Spring Data JPA

# Spring Data概述

Spring JPA通过为用户统一创建和销毁EntityManager，进行事务管理，简化JPA的配置等使用户的开发更加简便。Spring Data JPA是在Spring JPA的基础上，对持久层做了简化。用户只需声明持久层的接口，不需要实现该接口。Spring Data JPA内部会根据不同的策略、通过不同的方法创建Query操作数据库。

Spring Data JPA jar文件：

* spring-data-commons-1.8.0.RELEASE.jar：提供Spring Data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。
* spring-data-jpa-1.6.0.RELEASE.jar：简化创建 JPA 数据访问层

现在主流的JPA实装包括Eclipselink、Toplink、OpenJPA和Hibernate。

SpringFramework的JPA机能(org.springframework.orm.jpa 中提供)对这四种实装都能支持。但Spring JPA Data只支持JPA2.0，不支持JPA1.0。所以在TopLink上使用时可能会出错。

# Spring Data基本使用方法

|  |
| --- |
| Spring Data JPA 简化持久层开发大致需要如下三个步骤。  1) 声明持久层接口，该接口继承Repository <T,ID>或其子接口，T是领域实体，ID是领域实体的主键类型。例：  public interface UserRepository extends Repository<User, Long> {……}  2) 在持久层的接口中声明需要的业务方法，Spring Data JPA将会根据指定的策略  为该方法生成实现代码。用户不需要实现该接口。  例：List<User> findByLastname(String lastname);  3) 在Spring的配置文件中添加配置，为声明的接口设定代理对象。  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  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.xsd  http://www.springframework.org/schema/data/jpa  http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">    <jpa:repositories base-package="com.jpa.data.sample"  entity-manager-factory-ref="entityManagerFactory"  transaction-manager-ref="transactionManager"/>  注：  ①Spring会在" base-package"中定义的package和其子package中搜寻继承了Repository的接口。  ②entityManagerFactory和transactionManager的定义，请参照Spring JPA的使用方法。  4) 获得并使用repository的实例。  ①在Spring Container中使用  public class SomeClient {  @Autowired  private UserRepository repository;  public void doSomething() {  User u = new User();  User user = repository.save(u);  }  }  ② 在Spring Container外使用  RepositoryFactorySupport factory = … // Instantiate factory here  UserRepository repository = factory.getRepository(UserRepository.class); |

## Repository 接口

Repository是SpringData的核心接口，它并不提供任何方法，用户需要自己定义需要的方法。继承Repository接口的两种方法

|  |
| --- |
| ①直接继承  public interface UserDao extends Repository<User, Long> { …… }    ②使用@RepositoryDefinition注解  @RepositoryDefinition(domainClass = User.class, idClass = Long.class)  public interface UserDao { …… } |

#其他Repository接口

|  |  |
| --- | --- |
| CrudRepository  (Spring Data提供) | 继承Repository，提供增删改查方法，可以直接调用。 |
| PagingAndSortingRepository  (Spring Data提供) | 继承CrudRepository，增加了分页查询和排序两个方法 |
| JpaRepository  (Spring Data JPA提供) | 继承PagingAndSortingRepository，是针对JPA技术的接口，提供flush()，saveAndFlush()，deleteInBatch()等方法 |

