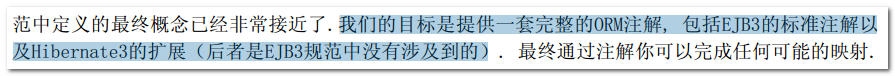
Hibernate5 第五天

# JPA注解开发

jpa是sun公司的一个ORM规范，只有接口和注解，没有具体实现。

jpa是EJB3中的。

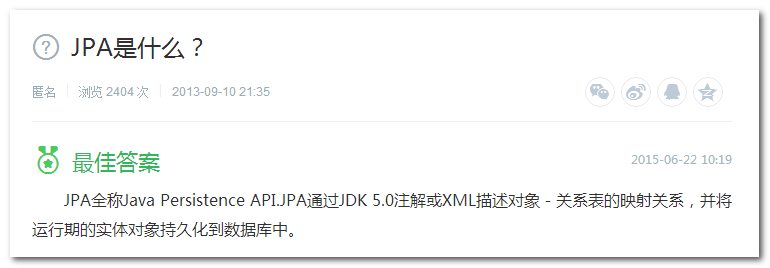
Hibernate官网中有这么一句话：



hibernate中有两套注解规范：一套jpa，一套自己的；

使用注解开发，开发效率高！

网络搜索：



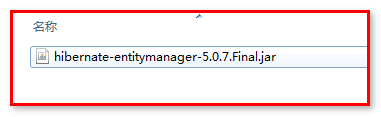
## 单表常用注解

新建工程：Hibernate5\_day05\_jpa

导入jar、配置文件、工具类等

同时引入：【1-Hibernate\_D01\_第一天\Hibernate\_D01\_课前资料\hibernate5\资料\hibernate-release-5.0.7.Final\lib\jpa】

表示对jpa接口包的实现



第一步：在cn.itcast.a\_single包中建立Book实体

第二步：最简注解示例: (使用了注解的默认值)

【最最小化配置】：

默认表名就是类名，属性字段名和数据库列名一致

@Entity

**public** **class** Book {

@Id

@GeneratedValue(strategy=GenerationType.***IDENTITY***)

**private** Integer id;

**private** String name;

**private** Double price;

}

第三步：Hibernate.cfg.xml配置映射:

<!-- 配置实体映射文件 -->

<mapping class=*"cn.itcast.a\_single.Book"*/>

建表测试

第四步：测试：创建TestBook.java

**public** **class** TestBook {

@Test

**public** **void** save() {

// 开启一个Session

Session session = HibernateUtils.*openSession*();

Transaction tran = session.beginTransaction();

Book book = **new** Book();

book.setName("降龙十八掌");

book.setPrice(10d);

session.save(book);

tran.commit();

session.close();

}

}

【推荐标准最小化配置】：

@Entity

@Table(name="t\_book")

**public** **class** Book {

@Id

@GeneratedValue(strategy=GenerationType.***AUTO***)

**private** Integer id;

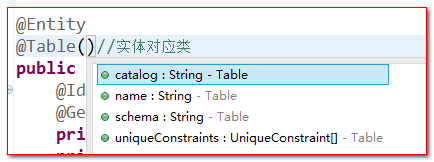
**private** String name;

