Java私塾《深入浅出学Spring Data JPA》

——系列系列精品教程

整体课程概览

- 系统学习Spring Data JPA 的核心开发知识，循序渐进，系统掌握
- 第一章：Spring Data JPA入门
  包括：是什么、能干什么、有什么、HelloWorld等
- 第二章：JpaRepository基本功能
  包括：代码示例JpaRepository提供的CRUD功能，还有翻页、排序等功能
- 第三章：JpaRepository的查询
  包括：解析方法名称以自动生成查询、 NamedQueries、 @Query指定查询、本地查询、命名化参数、更新查询、创建查询的顺序等内容
- 第四章：客户化扩展JpaRepository
  包括：讲述如何在JpaRepository基础上扩展我们自己需要的功能
- 第五章：Specifications查询
  包括：Specifications基础、 Criteria查询基本概念、 Criteria查询实现、多表联接等内容

第一章：Spring Data JPA入门

Spring Data JPA是什么

- Spring Data是什么

  Spring Data是一个用于简化数据库访问，并支持云服务的开源框架。其主要目标是使得对数据的访问变得方便快捷，并支持map-reduce框架和云计算数据服务。 Spring Data 包含多个子项目：

Commons - **提供共享的基础框架，适合各个子项目使用，支持跨数据库持久化**
JPA - **简化创建 JPA 数据访问层和跨存储的持久层功能**
Hadoop - 基于 Spring 的 Hadoop 作业配置和一个 POJO 编程模型的 MapReduce 作业
Key-Value - 集成了 Redis 和 Riak ，提供多个常用场景下的简单封装
Document - 集成文档数据库：CouchDB 和 MongoDB 并提供基本的配置映射和资料库支持
Graph - 集成 Neo4j 提供强大的基于 POJO 的编程模型
Graph Roo AddOn - Roo support for Neo4j
JDBC Extensions - 支持 Oracle RAD、高级队列和高级数据类型
Mapping - 基于 Grails 的提供对象映射框架，支持不同的数据库
Examples - 示例程序、文档和图数据库
Guidance - 高级文档

- Spring Data JPA是什么

  由Spring提供的一个用于简化JPA开发的框架

Spring Data JPA能干什么和有什么

- Spring Data JPA能干什么

  可以极大的简化JPA的写法，可以在几乎不用写实现的情况下，实现对数据的访问和操作。除了CRUD外，还包括如分页、排序等一些常用的功能。

- Spring Data JPA有什么

  主要来看看Spring Data JPA提供的接口，也是Spring Data JPA的核心概念：

1：Repository：最顶层的接口，是一个空的接口，目的是为了统一所有Repository的类型，且能让组件扫描的时候自动识别。

2：CrudRepository ： 是Repository的子接口，提供CRUD的功能

3：PagingAndSortingRepository：是CrudRepository的子接口，添加分页和排序的功能

4：JpaRepository：是PagingAndSortingRepository的子接口，增加了一些实用的功能，比如：批量操作等。

5：JpaSpecificationExecutor：用来做负责查询的接口

6：Specification：是Spring Data JPA提供的一个查询规范，要做复杂的查询，只需围绕这个规范来设置查询条件即可

HelloWorld-1

- 环境构建

在Eclipse里面构建一个普通的Java工程，主要就是要加入一堆的jar包。

1：首先去官网下载Spring Data Common 和 Spring Data JPA的包，把里面dist的jar包加入到工程中，这里是spring-data-commons-1.5.0.RELEASE.jar和spring-data-jpa-1.3.2.RELEASE.jar

2：把Spring3.2.3的jar包添加到工程中

3：JPA的实现选用的是Hibernate4.2.0，总共还需要额外加入如下的jar：

antlr-2.7.7.jar、aopalliance-1.0.jar、asm-3.2.jar、aspectjrt-1.7.1.jar、aspectjweaver-1.7.1.jar、commons-beanutils-1.8.3.jar、commons-codec-1.7.jar、commons-collections-3.2.1.jar、commons-dbcp-1.4.jar、commons-fileupload-1.2.2.jar、commons-io-2.4.jar、commons-lang3-3.1.jar、commons-logging-1.1.1.jar、commons-pool-1.6.jar、dom4j-1.6.1.jar、hibernate-commons-annotations-4.0.1.Final.jar、hibernate-core-4.2.0.Final.jar、hibernate-entitymanager-4.2.0.Final.jar、hibernate-jpa-2.0-api-1.0.1.Final.jar、javassist-3.15.0-GA.jar、jboss-logging-3.1.0.GA.jar、jboss-transaction-api_1.1_spec-1.0.0.Final.jar、mysql-connector-java-5.1.9.jar、slf4j-api-1.7.3.jar

