JPA前言

介绍

```
JPA 的始作俑者就是 Hibernate 的作者，–Hibernate 从 3.2 开始兼容 JPA

JPA本质上是一种ORM规范，不是ORM框架，类似于JDBC和Mysql的JDBC驱动。

因为JPA并未提供ORM实现，它只是制定了一些规范，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，
现在流行的实现产品有Hibernate 3.2、TopLink 10.1、OpenJPA都提供了实现JPA的实现。
```

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、GROUNP 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

三大核心技术

- **ORM映射元数据**

  JPA支持XML和JDK5.0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续化到数据表中

- **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

- **JPQL（查询语言）**

  这是持续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

使用步骤

1. **导入相关重要Jar包**
    - mysql的jdbc驱动

```
<!--mysql的jdbc驱动-->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>6.0.6</version>
</dependency>
```

    - JPA规范的jar包

```
<dependency>
    <groupId>org.hibernate.javax.persistence</groupId>
    <artifactId>hibernate-jpa-2.1-api</artifactId>
    <version>1.0.0.Final</version>
</dependency>
```

    - Hibernate核心包

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>4.3.11.Final</version>
</dependency>
```

    - Hibernate针对JPA的实现包

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>4.3.11.Final</version>
</dependency>
```

    - ehcache的二级缓存框架

```
<!--ehcache二级缓存-->
<dependency>
    <groupId>net.sf.ehcache</groupId>
    <artifactId>ehcache</artifactId>
    <version>2.4.3</version>
</dependency>

<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-ehcache</artifactId>
    <version>4.2.4.Final</version>
</dependency>
```

    - slf4j的日志框架

2. **创建persistence.xml文件，在这个文件中配置持续化单元**

- 需要指定哪个数据库进行交互
- 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

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
             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_2_0.xsd">
    <persistence-unit name="jpa-1" transaction-type="RESOURCE_LOCAL">
        <!--
        配置使用什么 ORM 产品来作为 JPA 的实现
        1. 实际上配置的是  javax.persistence.spi.PersistenceProvider 接口的实现类
        2. 若 JPA 项目中只有一个 JPA 的实现产品, 则也可以不配置该节点.
        -->
        <provider>org.hibernate.ejb.HibernatePersistence</provider>

        <!-- 添加持久化类 -->
        <class>data.ClazzEntity</class>
        <class>data.StudentEntity</class>
        <class>data.TeacherEntity</class>


        <!--
        配置二级缓存的策略
        ALL: 所有的实体类都被缓存
        NONE: 所有的实体类都不被缓存.
        ENABLE_SELECTIVE: 标识 @Cacheable(true) 注解的实体类将被缓存
        DISABLE_SELECTIVE: 缓存除标识 @Cacheable(false) 以外的所有实体类
        UNSPECIFIED: 默认值, JPA 产品默认值将被使用
        -->
        <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>

        <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:///test"/>
            <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 -->
            <!--格式化SQL-->
            <property name="hibernate.format_sql" value="true"/>
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.hbm2ddl.auto" value="update"/>

            <!-- 二级缓存相关 -->
```

```
            <property name="hibernate.cache.use_second_level_cache" value="true"/>
            <property name="hibernate.cache.region.factory_class"
                      value="org.hibernate.cache.ehcache.EhCacheRegionFactory"/>
            <property name="hibernate.cache.use_query_cache" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

