JPA

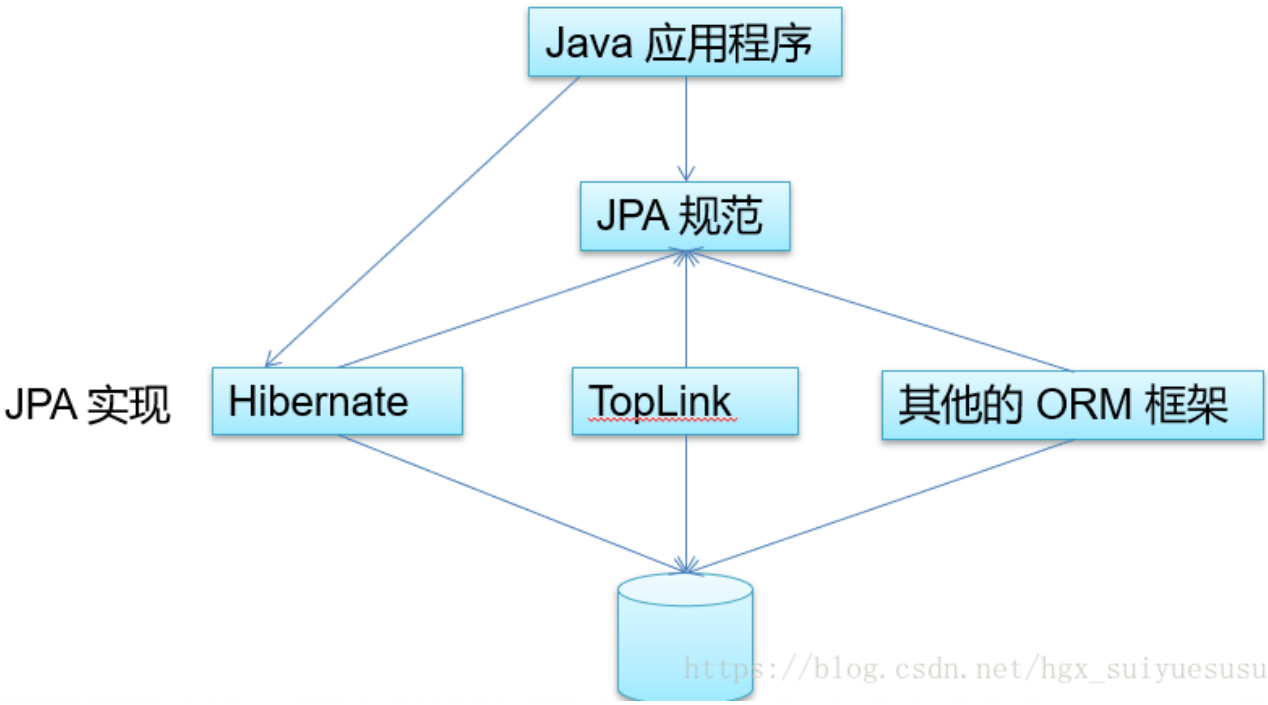
@Author:hanguixian

@Email:hn_hanguixian@163.com

一 JPA概述

JPA 是什么

- Java Persistence API：用于对象持久化的 API
- Java EE 5.0 平台标准的 ORM 规范，使得应用程序以统一的方式访问持久层



JPA和Hibernate的关系

- JPA 是 hibernate 的一个抽象（就像JDBC和JDBC驱动的关系）：JPA 是规范：JPA 本质上就是一种 ORM 规范，不是ORM 框架 —— 因为 JPA 并未提供 ORM 实现，它只是制订了一些规范，提供了一些编程的 API 接口，但具体实现则由 ORM 厂商提供实现
- Hibernate 是实现：Hibernate 除了作为 ORM 框架之外，它也是一种JPA 实现从功能上来说， JPA 是 Hibernate 功能的一个子集

JPA 的供应商

- **JPA 的目标之一是制定一个可以由很多供应商实现的 API，目前Hibernate 3.2+、TopLink 10.1+ 以及 OpenJPA 都提供了 JPA 的实现**

Hibernate

- JPA 的起始就是 Hibernate 的作者Hibernate
- 从 3.2 开始兼容 JPA

OpenJPA

- OpenJPA 是 Apache 组织提供的开源项目

TopLink

- TopLink 以前需要收费，如今开源了

JPA的优势

- 标准化: 提供相同的 API，这保证了基于JPA 开发的企业应用能够经过少量的修改就能够在不同的 JPA 框架下运行。
- 简单易用，集成方便: JPA 的主要目标之一就是提供更加简单的编程模型，在 JPA 框架下创建实体和创建 Java 类一样简单，只需要使用 javax.persistence.Entity 进行注释；JPA 的框架和接口也都非常简单
- 可媲美JDBC的查询能力: JPA的查询语言是面向对象的，JPA定义了独特的JPQL，而且能够支持批量更新和修改、JOIN、GROUP BY、HAVING 等通常只有 SQL 才能够提供的高级查询特性，甚至还能够支持子查询。
- 支持面向对象的高级特性: JPA 中能够支持面向对象的高级特性，如类之间的继承、多态和类之间的复杂关系，最大限度的使用面向对象的模型

JPA 包括 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（JPQL）：这是持久化操作中很重要的一个方面，通过面向对象而非面向数据库的查询语言查询数据，避免程序和具体的 SQL 紧密耦合。

二 JPA HelloWorld

使用JPA持久化对象的步骤

1.创建 persistence.xml, 在这个文件中配置持久化单元

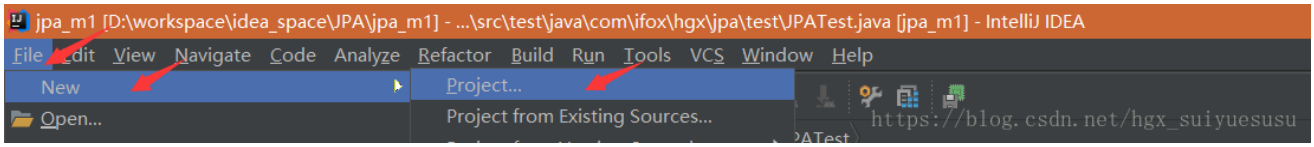
- 需要指定跟哪个数据库进行交互;
- 需要指定 JPA 使用哪个持久化的框架以及配置该框架的基本属性

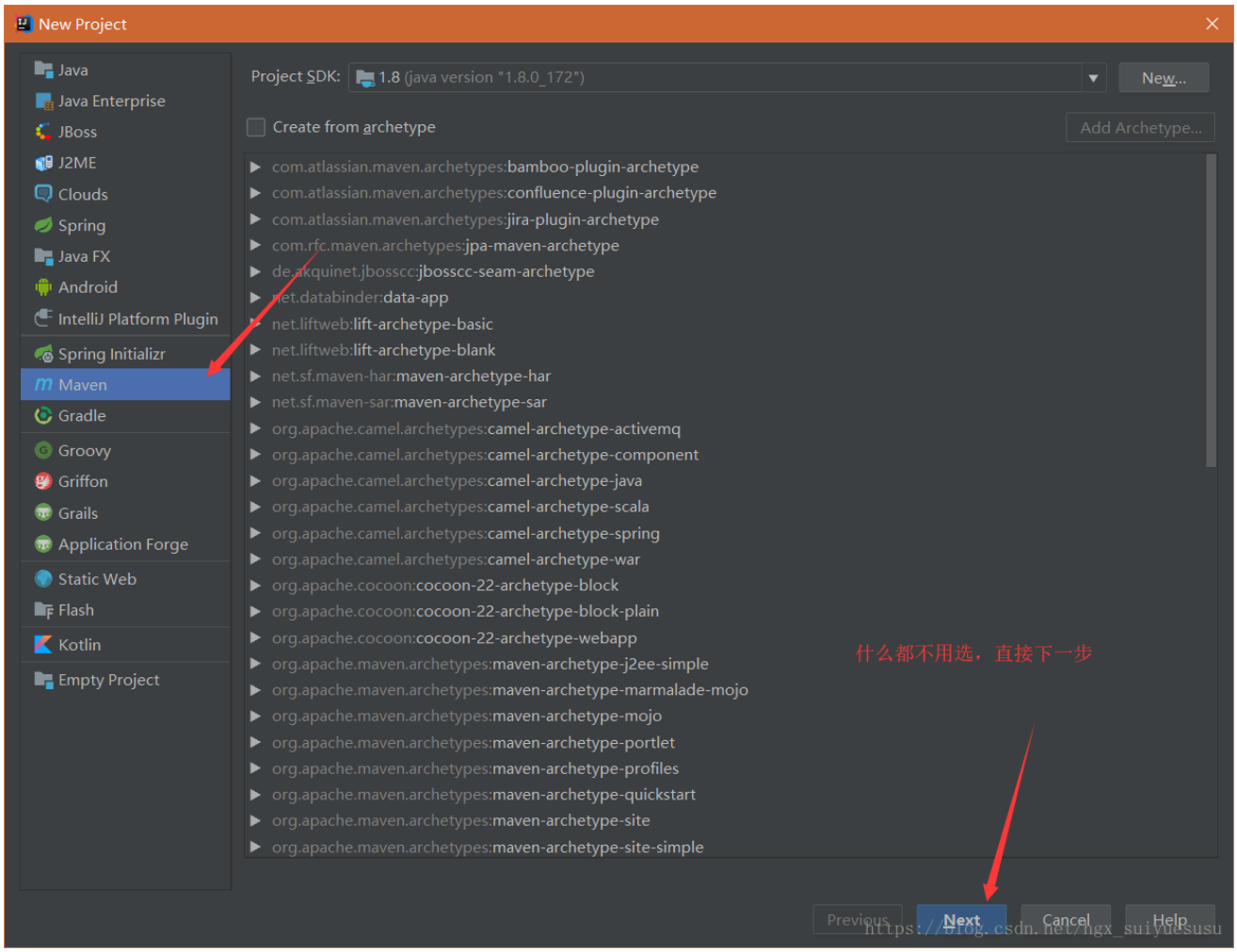
2.创建实体类, 使用 annotation 来描述实体类跟数据库表之间的映射关系.

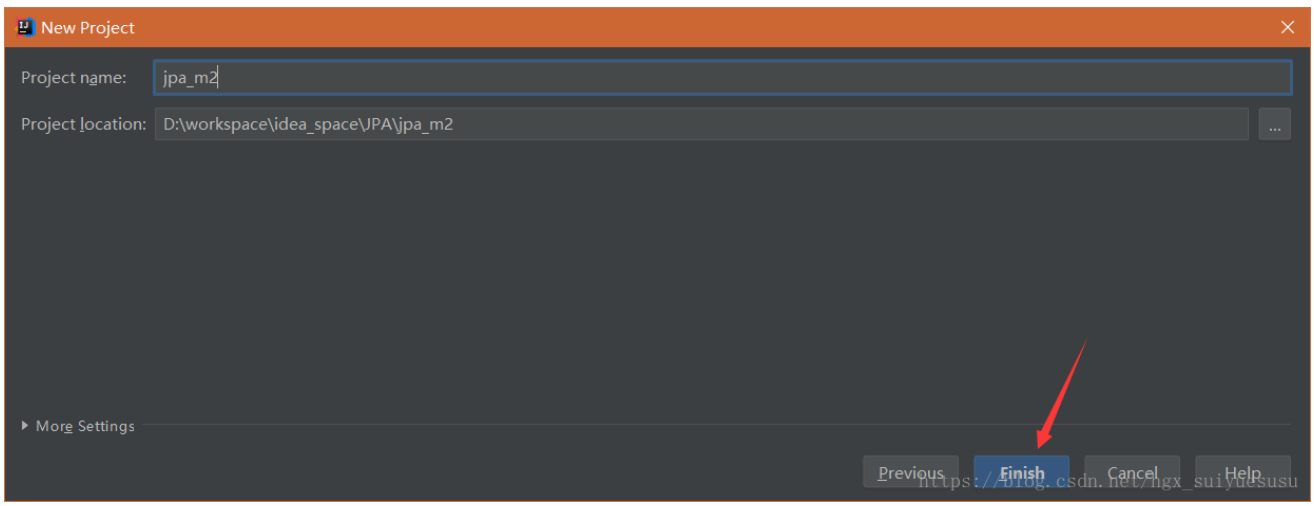
3.使用 JPA API 完成数据增加、删除、修改和查询操作

- 创建 EntityManagerFactory (对应 Hibernate 中的 SessionFactory)
- 创建 EntityManager (对应 Hibernate 中的Session)

