Spring Data JPA中文文档[1.4.3]

被 孙 瑞鸿添加，被 孙 瑞鸿最后更新于十二月 17, 2013

前言

反正也没人看，省略吧！

本文档对应的是Spring Data JPA 1.4.3 RELEASE

作者 & 译者

作者：Oliver Gierke, Thomas Darimont

译者：大熊 QQ：304853988

由于本人利用闲暇时间翻译，再加上本人水平有限，翻译可能过于粗糙，未能翻译出Spring Data JPA原文档的意思，请各位谅解，如果有什么问题，可以联系本人！

本翻译文档仍未做任何校对(PS:这是翻译第一版，先出炉)，请大家多多包含！

最后，请大家尊重本人的劳动成果，本译文可用户私人或者拷贝予他人免费使用，但不允许用于任何商业用途。

第一部分：文档

1.使用 Spring Data Repositories

Spring Data Repository的存在，是为了把你从大量重复、繁杂的数据库层操作中解放出来。

1.1核心概念

Spring Data Repository的核心接口是Repository(好像也没什么好惊讶的)。这个接口需要领域类(Domain Class)跟领域类的ID类型作为参数。这个接口主要是让你能知道继承这个类的接口的类型。CrudRepository提供了对被管理的实体类的一些常用CRUD方法。

例**1.1 CrudRepository**接口

```
public interface CrudRepository<T, ID extends Serializable>
    extends Repository<T, ID> {

    <S extends T> S save(S entity);①

    T findOne(ID primaryKey);②

    Iterable<T> findAll();③

    Long count();④

    void delete(T entity);⑤

    boolean exists(ID primaryKey);⑥

    // … 省略其他方法
}
```

① 保存给定的实体。

②返回指定ID的实体。

③返回全部实体。

④返回实体的总数。

⑤删除指定的实体。

⑥判断给定的ID是否存在。

通常我们要扩展功能的方法，那么我们就需要在接口上做子接口。那么我们要添加功能的时候，就在CrudRepository的基础上去增加。

PagingAndSortingRepository 是一个继承CrudRepository的接口，他扩展了分页与排序的功能。

例**1.2 PagingAndSortingRepository**

```
public interface PagingAndSortingRepository<T, ID extends Serializable>
  extends CrudRepository<T, ID> {

  Iterable<T> findAll(Sort sort);

  Page<T> findAll(Pageable pageable);
}
```

如果我们需要查询第二页的用户数据(每页包含20条数据)，那么我们可以简单的这么做：

用户分页查询

```
PagingAndSortingRepository<User, Long> repository = // … get access to a bean
Page<User> users = repository.findAll(new PageRequest(1, 20));
```

1.2查询方法

一般的增删改查功能都会有一些查询语句去查询数据库，在Spring Data，你只需要简单的做四个步骤即可实现！

1.声明一个继承与Repository或者它的子接口的接口，并且输入类型参数，如下：

声明接口

```
public interface PersonRepository extends Repository<User, Long> { … }
```

2.声明查询的方法在接口上

声明方法

```
List<Person> findByLastname(String lastname);
```

你没有看错，你只要声明，不需要些实现！SpringData会创建代理对象帮你完成那些繁琐的事情。

3.在Spring上配置

Spring配置

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="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">

  <repositories base-package="com.acme.repositories" />

</beans>
```

注意，上面的命名空间使用了JPA的命名空间

4.在业务中使用

调用数据操作

```
public class SomeClient {

  @Autowired
  private PersonRepository repository;

  public void doSomething() {
    List<Person> persons = repository.findByLastname("Matthews");
  }
}
```

这部分的代码将在下部分中解释。

1.2.1 声明Repository接口

在上面的第一步操作中定义了接口，这些接口必须都继承与Repository或者其子类，并且标注领域类(Domain Class)以及ID类型。如果你想暴露CRUD方法，那么你可以直接继承CrudRepository接口。

通常，我们的Repository会继承Repository, CrudRepository 或者PagingAndSortingRepository中的一个。但是你如果不想用SpringData的接口的话，你也可以把自己的接口声明@Repository即可。继承CrudRepository接口可以让你暴露出很多方法去操作你的实体类。如果你仅仅想暴露几个接口给其他人使用，那么你只需要从CrudRepository中拷贝几个需要的方法到自己的Repository中。

例**1.3** 选择性的暴露接口

```
1  interface MyBaseRepository<T, ID extends Serializable> extends Repository<T, ID> {
2    T findOne(ID id);
3    T save(T entity);
4  }
5
6  interface UserRepository extends MyBaseRepository<User, Long> {
7
8    User findByEmailAddress(EmailAddress emailAddress);
9  }
```

在这里我们只暴露出findOne(...)跟save(...)两个方法出来。对于UserRepository，他除了有根据ID查询的方法、保存实体的方法之外，还有根据Email地址查询用户的方法。

1.2.2 定义查询方法

SpringData通过方法名有两种方式去解析出用户的查询意图：一种是直接通过方法的命名规则去解析，第二种是通过Query去解析，那么当同时存在几种方式时，SpringData怎么去选择这两种方式呢？好了，SpringData有一个策略去决定到底使用哪种方式：

查询策略：

接下来我们将介绍策略的信息，你可以通过配置<repository>的query-lookup-strategy属性来决定。

CREATE

通过解析方法名字来创建查询。这个策略是删除方法中固定的前缀，然后再来解析其余的部分。

USE_DECLARED_QUERY

它会根据已经定义好的语句去查询，如果找不到，则会抛出异常信息。这个语句可以在某个注解或者方法上定义。根据给定的规范来查找可用选项，如果在方法被调用时没有找到定义的查询，那么会抛出异常。

CREATE_IF_NOT_FOUND(默认)

这个策略结合了以上两个策略。他会优先查询是否有定义好的查询语句，如果没有，就根据方法的名字去构建查询。这是一个默认策略，如果不特别指定其他策略，那么这个策略会在项目中沿用。

构建查询

查询构造器是内置在SpringData中的，他是非常强大的，这个构造器会从方法名中剔除掉类似find...By, read...By, 或者get...By的前缀，然后开始解析其余的名字。你可以在方法名中加入更多的表达式，例如你需要Distinct的约束，那么你可以在方法名中加入Distinct即可。在方法中，第一个By表示着查询语句的开始，你也可以用And或者Or来关联多个条件。

例**1.4** 通过方法名字构建查询