HelloWorld-2

- 实体对象，就是以前的实现方式

```
@Entity
@Table(name="tbl_user")
public class UserModel {
    @Id
    private Integer uuid;
    private String name;
    private Integer age;
//省略getter/setter
}
```

- DAO的接口

```
public interface UserRepository extends JpaRepository<UserModel,
    Integer>{
    //空的，可以什么都不用写
}
```

无需提供实现，Spring Data JPA会为我们搞定一切

HelloWorld-3

n 写个逻辑层的Service，其实就相当于DAO的客户端，用来测试

```
@Service
@Transactional
public class Client {
    @Autowired
    private UserRepository ur;

    public void testAdd(UserModel um){    ur.save(um);        }

    public static void main(String[] args) {
          ApplicationContext ctx = new
    ClassPathXmlApplicationContext("applicationContext.xml");

          Client c = (Client)ctx.getBean("client");
          UserModel um = new UserModel();
          um.setAge(1);
          um.setName("张三");
          um.setUuid(1);

          c.testAdd(um);
    } }
```

HelloWorld-4

n 同样需要在Spring的配置文件中配置，基本跟使用注解的配置类似：

```
<?xml version="1.0" encoding="UTF-8"?>
<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:aop="http://www.springframework.org/schema/aop"
xmlns:tx="http://www.springframework.org/schema/tx"
xmlns:jpa="http://www.springframework.org/schema/data/jpa"
xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-
    3.0.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
http://www.springframework.org/schema/data/jpa http://www.springframework.org/schema/data/jpa/spring-jpa.xsd
">
<context:component-scan base-package="cn.javass">
    <context:exclude-filter
    type="annotation “ expression="org.springframework.stereotype.Controller"/>
</context:component-scan>
<aop:aspectj-autoproxy proxy-target-class="true"/>
<!-- 开启注解事务 只对当前配置文件有效 -->
<tx:annotation-driven transaction-manager="transactionManager" proxy-target-class="true"/>
```

HelloWorld-5

```
<jpa:repositories
        base-package="cn.javass"
        repository-impl-postfix="Impl"
        entity-manager-factory-ref="entityManagerFactory"
        transaction-manager-ref="transactionManager">
</jpa:repositories>
<bean id="entityManagerFactory"
      class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="packagesToScan" value="cn.javass"/>
    <property name="persistenceProvider">
        <bean class="org.hibernate.ejb.HibernatePersistence"/>
    </property>
    <property name="jpaVendorAdapter">
        <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
            <property name="generateDdl" value="false"/>
            <property name="database" value="MYSQL"/>
            <property name="databasePlatform"
value="org.hibernate.dialect.MySQL5InnoDBDialect"/>
            <property name="showSql" value="true"/>
        </bean>
    </property>
```

HelloWorld-6

```
        <property name="jpaDialect">
            <bean class="org.springframework.orm.jpa.vendor.HibernateJpaDialect"/>
        </property>
        <property name="jpaPropertyMap">
            <map>
                <entry key="hibernate.query.substitutions" value="true 1, false 0"/>
                <entry key="hibernate.default_batch_fetch_size" value="16"/>
                <entry key="hibernate.max_fetch_depth" value="2"/>
                <entry key="hibernate.generate_statistics" value="true"/>
                <entry key="hibernate.bytecode.use_reflection_optimizer" value="true"/>
                <entry key="hibernate.cache.use_second_level_cache" value="false"/>
                <entry key="hibernate.cache.use_query_cache" value="false"/>
            </map>
        </property>
    </bean>
```

