# JPA概述

1. 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，它只是定制了一些规范，提供了一些编程的API接口，具体实现则由ORM厂商提供实现。
2. Hibernate是实现：Hibernate除了作为ORM框架之外，他也是一种JPA实现，因此从功能上来说，JPA是Hibernate功能的一个子集。
3. JDBC是操作数据库的规范：例如获取数据库连接，返回结果集，等等操作，而具体的实现是由每个厂商自己实现，这就是驱动，例如Mysql驱动，Oracle驱动等等。
4. **总结：JDBC是操作数据的规范，即操作数据的接口定义，具体是由数据库驱动实现，ORM框架，封装了JDBC操作，使其更易于操作数据库，例如Hibernate、TopLink等，而JPA则是抽象了ORM操作数据的规范,其直接实现的ORM框架有，Hibernate、TopLink等。**

# JPA基本操作步骤

## JPA 基本操作步骤

1. 创建persistence.xml，在这个配置文件中配置持久化单元

* 需要自动跟那个数据库进行交互
* 需要制定JPA使用那个持久化的框架级配置改框架的基本属性

1. 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
2. 使用JPA API完成数据的CRUD操作

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

## JPA之Hello World

1. 创建JPA工程，版本选择2.0
2. 利用maven管理依赖；
3. <dependencies>  
    <dependency>  
    <groupId>org.hibernate</groupId>  
    <artifactId>hibernate-entitymanager</artifactId>  
    <version>4.2.4.Final</version>  
    </dependency>  
    <dependency>  
    <groupId>mysql</groupId>  
    <artifactId>mysql-connector-java</artifactId>  
    <version>5.1.47</version>  
    </dependency>  
    <dependency>  
    <groupId>org.projectlombok</groupId>  
    <artifactId>lombok</artifactId>  
    <version>1.18.12</version>  
    </dependency>  
   </dependencies>
4. 配置persistence.xml

* **配置使用什么ORM产品来作为JPA的实现：**
* 实际上配置的是

javax.persistence.spi.PersistenceProvider接口的实现类

* 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> |

* **配置连接信息：Connection**

|  |
| --- |
| <property name="hibernate.connection.url"  value="jdbc:mysql://localhost:3306/jpa?useSSL=false&amp;  serverTimezone=Asia/Shanghai"/>  <property name="hibernate.connection.driver\_class"  value="com.mysql.jdbc.Driver"/>  <property name="hibernate.connection.username" value="root"/>  <property name="hibernate.connection.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"/> |

1. 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，并在persistence.xml配置

|  |
| --- |
| <class>com.neusoft.customerUser</class> |

1. 创建主函数

|  |
| --- |
| //1. 创建EntitymanagerFactory  String persistenceUnitName = "jpa-1" //persistence中  EntityManagerFactory entityManagerFactory  = Persistence.createEntityManagerFactory(persistenceUnitName);  //2. 创建EntityManager  EntityManager entityManager =  entityManagerFactory.createEntityManager();  //3. 开启事物  EntityTransaction transaction = entityManager.getTransaction();  transaction.begin();  //4. 进行持久化操作  Customer customer = new Customer();  customer.setAge(12);  ...  entityManager.persist(customer);  //5. 提交事物  transaction.commit();  //6. 关闭EntityManager  entityManager.close();  //7. 关闭EntityManagerFactory  entityManagerFactory.close(); |

# JPA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
* **Example:**

|  |
| --- |
| **@Table(name = "PERSON\_TABLE")**  **@Entity**  **public class Person{}** |

## @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。

## @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；
* 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
* **Example:**

|  |
| --- |
| **@Temporal(TemporalType.TIMESTAMP)**  **@Column(name = "create\_time")**  **private Date createTime;** |