在IntelliJ IDEA 下，创建JPA的Maven工程







- pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.ifox.hgx</groupId>
    <artifactId>jpa_m2</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-core</artifactId>
            <version>5.3.1.Final</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-entitymanager -->
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-entitymanager</artifactId>
            <version>5.3.1.Final</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.46</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/junit/junit -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

persistence.xml

- JPA 规范要求在类路径的 META-INF 目录下放置persistence.xml，文件的名称是固定的
- persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="2.1"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_1.xsd">
```

```
    <persistence-unit name="jpa_m2" transaction-type="RESOURCE_LOCAL">
        <!--配置使用什么 ORM 产品来作为 JPA 的实现 -->
        <!--5.2版本以后，无法使用HibernatePersistence-->
        <!--如果JPA项目中只有一个实现产品，可以配置不这个节点-->
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

        <!--添加持久化类-->
        <class>jpa.entity.Customer</class>

        <properties>
            <!-- 连接数据库的基本信息 -->
            <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql:///jpa"/>
            <property name="javax.persistence.jdbc.user" value="root"/>
            <property name="javax.persistence.jdbc.password" value="123456"/>

            <!-- 配置 JPA 实现产品的基本属性. 配置 hibernate 的基本属性 -->
            <property name="hibernate.format_sql" value="true"/>
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.hbm2ddl.auto" value="update"/>

        </properties>
    </persistence-unit>

</persistence>
```

实体类:Customer

```
package jpa.entity;

import javax.persistence.*;

@Table(name = "JPA_CUSTOMTERS")
@Entity
public class Customer {


    private Integer id ;
    private String lastName ;

    private String email ;
    private Integer age ;

//    GeneratedValue 生成方式:策略为 GenerationType.AUTO 自动选择
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Id
    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
```

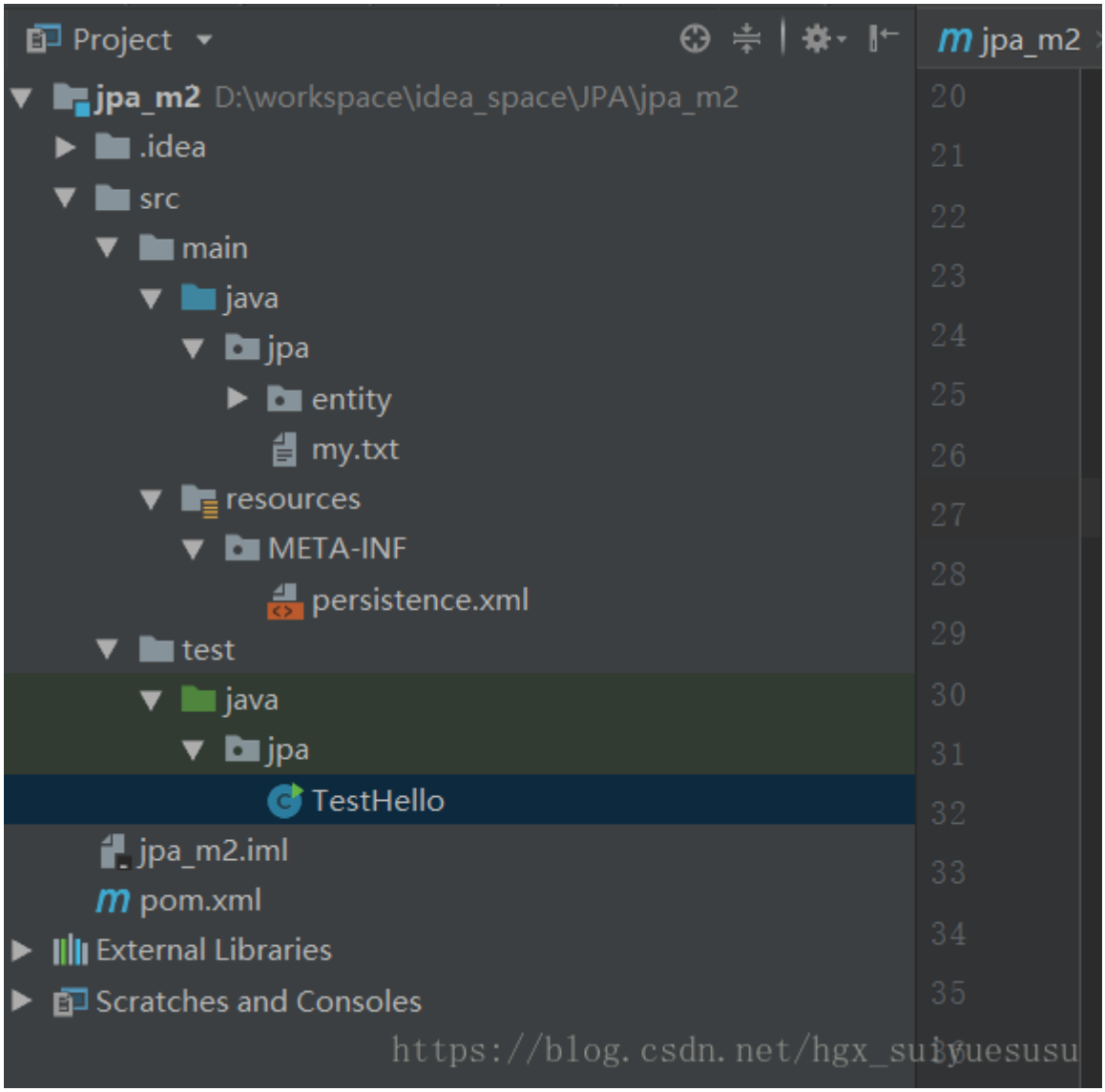
```
        this.id = id;
    }

    @Column(name = "LAST_NAME")
    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public Integer getAge() {
        return age;
    }

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




}
```

测试:

```
package jpa;

import jpa.entity.Customer;

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

public class TestHello {
    public static void main(String[] args) {

        //1. 创建 EntityManagerFactory
        //注意此处需要和persistence.xml下的persistence-unit name值相同
        String persistenceUnitName = "jpa_m2";
```

```
        EntityManagerFactory entityManagerFactory =
Persistence.createEntityManagerFactory(persistenceUnitName);


        //2. 创建 EntityManager. 类似于 Hibernate 的 SessionFactory
        EntityManager entityManager = entityManagerFactory.createEntityManager();

        //3. 开启事务
        EntityTransaction transaction = entityManager.getTransaction();
        transaction.begin();

        //4. 进行持久化操作
        Customer customer = new Customer();
        customer.setAge(12);
        customer.setEmail("tom@153.com");
        customer.setLastName("Tom");

        entityManager.persist(customer);

        //5. 提交事务
        transaction.commit();

        //6. 关闭 EntityManager
        entityManager.close();

        //7. 关闭 EntityManagerFactory
        entityManagerFactory.close();
    }
}
```

工程结构



三 基本注解

@Entity

- @Entity 标注用于实体类声明语句之前，指出该Java 类为实体类，将映射到指定的数据库表。如声明一个实体类 Customer，它将映射到数据库中的 customer 表上。

@Table

- 当实体类与其映射的数据库表名不同名时需要使用 @Table 标注说明，该标注与 @Entity 标注并列使用，置于实体类声明语句之前，可写于单独语句行，也可与声明语句同行。
- @Table 标注的常用选项是 name，用于指明数据库的表名
- @Table标注还有一个两个选项 catalog 和 schema 用于设置表所属的数据库目录或模式，通常为数据库名。uniqueConstraints 选项用于设置约束条件，通常不须设置。

@Id

- @Id 标注用于声明一个实体类的属性映射为数据库的主键列。该属性通常置于属性声明语句之前，可与声明语句同行，也可写在单独行上。
- @Id标注也可置于属性的getter方法之前。

@GeneratedValue

- **@GeneratedValue 用于标注主键的生成策略，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：SqlServer 对应 identity，MySQL 对应 auto increment。**
- **在 javax.persistence.GenerationType 中定义了以下几种可供选择的策略：**
  - IDENTITY：采用数据库 ID自增长的方式来自增主键字段，Oracle 不支持这种方式；
  - AUTO： JPA自动选择合适的策略，是默认选项；注意:此方式下 5.3版本mysql默认选择SEQUENCE的主键增加方式
  - SEQUENCE：通过序列产生主键，通过 @SequenceGenerator 注解指定序列名，MySql 不支持这种方式
  - TABLE：通过表产生主键，框架借由表模拟序列产生主键，使用该策略可以使应用更易于数据库移植。

@Basic

- @Basic 表示一个简单的属性到数据库表的字段的映射,对于没有任何标注的 getXxxx() 方法,默认即为@Basic
- fetch: 表示该属性的读取策略,有 EAGER 和 LAZY 两种,分别表示主支抓取和延迟加载,默认为 EAGER.
- optional:表示该属性是否允许为null, 默认为true

@Column

- 当实体的属性与其映射的数据库表的列不同名时需要使用@Column 标注说明，该属性通常置于实体的属性声明语句之前，还可与 @Id 标注一起使用。
- @Column 标注的常用属性是 name，用于设置映射数据库表的列名。此外，该标注还包含其它多个属性，如：unique 、nullable、length 等。
- @Column 标注的 columnDefinition 属性: 表示该字段在数据库中的实际类型.通常 ORM 框架可以根据属性类型自动判断数据库中字段的类型,但是对于Date类型仍无法确定数据库中字段类型究竟是DATE,TIME还是TIMESTAMP.此外,String的默认映射类型为VARCHAR, 如果要将 String 类型映射到特定数据库的 BLOB 或 TEXT 字段类型.
- @Column标注也可置于属性的getter方法之前

@Transient

- 表示该属性并非一个到数据库表的字段的映射,ORM框架将忽略该属性.
- 如果一个属性并非数据库表的字段映射,就务必将其标示为@Transient,否则,ORM框架默认其注解为@Basic

@Temporal

- 在核心的 Java API 中并没有定义 Date 类型的精度(temporal precision). 而在数据库中,表示 Date 类型的数据有 DATE, TIME, 和 TIMESTAMP 三种精度(即单纯的日期,时间,或者两者 兼备). 在进行属性映射时可使用@Temporal注解来调整精度.

代码:

```
package com.ifox.hgx.jpa.entity;

import javax.persistence.*;
```

```java
import java.util.Date;

//对应数据库，表名
@Table(name = "JPA_CUSTOMTERS")
@Entity
public class Customer {


    private Integer id ;
    private String lastName ;

    private String email ;
    private Integer age ;

    private Date createTime ;
    private Date birth ;

//  调整精度,年月日时分秒，如:2018-06-21 22:01:55.964000
    @Temporal(TemporalType.TIMESTAMP)
    public Date getCreateTime() {
        return createTime;
    }

    public void setCreateTime(Date createTime) {
        this.createTime = createTime;
    }

//  调整精度，年月日，如:2018-06-21
    @Temporal(TemporalType.DATE)
    public Date getBirth() {
        return birth;
    }

    public void setBirth(Date birth) {
        this.birth = birth;
    }

//    GeneratedValue 生成方式:策略为 GenerationType.AUTO 自动选择
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Id
    public Integer getId() {
        return id;
    }

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

    @Column(name = "LAST_NAME")
    public String getLastName() {
        return lastName;
    }
```

```
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    @Column(length = 50)
    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public Integer getAge() {
        return age;
    }

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

    //不是需要映射的字段的 需要加上注解Transient
    @Transient
    public String getInfo(){
        return "lastName:"+lastName+" Email:" +email ;
    }

    @Override
    public String toString() {
        return "Customer{" +
                "id=" + id +
                ", lastName='" + lastName + '\'' +
                ", email='" + email + '\'' +
                ", age=" + age +
                ", createTime=" + createTime +
                ", birth=" + birth +
                '}';
    }
}
```

用 table 来生成主键详解

- 将当前主键的值单独保存到一个数据库的表中，主键的值每次都是从指定的表中查询来获得
- 这种方法生成主键的策略可以适用于任何数据库，不必担心不同数据库不兼容造成的问题。

代码:

```
package com.ifox.hgx.jpa.entity;

import javax.persistence.*;
import java.util.Date;
```

```java
@Table(name = "JPA_CUSTOMTERS")
@Entity
public class Customer {


    private Integer id ;
    private String lastName ;

    private String email ;
    private Integer age ;

    private Date createTime ;
    private Date birth ;

