

# JPA 第一天

## 第1章 1.ORM 概述[了解]

ORM (Object-Relational Mapping) 表示对象关系映射。在面向对象的软件开发中,通过 ORM,就可以把对象映射到关系型数据库中。只要有一套程序能够做到建立对象与数据库的关联,操作对象就可以直接操作数据库数据,就可以说这套程序实现了 ORM 对象关系映射

简单的说: ORM 就是建立实体类和数据库表之间的关系,从而达到操作实体类就相当于操作数据库表的目的。

### 1.1 为什么使用 ORM

当实现一个应用程序时(不使用 O/RMapping),我们可能会写特别多数据访问层的代码,从数据库保存数据、修改数据、删除数据,而这些代码都是重复的。而使用 ORM 则会大大减少重复性代码。对象关系映射(Object Relational Mapping, 简称 ORM), 主要实现程序对象到关系数据库数据的映射。

### 1.2 常见 ORM 框架

常见的 orm 框架: Mybatis (ibatis)、Hibernate、Jpa

## 第2章 hibernate 与 JPA 的概述[了解]

### 2.1 hibernate 概述

Hibernate 是一个开放源代码的对象关系映射框架，它对 JDBC 进行了非常轻量级的对象封装，它将 POJO 与数据库表建立映射关系，是一个全自动的 orm 框架，hibernate 可以自动生成 SQL 语句，自动执行，使得 Java 程序员可以随心所欲的使用对象编程思维来操纵数据库。

## 2.2 JPA 概述

JPA 的全称是 Java Persistence API，即 Java 持久化 API，是 SUN 公司推出的一套基于 ORM 的规范，内部是由一系列的接口和抽象类构成。

JPA 通过 JDK 5.0 注解描述对象—关系表的映射关系，并将运行期的实体对象持久化到数据库中。

## 2.3 JPA 的优势

### 1. 标准化

JPA 是 JCP 组织发布的 Java EE 标准之一，因此任何声称符合 JPA 标准的框架都遵循同样的架构，提供相同的访问 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。

### 2. 容器级特性的支持

JPA 框架中支持大数据集、事务、并发等容器级事务，这使得 JPA 超越了简单持久化框架的局限，在企业应用发挥更大的作用。

### 3. 简单方便

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，没有任何的约束和限制，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，没有太多特别的规则和设计模式的要求，开发者可以很容易的掌握。JPA 基于非侵入式原则设计，因此可以很容易的和其它框架或者容器集成

### 4. 查询能力

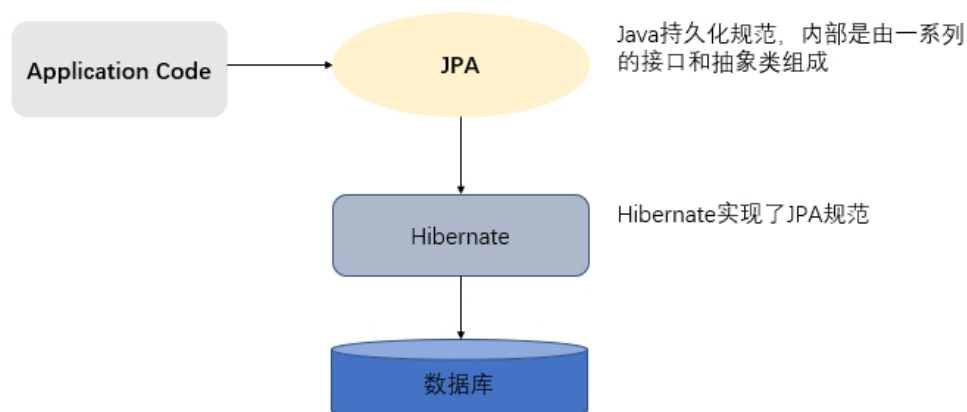
JPA 的查询语言是面向对象而非面向数据库的，它以面向对象的自然语法构造查询语句，可以看成是 Hibernate HQL 的等价物。JPA 定义了独特的 JPQL (Java Persistence Query Language)，JPQL 是 EJB QL 的一种扩展，它是针对实体的一种查询语言，操作对象是实体，而不是关系数据库的表，而且能够支持批量更新和修改、JOIN、GROUP BY、HAVING 等通常只有 SQL 才能够提供的高级查询特性，甚至还能够支持子查询。

