	59
13.5.2 从数据库反向工程生成实体和Spring DAO	60
13.5.3 编写并运行测试代码	71
13.5.3.1 支持标注事务时的调试代码	71
13.5.3.2 不支持标注事务时的调试代码	73
13.6 小结	74
13.7 参考资料	75

在本章我们将会介绍 JPA 的开发，一共包括三部分内容：独立运行的 JPA 应用开发；Spring 整合 JPA 开发；EJB 查询语言开发。其实关于 JPA 还有一部分是基于 EJB 容器环境的开发，那一部分内容我们将会放到后面的 EJB 开发一章来介绍。由于 MyEclipse 6 对 JPA 开发提供了很方便的支持，因此我们的内容主要就集中在如何使用 MyEclipse 进行快速开发这个话题上。

13.1 介绍

13.1.1 JPA 简介

前面我们已经介绍了 Hibernate，但是实际在开发中所存在的这种进行数据库开发的 ORM 框架不止一种，包括 JDO，iBatis，TopLink，KODO，OpenJPA 等等多种开源的和商业的产品。那么这有什么问题呢？假设现在我们的项目是用 Hibernate 开发的，运行于 Oracle 数据库之上，然而上线运行一段时间后，发现有一些性能上的问题，而这时候我们想找人来做技术支持，希望它来帮我们解决这些问题，因为开发人员并不是个个都能读懂 Hibernate 的源码然后找到问题所在的。这是第一种可能：客户（通常是有钱的大客户）希望能在遇到问题时有人提供商业的技术支持和顾问服务。第二种可能：Oracle 公司推出了专用于对 Oracle 数据库特别优化过的 ORM 产品，名为 TopLink，然而很不幸，虽然它能解决我们的问题，但是，因为 Hibernate 的类库的包结构和 TopLink 的相差太远，一个是以 org.hibernate 开头的，另一个却是以 com.oracle 开头的，更要命的是，两者之间的类库根本就没有相似之处！那我们的项目并不能通过简单的将代码的包换一下，就能迁移成功，下载岂不是要所有涉及 Hibernate 的地方都得重写！换句话说，当我们使用 Hibernate 或者其它 ORM 框架开发时，我们的代码已经被绑死了！虽然代码是和数据库无关的，然而数据库访问的 Java 代码却是死死的和某种框架绑定在一起了。

针对这种情况，即 Java 的 ORM 框架再次出现诸侯割据，烽烟混战的局面，而 Java 程序员却被迫只能选择一种框架，而无法在项目开发之后变更框架的情况，Sun 公司——Java 和 Java EE 规范的领导者，就像在当年 JDBC 规范出现之前所作的那样，在 Java EE 5 的规范制定中，特地提出了 JPA(Java Persistence API)这一新的 ORM 规范，意图结束这种混乱的局面，并革命性的使用标注作为开发模式，很大程度上减少了开发人员的负担：包括背诵包和类的负担以及辛辛苦苦的编写 XML 配置文件来进行实体映射的负担，并使用 POJO 的方式进行开发。和之前的 JDBC 一样，在充分吸收现有 ORM 框架的基础上，得到了一个易于使用、伸缩性强的 ORM 规范，它不仅仅提供基础的映射功能，还使这些 JPA 的框架提供者能够提供高级功能。Sun 引入新的 JPA ORM 规范出于两个原因：其一，简化现有 Java EE 和 Java SE 应用的对象持久化的开发工作；其二，Sun 希望整合对 ORM 技术，实现天下归一。

JPA 由 EJB 3.0 软件专家组开发（Hibernate 的发明者 Gavin King 老师也加入其中），作为 JSR-220 实现的一部分。但它不局限于 EJB 3.0，你可以在 Web 应用、甚至桌面应用中使用。JPA 的宗旨是为 POJO 提供持久化标准规范，由此可见，经过这几年的实践探索，能够脱离容器独立运行，方便开发和测试的理念已经深入人心了。目前 Hibernate 3（确切说是配合 Hibernate Entity Manager）、TopLink、KODO 以及 OpenJPA 都提供了 JPA 的实现，甚至还包括一些以前的 JDO 产品。而截至目前，所有支持 Java EE 5 的服务器（需要通过 Sun 的认证才行），都支持 JPA 规范，这些服务器包括：Apache Geronimo-2.1（开源服务器），WebLogic Server v10.0（商业，BEA 公司产品，现并入 Oracle），IBM WASCE

2.0(商业服务器), Apusic Application Server (v5.0) (国产, 金蝶中间件, 商业), Oracle Application Server 11 (商业), SAP NetWeaver 7.1 (商业), Sun Java System Application Server Platform Edition 9 (商业, 单免费使用), TmaxSoft JEUS 6 (不详), GlassFish Application Server (开源软件)。在这里提一下 JBoss, 最新的 4 和 5 都支持 JPA, 然而, 它却没通过 Java EE 5 的认证, 它的 JPA 是基于 Hibernate 实现的, 因为 Hibernate 的作者现在加入了 JBoss 公司 (确切的说是 RedHat 公司, 它收购了 JBoss)。

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 JDK 5.0 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;
- 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。
- 查询语言, 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。

最后, 需要说明的是, JPA 只是一个规范, 就好像规定所有的手机都必须能打电话一样, 但是, JPA 本身不是电话, 我们能说手机提供了通话功能, 但是我们不能说通话功能就是一个手机。很显然手机提供了通话功能之外的其它很多附加的能力, 同样的, JPA 的实现者也可以根据自己的情况, 提供基本功能之外的附加功能。其系统结构图如图 13.1 所示。这样, 其实 JPA 一共分成了两部分, 一部分就是 JPA 规范, 它是一套编程的接口 (API), 供开发人员来进行调用; 另一部分就是 JPA 规范的实现者, 一般情况下他们在现有 ORM 框架的基础上, 再编写一套基于 JPA 的实现类, 当然这时候他们也会根据实际情况进行一些增强的功能。我们程序员, 则处于 JPA 规范之上, 一般来说不需要特别的熟悉底层 ORM 的详细信息, 当然如果能有所了解就更好了。ORM 框架则生成 SQL 语句, 调用 JDBC, 通过 JDBC 再去操作最终的数据库。

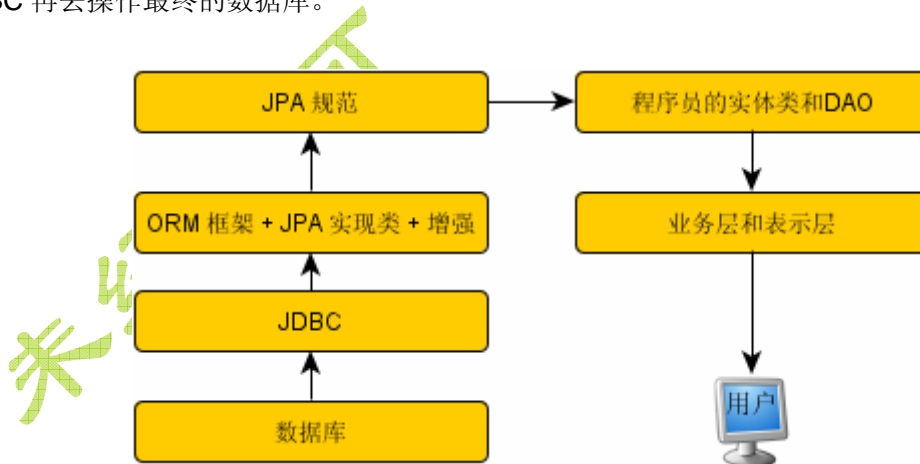


图 13.1 JPA 的系统结构图

目前支持 JPA 的开发工具 (IDE) 是比较多的, 包括 MyEclipse 6, Eclipse IDE for Java EE Developers (最新的 3.3) 版本, Netbeans 5.5/6, BEA Workshop 10, Oracle JDeveloper 等等, 其中不乏一些免费的选择。其中支持最标准的当属免费的 Netbeans 6 了, 不光能生成 JPA 代码, 还能附带生成增删改查的 JSF 页面, 以及支持最新的 Swing 桌面应用框架, 支持 Swing 应用中直接通过 JPA 修改底层数据。

13.1.2 MyEclipse 提供的 JPA 开发功能

MyEclipse 6 对 JPA 的开发提供了全面的支持，包括：添加、修改 JPA 的实现类库的功能；从数据库反向工程直接生成实体类和 DAO 的功能；增强的 JPA 实体信息编辑器；实体标注编写自动提示功能；实体信息验证；内置 Spring + JPA 整合向导和 Spring DAO 代码生成功能等等。

13.1.3 JPA 的代码结构和相关理论知识

注意：本节内容读者可以放在做完练习之后再来阅读，或者是碰到不熟悉的写法时再来阅读，会比较好一些。因为实际情况来说，每个框架都有成千上万的知识点，而对于开发人员来说，一般掌握最常用的部分即可。当然也可以先阅读，后练习。本节内容相对多而且有些，因为 JPA 的应用越来越广泛，而 MyEclipse 6 实际上是能够生成绝大多数的代码（甚至包括一对多等映射关系），所以我们提供这些内容主要是处于帮助大家理解并修改代码的目的。而 JPA 还支持纯 XML 配置非标注的开发方式，因为比较少用我们就先不做介绍了。

13.1.3.1 JPA 代码结构

由于 Hibernate 自动生成的代码封装的比较厉害，可能不太便于初学者理解。由图 13.1 我们可以看到，一个 JPA 的项目离不开途中所说的几个部分。首先当然是要有数据库服务器了，其次，需要 JDBC 的驱动程序，接着，是 JPA 的实现类和 JPA 本身的包，最后，我们才能开发 JPA 的应用。

其源代码目录结构一般如下：

META-INF/persistence.xml

Entity1.java

EntityDAO.java

开发过程一般是：JPA 配置文件声明持久化单元 ➡ 编写带标注的实体类 ➡ 编写 DAO 类。

13.1.3.2 配置文件 persistence.xml

首先第一个，就是 JPA 的配置文件 *persistence.xml*，一般位于目录 **META-INF** 下，注意大小写不能写错了。这个文件的结构如图 13.2 所示。

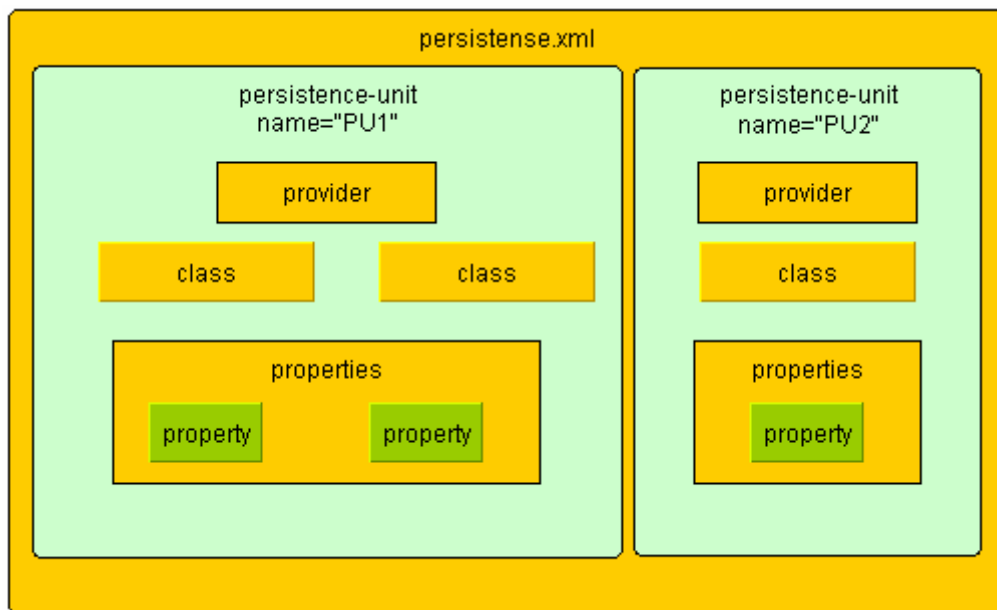


图 13.2 persistence.xml 的结构图

我们列出一份示例的代码清单，如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">

  <persistence-unit name="JPADemoPU"
    transaction-type="RESOURCE_LOCAL">
    <provider>
      oracle.toplink.essentials.PersistenceProvider
    </provider>
    <class>jpadao.Myuser</class>
    <properties>
      <property name="toplink.jdbc.driver"
        value="com.mysql.jdbc.Driver" />
      <property name="toplink.jdbc.url"
        value="jdbc:mysql://localhost:3306/test?useUnicode=true&characterEncoding=
        GBK" />
      <property name="toplink.jdbc.user" value="root" />
    </properties>
  </persistence-unit>

</persistence>
```

这个文件的内容包括一个或者多个持久化单元（persistence unit）的定义，这是和

Hibernate 不同的，每个持久化单元可以连接到各自的数据库上去。每个持久化单元都有自己的名字（`name` 属性）和事务类型的定义（`transaction-type` 属性）。名字用来访问数据的时候根据其取值来创建持久化单元。事务类型则分为 `RESOURCE_LOCAL`（本地事务，适用于独立运行的或者没有 EJB 容器的环境）和 `JTA`（Java Transaction API, Java 事务 API，一般用于带有 EJB 容器的环境，或者说运行于服务器中）。如上面代码所示，这份文件中定义了一个名为 `JPADemoPU` 的持久化单元。

在每个持久化单元的定义里面，还需要定义 `provider`，它指定了 JPA 的底层实现类。前面已经介绍过了，JPA 核心是一套接口（规范，本身无功能，只是规定应该有什么样的行为），它一定要有底层的实现类才可以。这个提供者，就是类似于 Hibernate `EntityManager`，KODO，TopLink 这样的框架中所提供的核心实现类了。JPA 的优点就在于，如果我们想切换到别的框架上，只需要将这个提供者改掉，相关的属性（见后面 `properties`）加以修改即可，别的所有的 Java 代码都不需要重新变动即可工作。

再接着的内容，就是 `class` 定义，它定义了有哪些类是实体类，可以用来做持久化管理。在例子中取值为 `jpadao.Myuser`，它是一个普通的 `JavaBean`。因为 JPA 中的实体类就是普通的加了些标注的 `JavaBean`，因此在独立运行的情况下，这样的符合要求的 `JavaBean` 有成千上亿个之多，如果一个个的去检查这些类是不是加了标注，那将是十分消耗时间的。所以，在普通模式下，我们通过多个 `class` 标签来列出哪些是需要被作为实体类来处理的，这样系统的复杂度和启动速度都会大大加快。这就好像是要你找出本市的所有女人一样（咱们假设不算流动人口），最直接的办法就是把所有女人集中到一个广场里，数一数有多少个是女人；然而还有一个办法，是通过查国家安全部门的身份证，因为每个人都是有标注的（身份证信息），可以表明它是男还是女，这样，就很快可以精确的找到北京有多少个人了，很显然，这些信息是要保存在一个地方的，就像是在这个配置文件中通过 `class` 标签来声明有多少个类是实体类一样。

最后面出现的内容是 `properties`（复数的属性）标签，它的下面是对应的定义了一个或者多个的 `property`（单个属性）子标签。这些属性的名称和取值，一般都是和特定的 JPA 实现相关的。不过，一般来说这些属性包括了和数据库相关联的信息，包括驱动，URL，用户名和密码等等。另外，还可以多设置一些额外的信息例如和特定的 ORM 实现相关的扩展，例如 Hibernate 可以设置一个属性来打印出每次操作所使用的 SQL 语句。

关于这个文件还有一些别的配置信息，例如 `jta-data-source` 标记，是用于服务器环境下的数据源访问的，它指定实体 Bean 使用的数据源 JNDI 名称，将在 EJB 一节进行讨论，在这里我们暂时还不多做介绍。

13.1.3.3 实体类及标注

开发的第二步，就是编写实体类。和 Hibernate 中的实体类一样，它一般是和数据库中的某个表相对应的。不过在 JPA 中，实体类可以加入标注，因此它还担负着如何让实体类中的属性和数据库中的表和字段相对应的责任。简言之就是 JPA 实体类就是一个普通的带有属性的 `JavaBean` 外加实体标注。在图 11.3 中，左侧列出了 MySQL 数据库中的 `test` 数据库中的表 `user` 及其表的三个字段，而在右侧则列出了实体类及所标注的各个信息和表的对应关系。

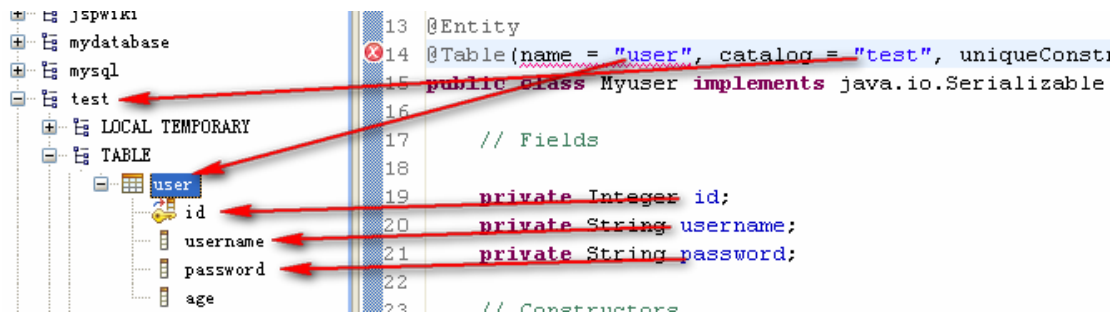


图 13.3 实体类标注和数据库表格的对应关系

现在我们来列出一个简单的实体类代码，并加以分析和介绍：

```
package users;

import javax.persistence.*;

@Entity(name="User")
@Table(name = "myuser", catalog = "test", uniqueConstraints = {})
public class Myuser implements java.io.Serializable {

    // 变量定义
    private Integer id;
    private String username;
    private String password;

    public Myuser() {
    }

    // Property accessors
    @Id
    @Column(name = "id", unique = true, nullable = false, insertable =
true, updatable = true)
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    public Integer getId() {
        return this.id;
    }

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

    @Column(name = "username", unique = false, nullable = false,
insertable = true, updatable = true, length = 200)
    public String getUsername() {
        return this.username;
    }
}
```

```

    }

    public void setUsername(String username) {
        this.username = username;
    }

    //省略更多属性定义
}

```

首先可能部分读者对 Java 5 的标注还不是特别的熟悉。比较正式的文档都会这样介绍：**Annotation** 提供一种机制，将程序中元素（如类、方法、属性等）和元数据联系起来。这样编译器可以将元数据保存的 **class** 文件中。代码分析工具就可以使用这些元数据执行的额外任务。注释采用“**at**”标记形式（**@**），后面是注释名称。那么标注呢，其实就是一个特殊的只是用来附加一些信息用的类，在英文中它的写法是 **Annotation**，本意就是注释的意思。关于这个词还有其它的一些翻译方式，例如叫**元数据**，**注解**，**注释**等等，都是指代的同一个东西。不管怎么说，它其实就相当于一个超市里商品的标签，虽然大家都知道店面上摆着一排苹果，从外观来看大概这些苹果也相同，但是定价和产地等信息呢，一般则是卖家提供的，所以他们会在上面贴一个标签，表明定价，产地等信息。再比如说，大家买一台电脑，店家肯定不是只告诉你一个总价钱了事，这样你也不会答应啊，所以，这台电脑的各个部件都会有说明信息，店家会给你一个列表，标明：CPU Intel 双核，主频 2G，内存大小 2G 等等类似的信息。这和 Java 中的标注也是类似的，它可以标注到类型的声明上，或者是构造器，变量和方法上。要用一个标注，首先当然是需要创建一个标注类了，选择菜单 **File > New > Annotation**，输入标注名，就可以创建标注了。下面列出了我们所创建的一个名为 **TestAnnotation** 的标注的代码清单，并列出了它的示例用法：

```

import java.lang.annotation.*;

@Documented// 是否产生JavaDoc的时候列出此标注信息
@Target({ElementType.TYPE, ElementType.CONSTRUCTOR,
ElementType.METHOD, ElementType.FIELD})
//指定目标，可以标到类型，构造器，方法和变量上
@Retention(RetentionPolicy.RUNTIME)//设置保持性
@Inherited

public @interface TestAnnotation {
    String value() default "test1";//这里定义了一个属性 value(通过方法名来定义，不是变量声明)，并指定了默认值test1
    String name() default "";
}

@TestAnnotation(name="Sample1", value="Value1")
// 标注的使用简单示例
class SampleUse {
    @TestAnnotation(value = "id_1")
    private int id;
}

```



