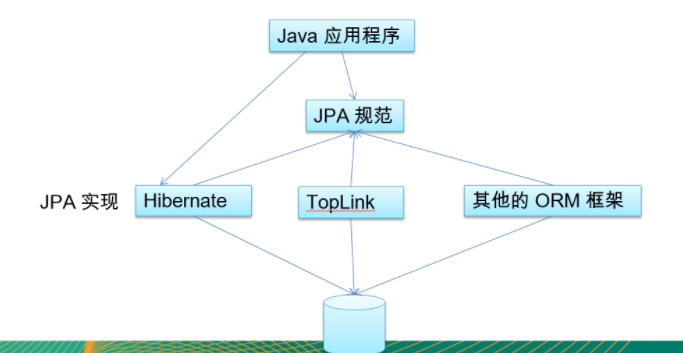
# 概念

JPA：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。



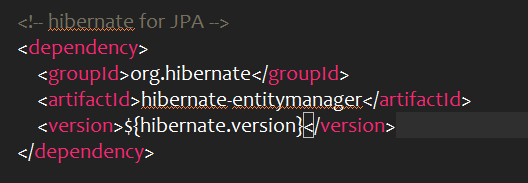
与JDBC类似，JPA统一了Java应用程序访问ORM框架的规范。以前我们的应用程序直接使用ORM框架，如hibernate，mybatis，但是不同的框架使用方法不一样，而JPA让我们以同样的方式访问不同的ORM框架。常用的框架是hibernate。

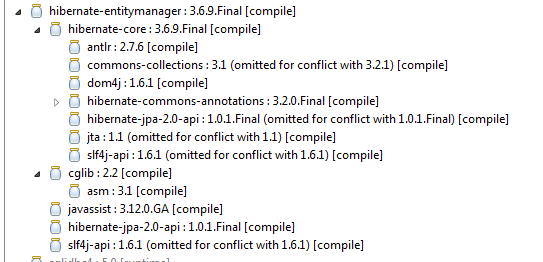
# JPA与hibernate的关系

JPA是一个规范，不是ORM框架，hibernate是JPA的实现。现在JPA具体实现框架有：Hibernate、OpenJPA、TopLink。

# hibernate JPA使用的jar包

使用hibernate-entitymanager.jar包





# JPA的优势

## （1）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。

## （2）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（@Entity） 进行注释，JPA 的框架和接口也都非常简单。

## （3）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语句，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。

## （4）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包括的技术

## （1）ORM映射元数据

JPA支持XML和JDK5.0注解两种元数据的形式，元数据描述了对象和表之间的映射关系，框架将实体对象持久化到数据库表中。

## （2）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。

## （3）查询语言（JPQL Java 持久性查询语言）

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

# JPA搭建环境的思路（重点）

## （1）创建 persistence.xml配置文件

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。

示例：



## （2）创建实体类

使用注解（annotation）来描述实体类和数据库表之间的映射关系。

## （3）编写Dao层

使用JPA API来完成数据的增、删、改、查。

说明：

EntityManagerFactory (对应 Hibernate 中的 SessionFactory);

EntityManager (对应 Hibernate 中的Session);

# 使用JPA进行数据操作的基本步骤（重点）

## 基本方法

### 导入相关的jar包

导入hibernate-jpa-2.0-api-1.0.1.Final.jar这个jar包（说明：导入hibernate-entitymanager这个依赖就包含这个jar包。）

### 配置数据源

通过jetty-env.xml文件配置数据源

### 编写persistence.xml文件

建立持久性单元

### 建立实体类

### 建立实体管理器（EntityManager）

### 操作实体

### 关闭实体管理器

## 持久化实体（创建实体）

将内存中的实体对象写入到数据表中，在表中反应的是新增了一行记录，对应SQL的insert语句，持久化实体可通过persist()方法。例如：

em.persist(customer);//em为实体管理器对象,customer为客户实体对象

### 示例代码

public T create(T t) {

EntityManager entityManager = null;

try {

//创建实体管理器

entityManager = createEntityManager();

entityManager.getTransaction().begin();