```
public interface PersonRepository extends Repository<User, Long> {

  List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);

  // 需要在语句中使用Distinct关键字，你需要做的是如下
  List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);
  List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);

  // 如果你需要忽略大小写，那么你要用IgnoreCase关键字，你需要做的是如下
  List<Person> findByLastnameIgnoreCase(String lastname);
  // 所有属性都忽略大小写呢？AllIgnoreCase可以帮到您
  List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);

  // 同样的，如果需要排序的话，那你需要：OrderBy
  List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
  List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
}
```

根据方法名解析的查询结果跟数据库是相关，但是，还有几个问题需要注意：

- 多个属性的查询可以通过连接操作来完成，例如And，Or。当然还有其他的，例如Between，LessThan，GreaterThan，Like。这些操作时跟数据库相关的，当然你还需要看看相关的数据库文档是否支持这些操作。
- 你可以使用IngoreCase来忽略被标记的属性的大小写，也可以使用AllIgnoreCase来忽略全部的属性，当然这个也是需要数据库支持才允许的。
- 你可以使用OrderBy来进行排序查询，排序的方向是Asc跟Desc，如果需要动态排序，请看后面的章节。

属性表达式

好了，将了那么多了，具体的方法名解析查询需要怎样的规则呢？这种方法名查询只能用在被管理的实体类上，就好像之前的案例。假设一个类Person中有个Address，并且Address还有ZipCode，那么根据ZipCode来查询这个Person需要怎么做呢？

```
List<Person> findByAddressZipCode(ZipCode zipCode);
```

在上面的例子中，我们用x.address.zipCode去检索属性，这种解析算法会在方法名中先找出实体属性的完整部分(AddressZipCode)，检查这部分是不是实体类的属性，如果解析成功，则按照驼峰式从右到左去解析属性，如：AddressZipCode将分为AddressZip跟Code，在这个时候，我们的属性解析不出Code属性，则会在此用同样的方式切割，分为Address跟ZipCode（如果第一次分割不能匹配，解析器会向左移动分割点），并继续解析。

为了避免这种解析的问题，你可以用"_"去区分，如下所示：

```
List<Person> findByAddress_ZipCode(ZipCode zipCode);
```

特殊参数处理

上面的例子已经展示了绑定简单的参数，那么除此之外，我们还可以绑定一些指定的参数，如Pageable和Sort来动态的添加分页、排序查询。

在查询方法中使用分页和排序

```
Page<User> findByLastname(String lastname, Pageable pageable);
```

```
List<User> findByLastname(String lastname, Sort sort);

List<User> findByLastname(String lastname, Pageable pageable);
```

第一个方法通过传递org.springframework.data.domain.Pageable来实现分页功能，排序也绑定在里面。如果需要排序功能，那么需要添加参数org.springframework.data.domain.Sort，如第二行中，返回的对象可以是List，当然也可以是Page类型的。

1.2.3 创建Repository实体

创建已定义的Repository接口，最简单的方式就是使用Spring配置文件，当然，需要JPA的命名空间。

XML配置

你可以使用JPA命名空间里面的repositories去自动检索路径下的repositories元素：

XML配置

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="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">

  <repositories base-package="com.acme.repositories" />

</beans:beans>
```

在本例中，Spring能够通过base-package检测出指定路径下所有继承Repository或者其子接口的接口(有点绕口)。每找到一个接口的时候，FactoryBean就会创建一个合适的代理去处理以及调用里面的查询方法。每个注册的Bean的名称都是源于接口名称，例如：UserRepository将会被注册为userRepository。base-package允许使用通配符作为扫描格式。

使用过滤器

在默认的设置中，将使用全路径扫描的方式去检索接口，当然，你在业务上可能需要更细致的操作，这时候，你可以在<repositories>中使用<include-filter>或者<exclude-filter>。这样的话，你可以指定扫描的路径包含或者不包含指定的路径。

例如我们现在想过滤掉一些指定的接口，那么你可以这么做：

例**1.6** 使用排除过滤

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

这个例子中，我们排除了所有以SomeRepository结尾的接口。

JavaConfig

你可以在JavaConfig中使用@Enable${store}Repositories注解来实现。那么代码就是如下：

例**1.7** 使用**JavaConfig**

```
@Configuration
@EnableJpaRepositories("com.acme.repositories")
class ApplicationConfiguration {

  @Bean
  public EntityManagerFactory entityManagerFactory() {
    // …
  }
}
```