```

@TestAnnotation("id_2")//当只有一个名为value的属性的时候,可以省略属性名
value
public int getID() {
    return id;
}
}

```

。它使用 **@interface** 这样的关键字来定义的。好了,这些标注到底有什么用?Java 的发明者为什么要发明这个玩意呢?希望通过刚才的例子,读者已经可以简单理解到标注可以自己带一些值附加到当前类上,那么很好,它的好处就在这里:给类多一些附加的信息。这些信息可以通过反射 (reflection) 的 API 读取出来,然后进行更进一步的处理。必须澄清一个概念:依赖注入和代码写标注以及容器解析标注是三码事,要实现标注注入缺一不可。把自己写的类加个写个 **@Entity** 或者 **@EJB**,不放入对应的容器里执行,那就还是普通的类而已。所有的 Annotation 本身什么也不能做,要做注入需要通过容器来获取对应的实例然后解析上面的标注后做对应的操作。换句话说你自己 new 一个带标注的 Bean,是什么效果也没有的,只能通过容器来初始化,用类似于 **getBean** 的方式,容器才能有机会进行注入操作。

JPA 通过定义一些不同的标注类型,将原来的配置文件信息从 XML 配置文件中转移到了 Java 类中。那么有什么好处呢?有一些资料已经列出了它的一些好处:

类似于 Hibernate 的传统模式的 ORM 框架大都是采用 xml 作为配置文件,但采用文本的 xml 配置存在一些缺陷:

- 描述符多,不容易记忆和掌握
- 无法做自动的校验,需要人工排错
- 当系统变大时,大量的 xml 配置难以管理
- 读取和解析 xml 配置非常耗时,导致应用启动缓慢,不利于测试和维护
- 做 O/R Mapping 的时候需要在 java 文件和 xml 配置文件之间交替,增大了工作量
- 运行中保存 xml 配置需要消耗额外的内存

采用标注可以很好的解决这些问题:

1. 描述符大量减少。以往在 xml 配置中往往需要描述 java 属性的类型,关系等等。而标注本身就是 java 语言,从而省略了大量的描述符
2. 编译期校验。错误的批注在编译期间就会报错。
3. 标注批注在 java 代码中,避免了额外的文件维护工作
4. 标注被编译成 java bytecode,消耗小的多内存,读取也非常迅速,往往比 xml 配置解析快几个数据量级,利于测试和维护

好了,这就是它的优点,最大的好处恐怕不在于 3,而在于 1 和 4 勉强算一个,因为 3 和 4 都牵扯到改动信息的时候,需要找到类的源代码才可以,而且必须通过编译这一步,所以有时候,例如成百上千的标注需要修改的时候,优势就不是那么的明显了。

注意:由于 JPA 依赖标注进行工作,所以 JAP 应用和标注代码都必须运行在 Java 5(JDK 1.5)或者更高版本上!

好了,现在我们的话题已经跑的足够远了,需要回过头来再看看刚才的实体类定义了。先来看看类的标注。

@Entity: 将 JavaBean 标注为一个实体,表示需要保存到数据库中,默认情况下类名即为表名,通过 **name** 属性可以显式指定实体名。在这个地方某些资料介绍有误,实体名并不等于表名,只能说默认情况下实体名和表名以及类名这三者是一致的。然而,实体名最

主要的用途是为了进行查询时（通过 JPA QL）区分不同的实体，例如下面的查询代码：

`select model from User model` 和 `select model from Myuser model`，将会最终去寻找不同的实体定义。而例子中的实体名则是 `User`。实际使用中可以不用写 `name` 属性，只写一个 `@Entity` 的标注，这时候实体名等于类名（不含包路径，例如 `entity.MyUser` 最后对应的实体名是 `MyUser`）。和 Hibernate 中一样，这个实体名必须是**唯一**的！而此标注也是**必须**的！

@Table: 顾名思义，定义当前 Entity 对应数据库中的表，需要注意的是主表（primary table），因为其它的表还可以通过 `@SecondaryTable` 或者 `@SecondaryTables` 来进行标注。在例子中它对应的数据库中的表是 `myuser`（用 `name` 属性定义），所在的数据库名为 `test`（用 `catalog` 属性定义）。实践中此标记及其附属的属性都可以省略不写，这时候对应的数据库表就是对应着实体的名字。这个标注可以不用定义，不是必须的。

@Id: 定义了实体的主键信息。在一个实体类中只有一个主键标注，而此标注也是**必须**的！它本身没有任何额外的属性可以设置。默认情况下它对应数据库中的类型和名字和当前属性或者实体的变量相同的列（这是因为属性可以标注在属性和变量上，后文会有进一步的介绍）。如果下面有 `@Column` 标记，那么对应的数据库列信息以此标记中的内容为准。

@GeneratedValue: 一般它和 ID 的标注配合使用，用来制定主键的生产策略。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` 这个枚举类中定义了以下几种可供选择的策略：

- 1) **IDENTITY:** 表自增键字段，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（不幸的是大部分时候都会选择成 `Sequence`）；
- 3) **SEQUENCE:** 通过序列产生主键，还可以进一步通过 `@SequenceGenerator` 标注来指定更详细的生产方式，`MySQL` 不支持这种方式（注：`@SequenceGenerator` 用法示例：

```
@GeneratedValue(strategy=GenerationType.SEQUENCE)
@SequenceGenerator(name="SEQ_TEST", // 此生成器的别名
sequenceName="User_SEQ", // 对应的 Oracle sequence 的名字
allocationSize=25)
```

);

- 4) **TABLE:** 通过表产生主键，框架借由表模拟序列（sequence）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 厂商所默认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`。

另外还有一个重要的属性就是 **generator**，它可以更进一步制定主键生成器所采用的参数，例如 `sequence` 表的表名可以这样定义：`@GeneratedValue(strategy=SEQUENCE, generator="CUST_SEQ")`。同样的也可以用于 `TABLE` 方式的生成器上：`@GeneratedValue(strategy=TABLE, generator="CUST_GEN")`。

@Column: 属性或者变量对应的表字段。一般来说我们并不需要指定表字段的类型，因为 JPA 会根据反射从实体属性中获取类型；如果是字符串类型，我们可以指定字段长度，以便可以自动生成 DDL 语句；还可以指定一些其它的属性例如是否唯一（unique），是否可以为空（nullable），是否可以插入和更新等等（insertable 和 updatable）；`name` 属性指定了它对应于数据库中的列名。如果是时间类型，一般还需要指定精度，用 `@Temporal` 来进行标注。同样的这个标注也不是必须的，如果没有写任何参数的话，它的名字和类型都和当前标注的属性或者变量的名字和类型相同。

@Temporal: 如果属性是时间类型，因为数据表对时间类型有更严格的划分，所以必须指定具体时间类型。在 `javax.persistence.TemporalType` 枚举中定义了 3 种时间类型：

- 1) `DATE` : 等于 `java.sql.Date`;
- 2) `TIME` : 等于 `java.sql.Time`;
- 3) `TIMESTAMP` : 等于 `java.sql.Timestamp`。

用法示例：

```
@Column(name = "BIRTHDAY")
@Temporal(TemporalType.DATE)
```

。同样的这个标注不是必须的，但是它没有默认值，所以必须指定一个取值。

常用的这些标注基本上已经介绍完毕，那么一个最简单的实体定义代码示例如下：

```
@Entity
public class Myuser implements java.io.Serializable {

    @Id
    private Integer id;
    ....
}
```

。在这种情况下，所有的取值都采用默认值，表名和实体名相对应，而变量名及其类型则和数据库表的列相对应。所以说标注方式大大简化了 JPA 的开发。

为了不使读者疑惑，我们还必须介绍属性标注的两种位置：标注在属性(property)和标注在变量(field)上。到底采用那种位置，取决于 `@Id` 标注出现的位置。而且一旦采用了一种标注方式后，就不能再混合使用了。例如：

```
@Id
@Column
private Integer id;
```

这种方式就是 `field` 方式，那么所有其它的列定义都应该标在变量上。

而：

```
@Id
@Column
Public Integer getId() { return ...}
```

这样的方式就是属性方式。默认情况下，JPA 采用的是属性定义的方式，也就是根据方法来确定有哪些属性。

@Transient: 实体 bean 中所有的非 `static` 非 `transient` 的属性都可以被持久化，除非你将其注解为 `@Transient`。所有没有定义注解的属性等价于在其上面添加了 `@Basic` 注解。虽然默认情况一般都工作的很好，然而实际开发中有一种情况可能会出现：在实体类中提供了一个算总帐的方法，但是这个总帐属性却没有和数据库的任何列对应，这时候你可以通过 `@Transient` 标注使它不会保存到数据库中，类似的也可以这样来不保存其它的属性到数据库中。例如：

```
@Transient
Public double getTotalCost() {...}
```

。

其它可以标在属性上的标注还有：

@Lob: 表示属性将被持久化为 `Blob` 或者 `Clob` 类型(大数据类型，例如超过 10MB 的数据)，具体取决于属性的类型。`java.sql.Clob`, `Character[]`, `char[]` 和 `java.lang.String` 这些类型的属性都被持久化为 `Clob` 类型；而 `java.sql.Blob`, `Byte[]`, `byte[]` 和 `Serializable` 类型则被

持久化为 Blob 类型。用法示例：

```
@Lob
String mailText;
```

@Basic: 刚刚已经提到了 @Basic，它一般可以来控制是否进行延迟加载。用法示例：

```
@Basic(fetch=FetchType.LAZY) //采用延迟加载，FetchType.EAGER 一般不采用，它是非延迟加载模式
```

关于实体类还有一个话题没有谈，那就是实体类不能像普通的 JavaBean 那样随心所欲的进行编写，它有一些这样的限制：

- Entity 类必须要有一个无参数的 public 或者 protected 的 Constructor。
- 如果在应用中需要将该 Entity 类分离出来在分布式环境中作为参数传递，该 Entity 类需要实现 java.io.Serializable 接口。
- Entity 类不可以是 final，也不可有 final 的方法。
- abstract 类和 Concrete 实体类都可以作为 Entity 类。
- Entity 类中的属性变量不可以是 public。Entity 类的属性必须通过 getter/setter 或者其他的方法获得。

最后，实体类还支持继承，一对一，一对多和多对多，以及命名查询（NamedQuery，在 Netbeans 开发工具生成 JPA 实体的时候经常可以看到）等标注。实际上 MyEclipse 能够自动生成这样的代码（Netbeans 开发工具也可以生成），所以在这里我们略作介绍：

一对一映射：双向一对一关系需要在关系维护端（owner side）的 one2one Annotation 定义 mappedBy 属性。建表时在关系被维护端（inverse side）建立外键列指向关系维护端的主键列。

```
@OneToOne(optional = true, cascade = CascadeType.ALL, mappedBy = "country")
private Capital capital;
```

一对多映射：双向一对多关系，一是关系维护端（owner side），多是关系被维护端（inverse side）。建表时在关系被维护端建立外键列指向关系维护端的主键列。

```
@OneToMany(targetEntity = Child.class, cascade = CascadeType.ALL, mappedBy = "father")
```

```
public List<Child> getChildren() { return children; }
```

多对多映射：

多对多映射采取中间表连接的映射策略，建立的中间表将分别引入两边的主键作为外键。（比较少用，建议不要用，性能问题严重）

下面则给大家列出一个 NamedQuery 的代码片段：

```
@Entity
@Table(name = "address")
@NamedQueries( {
    @NamedQuery(name = "Address.findByAddressID", query = "SELECT a FROM Address a WHERE a.addressID = :addressID"),
    @NamedQuery(name = "Address.findByCity", query = "SELECT a FROM Address a WHERE a.city = :city"),
    @NamedQuery(name = "Address.findByStreet", query = "SELECT a FROM
```

```

Address a WHERE a.street = :street"),
    @NamedQuery(name = "Address.findByZip", query = "SELECT a FROM
Address a WHERE a.zip = :zip")
})
public class Address implements Serializable {

```

13.1.3.4 使用 EntityManager 来管理实体

这样，我们关于实体类的标注的介绍暂告一段落，接下来我们说以下如何编写 DAO，换句话说就是如何进行增删改查这样的持久化操作。之所以介绍是因为 MyEclipse 生成的代码中也包含了这些由基本内容一一组合起来的代码，读者阅读后可以对它有深入的了解，便于自己修改代码。要操作实体类，就必须接触下面所说的几个类：*Persistence*，*EntityManagerFactory* 和 *EntityManager*。它们的相互关系如图 13.4 所示。

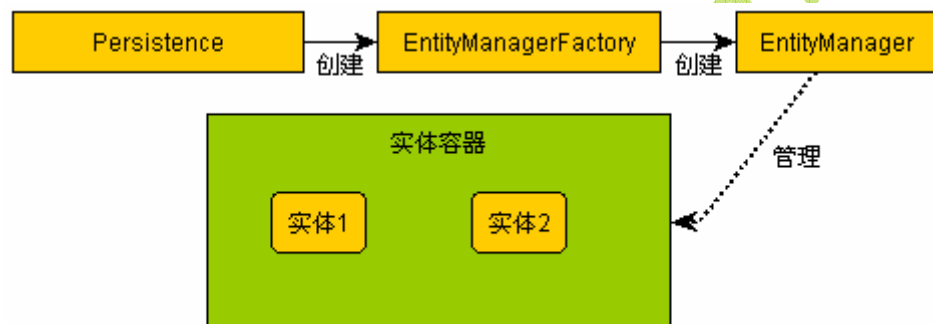


图 13.4 实体管理器的创建经过

```

G Persistence
  SF PERSISTENCE_PROVIDER : String
  SF providers : Set
  SF nonCommentPattern : Pattern
  C Persistence()
  S createEntityManagerFactory(String) : EntityManagerFactory
  S createEntityManagerFactory(String, Map) : EntityManagerFactory
  S findAllProviders() : void
  S providerNamesFromReader(BufferedReader) : List

I EntityManagerFactory
  S createEntityManager() : EntityManager
  S createEntityManager(Map) : EntityManager
  S close() : void
  S isOpen() : boolean

```

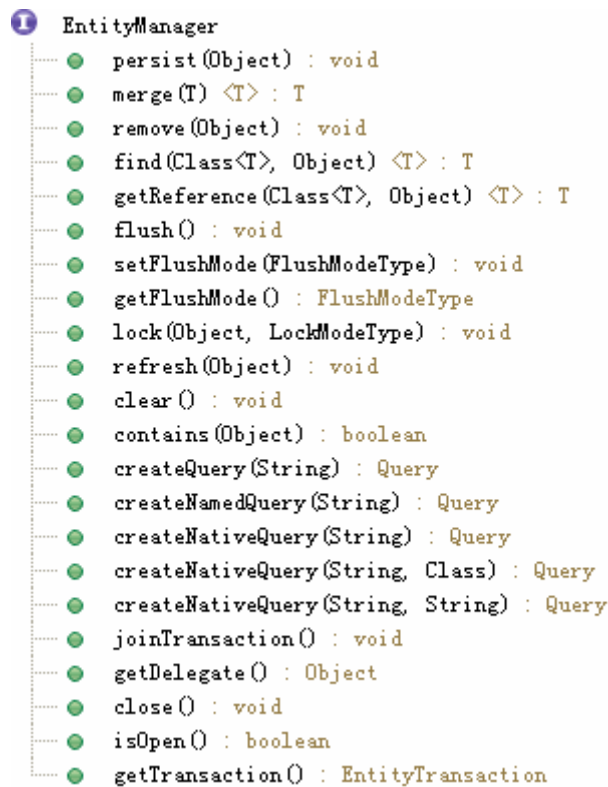



图 13.5 JPA 实体工厂类图

那么这几个类之间有何关系呢？在独立运行的模式下，EJB3 定义了一个 `javax.persistence.Persistence` 类用于启动 EJB3 运行环境。只有 `EntityManager` 才是最后和实体打交道的对象，包括保存，更新，删除和查询等等操作，而要获得 `EntityManager`，首先需要通过 `Persistence` 类调用其 `createEntityManagerFactory()` 方法获得 `EntityManagerFactory`，然后调用 `EntityManagerFactory` 的 `createEntityManager()` 方法获得。我们可以认为，`Persistence` 相当于 JDBC 的驱动类，`EntityManagerFactory` 则相当于连接工厂，而 `EntityManager` 则最后和特定的数据库建立连接并互相操作。下面是一段示例代码：

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("JPADemoPU");
EntityManager entityManager = emf.createEntityManager();
```

当调用 `Persistence.createEntityManagerFactory()` 的时候，`Persistence` 会是否存在 `META-INF/persistence.xml` 配置文件，然后根据此文件再来初始化 `EntityManagerFactory`。从图 13.5 中可以看到 `Persistence` 和 `EntityManagerFactory` 都是相当简单的类，它们之间的关系就是一个创建与被创建的工厂模式（就跟拖拉机厂造拖拉机一样）。`Persistence` 类的 `createEntityManagerFactory()` 方法有两个参数：必选的持久化单元名称（`String` 类型）和可选的额外属性设置（`Map` 类型）。同样的 `EntityManagerFactory` 中的 `createEntityManager()` 方法也可以带一个可选的额外属性设置参数（`Map` 类型）。

另外在容器环境下（一般是位于 EJB 服务器中，后面的 EJB 章节中会加以介绍），创建 `EntityManagerFactory` 或者 `EntityManager` 可以不用通过 `Persistence` 来进行，而使用标注即可，一般其代码如下所示：

```
@PersistenceContext
EntityManager em;

@PersistenceUnit
```

EntityManagerFactory emf;

。前面已经讲过，标注仅仅是一个标签而已，一定要有容器来处理这个标签，这些值才能被设置进去，之后就可以直接使用变量来进行操作了。

在介绍 **EntityManager** 之前，我们需要了解以下实体的生命周期这个概念。我们知道实体像人一样，具有出生，学习，服务，死亡等状态，对实体来说，它们的状态规定为：

1. 新实体(new)
2. 持久化实体(managed)
3. 分离的实体(detached)
4. 删除的实体(removed)

而通过 **EntityManager**，可以在它们的不同状态之间实现切换。我们可以简单的通过下表来确定一个实体的状态：

状态名	作为 Java 对象存在	在实体管理器中存在	在数据库中存在
new	√	×	×
managed	√	√	√
detached	×	×	√
removed	√	√	×

而在图 13.6 中则列出了状态改变和 **EntityManager** 之间的关系。图中的 em 表示实体管理器 (**EntityManager**)，而 tx 则表示 **EntityTransaction**，实体事务管理器。和 **Hibernate** 一样，对实体类的某些操作必须在事务中进行才能成功，这也是本章后面的内容介绍为什么要使用 **Spring** 来整合 **JPA** 的一个重要原因。

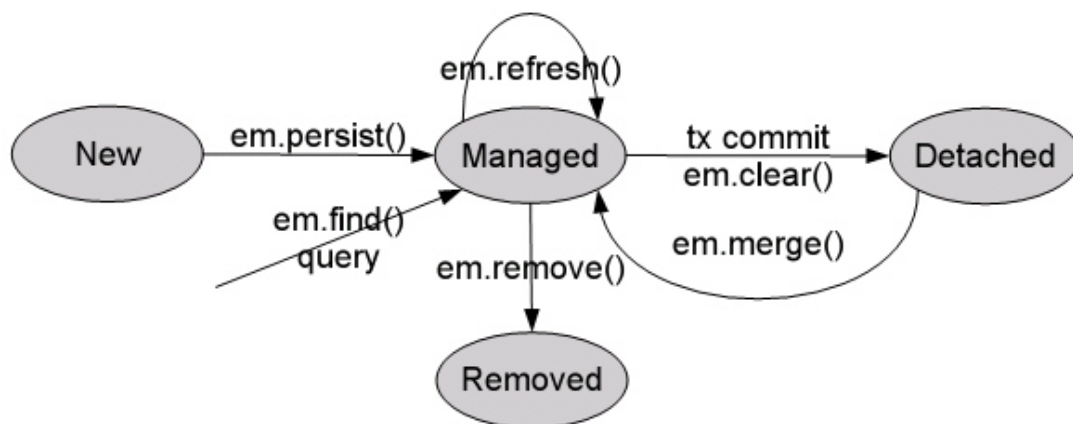


图 13.6 实体状态转换关系图

有了 **EntityManager** 之后，我们就可以像在 **Hibernate** 的 **Session** 对象中所做的那样，对实体进行各种各样的操作了，包括增删改查，刷新，根据主键查找对象等等。图 13.5 列出了 **EntityManager** 所具有的一些方法，而这些方法的大部分都是和 **Hibernate** 中的 **Session** 对象很相似的，毕竟所有的 **ORM** 都是实现增删改查嘛。我们介绍一些主要的方法：