//保存对象

entityManager.persist(t);

entityManager.getTransaction().commit();

return t;

} catch (Exception e) {

if(entityManager!=null){

entityManager.getTransaction().rollback();

}

throw new SysException(e);

} finally{

closeEntityManager(entityManager);

}

}

private static EntityManagerFactory entityManagerFactory;

private static EntityManagerFactory getFactoryInstance(){

if(entityManagerFactory==null){

Context context;

DataSource ds = null;

Map<String,String> properties = new HashMap<String, String>();

Connection conn = null;

try {

context = new InitialContext();

//ds = (DataSource) context.lookup("java:comp/env/jdbc/customerservice\_db\_cus");

ds = (DataSource) context.lookup("java:comp/env/jdbc/mypagesybase");

conn = ds.getConnection();

String dbType = conn.getMetaData().getDatabaseProductName();

if(dbType.toLowerCase().contains(Constant.DB\_ORACLE)){//数据库类型是oracle

properties.put("hibernate.dialect", "org.hibernate.dialect.Oracle10gDialect");

}else if(dbType.contains(Constant.DB\_SQL\_SERVER)){//数据库类型是sqlserver

properties.put("hibernate.dialect","org.hibernate.dialect.SQLServerDialect");

}else if(dbType.toLowerCase().contains(Constant.DB\_DB2)){//数据库类型是db2

properties.put("hibernate.dialect","org.hibernate.dialect.DB2Dialect");

}else if(dbType.toLowerCase().contains(Constant.DB\_SYBASE)){ properties.put("hibernate.dialect","org.hibernate.dialect.SybaseDialect");

}else{

//query database

Statement stmt = conn.createStatement();

String value = null;

ResultSet rs = stmt.executeQuery("select \* from t\_CONFIG where key\_code='"+Constant.DB\_TYPE+"'");

while(rs.next()){

value = rs.getString("value");

properties.put("hibernate.dialect",value);

}

if(value==null){

System.err.println("-------------------------------------------");

System.err.println("[ERROR] database db\_mypage key 'dbtype' not found in table t\_config!");

System.err.println("[ERROR] Please contact local IT!");

System.err.println("-------------------------------------------");

throw new SQLException("[ERROR] You should configure key 'dbtype' in table t\_config in database mypage!");

}

}

} catch (NamingException e) {

e.printStackTrace();

} catch (SQLException e) {

e.printStackTrace();

}finally{

try {

conn.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

entityManagerFactory = Persistence.createEntityManagerFactory("persistenceUnit1",properties);

}

return entityManagerFactory;

}

## 修改实体

已持久化的实体，修改后可以通过merge()方法将其重新保存，例如：

em.merge(bsCustomer);

### 示例代码

EntityManager entityManager = null;

try {

entityManager = createEntityManager();

entityManager.getTransaction().begin();

entityManager.merge(t);

entityManager.getTransaction().commit();

return t;

} catch (Exception e) {

if(entityManager!=null){

entityManager.getTransaction().rollback();

}

throw new SysException(e);

} finally{

closeEntityManager(entityManager);

}

## 删除实体

将持久化的实体从数据库中删除可以通过remove()方法。例如：em.remove(customer);

### 示例代码

public void remove(Serializable id) {

EntityManager entityManager = null;

try {

entityManager = createEntityManager();

entityManager.getTransaction().begin();

T t = (T) entityManager.find(persistentClass, id);

if(t!=null){

t = entityManager.merge(t);

entityManager.remove(t);

}

entityManager.getTransaction().commit();

} catch (Exception e) {

if(entityManager!=null){

entityManager.getTransaction().rollback();

}

throw new SysException(e);

} finally{

closeEntityManager(entityManager);

}

}

## 根据主键查询实体

通过实体管理器的find()方法或getReference()方法可以根据主键查询实体，与前者不同，后者在没有找到时不是返回null，而是抛出异常。例如：

BsCustomer bsCustomer = em.find(BsCustomer.class, custId);