**private** Double price;

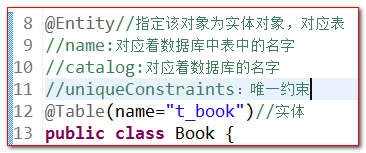
}

更多常用注解

实体和表本身相关：

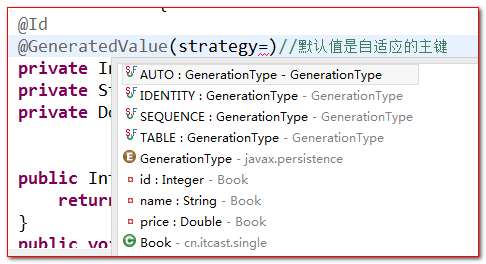


@Table属性的详细解释



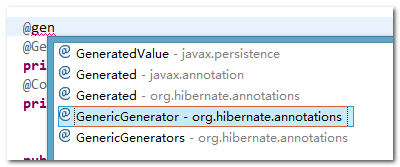
主键相关的：

@generatedValue的属性表示主键生成策略



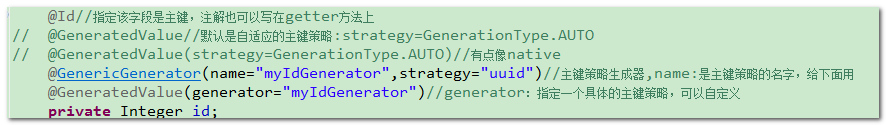
Auto相当于native,默认值

自定义主键策略(下面使用hibernate的实现):



自定义主键字段：

例如，将主键设置成UUID的形式



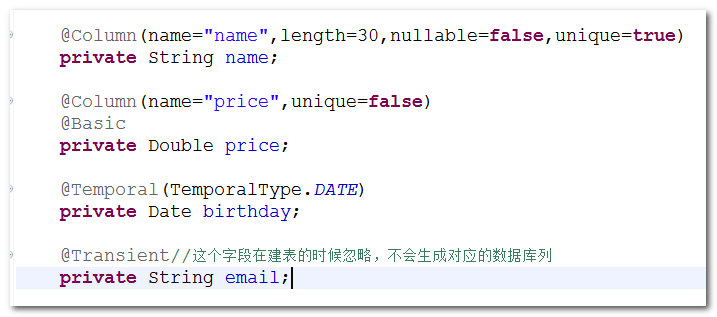
@GeneratedValue只支持 IDENTITY SEQUENCE 和 TABLE