- **persist(Object)** – 使实体类从 new 状态或者 removed 转变到 managed 状态，并将数据保存到底层数据库中。
- **remove(Object)** – 将实体变为 removed 状态，当实体管理器关闭或者刷新时，会真正的删除数据
- **find(Class entityClass, Object key)** – 以主键查询实体对象，entityClass 是实体的类，key 是主键值
- **flush()** – 将实体和底层的数据库进行同步，当我们调用 **persist()**，**merge()** 或

remove() 这些方法时，更新并不会立刻同步到数据库中，直到容器决定刷新到数据库中时才会执行，我们可以调用 flush() 强制更新

- createQuery() – 根据 JPA QL 定义查询对象
- createNativeQuery() – 允许开发人员根据特定数据库的 SQL 语法来进行查询操作，只有 JPA QL 不能满足要求时才使用它
- createNamedQuery() – 根据实体中标注的命名查询创建查询对象，大家还记得前面所提到的 @NamedQuery 标注嘛？彼处的 name 属性值则对应着此方法的参数。
- merge (Object) – 将一个 detached 的实体持久化到数据库中，并转换为 managed 状态
- close() – 关闭实体管理器，并且会尝试更新所有数据

下面是一些示例的代码：

```
// 保存数据
HelloEntityBean hello = new HelloEntityBean( 1, "foo" );
EntityTransaction trans = entityManager.getTransaction();
trans.begin();

// 持久化 hello,在此操作之前 hello 的状态为 new
entityManager.persist( hello );

// 这时 hello 的状态变为 managed

trans.commit();

entityManager.close();

// 这时 hellow 的状态变为 detached.
// 获取 Entity
如果知道 Entity 的唯一标示符，我们可以用 find() 方法来获得 Entity。

Father father = manager.find( Father.class, new Integer( 1 ) );

// 由于 JDK1.5 支持自动转型，也可以如下使用

Father father = manager.find( Father.class, 1 );

/*
 * 或者,可以用 Entity 名字作为查找。但无法利用 JDK 1.5 的自动转型功能,
 * 需要使用对象作为查找主键，并需要对获得 Entity 进行转型
 */

Father father = (Father)manager.find( "com.redsoft.samples.Father", new Integer( 1 ) );
// 更新 Entity
对 Entity 的更新必须在事务内完成。和 persist 中一样，关系元数据的 cascade 属性对是否集联删除有影响。
```

```

transaction.begin();
Father father = manager.find( Father.class, 1 );

// 更新原始数据类型
father.setName( "newName" );

// 更新对象引用
Son newSon = new Son();
father.setSon( newSon );

// 提交事务，刚才的更新同步到数据库
transaction.commit();
// 删除 Entity
对 Entity 的删除必须在事务内完成。

transaction.begin();

Father father = manager.find( Father.class, 1 );

// 如果 father/son 的 @OneToOne 的 cascade=CascadeType.ALL，在删除 father 时
候，也会把 son 删除。
// 把 cascade 属性设为 cascade=CascadeType.REMOVE 有同样的效果。
manager.remove( father );

// 提交事务，刚才的更新同步到数据库
transaction.commit();
//脱离/附合(Detach/Merge)
// Entity 能脱离 EntityManager，避免长时间保持 EntityManager 打开占用资源和可以在
不同的 JVM 之间传递 Entity
EntityManager entityManager = emf.createEntityManager();
Father father = manager.find( Father.class, 1 );
// 当 entityManager 关闭的时候，当前被 entityManager 管理的 Entity 都会自动的脱离
EntityManager，状态转变为 detached
entityManager.close();
// 脱离 EntityManager 后，我们仍然可以修改 Father 的属性
father.setName( "newName" );
// 在稍后的，我们可以将 father 重新附和到一个新的或者原来的 EntityManager 中
EntityManager newEntityManager = emf.createEntityManager();
// 附合(merge)需要在事务中进行
newEntityManager.getTransaction().begin();
newEntityManager.merge( father );
newEntityManager.getTransaction().commit();

```

13.1.3.5 Query 对象

最后需要提及的一个内容恐怕就是 JPA 中进行查询了，即 Query 对象。JPA 使用 `javax.persistence.Query` 接口代表一个查询实例，Query 实例由 `EntityManager` 通过指定查询语句构建。该接口拥有众多执行数据查询的接口方法，可以参考图 13.7。下面对一些方法略作简介：

- `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 `Query setParameter(String name, Object value)` 方法绑定命名参数；
- `Query setMaxResults(int maxResult)`：设置返回的最大结果数；
- `Query setFirstResult(int firstResult)`：设置返回的结果开始下标。
- `int executeUpdate()`：如果查询语句是新增、删除或更改的语句，通过该方法执行更新操作；

如果读者和以前的 Hibernate 中的 Query 对象进行对比的话，会发现几乎没有任何的区别，所以，以前进行分页的代码在这里仍然有效，大家可以参考 7.2.2 Hibernate 要点一节的内容来了解如何分页。现在我们给读者列出一些简单查询的例子来：

```
String queryString = "select model from User model";//这里给的是实体名
List result = entityManager.createQuery(queryString).getResultList();//普通查询
queryString = "select * from UserTable";//这里给的是表名
result = entityManager.createNativeQuery(queryString).getResultList();//SQL 查询
```

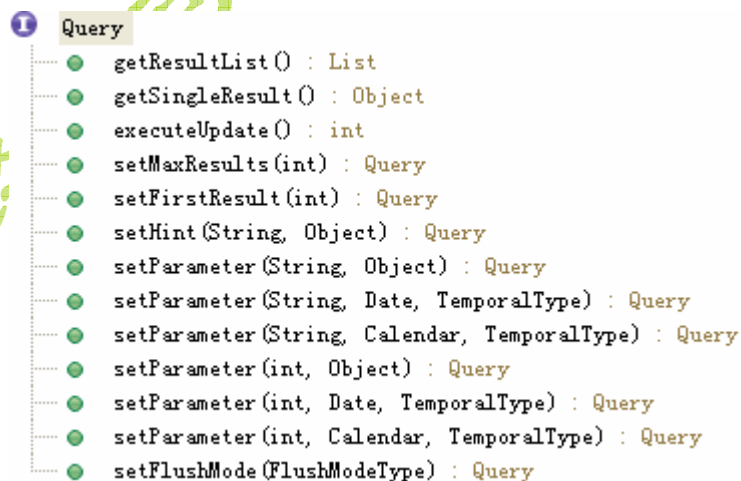


图 13.7 Query 对象方法列表

另外 JPA QL 还支持参数查询，它和 SQL 中的参数查询类似。EJB3 QL 支持两种方式的参数定义方式：命名参数和位置参数。在同一个查询中只允许使用一种参数定义方式。下面就简单的做一些介绍。

命名参数：


```

Query query = entityManager.createQuery( "select o from Order o where o.id
=:myId");
// 设置查询中的参数
query.setParameter( "myId", 2 );

// 可以使用多个参数
Query query = entityManager.createQuery( "select o from Order o where o.id
=:myId and o.customer = :customerName" );

// 设置查询中的参数
query.setParameter( "myId", 2 );
query.setParameter( "customerName", "foo" );

```

注意：不允许在同一个查询中使用两个相同名字的命名参数。

位置参数：

```

Query query = entityManager.createQuery( "select o from Order o where o.id
=?1");
// 设置查询中的参数
query.setParameter( 1, 2 );// 1 表示第一个参数, 2 是参数的值
//或者
Query query = entityManager.createQuery( "select o from Order o where o.id
=?1").setParameter( 1, 2 );
// 可以使用多个参数
Query query = entityManager.createQuery( "select o from Order o where o.id = ?1
and o.customer = ?2" );
// 设置查询中的参数
query.setParameter( 1, 2 );
query.setParameter( 2, "foo" );

```

一般我们建议使用命名参数，比较便于阅读和修改，尤其是参数比较多时，使用下标来定位显然是要把人累死。

13.1.3.6 JPA 查询语言（JPA QL）简介

现在新推出的 JPA 查询语言（JPA QL）已经正式和 EJB 3 查询语言（EJB 3 QL）进行了区分。JPA 的查询语言是面向对象而非面向数据库的，它以面向对象的自然语法构造查询语句，可以看成是 Hibernate HQL 的等价物。在这里我们不打算做 100% 的介绍，因为 SUN 的官方文档关于它有足足几十页那么多的内容，仅列出部分内容供大家作一些了解。

简单的查询：

你可以使用以下语句返回所有 Topic 对象的记录：

```
SELECT t FROM Topic t
```

t 表示 Topic 的别名，在 Topic t 是 Topic AS t 的缩写。

如果需要按条件查询 Topic，我们可以使用：

```
SELECT DISTINCT t FROM Topic t WHERE t.topicTitle = ?1
```

通过 WHERE 指定查询条件，?1 表示用位置标识参数，尔后，我们可以通过 Query 的

setParameter(1, "主题 1")绑定参数。而 DISTINCT 表示过滤掉重复的数据。

如果需要以命名绑定绑定数据，可以改成以下方式：

```
SELECT DISTINCT t FROM Topic t WHERE t.topicTitle = :title
```

这时，需要通过 Query 的 setParameter("title", "主题 1")绑定参数。

使用其它的关系操作符

使用空值比较符，比如查询附件不空的所有帖子对象：

```
SELECT p FROM Post p WHERE p.postAttach IS NOT NULL
```

范围比较符包括 BETWEEN..AND 和 >、>=、<、<=、<> 这些操作符。比如下面的语句查询浏览次数在 100 到 200 之间的所有论坛主题：

```
SELECT t FROM Topic t WHERE t.topicViews BETWEEN 100 AND 200
```

集合关系操作符

和其它实体是 One-to-Many 或 Many-to-Many 关系的实体，通过集合引用关联的实体，我们可以通过集合关系操作符进行数据查询。下面的语句返回所有没有选项的调查论坛的主题：

```
SELECT t FROM PollTopic t WHERE t.options IS EMPTY
```

我们还可以通过判断元素是否在集合中进行查询：

```
SELECT t FROM PollTopic t WHERE :option MEMBER OF t.options
```

这里参数必须绑定一个 PollOption 的对象，JPA 会自动将其转换为主键比较的 SQL 语句。

子查询

JPA 可以进行子查询，并支持几个常见的子查询函数：EXISTS、ALL、ANY。如下面的语句查询出拥有 6 个以上选项的调查主题：

```
SELECT t FROM PollTopic t WHERE (SELECT COUNT(o) FROM t.options o) > 6
```

可用函数

JPA 查询支持一些常见的函数，其中可用的字符串操作函数有：

- ◆ CONCAT(String, String)：合并字段串；
- ◆ LENGTH(String)：求字段串的长度；
- ◆ LOCATE(String, String [, start])：查询字段串的函数，第一个参数为需要查询的字段串，看它在出现在第二个参数字符串的哪个位置，start 表示从哪个位置开始查找，返回查 找到的位置，没有找到返回 0。如 LOCATE ('b1','a1b1c1',1)返回为 3；
- ◆ SUBSTRING(String, start, length)：子字段串函数；
- ◆ TRIM([[[LEADING|TRAILING|BOTH] char] FROM] (String)：将字段串前后的特殊字符去除，可以通过选择决定具体的去除位置和字符；
- ◆ LOWER(String)：将字符串转为小写；
- ◆ UPPER(String)：将字符串转为大写。

数字操作函数有：

- ◆ ABS(number)：求绝对值函数；
- ◆ MOD(int, int)：求模的函数；
- ◆ SQRT(double)：求平方函数；
- ◆ SIZE(Collection)：求集合大小函数。

更新语句

可以用 EntityManager 进行实体的更新操作，也可以通过查询语言执行数据表的字段更新，记录删除的操作：

下面的语句将某一个调查选项更新为新的值：

```
UPDATE PollOption p SET p.optionItem = :value WHERE p.optionId = :optionId
```

我们使用 Query 接口的 `executeUpdate()` 方法执行更新。下面的语句删除一条记录：

```
DELETE FROM PollOption p WHERE p.optionId = :optionId
```

排序和分组

我们还可以对查询进行排序和分组。如下面的语句查询出主题的发表时间大于某个时间值，返回的结果按浏览量降序排列：

```
SELECT t FROM Topic t WHERE t.topicTime > :time ORDER BY t.topicViews DESC
```

下面的语句计算出每个调查主题对应的选项数目：

```
SELECT COUNT(p), p.pollTopic.topicId FROM PollOption p GROUP BY p.pollTopic.topicId
```

这里，我们使用了计算数量的聚集函数 `COUNT()`，其它几个可用的聚集函数分别为：

◆AVG：计算平均值，返回类型为 `double`；

◆MAX：计算最大值；

◆MIN：计算最小值；

◆SUM：计算累加和；

我们还可以通过 `HAVING` 对聚集结果进行条件过滤：

```
SELECT COUNT(p), p.pollTopic.topicId FROM PollOption p GROUP BY p.pollTopic.topicId HAVING p.pollTopic.topicId IN(1,2,3)
```

13.1.3.7 回调方法

实体类支持一些回调方法的标注：

`@PrePersist`

`@PostPersist`

`@PreRemove`

`@PostRemove`

`@PreUpdate`

`@PostUpdate`

`@PostLoad`

。它们标注在某个方法之前，没有任何参数。这些标注下的方法在实体的状态改变前后时进行调用，相当于拦截器，参考图 13.6 实体状态转换关系图，`pre` 表示在状态切换前触发，`post` 则表示在切换后触发。

`@PostLoad` 事件在下列情况触发：

1. 执行 `EntityManager.find()` 或 `getReference()` 方法载入一个实体后；
2. 执行 JPA QL 查询过后；
3. `EntityManager.refresh()` 方法被调用后。

`@PrePersist` 和 `@PostPersist` 事件在实体对象插入到数据库的过程中发生，`@PrePersist` 事件在调用 `EntityManager.persist()` 方法后立刻发生，级联保存也会发生此事件，此时的数据还没有真实插入进数据库。`@PostPersist` 事件在数据已经插入进数据库后发生。

`@PreUpdate` 和 `@PostUpdate` 事件的触发由更新实体引起，`@PreUpdate` 事件在实体的状态同步到数据库之前触发，此时的数据还没有真实更新到数据库。`@PostUpdate` 事件在实体的状态同步到数据库后触发，同步在事务提交时发生。

`@PreRemove` 和 `@PostRemove` 事件的触发由删除实体引起，`@PreRemove` 事件在实体从数据库删除之前触发，即调用了 `EntityManager.remove()` 方法或者级联删除

时发生，此时的数据还没有真实从数据库中删除。

@PostRemove 事件在实体已经从数据库中删除后触发。

另外它们还可以单独定义在一个类里面，然后在实体类上通过在类型前加 **@EntityListener(className=<class>)** 标注来添加，相当于实体监听器。

我们简单举一个例子（实际中用途并不大，而且这样加入打印信息会明显降低 JPA 的运行速度）：

```
@Entity
public class HelloBean {
    ....
    @PostLoad
    public void postLoad() {
        System.out.println("载入了实体 Bean[ " + this. + " ]");
    }
}
```

13.2 准备工作

好了，出于对 JPA 的重视，列出了一堆的相关知识供大家了解（注意上面的内容仍然不全面，但是足够大部分的情况下使用了，读者可以阅读本章参考资料部分了解更多信息），也许大家已经不耐烦了，套用一句流行语：够了！让我们来实践吧！

要开发 JPA 我们只需要准备下列东西：

MyEclipse 6（自带 Hibernate JPA 实现和 Toplink，OpenJPA 等必须类库）；

JDK/JRE 1.5 或者更高版本（MyEclipse 6 的 ALL In ONE 版本已经带）；

必须的数据库服务器软件及其 JDBC 驱动，我们这里打算使用 MySQL，其安装和使用方法在本书第四章和第一章都有介绍。也可以使用 MyEclipse 自带的 Derby。

如果读者想不用 MyEclipse 开发，那需要自己下载 Hibernate 或者 TopLink 的类库，然后配置到类路径中使用任何开发工具进行编译运行即可。参考本章的参考资料一节。

13.3 创建 JPAHello 项目

本节内容相对来说比较简单，主要包括：

1. 创建一个项目，并添加 **JPA Capabilities**（JPA 开发功能）；
2. 使用 **DB Explorer** 试图来选中表格并反向工程生成 JPA 代码。

通过这些步骤可以让大家了解 MyEclipse 6 下快速开发 JPA 的过程。

13.3.1 创建表格

请参考第 5 章创建数据库表格的内容来建立项目所需要的表格（包括用于 MySQL 数据库和 Derby 数据库等等的 SQL 语句），这里仅仅列出来 MySQL 建表的 SQL 语句：

```
CREATE TABLE Student (
    id int NOT NULL auto_increment,
```

```
username varchar(200) NOT NULL,
password varchar(20) NOT NULL,
age int,
PRIMARY KEY (id)
) ENGINE=MyISAM DEFAULT CHARSET=GBK
```

。需要注意的是自增属性一定要设置，否则主键生成器就成了问题了。读者可以参考 [5.2 创建数据库表格](#) 一节来了解这个建表语句的更多细节内容以及建表的操作步骤，还有相关的注意事项。

13.3.2 创建 HelloJPA Java Project

在 MyEclipse 中我们可以将 *JPA* 开发功能添加到很多种项目上，一般最常见的恐怕就是添加到 Java 项目或者 Web 项目上了。在本节我们打算用简单的 *Java Project*，便于展示 JPA 如何工作。

首先确保已经打开了 **MyEclipse Java Enterprise** 透视图，然后进行下列操作：

1. 从 MyEclipse 菜单栏选择 **File > New > Java Project**，接着会打开 **New Java Project** 向导；
2. 输入 *HelloJPA* 到 **Project name**；
3. 在 **Project Layout** 下选中 **Create separate source and output folders** 单选钮；
4. 点击 **Finish** 按钮关闭对话框。

这样 Java 项目就建立完毕了。稍等片刻会弹出一个切换透视图的对话框，为了避免造成更多的麻烦，我们一般选择 **No** 按钮就可以了。

13.3.3 添加 JPA Capabilities 到现有项目

创建了项目之后，我们需要给项目添加 *JPA Capabilities*，整个处理过程将执行一系列的操作：

1. 添加 JPA 类库 (JARs) 到项目的类路径；
2. 在项目中创建并配置 *persistence.xml*；
3. 给项目加入 JDBC 驱动类库；

最后，它还可以将项目和数据库浏览器中的连接关联起来，让我们可以在项目里面选择表信息，以及为以后的反向工程生成 JPA 代码打下基础。

要给项目添加 **Hibernate** 功能，请按照下面的步骤进行：

在 **Package Explorer** 中选择 *HelloJPA* 项目；

接下来，从 MyEclipse 菜单栏选择 **MyEclipse > Project Capabilities > Add JPA Capabilities ...** 来启动 **Add Hibernate Capabilities** 向导，如图 13.8 所示。

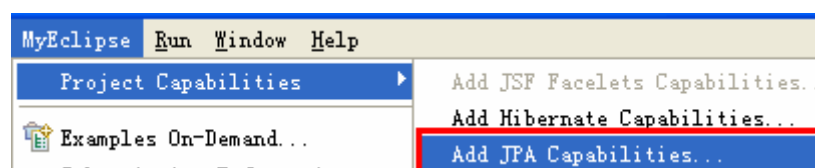


图 13.8 启动添加 JPA 功能向导

接着将会弹出 **Add JPA Capabilities** 对话框，如下图所示：

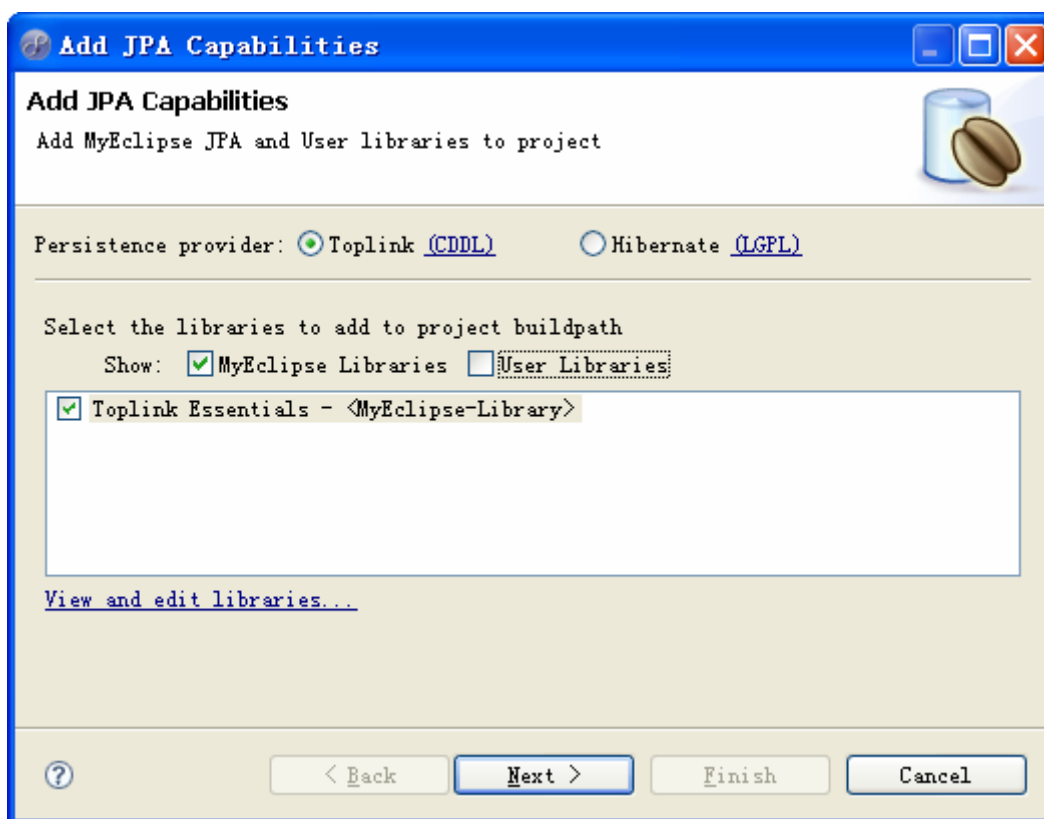


图 13.9 Add JPA Capabilities 向导对话框

在这个对话框中保持默认值就可以了，然后点击 **Next** 按钮即可进入下一页的设置。不过，还是稍微给大家介绍一下这个对话框的相关设置。**Persistence provider**: 右侧列出了两套持久化提供者（也就是 JPA 框架）的名字，分别是 *TopLink* 和 *Hibernate*，括号内的字母 *CDDL* 和 *LGPL* 则是它们的许可协议（这两个都是开源免费的，可以放心使用）。下方的列表框中的 *Toplink Essentials - <MyEclipse-Library>* 则是对应的类库的名字，选中它的话就可以加入到项目的类路径中去。图 13.10 列出了点击 **Hibernate** 单选钮后显示类库列表，我们只需要了解一下 *Hibernate* 的 JPA 实现是 *Hibernate* 核心类库（Core）+ 标注支持（Annotations）+ 实体管理器（Entity Manager）三个包，这在本章的参考资料一节也有所体现，如果读者希望升级的话可以参考一下该手工下载哪些包。如果读者使用的是 *Oracle* 数据库，推荐使用 *TopLink*，否则一般大多数人使用的是 *Hibernate*。在这里我们选择的是 *Hibernate*，这样便于读者和以前的单独 *Hibernate* 开发有所对比。