### 示例代码

private Class<T> persistentClass;

public T find(Serializable id) {

EntityManager entityManager = null;

try {

entityManager = createEntityManager();

T t = (T) entityManager.find(persistentClass, id);

if (t == null){

return null;

}

return t;

} catch (Exception e) {

throw new SysException(e);

} finally{

closeEntityManager(entityManager);

}

}

## 刷新实体

如果当前被管理的实体已经不是数据库中最新的数据，则可以通过refresh()方法刷新实体。

### 示例代码

例如：em. refresh(bsCustomer);

## 刷新实体到数据库（手动刷新）

当调用persist()、merge()和remove()这些方法时，更新并不会立刻同步到数据库中，直到容器决定刷新到数据库中才会执行。在默认情况下，容器决定刷新是在“相关查询”（除find()和getReference()之外）执行前或事务提交时发生的，如果你需要在事务提交之前将更新刷新到数据库中，可调用flush()方法，即手动来刷新数据库。

### 示例代码

例如：em.flush(bsCustomer);

## 设置刷新模式

如果不手动调用刷新，则只能依赖于容器的自动刷新。在默认情况下容器是自动刷新的(模式为FlushModeType.AUTO)。可通过setFlushMode()方法设置刷新模式

### 示例代码

例如：em.setFlushMode(FlushModeType.COMMIT);

## 分离实体

当处理了大量的实体后，这些实体都会存在于实体管理器中，这将会消耗大量的内存，使程序运行变慢。如果要减少消耗，则可以使用clear()方法，将正在被管理的实体从持久化内容中分离出来。

### 示例代码

例如：em.clear();

## 执行复杂查询

要执行复杂查询，需要利用实体管理器（EntityManager）建立Query对象。

### 示例代码

public List<T> query(SqlParameters sqlParameters) {

EntityManager entityManager = null;

try {

entityManager = createEntityManager();

Query query = createQuery(entityManager, sqlParameters);

List<T> resultList = query.getResultList();

return resultList;

} catch (Exception e) {

throw new SysException(e);

} finally{

closeEntityManager(entityManager);

}

}

private Query createQuery(EntityManager em, SqlParameters sqlParameters) {

StringBuffer sql = sqlParameters.getSql();

Map<String, Object> parameters = sqlParameters.getParameters();

Query query = em.createQuery(sql.toString());

Set<String> keys = parameters.keySet();

Iterator<String> iterator = keys.iterator();

String key;

Object data;

while (iterator.hasNext()) {

key = iterator.next();

data = parameters.get(key);

query.setParameter(key, data);

}

return query;

}

## 使用原生SQL查询（重点）

EntityManager API 提供了创建 Query 实例以执行原生 SQL 语句的**createNativeQuery()**方法。

### 查询的结果是对象数组的集合

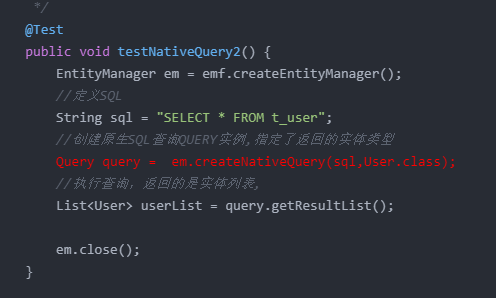
使用的方法：public Query createNativeQuery(String sqlString);

示例：



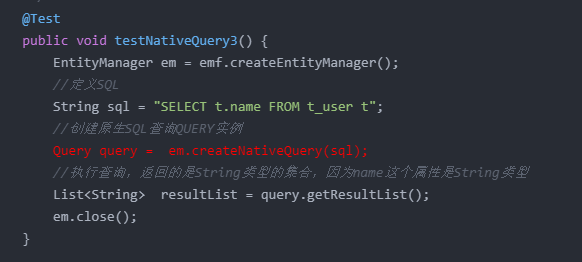
### 查询的结果是实体的集合

使用的方法：public Query createNativeQuery(String sqlString, Class resultClass);