# 主键生成策略-table

* **应用场景**：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
* **使用场景**：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。
* Example:

|  |
| --- |
| **Table:**  CREATE TABLE `jpa\_id\_generator` (  `id` int(10) NOT NULL AUTO\_INCREMENT COMMENT '主键',  `pk\_name` varchar(50) COLLATE utf8\_bin NOT NULL COMMENT '主键名字，值一般为：表名\_id',  `pk\_value` int(10) NOT NULL COMMENT '主键的插入次数',  PRIMARY KEY (`id`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8\_bin;  **Java代码：**  @Id  @Column(name = "id")  @TableGenerator(name = "id\_generator",  table = "jpa\_id\_generator",  pkColumnName = "pk\_name",  pkColumnValue = "customer\_id",  valueColumnName = "pk\_value",  allocationSize = 1,  initialValue = 0)  @GeneratedValue(strategy = GenerationType.TABLE, generator = "id\_generator")  private Integer id; |

# 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：

1. 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
2. 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：

1. createEntityManager()：用于创建实体管理器对象实例。
2. 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。
3. 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。
4. 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

### 对象说明

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, 根椐主键查找 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 (entityClass, primaryKey)

* 方法注释：

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 (entityClass, primaryKey)

* 方法注释：

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)

* 方法注释：

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。

1. 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。
2. 如果对删除状态的 Entity 进行 persist() 操作，会转换为持久化状态。
3. 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)

* **方法注释：**

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)

* **方法注释：**
* merge (T entity)：merge() 用于处理 Entity 的同步。即数据库的插入和更新操作
* **示意图：**

不存在

存在

返回News持久化对象的引用

存在

把new1游离对象的属性拷贝到New持久化对象中,计划执行一条update语句

从数据库加载id为1的News持久化对象

数据库中是否存

在id为1的记录

不存在

em 缓存中是否存在OID为1的News

持久化对象

创建一个新的News对象,把new1游离对象的属性拷贝到新建的News对象中,持久化这个News对象,计划执行一条insert语句

游离对象

临时对象

news1的状态

### flush ()

* **方法注释：**

**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。即没提交事物之前，就执行update语句，然后一起提交事物。

### setFlushMode

* **方法注释：**

**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

* 1. FlushModeType.AUTO 为自动更新数据库实体，
  2. FlushModeType.COMMIT 为直到提交事务时才更新数据库记录。

### getFlushMode ()

* **方法注释：**

**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)

* **方法注释：**

**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。即强制查询数据库，不走Hibernate一级缓存。

### clear ()

* **方法注释：**

**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)

* **方法注释：**

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 ()

* **方法注释：**

isOpen ()：判断当前的实体管理器是否是打开状态。

### getTransaction ()

* **方法注释：**

**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 ()

* **方法注释：**

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。

### 自定义sql查询

#### createQuery (String qlString)

* **方法注释：**

**createQuery** (String qlString)：创建一个查询对象。

#### createNamedQuery (String name)

* **方法注释：**

**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)

* **方法注释：**

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 (sqls,resultSetMapping)

* **方法注释：**

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。

### 事物对象

* **EntityTransaction：**

注释：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。

* 1. **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。

* 1. **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。

* 1. **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。

* 1. **setRollbackOnly ()**

使当前事务只能被撤消。

* 1. **getRollbackOnly ()**

查看当前事务是否设置了只能撤消标志。

* 1. **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。

# 映射关联关系

## 单向多对一的关联关系

* **约定：**

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)

* **@ManyToOne**

|  |  |  |
| --- | --- | --- |
| **属性名** | **注释** | **例如** |
| targetEntity | 目标类的元数据 | targetEntity = Customer.class |
| fetch | 加载方式 | fetch = FetchType.LAZY |

* **@JoinColumn**

1. name：

**当前类的关联字段：如果没有指定Column,则值为实体属性名，否则是数据库字段名。**

* **jpa默认的关联字段，是在关联对象中存在的。**

**例如name="CUSTOMER\_ID",则在Order不用创建对应字段，而是默认Customer实例中Id字段就是CUSTOMER\_ID，但是在JPA\_ORDER表中则有CUSTOMER\_ID；如果在Order实体中创建customerId,则必须告诉jpa, insertable = false, updatable = false，否则报错。**