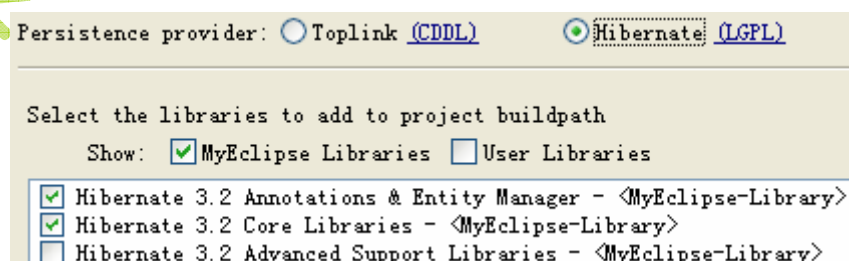


图 13.10 Hibernate JPA 类库

向导的最后一页，显示的是和数据库以及 *persistence.xml* 设置有关的选项。在这一页我们可以给持久化单元输入名字，并指定 JPA 数据源的连接信息。数据库连接信息（Driver）

列表中列出的是 **Database Explorer** 中的建立好的连接列表，这里选择 *mysql5*。在 **Catalog/Schema** 中可以选择数据库的名字，如果看不到列表的话点击 **Update List** 按钮就可以连接到数据库并取出连接列表。**Enable dynamic DB table creation** 这个选项指定是否动态创建表（根据实体类中的标注信息和字段列表自动生成建表语句然后执行），一般情况下，除非是做练习，懒得写 SQL，我们都不要启用这个选项。最后点击 **Finish** 按钮关闭对话框，完成整个向导。



图 13.11 配置持久化单元

13.3.4 使用 JPA 配置文件编辑器修改文件

向导完成后，我们可以看到项目的目录结构如图 13.12 所示。能够注意到文件 *persistence.xml* 上面有个红叉，这表明这个文件里面有错误。如果大家还有印象的话，当时我们做 **Hibernate** 例子的时候也有这样的错，这其实是 **MyEclipse** 自身的一个 **Bug**，双击 *persistence.xml* 就可以用 **MyEclipse** 打开并编辑这份 XML 配置文件，并对其中的一些属性进行验证。我们先来看这个出错的 *persistence.xml* 文件的源代码：

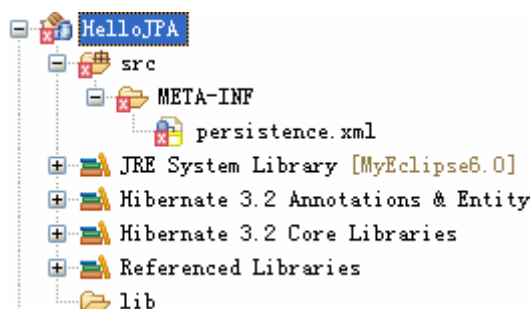


图 13.12 JPA 项目结构

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
  version="1.0">

  <persistence-unit name="HelloJPAPU"
    transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <properties>
      <property name = "hibernate.connection.driver_class" value =
        "com.mysql.jdbc.Driver"/>
      <property name = "hibernate.connection.url" value =
        "jdbc:mysql://localhost:3306/test?useUnicode=true&characterEncoding=G
        BK"/>
      <property name = "hibernate.connection.username" value =
        "root"/>
    </properties>
  </persistence-unit>

</persistence>

```

。其实出错的原因很简单，就是`&`没有进行 XML 转义，我们只需要把它改成`&`就可以了。这个文件其它的内容，就是定义了持久化单元的名字：*HelloJPAPU*，驱动类名，数据库连接 URL 以及用户名。如果先前的时候读者选中了创建动态表格，那么现在还可以看到一句额外的属性`<property name = "hibernate.hbm2ddl.auto" value = "update"/>`。然而，这里其实是缺少了很关键的属性的，包括数据库密码和方言设置（Dialect，Hibernate 特有的参数），其它还有的一些参数包括 Cache（缓存）控制等等。最后，我们还加入了调试属性 *hibernate.show_sql* 来显示 Hibernate 所生成的 SQL 语句。这样，最终修改过的 *persistence.xml* 源代码清单如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
  version="1.0">

  <persistence-unit name="HelloJPAPU"
    transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <properties>
      <!-- JDBC 驱动类名 -->

```

```

        <property name = "hibernate.connection.driver_class" value =
"com.mysql.jdbc.Driver"/>
        <!-- 数据库连接 URL -->
        <property name = "hibernate.connection.url" value =
"jdbc:mysql://localhost:3306/test?useUnicode=true&characterEncoding=GBK"/>
        <!-- 数据库密码 -->
        <property name = "hibernate.connection.password" value = ""/>
        <!-- 数据库用户名 -->
        <property name = "hibernate.connection.username" value =
"root"/>
        <!-- Hibernate 方言(只有Hibernate 才需要设置) -->
        <property name = "hibernate.dialect" value =
"org.hibernate.dialect.MySQLDialect"/>
        <!-- Hibernate 显示调试 SQL -->
        <property name = "hibernate.show_sql" value = "true"/>
        <!-- Hibernate 缓存设置 (默认可不设置) -->
        <property name = "hibernate.cache.provider_class" value =
"org.hibernate.cache.NoCacheProvider"/>
    </properties>
</persistence-unit>
</persistence>

```

。这些属性的名字和取值，不同的 JPA 实现一般是不一样的，为了便于读者对比，我们再列一份 TopLink 下的 **persistence.xml** 源代码清单：

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
    version="1.0">

    <persistence-unit name="JPADemoPU"
        transaction-type="RESOURCE_LOCAL">
        <provider>
            oracle.toplink.essentials.PersistenceProvider
        </provider>
        <class>jpadao.Myuser</class>
        <properties>
            <property name="toplink.jdbc.driver"
                value="com.mysql.jdbc.Driver" />
            <property name="toplink.jdbc.url"
                value="jdbc:mysql://localhost:3306/test?useUnicode=true&chara

```

```

cterEncoding=GBK" />
    <property name="toplink.jdbc.user" value="root" />
  </properties>
</persistence-unit>

</persistence>


```

。可以看到其属性一般是不怎么相同的，具体的内容一般要参考一些官方的文档才能全面了解。

到这里为止，并不能看到 JPA 比传统的 Hibernate 开发有什么精简的地方，因此我们还得继续往下看才行。

13.3.5 使用反向工程快速生成 JPA 实体类和 DAO

MyEclipse 工具的优势又要再次展示出来，它可以让我们几乎不写一行代码就生成实体类（甚至可以生成一对多，多对多等等）以及 DAO 类，并更新配置文件。

首先打开 **MyEclipse Database Explorer** 透视图。切换透视图有两种办法，如何切换请参考 3.1.3 透视图（Perspective）切换器。一种比较快办法是如那一节介绍的，点击工具栏上的点击  按钮可以显示多个透视图供切换，如图 3.3 所示，然后单击其中的 **MyEclipse Database Explorer** 即可切换到此透视图；另一种办法是选择菜单 **Window > Open Perspective > Other > MyEclipse Database Explorer** 来显示打开透视图对话框，然后点击 **OK** 按钮。

接着选中 **DB Browser** 视图中的数据库连接 *mysql5*（根据您的情况，也可以使用 Derby，Oracle 等等其它数据库，但是必须和 13.3.3 节中 JPA 项目配置的数据库连接一致），点击并展开数据库里面的树状表结构，直到看到你希望处理的数据库表为止，单击选中表。**注意：**你可以选中或者一个多个要处理的表。在这个例子中要找的表是 **Student**。接着点击右键在上下文菜单中选择 **JPA Reverse Engineering...**，这将启动 **JPA Reverse Engineering** 向导。如下图所示：

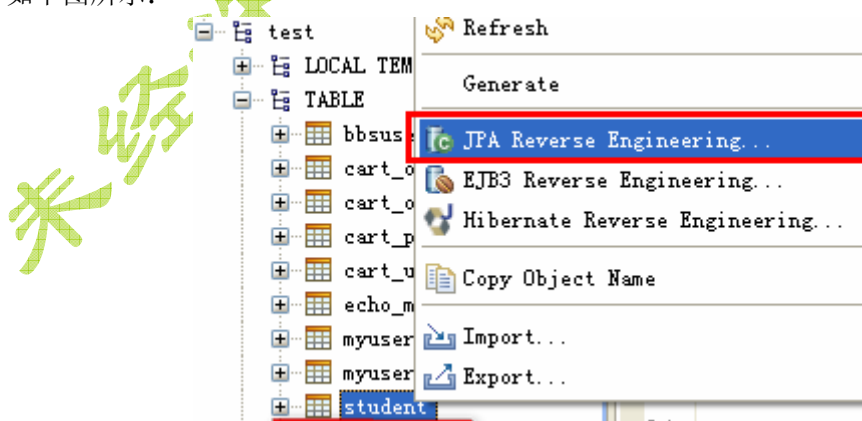


图 13.13 在 DB Browser 视图中启动反向工程向导

之后启动的 **JPA Reverse Engineering** 向导则如图 13.14 所示。在这个图里面有一些选项需要进行设置，这些选项包括：

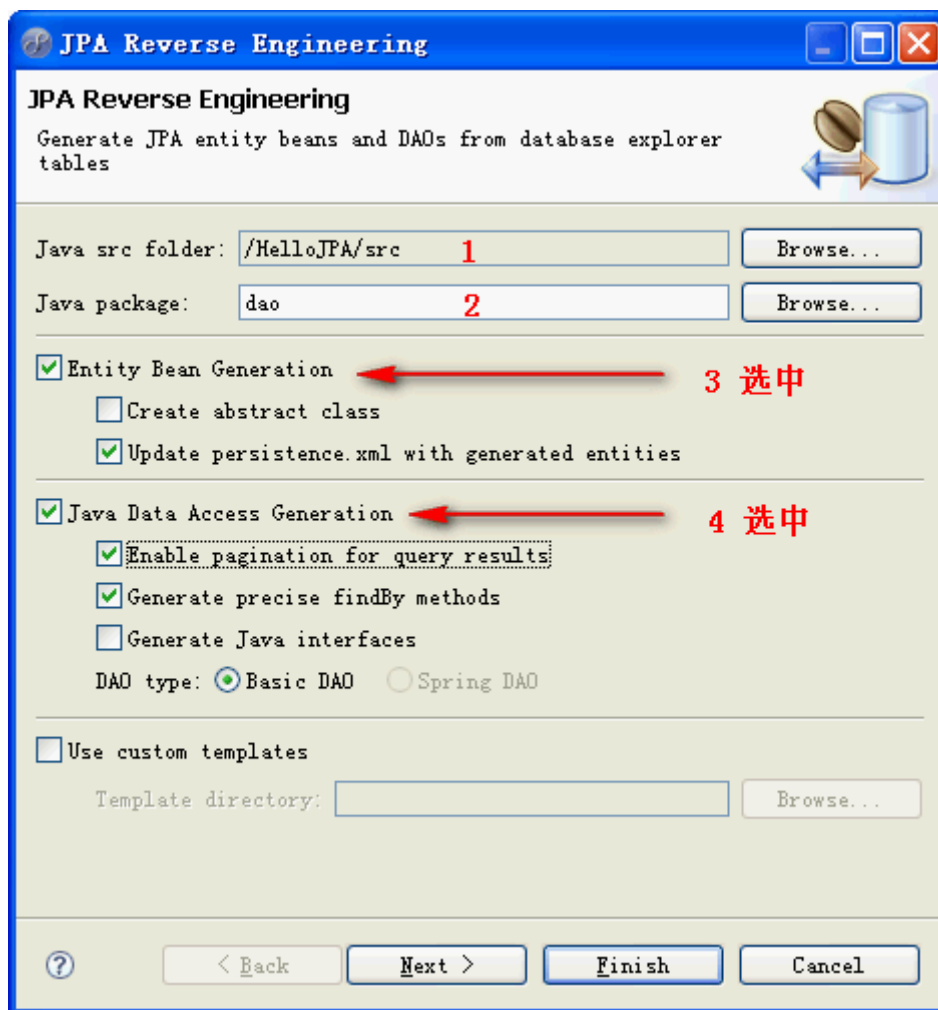


图 13.14 JPA 反向工程向导第 1 页

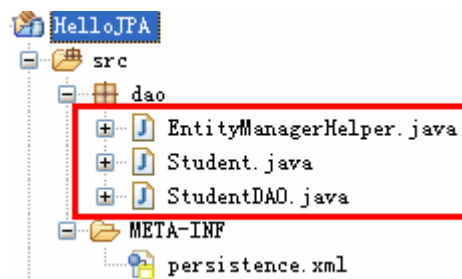
1. **Java src folder (Java 源代码目录)**: 这些源代码目录将用来存放最终生成的文件。点击最右侧的 **Browse** 按钮，查看并选中可用的 JPA 项目以及源码目录。这里选中 **HelloJPA** 项目中的 **src** 文件夹。
2. **Java package (Java 包)**: 点击右侧的 **Browse** 按钮，来选中现有的包，或者新建一个其它的包来存放生成的代码所在的包。在这里我们输入了 **dao** 作为包名。
3. **Entity Bean Generation (实体类生成)**: 选中该选项后，MyEclipse 将会生成带有正确标注的普通 Java 类，这样她们就能作为 JPA 的实体类来使用。下面有两个子选项。
 - **Create abstract class (创建抽象类)**: 如果希望每次都在重新覆盖生成了实体类后，还留有自己加入的自定义代码（例如简单的计算总数的代码），MyEclipse 可以只在生成时更新抽象的基类，而不会覆盖具体的实体类里面的代码，这样我们可以留着自己编写的功能代码而不用担心每次反向工程生成实体类后，都得把原来的代码再加入一次，这时候只要选中这个选项就可以了。
 - **Update persistence.xml with generated entities (更新生成的实体到 persistence.xml)**: 和 Hibernate 类似，这个选项可以自动把生成的 JPA 实体类加入到 JPA 配置文件 *persistence.xml* 中。

4. **Java Data Access Generation (Java 数据访问层生成)**: 该选项将告诉 MyEclipse 是否生成增删改查实体的 DAO (数据访问对象) 工具类。这些代码将会很好的包装实体管理器代码和对应的实体操作代码, 还可以附带生成查询语句等。它还有三个子选项可以使用。

- **Enable Pagination for query results (对查询结果启用分页)**: 在最终的代码中生成分页查询, 例如
- **Generate Precise findBy Methods (生成精确的 findBy 方法)**: 选中后 MyEclipse 将会生成 *findByXXX* 方法, 其中的 XXX 对应着实体的每一个属性 (每个实体对应一个 *findByXXX*)。这样可以从数据库中使用任何属性作为查询的条件。
- **Generate Java Interfaces (生成 Java 接口)**: 选中后, 将会给 DAO 类生成一个接口, 主要是为了便于和 Spring 进行整合, 一般为了减少代码复杂度, 不选中即可。
- **DAO Type (DAO 的类型)**: 在右侧的单选钮中可以选择所生成的 DAO 的类型。在我们的项目中只能选中普通的 DAO (Basic DAO), 而 Spring DAO 单选钮则必须在项目添加了 Spring 开发功能后才能选中。

注意: Spring DAO 这个单选钮是灰色, 如果可用则能生成 Spring+JPA 的 DAO (JPADaoSupport), 如何生成这样的 DAO 在以后会做介绍。

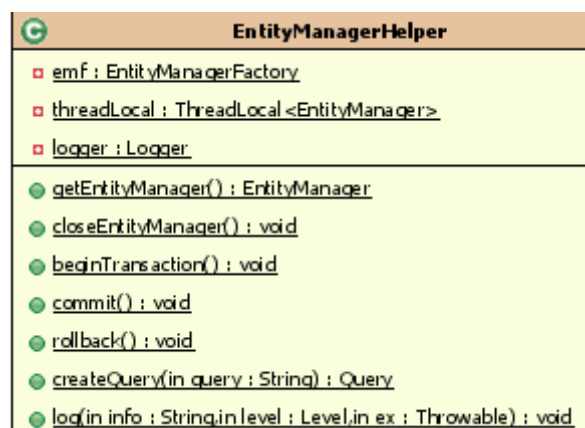
在设置完了第一页的选项后, 如果是单表映射, 可以直接点击 **Finish** 按钮稍等片刻后就可以结束代码的生成。代码生成完毕后会提示我们切换到 **MyEclipse Persistence** 透视图, 在这里我们选择 **No** 按钮, 然后切换回 **MyEclipse Java Enterprise** 透视图。至此代码的生成已经全部完成, 非常方便快捷, 的确提高了开发的效率。现在我们看看生成的代码的目录结构, 如下图所示。一共生成了三个类, 包括实体类 *Student*, 实体的 DAO 类



StudentDAO, 以及实体管理器的工具类 *EntityManagerHelper*。读者这时候可以对照 13.1.3 节的代码结构介绍来加深了解。由于我们生成的时候选择了支持对查询结果进行分页, 因此所生成的 DAO 代码中, 会有一些稍微比较难懂的地方, 后面我们再来做解释。首先如果打开 *persistence.xml*, 会发现其中已经加入了所生成的

实体类: `<class>dao.Student</class>`。

现在看看生成的类的 UML 图, 如图 13.15 所示。因为实体类主要就是 *getter* 和 *setter* 方法以及标注, 因此在此我们就不在图中列出了。



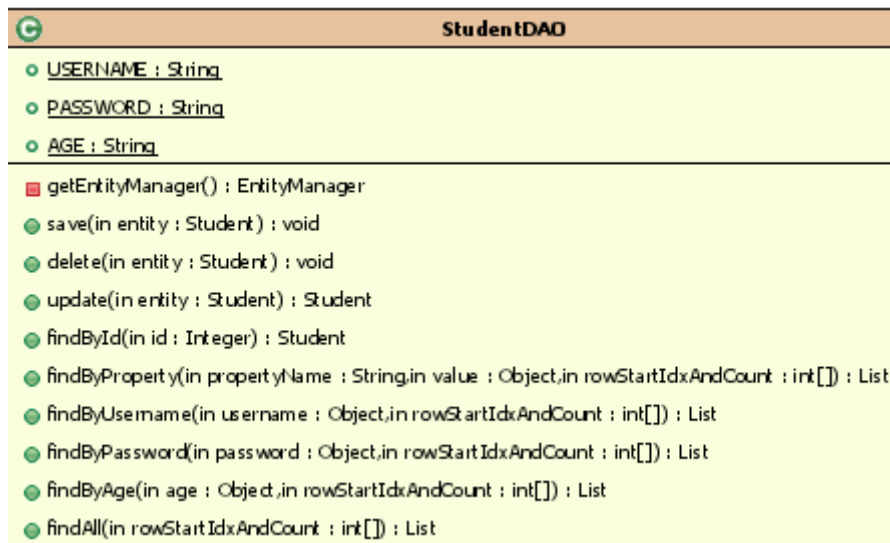


图 13.15 生成的 JPA 代码类图

首先瞄一眼实体类 **dao.Student** 的源代码清单：