### 查询单个属性的集合

使用方法：public Query createNativeQuery(String sqlString);



### 查询多个属性值的数组集合

使用方法：public Query createNativeQuery(String sqlString);



## JPQL（重点）

### 基本概念

在 [Java EE](http://lib.csdn.net/base/javaee) 中，JPQL（ [Java](http://lib.csdn.net/base/javase) 持久性查询语言）是专门为Java 应用程序访问和导航实体实例设计的。

JPQL 和 SQL 有很多相似之处。归根结底，它们都用于访问和操作[数据库](http://lib.csdn.net/base/mysql)数据。而且二者都使用非过程语句 — 通过特殊解释程序识别的命令。此外，JPQL 在语法上与 SQL 也相似。JPQL 和 SQL 的主要区别在于，前者处理 JPA 实体，后者直接在数据库空间内对表、列、行等关系数据进行处理。

要从 Java 代码内发出 JPQL 查询，您需要利用 EntityManager API 和 Query API 的相应方法。

### 执行以下一般步骤

（1）使用注入或通过 EntityManagerFactory 实例获取一个 EntityManager 实例。

（2）通过调用相应 EntityManager 的方法（如 createQuery），创建一个 Query 实例。

（3）如果有查询参数，使用相应 Query 的 setParameter 方法进行设置。

（4）如果需要，使用 setMaxResults 和/或 setFirstResult Query 的方法设置要检索的实例的最大数量和/或指定检索的起始实例位置。

（5）如果需要，使用 setHint Query 的方法设置供应商特定的提示。

（6）如果需要，使用 setFlushMode Query 的方法设置查询执行的刷新模式，覆盖实体管理器的刷新模式。

（7）使用相应 Query接口的方法 getSingleResult 或 getResultList 执行查询。如果进行更新或删除操作，您必须使用 executeUpdate 方法，它返回已更新或删除的实体实例的数量。

JPQL的查询可以分为命名查询和动态查询。

### 动态查询

可以使用EntityManager.createQuery方法创建动态查询，唯一的要求是把合法的JPQL语句传递给此方法。如下：

Query query = em.createQuery(“select p from Person p where p.id=1033”);

其中where语句是可选的。在这里JPQL看上去和SQL语句很像,但应当注意到的是from后边的Person是实体Bean而不是数据表。

在所写的JPQL语句中你可以像示例中那样的直接将查询条件写在语句中，但是还有更好的方法。在这里你可以使用设置查询参数的方式，其中又有位置参数和命名参数的分别。

#### 使用位置参数

使用位置参数如下所示：

Query query = em.createQuery(“select p from Person p where p.id=?1”);

Query.setParameter(1, 1033);//第一个参数是位置，第二个参数是查询条件

#### 使用命名参数

使用命名参数如下所示：

Query query = em.createQuery(“select p from Person p where p.id=:id”);

Query.setParameter(“id”, 1033);//第一个参数是参数名称，第二个参数是查询条件

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”:”。

如果你需要传递java.util.Date或java.util.Calendar参数进一个参数查询，你需要使用一个特殊的setParameter()方法。因为一个Date或Calendar对象能够描述一个真实的日期、时间或时间戳，所以我们需要告诉Query对象怎么使用这些参数，我们把javax.persistence.TemporalType作为参数传递进setParameter方法，告诉查询接口在转换java.util.Date或java.util.Calendar参数到本地SQL时使用什么数据库类型。

### 查询结果

使用Query. getSingleResult()方法得到查询的单个实例,返回Object。要确保使用此方法时查询只检索到一个实体。

使用Query. getResultList()方法得到查询的实例集合,返回List。通常我们会如示例中所示的获取查询返回的实体，但是在JPQL里我们也可以得到实体的部分属性，就如同使用SQL得到表中的部分列一样。如果是获取部分属性的话，Query.getResultList方法返回的会是Object数组的List每个Object数组项相当于是一条结果，数组的成员是属性值，顺序和所写的JPQL中的SELECT中所写顺序一致。