1. **referencedColumnName，如果不写，默认是主键**

**关联表中的关联字段：如果没有指定Column,则值为实体属性名，否则是数据库字段名。**

* **Example:**

1. **第一种情况：**

**Order实体：**

@Column(name = "customer\_id")

private Integer customerId;

**Customer实体：**

@Id

@Column(name = "id")

@GeneratedValue(strategy = GenerationType.AUTO)

private Integer id;

|  |
| --- |
| **@JoinColumn(name = "customer\_id", referencedColumnName = "id", insertable = false, updatable = false)**  **@ManyToOne(targetEntity = Customer.class)**  **private Customer customer;** |

1. **第二种情况：Order实体，不创建customerId属性**

|  |
| --- |
| **@JoinColumn(name = "customer\_id")**  **@ManyToOne(targetEntity = Customer.class)**  **private Customer customer;** |

## 单向一对多的关联关系

* **@OneToMany属性：**

1. targetEntity,指定目标类的元数据
2. fetch：指定加载方式，默认是懒加载，与多对一相反
3. mappedBy: 指定它是这一关联中的被维护端，many 为维护端，例如：mappedBy="customer"

* **@JoinColumn属性：**

1. name:指定many端的关联字段，最后在jpa\_order表中创建字段customer\_id

* **Example:**

|  |
| --- |
| @JoinColumn(name = "customer\_id")  @OneToMany(**targetEntity = Order.class**)  private Set<Order> orders; |

* **注意的地方：**

1. **新增：**单向1-n 关联关系执行保存时，一定会出现UPDATE语句，因为n的一端在插入时不会同时插入外键列；
2. **查询：**默认对关联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
3. **删除：**默认情况下，若删除1的一端，则会先把关联的n的一端外键置空，然后进行删除，可以通过@OneToMany的cascade（级联）属性来修改默认的删除策略。CascadeType.REMOVE
4. **修改：无特殊地方**

## 双向多对一的关联关系

* **解释：**

**双向多对一和双向一对多是一样的，并且，只是在1的端添加set<Order>,在n的端添加Customer，但是需要注意，2个类中定义的关联字段一定一致，例如，标红的地方**

* **Example**

|  |
| --- |
| **Customer类：**  **@JoinColumn(name = "customer\_id")**  **@OneToMany(targetEntity = Order.class)**  **private Set<Order> orders;**  **Order 类：**  **@JoinColumn(name = "customer\_id")**  **@ManyToOne(targetEntity = Customer.class)**  **private Customer customer;** |

* **注意的地方：**

1. **新增：**

* **若是双向1-n的关联关系，执行保存时，先保存n的一端，在保存1的一端，默认情况下，会多出（n的保存数量） \* 2 条UPDATE语句，因为双方都在维护关联关系；**
* **若先保存1的一端，则会出现（n的保存数量）条UPDATE语句**
* **因此在1-n关联时，建议使用n的一方维护关联关系，而1的一方不维护关联关系，这样会有效的减少sql语句，可以使用这个注解在规定：@OneToMany 中的mappedBy="customer"，注意这个属性，不能与@JoinColumn同时存在。**

## 双向一对一映射

* **数据库设计1对1的字段设计方式**

**例子：manager(经理)、department（部门）是1-1的关联关系**

1. **利用主键的方式：即可以将manager的主键作为department的主键**
2. **利用外键的方式：即可以在department中创建manager\_id字段。**

* **注解使用：**

**例子：Manager(经理)、Department（部门）, 1-1的关联关系**

1. **在需要添加外键的实体类中（Department）使用@JoinCloumn来映射，注意：1-1关联关系，所以要添加unique=true**
2. **在Manager使用@OneToOne注解，建议指定由谁维护关联关系。**

* **Example：**

|  |
| --- |
| **Department(部门)（外键）:**  @JoinColumn(name = "mgr\_id", unique = true)  @OneToOne(fetch = FetchType.LAZY)  private Manager manager;  **Manager(经理)**  @OneToOne(mappedBy = "manager")  private Department dept; |

* **注意的地方:**

1. **新增：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。**
2. **查询：**

* **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@OneToOne的fetch属性来修改加载策略**

**,这样查询到的关联对象，是个代理类**

* **默认情况下，若获取不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@OneToOne的fetch属性来修改加载策略，但是依然会再发送sql语句来初始化关联对象。因此不建议在不维护关联关系一方来修改fetch属性**
* **双向1-1不延迟加载的问题：**

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.