3. **创建实体类，使用注解来描述实体类跟数据库表之间的映射关系**

```
@Entity
@Table(name = "class", schema = "test", catalog = "")
public class ClazzEntity {
    private int id;
    private String name;
    private int age;

    @Id
    @Column(name = "id", nullable = false)
    public int getId() {
        return id;
    }

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

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

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

    @Basic
    @Column(name = "age", nullable = false)
    public int getAge() {
        return age;
    }

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

1. **使用 JPA API完成数据增加、删除、修改、和查询操作**

JPA的基本注解

@Entity

@Entity注解用于实体类声明之前，指定该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，该注解与@Entity注解并列使用，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。

@Id

用于声明一个实体类的属性映射为数据库的主键列。该注释亦可置于属性的getter方法之前。

@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自动选择一个最合适底层数据库的主键生成策略。SQLServer对应identity、MySql对应auto increment。

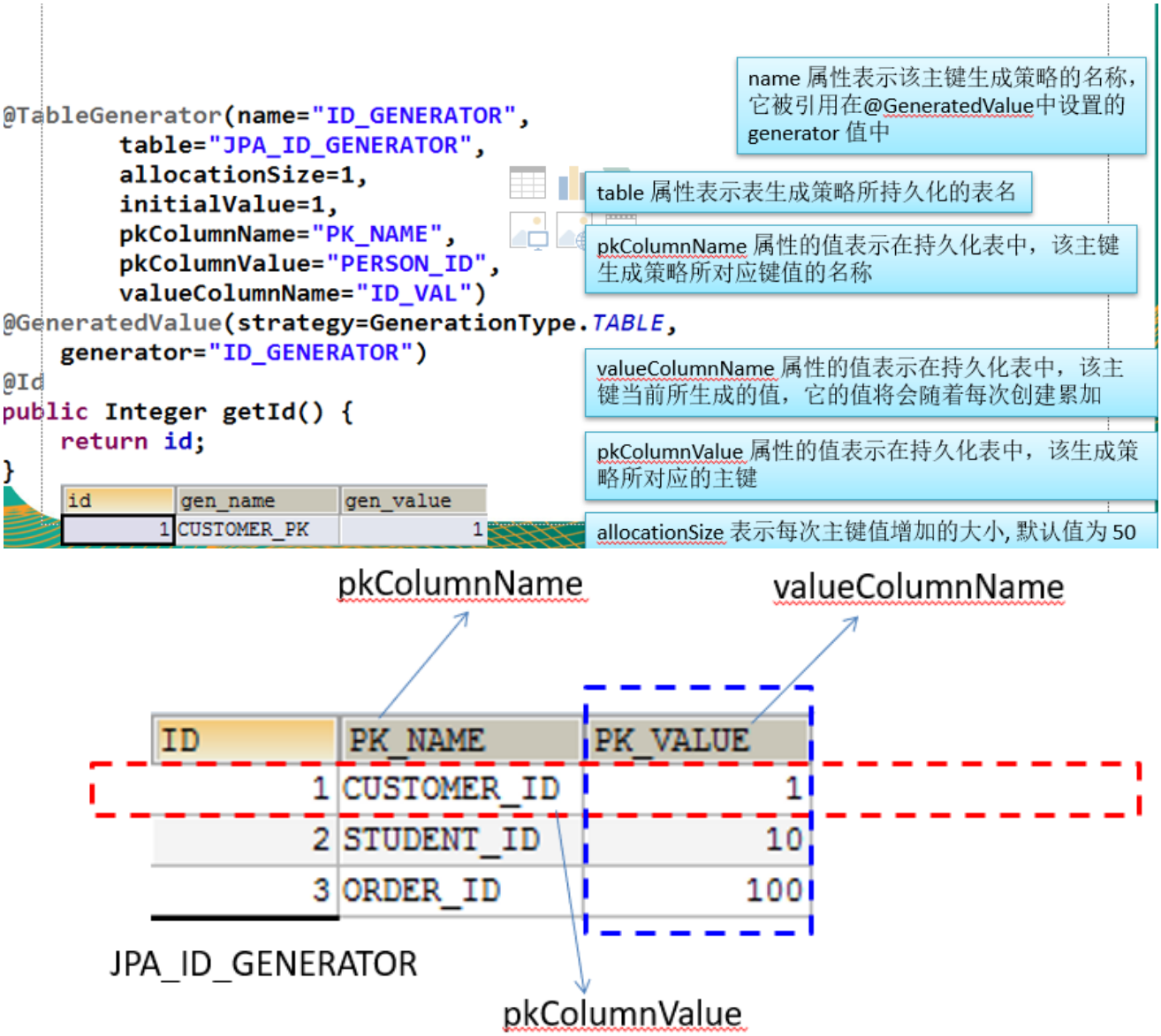
可选的主键生成策略有 ：

- **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，是默认的选择。
- **sequence**：通过序列产生主键，通过@SequenceGenerrator注解指定序列名，Mysql不支持这种方式。
- **table**：通过其他表产生主键，框架借由表模拟序列产生主键，使用该策略可以应用更易于数据库移植。

用table来生成主键

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后经过一定的算法后来获得。

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兼容造成的问题。



通过pkColumnName和pkColumnValue确定ID表中的一行，再通过valueColumnName确定主键列。随即每次实体表生成主键时都会查询该ID表中的主键列，再通过一定的算法后插入到实体表的主键列，而ID表中主键列也会自增长。

@Column

该实体的属性与其映射的数据库的列不同名时需要使用该注解。该属性通常置于实体的属性声明语句之前，还可与@Id注解一起使用。

@Basic

用于设置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getXxx()方法，默认即为@Basic

@Transient

表示该属性并非一个到数据库表的字段的映射，ORM框架将会忽略该属性。

@Temporal

在核心的 Java API中并没有定义Date类型的精度，而在数据库中，表示Date类型的数据由DATE、TIME、TIMESTAMP三种精度。在进行时间属性映射时可使用@Temporal注解来调整精度。

相关类和API

Persistence

用于获取EntityManagerFactory实例。