```

package dao;

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

/**
 * Student entity.
 *
 * @author MyEclipse Persistence Tools
 */
@Entity
@Table(name = "student", catalog = "test", uniqueConstraints = {})
public class Student implements java.io.Serializable {

    // Fields

    private Integer id;
    private String username;
    private String password;
    private Integer age;

    // Constructors

    /** default constructor */
    public Student() {
  
```

```
}

/** minimal constructor */
public Student(Integer id, String username, String password) {
    this.id = id;
    this.username = username;
    this.password = password;
}

/** full constructor */
public Student(Integer id, String username, String password, Integer
age) {
    this.id = id;
    this.username = username;
    this.password = password;
    this.age = age;
}

// Property accessors
@Id
@Column(name = "id", unique = true, nullable = false, insertable =
true, updatable = true)
public Integer getId() {
    return this.id;
}

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

@Column(name = "username", unique = false, nullable = false,
insertable = true, updatable = true, length = 200)
public String getUsername() {
    return this.username;
}

public void setUsername(String username) {
    this.username = username;
}

@Column(name = "password", unique = false, nullable = false,
insertable = true, updatable = true, length = 20)
public String getPassword() {
    return this.password;
}
```

```

    }

    public void setPassword(String password) {
        this.password = password;
    }

    @Column(name = "age", unique = false, nullable = true, insertable =
true, updatable = true)
    public Integer getAge() {
        return this.age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }
}

```

。大部分代码都生成的比较完整了，包括属性和必要的实体标注，不过有些地方还是要稍微做些调整的，主要是主键生成器的问题，在下一节 13.3.6 咱们再讨论。

生成的类 `dao.EntityManagerHelper` 主要封装了获取 `EntityManager` 和操作事务（`Transaction`）的代码，并使用了一种叫做线程局部（`thread-local`）的模式来存放对象。下面列出了这些方法的说明：

方法	说明
<code>getEntityManager()</code>	获取实体管理器对象
<code>closeEntityManager()</code>	关闭实体管理器对象
<code>beginTransaction()</code>	开始事务
<code>commit()</code>	提交事务
<code>rollback()</code>	回滚事务
<code>createQuery()</code>	创建查询对象
<code>log()</code>	打印日志信息

。基本上这些方法都封装的比较好，可以用来编写数据操作的代码，例如 DAO，不过读者可以根据自己的需要进行必要的改进。创建实体管理器工厂的代码是：

```
emf = Persistence.createEntityManagerFactory("HelloJPAPU");
```

。这是关键代码，`HelloJPAPU` 则是持久化单元的名字，位于 `persistence.xml` 中。创建日志调试信息的代码是：

```
logger = Logger.getLogger("HelloJPAPU");
```

```
logger.setLevel(Level.ALL);
```

。为了便于读者对比，我们列出类 `dao.EntityManagerHelper` 的代码清单：

```

package dao;

import java.util.logging.Level;
import java.util.logging.Logger;

import javax.persistence.EntityManager;

```



```
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;
/**
 * @author MyEclipse Persistence Tools
 */
public class EntityManagerHelper {

    private static final EntityManagerFactory emf;
    private static final ThreadLocal<EntityManager> threadLocal;
    private static final Logger logger;

    static {
        emf = Persistence.createEntityManagerFactory("HelloJPAPU");

        threadLocal = new ThreadLocal<EntityManager>();
        logger = Logger.getLogger("HelloJPAPU");
        logger.setLevel(Level.ALL);
    }

    public static EntityManager getEntityManager() {
        EntityManager manager = threadLocal.get();
        if (manager == null || !manager.isOpen()) {
            manager = emf.createEntityManager();
            threadLocal.set(manager);
        }
        return manager;
    }

    public static void closeEntityManager() {
        EntityManager em = threadLocal.get();
        threadLocal.set(null);
        if (em != null) em.close();
    }

    public static void beginTransaction() {
        getEntityManager().getTransaction().begin();
    }

    public static void commit() {
        getEntityManager().getTransaction().commit();
    }

    public static void rollback() {
```

```

    getEntityManager().getTransaction().rollback();
}

public static Query createQuery(String query) {
    return getEntityManager().createQuery(query);
}

public static void log(String info, Level level, Throwable ex) {
    logger.log(level, info, ex);
}
}

```

。接下来我们要看的就是生成的 DAO 类 `dao.StudentDAO` 的代码了。要看懂它的代码，必须先弄明白一个 JDK 1.5 中新增的语法功能：**vararg(动态参数)**。简单说，它提供了可以给一个方法加入 0 个，单个或者多个参数，并封装到一个数组中的功能，例如下面的例子：

```

public static void test(int... is) {
    System.out.println(is.length);
    for (int i = 0; i < is.length; i++) {
        System.out.println(is[i]);
    }
}

```

。这段代码定义了一个方法，可以传入参数，它约等于下面的定义：

```
public static void test(int[] is)
```

。也就是说所有的参数会被 JVM 自动的封装成一个数组，但是后面的定义方式不能传递 0 个数。下面是这个方法可以被调用的方式：

```

test();
test(1);
test(1,2);
test(1,2,3,4);

```

。这些都是合法的使用方式。如果方法中待传入的参数除了动态参数外，还有其它参数，则必须将动态参数方法在参数列表的最后面，例如：

```
public List<Student> findByAge(Object age, int... rowStartIdxAndCount)
```

。好了，有了这些背景知识，现在我们可以来讨论 DAO 类的代码了。先看一下主要的方法列表：

方法	说明
<code>getEntityManager()</code>	获取实体管理器
<code>save()</code>	保存实体
<code>delete()</code>	删除实体
<code>update()</code>	更新实体
<code>findById()</code>	根据主键(ID)查找实体
<code>findByProperty()</code>	根据属性查找实体（带分页功能）
<code>findByUsername()</code>	根据用户名查找实体（带分页功能）
<code>findByPassword()</code>	根据密码查找实体（带分页功能）

findByAge()	根据年龄查找实体（带分页功能）
findAll()	找出所有实体（带分页功能）

。仔细阅读代码的注释，读者将会发现 `save()`、`delete()` 和 `update()` 方法里面的注释有类似于这样的说明：

This operation must be performed within the a databasetransaction context for the entity's data to be permanently saved to thepersistence store, i.e., database. This method uses the{@link javax.persistence.EntityManager#persist(Object) EntityManager#persist} operation.

```
<pre>
```

```
EntityManagerHelper.beginTransaction();
StudentDAO.save(entity);
EntityManagerHelper.commit();
```

```
</pre>
```

。它的意思是说类似于这样的操作由于要和数据库打交道，所以必须要放在事务中进行（和之前的 Hibernate 是类似的），示例代码是：

```
EntityManagerHelper.beginTransaction();
StudentDAO.save(entity);
EntityManagerHelper.commit();
```

。关于分页代码的调用方法，我可以演示一下根据年龄查找的，其实它最多只接受两个参数，第一个是数据结果开始下标，第二个是结束下标，可以不穿参数，也可以一次只传一个参数，或者一次传两个，但是多了就不行了，具体参考 `findByProperty()` 方法的实现代码。现在我们来查看调用的示例代码：

```
StudentDAO dao = new StudentDAO();
List<Student> result = dao.findByAge(20); // 查出所有年龄为 20 的人
List<Student> result = dao.findByAge(20, 10); // 查出所有年龄为 20 的人，包括从第 10 个
到最后一个
List<Student> result = dao.findByAge(20, 10, 5); // 查出所有年龄为 20 的人，从第 10 个开
始，到第 15 个结束（共最多 5 条记录）
```

。

为了便于读者比较，这里把 `dao.StudentDAO` 的完整代码列出：

```
package dao;

import java.util.List;
import java.util.logging.Level;
import javax.persistence.EntityManager;
import javax.persistence.Query;

/**
 * A data access object (DAO) providing persistence and search support
 for
 * Student entities. Transaction control of the save(), update() and
 delete()
 * operations must be handled externally by senders of these methods or
```

```

must be
 * manually added to each of these methods for data to be persisted to
the JPA
 * datastore.
 *
 * @see dao.Student
 * @author MyEclipse Persistence Tools
 */

public class StudentDAO {
    // property constants
    public static final String USERNAME = "username";
    public static final String PASSWORD = "password";
    public static final String AGE = "age";

    private EntityManager getEntityManager() {
        return EntityManagerHelper.getEntityManager();
    }

    /**
     * Perform an initial save of a previously unsaved Student entity.
All
     * subsequent persist actions of this entity should use the #update()
     * method. This operation must be performed within the a database
     * transaction context for the entity's data to be permanently saved
to the
     * persistence store, i.e., database. This method uses the
     * {@link javax.persistence.EntityManager#persist(Object)
EntityManager#persist}
     * operation.
     *
     * <pre>
     * EntityManagerHelper.beginTransaction();
     * StudentDAO.save(entity);
     * EntityManagerHelper.commit();
     * </pre>
     *
     * @param entity
     *         Student entity to persist
     * @throws RuntimeException
     *         when the operation fails
     */
    public void save(Student entity) {
        EntityManagerHelper.log("saving Student instance", Level.INFO,

```

```

null);

    try {
        getEntityManager().persist(entity);
        EntityManagerHelper.log("save successful", Level.INFO,
null);
    } catch (RuntimeException re) {
        EntityManagerHelper.log("save failed", Level.SEVERE, re);
        throw re;
    }
}

/**
 * Delete a persistent Student entity. This operation must be performed
 * within the a database transaction context for the entity's data
to be
 * permanently deleted from the persistence store, i.e., database.
This
 * method uses the
 * {@link javax.persistence.EntityManager#remove(Object)
EntityManager#delete}
 * operation.
 *
 * <pre>
 * EntityManagerHelper.beginTransaction();
 * StudentDAO.delete(entity);
 * EntityManagerHelper.commit();
 * entity = null;
 * </pre>
 *
 * @param entity
 *      Student entity to delete
 * @throws RuntimeException
 *      when the operation fails
 *
public void delete(Student entity) {
    EntityManagerHelper.log("deleting Student instance",
Level.INFO, null);
    try {
        entity = getEntityManager().getReference(Student.class,
            entity.getId());
        getEntityManager().remove(entity);
        EntityManagerHelper.log("delete successful", Level.INFO,
null);
    } catch (RuntimeException re) {

```



```

        EntityManagerHelper.log("delete failed", Level.SEVERE, re);
        throw re;
    }
}

/**
 * Persist a previously saved Student entity and return it or a copy
of it
 * to the sender. A copy of the Student entity parameter is returned
when
 * the JPA persistence mechanism has not previously been tracking the
 * updated entity. This operation must be performed within the a
database
 * transaction context for the entity's data to be permanently saved
to the
 * persistence store, i.e., database. This method uses the
 * {@link javax.persistence.EntityManager#merge(Object)
EntityManager#merge}
 * operation.
 *
 * <pre>
 * EntityManagerHelper.beginTransaction();
 * entity = StudentDAO.update(entity);
 * EntityManagerHelper.commit();
 * </pre>
 *
 * @param entity
 *         Student entity to update
 * @returns Student the persisted Student entity instance, may not
be the
 *         same
 * @throws RuntimeException
 *         if the operation fails
 */
public Student update(Student entity) {
    EntityManagerHelper.log("updating Student instance",
Level.INFO, null);
    try {
        Student result = getEntityManager().merge(entity);
        EntityManagerHelper.log("update successful", Level.INFO,
null);
        return result;
    } catch (RuntimeException re) {
        EntityManagerHelper.log("update failed", Level.SEVERE, re);
    }
}

```

```

        throw re;
    }
}

public Student findById(Integer id) {
    EntityManagerHelper.log("finding Student instance with id: " + id,
        Level.INFO, null);
    try {
        Student instance = getEntityManager().find(Student.class,
id);

        return instance;
    } catch (RuntimeException re) {
        EntityManagerHelper.log("find failed", Level.SEVERE, re);
        throw re;
    }
}

/**
 * Find all Student entities with a specific property value.
 *
 * @param propertyName
 *         the name of the Student property to query
 * @param value
 *         the property value to match
 * @param rowStartIdxAndCount
 *         Optional int varargs. rowStartIdxAndCount[0] specifies
the the
 *         row index in the query result-set to begin collecting the
 *         results. rowStartIdxAndCount[1] specifies the the
maximum
 *         number of results to return.
 * @return List<Student> found by query
 */
@SuppressWarnings("unchecked")
public List<Student> findByProperty(String propertyName,
    final Object value, final int... rowStartIdxAndCount) {
    EntityManagerHelper.log("finding Student instance with property:
"
        + propertyName + ", value: " + value, Level.INFO, null);
    try {
        final String queryString = "select model from Student model
where model."
            + propertyName + " = :propertyValue";
        Query query = getEntityManager().createQuery(queryString);
    }
}

```

```

        query.setParameter("propertyValue", value);
        if (rowStartIdxAndCount != null && rowStartIdxAndCount.length
> 0) {
            int rowStartIdx = Math.max(0, rowStartIdxAndCount[0]);
            if (rowStartIdx > 0) {
                query.setFirstResult(rowStartIdx);
            }

            if (rowStartIdxAndCount.length > 1) {
                int rowCount = Math.max(0, rowStartIdxAndCount[1]);
                if (rowCount > 0) {
                    query.setMaxResults(rowCount);
                }
            }

            return query.getResultList();
        } catch (RuntimeException re) {
            EntityManagerHelper.log("find by property name failed",
                Level.SEVERE, re);
            throw re;
        }
    }

    public List<Student> findByUsername(Object username,
        int... rowStartIdxAndCount) {
        return findByProperty(USERNAME, username, rowStartIdxAndCount);
    }

    public List<Student> findByPassword(Object password,
        int... rowStartIdxAndCount) {
        return findByProperty(PASSWORD, password, rowStartIdxAndCount);
    }

    public List<Student> findByAge(Object age, int...
rowStartIdxAndCount) {
        return findByProperty(AGE, age, rowStartIdxAndCount);
    }

    /**
     * Find all Student entities.
     *
     * @param rowStartIdxAndCount
     *         Optional int varargs. rowStartIdxAndCount[0] specifies
the the

```

```

*          row index in the query result-set to begin collecting the
*          results. rowStartIdxAndCount[1] specifies the the
maximum
*          count of results to return.
* @return List<Student> all Student entities
*/
@SuppressWarnings("unchecked")
public List<Student> findAll(final int... rowStartIdxAndCount) {
    EntityManagerHelper.log("finding all Student instances",
Level.INFO,
        null);
    try {
        final String queryString = "select model from Student model";
        Query query = getEntityManager().createQuery(queryString);
        if (rowStartIdxAndCount != null && rowStartIdxAndCount.length
> 0) {
            int rowStartIdx = Math.max(0, rowStartIdxAndCount[0]);
            if (rowStartIdx > 0) {
                query.setFirstResult(rowStartIdx);
            }

            if (rowStartIdxAndCount.length > 1) {
                int rowCount = Math.max(0, rowStartIdxAndCount[1]);
                if (rowCount > 0) {
                    query.setMaxResults(rowCount);
                }
            }
        }
        return query.getResultList();
    } catch (RuntimeException re) {
        EntityManagerHelper.log("find all failed", Level.SEVERE, re);
        throw re;
    }
}
}

```

13.3.6 调整生成的实体类标注

和以前开发 **Hibernate** 的时候一样，默认生成的实体类代码很多时候都不符合我们的要求，必须加以修改。实体类的代码在上一节我们已经列出，现在我们考虑的是要对它的主键进行修改，加入自动生成主键值的功能，采用的生成策略是 **IDENTITY**，可以配合 **MySQL** 的自增字段（**increment**）来使用。下面已粗斜体列出新加入的代码，位于类 **dao.Student**

中:

```
// Property accessors
@Id
@Column(name = "id", unique = true, nullable = false, insertable =
true, updatable = true)
@GeneratedValue(strategy=javax.persistence.GenerationType.I
DENTITY)
public Integer getId() {
    return this.id;
}
```

。关于此标注的详细意义，我们已经在 [13.1.3.3 实体类及标注](#) 一节详细讨论过了，读者可参考那一章的介绍。如果不设置的话，默认情况下，JPA 持续性提供程序选择最适合于基础数据库的主键生成器类型，其默认值为 *GenerationType.AUTO*，一般情况下不会出现问题，不过最好还是指定一下。

13.3.7 编写测试代码

现在，所有的代码已经准备就绪，我们要做的就是编写一个测试类，来展示如何使用这些代码来和数据库打交道。这些调用 DAO 的代码更多的时候有个术语，叫业务层，或者有人称作领域模型。出于简单的考虑，这里只写一个带有 `main` 方法的类进行测试，它的名字是 `test.JPATest`。

选择菜单 **File > New > Package**，在出现的创建包的对话框中的 **Name** 框中输入 `test` 后，点击 **Finish** 按钮后就创建完了包。接着选择菜单 **File > New > Class**，在出现的创建类的对话框中的 **Name** 框中输入 `JPATest` 后，点击 **Finish** 按钮后就创建完了测试类。现在我们把它的代码改为如下所示：

```
package test;
import java.util.List;
import dao.*;

public class JPATest {
    public static void main(String[] args) {
        // 1. 创建 DAO
        StudentDAO dao = new StudentDAO();
        // 2. 创建实体类
        Student user = new Student();
        user.setUsername("hellojpa test");
        user.setPassword("jpa password");
        user.setAge(20);

        // 开始事务
        EntityManagerHelper.beginTransaction();
        // 3. 保存
        dao.save(user);
    }
}
```



```
// 提交事务真正保存实体到数据库
EntityManagerHelper.commit();

// 4. 列出数据库中所有对象
List<Student> result = dao.findAll();

for(Student o : result) {
    System.out.println(o.getId());
    System.out.println(o.getUsername());
}

// 5. 更改用户名
user.setUsername("测试JPA");

// 开始事务
EntityManagerHelper.beginTransaction();
// 保存（JPA会自动更新变动过的字段）
dao.update(user);

// 提交事务真正保存实体到数据库
EntityManagerHelper.commit();

// 6. 查找所有年龄为20的用户，从第1个开始获取（第0个是第一条记录）
result = dao.findByAge(20, 1);

for(Student o : result) {
    System.out.println(o.getId());
    System.out.println(o.getUsername());
}

// 7. 根据 ID 查找
user = dao.findById(2);

System.out.println(user.getUsername());

// 8. 删除
// 开始事务
EntityManagerHelper.beginTransaction();
// 保存（JPA会自动更新变动过的字段）
dao.delete(user);

// 提交事务真正保存实体到数据库
EntityManagerHelper.commit();
}
}
```

注意：代码中凡是涉及到数据库修改的部分（增加和更新以及删除）都用到了事务，这样做的好处是如果操作失败（例如数据库故障），所有的值都会返回到原始状态而避免出现数据不一致的情况。例如大家常说的转帐操作，给账户 A 减去 100，然后给 B 加上 100，如果不放在事务中执行的话，很可能出现了 A 减去了 100，然而 B 却没得到 100 的中间状态，而用事务执行可以避免这种情况，如果给 B 加 100 失败的话，那么 A、B 的账户状态都会回到未转帐之前的值，避免坏数据的产生。如果不加事务的话，数据根本**无法**保存进数据库中！

选择菜单 **Run > Run**，就可以执行这个测试类，所得到的输出信息如下所示：