HelloWorld-7

```
<!--事务管理器配置-->
    <bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory"/>
    </bean>

    <bean name="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
          <property name="driverClassName"><value>org.gjt.mm.mysql.Driver</value></property>
          <property
    name="url"><value>jdbc:mysql://localhost:3306/cc?useUnicode=true&amp;characterEncoding=UTF
    -8</value></property>
          <property name="username"> <value>root</value> </property>
          <property name="password" value="cc"/>
    </bean>
</beans>
```

- 配置完成后，可以去运行Client测试一下了，当然数据库和表需要先准备好

- 也可以在<jpa:repositories>下面添加filter，形如：

```
<repositories base-package="com.acme.repositories">
    <context:exclude-filter type="regex" expression=".*SomeRepository" />
</repositories>
```

第二章：JpaRepository基本功能

JpaReposiory的基本功能示范

- 具体的看代码演示

  其中：Pageable接口的实现类是PageRequest，Page接口的实现类是PageImpl。

  示例如下：

```
Page<UserModel> p =  ur.findAll(new PageRequest(0,2,new Sort(new
    Order(Direction.DESC,"uuid"))));
System.out.println("list="+p.getContent());
```

第三章：JpaRepository的查询

JpaRepository的查询功能-1

- 直接在接口中定义查询方法，如果是符合规范的，可以不用写实现，目前支持的关键字写法如下：

Keyword	Sample	JPQL snippet
And	findByLastnameAndFirstname	… where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	… where x.lastname = ?1 or x.firstname = ?2
Between	findByStartDateBetween	… where x.startDate between 1? and ?2
LessThan	findByAgeLessThan	… where x.age < ?1
GreaterThan	findByAgeGreaterThan	… where x.age > ?1
After	findByStartDateAfter	… where x.startDate > ?1
Before	findByStartDateBefore	… where x.startDate < ?1
IsNull	findByAgeIsNull	… where x.age is null

JpaRepository的查询功能-2

Keyword	Sample	JPQL snippet
IsNotNull,NotNull	findByAge(Is)NotNull	… where x.age not null
Like	findByFirstnameLike	… where x.firstname like ?1
NotLike	findByFirstnameNotLike	… where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	… where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	… where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	… where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	… where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	… where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	… where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> age)	… where x.age not in ?1
True	findByActiveTrue()	… where x.active = true
False	findByActiveFalse()	… where x.active = false

JpaRepository的查询功能-3

- Spring Data JPA框架在进行方法名解析时，会先把方法名多余的前缀截取掉，比如find、findBy、read、readBy、get、getBy，然后对剩下部分进行解析。
- 假如创建如下的查询：findByUserDepUuid()，框架在解析该方法时，首先剔除findBy，然后对剩下的属性进行解析，假设查询实体为Doc

1：先判断 userDepUuid （根据 POJO 规范，首字母变为小写）是否为查询实体的一个属性，如果是，则表示根据该属性进行查询；如果没有该属性，继续第二步；

2：从右往左截取第一个大写字母开头的字符串此处为Uuid），然后检查剩下的字符串是否为查询实体的一个属性，如果是，则表示根据该属性进行查询；如果没有该属性，则重复第二步，继续从右往左截取；最后假设user为查询实体的一个属性；

3：接着处理剩下部分（DepUuid），先判断 user 所对应的类型是否有depUuid属性，如果有，则表示该方法最终是根据“ Doc.user.depUuid” 的取值进行查询；否则继续按照步骤 2 的规则从右往左截取，最终表示根据 “Doc.user.dep.uuid” 的值进行查询。

4:可能会存在一种特殊情况，比如 Doc包含一个 user 的属性，也有一个 userDep 属性，此时会存在混淆。可以明确在属性之间加上 "_" 以显式表达意图，比如 "findByUser_DepUuid()" 或者 "findByUserDep_uuid()"

JpaRepository的查询功能-4

n 特殊的参数： 还可以直接在方法的参数上加入分页或排序的参数，比如：

```
Page<UserModel> findByName(String name, Pageable pageable);
List<UserModel> findByName(String name, Sort sort);
```

n 也可以使用JPA的NamedQueries，方法如下：

1：在实体类上使用@NamedQuery，示例如下：

```
@NamedQuery(name = "UserModel.findByAge",query = "select o from UserModel
    o where o.age >= ?1")
```