```
        Map<String,Object> propertis=new HashMap<String, Object>();
        //打开二级缓存
        propertis.put("hibernate.cache.use_second_level_cache",true);
        entityManagerFactory = Persistence.createEntityManagerFactory("jpa-
1",propertis);
```

EntityManagerFactory

主要用来创建EntityManager实例

EntityManager

再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EntityBean，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：
    1. **新建状态**：新创建的对象，尚未拥有持久性主键且并未与持久化建立上下文环境，可看作一个普通的Entity Bean实体，对该对象进行变化再提交不会影响到底层数据库数据。
    2. **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，对该对象进行变化再提交会影响到底层数据库数据。
    3. **游离状态**：拥有持久化主键，但未与持久化建立上下文环境，对该对象进行变化再提交不会影响到底层数据库数据。
    4. **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O/R映射的底层数据在数据库已被删除。

- 实体对象如果是**新建状态**，persist()方法会将转换为持久化状态。
- 实体对象已经处于**持久化状态**，则 persist() 方法什么都不做。
- 实体对象是删除状态，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)。

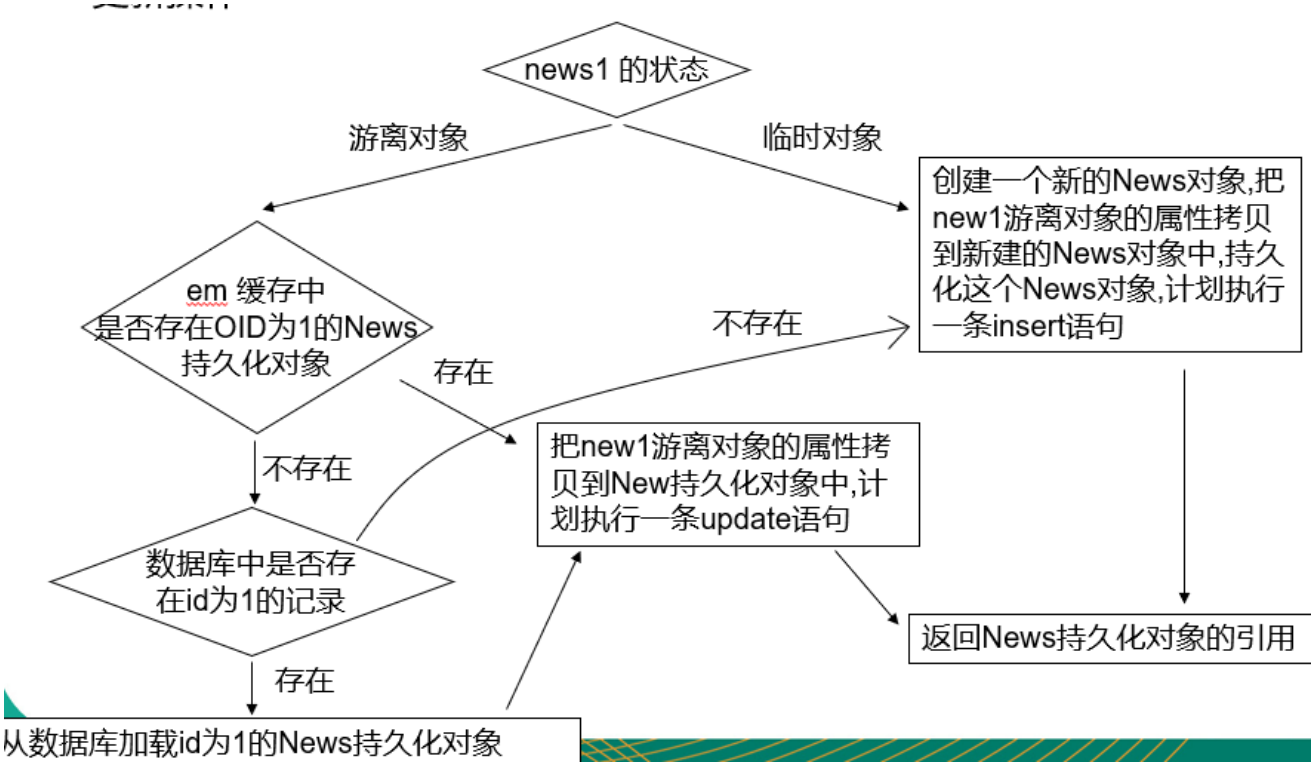
- 实体对象是游离对象，若entityManager**存在**该持续化对象，则将游离对象属性赋值到持久化对象后执行update操作，返回持久化对象引用。

- 实体对象是游离对象，若entityManager**不存在**该持续化对象，那么同游离对象中的主键从数据库中查询。

  若查询存在该主键存在的数据则创建一个新的对象并将游离对象的数据拷贝到新对象，然后持久化该新对象并执行update操作。并返回持久化对象实例引用。

  若不存在，则同步上面操作，但执行的是insert操作。

- 实体对象是新建对象，创建一个新的对象并将游离对象的数据拷贝到新对象，然后持久化该新对象并执行insert操作。并返回持久化对象实例引用。



EntityTransaction

> 用来管理资源层实体管理器的事务操作。

映射关联关系

单向多对一

单表