### 5. 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，这样的支持能够让开发者最大限度的使用面向对象的模型设计企业应用，而不需要自行处理这些特性在关系数据库的持久化。

## 2.4 JPA 与 hibernate 的关系

JPA 规范本质上就是一种 ORM 规范，注意不是 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，但具体实现则由服务厂商来提供实现。



JPA 和 Hibernate 的关系就像 JDBC 和 JDBC 驱动的关系，JPA 是规范，Hibernate 除了作为 ORM 框架之外，它也是一种 JPA 实现。JPA 怎么取代 Hibernate 呢？JDBC 规范可以驱动底层数据库吗？答案是否定的，也就是说，如果使用 JPA 规范进行数据库操作，底层需要 hibernate 作为其实现类完成数据持久化工作。

## 第3章 JPA 的入门案例

### 3.1 需求介绍

本章节我们是实现的功能是保存一个客户到数据库的客户表中。

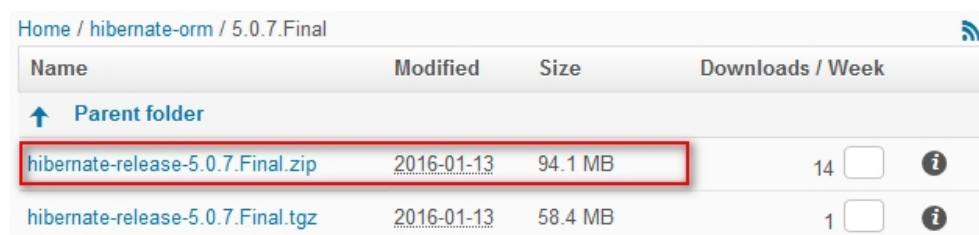
## 3.2 开发包介绍

由于 JPA 是 sun 公司制定的 API 规范，所以我们不需要导入额外的 JPA 相关的 jar 包，只需要导入 JPA 的提供商的 jar 包。我们选择 Hibernate 作为 JPA 的提供商，所以需要导入 Hibernate 的相关 jar 包。

下载网址：

<http://sourceforge.net/projects/hibernate/files/hibernate-orm/5.0.7.Final/>

页面显示如下图：



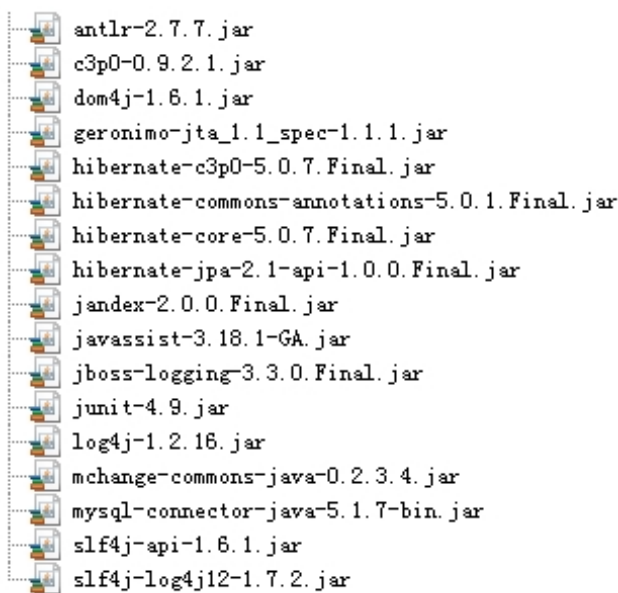
Home / hibernate-orm / 5.0.7.Final			
Name	Modified	Size	Downloads / Week
↑ Parent folder			
<a href="#">hibernate-release-5.0.7.Final.zip</a>	2016-01-13	94.1 MB	14 <input type="checkbox"/>
<a href="#">hibernate-release-5.0.7.Final.tgz</a>	2016-01-13	58.4 MB	1 <input type="checkbox"/>

## 3.3 搭建开发环境[重点]

### 3.3.1 导入 jar 包

对于 JPA 操作，只需要从 hibernate 提供的资料中找到我们需要的 jar 导入到工程中即可。

- 传统工程导入 jar 包



- maven 工程导入坐标

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.hibernate.version>5.0.7.Final</project.hibernate.version>
</properties>

<dependencies>
    <!-- junit -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>

    <!-- hibernate对jpa的支持包 -->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>${project.hibernate.version}</version>
    </dependency>

    <!-- c3p0 -->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-c3p0</artifactId>
        <version>${project.hibernate.version}</version>
```