2：在自己实现的DAO的Repository接口里面定义一个同名的方法，示例如下：

```
public List<UserModel> findByAge(int age);
```

3：然后就可以使用了，Spring会先找是否有同名的NamedQuery，如果有，那么就不会按照接口定义的方法来解析。

使用@Query-1

n 可以在自定义的查询方法上使用@Query来指定该方法要执行的查询语句，比如：

```
@Query("select o from UserModel o where o.uuid=?1")
public List<UserModel> findByUuidOrAge(int uuid);
```

注意：

1：方法的参数个数必须和@Query里面需要的参数个数一致

2：如果是like，后面的参数需要前面或者后面加“%”，比如下面都对：

```
@Query("select o from UserModel o where o.name like ?1%")
public List<UserModel> findByUuidOrAge(String name);

@Query("select o from UserModel o where o.name like %?1")
public List<UserModel> findByUuidOrAge(String name);

@Query("select o from UserModel o where o.name like %?1%")
public List<UserModel> findByUuidOrAge(String name);
```

当然，这样在传递参数值的时候就可以不加‘%’了，当然加了也不会错

使用@Query-2

- 还可以使用@Query来指定本地查询，只要设置nativeQuery为true，比如：

```
@Query(value="select * from tbl_user where name like %?1" ,nativeQuery=true)
public List<UserModel> findByUuidOrAge(String name);
```

注意：当前版本的本地查询不支持翻页和动态的排序

- 使用命名化参数，使用@Param即可，比如：

```
@Query(value="select o from UserModel o where o.name like %:nn")
public List<UserModel> findByUuidOrAge(@Param("nn") String name);
```

- 同样支持更新类的Query语句，添加@Modifying即可，比如：

```
@Modifying
@Query(value="update UserModel o set o.name=:newName where o.name like %:nn")
public int findByUuidOrAge(@Param("nn") String name,@Param("newName") String
    newName);
```

注意：

1：方法的返回值应该是int，表示更新语句所影响的行数

2：在调用的地方必须加事务，没有事务不能正常执行

JpaRepository的查询功能-5

- 创建查询的顺序

Spring Data JPA 在为接口创建代理对象时，如果发现同时存在多种上述情况可用，它该优先采用哪种策略呢？

<jpa:repositories> 提供了 query-lookup-strategy 属性，用以指定查找的顺序。它有如下三个取值：

1：create-if-not-found：如果方法通过@Query指定了查询语句，则使用该语句实现查询；如果没有，则查找是否定义了符合条件的命名查询，如果找到，则使用该命名查询；如果两者都没有找到，则通过解析方法名字来创建查询。这是 query-lookup-strategy 属性的默认值

2：create：通过解析方法名字来创建查询。即使有符合的命名查询，或者方法通过@Query指定的查询语句，都将会被忽略

3：use-declared-query：如果方法通过@Query指定了查询语句，则使用该语句实现查询；如果没有，则查找是否定义了符合条件的命名查询，如果找到，则使用该命名查询；如果两者都没有找到，则抛出异常

第四章：客户化扩展JpaRepository

客户化扩展JpaRepository

- 如果你不想暴露那么多的方法，可以自己订制自己的Repository，还可以在自己的Repository里面添加自己使用的公共方法
- 当然更灵活的是自己写一个实现类，来实现自己需要的方法

1：写一个与接口同名的类，加上后缀为Impl，这个在前面xml里面配置过，可以自动被扫描到。这个类不需要实现任何接口。

2：在接口中加入自己需要的方法，比如：

```
public Page<Object[]> getByCondition(UserQueryModel u);
```

3：在实现类中，去实现这个方法就好了，会被自动找到

```
public class UserRepositoryImpl {
    @PersistenceContext
    private EntityManager em;
    public Page<Object[]> getByCondition(UserQueryModel u){
String hql = "select o.uuid,o.name from UserModel o where 1=1 and o.uuid=:uuid";
          Query q = em.createQuery(hql);
          q.setParameter("uuid", u.getUuid());
          q.setFirstResult(0);
          q.setMaxResults(1);
Page<Object[]> page = new PageImpl<Object[]>(q.getResultList(),new PageRequest(0,1),3);
          return page;
}}
```