```
@Table(name = "customer")
@Entity
public class CustomerEntity {
    private Integer id;
    private String lastName;

    @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;
    }
}
```

多表

```
@Table(name = "order",schema = "jpa")
@Entity
public class OrderEntity {

    private Integer id;
    private String orderName;
    private CustomerEntity customerEntity;

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

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

    @Column(name = "ORDER_NAME",nullable = false,length = 10)
    public String getOrderName() {
        return orderName;
    }

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


    @JoinColumn(name ="CUSTOMER_ID" )
    @ManyToOne(targetEntity = CustomerEntity.class,fetch = FetchType.LAZY)
    public CustomerEntity getCustomerEntity() {
        return customerEntity;
    }

    public void setCustomerEntity(CustomerEntity customerEntity) {
        this.customerEntity = customerEntity;
    }
}
```

注意

- **对Entity Bean进行persist时需要注意顺序**

  若先对多表对象进行persist，那么有N个多表对象就会多出N条update语句，因为多表对象在进行persist时单表对象还未进行persist，所以多表对象中的外键列插入的时null值，轮到单表对象进行persist时，多方为了维护两者的关联关系，故仍需在对多表对象进行update更新，将外键值替换成单表对象中的主键值。

所以建议在对多对一关联关系的新建对象进行persite时，先对One端进行persist

单向一对多

单表

```
@Table(name = "customer")
@Entity
public class CustomerEntity {
    private Integer id;
    private String lastName;
    private List<OrderEntity> orderEntities;