如果使用其它类型的生成策略需要自定义

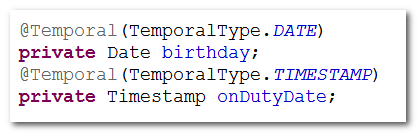
@GenericGenerator 用于自定义生成策略

@GeneratedValue 的 generator 用于指定自定义生成策略

其他字段相关的：



@Temporal 生成日期类型（扩展）



默认实体类所有属性都生成表中列

@Transient：

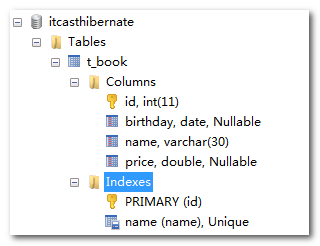
可以通过 @Transient 控制数据库不生成对应列

@Basic：

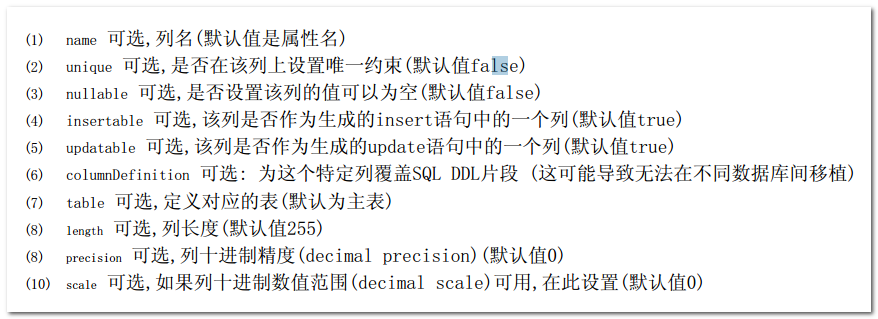
表示该字段有效，数据库中会生成对应的列（默认）

如果在成员属性没有加入任何注解，则默认在前面加入了@Basic

生成的数据库结构：



@Column属性字段官方参考配置：



【较完整配置】：

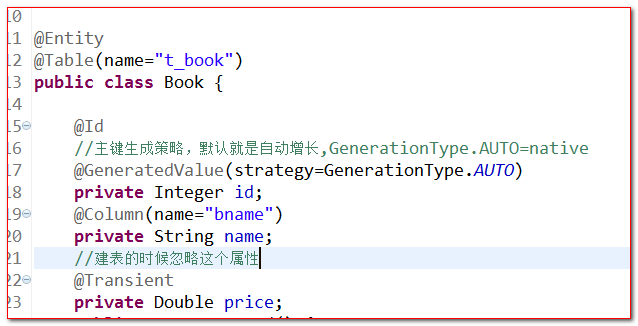


【补充】：

注解：可以放到属性上面设置，也可以在getter方法上设置，效果一样。但是：要么都放属性，要么都放getter，不能混着用。

这里我们放置到属性上即可

配置Book.java



## 多表常用注解

### 一对多

新建订单表实体类，并与客户表建立实体关系。

创建包：cn.itcast.b\_oneToMany

最小化配置：

配置Customer.java

@Entity

@Table(name="t\_customer")

**public** **class** Customer {

@Id

@GeneratedValue(strategy=GenerationType.***AUTO***)

**private** Integer id;

**private** String name;

**private** String city;

@OneToMany(mappedBy="customer")// 相当于inverse=true，配置当前Customer对象在Order实体中的名称

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

}

配置Order.java

@Entity

@Table(name="t\_order")

**public** **class** Order {

@Id

@GeneratedValue(strategy=GenerationType.***AUTO***)

**private** Integer id;

**private** String name;

**private** Double price;

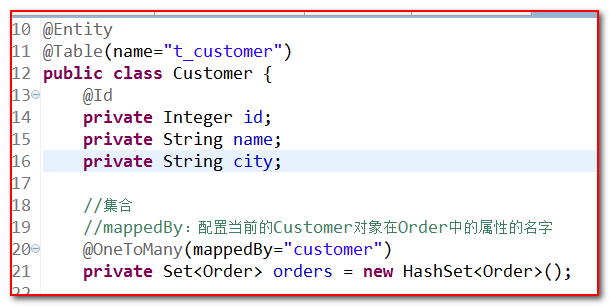
@ManyToOne

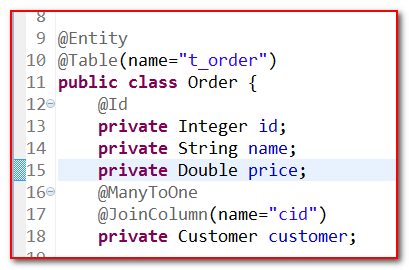
@JoinColumn(name="cid")

**private** Customer customer ;

}

推荐配置（简单）：





测试类testOneToMany：新增：

**public** **class** TestOneToMany {

@Test

**public** **void** testSave(){

Session session = HibernateUtils.*openSession*();

Transaction tran = session.beginTransaction();

//客户对象

Customer c = **new** Customer();

c.setName("黄蓉");

c.setCity("桃花岛");

//订单对象

Order o = **new** Order();

o.setName("中国电线");

o.setPrice(12d);

//建立关联关系

c.getOrders().add(o);

o.setCustomer(c);

session.save(c);

session.save(o);

tran.commit();

session.close();

}

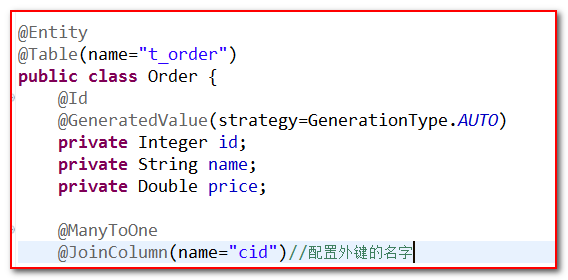
}

更多详细配置：

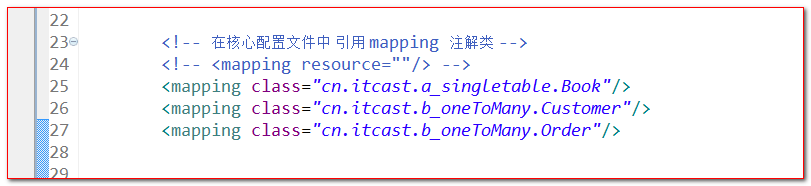
Customer.java



Order.java



映射文件加入到核心文件：



### 多对多

示例：学生和课程

建立实体类，并加上注解：

【最小化配置】：

Student.java

@Entity

@Table(name="t\_student")