独立使用

你可以不在Spring容器里面使用repository。但是你还需要Spring的依赖包在你的classpath中，你需要使用RepositoryFactory来实现，代码如下：

例**1.8** 独立模式下使用

```
RepositoryFactorySupport factory = … // 初始化
UserRepository repository = factory.getRepository(UserRepository.class);
```

1.3 自定义Repository实现

我们可以自己实现repository的方法。

1.3.1 在repository中添加自定义方法

为了丰富我们的接口我们通常会自定义自己的接口以及对应的实现类。

例1.9 自定义接口

```
interface UserRepositoryCustom {

  public void someCustomMethod(User user);
}
```

自定义接口的实现类

```
class UserRepositoryImpl implements UserRepositoryCustom {

  public void someCustomMethod(User user) {
    // 实现
  }
}
```

扩展CRUDRepository

```
public interface UserRepository extends CrudRepository<User, Long>, UserRepositoryCustom {

  // 声明查询方法
}
```

这样的话，就能够在常用的Repository中实现自己的方法。

配置

在XML的配置里面，框架会自动搜索base-package里面的实现类，这些实现类的后缀必须满足repository-impl-postfix中指定的命名规则，默认的规则是：Impl

例1.12 配置实例

```
<repositories base-package="com.acme.repository" />

<repositories base-package="com.acme.repository" repository-impl-postfix="FooBar" />
```

第一个配置我们将找到com.acme.repository.UserRepositoryImpl，而第二个配置我们将找到com.acme.repository.UserRepositoryFooBar。

人工装载

前面的代码中，我们使用了注释以及配置去自动装载。如果你自己定义的实现类需要特殊的装载，那么你可以跟普通bean一样声明出来就可以了，框架会手工的装载起来，而不是创建本身。

例1.13 人工装载实现类(I)

```
<repositories base-package="com.acme.repository" />

<beans:bean id="userRepositoryImpl" class="…">
  <!-- 其他配置 -->
</beans:bean>
```

1.3.2 为所有的repository添加自定义方法

假如你要为所有的repository添加一个方法，那么前面的方法都不可行。你可以这样做：

1. 你需要先声明一个中间接口，然后让你的接口来继承这个中间接口而不是Repository接口，代码如下：

   例1.14 中间接口

   ```
   public interface MyRepository<T, ID extends Serializable>
     extends JpaRepository<T, ID> {

     void sharedCustomMethod(ID id);
   }
   ```

2. 这时候，我们需要创建我们的实现类，这个实现类是基于Repository中的基类的，这个类会作为Repository代理的自定义类来执行。

   例1.15 自定义基类

   ```
   public class MyRepositoryImpl<T, ID extends Serializable>
     extends SimpleJpaRepository<T, ID> implements MyRepository<T, ID> {

     private EntityManager entityManager;

     // 可以选择两个构造函数中的一个
   ```

```
  public MyRepositoryImpl(Class<T> domainClass, EntityManager entityManager) {
    super(domainClass, entityManager);

    // This is the recommended method for accessing inherited class dependencies.
    this.entityManager = entityManager;
  }

  public void sharedCustomMethod(ID id) {
    // implementation goes here
  }
}
```

3. 我们需要创建一个自定义的FactoryBean去替代默认的工厂类。代码如下：

例**1.16** 自定义工厂类

```
public class MyRepositoryFactoryBean<R extends JpaRepository<T, I>, T, I extends Serializable>
  extends JpaRepositoryFactoryBean<R, T, I> {

  protected RepositoryFactorySupport createRepositoryFactory(EntityManager entityManager) {

    return new MyRepositoryFactory(entityManager);
  }

  private static class MyRepositoryFactory<T, I extends Serializable> extends JpaRepositoryFactory {

    private EntityManager entityManager;

    public MyRepositoryFactory(EntityManager entityManager) {
      super(entityManager);

      this.entityManager = entityManager;
    }

    protected Object getTargetRepository(RepositoryMetadata metadata) {

      return new MyRepositoryImpl<T, I>((Class<T>) metadata.getDomainClass(), entityManager);
    }

    protected Class<?> getRepositoryBaseClass(RepositoryMetadata metadata) {

      // The RepositoryMetadata can be safely ignored, it is used by the JpaRepositoryFactory
      //to check for QueryDslJpaRepository's which is out of scope.
      return MyRepository.class;
    }
  }
}
```

4. 最后，在XML中配置factory-class即可：

例**1.17** 配置

```
<repositories base-package="com.acme.repository"
  factory-class="com.acme.MyRepositoryFactoryBean" />
```

1.4 Spring Data扩展

这部分我们将会把SpringData扩展到其他框架中，目前我们继承的目标是SpringMVC。

1.4.1 Web支持

SpringData支持很多web功能。当然你的应用也要有SpringMVC的Jar包，有的还需要继承Spring HATEOAS。

通常来说，你可以在你的JavaConfig配置类中加入@EnableSpringDataWebSupport即可：

例**1.18** 启用**web**支持

```
@Configuration
@EnableWebMvc
@EnableSpringDataWebSupport
class WebConfiguration { }
```

这个注解注册了几个功能，我们稍后会说，他也能检测Spring HATEOAS，并且注册他们。

如果你用XML配置的话，那么你可以用下面的配置：

例**1.19** 在**XML**中配置

```
<bean class="org.springframework.data.web.config.SpringDataWebConfiguration" />
```

```
<!-- If you're using Spring HATEOAS as well register this one *instead* of the former -->
<bean class="org.springframework.data.web.config.HateoasAwareSpringDataWebConfiguration" />
```