    @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;
    }

    @JoinColumn(name ="CUSTOMER_ID" )
    @OneToMany
    public List<OrderEntity> getOrderEntities() {
        return orderEntities;
    }

    public void setOrderEntities(List<OrderEntity> orderEntities) {
        this.orderEntities = orderEntities;
    }
}
```

多表

```java
@Table(name = "order",schema = "jpa")
@Entity
public class OrderEntity {//Many端

    private Integer id;
    private String orderName;

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

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

    @Column(name = "ORDER_NAME",nullable = false,length = 10)
    public String getOrderName() {
        return orderName;
    }

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

同理使用@OneToMany注解让单表维护关联关系，但不同的是，@JoinColumn注解是指定多表的外键列，而不是添加列。

且在对所有关系表进行persist时，都会存在对多表进行update。

双向多对一

> 两边同时维护关联关系（错误）

单表

```java
@Table(name = "customer")
@Entity
public class CustomerEntity {
    private Integer id;
    private List<OrderEntity> orderEntities;


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

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

    @JoinColumn(name = "CUSTOMER_ID")
    @OneToMany
    public List<OrderEntity> getOrderEntities() {
        return orderEntities;
    }

    public void setOrderEntities(List<OrderEntity> orderEntities) {
        this.orderEntities = orderEntities;
    }


}
```

多表

```
@Table(name = "order",schema = "jpa")
@Entity
public class OrderEntity {
    private Integer id;
    private CustomerEntity customerEntity;

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

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


    @JoinColumn(name ="CUSTOMER_ID" )
    @ManyToOne(targetEntity = CustomerEntity.class,fetch = FetchType.EAGER,cascade =
{CascadeType.REMOVE} )
    public CustomerEntity getCustomerEntity() {
        return customerEntity;
    }

    public void setCustomerEntity(CustomerEntity customerEntity) {
        this.customerEntity = customerEntity;
    }

}
```

若两端同时维护关联关系的话，在对新建对象进行persiste时可能会出现的情况。

- **先persiste单表对象**

  有N个多表对象就会多出N条update语句。

因为在多表对象进行persist之前，多表对象已经存在单表对象的主键（也就是已经单向维护了关联关系），所以多表对象不需要再重新执行update更新。但由于单表对象在维护关联关系，故单表一方需要额外对单表对象中的所有多表对象执行update。

- **先persiste多表对象**

  有N个多表对象就会多出N+1条update语句。

多表一方的情况跟单向多对一关联关系时一致，也是由于多表对象执行persist时未关联到单表数据，故在单表对象persist之后，多表一方为了维护关联需要额外执行update，也就是N条update语句。

而另外的1条update语句就是单表一方最终也是为了维护所有多表对象关联关系而执行的update语句。

> 多表维护关联关系（正确）

单表

```
@Table(name = "customer")
@Entity
public class CustomerEntity {
    private Integer id;
    private List<OrderEntity> orderEntities;


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

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


    @OneToMany(mappedBy = "customerEntity")
    public List<OrderEntity> getOrderEntities() {
        return orderEntities;
    }

    public void setOrderEntities(List<OrderEntity> orderEntities) {
        this.orderEntities = orderEntities;
    }


}
```

多表

```
@Table(name = "order",schema = "jpa")
@Entity
```

```
public class OrderEntity {
    private Integer id;
    private CustomerEntity customerEntity;

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

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


    @JoinColumn(name ="CUSTOMER_ID" )
    @ManyToOne(targetEntity = CustomerEntity.class,fetch = FetchType.EAGER,cascade =
{CascadeType.REMOVE} )
    public CustomerEntity getCustomerEntity() {
        return customerEntity;
    }

    public void setCustomerEntity(CustomerEntity customerEntity) {
        this.customerEntity = customerEntity;
    }

}
```

优势：这样的话就不会存在update的SQL语句，故建议在双向多对一情况时，单表放弃维护关联关系。

双向一对一

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使用@JoinColumn建立外键列指向关系被维护端的主键列。

维持关联关系一方

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

    private Integer id;
    private String name;
    private ManagerEntity managerEntity;