    @Temporal(TemporalType.TIMESTAMP)
    public Date getCreateTime() {
        return createTime;
    }

    public void setCreateTime(Date createTime) {
        this.createTime = createTime;
    }

    @Temporal(TemporalType.DATE)
    public Date getBirth() {
        return birth;
    }

    public void setBirth(Date birth) {
        this.birth = birth;
    }

    // pkColumnName = "PK_NAME",pkColumnValue = "CUSTOMER_ID" 确定行
    // valueColumnName = "PK_VALUE" 确定列
    // allocationSize 每次增加多少
    // name = "ID_GENERATOR",对应@GeneratedValue 的 generator = "ID_GENERATOR"
    @TableGenerator(name = "ID_GENERATOR",table = "jap_id_generators",
            pkColumnName = "PK_NAME",pkColumnValue = "CUSTOMER_ID",
            valueColumnName = "PK_VALUE",initialValue = 1,allocationSize = 100

    )
    @GeneratedValue(strategy = GenerationType.TABLE,generator = "ID_GENERATOR")
    @Id
    public Integer getId() {
        return id;
    }

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

    @Column(name = "LAST_NAME")
```

```
    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    @Column(length = 50)
    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public Integer getAge() {
        return age;
    }

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

    //不是需要映射的字段的 需要加上注解Transient
    @Transient
    public String getInfo(){
        return "lastName:"+lastName+" Email:" +email ;
    }

    @Override
    public String toString() {
        return "Customer{" +
                "id=" + id +
                ", lastName='" + lastName + '\'' +
                ", email='" + email + '\'' +
                ", age=" + age +
                ", createTime=" + createTime +
                ", birth=" + birth +
                '}';
    }
}
```

详细:

```java
@TableGenerator(name = "ID_GENERATOR", table = "jap_id_generators",
        pkColumnName = "PK_NAME", pkColumnValue = "CUSTOMER_ID",
        valueColumnName = "PK_VALUE", initialValue = 1, allocationSize = 100
)
@GeneratedValue(strategy = GenerationType.TABLE, generator = "ID_GENERATOR")
@Id
public Integer getId() { return id; }
```

name 属性表示该主键生成策略的名称，它被引用在@GeneratedValue中设置的 generator 值中

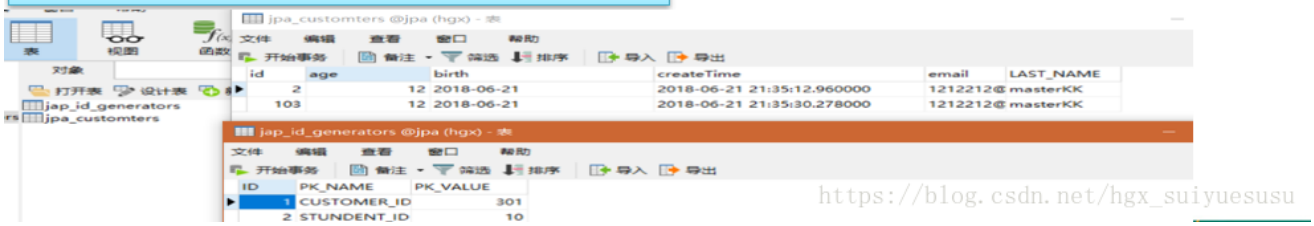
table 属性表示表生成策略所持久化的表名

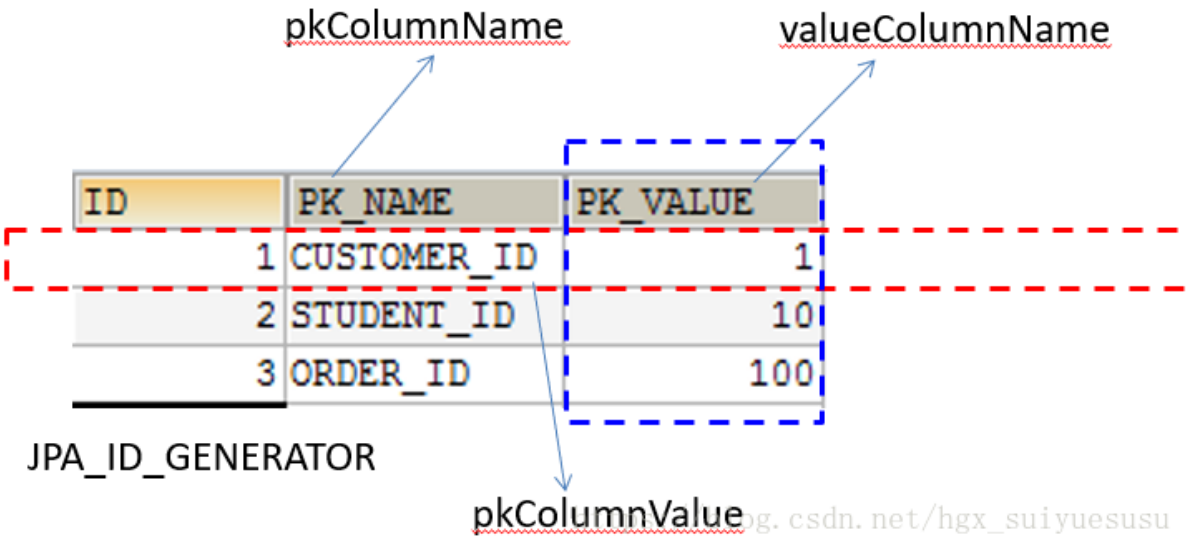
pkColumnName 属性的值表示在持久化表中，该主键生成策略所对应键值的名称

valueColumnName 属性的值表示在持久化表中，该主键当前所生成的值，它的值将会随着每次创建累加

pkColumnValue 属性的值表示在持久化表中，该生成策略所对应的主键

allocationSize 表示每次主键值增加的大小, 默认值为 50





四 API

JPA相关接口/类：Persistence

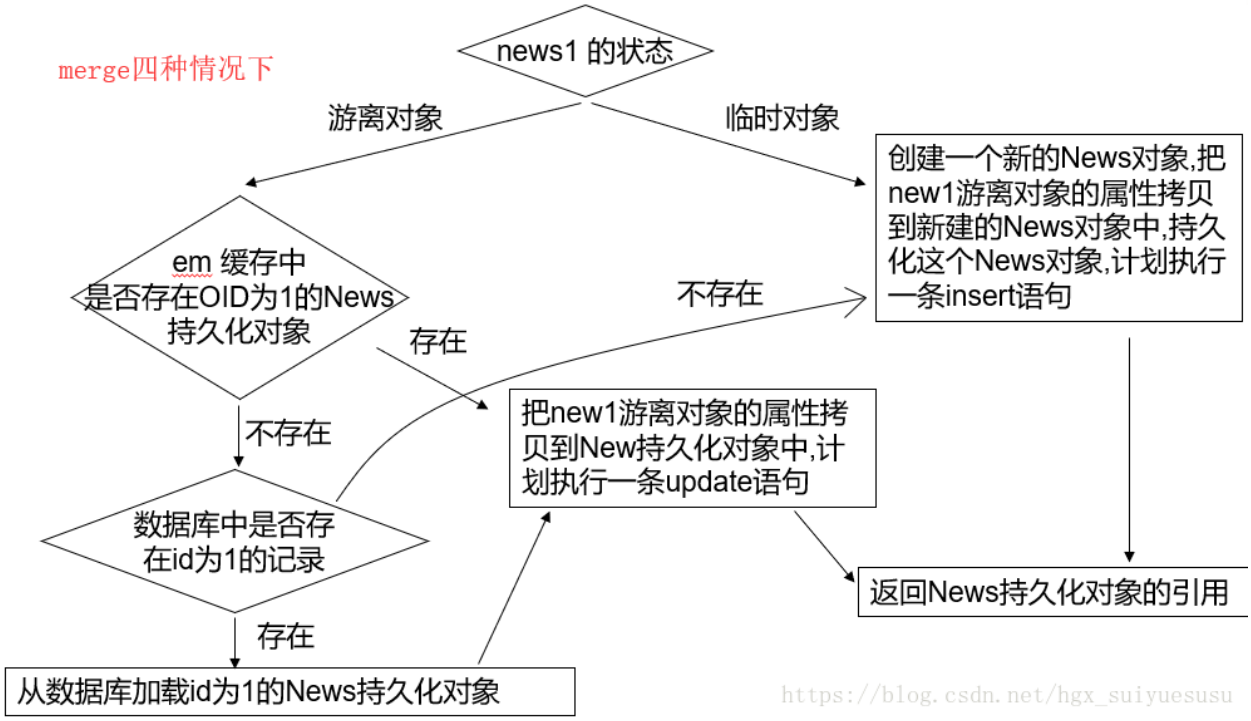
- Persistence 类是用于获取 EntityManagerFactory 实例。该类包含一个名为 createEntityManagerFactory 的 静态方法 。
- createEntityManagerFactory 方法有如下两个重载版本。
- 带有一个参数的方法以 JPA 配置文件 persistence.xml 中的持久化单元名为参数
- 带有两个参数的方法：前一个参数含义相同，后一个参数 Map类型，用于设置 JPA 的相关属 - 性，这时将忽略其它地方设置的属性。Map 对象的属性名必须是 JPA 实现库提供商的名字空间约定的属性名。

EntityManagerFactory

- EntityManagerFactory 接口主要用来创建 EntityManager 实例。该接口约定了如下4个方法：
  - `createEntityManager()`: 用于创建实体管理器对象实例。
  - `createEntityManager(Map map)`: 用于创建实体管理器对象实例的重载方法，Map 参数用于提供 EntityManager 的属性。
  - `isOpen()`: 检查 EntityManagerFactory 是否处于打开状态。实体管理器工厂创建后一直处于打开状态，除非调用close()方法将其关闭。
  - `close()`: 关闭 EntityManagerFactory 。 EntityManagerFactory 关闭后将释放所有资源，isOpen()方法测试将返回 false，其它方法将不能调用，否则将导致IllegalStateException异常。

EntityManager

- 在 JPA 规范中, EntityManager 是完成持久化操作的核心对象。实体作为普通 Java 对象，只有在调用 EntityManager 将其持久化后才会变成持久化对象。EntityManager 对象在一组实体类与底层数据源之间进行 O/R 映射的管理。它可以用来管理和更新 Entity Bean, 根据主键查找 Entity Bean, 还可以通过JPQL语句查询实体。
- 实体的状态:
- 新建状态: 新创建的对象，尚未拥有持久性主键。
- 持久化状态：已经拥有持久性主键并和持久化建立了上下文环境
- 游离状态：拥有持久化主键，但是没有与持久化建立上下文环境
- 删除状态: 拥有持久化主键，已经和持久化建立上下文环境，但是从数据库中删除。
- `find (Class<T> entityClass,Object primaryKey)`: 返回指定的 OID 对应的实体类对象，如果这个实体存在于当前的持久化环境，则返回一个被缓存的对象；否则会创建一个新的 Entity, 并加载数据库中相关信息；若 OID 不存在于数据库中，则返回一个 null。第一个参数为被查询的实体类类型，第二个参数为待查找实体的主键值。
- `getReference (Class<T> entityClass,Object primaryKey)`：与find()方法类似，不同的是：如果缓存中不存在指定的 Entity, EntityManager 会创建一个 Entity 类的代理，但是不会立即加载数据库中的信息，只有第一次真正使用此 Entity 的属性才加载，所以如果此 OID 在数据库不存在，getReference() 不会返回 null 值，而是抛出EntityNotFoundException
- `persist (Object entity)`: 用于将新创建的 Entity 纳入到 EntityManager 的管理。该方法执行后，传入 persist() 方法的 Entity 对象转换成持久化状态。
- 如果传入 persist() 方法的 Entity 对象已经处于持久化状态，则 persist() 方法什么都不做。
- 如果对删除状态的 Entity 进行 persist() 操作，会转换为持久化状态。
- 如果对游离状态的实体执行 persist() 操作，可能会在 persist() 方法抛出 EntityExistException(也有可能是在flush或事务提交后抛出)。
- `remove (Object entity)`: 删除实例。如果实例是被管理的，即与数据库实体记录关联，则同时会删除关联的数据库记录。
- `merge (T entity)`: merge() 用于处理 Entity 的同步。即数据库的插入和更新操作



- `flush ()`: 同步持久上下文环境，即将持久上下文环境的所有未保存实体的状态信息保存到数据库中。
- `setFlushMode (FlushModeType flushMode)`: 设置持久上下文环境的Flush模式。参数可以取2个枚举
- FlushModeType.AUTO 为自动更新数据库实体，
- FlushModeType.COMMIT 为直到提交事务时才更新数据库记录。
- `getFlushMode ()`: 获取持久上下文环境的Flush模式。返回FlushModeType类的枚举值。
- `refresh (Object entity)`: 用数据库实体记录的值更新实体对象的状态，即更新实例的属性值。
- `clear ()`: 清除持久上下文环境，断开所有关联的实体。如果这时还有未提交的更新则会被撤消。
- `contains (Object entity)`: 判断一个实例是否属于当前持久上下文环境管理的实体。
- `isOpen ()`: 判断当前的实体管理器是否是打开状态。
- `getTransaction ()`: 返回资源层的事务对象。EntityTransaction实例可以用于开始和提交多个事务。
- `close ()`: 关闭实体管理器。之后若调用实体管理器实例的方法或其派生的查询对象的方法都将抛出IllegalstateException 异常，除了getTransaction 和 isOpen方法(返回 false)。不过，当与实体管理器关联的事务处于活动状态时，调用 close 方法后持久上下文将仍处于被管理状态，直到事务完成。
- `createQuery (String qlString)`: 创建一个查询对象。
- `createNamedQuery (String name)`: 根据命名的查询语句块创建查询对象。参数为命名的查询语句。
- `createNativeQuery (String sqlString)`: 使用标准 SQL语句创建查询对象。参数为标准SQL语句字符串。
- `createNativeQuery (String sqls, String resultSetMapping)`: 使用标准SQL语句创建查询对象，并指定返回结果集 Map的 名称。

EntityTransaction

- EntityTransaction 接口用来管理资源层实体管理器的事务操作。通过调用实体管理器的getTransaction方法获得其实例。
- `begin () :` 用于启动一个事务，此后的多个数据库操作将作为整体被提交或撤消。若这时事务已启动则会抛出 IllegalStateException 异常。
- `commit ()`: 用于提交当前事务。即将事务启动以后的所有数据库更新操作持久化至数据库中。
- `rollback ()`: 撤消(回滚)当前事务。即撤消事务启动后的所有数据库更新操作，从而不对数据库产生影响。
- `setRollbackOnly ()`: 使当前事务只能被撤消。
- `getRollbackOnly ()`: 查看当前事务是否设置了只能撤消标志。

- `isActive ()`:查看当前事务是否是活动的。如果返回true则不能调用begin方法，否则将抛出 IllegalStateException 异常；如果返回 false 则不能调用 commit、rollback、setRollbackOnly 及 getRollbackOnly 方法，否则将抛出 IllegalStateException 异常。

代码演示:

```
package com.ifox.hgx.jpa.test;

import com.ifox.hgx.jpa.entity.Customer;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import java.util.Date;

public class JPATest {

    private EntityManagerFactory entityManagerFactory ;
    private EntityManager entityManager ;
    private EntityTransaction transaction ;

    @Before
    public void init(){
        entityManagerFactory = Persistence.createEntityManagerFactory("jpa_m1") ;
        entityManager = entityManagerFactory.createEntityManager() ;
        transaction = entityManager.getTransaction() ;
        transaction.begin();
    }

    @After
    public void distroy(){
        transaction.commit();
        entityManager.close();
        entityManagerFactory.close();
    }

    //类似Hibernate 中Session的 get的方法
    @Test
    public void testFind(){
        Customer customer = entityManager.find(Customer.class,1) ;
        System.out.println("----------------------------------------");
        System.out.println(customer);
    }
    /*
    Hibernate:
    select
        customer0_.id as id1_0_0_,
```

```
        customer0_.age as age2_0_0_,
        customer0_.birth as birth3_0_0_,
        customer0_.createTime as createTi4_0_0_,
        customer0_.email as email5_0_0_,
        customer0_.LAST_NAME as LAST_NAM6_0_0_
    from
        JPA_CUSTOMTERS customer0_
    where
        customer0_.id=?
    ----------------------------------
    Customer{id=1, lastName='masterKK', email='1212212@qq.com', age=12, createTime=2018-06-21
22:01:55.964, birth=2018-06-21}

     */

    //类似Hibernate 中Session的 load的方法
    @Test
    public void testLoad(){
        Customer customer = entityManager.getReference(Customer.class,1);
        //返回了代理对象，可能出现懒加载异常
        System.out.println(customer.getClass().getName());
        System.out.println("----------------------------------");
        System.out.println(customer);
    }
    /*

    com.ifox.hgx.jpa.entity.Customer$HibernateProxy$uCEfpt3v
    ----------------------------------
    Hibernate:
    select
        customer0_.id as id1_0_0_,
        customer0_.age as age2_0_0_,
        customer0_.birth as birth3_0_0_,
        customer0_.createTime as createTi4_0_0_,
        customer0_.email as email5_0_0_,
        customer0_.LAST_NAME as LAST_NAM6_0_0_
    from
        JPA_CUSTOMTERS customer0_
    where
        customer0_.id=?
    Customer{id=1, lastName='masterKK', email='1212212@qq.com', age=12, createTime=2018-06-21
22:01:55.964, birth=2018-06-21}

     */



    /*
        类似于 hibernate 的 save 方法. 使对象由临时状态变为持久化状态.
        和 hibernate 的 save 方法的不同之处: 若对象有 id, 则不能执行 insert 操作, 而会抛出异常.
     */
    @Test
    public void testPersist(){
```

```
        Customer customer = new Customer() ;
        customer.setBirth(new Date());
        customer.setCreateTime(new Date());
        customer.setLastName("AAA");
        customer.setEmail("AAA@163.com");
        customer.setAge(10);
        //customer.setId(100);

        entityManager.persist(customer);
        System.out.println(customer.getId());
    }

    //类似于 hibernate 中 Session 的 delete 方法. 把对象对应的记录从数据库中移除
    //但注意: 该方法只能移除 持久化 对象. 而 hibernate 的 delete 方法实际上还可以移除 游离对象.
    @Test
    public void testRemove(){
//        Customer customer = new Customer() ;
//        customer.setId(1);
//        此时数据库中有这个对象对应的数据，但是和EntityManager没有关联，是一个游离对象

        Customer customer = entityManager.find(Customer.class,2) ;
        entityManager.remove(customer);
    }

    /**
     * 总的来说: 类似于 hibernate Session 的 saveOrUpdate 方法.
     */
    //1. 若传入的是一个临时对象
    //会创建一个新的对象，把临时对象的属性复制到新的对象中，然后对新的对象执行持久化操作. 所以
    //新的对象中有 id，但以前的临时对象中没有 id.
    @Test
    public void testMerge1(){
        Customer customer = new Customer();
        customer.setAge(18);
        customer.setBirth(new Date());
        customer.setCreateTime(new Date());
        customer.setEmail("cc@163.com");
        customer.setLastName("CC");

        Customer customer2 = entityManager.merge(customer);

        System.out.println("customer#id:" + customer.getId());
        System.out.println("customer2#id:" + customer2.getId());
    }


    //若传入的是一个游离对象，即传入的对象有 OID.
    //1. 若在 EntityManager 缓存中没有该对象
    //2. 若在数据库中也没有对应的记录
    //3. JPA 会创建一个新的对象，然后把当前游离对象的属性复制到新创建的对象中
    //4. 对新创建的对象执行 insert 操作.
    @Test
    public void testMerge2(){
```

```
        Customer customer = new Customer();
        customer.setAge(18);
        customer.setBirth(new Date());
        customer.setCreateTime(new Date());
        customer.setEmail("dd@163.com");
        customer.setLastName("DD");

        customer.setId(100);

        Customer customer2 = entityManager.merge(customer);

        System.out.println("customer#id:" + customer.getId());
        System.out.println("customer2#id:" + customer2.getId());
    }


    //若传入的是一个游离对象，即传入的对象有 OID.
    //1. 若在 EntityManager 缓存中没有该对象
    //2. 若在数据库中也有对应的记录
    //3. JPA 会查询对应的记录，然后返回该记录对一个的对象，再然后会把游离对象的属性复制到查询到的对象中.
    //4. 对查询到的对象执行 update 操作.
    @Test
    public void testMerge3(){
        Customer customer = new Customer();
        customer.setAge(18);
        customer.setBirth(new Date());
        customer.setCreateTime(new Date());
        customer.setEmail("ee@163.com");
        customer.setLastName("EE");

        customer.setId(4);

        Customer customer2 = entityManager.merge(customer);

        System.out.println(customer == customer2); //false
    }

    //若传入的是一个游离对象，即传入的对象有 OID.
    //1. 若在 EntityManager 缓存中有对应的对象
    //2. JPA 会把游离对象的属性复制到查询到EntityManager 缓存中的对象中.
    //3. EntityManager 缓存中的对象执行 UPDATE.
    @Test
    public void testMerge4(){
        Customer customer = new Customer();
        customer.setAge(18);
        customer.setBirth(new Date());
        customer.setCreateTime(new Date());
        customer.setEmail("dd@163.com");
        customer.setLastName("DD");

        customer.setId(4);
        Customer customer2 = entityManager.find(Customer.class, 4);
```

```
        entityManager.merge(customer);

        System.out.println(customer == customer2); //false
    }


    /**
     * 同 hibernate 中 Session 的 refresh 方法.
     */
    @Test
    public void testRefresh(){

        Customer customer = entityManager.find(Customer.class, 1);
        customer = entityManager.find(Customer.class, 1);

        entityManager.refresh(customer);
    }

    /**
     * 同 hibernate 中 Session 的 flush 方法.
     */
    @Test
    public void testFlush(){
        Customer customer = entityManager.find(Customer.class, 1);
        System.out.println(customer);

        customer.setLastName("AA");

        entityManager.flush();
    }

}
```

五 映射关联关系

映射单向多对一的关联关系

- Customer类

```
package com.ifox.hgx.jpa.entity;

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

@Table(name = "JPA_CUSTOMTERS")
@Entity
public class Customer {
```

```
    private Integer id ;
    private String lastName ;

    private String email ;
    private Integer age ;

    private Date createTime ;
    private Date birth ;

    @Temporal(TemporalType.TIMESTAMP)
    public Date getCreateTime() {
        return createTime;
    }

    public void setCreateTime(Date createTime) {
        this.createTime = createTime;
    }

    @Temporal(TemporalType.DATE)
    public Date getBirth() {
        return birth;
    }

    public void setBirth(Date birth) {
        this.birth = birth;
    }

    // pkColumnName = "PK_NAME",pkColumnValue = "CUSTOMER_ID"确定行
    // valueColumnName = "PK_VALUE"确定列
    //allocationSize 每次增加多少
//    @TableGenerator(name = "ID_GENERATOR",table = "jap_id_generators",
//            pkColumnName = "PK_NAME",pkColumnValue = "CUSTOMER_ID",
//            valueColumnName = "PK_VALUE",initialValue = 1,allocationSize = 100
//
//    )
//    @GeneratedValue(strategy = GenerationType.TABLE,generator = "ID_GENERATOR")

//    GeneratedValue 生成方式:策略为 GenerationType.AUTO 自动选择
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Id
    public Integer getId() {
        return id;
    }

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

    @Column(name = "LAST_NAME")
    public String getLastName() {
        return lastName;
```

```
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    @Column(length = 50)
    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public Integer getAge() {
        return age;
    }

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

    //不是需要映射的字段的 需要加上注解Transient
    @Transient
    public String getInfo(){
        return "lastName:"+lastName+" Email:" +email ;
    }


}
```

- Order类

```
package com.ifox.hgx.jpa.entity;

import javax.persistence.*;

@Table(name = "JPA_ORDER")
@Entity
public class Order {
    private Integer id ;
    private String orderName ;
    private Customer customer ;

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

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

    @Column(name = "ORDER_NAME")
    public String getOrderName() {
        return orderName;
    }

    public void setOrderName(String orderName) {
        this.orderName = orderName;
    }

//  @ManyToOne(fetch = FetchType.LAZY) 懒加载

    @JoinColumn(name = "CUSTOMER_ID")
    @ManyToOne(fetch = FetchType.LAZY)
    public Customer getCustomer() {
        return customer;
    }
    public void setCustomer(Customer customer) {
        this.customer = customer;
    }
}
```

- 测试示例:

```
package com.ifox.hgx.jpa.test;

import com.ifox.hgx.jpa.entity.Customer;
import com.ifox.hgx.jpa.entity.Order;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import java.util.Date;

public class TestMappingRelations {

    private EntityManagerFactory entityManagerFactory ;
    private EntityManager entityManager ;
    private EntityTransaction transaction ;

    @Before
    public void init(){
        entityManagerFactory = Persistence.createEntityManagerFactory("jpa_m1") ;
        entityManager = entityManagerFactory.createEntityManager() ;
```

```
        transaction = entityManager.getTransaction() ;
        transaction.begin();
    }

    @After
    public void distroy(){
        transaction.commit();
        entityManager.close();
        entityManagerFactory.close();
    }

//  保存多对一时，建议先保存 1 的一端，后保存 n 的一端，这样不会多出额外的 UPDATE 语句.
    @Test
    public void testManyToOnePersist(){
        Customer customer = new Customer();
        customer.setAge(19);
        customer.setBirth(new Date());
        customer.setCreateTime(new Date());
        customer.setEmail("GG@163.com");
        customer.setLastName("GG");

        Order order1 = new Order() ;
        Order order2 = new Order() ;
        order1.setOrderName("GG-0-1");
        order2.setOrderName("GG-0-2");

        order1.setCustomer(customer);
        order2.setCustomer(customer);

        //3条insert sql语句
//        entityManager.persist(customer);
//        entityManager.persist(order1);
//        entityManager.persist(order2);

        //3条insert 2条update
        entityManager.persist(order1);
        entityManager.persist(order2);
        entityManager.persist(customer);
    }
    //默认情况下，使用左外连接的方式来获取 n 的一端的对象和其关联的 1 的一端的对象.
    //可使用 @ManyToOne 的 fetch 属性来修改默认的关联属性的加载策略
    //默认不使用懒加载
    @Test
    public void testManyToOneFind(){
        Order order = entityManager.find(Order.class,1) ;
        System.out.println(order.getOrderName());
        System.out.println(order.getCustomer().getLastName());
    }



    @Test
    public void testManyToOneUpdate(){
```

```
        Order order = entityManager.find(Order.class,1) ;
        order.getCustomer().setLastName("FF");

    }

    //不能直接删除 1 的一端，因为有外键约束.
    @Test
    public void testManyToOneRemove(){
//        Order order = entityManager.find(Order.class,1) ;
//        entityManager.remove(order);

        Customer customer = entityManager.find(Customer.class,1) ;
        entityManager.remove(customer);
    }


}
```

- persistence.xml中加入:

```
<!--添加持久化类-->
        <class>com.ifox.hgx.jpa.entity.Customer</class>
        <class>com.ifox.hgx.jpa.entity.Order</class>
```

映射单向一对多的关联关系

- Customer类中:

```
package com.ifox.hgx.jpa.entity;

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

@Table(name = "JPA_CUSTOMTERS")
@Entity
public class Customer {


    private Integer id ;
    private String lastName ;

    private String email ;
    private Integer age ;

    private Date createTime ;
    private Date birth ;
```

```
    private Set<Order> orders = new HashSet<>() ;

    //1-n 添加时,需要cascade = CascadeType.PERSIST 级联添加
    //1-n 级联删除,需要CascadeType.REMOVE
    @JoinColumn(name = "CUSTOMER_ID")
    @OneToMany(cascade = {CascadeType.PERSIST,CascadeType.REMOVE})
    public Set<Order> getOrders() {
        return orders;
    }

    public void setOrders(Set<Order> orders) {
        this.orders = orders;
    }

    @Temporal(TemporalType.TIMESTAMP)
    public Date getCreateTime() {
        return createTime;
    }

    public void setCreateTime(Date createTime) {
        this.createTime = createTime;
    }

    @Temporal(TemporalType.DATE)
    public Date getBirth() {
        return birth;
    }

    public void setBirth(Date birth) {
        this.birth = birth;
    }

    // pkColumnName = "PK_NAME",pkColumnValue = "CUSTOMER_ID"确定行
    // valueColumnName = "PK_VALUE"确定列
    //allocationSize 每次增加多少
//    @TableGenerator(name = "ID_GENERATOR",table = "jap_id_generators",
//            pkColumnName = "PK_NAME",pkColumnValue = "CUSTOMER_ID",
//            valueColumnName = "PK_VALUE",initialValue = 1,allocationSize = 100
//
//    )
//    @GeneratedValue(strategy = GenerationType.TABLE,generator = "ID_GENERATOR")

//    GeneratedValue 生成方式:策略为 GenerationType.AUTO 自动选择
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Id
    public Integer getId() {
        return id;
    }

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

```
    @Column(name = "LAST_NAME")
    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    @Column(length = 50)
    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public Integer getAge() {
        return age;
    }

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

    //不是需要映射的字段的 需要加上注解Transient
    @Transient
    public String getInfo(){
        return "lastName:"+lastName+" Email:" +email ;
    }

    @Override
    public String toString() {
        return "Customer{" +
                "id=" + id +
                ", lastName='" + lastName + '\'' +
                ", email='" + email + '\'' +
                ", age=" + age +
                ", createTime=" + createTime +
                ", birth=" + birth +
                ", orders=" + orders +
                '}';
    }
}
```

- Order类:

```
package com.ifox.hgx.jpa.entity;
```

```
import javax.persistence.*;

@Table(name = "JPA_ORDER")
@Entity
public class Order {
    private Integer id ;
    private String orderName ;
//    private Customer customer ;

    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Id
    public Integer getId() {
        return id;
    }

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

    @Column(name = "ORDER_NAME")
    public String getOrderName() {
        return orderName;
    }

    public void setOrderName(String orderName) {
        this.orderName = orderName;
    }

////  @ManyToOne(fetch = FetchType.LAZY) 懒加载
//    @JoinColumn(name = "CUSTOMER_ID")
//    @ManyToOne(fetch = FetchType.LAZY)
//    public Customer getCustomer() {
//        return customer;
//    }
//
//    public void setCustomer(Customer customer) {
//        this.customer = customer;
//    }
}
```

测试示例:

```
package com.ifox.hgx.jpa.test;

import com.ifox.hgx.jpa.entity.Customer;
import com.ifox.hgx.jpa.entity.Order;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import javax.persistence.EntityManager;
```

```java
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import java.util.Date;

public class TestMapping_OneToMany {

    private EntityManagerFactory entityManagerFactory ;
    private EntityManager entityManager ;
    private EntityTransaction transaction ;

    @Before
    public void init(){
        entityManagerFactory = Persistence.createEntityManagerFactory("jpa_m1") ;
        entityManager = entityManagerFactory.createEntityManager() ;
        transaction = entityManager.getTransaction() ;
        transaction.begin();
    }

    @After
    public void destroy(){
        transaction.commit();
        entityManager.close();
        entityManagerFactory.close();
    }

    //单向 1-n 关联关系执行保存时，一定会多出 UPDATE 语句.
    //因为 n 的一端在插入时不会同时插入外键列.
    //
    @Test
    public void testOneToManyPersist(){
        Customer customer = new Customer() ;
        customer.setLastName("KKK");
        customer.setCreateTime(new Date());
        customer.setBirth(new Date());
        customer.setEmail("KKK@qq.com");
        customer.setAge(34);

        Order order = new Order() ;
        order.setOrderName("KK-01-1");

        Order order1 = new Order() ;
        order1.setOrderName("KK-02-2");

        customer.getOrders().add(order);
        customer.getOrders().add(order1) ;

        entityManager.persist(order);
        entityManager.persist(order);

        entityManager.persist(customer);
```

```
    }

    //默认对关联的多的一方使用懒加载的加载策略.
    //可以使用 @OneToMany 的 fetch 属性来修改默认的加载策略
    @Test
    public void testOneToManyFind(){
        Customer customer = entityManager.find(Customer.class,1) ;
        System.out.println(customer.getLastName());
        System.out.println(customer.getOrders());
    }

    //默认情况下, 若删除 1 的一端, 则会先把关联的 n 的一端的外键置空, 然后进行删除.
    //可以通过 @OneToMany 的 cascade 属性来修改默认的删除策略.
    @Test
    public void testOneToManyRemove(){
        Customer customer  = entityManager.find(Customer.class,1) ;
        entityManager.remove(customer);
    }

    @Test
    public void testOneToManyUpdate(){
        Customer customer  = entityManager.find(Customer.class,1) ;
        customer.getOrders().iterator().next().setOrderName("0-xxx-x");
    }
}
```

映射双向多对一的关联关系

- 双向一对多关系中，必须存在一个关系维护端，在 JPA 规范中，要求 many 的一方作为关系的维护端(owner side), one 的一方作为被维护端(inverse side).
- 可以在 one 方指定 @OneToMany 注释并设置 mappedBy 属性，以指定它是这一关联中的被维护端，many 为维护端.
- 在 many 方指定 @ManyToOne 注释，并使用 @JoinColumn 指定外键名称.

- Customer类

```
package com.ifox.hgx.jpa.entity;

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

@Table(name = "JPA_CUSTOMTERS")
@Entity
public class Customer {


    private Integer id ;
    private String lastName ;
```

```
    private String email ;
    private Integer age ;

    private Date createTime ;
    private Date birth ;

    private Set<Order> orders = new HashSet<>() ;

    //1-n 添加时,需要cascade = CascadeType.PERSIST 级联添加
    //1-n 级联删除,需要CascadeType.REMOVE
    //@JoinColumn(name = "CUSTOMER_ID")
    @OneToMany(cascade = {CascadeType.PERSIST,CascadeType.REMOVE},mappedBy = "customer",fetch =
FetchType.EAGER)
    public Set<Order> getOrders() {
        return orders;
    }

    public void setOrders(Set<Order> orders) {
        this.orders = orders;
    }

    @Temporal(TemporalType.TIMESTAMP)
    public Date getCreateTime() {
        return createTime;
    }

    public void setCreateTime(Date createTime) {
        this.createTime = createTime;
    }

    @Temporal(TemporalType.DATE)
    public Date getBirth() {
        return birth;
    }

    public void setBirth(Date birth) {
        this.birth = birth;
    }

    // pkColumnName = "PK_NAME",pkColumnValue = "CUSTOMER_ID"确定行
    // valueColumnName = "PK_VALUE"确定列
    //allocationSize 每次增加多少
//    @TableGenerator(name = "ID_GENERATOR",table = "jap_id_generators",
//            pkColumnName = "PK_NAME",pkColumnValue = "CUSTOMER_ID",
//            valueColumnName = "PK_VALUE",initialValue = 1,allocationSize = 100
//
//    )
//    @GeneratedValue(strategy = GenerationType.TABLE,generator = "ID_GENERATOR")

//    GeneratedValue 生成方式:策略为 GenerationType.AUTO 自动选择
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Id
```

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

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

    @Column(name = "LAST_NAME")
    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    @Column(length = 50)
    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public Integer getAge() {
        return age;
    }

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

    //不是需要映射的字段的 需要加上注解Transient
    @Transient
    public String getInfo(){
        return "lastName:"+lastName+" Email:" +email ;
    }

    @Override
    public String toString() {
        return "Customer{" +
                "id=" + id +
                ", lastName='" + lastName + '\'' +
                ", email='" + email + '\'' +
                ", age=" + age +
                ", createTime=" + createTime +
                ", birth=" + birth +
                ", orders=" + orders +
                '}';
    }
}
```

- Order类

```
package com.ifox.hgx.jpa.entity;

import javax.persistence.*;

@Table(name = "JPA_ORDER")
@Entity
public class Order {
    private Integer id ;
    private String orderName ;
    private Customer customer ;



    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Id
    public Integer getId() {
        return id;
    }

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

    @Column(name = "ORDER_NAME")
    public String getOrderName() {
        return orderName;
    }

    public void setOrderName(String orderName) {
        this.orderName = orderName;
    }

//  @ManyToOne(fetch = FetchType.LAZY) 懒加载
    @JoinColumn(name = "CUSTOMER_ID")
    @ManyToOne(fetch = FetchType.LAZY)
    public Customer getCustomer() {
        return customer;
    }

    public void setCustomer(Customer customer) {
        this.customer = customer;
    }
}
```

- 测试示例:

```java
package com.ifox.hgx.jpa.test;

import com.ifox.hgx.jpa.entity.Customer;
import com.ifox.hgx.jpa.entity.Order;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import java.util.Date;

public class TestMapping_TowWayOneToMany {

    private EntityManagerFactory entityManagerFactory;
    private EntityManager entityManager;
    private EntityTransaction transaction;

    @Before
    public void init() {
        entityManagerFactory = Persistence.createEntityManagerFactory("jpa_m1");
        entityManager = entityManagerFactory.createEntityManager();
        transaction = entityManager.getTransaction();
        transaction.begin();
    }

    @After
    public void destroy() {
        transaction.commit();
        entityManager.close();
        entityManagerFactory.close();
    }

    //若是双向 1-n 的关联关系，执行保存时
    //若先保存 n 的一端，再保存 1 的一端，默认情况下，会多出 n 条 UPDATE 语句.
    //若先保存 1 的一端，则会多出 n 条 UPDATE 语句
    //在进行双向 1-n 关联关系时，建议使用 n 的一方来维护关联关系，而 1 的一方不维护关联系，这样会有效的减少 SQL 语句.
    //注意：若在 1 的一端的 @OneToMany 中使用 mappedBy 属性，则 @OneToMany 端就不能再使用 @JoinColumn 属性了.
    @Test
    public void testTowWayOneToManyPersist() {
        Customer customer = new Customer();
        customer.setLastName("CCC");
        customer.setCreateTime(new Date());
        customer.setBirth(new Date());
        customer.setEmail("CCC@qq.com");
        customer.setAge(34);

        Order order = new Order();
        order.setOrderName("CC-01-1");
```

```
        Order order1 = new Order();
        order1.setOrderName("CC-02-2");

        customer.getOrders().add(order);
        customer.getOrders().add(order1);

        order.setCustomer(customer);
        order1.setCustomer(customer);

        //3insert 2update
//        entityManager.persist(order);
//        entityManager.persist(order);
//
//        entityManager.persist(customer);

        //如果使用了mappedBy = "customer" 则只用3条insert语句
        entityManager.persist(customer);

        entityManager.persist(order);
        entityManager.persist(order);


    }

    //可以使用 @OneToMany 的 fetch 属性来修改默认的加载策略,可以使用 @OneToMany 的 fetch 属性来修改默认的加载策略
    @Test
    public void testTowWayOneToManyFind(){
        Customer customer = entityManager.find(Customer.class, 1);
        System.out.println(customer.getLastName());

        System.out.println(customer.getOrders().size());
    }

    //默认情况下, 若删除 1 的一端, 则会先把关联的 n 的一端的外键置空, 然后进行删除.
    //可以通过 @OneToMany 的 cascade 属性来修改默认的删除策略. CascadeType.REMOVE 级联删除
    @Test
    public void tesTowWayOneToManyRemove(){
        Customer customer = entityManager.find(Customer.class, 3);
        entityManager.remove(customer);
    }

    @Test
    public void testTowWayUpdate(){
        Customer customer = entityManager.find(Customer.class, 1);
        customer.getOrders().iterator().next().setOrderName("O-XXX-1");

    }

}
```

映射双向一对一的关联关系

- 基于外键的 1-1 关联关系：在双向的一对一关联中，需要在关系被维护端(inverse side)中的 @OneToOne 注释中指定 mappedBy，以指定是这一关联中的被维护端。同时需要在关系维护端(owner side)建立外键列指向关系被维护端的主键列。
- 如果延迟加载要起作用, 就必须设置一个代理对象.
- Manager 其实可以不关联一个 Department
- 如果有 Department 关联就设置为代理对象而延迟加载, 如果不存在关联的 Department 就设置 null, 因为外键字段是定义在 Department 表中的,Hibernate 在不读取 Department 表的情况是无法判断是否有关联有 Deparmtment, 因此无法判断设置 null 还是代理对象, 而统一设置为代理对象,也无法满足不关联的情况, 所以无法使用延迟加载,只 有显式读取 Department.

- Department 类

```
package com.ifox.hgx.jpa.entity;

import javax.persistence.*;

@Table(name="JPA_DEPARTMENTS")
@Entity
public class Department {

    private Integer id;
    private String deptName;

    private Manager mgr;

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

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

    @Column(name="DEPT_NAME")
    public String getDeptName() {
        return deptName;
    }

    public void setDeptName(String deptName) {
        this.deptName = deptName;
    }

    //使用 @OneToOne 来映射 1-1 关联关系。
    //若需要在当前数据表中添加主键则需要使用 @JoinColumn 来进行映射. 注意, 1-1 关联关系, 所以需要添加
unique=true
```

```
    @JoinColumn(name="MGR_ID", unique=true)
    @OneToOne(fetch=FetchType.LAZY)
    public Manager getMgr() {
        return mgr;
    }

    public void setMgr(Manager mgr) {
        this.mgr = mgr;
    }
}
```

- Manager 类

```
package com.ifox.hgx.jpa.entity;

import javax.persistence.*;

@Table(name = "JPA_MANAGERS")
@Entity
public class Manager {

    private Integer id;
    private String mgrName;

    private Department dept;

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

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

    @Column(name = "MGR_NAME")
    public String getMgrName() {
        return mgrName;
    }

    public void setMgrName(String mgrName) {
        this.mgrName = mgrName;
    }

    //对于不维护关联关系，没有外键的一方，使用 @OneToOne 来进行映射，建议设置 mappedBy=true
    @OneToOne(mappedBy = "mgr")
    public Department getDept() {
        return dept;
    }
```

```
    public void setDept(Department dept) {
        this.dept = dept;
    }
}
```

测试示例:

```
package com.ifox.hgx.jpa.test;

import com.ifox.hgx.jpa.entity.Department;
import com.ifox.hgx.jpa.entity.Manager;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

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

public class TestMapping_TwoWayOneToOne {

    private EntityManagerFactory entityManagerFactory;
    private EntityManager entityManager;
    private EntityTransaction transaction;

    @Before
    public void init() {
        entityManagerFactory = Persistence.createEntityManagerFactory("jpa_m1");
        entityManager = entityManagerFactory.createEntityManager();
        transaction = entityManager.getTransaction();
        transaction.begin();
    }

    @After
    public void destroy() {
        transaction.commit();
        entityManager.close();
        entityManagerFactory.close();
    }

    //双向 1-1 的关联关系，建议先保存不维护关联关系的一方，即没有外键的一方，这样不会多出 UPDATE 语句.
    @Test
    public void testTwoWayOneToOnePersist(){
        Manager mgr = new Manager();
        mgr.setMgrName("M-BB");

        Department dept = new Department();
        dept.setDeptName("D-BB");
```

```
        //设置关联关系
        mgr.setDept(dept);
        dept.setMgr(mgr);

        //执行保存操作
        entityManager.persist(mgr);
        entityManager.persist(dept);
    }

    //1.默认情况下，若获取维护关联关系的一方，则会通过左外连接获取其关联的对象.
    //但可以通过 @OntToOne 的 fetch 属性来修改加载策略. lazy 下:
com.ifox.hgx.jpa.entity.Manager$HibernateProxy$vI5wL1QS 代理对象
    @Test
    public void testTwoWayOneToOneFind(){
        Department dept = entityManager.find(Department.class, 2);
        System.out.println(dept.getDeptName());
        System.out.println(dept.getMgr().getClass().getName());
    }

    //1. 默认情况下，若获取不维护关联关系的一方，则也会通过左外连接获取其关联的对象.
    //可以通过 @OneToOne 的 fetch 属性来修改加载策略. 但依然会再发送 SQL 语句来初始化其关联的对象
    //这说明在不维护关联关系的一方，不建议修改 fetch 属性.
    @Test
    public void testTwoWayOneToOneFind2(){
        Manager mgr = entityManager.find(Manager.class, 1);
        System.out.println(mgr.getMgrName());
        System.out.println(mgr.getDept().getClass().getName());
    }
}
```

persistence.xml中加入:

```
 <!--添加持久化类-->
<class>com.ifox.hgx.jpa.entity.Manager</class>
<class>com.ifox.hgx.jpa.entity.Department</class>
```

映射双向多对多的关联关系

- 在双向多对多关系中，我们必须指定一个关系维护端(owner side),可以通过 @ManyToMany 注释中指定 mappedBy 属性来标识其为关系维护端。

- Item 实体类

```
package com.ifox.hgx.jpa.entity;
```

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

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

@Table(name="JPA_ITEMS")
@Entity
public class Item {

    private Integer id;
    private String itemName;

    private Set<Category> categories = new HashSet<>();

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

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

    @Column(name="ITEM_NAME")
    public String getItemName() {
        return itemName;
    }

    public void setItemName(String itemName) {
        this.itemName = itemName;
    }

    //使用 @ManyToMany 注解来映射多对多关联关系
    //使用 @JoinTable 来映射中间表
    //1. name 指向中间表的名字
    //2. joinColumns 映射当前类所在的表在中间表中的外键
    //2.1 name 指定外键列的列名
    //2.2 referencedColumnName 指定外键列关联当前表的哪一列
    //3. inverseJoinColumns 映射关联的类所在中间表的外键
    @JoinTable(name="ITEM_CATEGORY",
            joinColumns={@JoinColumn(name="ITEM_ID", referencedColumnName="ID")},
            inverseJoinColumns={@JoinColumn(name="CATEGORY_ID", referencedColumnName="ID")})
    @ManyToMany
    public Set<Category> getCategories() {
        return categories;
```

```
    }

    public void setCategories(Set<Category> categories) {
        this.categories = categories;
    }
}
```

- Category 实体类

```
package com.ifox.hgx.jpa.entity;


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

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

@Table(name="JPA_CATEGORIES")
@Entity
public class Category {

    private Integer id;
    private String categoryName;

    private Set<Item> items = new HashSet<>();

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

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

    @Column(name="CATEGORY_NAME")
    public String getCategoryName() {
        return categoryName;
    }

    public void setCategoryName(String categoryName) {
        this.categoryName = categoryName;
    }

    @ManyToMany(mappedBy="categories")
```

```
    public Set<Item> getItems() {
        return items;
    }

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

```
@JoinTable(name="ITEM_CATEGORY",
        joinColumns={@JoinColumn(name="ITEM_ID", referencedColumnName="ID")},
        inverseJoinColumns={@JoinColumn(name="CATEGORY_ID", referencedColumnName="ID")})
@ManyToMany
public Set<Category> getCategories() {
    return categories;
}
```

```
@ManyToMany(mappedBy="categories")
public Set<Item> getItems() {
    return items;
}
```

```
@ManyToMany
@JoinTable(name="中间表名称",
joinColumns=@joinColumn(name="本类的外键",
referencedColumnName="本类与外键对应的主键"),
inversejoinColumns=@JoinColumn(name="对方类的外键",
referencedColunName="对方类与外键对应的主键")
)
```


- persistence.xml中加入:

```
<class>com.ifox.hgx.jpa.entity.Category</class>
<class>com.ifox.hgx.jpa.entity.Item</class>
```

- 测试示例:

```
package com.ifox.hgx.jpa.test;

import com.ifox.hgx.jpa.entity.Category;
import com.ifox.hgx.jpa.entity.Item;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

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

public class Test_TwoWayManyToMany {


    private EntityManagerFactory entityManagerFactory;
    private EntityManager entityManager;
```

```
    private EntityTransaction transaction;

    @Before
    public void init() {
        entityManagerFactory = Persistence.createEntityManagerFactory("jpa_m1");
        entityManager = entityManagerFactory.createEntityManager();
        transaction = entityManager.getTransaction();
        transaction.begin();
    }

    @After
    public void destroy() {
        transaction.commit();
        entityManager.close();
        entityManagerFactory.close();
    }

    //多对多的保存
    @Test
    public void testManyToManyPersist(){
        Item i1 = new Item();
        i1.setItemName("i-1");

        Item i2 = new Item();
        i2.setItemName("i-2");

        Category c1 = new Category();
        c1.setCategoryName("C-1");

        Category c2 = new Category();
        c2.setCategoryName("C-2");

        //设置关联关系
        i1.getCategories().add(c1);
        i1.getCategories().add(c2);

        i2.getCategories().add(c1);
        i2.getCategories().add(c2);

        c1.getItems().add(i1);
        c1.getItems().add(i2);

        c2.getItems().add(i1);
        c2.getItems().add(i2);

        //执行保存
        entityManager.persist(i1);
        entityManager.persist(i2);
        entityManager.persist(c1);
        entityManager.persist(c2);
    }
```