### 查询中使用构造器

可以在SELECT子句中使用构造器返回一个或多个java实例。如下所示：

Query query = em.createQuery("select new com.demo.bean.Person(p.id, p.name) from Person p order by p.id desc");

### 查询分页

JPA提供了在结果集合中使用分页的功能，使用这个功能我们可以轻松的达到对查询结果分页的目的。如下：

Query.setMaxResults(10);//设置分页大小，在这里为每页10条记录

Query.setFirstResult(10);//指定第一个结果的位置。这里是指定第11条作为起始结果

这里只要在setFirstResult中使用动态参数即可方便的对结果进行分页了。

### 使用操作符

在where子句中我们可以使用一些操作符来进行条件的选择。例如：

NOT操作符

select p from Person p where not(p.id = 1036)

//查询id不是1036的所有人

### 批量删除和更新

JPQL支持批量删除和批量更新的操作。和查询相同的也是要使用EntityManager.createQuery方法来创建一个Query实例，不同的是在这里要使用Query.executeUpdate方法来执行删除和更新，该方法返回的值是操作的记录数。

### 命名查询

可以在实体bean上通过@NamedQuery or @NamedQueries预先定义一个或多个查询语句，减少每次因书写错误而引起的BUG。通常把经常使用的查询语句定义成命名查询。

#### 定义单个命名查询

@NamedQuery(name="getPerson", query= "select p from  Person p where p.id=?1")

@Entity

public class Person implements Serializable{}

#### 定义多个命名查询

应在@javax.persistence.NamedQueries里定义@NamedQuery：

@NamedQueries({

@NamedQuery(name="getPerson", query= "select p from  Person p where p.id=?1"),

@NamedQuery(name="getPersonList", query= "select p from Person as p where p.age>?1")

})

@Entity

public class Person implements Serializable{}

当命名查询定义好了之后，我们就可以通过名称执行其查询。代码如下：

Query query = em.createNamedQuery("getPerson");

query.setParameter(1, 1);

说明：

（1）JPQL语句的大小写敏感性：除了Java 类和属性名称外，查询都是大小写不敏感的。

（2）使用参数查询的时候除了基本类型的参数还可以使用实体参数。

# persistence.xml配置文件示例

## 文件内容

<?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" version="1.0"

xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence\_1\_0.xsd">

<!--配置持久化单元-->

//name属性用于定义持久化单元的名字 (name必选,空值也合法)，transaction-type 指定事务类型(可选)

<persistence-unit name="persistenceUnit" transaction-type="RESOURCE\_LOCAL">

//javax.persistence.PersistenceProvider接口的一个实现类(可选)

<provider>org.hibernate.ejb.HibernatePersistence</provider>

<!--数据库连接 Jta-data-source和 non-jta-data-source用于分别指定持久化提供商使用的JTA和/或non-JTA数据源的全局JNDI名称(可选)-->

<non-jta-data-source>java:comp/env/jdbc/mypagesybase</non-jta-data-source>

<!-- 声明orm.xml所在位置.即指定映射文件 (可选) -->

<mapping-file>product.xml</mapping-file>

<!-- 以包含persistence.xml的jar文件为基准的相对路径,添加额外的jar文件，指定托管类的附加JAR(可选) -->

<jar-file>../lib/model.jar</jar-file>

**<!--持久化类-->**

<class>com.aiatss.mypage.domain.model.WebLeadAssignment</class>

<!--设置属性-->

<properties>

<!--自动输出schema创建DDL语句 value的值可以是 create-drop / create / update 值得注意的是create-drop会在SESSION关闭后删除所有表开启时再创建。 -->