```

Hibernate: insert into test.student (age, password, username) values
(?, ?, ?)
Hibernate: select student0_.id as id0_, student0_.age as age0_,
student0_.password as password0_, student0_.username as username0_ from
test.student student0_
1
学生1
2
学生2
3
张三
17
hellojpa test
11
spring hibernate 标注事务测试
log4j:WARN No appenders could be found for logger
(org.hibernate.cfg.annotations.Version).
log4j:WARN Please initialize the log4j system properly.
2008-3-11 17:39:40 dao.EntityManagerHelper log
信息: saving Student instance
2008-3-11 17:39:40 dao.EntityManagerHelper log
信息: save successful
2008-3-11 17:39:40 dao.EntityManagerHelper log
信息: finding all Student instances
2008-3-11 17:39:41 dao.EntityManagerHelper log
信息: saving Student instance
2008-3-11 17:39:41 dao.EntityManagerHelper log
信息: save successful
Hibernate: update test.student set age=?, password=?, username=? where
id=?
2008-3-11 17:39:41 dao.EntityManagerHelper log
信息: finding Student instance with property: age, value: 20
Hibernate: select student0_.id as id0_, student0_.age as age0_,
student0_.password as password0_, student0_.username as username0_ from
test.student student0_ where student0_.age=?
17

```

测试JPA

2008-3-11 17:39:41 dao.EntityManagerHelper log

信息: finding Student instance with id: 2

学生2

2008-3-11 17:39:41 dao.EntityManagerHelper log

信息: deleting Student instance

2008-3-11 17:39:41 dao.EntityManagerHelper log

信息: delete successful

Hibernate: delete from test.student where id=?

。测试后的数据库中记录如图 13.16 所示。

id	username	password	age
1	学生1	1234	20
3	张三	1234	100
17	测试JPA	jpa password	20
11	spring hibernate 标注事务测试	密码	200

图 13.16 运行后的数据记录

测试代码看起来相对比较长,这是因为完整展示了大部分的操作功能,每一步的功能已经在代码中做了详细的注释了。运行时候产生的日志信息显示每个操作都会产生对应的 SQL 语句,最后通过 JDBC 在数据库中执行,展示了 JPA 的底层工作过程。

13.4 JPA 工具高级部分

本节我们将介绍 JPA 工具的高级部分,包括特定的透视图和一对多代码生成等功能。

13.4.1 MyEclipse Java Persistence Perspective 透视图

MyEclipse 提供了专门用于 JPA 开发的透视图: **MyEclipse Java Persistence**。在这个透视图上提供了一系列的工具来辅助 JPA 的开发过程。如何切换透视图请参考 [3.1.3 透视图 \(Perspective\) 切换器](#) 一节的内容。图 13.17 显示了透视图的外观。这个透视图将数据库浏览器和 Java 代码浏览合并起来,便于核对 Java 代码执行完毕后的数据库记录;并在普通的大纲视图之外提供了 **JPA Outline** (JPA 大纲) 视图,当打开实体类的时候可以在此视图上点击类或者属性快速的在 **JPA Details** 视图中进行编辑。这个透视图的主要视图有: **DB Browser** (数据库浏览器), **Package Explorer** (包浏览器), **Table/Object Info** (表/对象信息), **Problems** (问题), **JPA Outline** (JPA 视图), **JPA Details** (JPA 详细信息), **Outline** (大纲)。如果要进行 JPA 开发,通过这个透视图可以直接就进行反向工程向导,编辑实体类代码并进行测试,比较推荐采用这个透视图进行 JPA 相关的开发。

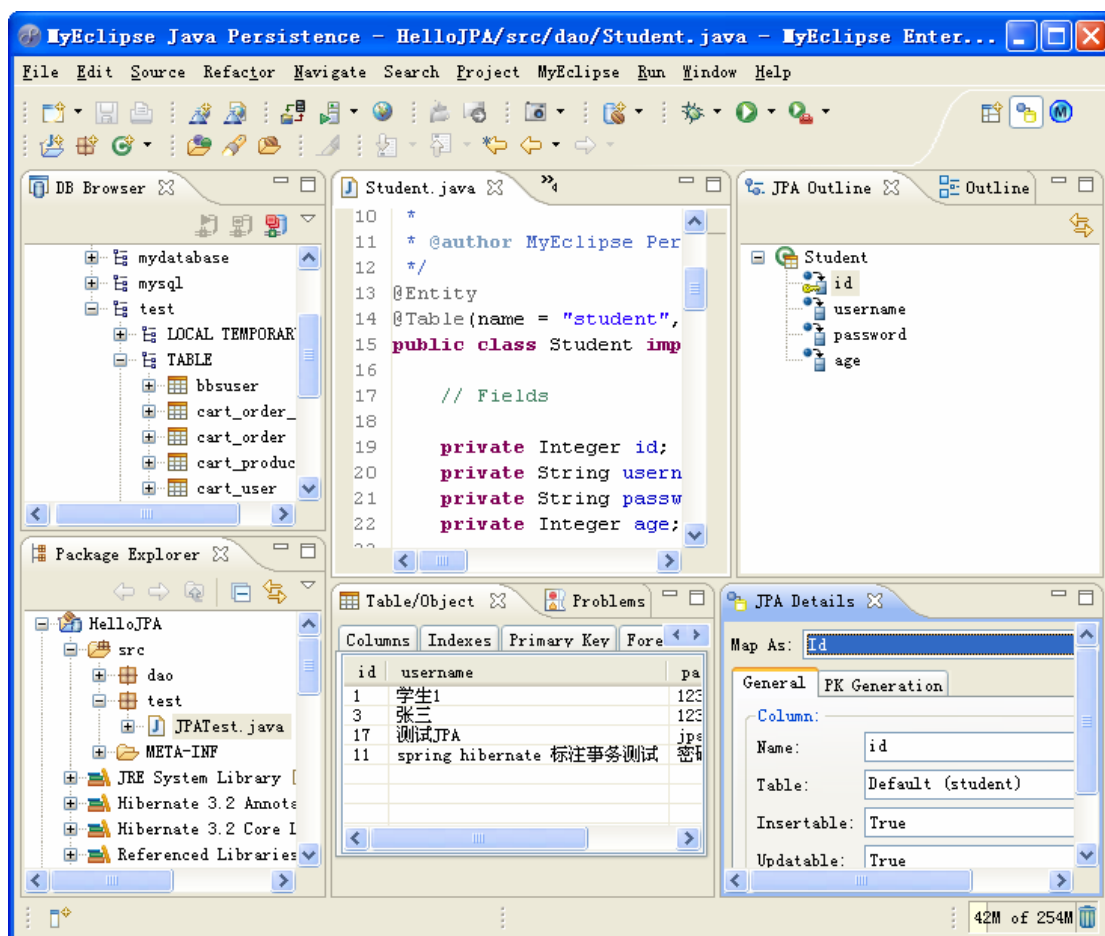


图 13.17 MyEclipse Java Persistence Perspective 透视图

13.4.2 JPA Details 视图

当在 **JPA Outline** 视图中选中不同的部分时，就可以在 **JPA Details** 视图中显示并编辑其相关的信息，图 13.18 是选中 **Student** 实体时候的相关选项。

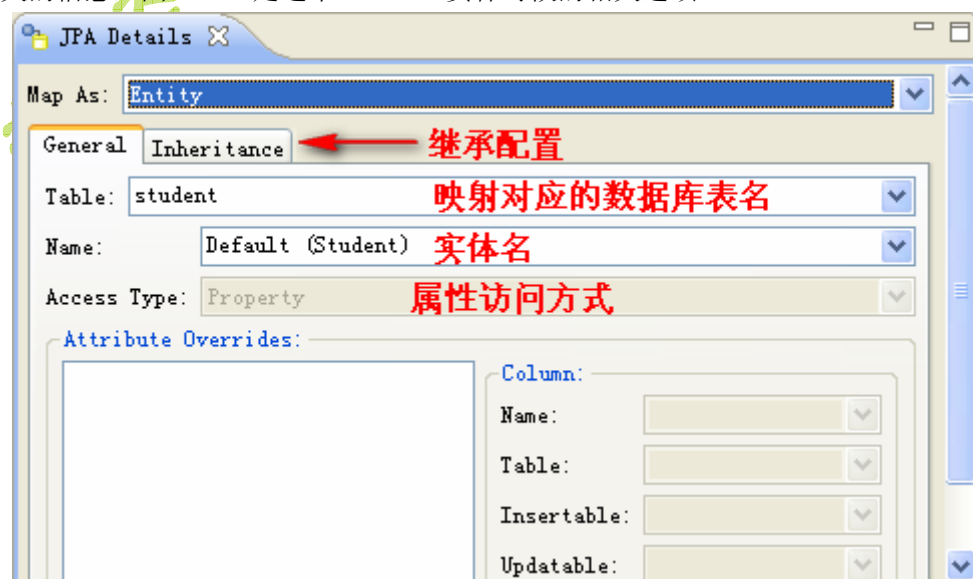


图 13.18 实体编辑选项

当选中 ID 属性时, 就可以设置实体的主键相关的属性了。图 13.19 左侧显示了和 id 属性相关的列配置信息, 包括对应的 Column Name (列名), Table (表名), Insertable (可插入), Updatable (可更新) 以及 Temporal (时间) 格式的可选配置方式 (不过主键一般很少有人用时间格式的字段来做); 通过下拉框, 可以很容易的对信息进行修改。图 13.19 右侧则显示了和主键生成器有关的配置, 复选框 Primary Key Generation 可以决定是否采用主键生成策略, 并在 Strategy 右侧下拉框中选择所需的策略, 并可以配置 Generator Name。另外如果使用 Table Generator 和 Sequence Generator 的时候, 都可以直接在这里进行配置, 无需再修改源代码。

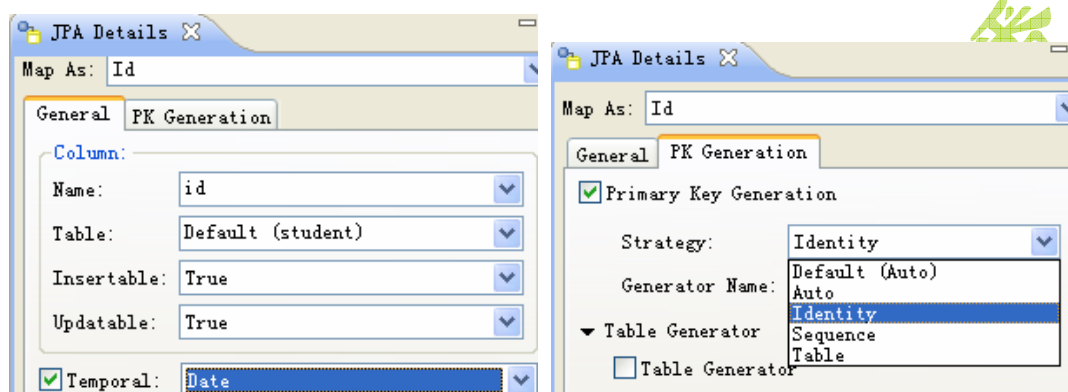


图 13.19 ID 属性的配置编辑器

当我们选中了实体的某个属性的时候, 也能对其属性进行编辑。除了和 ID 属性编辑时相同的那些选项之外, 还可以修改 Fetch Type (获取类型, 读取数据的方式, 默认情况下, JPA 持续性提供程序使用 EAGER: 这将要求持续性提供程序运行时必须主动获取数据。另一个选项是懒惰加载: FetchType.LAZY: 这将提示持续性提供程序在首次访问数据 (如果可以) 时应不急于获取数据) 以及 Optional (是否可选, 默认情况下, JPA 持续性提供程序假设所有 (非基元) 字段和属性的值可以为空)。

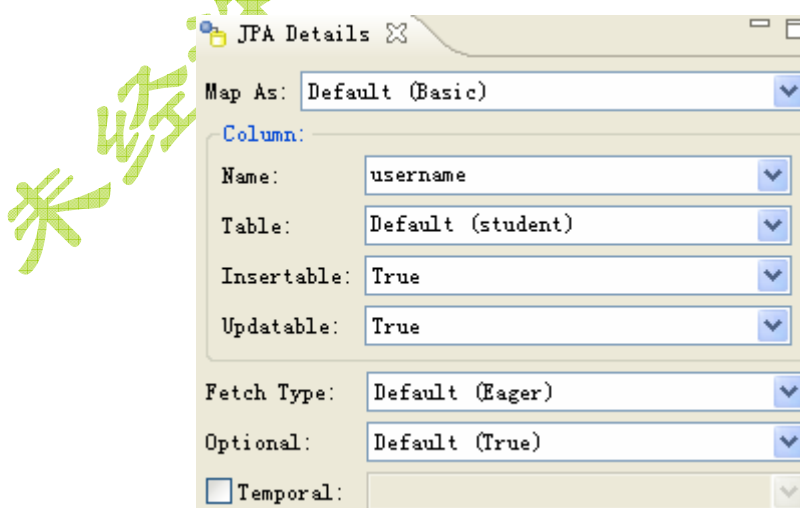


图 13.20 普通属性的配置编辑器

除此之外, 当编写有一对多等属性时, 都有相应的编辑选项, 和数据库相关的部分还可

以让用户直接选择数据库中相关联的表或者字段，这样读者不用把所有的标注都死记硬背下来，只需要了解相关知识点就能方便的进行修改。

13.4.3 JPA 代码编辑辅助

在编辑实体类中的标注例如 `@Table`, `@Column` 时, MyEclipse 提供数据库中表名列表以及列名列表的编辑提示, 一般来说编辑时按下快捷键 `Alt + /` 即可显示提示, 如果快捷键无效 (MyEclipse 6 下经常出现无法找到快捷键的情况) 可以选择菜单 `Edit > Content Assist > Default` 即可弹出。除此之外, 当用户输入的表名或者列名在数据库中不存在的时候, 还会以红色底线提示, 不过, 有时候拿到别人的代码而本机还没有建立好对应的数据库的时候, 或者说没有在 MyEclipse Database Explorer 中配置对应的数据库连接, 这会造成困惑, 这种错误不要理会就行了, 不会影响编译和运行过程。总的来说这些功能能对开发过程提供一些方便。

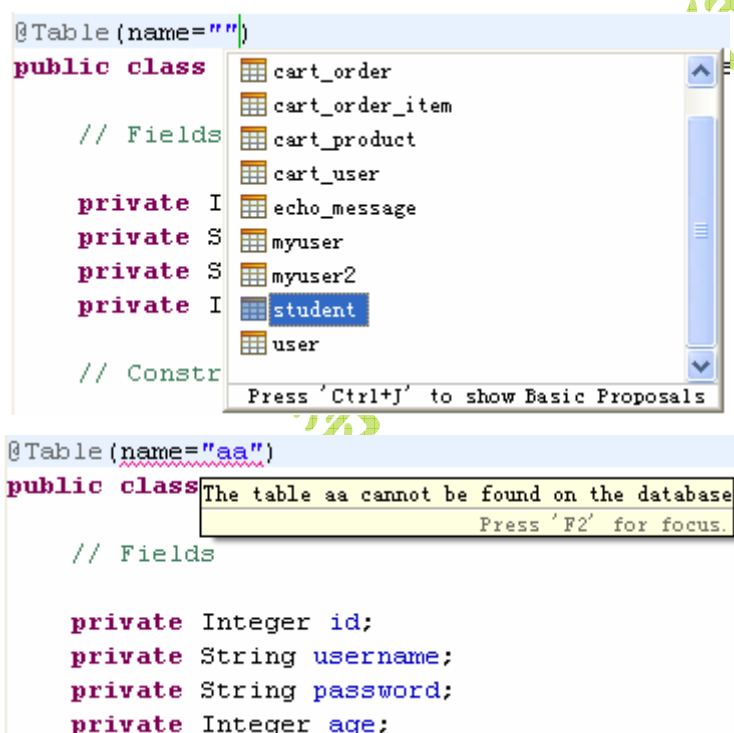


图 13.21 JPA 代码编辑辅助

13.4.4 生成一对多等复杂映射代码

在 13.3.5 一节中, 我们已经使用了最基础的反向工程功能来生成代码, 其实在它的后面还有两页可以设置, 在这里我们通过开发一个一对多的应用加以详细的展示和介绍。

首先我们要使用支持外键的数据库 (MySQL 5 目前还不直接支持外键, 虽然最终生成的 Java 代码修改数据库连接参数后也可以用于 MySQL), 在这里用 *MyEclipse Derby* 就行了, 它支持外键的创建, 并能被 MyEclipse 探测到。要建造的表相对比较简单, 一共有两个, 一个是 *Family* (家庭), 另一个是 *Member* (成员)。很显然, 一个家庭有多个成员, 这就是一对多的关系的实例, 最终的程序必须能够正确的处理在家庭中添加和删除成员 (例如死亡或者婚姻等等原因引起的人口变动)。

需要注意的是建表语句的主键都采用了自增的字段, 读者可以参考 [5.2 创建数据库表格](#)

一节的内容了解如果用 MySQL 建表该怎么写，并参考 4.2.7 编辑和执行 SQL 代码段一节的内容了解如何执行 SQL，而且要使用 MyEclipse 的反向工程功能生成一对多代码的话，必须放在 Derby 数据库中建表。

Family（家庭）表，一共两个字段，ID 和家庭名字，SQL 清单如下：

```
create table Family (
    id int primary key generated always as identity,
    familyName VARCHAR(20) not null
);
```

。 **Member**（家庭成员）表，一共 4 个字段，ID、名字、密码、年龄以及家庭编号（这是个外键，指向其所在的家庭，对应着家庭表中的 ID 值），SQL 代码如下：