第五章：Specifications查询

Specifications基础

- Spring Data JPA支持JPA2.0的Criteria查询，相应的接口是JpaSpecificationExecutor。

- Criteria 查询：是一种类型安全和更面向对象的查询

- 这个接口基本是围绕着Specification接口来定义的， Specification接口中只定义了如下一个方法：

```
Predicate toPredicate(Root<T> root, CriteriaQuery<?> query,
CriteriaBuilder cb);
```

要理解这个方法，以及正确的使用它，就需要对JPA2.0的Criteria查询有一个足够的熟悉和理解，因为这个方法的参数和返回值都是JPA标准里面定义的对象。

Criteria查询基本概念

- Criteria 查询是以元模型的概念为基础的，元模型是为具体持久化单元的受管实体定义的，这些实体可以是实体类，嵌入类或者映射的父类。
- CriteriaQuery接口：代表一个specific的顶层查询对象，它包含着查询的各个部分，比如：select 、from、where、group by、order by等
  注意：CriteriaQuery对象只对实体类型或嵌入式类型的Criteria查询起作用
- Root接口：代表Criteria查询的根对象，Criteria查询的查询根定义了实体类型，能为将来导航获得想要的结果，它与SQL查询中的FROM子句类似

1：Root实例是类型化的，且定义了查询的FROM子句中能够出现的类型。
2：查询根实例能通过传入一个实体类型给 AbstractQuery.from方法获得。
3：Criteria查询，可以有多个查询根。
4：AbstractQuery是CriteriaQuery 接口的父类，它提供得到查询根的方法。

- CriteriaBuilder接口：用来构建CritiaQuery的构建器对象
- Predicate：一个简单或复杂的谓词类型，其实就相当于条件或者是条件组合。

Criteria查询-1

- 基本对象的构建

1：通过EntityManager的getCriteriaBuilder或EntityManagerFactory的getCriteriaBuilder方法可以得到CriteriaBuilder对象

2：通过调用CriteriaBuilder的createQuery或createTupleQuery方法可以获得CriteriaQuery的实例

3：通过调用CriteriaQuery的from方法可以获得Root实例

- 过滤条件

1：过滤条件会被应用到SQL语句的FROM子句中。在criteria 查询中，查询条件通过Predicate或Expression实例应用到CriteriaQuery对象上。

2：这些条件使用 CriteriaQuery .where 方法应用到CriteriaQuery 对象上

3：CriteriaBuilder也作为Predicate实例的工厂，通过调用CriteriaBuilder 的条件方法（ equal，notEqual， gt， ge，lt， le，between，like等）创建Predicate对象。

4：复合的Predicate 语句可以使用CriteriaBuilder的and，or andnot 方法构建。

Criteria查询-2

- 构建简单的Predicate示例：

```
Predicate p1=cb.like(root.get(“name”).as(String.class),
    “%”+uqm.getName()+“%”);
Predicate p2=cb.equal(root.get("uuid").as(Integer.class), uqm.getUuid());
Predicate p3=cb.gt(root.get("age").as(Integer.class), uqm.getAge());
```

- 构建组合的Predicate示例：

```
Predicate p = cb.and(p3,cb.or(p1,p2));
```

Criteria查询-3

n 当然也可以形如前面动态拼接查询语句的方式，比如：

```
Specification<UserModel> spec = new Specification<UserModel>() {
public Predicate toPredicate(Root<UserModel> root,
        CriteriaQuery<?> query, CriteriaBuilder cb) {
    List<Predicate> list = new ArrayList<Predicate>();

    if(um.getName()!=null && um.getName().trim().length()>0){
        list.add(cb.like(root.get("name").as(String.class),
    "%"+um.getName()+"%"));
    }
    if(um.getUuid()>0){
        list.add(cb.equal(root.get("uuid").as(Integer.class), um.getUuid()));
    }
    Predicate[] p = new Predicate[list.size()];
    return cb.and(list.toArray(p));
}
};
```