    @Column(name = "DEPARTMENT_ID")
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Id
    public Integer getId() {
```

```
        return id;
    }

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

    @Column(name = "DEPARTMENT_NAME")
    public String getName() {
        return name;
    }

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

    @JoinColumn(name = "MANAGER_ID", unique = true)
    @OneToOne
    public ManagerEntity getManagerEntity() {
        return managerEntity;
    }

    public void setManagerEntity(ManagerEntity managerEntity) {
        this.managerEntity = managerEntity;
    }

}
```

没有维持关联关系一方

```
@Table(name = "Manager")
@Entity
public class ManagerEntity {
    private Integer id;
    private String name;
    private DepartmentEntity departmentEntity;

    @Column(name = "MANAGER_ID")
    @GeneratedValue
    @Id
    public Integer getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }
    @Column(name = "MANAGER_NAME")
    public void setName(String name) {
```

```
        this.name = name;
    }

    @OneToOne(mappedBy = "managerEntity")
    public DepartmentEntity getDepartmentEntity() {
        return departmentEntity;
    }

    public void setDepartmentEntity(DepartmentEntity departmentEntity) {
        this.departmentEntity = departmentEntity;
    }
}
```

建议

1. **persist持久化对象时，先持久化没有维持关联关系对象，这样不会额外出现update语句**
2. **如果被维护表设置了懒加载后，即使没有调用到其他表数据，仍会发送两条SQL查询语句,而直接加载只有一条SQL，所以如果查询被维护表时建议使用直接加载策略。**

双向多对多

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

维持关联关系对象

```
@Table(name = "Category")
@Entity
public class CategoryEntity {
    private Integer id;
    private String name;
    private Set<ItemEntity> itemEntities = new HashSet<ItemEntity>();

    @Column(name = "CATEGORY_ID")
    @GeneratedValue
    @Id
    public Integer getId() {
        return id;
    }

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

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

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

```
    }

    @JoinTable(name = "CATEGORY_ITME",
            joinColumns = {@JoinColumn(name = "CATEGORY_ID", referencedColumnName =
"CATEGORY_ID")},
            inverseJoinColumns = {@JoinColumn(name = "ITEM_ID", referencedColumnName =
"ITEM_ID")}
    )
    @ManyToMany
    public Set<ItemEntity> getItemEntities() {
        return itemEntities;
    }

    public void setItemEntities(Set<ItemEntity> itemEntities) {
        this.itemEntities = itemEntities;
    }
}
```

没有关联关系对象

```
@Table(name = "ITEM")
@Entity
public class ItemEntity {
    private Integer id;
    private String name;
    private Set<CategoryEntity> categoryEntities=new HashSet<CategoryEntity>();

    @Column(name = "ITEM_ID")
    @GeneratedValue
    @Id
    public Integer getId() {
        return id;
    }

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

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

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

    @ManyToMany(mappedBy = "itemEntities")
    public Set<CategoryEntity> getCategoryEntities() {
        return categoryEntities;
    }

    public void setCategoryEntities(Set<CategoryEntity> categoryEntities) {
```

```
        this.categoryEntities = categoryEntities;
    }
}
```

- name=中间表名称
- joinColumns=@joinColumn(name="本类在中间表中的外键字段名",
  referencedColumnName="本类与外键对应的主键字段名")
- inversejoinColumns=@JoinColumn(name="对方类在中间表中的外键字段名",
  referencedColunName="对方类与外键对应的主键字段名")

建议总结

1. 在维护双向关联关系时，建议最好不要双方都维护关联关系，通过mappeyBy属性让其中一方放弃维护，转而让指定的一方中的属性去维护关联关系。
2. 先persist不维护一方实体对象后再persist维护一方实体对象。

使用二级缓存

介绍

在JPA中，EntityManager会对每次查询都保存在一级缓存，那么当查询数据时，首先会从一级缓存中查询是否存在，如果不存在再其访问数据库去拉去数据。

而当关闭EntityManager时就会清空所有一级缓存数据，故会重新访问数据库拉去数据。

所以二级缓存的存在就是为了跨EntityManager，将所有查询数据的生命周期同步到应用程序的生命周期，伴随程序的终结而终结，即便EntityManager关闭了，仍能再从二级缓存中去获取到数据。