```
create table Member(
    id int primary key generated always as identity,
    username varchar(200) NOT NULL,
    password varchar(20) NOT NULL,
    age int,
    familyId int references Family(id)
);
```

。当所有的表建立完毕后，我们就可以进行下面的工作了，来创建这个项目。

大部分的步骤都可以参考 13.3 创建 JPAHello 项目一节的内容，因此就不再赘述了。我们新建的这个项目名叫 *JPAOne2Many*，英文的 2 和 to 谐音，所以这个名字的意思就是 JPA 一对多，当然读者也可以用中文名字来建这个项目。然后添加 JPA 开发功能的时候，在 **Add JPA Capabilities** 对话框的第一页选择 *TopLink*（当然继续用 *Hibernate* 也是没任何问题的），第二页 *Driver*（数据库连接名）选择 *MyEclipse Derby*，*Catalog*（数据库名）默认配置是选择 *CLASSICCARS*，如果读者自己改了连接属性的用户名，例如改成了 *APP*，那就选择 *APP*，最后点击 **Finish** 按钮关闭对话框即可。

接下来就是用反向工程的方式来创建我们的 JPA 代码了。在选中 **DB Browser** 视图中的数据库连接 *MyEclipse Derby*，点击并展开数据库里面的树状表结构，单击选中表 **Family**。接着点击右键在上下文菜单中选择 **JPA Reverse Engineering...**，这将启动 **JPA Reverse Engineering** 向导。向导第一页的设置，**Java src folder** 需要选中 */JPAOne2Many/src*，然后其它选项保持和 13.3.5 节一样就行了。不要点击 **Finish** 按钮关闭向导，接着点击 **Next** 按钮来到第二页进行设置，这一页的截图如图 13.22 所示。下表列出了这些选项的详细解释：

选项	描述
Rev-eng settings file	这个文件包含了反向工程的配置和选项以供以后使用。点击 Setup... 按钮来选择现有的文件或者创建一个新的文件。如果找不到一个这样的配置文件的话向导将会自动创建此文件。
Custom rev-eng strategy	允许你指定一个自定义的反向工程策略类，这个类允许你用编程的方式来定义反向工程处理过程的各个方面。解即可。
Generate version and timestamp tags	如果启用，将会在将会在生成的实体类文件中添加 @Version 和 timestamp 标记出现（这可能是 MyEclipse 的一个 BUG，JPA 里面没有 TimeStamp 标记）。
Enable many-to-many	选中此复选框之后，会启用多对多表结构的探测。

detections	
Customized Type Mappings	允许你来指定一个自定义的 JDBC 类型到 Java 类型的转换，使用 Length、Scale、Precision 和 Nullability 作为精度控制对应原来的 JDBC 类型。

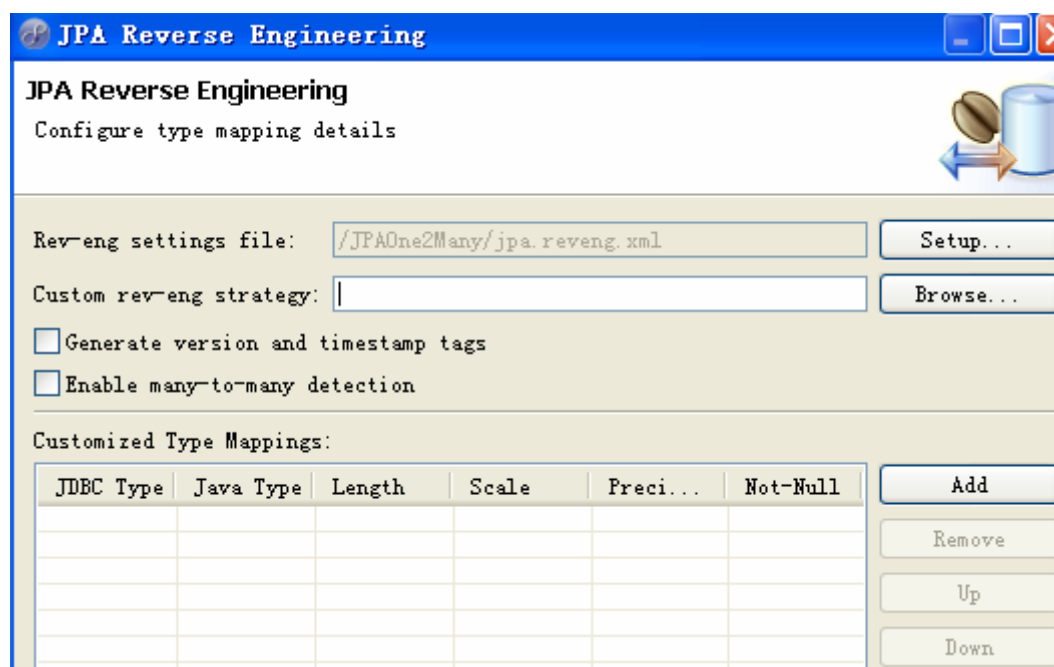


图 13.22 JPA 反向工程向导第 2 页

还好，在这一页什么设置都不需要做，点击 **Next** 按钮进入最后一页，设置反向工程中的多表映射。按照如图 13.23 那样进行设置即可。

图中红色的 1 所框中的内容列出了当前需要反向工程的表，图中显示出已经自动探测出了我们的 **Family** 表和 **Member** 表（因为先前的时候我们只选中了 **Family** 表来启动反向工程的向导）。

2 框中的两个复选框则指示是否包含引用到这个表和这个表引用的其它表，如果数据库支持外键的话，相关的表格会自动出现在 1 框中。遗憾的是，MySQL 还不支持外键，图中使用的是支持外键的 MyEclipse Derby，选中两个复选框后，相关联的表也就自动出现在 1 框中了，这样就可以生成一对多的映射代码了。

3 框中的则是生成通过外键关联到当前表格的关联表的尚未反向工程过的代码。

关于设置的详细信息，可以参考这张表格：

选项	描述
Class name	对应当前数据库表格的数据对象类的完整名称
Include referenced / referencing tables	包含反向工程时当前数据库表引用的表格以及其它引用到当前表的数据库表
Generate support for ListedTable(fk)->UnlistedTable and UnlistedTable(fk)->ListedTable	生成关联到当前表格的关联表的尚未反向工程过的代码，这些表在当前配置页面尚未被显示。

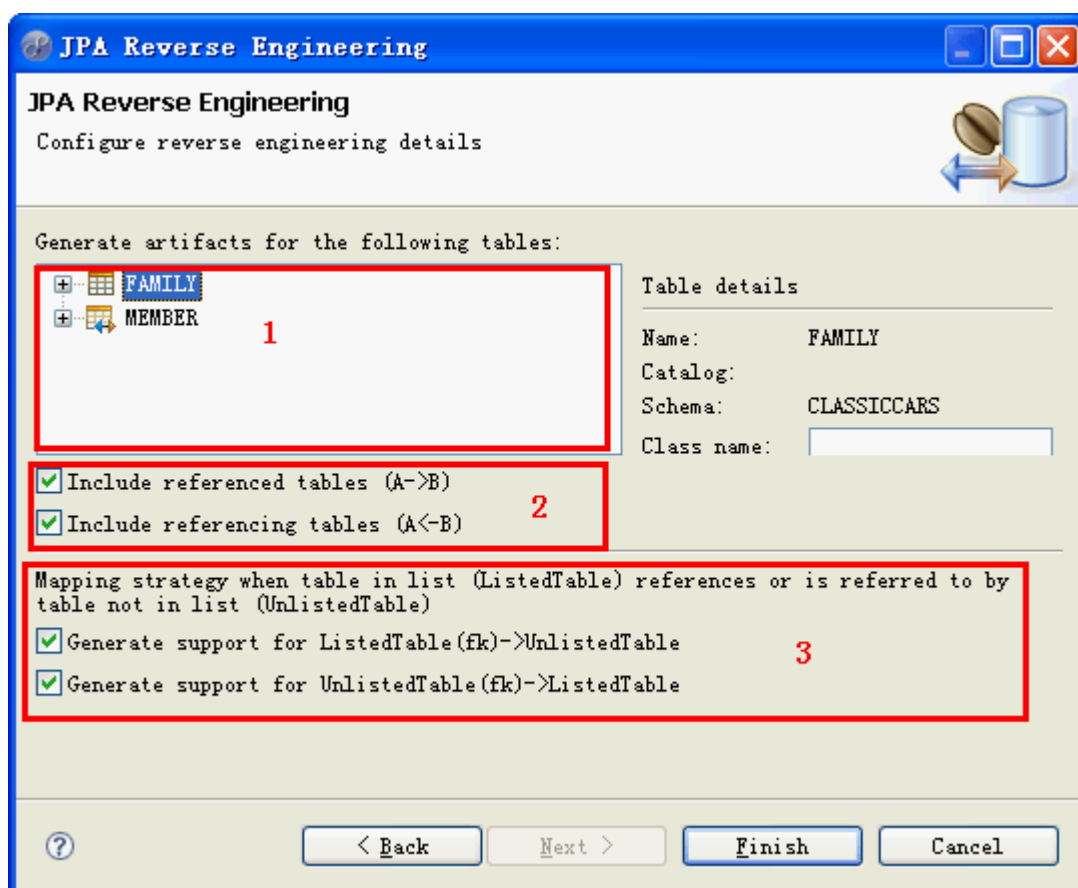


图 13.23 JPA 反向工程向导第 3 页

现在，我们只需要点击一下 **Finish** 按钮，所有的代码就都生成完毕了！包括必要的一对多的代码和对应的标注。然而，这些代码还需要稍加调整，加入主键生成器，来满足我们的需要。现在要做的就是分别对 **dao.Member** 类和 **dao.Family** 类的 ID 定义代码进行修改，它们都是非常相似的，可以通过 JPA Details 视图修改成如下所示：

```
// Property accessors
@Id
@Column(name = "ID", unique = true, nullable = false, insertable =
true, updatable = true)
@GeneratedValue(strategy=GenerationType.IDENTITY)
public Integer getId() {
    return this.id;
}
```

。粗斜体部分即显示了需要做的修改内容。为了节省篇幅，我们仅仅列出带有一对多和多对一标注的实体类的完整代码，其它的就不再这里列出了。

dao.Family

```
package dao;

import java.util.HashSet;
import java.util.Set;
import javax.persistence.*;
```

```

/**
 * Family entity.
 *
 * @author MyEclipse Persistence Tools
 */
@Entity
@Table(name = "FAMILY", schema = "CLASSICCARS", uniqueConstraints = {})
public class Family implements java.io.Serializable {

    // Fields

    private Integer id;
    private String familyname;
    private Set<Member> members = new HashSet<Member>(0);

    // Constructors

    /** default constructor */
    public Family() {
    }

    /** minimal constructor */
    public Family(Integer id, String familyname) {
        this.id = id;
        this.familyname = familyname;
    }

    /** full constructor */
    public Family(Integer id, String familyname, Set<Member> members) {
        this.id = id;
        this.familyname = familyname;
        this.members = members;
    }

    // Property accessors
    @Id
    @Column(name = "ID", unique = true, nullable = false, insertable =
true, updatable = true)
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    public Integer getId() {
        return this.id;
    }
}

```

```

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


    @Column(name = "FAMILYNAME", unique = false, nullable = false,
insertable = true, updatable = true, length = 20)
    public String getFamilyname() {
        return this.familyname;
    }

    public void setFamilyname(String familyname) {
        this.familyname = familyname;
    }

    @OneToMany(cascade = { CascadeType.ALL }, fetch = FetchType.LAZY,
mappedBy = "family")
    public Set<Member> getMembers() {
        return this.members;
    }

    public void setMembers(Set<Member> members) {
        this.members = members;
    }
}

```

 **dao.Member**

```

package dao;

import javax.persistence.*;

/**
 * Member entity.
 *
 * @author MyEclipse Persistence Tools
 */
@Entity
@Table(name = "MEMBER", schema = "CLASSICCARS", uniqueConstraints = {})
public class Member implements java.io.Serializable {

    // Fields

    private Integer id;

```



```

private Family family;
private String username;
private String password;
private Integer age;

// Constructors

/** default constructor */
public Member() {
}

/** minimal constructor */
public Member(Integer id, String username, String password) {
    this.id = id;
    this.username = username;
    this.password = password;
}

/** full constructor */
public Member(Integer id, Family family, String username, String
password,
    Integer age) {
    this.id = id;
    this.family = family;
    this.username = username;
    this.password = password;
    this.age = age;
}

// Property accessors
@Id
@Column(name = "ID", unique = true, nullable = false, insertable =
true, updatable = true)
@GeneratedValue(strategy=GenerationType.IDENTITY)
public Integer getId() {
    return this.id;
}

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

@ManyToOne(cascade = {}, fetch = FetchType.LAZY)
@JoinColumn(name = "FAMILYID", unique = false, nullable = true, insertable = true,

```

```

updatable = true)

    public Family getFamily() {
        return this.family;
    }

    public void setFamily(Family family) {
        this.family = family;
    }

    @Column(name = "USERNAME", unique = false, nullable = false,
insertable = true, updatable = true, length = 200)
    public String getUsername() {
        return this.username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    @Column(name = "PASSWORD", unique = false, nullable = false,
insertable = true, updatable = true, length = 20)
    public String getPassword() {
        return this.password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    @Column(name = "AGE", unique = false, nullable = true, insertable =
true, updatable = true)
    public Integer getAge() {
        return this.age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }
}

```

最后，让我们来测试并验证执行结果，测试类 **dao.Test** 源代码清单如下：

```
package dao;
```

```

public class Test {

    public static void main(String[] args) {
        FamilyDAO dao = new FamilyDAO();

        // 创建家庭
        Family family = new Family();
        family.setFamilyname("幸福老家");

        // 创建成员
        Member member = new Member();
        member.setAge(28);
        member.setFamily(family); // 设置成员所在家庭
        member.setUsername("Bean刘");
        member.setPassword("芝麻开门");

        // 添加成员
        family.getMembers().add(member);
        // 开始事务
        EntityManagerHelper.beginTransaction();

        // 保存数据
        dao.save(family);

        // 提交事务真正保存实体到数据库
        EntityManagerHelper.commit();
    }
}

```

。这段代码非常简单，就是创建一个家庭和一个成员，然后保存进数据库，需要注意的是一定要加事务处理的代码。可能有读者疑问说怎么没有单独的保存 *member* 的代码，这是因为在一对多中可以设置 **Cascade(层叠)** 操作属性，参考下面 **Family** 实体类中的标注片段：

```

@OneToMany(cascade = { CascadeType.ALL }, fetch = FetchType.LAZY,
mappedBy = "family")
public Set<Member> getMembers() {
    return this.members;
}

```

。注意到那个 *cascade = { CascadeType.ALL }* 了嘛？它和 **Hibernate** 中的概念是一致的，它的意思是当每次对家庭类进行操作的时候（包括增删改，就是直接对 *members* 这个变量的操作），顺便也会把所带的成员都给一并处理了，让人想起古时候的炒家，差不多就是这个道理。还有一些取值是可以选择分别操作，或者是只有保存或者删除时才会同步。*cascade* 的默认值是 **CascadeType** 的空数组。默认情况下，**JPA** 不会将任何持续性操作层叠到关联的目标。如果希望某些或所有持续性操作层叠到关联的目标，请将 *cascade* 设置为一个或多个 **CascadeType** 实例，其中包括：

1. ALL — 针对拥有实体执行的任何持续性操作均层叠到关联的目标。
2. MERGE — 如果合并了拥有实体，则将 merge 层叠到关联的目标。
3. PERSIST — 如果持久保存拥有实体，则将 persist 层叠到关联的目标。
4. REFRESH — 如果刷新了拥有实体，则 refresh 为关联的层叠目标。
5. REMOVE — 如果删除了拥有实体，则还删除关联的目标。

另外还有这里数据是懒惰加载的：fetch = FetchType.LAZY。

最后，让我们来运行一下看看结果，选择菜单 **Run > Run**，运行 **dao.Test** 类，可以看到 **Console** 视图输出如下：

```
2008-3-12 22:37:13 dao.EntityManagerHelper log
信息: saving Family instance
2008-3-12 22:37:13 dao.EntityManagerHelper log
信息: save successful
[TopLink Info]: 2008.03.12
10:37:12.359--ServerSession(10605044)--TopLink, version: Oracle
TopLink Essentials - 2.0 (Build b40-rc (03/21/2007))
[TopLink Info]: 2008.03.12
10:37:12.953--ServerSession(10605044)--file:/E:/workspace/JPAOne2Many
/bin/-JPAOne2ManyPU login successful
```

。没有报错，最后，查看数据库检查执行结果，如图所示，很好，执行的很正确。

ID	FAMILYNAME	Preview	Row Count
1	beansoft		
2	beansoft JPA测试		
3	幸福老家		

ID	USERNAME	PASSWORD	AGE	FAMILYID
1	Bean刘	芝麻开门	28	3

图 13.24 Family 表（左）和 Member 表（右）中的数据

这时候，如果读者把数据库切换到 MySQL 中（记得添加驱动程序，修改 persistence.xml 中的数据库连接参数，以及实体类中的 catalog 或者 schema 属性），这些代码依然是可以执行成功的，这就是 JPA 的优势所在。这也充分证明了使用 MyEclipse 来开发 JPA 的确是十分简单和强大的。

13.5 Spring 整合 JPA 开发

目前 Spring 是越来越流行了，而如何进行 Spring 整合 JPA 则是个不得不提的话题了。不过读者还是应该明白像上一章开头所讲述的那样，Spring 整合 Web 层和整合 ORM 层是分开的，所以读者不用在这里去掌握 Spring + Struts + JPA 之类的整合技术（如果出现了 ASM 出错之类的问题，请参考过去章节的内容解决即可）。和 Spring 一章内容所介绍整合 Hibernate 时候的方式一样，Spring 整合 JPA 也有大致 3 种方式，在此我们就不打算一一做介绍了，因为 MyEclipse 已经完整提供了整合代码的生成，不用手工编写任何代码了。当然，必要的代码调整还是需要的。

我们的步骤包括：

1. 创建一个添加了 JPA Capabilities 的项目
2. 给项目加入 Spring Capabilities
3. 从数据库反向工程生成 JPA 实体和 Spring DAO 代码到我们的项目中

注意：如果添加顺序是反过来的（先添加 Spring first，再添加 JPA），也能产生同样的结果。

13.5.1 添加 Spring 开发功能

大部分的步骤都可以参考 [13.3 创建 JPAHello 项目](#) 一节的内容，因此就不再赘述了。首先需要我们创建一个 Java 或者 Web 项目，名为 *JPASpring*。接着首先添加 JPA 开发功能，完成后再添加 Spring 开发功能，顺序千万不好搞错了哦！

在参考 [13.3.3 添加 JPA Capabilities 到现有项目](#) 一节添加完 JPA Capabilities 后（不要忘了调整 *persistence.xml* 中的连接字符串，并添加方言支持，如果是 Toplink 则不需要进行这样的设置），选择菜单 **MyEclipse > Project Capabilities > Add Spring Capabilities ...** 来启动 **Add Spring Capabilities** 向导，如图 13.25 所示。注意选中 **Add Spring-JAP support** 复选框。当然读者如果是 Web 项目的话，请参考第 11 章的内容，可能还需要选中 **Spring 2.0 Web Libraries** 类库并设置复制所有 JAR 文件到 */WebRoot/WEB-INF/lib* 目录下。接着点击 **Next** 按钮进入第二页的设置 Spring 配置文件，这一页保持默认值即可，点击 **Next** 按钮进入最后一页的设置。第三页的设置如图 13.26 所示，选中 **Enable annotation-driven transaction support** 复选框后，就可以在将来的类里面使用 *@Transactional* 的事务标注了。最后点击 **Finish** 按钮关闭对话框后，即可完成添加 Spring 开发功能的向导。

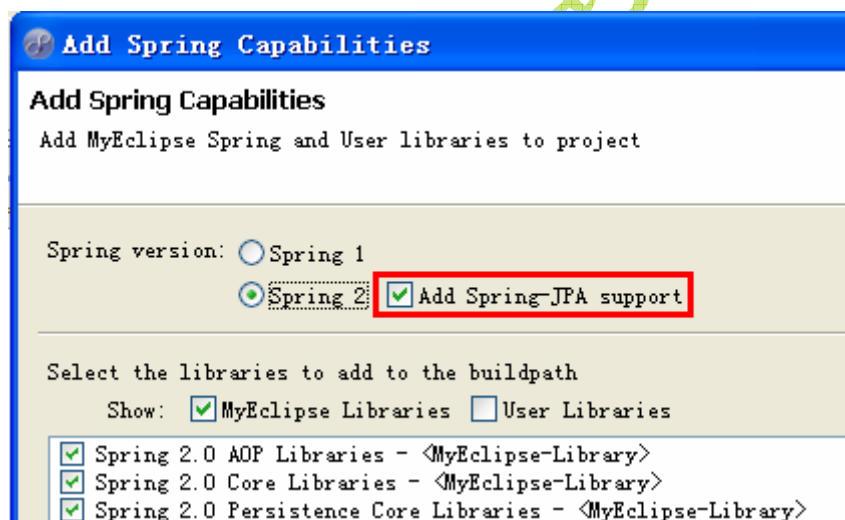


图 13.25 添加 Spring 开发功能第一页

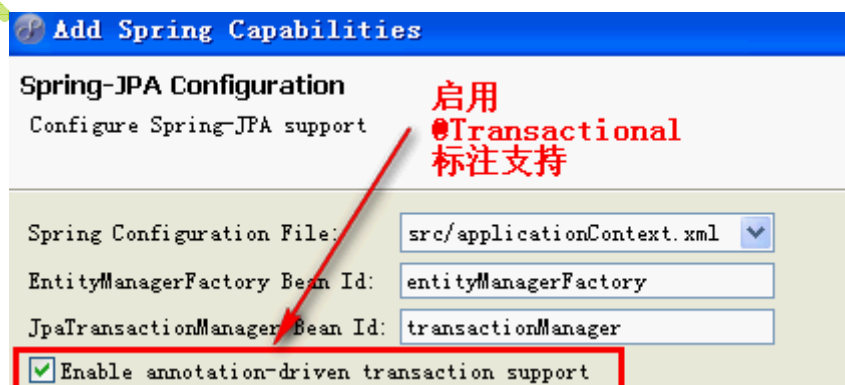


图 13.26 添加 Spring 开发功第三页：JPA 事务标注支持

当这些操作都完成后，我们就可以看到所生成的 Spring 配置文件的内容了，这是文件 `applicationContext.xml` 的代码清单：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.0.xsd"
       xmlns:tx="http://www.springframework.org/schema/tx">

    <bean id="entityManagerFactory"
          class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean"
    >

        <property name="persistenceUnitName" value="JPASpringPU" />
    </bean>

    <bean id="transactionManager"
          class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory"
            ref="entityManagerFactory" />
    </bean>

    <tx:annotation-driven transaction-manager="transactionManager" />
</beans>
```

。读者可以和第 9 章 Spring 开发中和 Hibernate 整合部分进行对比，会发现其实需要配置的内容都是一样的，无非一个是 `HibernateSessionFactory + transactionManager`，另一个是 `EntityManagerFactoryBean + transactionManager`。读者也可以用非标注之外的配置方式进行，如那一章所述还有 3 种配置的方式。这里面的两个 bean 中，`entityManagerFactory` 用来获取实体管理器（通过属性 `persistenceUnitName` 和配置文件 `persistence.xml` 中的持久化单元名字 `JPASpringPU` 相对应），而 `transactionManager` 则用来管理 JPA 的事务。`tx:annotation-driven` 标记则表明了当前的项目支持通过 Spring 的 `@Transactional` 标注来给类自动加入 JPA 事务管理功能。

13.5.2 从数据库反向工程生成实体和 Spring DAO

做好了准备工作后，就可以通过反向过程向导来生成实体类和基于 Spring 的 DAO 了。参考 13.3.5 使用反向工程快速生成 JPA 实体类和 DAO 一节的内容进行操作，唯一不同的就是第一页的选项需要略作修改。读者需要按照图 13.27 的方式进行选择：

1. 建议选中复选框 **Generate Java interfaces**，这样就可以生成 DAO 的接口，而无需再对 Spring 配置文件进行修改，否则就要参考 10.5.2.5 用 Spring 2.0 的 `@Transactional` 标注解决事务提交问题（最佳方案）一节的内容，设置 `<tx:annotation-driven transaction-manager="transactionManager" proxy-target-class="true"/>` 里面的代理目

标类的属性;

2. 选中 **DAO type** 里面的单选钮 **Spring DAO**, 这样才能生成基于 Spring 的继承自 *JpaDaoSupport* 的 DAO 类。最后点击 **Finish** 按钮后, 所有的实体类以及 DAO 的代码就都生成了。当然如果读者还需要进行一对多的映射代码生成等功能, 还可以点击 **Next** 按钮进行后续的设置。