<property name="hibernate.hbm2ddl.auto" value="update" />

<property name="hibernate.max\_fetch\_depth" value="3"/>

<property name="hibernate.jdbc.fetch\_size" value="18"/>

<property name="hibernate.jdbc.batch\_size" value="10"/>

<property name="hibernate.show\_sql" value="false"/>

<!--格式化SQL-->

<property name="hibernate.format\_sql" value="false"/>

</properties>

</persistence-unit>

</persistence>

说明：这个文件放在META-INF文件夹下面。

## 作用

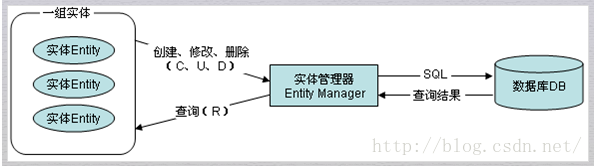
JPA的配置文件用来映射[PU](https://www.baidu.com/s?wd=PU&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)(Persist Unit)的，作用是映射表和类，里面也可以配置数据库连接信息。

# 实体（Entity）

JPA能够将普通的Java对象(有时被称作POJO) 映射到数据库，这些Java对象被称作Entity（实体）。一个实体的定义符合JavaBean的规范，因此常常称为实体Bean。实体Bean的每个属性都定义为私有的，并且有对应的Setter和Getter方法。JPA将实体映射到数据库有两种方法，一种是基于xml文件，一种是基于注解。

# 实体管理器（EntityManager）

EntityManager是应用程序访问持久化上下文中实体的接口，是Java实体对象与数据库交互的中介，它负责管理一组对应的实体，包括这组实体的CRUD操作。同时实体管理器也负责与持久化上下文(Persistence Context)进行交互，可以实现对实体不同状态进行转换操作等。



实体管理器通过实体管理器工厂（EntityManagerFactory）建立，但是当使用容器托管的实体管理器（在Java EE环境中）时，应用不和实体管理器工厂交互，实体管理器直接通过依赖注入或JNDI获得，容器负责与实体管理器工厂交互。

注入实体管理器可以用@PersitenceContext注解来注入EntityManager，其属性 type指明持久化上下文的作用范围，默认是TARNSACTION。若选择EXTENDED（扩展的方式），可使得多次调用使用同样的上下文，它的意义在于对有状态的Bean可以跨事务操作实体。

当使用应用托管实体管理器时，应用必须使用实体管理器工厂来管理实体管理器和持久化上下文的生命周期。EntityManagerFactory接口供应用来创建应用管理的实体管理器。EntityManagerFactory中的方法都是线程安全的，注入实体管理工厂在Java EE容器内，可以用@PersistenceUnit注解注入或者通过JNDI查找获得EntityManagerFactory，其unitName属性（可选）用于指明使用哪个持久性单元。

# 常用的注解

下列注解是JPA常用注解

## @Entity

这个注解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

## @Table

当实体类与其映射的数据库表名不同名时需要使用 @Table 注解说明，该标注与 @Entity 标注并列使用，置于实体类声明语句之前，@Table 注解的常用选项是 name，用于指明数据库的表名。

## @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，一般写在属性语句上面的一行。

示例：

@Id

@Column(name = "p1\_nric",nullable=false)

private String p1Nric;

## @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。

示例：

@Column(name = "sender", nullable = false)

private String sender;

## @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。 在 javax.persistence.GenerationType 中定义了以下几种可供选择的策略：

（1）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，适用于Sql server数据库。

（2）AUTO： JPA自动选择合适的策略，这是默认选项；

（3）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，用于Oracle数据库。

（4）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。

示例：

@Id

@SequenceGenerator(name = "email\_seq", sequenceName = "email\_seq", allocationSize = 1)

@GeneratedValue(strategy = GenerationType.AUTO, generator="email\_seq")

@Column(name = "id")

private Long id;

## @Transient

表示该属性并非是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。当实体对象中有一些独有属性时，可以使用该注解进行忽略。示例：

@Table(name = "CUSTOMERS")

@Entity

public class Customer {

@Column(name = "ID")

@GeneratedValue(strategy = GenerationType.AUTO)

@Id

private Integer id;

@Column(name = "Name")

private String name;

@Column(name = "Email")

private String email;

@Column(name = "Age")

private int age;