Criteria查询-4

n 也可以使用CriteriaQuery来得到最后的Predicate，示例如下：

```
Specification<UserModel> spec = new Specification<UserModel>() {
    public Predicate toPredicate(Root<UserModel> root,
                    CriteriaQuery<?> query, CriteriaBuilder cb) {
          Predicate p1 = cb.like(root.get("name").as(String.class),
    "%"+um.getName()+"%");
         Predicate p2 = cb.equal(root.get("uuid").as(Integer.class), um.getUuid());
          Predicate p3 = cb.gt(root.get("age").as(Integer.class), um.getAge());
          //把Predicate应用到CriteriaQuery中去,因为还可以给CriteriaQuery添加其他的
    功能，比如排序、分组啥的
          query.where(cb.and(p3,cb.or(p1,p2)));
          //添加排序的功能
          query.orderBy(cb.desc(root.get("uuid").as(Integer.class)));

          return query.getRestriction();
    }
};
```

多表联接-1

- 多表连接查询稍微麻烦一些，下面演示一下常见的1:M，顺带演示一下1:1
- 使用Criteria查询实现1对多的查询

1：首先要添加一个实体对象DepModel，并设置好UserModel和它的1对多关系，如下：

```
@Entity
@Table(name="tbl_user")
public class UserModel {
    @Id
    private Integer uuid;
    private String name;
    private Integer age;
   @OneToMany(mappedBy = "um", fetch = FetchType.LAZY, cascade = {CascadeType.ALL})
    private Set<DepModel> setDep;
    //省略getter/setter
}
```

DepModel见下页

多表联接-2

```
@Entity
@Table(name="tbl_dep")
public class DepModel {
    @Id
    private Integer uuid;
    private String name;
    @ManyToOne()
    @JoinColumn(name = "user_id", nullable = false)
    //表示在tbl_dep里面有user_id的字段
    private UserModel um = new UserModel();
    //省略getter/setter
}
```

2：配置好Model及其关系后，就可以在构建Specification的时候使用了，示例如下：

多表联接-3

```
Specification<UserModel> spec = new Specification<UserModel>() {
public Predicate toPredicate(Root<UserModel> root,        CriteriaQuery<?> query,
    CriteriaBuilder cb) {
    Predicate p1 = cb.like(root.get("name").as(String.class), "%"+um.getName()+"%");
    Predicate p2 = cb.equal(root.get("uuid").as(Integer.class), um.getUuid());
    Predicate p3 = cb.gt(root.get("age").as(Integer.class), um.getAge());
    SetJoin<UserModel,DepModel> depJoin =
          root.join(root.getModel().getSet("setDep",DepModel.class) , JoinType.LEFT);

    Predicate p4 = cb.equal(depJoin.get("name").as(String.class), "ddd");
//把Predicate应用到CriteriaQuery去,因为还可以给CriteriaQuery添加其他的功能，比如排序、分
    组啥的
    query.where(cb.and(cb.and(p3,cb.or(p1,p2)),p4));
    //添加分组的功能
    query.orderBy(cb.desc(root.get("uuid").as(Integer.class)));
    return query.getRestriction();
}};
```

多表联接-4

- 接下来看看使用Criteria查询实现1:1的查询

1：在UserModel中去掉setDep的属性及其配置，然后添加如下的属性和配置：

```
@OneToOne()
@JoinColumn(name = "depUuid")
private DepModel dep;
public DepModel getDep() { return dep;}
public void setDep(DepModel dep) {this.dep = dep;  }
```

2：在DepModel中um属性上的注解配置去掉，换成如下的配置：

```
@OneToOne(mappedBy = "dep", fetch = FetchType.EAGER, cascade =
   {CascadeType.ALL})
```

3：在Specification实现中，把SetJoin的那句换成如下的语句：

```
Join<UserModel,DepModel> depJoin =
root.join(root.getModel().getSingularAttribute("dep",DepModel.class),Join
   Type.LEFT);
//root.join(“dep”,JoinType.LEFT); //这句话和上面一句的功能一样，更简单
```