**public** **class** Student {

@Id

@GeneratedValue(strategy=GenerationType.***AUTO***)

**private** Integer id;

**private** String name;

/\*\*多对多，多的一端的配置\*/

@ManyToMany

@JoinTable(name="t\_s\_c",joinColumns=@JoinColumn(name="sid"),inverseJoinColumns=@JoinColumn(name="cid"))

**private** Set<Course> courses = **new** HashSet<Course>();

}

Course.java

@Entity

@Table(name="t\_course")

**public** **class** Course {

@Id

@GeneratedValue(strategy=GenerationType.***AUTO***)

**private** Integer id;

**private** String name;

/\*\*多对多，多的一端的配置\*/

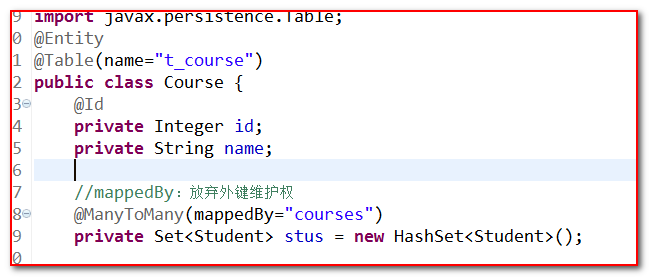
@ManyToMany(mappedBy="courses")

Set<Student> students = **new** HashSet<Student>();

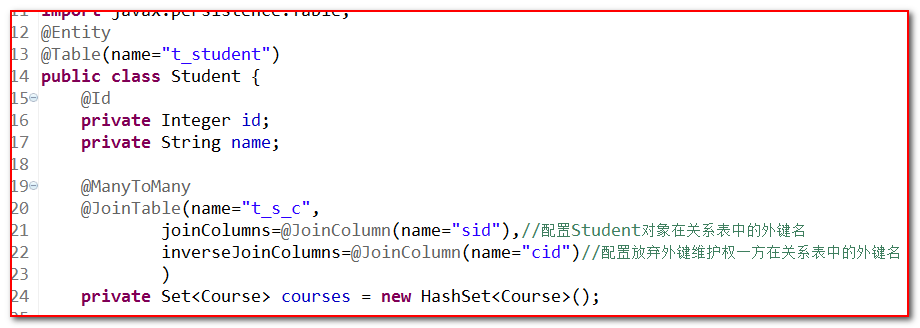
}

【推荐配置】：

Course.java



Student.java



测试：

**public** **class** TestManyToMany {

@Test

**public** **void** testSave(){

Session session = HibernateUtils.*openSession*();

Transaction tran = session.beginTransaction();

Student s = **new** Student();

s.setName("黄蓉");

Course c = **new** Course();

c.setName("struts2");

s.getCourses().add(c);

//c.getStudents().add(s);

session.save(s);

session.save(c);

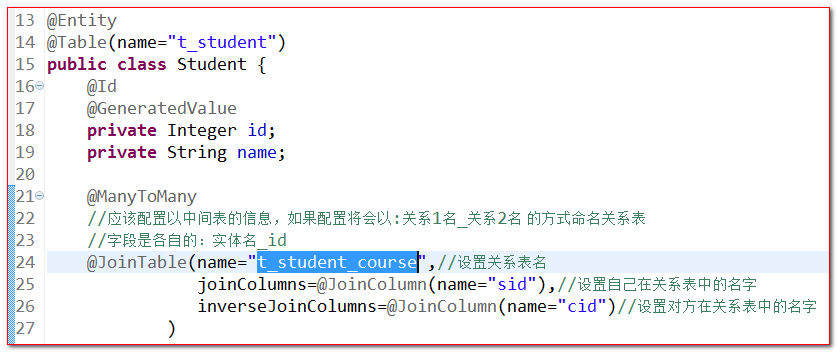
tran.commit();