```
    //对于关联的集合对象，默认使用懒加载的策略.
    //使用维护关联关系的一方获取，还是使用不维护关联关系的一方获取，SQL 语句相同.
    @Test
    public void testManyToManyFind(){
//      Item item = entityManager.find(Item.class, 5);
//      System.out.println(item.getItemName());
//
//      System.out.println(item.getCategories().size());

        Category category = entityManager.find(Category.class, 7);
        System.out.println(category.getCategoryName());
        System.out.println(category.getItems().size());
    }
}
```

报错的话

尝试:

pom.xml

```
<!--&lt;!&ndash; https://mvnrepository.com/artifact/org.hibernate/hibernate-entitymanager
&ndash;&gt;-->
        <!--<dependency>-->
            <!--<groupId>org.hibernate</groupId>-->
            <!--<artifactId>hibernate-entitymanager</artifactId>-->
            <!--<version>5.3.1.Final</version>-->
        <!--</dependency>-->

        <!-- https://mvnrepository.com/artifact/hibernate/hibernate-entitymanager -->
        <dependency>
            <groupId>hibernate</groupId>
            <artifactId>hibernate-entitymanager</artifactId>
            <version>3.4.0.GA</version>
            <type>pom</type>
        </dependency>
```

六 JPQL

JPQL语言

- JPQL语言，即 Java Persistence Query Language 的简称。JPQL 是一种和 SQL 非常类似的中间性和对象化查询语言，它最终会被编译成针对不同底层数据库的 SQL 查询，从而屏蔽不同数据库的差异。
- JPQL语言的语句可以是 select 语句、update 语句或delete语句，它们都通过 Query 接口封装执行

javax.persistence.Query

- Query接口封装了执行数据库查询的相关方法。调用 EntityManager 的 createQuery、create NamedQuery 及 createNativeQuery 方法可以获得查询对象，进而可调用 Query 接口的相关方法来执行查询操作。
- Query接口的主要方法
  - `int executeUpdate()` 用于执行update或delete语句。
  - `List getResultList()` 用于执行select语句并返回结果集实体列表。
  - `Object getSingleResult()` 用于执行只返回单个结果实体的select语句。
  - `Query setFirstResult(int startPosition)` 用于设置从哪个实体记录开始返回查询结果。
  - `Query setMaxResults(int maxResult)` 用于设置返回结果实体的最大数。与setFirstResult结合使用可实现分页查询。
  - `Query setFlushMode(FlushModeType flushMode)` 设置查询对象的Flush模式。参数可以取2个枚举值：FlushModeType.AUTO 为自动更新数据库记录，FlushMode Type.COMMIT 为直到提交事务时才更新数据库记录。
  - `setHint(String hintName, Object value)` 设置与查询对象相关的特定供应商参数或提示信息。参数名及其取值需要参考特定 JPA 实现库提供商的文档。如果第二个参数无效将抛出 IllegalArgumentException异常。
  - `setParameter(int position, Object value)` 为查询语句的指定位置参数赋值。Position 指定参数序号，value 为赋给参数的值。
  - `setParameter(int position, Date d, TemporalType type)` 为查询语句的指定位置参数赋 Date 值。Position 指定参数序号，value 为赋给参数的值，temporalType 取 TemporalType 的枚举常量，包括 DATE、TIME 及 TIMESTAMP 三个，，用于将 Java 的 Date 型值临时转换为数据库支持的日期时间类型（java.sql.Date、java.sql.Time及java.sql.Timestamp）。
  - `setParameter(int position, Calendar c, TemporalType type)` 为查询语句的指定位置参数赋 Calenda r值。position 指定参数序号，value 为赋给参数的值，temporalType 的含义及取舍同前。
  - `setParameter(String name, Object value)` 为查询语句的指定名称参数赋值。
  - `setParameter(String name, Date d, TemporalType type)` 为查询语句的指定名称参数赋 Date 值。用法同前。
  - `setParameter(String name, Calendar c, TemporalType type)` 为查询语句的指定名称参数设置 Calendar值。name为参数名，其它同前。该方法调用时如果参数位置或参数名不正确，或者所赋的参数值类型不匹配，将抛出 IllegalArgumentException 异常。

JPQL语句

select语句用于执行查询。其语法可表示为：

- select_clause
- form_clause
- [where_clause]
- [groupby_clause]
- [having_clause]
- [orderby_clause]

select-from 子句

- from 子句是查询语句的必选子句。
- Select 用来指定查询返回的结果实体或实体的某些属性
- From 子句声明查询源实体类，并指定标识符变量（相当于SQL表的别名）。
- 如果不希望返回重复实体，可使用关键字 distinct 修饰。select、from 都是 JPQL 的关键字，通常全大写或全小写，建议不要大小写混用。

查询所有实体

- 查询所有实体的 JPQL 查询字串很简单，例如：
  select o from Order o 或 select o from Order as o
- 关键字 as 可以省去。
- 标识符变量的命名规范与 Java 标识符相同，且区分大小写。
- 调用 EntityManager 的 createQuery() 方法可创建查询对象，接着调用 Query 接口的 getResultList() 方法就可获得查询结果集。

where 子句

- JPQL也支持包含参数的查询，例如：
  - select o from Orders o where o.id = :myId
  - select o from Orders o where o.id = :myId and o.customer = :customerName
- 注意：参数名前必须冠以冒号(`:`)，执行查询前须使用Query.setParameter(name, value)方法给参数赋值。
- 也可以不使用参数名而使用参数的序号，例如：
  - select o from Order o where o.id = ?1 and o.customer = ?2
  - 其中 ?1 代表第一个参数，?2 代表第一个参数。在执行查询之前需要使用重载方法 Query.setParameter(pos, value) 提供参数值。
    uery query = entityManager.createQuery("select o from Orders o where o.id = ?1 and o.customer = ?2");
    query.setParameter(1, 2);
    query.setParameter(2, "John");
    List orders = query.getResultList();
    … …

查询部分属性

- 如果只须查询实体的部分属性而不需要返回整个实体。例如：
  select o.id, o.customerName, o.address.streetNumber from Order o order by o.id
- 执行该查询返回的不再是Orders实体集合，而是一个对象数组的集合(Object[])，集合的每个成员为一个对象数组，可通过数组元素访问各个属性。

使用 Hibernate 的查询缓存

```
String jpql = "FROM Customer c WHERE c.age > ?1";
        Query query = entityManager.createQuery(jpql).setHint(QueryHints.HINT_CACHEABLE, true);

        //占位符的索引是从 1 开始
        query.setParameter(1, 1);
        List<Customer> customers = query.getResultList();
        System.out.println(customers.size());

        query = entityManager.createQuery(jpql).setHint(QueryHints.HINT_CACHEABLE, true);
```

order by子句

- order by子句用于对查询结果集进行排序。和SQL的用法类似，可以用 “asc“ 和 "desc“ 指定升降序。如果不显式注明，默认为升序。
  select o from Orders o order by o.id
  select o from Orders o order by o.address.streetNumber desc
  select o from Orders o order by o.customer asc, o.id desc

group by子句与聚合查询

- group by 子句用于对查询结果分组统计，通常需要使用聚合函数。常用的聚合函数主要有 AVG、SUM、COUNT、MAX、MIN 等，它们的含义与SQL相同。例如：
  select max(o.id) from Orders o
- 没有 group by 子句的查询是基于整个实体类的，使用聚合函数将返回单个结果值，可以使用Query.getSingleResult()得到查询结果。例如：

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

having子句

- Having 子句用于对 group by 分组设置约束条件，用法与where 子句基本相同，不同是 where 子句作用于基表或视图，以便从中选择满足条件的记录；having 子句则作用于分组，用于选择满足条件的组，其条件表达式中通常会使用聚合函数。
- 例如，以下语句用于查询订购总数大于100的商家所售商品及数量：
  select o.seller, o.goodId, sum(o.amount) from V_Orders o group by
  o.seller, o.goodId having sum(o.amount) > 100
- having子句与where子句一样都可以使用参数。

子查询

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

JPQL函数

- JPQL提供了以下一些内建函数，包括字符串处理函数、算术函数和日期函数。
- 字符串处理函数主要有：
  - concat(String s1, String s2)：字符串合并/连接函数。
    - substring(String s, int start, int length)：取字串函数。
  - trim([leading|trailing|both,][char c,] String s)：从字符串中去掉首/尾指定的字符或空格。
  - ower(String s)：将字符串转换成小写形式。
  - upper(String s)：将字符串转换成大写形式。
  - engtth(String s)：求字符串的长度。
  - locate(String s1, String s2[, int start])：从第一个字符串中查找第二个字符串(子串)出现的位置。若未找到则返回0。

- 算术函数主要有 abs、mod、sqrt、size 等。Size 用于求集合的元素个数。
- 日期函数主要为三个，即 current_date、current_time、current_timestamp，它们不需要参数，返回服务器上的当前日期、时间和时戳。

update语句

- update语句用于执行数据更新操作。主要用于针对单个实体类的批量更新

delete语句

- delete语句用于执行数据更新操作。

示例代码:

- 常用API

```
package com.ifox.hgx.jpa.test;

import com.ifox.hgx.jpa.entity.Customer;
import com.ifox.hgx.jpa.entity.Order;
import org.hibernate.jpa.QueryHints;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import javax.persistence.*;
import java.util.List;

public class TestJPQL {

    private EntityManagerFactory entityManagerFactory;
    private EntityManager entityManager;
    private EntityTransaction transaction;

    @Before
    public void init() {
        entityManagerFactory = Persistence.createEntityManagerFactory("jpa_m1");
        entityManager = entityManagerFactory.createEntityManager();
        transaction = entityManager.getTransaction();
        transaction.begin();
    }

    @After
    public void destroy() {
        transaction.commit();
        entityManager.close();
        entityManagerFactory.close();
    }


    //需要注意的是位置参数的是位置前加符号”?”，命名参数是名称前是加符号”:”。
```

```
    @Test
    public void testHelloJPQL() {
//        String jpql = "FROM Customer c WHERE c.age > ?1";
//        Query query = entityManager.createQuery(jpql);
//        query.setParameter(1, 1);

        String jpql = "FROM Customer c where c.age > :age";

        Query query = entityManager.createQuery(jpql);

        query.setParameter("age", 1);
        List<Customer> customers = query.getResultList();
        System.out.println(customers.size());

        /**
         * 注意:如果是双向关联的 可能会发生:StackOverflowError
         */
//        for (Customer customer:customers) {
//            System.out.println(customer);
//        }
    }

    //默认情况下，若只查询部分属性，则将返回 Object[] 类型的结果. 或者 Object[] 类型的 List.
    //也可以在实体类中创建对应的构造器，然后再 JPQL 语句中利用对应的构造器返回实体类的对象.
    @Test
    public void testPartlyProperties() {
        String jpql = "SELECT new Customer(c.lastName, c.age) FROM Customer c WHERE c.id > ?1";
        List result = entityManager.createQuery(jpql).setParameter(1, 1).getResultList();

        System.out.println(result);
        //输出结果:[Customer{id=null, lastName='CCC', email='null', age=34, createTime=null,
birth=null, orders=[]}]
    }

    //createNamedQuery 适用于在实体类前使用 @NamedQuery 标记的查询语句
    //在实体类上:@NamedQuery(name = "testNameQuery" ,query = "select c from Customer c where
c.age = ?1")
    @Test
    public void testNameQuery() {
        Customer customer = (Customer)
entityManager.createNamedQuery("testNameQuery").setParameter(1, 34).getSingleResult();
        System.out.println(customer);
    }

    //createNativeQuery 适用于本地 SQL
    @Test
    public void test() {
        String sql = "select age from JPA_CUSTOMTERS where id = ?";
        Query query = entityManager.createNativeQuery(sql).setParameter(1, 1);
        int age = (int) query.getSingleResult();
        System.out.println("age:" + age);
    }
```

```
    //使用 hibernate 的查询缓存.
    @Test
    public void testQueryCache(){
        String jpql = "FROM Customer c WHERE c.age > ?1";
        Query query = entityManager.createQuery(jpql).setHint(QueryHints.HINT_CACHEABLE, true);

        //占位符的索引是从 1 开始
        query.setParameter(1, 1);
        List<Customer> customers = query.getResultList();
        System.out.println(customers.size());

        query = entityManager.createQuery(jpql).setHint(QueryHints.HINT_CACHEABLE, true);