## 双向多对多关联关系

* **注解使用：**

**假定例子：item(产品)、category（类目）**

1. 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。

@ManyToMany

@JoinTable(name="中间表名称",

**joinColumns**={@JoinColumn(name="中间表外键列名",

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")}

)

* **Example:**

|  |
| --- |
| 关系维护端：Item  @JoinTable(name = "ITEM\_CATEGORY",  joinColumns={@JoinColumn(name="ITEM\_ID",  referencedColumnName="ID")},  inversejoinColumns={@JoinColumn(name="CATEGORY\_ID",  referencedColunName="ID")})  @ManyToMany  private Set<Category> categorys;  被维护端：Category  @ManyToMany(mappedBy = "categorys")  private Set<Item> items; |

* **注意的地方:**

1. **双向映射时，出现循环引用：java.lang.stackoverflowerror**

**原因：使用lombok中的@Data,会生成全属性的hashCode、toString方法，而关联属性，是双方互相引用的，所以造成循环引用，堆栈溢出。**

**解决：直接指定@Getter、@Setter方法，不用@Data**

1. **保存：**

|  |
| --- |
| /\*\*  \* 双向多对多保存  \*/  @Test  public void testManyToMany() {  Category category1 = new Category("类目1");  Category category2 = new Category("类目2");  Product product1 = new Product("产品1");  Product product2 = new Product("产品2");  //设置产品的类目  product1.getCategories().add(category1);  product1.getCategories().add(category2);  product2.getCategories().add(category1);  product2.getCategories().add(category2);  //设置类目产品  category1.getProducts().add(product1);  category1.getProducts().add(product2);  category2.getProducts().add(product1);  category2.getProducts().add(product2);  //保存  entityManager.persist(product1);  entityManager.persist(product2);  entityManager.persist(category1);  entityManager.persist(category2);  } |

1. **查询：**
   * 1. **对应关联的集合对象，默认使用懒加载策略，无论使用那一端**

# 二级缓存

* **概念释义：**

1. **JPA一级缓存：是在同一entityManager作用域内，如果查询相同的sql,则会触发一级缓存；**
2. **JPA二级缓存：会跨entityManager，进行缓存查询。**

* **使用步骤：**

1. 添加ehcache坐标依赖；

|  |
| --- |
| **<dependency>**  **<groupId>org.hibernate</groupId>**  **<artifactId>hibernate-ehcache</artifactId>**  **<version>4.2.0.Final</version>**  **</dependency>** |

1. 添加ehcache配置文件；

****

1. 在persistence.xml的<properties>配置开启二级缓存

****

1. 在persistence.xml中配置<shared-cache-mode>节点，参数如下：

* 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

1. 在实体类上添加@Cacheable注解

# JPQL

## 默认情况下查询，返回List<Object[]>

|  |
| --- |
| String jpql = "from Customer c where c.age > ?";  Query query = entityManager.createQuery(jpql);  query.setParameter(1, 0);  List resultList = query.getResultList(); |

## 指定映射对象,适用于,查询部分字段

|  |
| --- |
| @Test  public void testJPQL2() {  String jpql = "select new Customer(c.age, c.lastName) from Customer c where c.age > ?";  Query query = entityManager.createQuery(jpql);  query.setParameter(1, 0);  List resultList = query.getResultList();  System.out.println(resultList);  } |

## 本地sql

|  |
| --- |
| @Test  public void testNativeSql() {  String sql = "select \* from jpa\_customer where id = ?";  Query query = entityManager.createNativeQuery(sql);  Object customer =  query.setParameter(1, 1).getSingleResult();  System.out.println(customer);  } |