基本的**web**支持

上面的配置注册了一下的几个功能：

- DomainClassConverter将会让SpringMVC能从请求参数或者路径参数中解析出来。
- HandlerMethodArgumentResolver 能让SpringMVC从请求参数中解析出Pageable(分页)与Sort(排序)。

DomainClassConverter

这个类允许你在SpringMVC控制层的方法中直接使用你的领域类型(Domain types)，如下：

例**1.20** 使用领域类型

```
@Controller
@RequestMapping("/users")
public class UserController {

  @RequestMapping("/{id}")
  public String showUserForm(@PathVariable("id") User user, Model model) {

    model.addAttribute("user", user);
    return "userForm";
  }
}
```

正如你所见，上面的方法直接接收了一个User对象，你不需要做任何的搜索操作，这个转换器自动的设id的值进去对象中，并且最终调用了findOne方法查询出实体。(注：当前的Repository必须实现CrudRepository)

HandlerMethodArgumentResolver分页排序

这个配置项同时注册了PageableHandlerMethodArgumentResolver 和 SortHandlerMethodArgumentResolver，使得Pageable跟Sort能作为控制层的参数使用：

使用分页作为控制层参数

```
@Controller
@RequestMapping("/users")
public class UserController {

  @Autowired UserRepository repository;

  @RequestMapping
  public String showUsers(Model model, Pageable pageable) {

    model.addAttribute("users", repository.findAll(pageable));
    return "users";
  }
}
```

这个配置会让SpringMVC传递一个Pageable实体参数,下面是默认的参数：

page	你要获取的页数
size	一页中最大的数据量
sort	需要被排序的属性(格式：属性**1**，属性**2(, ASC | DESC)**)，默认是**asc**，使用多个字段排序，你可以使用**sort=first&sort=last,asc**

如果你需要对多个表写多个分页或排序，那么你需要用@Qualifier来区分，请求参数的前缀是${qualifire}_，那么你的方法可能变成这样：

多个分页

```
public String showUsers(Model model,
      @Qualifier("foo") Pageable first,
      @Qualifier("bar") Pageable second) { … }
```

你需要填写foo_page和bar_page等。

默认的Pageable相当于new PageRequest(0,20)，你可以用@PageableDefaults注解来放在Pageable上。

超媒体分页

Spring HATEOAS有一个PagedResources类，他丰富了Page实体以及一些让用户更容易导航到资源的请求方式。Page转换到PagedResources是由一个实现了Spring HATEOAS ResourceAssembler接口的实现类：PagedResourcesAssembler提供转换的。

使用**PagedResourcesAssembler**作为参数

```
@Controller
class PersonController {

  @Autowired PersonRepository repository;

  @RequestMapping(value = "/persons", method = RequestMethod.GET)
```

```
  HttpEntity<PagedResources<Person>> persons(Pageable pageable,
    PagedResourcesAssembler assembler) {

    Page<Person> persons = repository.findAll(pageable);
    return new ResponseEntity<>(assembler.toResources(persons), HttpStatus.OK);
  }
}
```

上面的toResources方法会执行以下的几个步骤：

- Page对象的内容会转换成为PagedResources对象。
- PagedResources会的到一个PageMetadata的实体附加，包含Page跟PageRequest。
- PagedResources会根据状态得到prev跟next链接，这些链接指向URI所匹配的方法中。分页参数会根据PageableHandlerMethodArgumentResolver配置，以让其在后面的方法中解析使用。

假设我们现在有一个30个人的信息在数据库中，你现在可以触发一个GET请求http://localhost:8080/persons 然后你就会得到类似于如下的数据：

```
{ "links" : [ { "rel" : "next",
                "href" : "http://localhost:8080/persons?page=1&size=20 }
  ],
  "content" : [
     … // 20 Person instances rendered here
  ],
  "pageMetadata" : {
    "size" : 20,
    "totalElements" : 30,
    "totalPages" : 2,
    "number" : 0
  }
}
```

1.4.2 Repository填充

如果你用过Spring JDBC，那么你肯定很熟悉使用SQL去填写数据源(DataSource)，在这里，我们可以使用XML或者Json去填写数据，而不再使用SQL填充。

假如你有一个data.json的文件，如下：

Json数据

```
[ { "_class" : "com.acme.Person",
 "firstname" : "Dave",
  "lastname" : "Matthews" },
  { "_class" : "com.acme.Person",
 "firstname" : "Carter",
  "lastname" : "Beauford" } ]
```

要PersonRepository填充这些数据进去，你需要做如下的声明：

例**1.24** 声明**jackson repository**填充

```
<?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:repository="http://www.springframework.org/schema/data/repository"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/repository
    http://www.springframework.org/schema/data/repository/spring-repository.xsd">

  <repository:jackson-populator location="classpath:data.json" />

</beans>
```

这个声明使得data.json能够通过Jackson ObjectMapper被其他地方读取，反序列化。

如果你要用XML的话，你需要使用unmarshaller-populator，你可以使用Spring OXM提供的组件：

使用**JAXB**

```
<?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:repository="http://www.springframework.org/schema/data/repository"
  xmlns:oxm="http://www.springframework.org/schema/oxm"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/repository
    http://www.springframework.org/schema/data/repository/spring-repository.xsd
    http://www.springframework.org/schema/oxm
    http://www.springframework.org/schema/oxm/spring-oxm.xsd">
```

```
  <repository:unmarshaller-populator location="classpath:data.json" unmarshaller-ref="unmarshaller" />

  <oxm:jaxb2-marshaller contextPath="com.acme" />

</beans>
```