@Transient

private String address;

}

数据库的DDL：

CREATE TABLE `customers` (

`ID` int(11) NOT NULL AUTO\_INCREMENT,

`Age` int(11) DEFAULT NULL,

`Email` varchar(255) DEFAULT NULL,

`Name` varchar(255) DEFAULT NULL,

PRIMARY KEY (`ID`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

数据库表中不会创建address列。

## @SequenceGenerator

指定序列名，一般用于Oracle数据库，示例：

@Entity

@Table(name = "jdps\_content")

@SequenceGenerator(name="SEQ\_JDPS\_content",allocationSize=1,initialValue=1,sequenceName="SEQ\_JDPS\_CONTENT")

public class  Content {

    @Id

    @GeneratedValue(strategy=GenerationType.SEQUENCE,generator="SEQ\_JDPS\_content")

    private int  id;

## @PersistenceContext

注入的是实体管理器，执行持久化操作，需要配置persistence.xml文件。

示例：

@PersistenceContext

private EntityManager em;

注入一堆保存实体类状态的数据结构，针对实体类的不同状态可以做出不同的反应，其实就是把数据从数据库里取出然后在内存里处理，再返回数据库的法则。

# 常用的接口和类

## EntityManagerFactory接口

用于与持久性单元的实体管理器工厂交互的接口。当应用程序完成使用实体管理器工厂和/或应用程序关闭时，应用程序应关闭实体管理器工厂。 一旦EntityManagerFactory关闭，其所有实体管理器都被视为处于关闭状态。

这是一个接口，常用的方法：

### public EntityManager createEntityManager();

//创建一个新的应用程序管理的EntityManager，该方法每次调用时都会返回一个新的EntityManager实例。isOpen方法将在返回的实例上返回true。

### public EntityManager createEntityManager(Map map);

//使用指定的Map属性创建新的应用程序管理的EntityManager。

### public void close();

关闭工厂，释放它所拥有的任何资源。关闭工厂实例后，在其上调用的所有方法都将抛出IllegalStateException，但isOpen除外，它将返回false。 一旦EntityManagerFactory关闭，其所有实体管理器都被视为处于关闭状态。

### public Map<String, Object> getProperties();

获取对实体管理器工厂有效的属性和关联值。 更改Map内容不会更改有效的配置。

## EntityManager接口

用于与持久性上下文交互的接口。EntityManager实例与持久性上下文相关联。 持久化上下文是一组实体实例，其中对于任何持久性实体标识，存在唯一的实体实例。 在持久化上下文中，管理实体实例及其生命周期。EntityManager API用于创建和删除持久性实体实例，按主键查找实体以及查询实体。可由给定EntityManager实例管理的实体集由持久性单元定义。 持久性单元定义由应用程序关联或分组的所有类的集合，并且必须在映射到单个数据库时将其共同定位。

这是一个接口，常用的方法：

### public void persist(Object entity);

//保存一个实体

### public <T> T merge(T entity);

//修改一个实体

### public void remove(Object entity);

//删除一个实体

### public <T> T find(Class<T> entityClass, Object primaryKey);

//根据主键查询实体

### public <T> T find(Class<T> entityClass, Object primaryKey,Map<String, Object> properties);

//查询一个实体

### public Query createQuery(String qlString);

创建Query实例以执行Java Persistence查询语言

参数：

qlString：Java Persistence查询语言

### public Query createNamedQuery(String name);

创建Query实例以执行命名查询（使用Java持久性查询语言或本地SQL）。

参数：

name：在元数据中定义的查询名字

返回：新的Query实例

### public <T> T unwrap(Class<T> cls);

返回指定类型的对象以允许访问特定于提供程序的API。 如果提供程序的EntityManager实现不支持指定的类，则抛出PersistenceException。