## JPQL查询，使用缓存（NativeQuery不好用）

* **前提：**

1. **使用JPQL查询，本地sql不支持；**
2. persistence.xml**配置了开启查询缓存:**

<property name="hibernate.cache.use\_query\_cache" value="true"/>

|  |
| --- |
| @Test  public void testCatchable() {  String jpql = "from Customer c where c.age > ?";  Query query = entityManager.createQuery(jpql)  .setHint(QueryHints.HINT\_CACHEABLE, true);  Object customer =  query.setParameter(1, 1).getSingleResult();  System.out.println(customer);  customer = query.setParameter(1, 1).getSingleResult();  System.out.println(customer);  } |

## JPQL,使用ORDER BY

|  |
| --- |
| @Test  public void testOrderBy() {  String jpql = "select c from Customer c where c.age > ? order by c.age";  Query query = entityManager.createQuery(jpql).setHint(QueryHints.HINT\_CACHEABLE, true);  Object customer = query.setParameter(1, 1).getSingleResult();  System.out.println(customer);  } |

## JPQL 使用 GROUP BY HAVING

**HAVING语句的存在弥补了WHERE关键字不能与聚合函数联合使用的不足**

|  |
| --- |
| **/\*\***  **\* JPQL 使用 GROUP BY HAVING**  **\* 查询订单数量 >=2 的客户信息**  **\*/**  **@Test**  **public void testGroupByAndHaving() {**  **String jpql = "SELECT o.customer FROM Order o GROUP BY o.customer HAVING count(o.id) >= 2";**  **List list = entityManager.createQuery(jpql).getResultList();**  **System.out.println(list);**  **}** |

## JPQL 使用 LEFT OUTER JOIN FETCH

* 请谨慎使用，因为默认是懒加载，这样就不会懒加载了；
* 正常情况不使用LEFT OUTER JOIN FETCH，会发送2条查询语句

|  |
| --- |
| @Test  public void testLeftJoinFetch() {  String jpql = "SELECT c FROM Customer c LEFT OUTER JOIN FETCH c.orders WHERE c.id = ?";  Customer customer = (Customer) entityManager.createQuery(jpql)  .setParameter(1, 1)  .getSingleResult();  System.out.println(customer.getLastName());  //正常情况不使用LEFT OUTER JOIN FETCH，会发送2条查询语句  System.out.println(customer.getOrders().size());  } |

## JPQL使用 子查询

|  |
| --- |
| /\*\*  \* JPQL 使用子查询  \* 查询lastName为 AA 客户的订单  \*/  @Test  public void testSubQuery() {  String jpql = "SELECT o FROM Order o " +  "WHERE o.customer = (SELECT c FROM Customer c WHERE c.lastName = ?)";  Query query = entityManager.createQuery(jpql);  query.setParameter(1, "AA");  List list = query.getResultList();  System.out.println(list);  } |

## JPQL 使用函数

* **字符串处理函数主要有：**

1. concat(String s1, String s2)：字符串合并/连接函数。
2. substring(String s, int start, int length)：取字串函数。
3. 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。
4. lower(String s)：将字符串转换成小写形式。
5. upper(String s)：将字符串转换成大写形式。
6. length(String s)：求字符串的长度。
7. 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。**

* **Example**

|  |
| --- |
| **/\*\***  **\* JPQL 使用 内置函数**  **\*/**  **@Test**  **public void testJpqlFunction() {**  **String jpql = "SELECT UPPER(c.email) FROM Customer c";**  **Query query = entityManager.createQuery(jpql);**  **List emails = query.getResultList();**  **System.out.println(emails);**  **}** |

## JPQL 使用 update、delete

|  |
| --- |
| /\*\*  \* JPQL 使用 update、delete  \*/  @Test  public void testUpdate() {  String jpql = "UPDATE Customer c set c.lastName = ? WHERE c.id = ?";  Query query = entityManager.createQuery(jpql);  query.setParameter(1, "AAAA").setParameter(2, 1);  query.executeUpdate();  } |