1.4.3 Legacy Web Support

在SpringMVC中绑定领域类(Domain class)

你在开发web项目的时候，你经常需要从URL或者请求参数中解析领域类中的ID，你可能是这么做得：

```
@Controller
@RequestMapping("/users")
public class UserController {

  private final UserRepository userRepository;

  @Autowired
  public UserController(UserRepository userRepository) {
    Assert.notNull(repository, "Repository must not be null!");
    userRepository = userRepository;
  }

  @RequestMapping("/{id}")
  public String showUserForm(@PathVariable("id") Long id, Model model) {

    // Do null check for id
    User user = userRepository.findOne(id);
    // Do null check for user

    model.addAttribute("user", user);
    return "user";
  }
}
```

首先你要注入一个UserRepository ，然后通过findOne查询出结果。幸运的是，Spring提供了自定义组件允许你从String类型到任意类型的转换。

属性编辑器

在Spring3.0之前，Java的PropertyEditor已经被使用。现在我们要集成它，SpringData提供了一个DomainClassPropertyEditorRegistrar类，他能在ApplicationContext中查找SpringData的Repositories，并且注册自定义的PropertyEditor。

```
<bean class="….web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">
  <property name="webBindingInitializer">
    <bean class="….web.bind.support.ConfigurableWebBindingInitializer">
      <property name="propertyEditorRegistrars">
        <bean class="org.springframework.data.repository.support.DomainClassPropertyEditorRegistrar" />
      </property>
    </bean>
  </property>
</bean>
```

如果你做了上面的工作，那么你在前面的例子中，会大大减少工作量：

```
@Controller
@RequestMapping("/users")
public class UserController {

  @RequestMapping("/{id}")
  public String showUserForm(@PathVariable("id") User user, Model model) {

    model.addAttribute("user", user);
    return "userForm";
  }
}
```

转换服务

在Spring3以后，PropertyEditor已经被转换服务取代了，SpringData现在用DomainClassConverter模仿

DomainClassPropertyEditorRegistrar中的实现。你可以使用如下的配置：

```
<mvc:annotation-driven conversion-service="conversionService" />

<bean class="org.springframework.data.repository.support.DomainClassConverter">
  <constructor-arg ref="conversionService" />
```

```
</bean>
```

如果你是用JavaConfig，你可以集成SpringMVC的WebMvcConfigurationSupport并且处理FormatingConversionService，那么你可以这么做：

```
class WebConfiguration extends WebMvcConfigurationSupport {

  // 省略其他配置

  @Bean
  public DomainClassConverter<?> domainClassConverter() {
    return new DomainClassConverter<FormattingConversionService>(mvcConversionService());
  }
}
```

Web分页

当你在写页面分页的时候，你需要填写大量的重复的代码区完成这个功能，可是现在你只需要传递一个HttpServletRequest就可以完成了，下面的例子省略了让你程序更加茁壮的异常处理代码：

```
@Controller
@RequestMapping("/users")
public class UserController {

  // DI code omitted

  @RequestMapping
  public String showUsers(Model model, HttpServletRequest request) {

    int page = Integer.parseInt(request.getParameter("page"));
    int pageSize = Integer.parseInt(request.getParameter("pageSize"));

    Pageable pageable = new PageRequest(page, pageSize);

    model.addAttribute("users", userService.getUsers(pageable));
    return "users";
  }
}
```

现在我觉得上面的配置还是麻烦了，那么SpringData使用PageableHandlerArgumentResolver帮你简化你的工作，你可以在JavaConfig中做如下配置：

```
@Configuration
public class WebConfig extends WebMvcConfigurationSupport {

  @Override
  public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
    converters.add(new PageableHandlerArgumentResolver());
  }
}
```

在XML配置中：

```
<bean class="….web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">
  <property name="customArgumentResolvers">
    <list>
      <bean class="org.springframework.data.web.PageableHandlerArgumentResolver" />
    </list>
  </property>
</bean>
```

那你现在的工作可以简化成为：

```
@Controller
@RequestMapping("/users")
public class UserController {

  @RequestMapping
  public String showUsers(Model model, Pageable pageable) {

    model.addAttribute("users", userRepository.findAll(pageable));
    return "users";
  }
}
```

PageableHandlerArgumentResolver会自动解析请求参数中的字段：

表格**1.2** 请求参数

page	你要获取的页数
page.size	每页最大数据量
page.sort	排序
page.sort.dir	排序的方向

通用的，多个分页参数的话，你可以这么做：

```
public String showUsers(Model model,
      @Qualifier("foo") Pageable first,
      @Qualifier("bar") Pageable second) { … }
```

配置通用的默认参数

PageableArgumentResolver默认会请求第一页，10条数据。但是我们可能需要更多的数据，那么我们可以使用@PageableDefaults来设置页码以及数据量，排序，以及排序方向：

```
public String showUsers(Model model,
  @PageableDefaults(pageNumber = 0, value = 30) Pageable pageable) { … }
```