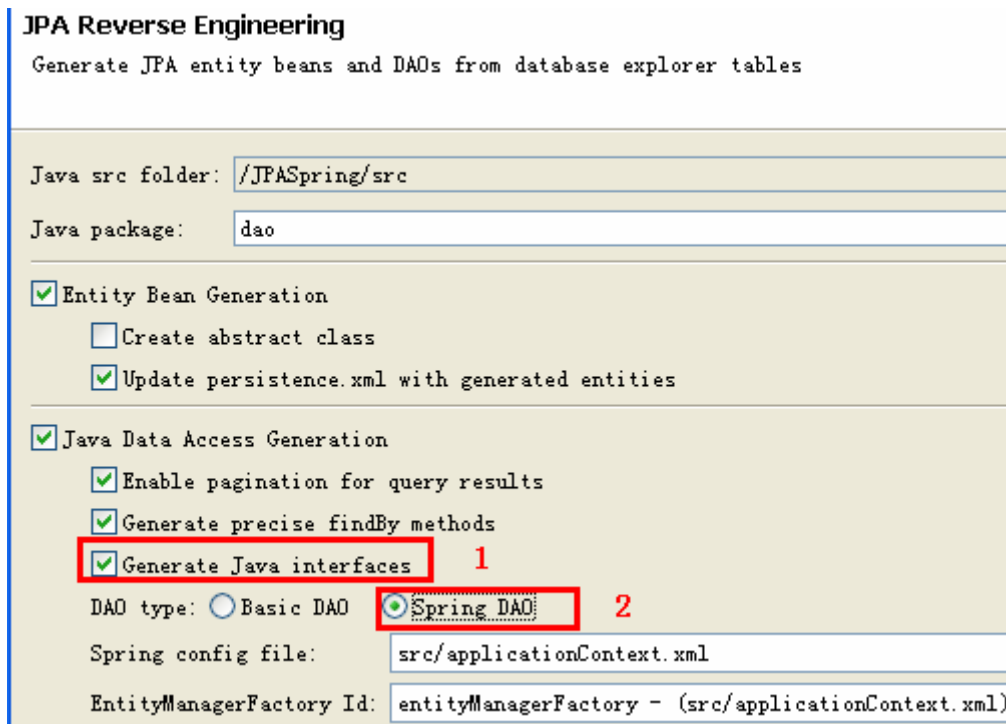


图 13.27 反向工程向导 Spring DAO 生成配置

最后我们所得到的项目文件结构如图 13.28 所示, 可以看到已经生成了所需的实体类 *dao.Student*, 以及 DAO 的接口和实现类: *IStudentDAO* 和 *StudentDAO*。这些类中的代码功能, 包括分页等等都是和之前未加 Spring 时候一样的, 用法也是很类似的。当然, 不要忘了对实体类 *Student* 里面的 ID 生成器进行调整, 使之成为 *IDENTITY*, 否则代码也是运行不了的。第二个要修改的类就是给 *StudentDAO* 加入 *@Transactional* 标注, 使它处于 JPA 事务管理器的保护之下。

注意: 如果不加此标注, 数据的改动将无法同步到数据库中去。另外, 此标注也可以加入到服务层里面, 此时就不需要加到 DAO 里面了, 当然此服务层必须配置成 Spring 的 bean。如果所有的地方都不加 *@Transactional* 标注, 也是可以的, 只不过那样就得手工编码来调用事务了, 一般很少有人会那样做。下面是加标注的示意代码:

Spring 配置文件中:

```
<bean id="studentManager" class="StudentManager">
    <property name="dao">
        <ref local="StudentDAO" />
    </property>
</bean>
```

Bean 类:

```
@Transactional
```

```

public class StudentManager implements IStudentManager {
    .... 其它业务方法
    /** 用户管理 DAO */
    private dao.StudentDAO dao;
    public dao.StudentDAO getDao() {
        return dao;
    }

    public void setDao(dao.StudentDAO dao) {
        this.dao = dao;
    }
}

```

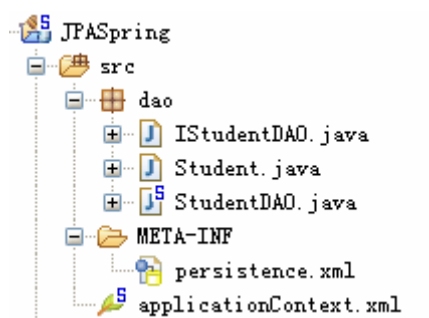


图 13.28 JPA+Spring 项目的目录结构

现在，我们就将经过调整的两个类的代码列出，供读者对比，修改过的代码以粗斜体的格式显示。

 **dao.Student** 的代码清单

```

package dao;

import javax.persistence.*;

/**
 * Student entity.
 *
 * @author MyEclipse Persistence Tools
 */
@Entity
@Table(name = "student", catalog = "test", uniqueConstraints = {})
public class Student implements java.io.Serializable {

    // Fields

    private Integer id;
    private String username;
    private String password;
    private Integer age;

```

```

// Constructors

/** default constructor */
public Student() {
}

/** minimal constructor */
public Student(Integer id, String username, String password) {
    this.id = id;
    this.username = username;
    this.password = password;
}

/** full constructor */
public Student(Integer id, String username, String password, Integer
age) {
    this.id = id;
    this.username = username;
    this.password = password;
    this.age = age;
}

// Property accessors
@Id
@Column(name = "id", unique = true, nullable = false, insertable =
true, updatable = true)
@GeneratedValue(strategy=GenerationType.IDENTITY)
public Integer getId() {
    return this.id;
}

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

@Column(name = "username", unique = false, nullable = false,
insertable = true, updatable = true, length = 200)
public String getUsername() {
    return this.username;
}

public void setUsername(String username) {
    this.username = username;
}

```

```

    @Column(name = "password", unique = false, nullable = false,
insertable = true, updatable = true, length = 20)
    public String getPassword() {
        return this.password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    @Column(name = "age", unique = false, nullable = true, insertable =
true, updatable = true)
    public Integer getAge() {
        return this.age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }
}

```

dao.StudentDAO 的代码清单

```

package dao;

import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceException;
import javax.persistence.Query;
import org.springframework.context.ApplicationContext;
import org.springframework.orm.jpa.JpaCallback;
import org.springframework.orm.jpa.support.JpaDaoSupport;
import org.springframework.transaction.annotation.Transactional;

/**
 * A data access object (DAO) providing persistence and search support
 for
 * Student entities. Transaction control of the save(), update() and
 delete()
 * operations can directly support Spring container-managed transactions
 or they
 * can be augmented to handle user-managed Spring transactions. Each of
 these

```

```

* methods provides additional information for how to configure it for
the
* desired type of transaction control.
*
* @see dao.Student
* @author MyEclipse Persistence Tools
*/
@Transactional
public class StudentDAO extends JpaDaoSupport implements IStudentDAO {
    // property constants
    public static final String USERNAME = "username";
    public static final String PASSWORD = "password";
    public static final String AGE = "age";

    /**
     * Perform an initial save of a previously unsaved Student entity.
All
     * subsequent persist actions of this entity should use the #update()
     * method. This operation must be performed within the a database
     * transaction context for the entity's data to be permanently saved
to the
     * persistence store, i.e., database. This method uses the
     * {@link javax.persistence.EntityManager#persist(Object)
EntityManager#persist}
     * operation.
     * <p>
     * User-managed Spring transaction example:
     *
     * <pre>
     * TransactionStatus txn = txManager
     * .getTransaction(new DefaultTransactionDefinition());
     * StudentDAO.save(entity);
     * txManager.commit(txn);
     * </pre>
     *
     * @see <a href =
     *
     "http://www.myeclipseide.com/documentation/quickstarts/jpaspring#cont
ainermanaged">Spring
     *     container-managed transaction examples</a>
     * @param entity
     *     Student entity to persist
     * @throws RuntimeException
     *     when the operation fails

```

```

    */
    public void save(Student entity) {
        logger.info("saving Student instance");
        try {
            getJpaTemplate().persist(entity);
            logger.info("save successful");
        } catch (RuntimeException re) {
            logger.error("save failed", re);
            throw re;
        }
    }

    /**
     * Delete a persistent Student entity. This operation must be performed
     * within the a database transaction context for the entity's data
     * to be
     * permanently deleted from the persistence store, i.e., database.
     * This
     * method uses the
     * {@link javax.persistence.EntityManager#remove(Object)
     * EntityManager#delete}
     * operation.
     * <p>
     * User-managed Spring transaction example:
     *
     * <pre>
     * TransactionStatus txn = txManager
     *     .getTransaction(new DefaultTransactionDefinition());
     * StudentDAO.delete(entity);
     * txManager.commit(txn);
     * entity = null;
     * </pre>
     *
     * @see <a href =
     *
     * "http://www.myeclipseide.com/documentation/quickstarts/jpaspring#cont
     * ainermanaged">Spring
     *     container-managed transaction examples</a>
     * @param entity
     *     Student entity to delete
     * @throws RuntimeException
     *     when the operation fails
     */
    public void delete(Student entity) {

```



```

        logger.info("deleting Student instance");
        try {
            entity = getJpaTemplate().getReference(Student.class,
                entity.getId());
            getJpaTemplate().remove(entity);
            logger.info("delete successful");
        } catch (RuntimeException re) {
            logger.error("delete failed", re);
            throw re;
        }
    }

    /**
     * Persist a previously saved Student entity and return it or a copy
     of it
     * to the sender. A copy of the Student entity parameter is returned
     when
     * the JPA persistence mechanism has not previously been tracking the
     * updated entity. This operation must be performed within the a
     database
     * transaction context for the entity's data to be permanently saved
     to the
     * persistence store, i.e., database. This method uses the
     * {@link javax.persistence.EntityManager#merge(Object)
     EntityManager#merge}
     * operation.
     * <p>
     * User-managed Spring transaction example:
     *
     * <pre>
     * TransactionStatus txn = txManager
     *     .getTransaction(new DefaultTransactionDefinition());
     * entity = StudentDAO.update(entity);
     * txManager.commit(txn);
     * </pre>
     *
     * @see <a href =
     *
     "http://www.myeclipseide.com/documentation/quickstarts/jpaspring#cont
     ainermanaged">Spring
     *     container-managed transaction examples</a>
     * @param entity
     *     Student entity to update
     * @returns Student the persisted Student entity instance, may not

```

```

be the
    *           same
    * @throws RuntimeException
    *           if the operation fails
    */
    public Student update(Student entity) {
        logger.info("updating Student instance");
        try {
            Student result = getJpaTemplate().merge(entity);
            logger.info("update successful");
            return result;
        } catch (RuntimeException re) {
            logger.error("update failed", re);
            throw re;
        }
    }

    public Student findById(Integer id) {
        logger.info("finding Student instance with id: " + id);
        try {
            Student instance = getJpaTemplate().find(Student.class, id);
            return instance;
        } catch (RuntimeException re) {
            logger.error("find failed", re);
            throw re;
        }
    }

    /**
     * Find all Student entities with a specific property value.
     *
     * @param propertyName
     *           the name of the Student property to query
     * @param value
     *           the property value to match
     * @param rowStartIdxAndCount
     *           Optional int varargs. rowStartIdxAndCount[0] specifies
the the
     *           row index in the query result-set to begin collecting the
     *           results. rowStartIdxAndCount[1] specifies the the
maximum
     *           number of results to return.
     * @return List<Student> found by query
     */

```

```

@SuppressWarnings("unchecked")
public List<Student> findByProperty(String propertyName,
    final Object value, final int... rowStartIdxAndCount) {
    logger.info("finding Student instance with property: " +
propertyName
        + ", value: " + value);
    try {
        final String queryString = "select model from Student model
where model."
            + propertyName + "= :propertyValue";
        return getJpaTemplate().executeFind(new JpaCallback() {
            public Object doInJpa(EntityManager em)
                throws PersistenceException {
                Query query = em.createQuery(queryString);
                query.setParameter("propertyValue", value);
                if (rowStartIdxAndCount != null
                    && rowStartIdxAndCount.length > 0) {
                    int rowStartIdx = Math.max(0,
rowStartIdxAndCount[0]);
                    if (rowStartIdx > 0) {
                        query.setFirstResult(rowStartIdx);
                    }

                    if (rowStartIdxAndCount.length > 1) {
                        int rowCount = Math.max(0,
rowStartIdxAndCount[1]);
                        if (rowCount > 0) {
                            query.setMaxResults(rowCount);
                        }
                    }
                }
                return query.getResultList();
            }
        });
    } catch (RuntimeException re) {
        logger.error("find by property name failed", re);
        throw re;
    }
}

public List<Student> findByUsername(Object username,
    int... rowStartIdxAndCount) {
    return findByProperty(USERNAME, username, rowStartIdxAndCount);
}

```

```

public List<Student> findByPassword(Object password,
    int... rowStartIdxAndCount) {
    return findByProperty(PASSWORD, password, rowStartIdxAndCount);
}

public List<Student> findByAge(Object age, int...
rowStartIdxAndCount) {
    return findByProperty(AGE, age, rowStartIdxAndCount);
}

/**
 * Find all Student entities.
 *
 * @param rowStartIdxAndCount
 *      Optional int varargs. rowStartIdxAndCount[0] specifies
the the
 *      row index in the query result-set to begin collecting the
 *      results. rowStartIdxAndCount[1] specifies the the
maximum
 *      count of results to return.
 * @return List<Student> all Student entities
 */
@SuppressWarnings("unchecked")
public List<Student> findAll(final int... rowStartIdxAndCount) {
    logger.info("finding all Student instances");
    try {
        final String queryString = "select model from Student model";
        return getJpaTemplate().executeFind(new JpaCallback() {
            public Object doInJpa(EntityManager em)
                throws PersistenceException {
                Query query = em.createQuery(queryString);
                if (rowStartIdxAndCount != null
                    && rowStartIdxAndCount.length > 0) {
                    int rowStartIdx = Math.max(0,
rowStartIdxAndCount[0]);
                    if (rowStartIdx > 0) {
                        query.setFirstResult(rowStartIdx);
                    }

                    if (rowStartIdxAndCount.length > 1) {
                        int rowCount = Math.max(0,
rowStartIdxAndCount[1]);
                        if (rowCount > 0) {

```

```

        query.setMaxResults(rowCount);
    }
}

return query.getResultList();
}

});
} catch (RuntimeException re) {
    logger.error("find all failed", re);
    throw re;
}
}

public static IStudentDAO
getFromApplicationContext(ApplicationContext ctx) {
    return (IStudentDAO) ctx.getBean("StudentDAO");
}
}

```

13.5.3 编写并运行测试代码

13.5.3.1 支持标注事务时的调试代码

最后，让我们新建测试类 **test.JPATest**，来尝试保存和更新数据：

```

package test;

import java.util.List;
import org.springframework.context.ApplicationContext;
package test;

import java.util.List;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
import dao.*;

public class JPATest {
    public static void main(String[] args) {
        ApplicationContext ctx =
            new
ClassPathXmlApplicationContext("applicationContext.xml");
        // 1.创建 DAO

```

```

        IStudentDAO dao = (IStudentDAO)ctx.getBean("StudentDAO");

        // 2.创建实体类
        Student user = new Student();
        user.setUsername("Spring JPA 标注测试");
        user.setPassword("1234");
        user.setAge(20);

        // 3. 保存
        dao.save(user);

        System.out.println("id=" + user.getId());

        // 4. 列出数据库中所有对象
        List<Student> result = dao.findAll();

        for(Student o : result) {
            System.out.println(o.getId());
            System.out.println(o.getUsername());
        }

        // 5. 更改用户名
        user.setUsername("测试 Spring JPA");

        // 6. 更新数据（不要用save()）
        dao.update(user);
    }
}

```

。这段代码非常简单，首先创建 **Spring** 容器，然后从中获取 **IStudentDAO** 的实例（这样做是因为 **Spring** 的代理机制默认是只支持接口的），然后用它来操作实体类。第九章的时候读者已经明白，**Spring** 的 AOP 机制会自动在我们调用 DAO 类的每个方法时，通过代理的方式打开事务和提交事务，这样，虽然我们已经去掉了 13.3.6 节中测试类中的事务操纵代码，然而最后数据依然还是能够保存和修改成功。

选择菜单 **Run > Run** 运行这段代码，然后查看数据库，可以看到刚刚加入并修改过的数据，如图 13.29 所示。

id	username	password	age
1	学生1	1234	20
3	张三	1234	100
29	测试 Spring JPA	1234	20
20	hellojpa test	jpa password	20
11	spring hibernate 标注事务测试	密码	200

Table structure for 'test'. 'student'

图 13.29 插入并更新后的数据

Console 视图中产生的输出则如下所示:

```
log4j:WARN No appenders could be found for logger
(org.springframework.context.support.ClassPathXmlApplicationContext).
log4j:WARN Please initialize the log4j system properly.
id=29
1
学生1
3
张三
29
测试 Spring JPA
20
hellojpa test
11
spring hibernate 标注事务测试
```

。这样，大家可以看到整个开发和调试过程的确是非常的简单。

13.5.3.2 不支持标注事务时的调试代码

在生成实体类和 DAO 的时候，如果读者没有选中加入支持标注的事务，或者虽然选中了，但是却没有在任何类中加入 `@Transactional` 标注，这时候，您需要考虑使用手工操作事务管理器的方式进行。其实关于手工操控事务的代码在生成的 `StudentDAO` 源代码的方法注释文档中已经写的比较清楚了，而一般这种开发模式也比较少。

注意：测试本节内容的时候，请务必删除任何类前面的 `@Transactional` 标注。

下面就是测试类的代码清单：

```
dao.JPATestNoAnnotation

package test;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.orm.jpa.JpaTransactionManager;
```

```

import org.springframework.transaction.TransactionStatus;
import
org.springframework.transaction.support.DefaultTransactionDefinition;

import dao.*;

public class JPATestNoAnnotation {
    public static void main(String[] args) {
        ApplicationContext ctx =
            new
ClassPathXmlApplicationContext("applicationContext.xml");

        // 1.创建 DAO
        IStudentDAO dao = (IStudentDAO)ctx.getBean("StudentDAO");
        JpaTransactionManager txManager = ((JpaTransactionManager)
ctx.getBean("transactionManager"));

        // 2.创建实体类
        Student user = new Student();
        user.setUsername("Spring JPA 无标注测试");
        user.setPassword("1234");
        user.setAge(20);

        // 3. 打开事务支持
        TransactionStatus txn = txManager
            .getTransaction(new DefaultTransactionDefinition());

        // 4. 保存
        dao.save(user);

        // 5. 提交事务
        txManager.commit(txn);

        System.out.println("id=" + user.getId());
    }
}

```

。运行后可以插入数据，当然不加事务代码是不行的。这也适用于需要在一次事务操作中完成对两个类进行修改的类似于银行转帐的操作。

13.6 小结

在本章简要讨论了 JPA 在 MyEclipse 中的开发方式，以及注意事项，总体而言，通过利用 MyEclipse 生成代码，以及使用标注，其开发过程要明显的比纯 XML 配置的 Hibernate 开发简单许多，这也是现在 JPA 模式越来越受到关注的原因。熟练的读者可以在 2 分钟内就完成所有代码的生成和调整。然而读者必须记住：JPA 必须用 JDK 1.5 或者更高版本才

能开发，因此，如果用的是低版本的一些服务器软件例如 Tomcat 4，Weblogic 8 及以下版本，因为它们只能运行在低版本的 JDK 上，那么您只能望 JPA 兴叹了。

13.7 参考资料

<http://www.hibernate.org/6.html> Hibernate JPA 实现下载地址，需要下载下列页面中的两个包：Core和EntityManager:

Package	Version	Release date	Category
Hibernate Core	3.2.6.ga	08.02.2008	Production Download
Hibernate EntityManager	3.3.1 GA	29.03.2007	Production Download

<https://glassfish.dev.java.net/downloads/persistence/JavaPersistence.html> TopLink 基础JPA包的下载地址，下载二进制文件列表中的一个（单个的 JAR 文件，一般是日期最新的那个），然后运行：

```
java -jar filename.jar
```

这将解压缩 README、许可文件以及 TopLink Essentials 库。而TopLink JPA的官方站点是在 <http://www.oracle.com/technology/global/cn/products/ias/toplink/JPA/index.html>。

<http://www.oracle.com/technology/global/cn/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html> JPA 批注参考

<http://dev.chinaitzhe.com/java/2007-12/119846030829349.html> J2SE综合——谈论泛型方法及动态的参数

http://e-docs.bea.com/kodo/docs41/full/html/ejb3_langref.html BEA JPQL 参考

<http://www.bea.com/kodo/> BEA 的 KODO 持久化框架

<http://openjpa.apache.org/> Apache OpenJPA

<http://java.sun.com/javaee/overview/faq/persistence.jsp> Sun JPA API FAQ

<https://glassfish.dev.java.net/nonav/javaee5/api/index.html?javax/persistence/package-summary.html> Glassfish JPA Javadoc

<http://java.sun.com/developer/technicalArticles/J2EE/jpa/> Sun JPA 入门教程（英文）

<http://www-128.ibm.com/developerworks/cn/java/j-annotate1/index.html> Tiger 中的注释，第 1 部分：向 Java 代码中添加元数据

<http://www-128.ibm.com/developerworks/cn/java/j-annotate2.html> Tiger 中的注释，第 2 部分：定制注释

<http://www.blogjava.net/windfree/archive/2006/12/12/87155.html> java元数据的学习