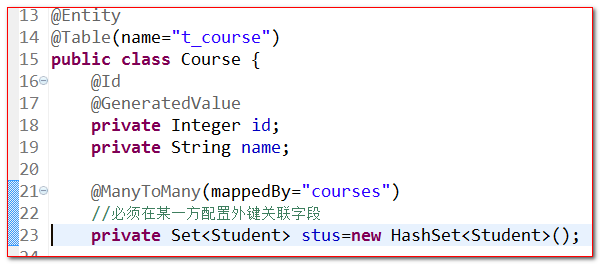
session.close();

}

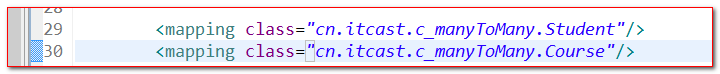
}

【更多配置】：





在hibernate.cfg.xml核心配置文件中配置映射：



建表测试：。。。。。

【注意】：

## 命名查询NamedQuery

@NamedQuery(name="Customer.findAll",query="from Customer")

@NamedQueries(value={@NamedQuery(name="query2",query="from Customer"),

@NamedQuery(name="query3",query="from Customer"),

@NamedQuery(name="query4",query="from Customer")

}

)

public class Customer {

}

**@NamedQuery 定义了命名查询语句，可以通过session.getNamedQuery()获取使用**

具体写法如下：

测试：

|  |
| --- |
| @Test  **public** **void** test02(){  Session session= HibernateUtils.*getCurrentSession*();  session.beginTransaction();    //query  List<Customer> list = session.getNamedQuery("Customer.findAll").list();  System.*out*.println(list.get(0).getOrders());    Customer c =(Customer)session.get(Customer.**class**, 1);  System.*out*.println(c.getOrders().size());    session.getTransaction().commit();  //session.close();    } |

【说明】发现在javax.persistence.\* 与org.hibernate.annotations.\*包中，都有NamedQuery包，导入哪个包都一样，但是在使用NamedQuery和NamedQueries的时候，需要导入统一的包中的资源

## 抓取策略

类抓取策略：

关联集合抓取策略：

@Fetch 控制生成SQL形式

FetchMode.JOIN 迫切做外连接

FetchMode.SELECT 多条SQL

FetchMode.SUBSELECT 子查询

@LazyCollection 控制是否延迟加载

LazyCollectionOption.TRUE 延迟加载

LazyCollectionOption.FALSE 立即加载

LazyCollectionOption.EXTRA 懒惰加载（及其懒惰）

jpa的：

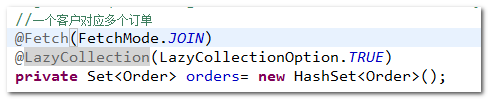
@OneToMany(mappedBy="customer"//自己在关联的对象中的属性（告诉jpa，两个对象怎么关联的，一方的id关联的）

,fetch=FetchType.*LAZY*//是否懒加载，默认是懒加载，抓取策略lazy延迟抓取，eager立即抓取

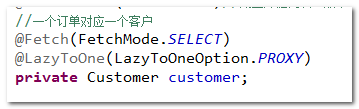
)

Hibernate的：

一的一方Customer.java：



多的一方Order.java：



测试类：

@Test

**public** **void** test03() {

Session session = HibernateUtils.*openSession*();

session.beginTransaction();

Customer customer = (Customer)session.get(Customer.**class**, 1);

System.***out***.println(customer);

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

session.getTransaction().commit();

session.close();