2.JPA Repositories

本章节包含了JPA Repository的具体实现。

2.1介绍

2.1.1Spring命名空间

SpringData使用了自定义的命名空间去定义repository。通常我们会使用repositories元素：

例2.1 设置命名空间

```
<?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: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.acme.repositories" />

</beans>
```

这个配置中启用了持久化异常处理，所有标志了@Repository的Bean将会被转换成为Spring的DataAccessException。

自定义命名空间属性

除了repositories，JPA命名空间还提供了其他的属性去控制：

表格2.1 JPA的其他属性

entity-manager-factory-ref	默认的话，是使用ApplicationContext中找到的EntityManagerFactory，如果有多个的时候，则需要特别指明这个属性，他将会对repositories路径中找到的类进行处理。
transaction-manager-ref	默认使用系统定义的PlatformTransactionManager，如果有多个事务管理器的话，则需特别指定。

2.1.2 基于注解的配置

SpringData JPA支持JavaConfig方式的配置：

例2.2 JavaConfig方式配置

```
1  @Configuration
2  @EnableJpaRepositories
3  @EnableTransactionManagement
4  class ApplicationConfig {
5
6    @Bean
7    public DataSource dataSource() {
8
9      EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
10     return builder.setType(EmbeddedDatabaseType.HSQL).build();
11   }
12
13   @Bean
```

```
  public EntityManagerFactory entityManagerFactory() {

    HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
    vendorAdapter.setGenerateDdl(true);

    LocalContainerEntityManagerFactoryBean factory = new LocalContainerEntityManagerFactoryBean();
    factory.setJpaVendorAdapter(vendorAdapter);
    factory.setPackagesToScan("com.acme.domain");
    factory.setDataSource(dataSource());
    factory.afterPropertiesSet();

    return factory.getObject();
  }

  @Bean
  public PlatformTransactionManager transactionManager() {

    JpaTransactionManager txManager = new JpaTransactionManager();
    txManager.setEntityManagerFactory(entityManagerFactory());
    return txManager;
  }
}
```

上面的配置中，我们设置了一个内嵌的HSQL数据库，我们也配置了EntityManagerFactory，并且使用Hibernate作为持久层。最后也定义了JPATransactionManager。最上面我们还使用了@EnableJpaRepositories注解。

2.2 持久实体

2.2.1 保存实体

保存实体，我们之前使用了CrudRepository.save(...)方法。他会使用相关的JPA EntityManager来调用persist或者merge，如果数据没存在于数据库中，则调用entityManager.persist(..)，否则调用entityManager.merge(...)。

实体状态监测策略

SpringData JPA提供三种策略去监测实体是否存在：

表格**2.2** 监测方式

ID属性检测(默认)	默认的会通过ID来监测是否新数据，如果ID属性是空的，则认为是新数据，反则认为旧数据
实现Persistable	如果实体实现了Persistable接口，那么就会通过isNew的方法来监测。详细看JavaDoc
实现EntityInfomation	这个是很少用的，详细请查看JavaDoc

2.3 查询方法

2.3.1 查询策略

你可以写一个语句或者从方法名中查询。

声明查询语句

虽然说方法名查询的方式很方便，可是你可能会遇到方法名查询规则不支持你所要查询的关键字或者方法名写的很长，不方便，或者很丑陋。那么你就需要通过命名查询或者在方法上使用@Query来解决。

2.3.2 查询创建器

通常我们可以使用方法名来解析查询语句，例如：

例**2.3** 方法名查询

```
public interface UserRepository extends Repository<User, Long> {

  List<User> findByEmailAddressAndLastname(String emailAddress, String lastname);
}
```

这个查询方法相当于如下的语句：

```
select u from User u where u.emailAddress = ?1 and u.lastname = ?2
```

以下表格中展示出方法名中支持的关键字：

表格**2.3** 支持的关键字

关键字	例子	**JPQL** 片段
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
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

2.3.3 使用JPA命名查询

XML命名查询定义

在XML中配置的话，需要在JPA在META-INF目录中的配置文件orm.xml中使用<named-query />。定义如下：

例**2.4 XML**命名查询配置

```
<named-query name="User.findByLastname">
  <query>select u from User u where u.lastname = ?1</query>
</named-query>
```

上面的Query定义了名称，那么将在运行时解析使用。

注解方式

使用注解方式，你不再需要使用配置文件，减少你的开发成本，接下来我们用注解的方式来做：

例**2.5** 基于注解的命名查询

```
@Entity
@NamedQuery(name = "User.findByEmailAddress",
  query = "select u from User u where u.emailAddress = ?1")
public class User {

}
```

声明接口

要使用上面的命名查询，我们的接口需要这么声明：

例**2.6** 声明接口

```
public interface UserRepository extends JpaRepository<User, Long> {

  List<User> findByLastname(String lastname);

  User findByEmailAddress(String emailAddress);
}
```

SpringData会先从域类中查询配置，根据".(原点)"区分方法名，而不会使用自动方法名解析的方式去创建查询。

2.3.4 使用@Query

命名查询适合用于小数量的查询，我们可以使用@Query来替代：

例**2.7** 使用**@Query**查询

```
public interface UserRepository extends JpaRepository<User, Long> {

  @Query("select u from User u where u.emailAddress = ?1")
  User findByEmailAddress(String emailAddress);
```

```
}
```

LIKE查询

在表达式中使用Like查询，例子如下：

@Query中使用**LIKE**

```
public interface UserRepository extends JpaRepository<User, Long> {

  @Query("select u from User u where u.firstname like %?1")
  List<User> findByFirstnameEndsWith(String firstname);
}
```

这个例子中，我们使用了%，当然，你的参数就没必要加入这个符号了。

原生查询

我们可以在@Query中使用本地查询，当然，你需要设置nativeQuery=true，必须说明的是，这样的话，就不再支持分页以及排序。

例**2.9** 声明本地查询

```
public interface UserRepository extends JpaRepository<User, Long> {

  @Query(value = "SELECT * FROM USERS WHERE EMAIL_ADDRESS = ?0", nativeQuery = true)
  User findByEmailAddress(String emailAddress);
}
```

2.3.5 使用命名参数

使用命名查询，我们需要用到@Param来注释到指定的参数上，如下：

例**2.10** 使用命名参数

```
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);
}
```

2.3.6 使用**SpELI**表达式

在Spring Data JPA 1.4以后，我们支持在@Query中使用SpELI表达式来接收变量。

表格**2.4** 在**SpELI**中支持的变量

变量	用途	描述
entityName	select x from #{#entityName} x	根据给定的Repository自动插入相关的entityName。有两种方式能被解析出来：如果域类型定义了@Entity属性名称。或者直接使用类名称。

以下的例子中，我们在查询语句中插入表达式(你也可以用@Entity(name = "MyUser")。

例**2.11** 使用**SpELI**表达式

```
1  @Entity
2  public class User {
3
4    @Id @GeneratedValue Long id;
5    String lastname;
6  }
7
8  public interface UserRepository extends JpaRepository<User,Long> {
9
10   @Query("select u from #{#entityName} u where u.lastname = ?1")
11   List<User> findByLastname(String lastname);
12 }
```

如果你想写一个通用的Repository接口，那么你也可以用这个表达式来处理：

例**2.12** 使用**SpELI**表达式

```
1  @MappedSuperclass
2  public abstract class AbstractMappedType {
3    …
4    String attribute
5  }
6
7  @Entity
```

```
public class ConcreteType extends AbstractMappedType { … }

@NoRepositoryBean
public interface MappedTypeRepository<T extends AbstractMappedType>
  extends Repository<T, Long> {

  @Query("select t from #{#entityName} t where t.attribute = ?1")
  List<T> findAllByAttribute(String attribute);
}

public interface ConcreteRepository
  extends MappedTypeRepository<ConcreteType> { … }
```

2.3.7 修改语句

之前我们演示了如何去声明查询语句，当然我们还有修改语句。修改语句的实现，我们只需要在加多一个注解@Modify：

例2.13 声明修改语句

```
@Modifying
@Query("update User u set u.firstname = ?1 where u.lastname = ?2")
int setFixedFirstnameFor(String firstname, String lastname);
```

这样一来，我们就使用了update操作来代替select操作。当我们发起update操作后，可能会有一些过期的数据产生，我们不需要自动去清除它们，因为EntityManager会有效的丢掉那些未提交的变更，如果你想EntityManager自动清除，那么你可以在@Modify上添加clearAutomatically属性(true)：

2.3.8 使用QueryHints

你可以使用@QueryHints来启用：

例2.14 使用QueryHints

```
public interface UserRepository extends Repository<User, Long> {

  @QueryHints(value = { @QueryHint(name = "name", value = "value")},
              forCounting = false)
  Page<User> findByLastname(String lastname, Pageable pageable);
}
```

2.4 Specifications

JPA2 引入了criteria API 去建立查询，Spring Data JPA使用Specifications来实现这个API。在Repository中，你需要继承JpaSpecificationExecutor：

```
public interface CustomerRepository extends CrudRepository<Customer, Long>, JpaSpecificationExecutor {
 …
}
```

下面先给个例子，演示如何利用findAll方法返回所有符合条件的对象：

```
List<T> findAll(Specification<T> spec);
```

Specification 接口定义如下：

```
public interface Specification<T> {
  Predicate toPredicate(Root<T> root, CriteriaQuery<?> query,
            CriteriaBuilder builder);
}
```

好了，那么我们如何去实现这个接口呢？代码如下：

例2.15 实现Specifications

```
public class CustomerSpecs {

  public static Specification<Customer> isLongTermCustomer() {
    return new Specification<Customer>() {
      public Predicate toPredicate(Root<Customer> root, CriteriaQuery<?> query,
           CriteriaBuilder builder) {

        LocalDate date = new LocalDate().minusYears(2);
        return builder.lessThan(root.get(Customer_.createdAt), date);
      }
    };
  }

  public static Specification<Customer> hasSalesOfMoreThan(MontaryAmount value) {
```

```
    return new Specification<Customer>() {
      public Predicate toPredicate(Root<T> root, CriteriaQuery<?> query,
            CriteriaBuilder builder) {

         // build query here
      }
    };
  }
}
```

接下来，我们如何去调用这个方法呢？

例**2.16** 调用

```
List<Customer> customers = customerRepository.findAll(isLongTermCustomer());
```

好了，那如果有多个需要结合的话，我们可以这么做：

例**2.17** 多个**Specifications**

```
MonetaryAmount amount = new MonetaryAmount(200.0, Currencies.DOLLAR);
List<Customer> customers = customerRepository.findAll(
  where(isLongTermCustomer()).or(hasSalesOfMoreThan(amount)));
```

2.5 事务

默认的CRUD操作在Repository里面都是事务性的。如果是只读操作，只需要设置事务readOnly为true，其他的操作则配置为@Transaction。如果你想修改一个Repository的事务性，你只需要在子接口中重写并且修改他的事务：