使用

1. **导入Hibernate的ehcache二级缓存产品**

```
        <dependency>
            <groupId>net.sf.ehcache</groupId>
            <artifactId>ehcache</artifactId>
            <version>2.4.3</version>
        </dependency>

        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-ehcache</artifactId>
            <version>4.2.4.Final</version>
        </dependency>
```

2. **添加ehcache配置文件**

复制echache的核心jar包ehcache-core.jar其中的一个配置文件ehcache-failsage.xml到类路径下，根据需求自行修改。

3. **设置开启二级缓存**

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
             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_2_0.xsd">
    <persistence-unit name="jpa-1" transaction-type="RESOURCE_LOCAL">
```

4. **Entity Bean添加@Cacheable注解**

```
@Cacheable(value = true)
@Table(name = "Category")
@Entity
public class CategoryEntity {
```

JPQL

介绍

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。

Hibernate的SQL语法与原生的SQL语法区别

由于Hibernate是面向对象操作底层数据，所以再SQL语法上也是基于对象属性，再有些地方与原生的SQL有些许区别。

1. 查询所有实体属性，如：

```
select c from CustomerEntity as c ;    --CustomerEntity并不是数据库表名，而是实体类名
```

select c表示查询所有字段，关键字 as 可以省去。

2. 查询部分实体属性，如：

```
select c.属性名 from CustomerEntity  c   --注意是属性名而不是数据库中的字段名
```

3. 设置动态参数，如：

- 根据Name设置参数值

```
select c from CustomerEntity c where c.id=:c_id and c.age=:c_age ;
```

:参数名

- 根据索引设置参数值

```
select c from CustomerEntity c where c.id=?1 and c.age=?2 ;
```

? 索引值

API

Query接口封装了执行数据库查询的相关方法。调用EntityManager的createQuery、createNamedQuery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。

> 创建Query对象

- **创建Hibernate的SQL语法的Query对象**
  1. 传入sql字符串

```
String sql = "select c from CustomerEntity c ";
Query query = entityManager.createQuery(sql);
```

  2. 传入@NameQuery注解中的name值

Entity Bean

```
@NamedQuery(name="query_all_customer",query = "select c from CustomerEntity c
")
@Table(name = "customer")
@Entity
public class CustomerEntity {
```

创建代码

```
Query query = entityManager.createNamedQuery("query_all_customer");
```

- **创建原生的SQL语法的Query对象**

```
String sql = "select * from customer c ";
Query query = entityManager.createNativeQuery(sql);
```

> Query接口的主要方法

> JPQL函数

- **字符串处理函数**

[^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。

[^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。

。。。

- **算术处理函数**

。。。

- **日期处理函数**

  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。

整合Spring

> 在MyBatis中整合Spring的过程是把SqlSessionFactory交由Spring管理，并且MyBatis也不设置数据源，让Spring创建SqlSession并导入数据源。
>
> 同理，在JPA整合Spring中，实际就是将EntityManagerFactory交由Spring管理，当程序需要EntityManager时，让Spring管理的EntityManagerFactory去创建EntityManage。

使用步骤

1. **除了JPA规范和Hibernate相关的Jar包外，还需导入其它Spring的Jar包**

```
    <dependencies>
        <!--mysql的jdbc驱动-->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.45</version>
        </dependency>

        <!--JPA规范的jar包-->
        <dependency>
            <groupId>org.hibernate.javax.persistence</groupId>
            <artifactId>hibernate-jpa-2.1-api</artifactId>
            <version>1.0.0.Final</version>
        </dependency>

        <!--Hibernate核心包-->
        <dependency>
            <groupId>org.hibernate</groupId>
```

```
        <artifactId>hibernate-core</artifactId>
        <version>4.2.4.Final</version>
    </dependency>
    <!--Hibernate针对JPA的实现包-->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>4.2.4.Final</version>
    </dependency>