```
</dependency>

<!-- log日志 -->
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>

<!-- Mysql and MariaDB -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.6</version>
</dependency>
</dependencies>
```

### 3.3.2 创建客户的数据库表和客户的实体类

- 创建客户的数据库表

```
/*创建客户表*/
CREATE TABLE cst_customer (
    cust_id bigint(32) NOT NULL AUTO_INCREMENT COMMENT '客户编号(主键)',
    cust_name varchar(32) NOT NULL COMMENT '客户名称(公司名称)',
    cust_source varchar(32) DEFAULT NULL COMMENT '客户信息来源',
    cust_industry varchar(32) DEFAULT NULL COMMENT '客户所属行业',
    cust_level varchar(32) DEFAULT NULL COMMENT '客户级别',
    cust_address varchar(128) DEFAULT NULL COMMENT '客户联系地址',
    cust_phone varchar(64) DEFAULT NULL COMMENT '客户联系电话',
    PRIMARY KEY (`cust_id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

- 创建客户的实体类

```
public class Customer implements Serializable {

    private Long custId;
    private String custName;
    private String custSource;
```

```
private String custIndustry;
private String custLevel;
private String custAddress;
private String custPhone;

public Long getCustId() {
    return custId;
}
public void setCustId(Long custId) {
    this.custId = custId;
}
public String getCustName() {
    return custName;
}
public void setCustName(String custName) {
    this.custName = custName;
}
public String getCustSource() {
    return custSource;
}
public void setCustSource(String custSource) {
    this.custSource = custSource;
}
public String getCustIndustry() {
    return custIndustry;
}
public void setCustIndustry(String custIndustry) {
    this.custIndustry = custIndustry;
}
public String getCustLevel() {
    return custLevel;
}
public void setCustLevel(String custLevel) {
    this.custLevel = custLevel;
}
public String getCustAddress() {
    return custAddress;
}
public void setCustAddress(String custAddress) {
    this.custAddress = custAddress;
}
public String getCustPhone() {
    return custPhone;
}
```

```
public void setCustPhone(String custPhone) {  
    this.custPhone = custPhone;  
}  
}
```

### 3.3.3 编写实体类和数据库表的映射配置[重点]

- 在实体类上使用 JPA 注解的形式配置映射关系

```
/**  
 *      * 所有的注解都是使用 JPA 的规范提供的注解，  
 *      * 所以在导入注解包的时候，一定要导入 javax.persistence 下的  
 */  
  
@Entity //声明实体类  
@Table(name="cst_customer") //建立实体类和表的映射关系  
public class Customer {  
  
    @Id//声明当前私有属性为主键  
    @GeneratedValue(strategy=GenerationType.IDENTITY) //配置主键的生成策略  
    @Column(name="cust_id") //指定和表中 cust_id 字段的映射关系  
    private Long custId;  
  
    @Column(name="cust_name") //指定和表中 cust_name 字段的映射关系  
    private String custName;  
  
    @Column(name="cust_source")//指定和表中 cust_source 字段的映射关系  
    private String custSource;  
  
    @Column(name="cust_industry")//指定和表中 cust_industry 字段的映射关系  
    private String custIndustry;  
  
    @Column(name="cust_level")//指定和表中 cust_level 字段的映射关系  
    private String custLevel;  
  
    @Column(name="cust_address")//指定和表中 cust_address 字段的映射关系  
    private String custAddress;  
  
    @Column(name="cust_phone")//指定和表中 cust_phone 字段的映射关系  
    private String custPhone;
```



```
public Long getCustId() {
    return custId;
}

public void setCustId(Long custId) {
    this.custId = custId;
}

public String getCustName() {
    return custName;
}

public void setCustName(String custName) {
    this.custName = custName;
}

public String getCustSource() {
    return custSource;
}

public void setCustSource(String custSource) {
    this.custSource = custSource;
}

public String getCustIndustry() {
    return custIndustry;
}

public void setCustIndustry(String custIndustry) {
    this.custIndustry = custIndustry;
}

public String getCustLevel() {
    return custLevel;
}

public void setCustLevel(String custLevel) {
    this.custLevel = custLevel;
}

public String getCustAddress() {
    return custAddress;
}

public void setCustAddress(String custAddress) {
    this.custAddress = custAddress;
}

public String getCustPhone() {
    return custPhone;
}

public void setCustPhone(String custPhone) {
    this.custPhone = custPhone;
}
}
```

- 常用注解的说明

```
@Entity
    作用：指定当前类是实体类。
@Table
    作用：指定实体类和表之间的对应关系。
    属性：
        name：指定数据库表的名称
@Id
    作用：指定当前字段是主键。
@GeneratedValue
    作用：指定主键的生成方式。。
    属性：
        strategy：指定主键生成策略。
@Column
    作用：指定实体类属性和数据库表之间的对应关系
    属性：
        name：指定数据库表的列名称。
        unique：是否唯一
        nullable：是否可以空
        inserttable：是否可以插入
        updateable：是否可以更新
        columnDefinition：定义建表时创建此列的 DDL
        secondaryTable：从表名。如果此列不建在主表上（默认建在主表），该属性
        定义该列所在从表的名字搭建开发环境[重点]
```

### 3.3.4 配置 JPA 的核心配置文件

在 java 工程的 src 路径下创建一个名为 META-INF 的文件夹，在此文件夹下创建一个名为 persistence.xml 的配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
    version="2.0">
```

```
<!--配置持久化单元
    name: 持久化单元名称
    transaction-type: 事务类型
        RESOURCE_LOCAL: 本地事务管理
        JTA: 分布式事务管理 -->
<persistence-unit name="myJpa" transaction-type="RESOURCE_LOCAL">
    <!--配置 JPA 规范的服务提供商 -->
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <properties>
        <!-- 数据库驱动 -->
        <property name="javax.persistence.jdbc.driver"
value="com.mysql.jdbc.Driver" />
        <!-- 数据库地址 -->
        <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/ssh" />
        <!-- 数据库用户名 -->
        <property name="javax.persistence.jdbc.user" value="root" />
        <!-- 数据库密码 -->
        <property name="javax.persistence.jdbc.password" value="111111" />

        <!--jpa 提供者的可选配置: 我们的 JPA 规范的提供者是 hibernate, 所以 jpa 的核心配置中兼容 hibernate 的配 -->
        <property name="hibernate.show_sql" value="true" />
        <property name="hibernate.format_sql" value="true" />
        <property name="hibernate.hbm2ddl.auto" value="create" />
    </properties>
</persistence-unit>
</persistence>
```

### 3.4 实现保存操作

```
@Test
public void test() {
    /**
     * 创建实体管理类工厂, 借助 Persistence 的静态方法获取
     * 其中传递的参数为持久化单元名称, 需要 jpa 配置文件中指定
     */
    EntityManagerFactory factory =
Persistence.createEntityManagerFactory("myJpa");
    //创建实体管理类
    EntityManager em = factory.createEntityManager();
    //获取事务对象
```

```
EntityTransaction tx = em.getTransaction();
//开启事务
tx.begin();
Customer c = new Customer();
c.setCustName("传智播客");
//保存操作
em.persist(c);
//提交事务
tx.commit();
//释放资源
em.close();
factory.close();
}
```

## 第4章 JPA 中的主键生成策略

通过 annotation（注解）来映射 hibernate 实体的, 基于 annotation 的 hibernate 主键标识为@Id, 其生成规则由@GeneratedValue 设定的. 这里的@Id 和@GeneratedValue 都是 JPA 的标准用法。

JPA 提供的四种标准用法为 TABLE, SEQUENCE, IDENTITY, AUTO。

具体说明如下：

**IDENTITY**: 主键由数据库自动生成（主要是自动增长型）

用法:

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long custId;
```

**SEQUENCE**: 根据底层数据库的序列来生成主键，条件是数据库支持序列。

用法:

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator="payablemoney_seq")
@SequenceGenerator(name="payablemoney_seq", sequenceName="seq_payment")
private Long custId;
```

```
//@SequenceGenerator 源码中的定义
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface SequenceGenerator {
    //表示该表主键生成策略的名称，它被引用在@GeneratedValue 中设置的“generator”值中
    String name();
    //属性表示生成策略用到的数据库序列名称。
    String sequenceName() default "";
    //表示主键初识值，默认为 0
    int initialValue() default 0;
    //表示每次主键值增加的大小，例如设置 1，则表示每次插入新记录后自动加 1，默认为 50
    int allocationSize() default 50;
}
```

## AUTO: 主键由程序控制

用法:

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long custId;
```

## TABLE: 使用一个特定的数据库表格来保存主键

用法:

```
@Id
@GeneratedValue(strategy = GenerationType.TABLE, generator="payablemoney_gen")
@TableGenerator(name = "pk_gen",
    table="tb_generator",
    pkColumnName="gen_name",
    valueColumnName="gen_value",
    pkColumnValue="PAYABLEMOENY_PK",
    allocationSize=1
)
private Long custId;

//@TableGenerator 的定义:
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface TableGenerator {
    //表示该表主键生成策略的名称，它被引用在@GeneratedValue 中设置的“generator”值中
```

```
String name();  
//表示表生成策略所持久化的表名，例如，这里表使用的是数据库中的“tb_generator”。  
String table() default "";  
//catalog 和 schema 具体指定表所在的目录名或是数据库名  
String catalog() default "";  
String schema() default "";  
// 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。例如在“tb_generator”中将  
“gen_name”作为主键的键值  
String pkColumnName() default "";  
// 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。例如，在  
“tb_generator”中将“gen_value”作为主键的值  
String valueColumnName() default "";  
//属性的值表示在持久化表中，该生成策略所对应的主键。例如在“tb_generator”表中，将“gen_name”  
的值为“CUSTOMER_PK”。  
String pkColumnValue() default "";  
//表示主键初识值，默认为 0。  
int initialValue() default 0;  
//表示每次主键值增加的大小，例如设置成 1，则表示每次创建新记录后自动加 1，默认为 50。  
int allocationSize() default 50;  
UniqueConstraint[] uniqueConstraints() default {};  
}  
  
//这里应用表 tb_generator, 定义为：  
CREATE TABLE tb_generator (  
    id NUMBER NOT NULL,  
    gen_name VARCHAR2(255) NOT NULL,  
    gen_value NUMBER NOT NULL,  
    PRIMARY KEY(id)  
)
```

## 第5章 JPA 的 API 介绍

### 5.1 Persistence 对象

Persistence 对象主要作用是用于获取 EntityManagerFactory 对象的。通过调用该类的 createEntityManagerFactory 静态方法，根据配置文件中持久化单元名称创建 EntityManagerFactory。

```
//1. 创建 EntityManagerFactory
@Test
String unitName = "myJpa";
EntityManagerFactory factory= Persistence.createEntityManagerFactory(unitName);
```

## 5.2 EntityManagerFactory

EntityManagerFactory 接口主要用来创建 EntityManager 实例

```
//创建实体管理类
EntityManager em = factory.createEntityManager();
```

由于 EntityManagerFactory 是一个线程安全的对象（即多个线程访问同一个 EntityManagerFactory 对象不会有线程安全问题），并且 EntityManagerFactory 的创建极其浪费资源，所以在使用 JPA 编程时，我们可以对 EntityManagerFactory 的创建进行优化，只需要做到一个工程只存在一个 EntityManagerFactory 即可

## 5.3 EntityManager

在 JPA 规范中，EntityManager 是完成持久化操作的核心对象。实体类作为普通 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。

我们可以通过调用 EntityManager 的方法完成获取事务，以及持久化数据库的操作

方法说明：

```
getTransaction : 获取事务对象
persist : 保存操作
merge : 更新操作
remove : 删除操作
find/getReference : 根据 id 查询
```

## 5.4 EntityTransaction

在 JPA 规范中，EntityTransaction 是完成事务操作的核心对象，对于 EntityTransaction 在我们的 java 代码中承接的功能比较简单

begin: 开启事务  
commit: 提交事务  
rollback: 回滚事务

## 第6章 抽取 JPAUtil 工具类

```
package cn.itcast.dao;

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

public final class JPAUtil {
    // JPA的实体管理器工厂：相当于Hibernate的SessionFactory
    private static EntityManagerFactory em;
    // 使用静态代码块赋值
    static {
        // 注意：该方法参数必须和persistence.xml中persistence-unit标签name属性取值一致
        em = Persistence.createEntityManagerFactory("myPersistUnit");
    }

    /**
     * 使用管理器工厂生产一个管理器对象
     *
     * @return
     */
    public static EntityManager getEntityManager() {
        return em.createEntityManager();
    }
}
```



## 第7章 使用 JPA 完成增删改查操作

### 7.1 保存

```
/**
 * 保存一个实体
 */
@Test
public void testAdd() {
    // 定义对象
    Customer c = new Customer();
    c.setCustName("传智学院");
    c.setCustLevel("VIP客户");
    c.setCustSource("网络");
    c.setCustIndustry("IT教育");
    c.setCustAddress("昌平区北七家镇");
    c.setCustPhone("010-84389340");
    EntityManager em = null;
    EntityTransaction tx = null;
    try {
        // 获取实体管理对象
        em = JPAUtil.getEntityManager();
        // 获取事务对象
        tx = em.getTransaction();
        // 开启事务
        tx.begin();
        // 执行操作
        em.persist(c);
        // 提交事务
        tx.commit();
    } catch (Exception e) {
        // 回滚事务
        tx.rollback();
        e.printStackTrace();
    } finally {
        // 释放资源
        em.close();
    }
}
```

## 7.2 修改

```
@Test
public void testMerge() {
    //定义对象
    EntityManager em=null;
    EntityTransaction tx=null;
    try{
        //获取实体管理对象
        em=JPAUtil.getEntityManager();
        //获取事务对象
        tx=em.getTransaction();
        //开启事务
        tx.begin();
        //执行操作
        Customer c1 = em.find(Customer.class, 6L);
        c1.setCustName("江苏传智学院");
        em.clear();//把c1对象从缓存中清除出去
        em.merge(c1);
        //提交事务
        tx.commit();
    }catch(Exception e){
        //回滚事务
        tx.rollback();
        e.printStackTrace();
    }finally{
        //释放资源
        em.close();
    }
}
```

## 7.3 删除

```
/**
 * 删除
 */
@Test
public void testRemove() {
```

```
// 定义对象
EntityManager em = null;
EntityTransaction tx = null;
try {
    // 获取实体管理对象
    em = JPAUtil.getEntityManager();
    // 获取事务对象
    tx = em.getTransaction();
    // 开启事务
    tx.begin();
    // 执行操作
    Customer c1 = em.find(Customer.class, 6L);
    em.remove(c1);
    // 提交事务
    tx.commit();
} catch (Exception e) {
    // 回滚事务
    tx.rollback();
    e.printStackTrace();
} finally {
    // 释放资源
    em.close();
}
}
```

## 7.4 根据 id 查询

```
/**
 * 查询一个： 使用立即加载的策略
 */
@Test
public void testGetOne() {
    // 定义对象
    EntityManager em = null;
    EntityTransaction tx = null;
    try {
        // 获取实体管理对象
        em = JPAUtil.getEntityManager();
        // 获取事务对象
        tx = em.getTransaction();
        // 开启事务
        tx.begin();
        // 执行操作
```

```
        Customer c1 = em.find(Customer.class, 1L);
        // 提交事务
        tx.commit();

        System.out.println(c1); // 输出查询对象
    } catch (Exception e) {
        // 回滚事务
        tx.rollback();
        e.printStackTrace();
    } finally {
        // 释放资源
        em.close();
    }
}

// 查询实体的缓存问题
@Test
public void testGetOne() {
    // 定义对象
    EntityManager em = null;
    EntityTransaction tx = null;
    try {
        // 获取实体管理对象
        em = JPAUtil.getEntityManager();
        // 获取事务对象
        tx = em.getTransaction();
        // 开启事务
        tx.begin();
        // 执行操作
        Customer c1 = em.find(Customer.class, 1L);
        Customer c2 = em.find(Customer.class, 1L);
        System.out.println(c1 == c2); // 输出结果是true, EntityManager也有缓存
        // 提交事务
        tx.commit();
        System.out.println(c1);
    } catch (Exception e) {
        // 回滚事务
        tx.rollback();
        e.printStackTrace();
    } finally {
        // 释放资源
        em.close();
    }
}
```

```
// 延迟加载策略的方法:
/**
 * 查询一个: 使用延迟加载策略
 */
@Test
public void testLoadOne() {
    // 定义对象
    EntityManager em = null;
    EntityTransaction tx = null;
    try {
        // 获取实体管理对象
        em = JPAUtil.getEntityManager();
        // 获取事务对象
        tx = em.getTransaction();
        // 开启事务
        tx.begin();
        // 执行操作
        Customer cl = em.getReference(Customer.class, 1L);
        // 提交事务
        tx.commit();
        System.out.println(cl);
    } catch (Exception e) {
        // 回滚事务
        tx.rollback();
        e.printStackTrace();
    } finally {
        // 释放资源
        em.close();
    }
}
```