}

## 缓存策略

一定要在Hibernate的灵魂文件中配置二级缓存，否则注解是无效，而且运行会报错

在实体上打开二级缓存策略：



也可以在核心文件配置，但建议是在实体上配置。

配置：类级别的二级缓存策略



配置：集合级别的二级缓存**（依赖于类级别的缓存，在2个类上都要添加@Cache）**



配置文件：

<!-- 映射二级缓存存放的实体 -->

<class-cache usage=*"read-write"* class=*"cn.itcast.b\_oneToMany.Customer"*/>

<class-cache usage=*"read-write"* class=*"cn.itcast.b\_oneToMany.Order"*/>

<!-- 映射二级缓存存放的是集合，集合中可以存放实体对象-->

<collection-cache usage=*"read-write"* collection=*"cn.itcast.b\_oneToMany.Customer.orders"*/>

测试：

/\*\*测试二级缓存\*/

@Test

**public** **void** testSecondCache(){

//开启一个Session

Session session = HibernateUtils.*openSession*();

Transaction tran = session.beginTransaction();

//查询id是1的客户，查询数据库

Customer customer = (Customer) session.get(Customer.**class**, 1);

System.***out***.println(customer.hashCode());

//不会查询数据库，因为从一级缓存中读取

Customer customer2 = (Customer) session.get(Customer.**class**, 1);

System.***out***.println(customer2.hashCode());

tran.commit();

session.close();//一级缓存没有了

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//开启一个Session

Session session1 = HibernateUtils.*openSession*();

Transaction tran1 = session1.beginTransaction();

//查询id是1的客户，不会查询数据库，从hibernate的二级缓存中获取

Customer customer3 = (Customer) session1.get(Customer.**class**, 1);

System.***out***.println(customer3.hashCode());//与customer1和customer2不一致，说明封装的值一样，但是对象不一样，二级缓存中存放的是对象的散装数据

//不会查询数据库，因为从一级缓存中读取

Customer customer4 = (Customer) session1.get(Customer.**class**, 1);

System.***out***.println(customer4.hashCode());

tran1.commit();

session1.close();//一级缓存没有了

}

最终：大家能看懂吗？

|  |
| --- |
| //客户：一方  @Entity  @Table(name="t\_customer")  //hql  @NamedQuery(name="Customer.findAll" ,query="from Customer")//配置一个命名查询  @NamedQueries({@NamedQuery(name="Customer.findAll" ,query="from Customer"),@NamedQuery(name="Customer.findAll2" ,query="from Customer")})  //sql  @NamedNativeQuery(name="Customer.findcount",query="select count(\*) from t\_customer")//配置一个  @NamedNativeQueries({@NamedNativeQuery(name="Customer.findcount",query="select count(\*) from t\_customer"),@NamedNativeQuery(name="Customer.findcount2",query="select count(\*) from t\_customer")})  @Cacheable(**true**)//jpa打开了缓存  @Cache(usage=CacheConcurrencyStrategy.*READ\_WRITE*)  **public** **class** Customer {  @Id  @GeneratedValue  **private** Integer id;//oid属性  **private** String name;  **private** String city;  //关联集合属性  @OneToMany(mappedBy="customer"//自己在关联的对象中的属性（告诉jpa，两个对象怎么关联的，一方的id关联的）  ,cascade=CascadeType.*ALL*//级联配置  ,fetch=FetchType.*LAZY*//是否懒加载，默认是懒加载  ,targetEntity=cn.itcast.hibernate.a\_singlepo.Order.**class**//orders里面的元素对应的实体类的类型  //实体类可以有接口或抽象类，这里配置的实现类  )  @Fetch(FetchMode.*SELECT*)//hibernate的抓取策略  @LazyCollection(LazyCollectionOption.*TRUE*)//hiernate  @Cache(usage=CacheConcurrencyStrategy.*READ\_WRITE*)  **private** Set<Order> orders = **new** HashSet<Order>(); |