    <!--ehcache二级缓存-->
    <dependency>
        <groupId>net.sf.ehcache</groupId>
        <artifactId>ehcache</artifactId>
        <version>2.4.3</version>
    </dependency>

    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-ehcache</artifactId>
        <version>4.2.4.Final</version>
    </dependency>
    <!--junit单元测试-->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.0</version>
    </dependency>

    <!--Spring所需Jar包-->
    <!--Spring Bean-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-beans</artifactId>
        <version>4.0.0.RELEASE</version>
    </dependency>

    <!--Spring Context-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>4.0.0.RELEASE</version>
    </dependency>

    <!--Spring Core-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>4.0.0.RELEASE</version>
    </dependency>

    <!--Spring Expression-->
    <dependency>
```

```
            <groupId>org.springframework</groupId>
            <artifactId>spring-expression</artifactId>
            <version>4.0.0.RELEASE</version>
        </dependency>

        <!--Spring ORM-->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-orm</artifactId>
            <version>4.0.0.RELEASE</version>
        </dependency>

        <!--Log4j-->
        <dependency>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
            <version>1.1.1</version>
        </dependency>

        <!--Spring与log4j整合包-->
        <dependency>
            <groupId>org.apache.log4j</groupId>
            <artifactId>com.springsource.org.apache.log4j</artifactId>
            <version>1.2.16</version>
        </dependency>

        <!--C3P0数据库连接池技术框架-->
        <dependency>
            <groupId>com.mchange</groupId>
            <artifactId>c3p0</artifactId>
            <version>0.9.2.1</version>
        </dependency>

        <!--Hibernate与c3p0整合包-->
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-c3p0</artifactId>
            <version>4.2.4.Final</version>
        </dependency>

        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-nop</artifactId>
            <version>1.7.24</version>
        </dependency>
    </dependencies>
```

2. **配置Spring文件**

   在将EntityManagerFactory交由Spring管理创建的话，那么就在不需要原来的persistence.xml的配置文件，那么原来的配置现在都通过<bean/>创建时注入给EntityManagerFactory。

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

```xml
    <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/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">
        <!--扫描包-->
        <context:component-scan base-package="spring"></context:component-scan>

        <!--导入properties参数-->
        <context:property-placeholder location="classpath:jdbc_mysql.properties">
</context:property-placeholder>
        <!--配置C3P0数据源-->
        <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
            <property name="driverClass" value="${jdbc.driver}"></property>
            <property name="jdbcUrl" value="${jdbc.url}"></property>
            <property name="user" value="${jdbc.username}"></property>
            <property name="password" value="${jdbc.password}"></property>
        </bean>

        <!--配置EntityManagerFactory-->
        <bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
            <!--配置数据源-->
            <property name="dataSource" ref="dataSource"></property>
            <!--配置JPA提供商的适配器-->
            <property name="jpaVendorAdapter">
                <bean
class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"/>
            </property>
            <!--配置实体类所在包-->
            <property name="packagesToScan" value="spring.table"></property>
            <!--配置JPA的基本属性-->
            <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="jpaTransactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
            <property name="entityManagerFactory" ref="entityManagerFactory"></property>
        </bean>
        <!--配置支持基于注解时事务配置-->
        <tx:annotation-driven transaction-manager="jpaTransactionManager">
</tx:annotation-driven>
```

```
</beans>
```

3. **获取EntityManager**

   在Dao层可通过@PersistenceContext注解获取到EntityManger

```
@Repository
public class Spring_JPA_Dao {
    @PersistenceContext
    EntityManager entityManager;
```

4. **事务管理**

   通过在Spring配置文件中配置JPA的事务管理器，随后注入给Spring的注解驱动，从而在service层可通过@Transactional声明该方法必须在事务下进行。

```
@Service
public class Spring_JPA_Service {
    @Autowired
    Spring_JPA_Dao spring_jpa_dao;

    @Transactional
    public void add(Spring_JPA spring_jpa) {
        spring_jpa_dao.add(spring_jpa);
    }
}
```