## 7.5 JPA 中的复杂查询

### JPQL 全称 Java Persistence Query Language

基于首次在 EJB2.0 中引入的 EJB 查询语言 (EJB QL), Java 持久化查询语言 (JPQL) 是一种可移植的查询语言, 旨在以面向对象表达式语言的表达式, 将 SQL 语法和简单查询语义绑定在一起。使用这种语言编写的查询是可移植的, 可以被编译成所有主流数据库服务器上的 SQL。

其特征与原生 SQL 语句类似, 并且完全面向对象, 通过类名和属性访问, 而不是表名和表的属性。

## 7.5.1 查询全部

```
//查询所有客户
@Test
public void findAll() {
    EntityManager em = null;
    EntityTransaction tx = null;
    try {
        //获取实体管理对象
        em = JPAUtil.getEntityManager();
        //获取事务对象
        tx = em.getTransaction();
        tx.begin();
        // 创建query对象
        String jpql = "from Customer";
        Query query = em.createQuery(jpql);
        // 查询并得到返回结果
        List list = query.getResultList(); // 得到集合返回类型
        for (Object object : list) {
            System.out.println(object);
        }
        tx.commit();
    } catch (Exception e) {
        // 回滚事务
        tx.rollback();
        e.printStackTrace();
    } finally {
        // 释放资源
        em.close();
    }
}
```

## 7.5.2 分页查询

```
//分页查询客户
@Test
public void findPaged () {
    EntityManager em = null;
    EntityTransaction tx = null;
    try {
        //获取实体管理对象
```

```
        em = JPAUtil.getEntityManager();
        //获取事务对象
        tx = em.getTransaction();
        tx.begin();

        //创建query对象
        String jpql = "from Customer";
        Query query = em.createQuery(jpql);
        //起始索引
        query.setFirstResult(0);
        //每页显示条数
        query.setMaxResults(2);
        //查询并得到返回结果
        List list = query.getResultList(); //得到集合返回类型
        for (Object object : list) {
            System.out.println(object);
        }
        tx.commit();
    } catch (Exception e) {
        // 回滚事务
        tx.rollback();
        e.printStackTrace();
    } finally {
        // 释放资源
        em.close();
    }
}
```

### 7.5.3 条件查询

```
//条件查询
@Test
public void findCondition () {
    EntityManager em = null;
    EntityTransaction tx = null;
    try {
        //获取实体管理对象
        em = JPAUtil.getEntityManager();
        //获取事务对象
        tx = em.getTransaction();
        tx.begin();
```

```
//创建query对象
String jpql = "from Customer where custName like ? ";
Query query = em.createQuery(jpql);
//对占位符赋值, 从1开始
query.setParameter(1, "传智播客%");
//查询并得到返回结果
Object object = query.getSingleResult(); //得到唯一的结果集对象
System.out.println(object);
tx.commit();
} catch (Exception e) {
    // 回滚事务
    tx.rollback();
    e.printStackTrace();
} finally {
    // 释放资源
    em.close();
}
}
```

## 7.5.4 排序查询

```
//根据客户id倒序查询所有客户
//查询所有客户
@Test
public void testOrder() {
    EntityManager em = null;
    EntityTransaction tx = null;
    try {
        //获取实体管理对象
        em = JPAUtil.getEntityManager();
        //获取事务对象
        tx = em.getTransaction();
        tx.begin();
        // 创建query对象
        String jpql = "from Customer order by custId desc";
        Query query = em.createQuery(jpql);
        // 查询并得到返回结果
        List list = query.getResultList(); // 得到集合返回类型
        for (Object object : list) {
            System.out.println(object);
        }
        tx.commit();
    } catch (Exception e) {
```



```
        // 回滚事务
        tx.rollback();
        e.printStackTrace();
    } finally {
        // 释放资源
        em.close();
    }
}
```

## 7.5.5 统计查询

```
//统计查询
@Test
public void findCount() {
    EntityManager em = null;
    EntityTransaction tx = null;
    try {
        //获取实体管理对象
        em = JPAUtil.getEntityManager();
        //获取事务对象
        tx = em.getTransaction();
        tx.begin();
        // 查询全部客户
        // 1.创建query对象
        String jpql = "select count(custId) from Customer";
        Query query = em.createQuery(jpql);
        // 2.查询并得到返回结果
        Object count = query.getSingleResult(); // 得到集合返回类型
        System.out.println(count);
        tx.commit();
    } catch (Exception e) {
        // 回滚事务
        tx.rollback();
        e.printStackTrace();
    } finally {
        // 释放资源
        em.close();
    }
}
```