例**2.18** 自定义事务

```
public interface UserRepository extends JpaRepository<User, Long> {

  @Override
  @Transactional(timeout = 10)
  public List<User> findAll();

  // 其他方法
}
```

这样会让findAll方法在10秒内执行否则会超时的非只读事务中。

另一种修改事务行为的方式在于使用门面或者服务层中，他们包含了多个repository。

例**2.19** 使用门面模式去定义事务

```
@Service
class UserManagementImpl implements UserManagement {

  private final UserRepository userRepository;
  private final RoleRepository roleRepository;

  @Autowired
  public UserManagementImpl(UserRepository userRepository,
    RoleRepository roleRepository) {
    this.userRepository = userRepository;
    this.roleRepository = roleRepository;
  }

  @Transactional
  public void addRoleToAllUsers(String roleName) {

    Role role = roleRepository.findByName(roleName);

    for (User user : userRepository.findAll()) {
      user.addRole(role);
      userRepository.save(user);
    }
}
```

这将会导致在调用addRoleToAllUsers方法的时候，创建一个或者加入一个事务中去。实际在Repository里面定义的事务将会被忽略，而外部定义的事务将会被应用。当然，要使用事务，你需要声明<tx:annotation-driven />（这个例子中，假设你已经使用了component-scan）。

2.5.1 事务性查询方法

要让方法在事务中，最简单的方式就是使用@Transactional注解：

例**2.20** 使用**@Transactional**

```
@Transactional(readOnly = true)
```

```
public interface UserRepository extends JpaRepository<User, Long> {

  List<User> findByLastname(String lastname);

  @Modifying
  @Transactional
  @Query("delete from User u where u.active = false")
  void deleteInactiveUsers();
}
```

一般的查询操作，你需要设置readOnly=true。在deleteInactiveUsers方法中，我们添加了Modifying注解以及覆盖了Transactional，这样导致了readOnly=false，在这个方法的执行上。

2.6 锁

想要为方法指定锁的类型，你只需要使用@Lock：

例**2.21** 声明锁

```
interface UserRepository extends Repository<User, Long> {

  // Plain query method
  @Lock(LockModeType.READ)
  List<User> findByLastname(String lastname);
}
```

当然你也可以覆盖原有的方法：

例**2.22** 覆盖原有的**CRUD**

```
interface UserRepository extends Repository<User, Long> {

  // Redeclaration of a CRUD method
  @Lock(LockModeType.READ);
  List<User> findAll();
}
```

2.7 审计

2.7.1 基础知识

SpringData为您跟踪谁创建或者修改数据，以及相应的时间提供了复杂的支持。你现在想要这些支持的话，仅仅需要使用几个注解或者实现接口即可。

注解方式

我们提供了@CreatedBy， @LastModifiedBy去捕获谁操作的实体，当然还有相应的时间@CreatedDate和@LastModifiedDate。

例**2.23** 审计实体

```
class Customer {

  @CreatedBy
  private User user;

  @CreatedDate
  private DateTime createdDate;

  // … 省略其他属性
}
```

正如你看到的，你可以选择性的使用这些注解。操作时间方面，你可以使用org.joda.time.DateTime或者java.util.Date或者long/Long表示。

基于接口的审计

如果你不想用注解来做审计的话，那么你可以实现Auditable接口。他暴露了审计属性的get/set方法。

如果你不想实现接口，那么你可以继承AbstractAuditable，通常来说，注解方式时更加方便的。

审计织入

如果你在用@CreatedBy或者@LastModifiedBy的时候，想织入当前的业务操作者，那你可以使用我们提供的AuditorAware<T>接口。T表示你想织入在这两个字段上的类型。

下面给出一个案例，我们将结合SpringSecurity来做：

例**2.24** 实现**AuditorAware**接口

```
class SpringSecurityAuditorAware implements AuditorAware<User> {

  public User getCurrentAuditor() {

    Authentication authentication = SecurityContextHolder.getContext().getAuthentication();

    if (authentication == null || !authentication.isAuthenticated()) {
```

```
      return null;
    }

    return ((MyUserDetails) authentication.getPrincipal()).getUser();
  }
}
```

在这个实现类中，我们使用SpringSecurity内置的Authentication来查找用户的UserDetails。

2.7.2 通用审计配置

SpringData JPA有一个实体监听器，他可以用于触发捕获审计信息。要用之前，你需要在orm.xml里面注册AuditingEntityListener，当然你还需要spring-sapects.jar引入：

例**2.25** 审计配置

```
<persistence-unit-metadata>
  <persistence-unit-defaults>
    <entity-listeners>
      <entity-listener class="….data.jpa.domain.support.AuditingEntityListener" />
    </entity-listeners>
  </persistence-unit-defaults>
</persistence-unit-metadata>
```

要启用这个审计，我们还需要在配置文件里面配置多一条：

例**2.26** 启用审计

```
<jpa:auditing auditor-aware-ref="yourAuditorAwareBean" />
```

正如你所见，你还需要提供一个实现AuditorAware接口的类：

例**2.27 AuditorAware**接口

```
public interface AuditorAware<T, ID extends Serializable> {

    T getCurrentAuditor();
}
```

通常你的系统中会有认证模块，那么你的认证模块里面最好织入这个AuditorAware方便审计。

赞同 成为第一个赞同者 无标签