参数：

cls：要返回的对象的类。这通常是底层的EntityManager实现类或它实现的接口。

返回：特定类的实例

## SessionImplementor接口

## SQLQuery接口

这个接口在hibernate-core-3.6.9.Final.jar这个包中，表示“本地sql”查询，并允许用户定义有关其执行的某些方面，例如：结果集值映射

## Query接口

这个接口在hibernate-core-3.6.9.Final.jar这个包里，Hibernate查询的面向对象表示。 通过调用Session.createQuery（）获取Query实例。 此接口公开了Session.iterate（）和Session.find（）提供的一些额外功能。

### 方法

#### public Query setResultTransformer(ResultTransformer transformer)

设置处理查询结果的策略。 这可用于更改查询结果的“形状”。

参数：transformer 申请一个transformer对象

返回：Query接口（方法链接）

## Query接口（常用）

这个接口在hibernate-jpa-2.0-api-1.0.1.Final.jar这个包里，接口所在的包名是javax.persistence，表示HQL/ JPQL查询或已编译的Criteria查询。 也充当JPA Query / TypedQuery契约的Hibernate扩展。

### 方法

#### <T> T unwrap(Class<T> cls)

返回指定类型的对象以允许访问特定于提供程序的API。如果提供程序的查询实现不支持指定的类，则抛出PersistenceException。

参数：

cls：要返回的对象的类。 这通常是底层查询实现类或它实现的接口。

返回：指定类的实例

#### Query setParameter(int position, Object value)

将参数绑定到位置参数。

参数：

position：位置

value：参数的值

返回值：相同的query实例

#### List getResultList()

执行SELECT查询并将查询结果作为无类型列表返回。

返回：结果列表

## Persistence类

提供对EntityManagerFactory的访问的Bootstrap类。

这是一个类，常用的方法：

### 方法

#### public static EntityManagerFactory createEntityManagerFactory(String persistenceUnitName)

为命名的持久性单元创建并返回EntityManagerFactory。

参数：持久性单元的名称

#### public static EntityManagerFactory createEntityManagerFactory(String persistenceUnitName, Map properties)

使用给定的属性为命名的持久性单元创建并返回EntityManagerFactory。

参数：

persistenceUnitName：持久性单元的名称

properties：创建工厂时要使用的其他属性。 这些属性的值将覆盖可能已在其他位置配置的任何值

## Transformers类

# [JPA EntityManagerFactory与Hibernate’s SessionFactory](https://www.cnblogs.com/drizzlewithwind/p/6159651.html)的区别

JPA使用EntityManagerFactory开闭session，而Hibernate使用SessionFactory开闭session。两者区别：

（1）EntityManagerFactory是JPA的标准API，如果使用EntityManagerFactory，在更换实现JPA的ORM框架时，需要改动的代码很少。如果使用SessionFactory，因为SessionFactory是Hibernate的，无法直接更换ORM框架，更换ORM的话，需要修改所有的代码。

（2）如果想使用Hibernate的某些高级特性，只能使用SessionFactory，使用EntityManagerFactory无法享受Hibernate的高级用法。

# 常见的错误

## 1、No identifier specified for entity

这个问题的原因是找不到主键，解决方法：在主键字段上面添加如下代码，

@Id

 @GeneratedValue(strategy= GenerationType.AUTO)

说明：strategy= GenerationType.AUTO中的AUTO应当换成你所使用的数据库的主键生成方式。

## 2、The datastream for token 236 should only be sent after other datastreams. It can not be a command datastream on it's own. This is an internal error.

解决方法：

只能利用传统的Connection和ResultSet的形式：  
resultSet = null;  
try {  
cstmt = getCurrentSession(entityClass).connection().prepareCall(  
"{call " + procedure + "}");  
resultSet = cstmt.executeQuery();  
} catch (HibernateException e) {  
this.closeConnection();  
e.printStackTrace();  
logger.log(RMTManagementLevel.ERROR, e);  
} catch (SQLException e) {  
this.closeConnection();  
e.printStackTrace();  
logger.log(RMTManagementLevel.ERROR, e);  
}

## 3、org.hibernate.exception.JDBCConnectionException:could not execute query

解决方法：需要修改表结构，为这个表增加主键。