#用户组定义实现(Spring Data提供的机能)：Spring Data的repository允许

用户自定义操作数据库的方法。用户可以与原有的repository结合起来使用。

|  |
| --- |
| #单个的repository添加用户行为  ①定义一个接口，在此接口中声明自定义的操作数据库的方法  interface UserRepositoryCustom {  public void someCustomMethod(User user);  }  ② ①中定义接口的实现  class UserRepositoryImpl implements UserRepositoryCustom {  public void someCustomMethod(User user) {  //操作数据库  }  }  ③定义一个接口同时继承Spring原有Repository和①中定义接口  public interface UserRepository extends CrudRepository < User, Long > , UserRepositoryCustom {……}  ④在Spring的配置文件中添加配置  <!--搜寻名为userRepositoryImpl的 class作为自定义repository的实现-->  <repositories base-package=" com.jpa.data.sample ">  <repository id="userRepository" />  </repositories>  <!--搜寻名为userRepositoryFooBar的 class作为自定义repository的实现-->  <repositories base-package=" com.jpa.data.sample " repository-impl-postfix="FooBar">  <repository id="userRepository" />  </repositories> |
| #所有的repository添加用户行为  ①定义一个接口，声明自定义的操作数据库的方法  public interface MyRepository < T, ID extends Serializable > extends JpaRepository < T, ID > {  void sharedCustomMethod(ID id);  }  ②①中定义接口的实现，并继承SimpleJpaRepository类  public class MyRepositoryImpl < T, ID extends Serializable >  extends SimpleJpaRepository < T, ID > implements MyRepository < T, ID > {  public void sharedCustomMethod(ID id) {  // implementation goes here  }  }  ③取代RepositoryFactoryBean成为Spring Data repositores的base class，用于生成，MyRepositoryImpl实例  public class MyRepositoryFactoryBean < T extends JpaRepository < ? , ? >  extends JpaRepositoryFactoryBean < T > {  protected RepositoryFactorySupport getRepositoryFactory(…) {  return new MyRepositoryFactory(…);  }  private static class MyRepositoryFactory extends JpaRepositoryFactory {  public MyRepositoryImpl getTargetRepository(…) {  return new MyRepositoryImpl(…);  }  public Class < ? extends RepositorySupport > getRepositoryClass() {  return MyRepositoryImpl.class;  }  }  }  ④在Spring的配置文件中添加配置  <repositories base-package=" com.jpa.data.sample "  factory-class="com.acme.MyRepositoryFactoryBean" /> |

## 创建Query的三种方式

Spring Data JPA中除了提供通过解析方法名的方式来创建Query之外，还提供了@Query和JPA NamedQueries两种方法。

### 解析方法名

Spring Data JPA会通过解析用户在持久层接口中定义的方法名来生成相应的query语句。例：持久层接口中定义如下：

|  |
| --- |
| public interface UserRepository extends Repository < User, Long > {  List < User > findByEmailAddressAndLastname(String emailAddress, String lastname);  }  #将会解析为如下的query：select u from User u where u.emailAddress = ?1 and u.lastname = ?2 |

解析时能被识别的keyword和包含这些keyword的方法会被解析成什么样的Query，如下表所示：

|  |  |  |
| --- | --- | --- |
| **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 |
| IsNull | findByAgeIsNull | where x.age is null |
| 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 |
| 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 |

### 使用@Query

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，如：

|  |
| --- |
| public interface UserRepository extends JpaRepository < User, Long > {  @Query("select u from User u where u.emailAddress = ?1")  User findByEmailAddress(String emailAddress);  }  注意：方法的参数个数必须和@Query里面需要的参数个数一致；如果是like，后面的参数需要前面或者后面加"%" |
| #使用@Param可以用命名参数来代替位置编号，将方法参数与 JPQL 中的命名参数对应。JPQL 语句中通过": 变量"的格式来指定参数。如：  public interface UserRepository extends JpaRepository < User, Long > {  @Query("select u from User u where u.firstname = :firstname or u.lastname = :lastname")  User findByLastnameOrFirstname(@Param("lastname") String lastname,  @Param("firstname") String firstname);  } |
| #更新类的Query语句，在@Query之前添加@Modifying  @Modifying  @Query("update User u set u.firstname = ?1 where u.lastname = ?2")  int setFixedFirstnameFor(String firstname, String lastname);  注意：方法的返回值应该是int，表示更新语句所影响的行数；在调用的地方必须加事务，没有事务不能正常执行。 |

### 使用JPA NamedQueries

|  |
| --- |
| #1在JPA配置文件中定义，在META-INF文件下的JPA的配置文件orm.xml中，通过<named-query/>元素进行定义。  <named-query name="User.findByLastname">  <query>select u from User u where u.lastname = ?1</query>  </named-query> |
| #2通过Annotation配置，在Entity Bean中使用@NamedQuery(或@NamedNativeQuery)进行配置。  @Entity  @NamedQuery(name = "User.findByEmailAddress",  query = "select u from User u where u.emailAddress = ?1")  public class User {} |

注意：上述两种方法都需要满足"DomainClass.methodName()"的命名规则；无论是在JPA配置文件中使用<named-query/>定义还是在Entity Bean中使用@NamedQuery进行配置，在持久层的接口中必须声明对应的方法。

|  |
| --- |
| public interface UserRepository extends JpaRepository < User, Long > {  List < User > findByLastname(String lastname);  User findByEmailAddress(String emailAddress);  } |

## 创建Query的策略

创建Query的策略有如下3种：

* create：只通过解析方法名来创建Query。忽略@Query和NamedQuery方法。
* use-declared-query：如果方法通过 @Query 指定了查询语句，则使用该语句创建Query；如果没有，则查找是否定义了符合条件的Named Query，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。
* create-if-not-found (default)：如果方法通过 @Query 指定了查询语句，则使用该语句创建查询；如果没有，则查找是否定义了符合条件的Named Query，如果找到，则使用该Named Query；如果两者都没有找到，则通过解析方法名字来创建Query。

<jpa:repositories>中提供的query-lookup-strategy 属性可以用来定义查找Query的顺序。定义方法如下：

|  |
| --- |
| <jpa:repositories base-package=" com.jpa.data.sample" query-lookup-strategy="create"/> |

# Specifications的使用

Spring Data JPA支持JPA2.0的Criteria查询，相应的接口是JpaSpe  
cificationExecutor。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。 Spring Data JPA支持JPA2.0的Criteria查询，相应的接口是JpaSpe  
cificationExecutor。Criteria 查询：**是一种类型安全和更面向对象的查询** 。

## 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

* 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。
* 查询根实例能通过传入一个实体类型给 AbstractQuery.from方法获得。
* Criteria查询，可以有多个查询根。
* 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

|  |
| --- |
| a. 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。  b. 这些条件使用 CriteriaQuery .where 方法应用到CriteriaQuery 对象上  c. CriteriaBuilder也作为Predicate实例的工厂，通过调用CriteriaBuilder 的条件方（ equal, notEqual, gt, ge, lt, le, between, like等）创建Predicate对象。  d. 复合的Predicate 语句可以使用CriteriaBuilder的and, or andnot 方法构建。 |
| #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));  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));  }  };  #也可以使用CriteriaQuery来得到最后的Predicate，  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();  }  }; |

JPA2.0提供了Criteria API(具体的适用方法请参照JPA2.0的官方文档)，可以用于动态的生成query，并且在运行时检证其正确性。Spring Data JPA支持Criteria查询。使用方法如下：

|  |
| --- |
| ① 自定义Repository接口并继承JpaSpecificationExecutor  public interface UserRepository extends CrudRepository<User, Long>, JpaSpecificationExecutor { ...}  ②自定义Specification，实现Specification接口  public Specification < User > isYoungFemaleUser() {  return new Specification < User > () {  public Predicate toPredicate(Root < User > root, CriteriaQuery < ? > query, CriteriaBuilder cb) {  Predicate isFemaleUser = cb.equal(root.get("sex").as(String.class), "girl");  Predicate isYoungUser = cb.lessThan(root.get("age").as(Integer.class), 18);  // 2个检索条件同时使用  query.where(cb.and(isFemaleUser, isYoungUser));  return query.getRestriction();  }  };  }  ③ 调用自定义的Repository中的方法(可由JpaSpecificationExecutor提供)，例如findAll  List<User> users = userRepository.findAll(isYoungFemaleUser ()); |
| @Override  public void search(final AppVersionSearchDto dto) {  PageRequest pageRequest = new PageRequest(dto.getCurrentPage() - 1, dto.getPageSize(),  new Sort(Direction.DESC, "createDate"));  Page<AppVersionEntity> page = appVersionDao.findAll(new Specification<AppVersionEntity>() {  @Override  public Predicate toPredicate(Root<AppVersionEntity> root, CriteriaQuery<?> query, CriteriaBuilder cb) {  List<Predicate> predicates = new ArrayList<Predicate>();  if (StringUtils.isNotBlank(dto.getAppType())) {  Predicate p = cb.equal(root.<String> get("appType"), dto.getAppType());  predicates.add(p);  }  if (StringUtils.isNotBlank(dto.getAppUserType())) {  Predicate p = cb.equal(root.<String> get("appUserType"), dto.getAppUserType());  predicates.add(p);  }  Predicate predicate = cb.and(predicates.toArray(new Predicate[predicates.size()]));  query.where(predicate);  return query.getRestriction();  }  }, pageRequest);  dto.setTotalPages(page.getTotalPages());  dto.setTotalRecord(page.getTotalElements());  dto.setList(bulidAppVersionDto(page.getContent()));  } |