        //占位符的索引是从 1 开始
        query.setParameter(1, 1);
        customers = query.getResultList();
        System.out.println(customers.size());
    }

    //查询 order 数量大于 2 的那些 Customer
    @Test
    public void testGroupBy(){
        String jpql = "SELECT o.customer FROM Order o "
                + "GROUP BY o.customer "
                + "HAVING count(o.id) >= 2";
        List<Customer> customers = entityManager.createQuery(jpql).getResultList();

        System.out.println(customers);
    }

    @Test
    public void testOrderBy(){
        String jpql = "FROM Customer c WHERE c.age > ?1 ORDER BY c.age DESC";
        Query query = entityManager.createQuery(jpql).setHint(QueryHints.HINT_CACHEABLE, true);

        //占位符的索引是从 1 开始
        query.setParameter(1, 1);
        List<Customer> customers = query.getResultList();
        System.out.println(customers.size());
    }

    /**
     * JPQL 的关联查询同 HQL 的关联查询.
     *
     * LEFT OUTER JOIN FETCH c.orders
     *
     * FETCH 很关键 会自动封装类
     */
    @Test
    public void testLeftOuterJoinFetch(){
        String jpql = "select c FROM Customer  c LEFT OUTER JOIN FETCH c.orders  WHERE c.id = ?
1";
```

```
        Customer customer =
                (Customer) entityManager.createQuery(jpql).setParameter(1, 1).getSingleResult();
        System.out.println(customer);
        System.out.println(customer.getLastName());
        System.out.println(customer.getOrders().size());

//      List<Object[]> result = entityManager.createQuery(jpql).setParameter(1,
12).getResultList();
//      System.out.println(result);
    }

    @Test
    public void testSubQuery(){
        //查询所有 Customer 的 lastName 为 YY 的 Order
        String jpql = "SELECT o FROM Order o "
                + "WHERE o.customer = (SELECT c FROM Customer c WHERE c.lastName = ?1)";

        Query query = entityManager.createQuery(jpql).setParameter(1, "KKK");
        List<Order> orders = query.getResultList();
        System.out.println(orders.size());
    }

    //使用 jpql 内建的函数
    @Test
    public void testJpqlFunction(){
        String jpql = "SELECT lower(c.email) FROM Customer c";

        List<String> emails = entityManager.createQuery(jpql).getResultList();
        System.out.println(emails);
    }

    //可以使用 JPQL 完成 UPDATE 和 DELETE 操作.
    @Test
    public void testExecuteUpdate(){
        String jpql = "UPDATE Customer c SET c.lastName = ?1 WHERE c.id = ?2";
        Query query = entityManager.createQuery(jpql).setParameter(1, "YYY").setParameter(2, 1);

        query.executeUpdate();
    }



}
```

- 实现分页排序查询

```
public CarManagementPageDTOS page(CarManagementPageRequest pageRequest) {
```

```
    String jpql = "select new com.enduser.dto.CarManagementPage(ee.userName, ec.id,
ec.endUserId, ec.parkedStatus,ec.plateNumber, ec.status, ecc.vin,ecc.imgUrl)" + "from
com.entity.enduer.EndUserCarInfoEO ec left join EndUserCarCeEO ecc on ec.id =
ecc.endUserCarInfoId left join com.entity.enduer.EndUserEO ee on ec.endUserId = ee.id where 1=1
";

    //拼装条件
    if (pageRequest.getUserName() != null) {
        jpql = jpql + " and ec.endUserId in ( SELECT id FROM com.entity.enduer.EndUserEO er
WHERE er.userName LIKE :userName) ";
    }
    if (pageRequest.getPlateNumber() != null) {
        jpql = jpql + " and ec.plateNumber  like :plateNumber ";
    }

    //排序
    jpql = jpql + " order by ec.modifyDate DESC  ";

    //创建jpql查询
    Query query = entityManager.createQuery(jpql);

    //设置参数
    if (pageRequest.getUserName() != null) {
        query.setParameter("userName", "%" + pageRequest.getUserName() + "%");
    }
    if (pageRequest.getPlateNumber() != null) {
        query.setParameter("plateNumber", "%" + pageRequest.getPlateNumber() + "%");
    }

    //得到总的记录数
    int totalCount = query.getResultList().size();


    /**
    public class PageDetail {
        //总记录数
        private int totalCount;
        //当页数量
        private int pageSize;
        //页码
        private int pageNo;
        //....
    */

    //设置分页信息
    PageResponseDetail pageResponseDetail = new PageResponseDetail();
    pageResponseDetail.setPageNo(pageRequest.getPageNo() - 1);
    pageResponseDetail.setPageSize(pageRequest.getPageSize());
    pageResponseDetail.setTotalCount(totalCount);

    //得到查询内容的实体
```

```
    List<CarManagementPageDTO> carManagementPageDTOList =
query.setFirstResult((pageRequest.getPageNo() - 1) *
pageRequest.getPageSize()).setMaxResults(pageRequest.getPageSize()).getResultList();

    /**
        public class CarManagementPageDTOS {
            //查询信息封装的实体
            private List<CarManagementPageDTO> carManagementPageDTOList ;
            //分页实体
            private PageResponseDetail pageResponseDetail ;
            //....
    */
    CarManagementPageDTOS carManagementPageDTOS = new
CarManagementPageDTOS(carManagementPageDTOList, pageResponseDetail);
    //封装了查询的信息,分页信息
    return carManagementPageDTOS;
}
```

七 Spring 整合 JPA

三种整合方式

- LocalEntityManagerFactoryBean：适用于那些仅使用 JPA 进行数据访问的项目，该 FactoryBean 将根据JPA PersistenceProvider 自动检测配置文件进行工作，一般从“META-INF/persistence.xml”读取配置信息，这种方式最简单，但不能设置 Spring 中定义的DataSource，且不支持 Spring 管理的全局事务
- 从JNDI中获取：用于从 Java EE 服务器获取指定的EntityManagerFactory，这种方式在进行 Spring 事务管理时一般要使用 JTA 事务管理
- LocalContainerEntityManagerFactoryBean：适用于所有环境的 FactoryBean，能全面控制 EntityManagerFactory 配置,如指定 Spring 定义的 DataSource 等等。

示例:

- pom.xml

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

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.ifox.hgx</groupId>
  <artifactId>jpa_ Integrated_spring</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging>

  <name>jpa_ Integrated_spring Maven Webapp</name>
  <!-- FIXME change it to the project's website -->
```

```
  <url>http://www.example.com</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>

  <dependencies>
    <!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>5.0.3.RELEASE</version>
    </dependency>

    <!-- https://mvnrepository.com/artifact/org.springframework/spring-core -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>5.0.3.RELEASE</version>
    </dependency>

    <!-- https://mvnrepository.com/artifact/org.springframework/spring-beans -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-beans</artifactId>
      <version>5.0.3.RELEASE</version>
    </dependency>

    <!-- https://mvnrepository.com/artifact/org.springframework/spring-web -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-web</artifactId>
      <version>5.0.3.RELEASE</version>
    </dependency>

    <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>5.0.3.RELEASE</version>
    </dependency>

    <!-- https://mvnrepository.com/artifact/org.springframework/spring-jdbc -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-jdbc</artifactId>
      <version>5.0.3.RELEASE</version>
    </dependency>

    <!-- https://mvnrepository.com/artifact/org.springframework/spring-orm -->
    <dependency>
```

```
      <groupId>org.springframework</groupId>
      <artifactId>spring-orm</artifactId>
      <version>5.0.3.RELEASE</version>
    </dependency>


    <!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>5.2.13.Final</version>
    </dependency>

    <!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-entitymanager -->
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-entitymanager</artifactId>
      <version>5.2.13.Final</version>
    </dependency>

    <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.38</version>
    </dependency>

    <!-- https://mvnrepository.com/artifact/com.mchange/c3p0 -->
    <dependency>
      <groupId>com.mchange</groupId>
      <artifactId>c3p0</artifactId>
      <version>0.9.5.2</version>
    </dependency>


    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

- 数据库连接信息:db.properties

```
jdbc.url = jdbc:mysql://localhost:3306/JPA
jdbc.user = root
jdbc.password = 123456
jdbc.driverClass = com.mysql.jdbc.Driver
```

- spring配置:applicationContext.xml

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


    <!--配置自动扫描的包-->
    <context:component-scan base-package="com.ifox.hgx.jpa.spring"></context:component-scan>

    <!--配置 c3p0数据源-->
    <context:property-placeholder location="classpath:db.properties"></context:property-
placeholder>

    <bean class="com.mchange.v2.c3p0.ComboPooledDataSource" id="dataSource">
        <property name="jdbcUrl" value="${jdbc.url}"></property>
        <property name="driverClass" value="${jdbc.driverClass}"></property>
        <property name="user" value="${jdbc.user}"></property>
        <property name="password" value="${jdbc.password}"></property>

    </bean>

    <!-- 配置 EntityManagerFactory -->
    <bean id="entityManagerFactory"
          class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="dataSource"></property>
        <!-- 配置 JPA 提供商的适配器. 可以通过内部 bean 的方式来配置 -->
        <property name="jpaVendorAdapter">
            <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"></bean>
        </property>
        <!-- 配置实体类所在的包 -->
        <property name="packagesToScan" value="com.ifox.hgx.jpa.spring.entities"></property>
        <!-- 配置 JPA 的基本属性. 例如 JPA 实现产品的属性 -->
        <property name="jpaProperties">
            <props>
                <prop key="hibernate.show_sql">true</prop>
                <prop key="hibernate.format_sql">true</prop>
                <prop key="hibernate.hbm2ddl.auto">update</prop>
            </props>
        </property>
    </bean>

    <!-- 配置 JPA 使用的事务管理器 -->
    <bean id="transactionManager"
          class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory"></property>
```

```
    </bean>

    <!-- 配置支持基于注解是事务配置 -->
    <tx:annotation-driven transaction-manager="transactionManager"/>
</beans>
```

- Person.java 实体类

```
@Table(name = "JPA_PERSON")
@Entity
public class Person {
    private Integer id ;
    private String lastName ;
    private String email ;
    private Integer age ;

    public Person() {
    }

    public Person(String lastName, String email, Integer age) {
        this.lastName = lastName;
        this.email = email;
        this.age = age;
    }

    //    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @GeneratedValue
    @Id
    public Integer getId() {
        return id;
    }

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

    @Column(name = "LAST_NAME")
    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
```

```
    public Integer getAge() {
        return age;
    }

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

    @Override
    public String toString() {
        return "Person{" +
                "id=" + id +
                ", lastName='" + lastName + '\'' +
                ", email='" + email + '\'' +
                ", age=" + age +
                '}';
    }
}
```

- Dao层:PersonDao

```
package com.ifox.hgx.jpa.spring.dao;

import com.ifox.hgx.jpa.spring.entities.Person;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Repository
public class PersonDao {

    //标注成员变量
    @PersistenceContext
    private EntityManager entityManager ;

    @Transactional
    public void save(Person person){
        entityManager.persist(person);
    }
}
```

- Service层: PersonService

```
package com.ifox.hgx.jpa.spring.service;

import com.ifox.hgx.jpa.spring.dao.PersonDao;
```

```
import com.ifox.hgx.jpa.spring.entities.Person;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class PersonService {

    @Autowired
    private PersonDao personDao ;

    public void save(Person person, Person person2){
        personDao.save(person);
        personDao.save(person2);
    }

}
```

- 测试类:JPATest

```
package com.ifox.hgx.jap.spring;

import com.ifox.hgx.jpa.spring.entities.Person;
import com.ifox.hgx.jpa.spring.service.PersonService;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import javax.sql.DataSource;
import java.sql.Connection;

public class JPATest {

    private ApplicationContext applicationContext = null ;
    private PersonService personService = null ;

    {
        applicationContext = new ClassPathXmlApplicationContext("applicationContext.xml") ;
        personService = applicationContext.getBean(PersonService.class) ;
    }

    @Test
    public void testDataSource() throws Exception{
        DataSource dataSource = applicationContext.getBean(DataSource.class) ;
        Connection connection = dataSource.getConnection();
        System.out.println(connection);

    }

    @Test
    public void testPersonSave(){
        Person person = new Person("AAA","12132@qq.com",13) ;
```

```
        Person person1 = new Person("BBB","12312@qq.com",23) ;

        personService.save(person,person1);


    }
}
```

- 目录结构: