官网学习：[https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#dependencies.train-names](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/" \l "dependencies.train-names)

依赖申明：

groupId：org.springframework.data

artifactId：spring-data-jpa

 Spring Framework version： 5.2.5.RELEASE or better

# 前言

Spring Data JPA为Java Persistence API（JPA）提供了repository支持.它简化了需要访问JPA数据源的应用程序的开发.

# 1.spring data jpa项目信息

* 版本控制[-https://github.com/spring-projects/spring-data-jpa](https://github.com/spring-projects/spring-data-jpa)
* Bugtracker[-](https://github.com/spring-projects/spring-data-jpa) [https:](https://jira.spring.io/browse/DATAJPA) //jira.spring.io/browse/DATAJPA
* 发布版本[-](https://github.com/spring-projects/spring-data-jpa)https: [//repo.spring.io/libs-release](https://repo.spring.io/libs-release)
* 里程碑版本[-](https://github.com/spring-projects/spring-data-jpa)https: [//repo.spring.io/libs-milestone](https://repo.spring.io/libs-milestone)
* 快照版本[-](https://github.com/spring-projects/spring-data-jpa)https: [//repo.spring.io/libs-snapshot](https://repo.spring.io/libs-snapshot)

# 2.新功能及依赖说明

## 2.1.Spring Data JPA 1.11的新增功能

Spring Data JPA 1.11添加了以下功能：

* 改进了与Hibernate 5.2的兼容性.
* 支持[Query by Example](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/" \l "query-by-example).任意匹配模式.
* 分页查询的优化.
* 支持对声明出的query方法进行exists检查.

## 2.2.Spring Data JPA 1.10的新增功能

Spring Data JPA 1.10添加了以下功能：

* 在repository查询方法中支持[Projections映射](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/" \l "projections).
* 支持[Query by Example](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/" \l "query-by-example)..
* 以下注解已启用,以构成组合注解：@EntityGraph，@Lock，@Modifying，@Query，@QueryHints，和@Procedure.
* 支持集合表达式中的Contains关键字.
* 为JSR-310和ThreeTenBP的ZoneId 提供AttributeConverter实现.
* 升级到Querydsl 4，Hibernate 5，OpenJPA 2.4和EclipseLink 2.6.1.

## 2.3.依赖配置

由于各个Spring Data模块的起始日期不同，因此大多数模块带有不同的大/小版本号.查找兼容版本的最简单方法是依据我们附带定义的兼容版本的Spring Data Release Train BOM.在Maven项目中，您将在POM 的<dependencyManagement />标签中声明此依赖关系，如下所示：

**例子1.使用Spring Data Release train BOM**

< <dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.data</groupId> <artifactId>spring-data-releasetrain</artifactId> <version>Moore-SR6</version>

<scope>import</scope> <type>pom</type> </dependency>

</dependencies>

</dependencyManagement>

当前的发行版是Moore-SR6.train名称按字母顺序升序，[此处](https://github.com/spring-projects/spring-data-commons/wiki/Release-planning)列出了当前可用的train.版本名称遵循以下格式：${name}-${release}，其中release可以是以下其中任何一项：

* BUILD-SNAPSHOT：当前快照版本
* M1，M2等：里程碑版本
* RC1，RC2等：发布候选版本
* RELEASE：GA版本
* SR1，SR2等：服务发布版本

在我们的[Spring Data examples repository](https://github.com/spring-projects/spring-data-examples/tree/master/bom)可以找到使用BOM的工作示例.有了它，您可以声明要使用的Spring Data模块，而无需在<dependencies />块中使用版本，如下所示：

**例子2.声明对Spring Data模块的依赖**

<dependencies> <dependency> <groupId>org.springframework.data</groupId> <artifactId>spring-data-jpa</artifactId> </dependency> <dependencies>

### 2.3.1 使用Spring Boot进行依赖管理

Spring Boot为您选择了Spring Data模块的最新版本.如果仍要升级到较新的版本，请将该属性配置为spring-data-releasetrain.version要使用的 [train name and iteration](https://github.com/spring-projects/spring-data-examples/tree/master/bom) .

### 2.3.2 Spring框架版本

当前版本的Spring Data模块要求使用5.2.5.RELEASE或更高版本的Spring Framework.这些模块也可以与bug修正后的较低小版本一起使用.不过强烈建议使用该版本中的最新版本.

# 3.整体认识Spring Data JPA

## 3.1 市场上的ORM框架

#### MyBatis

MybBatis本是Apache的一个开源项目iBatis,2010年这个项目由Apache SoftwareFoundation 迁移到了Google Code,并且改名为MyBati.MyBatis着力于POJO和SQL之间的映射关系，可以进行更为细致的SQL，使用起来十分的灵活，上手简单，容易掌握，所以深受开发者的喜欢，目前市场占有率最高，比较适合互联应用公司的API场景.

#### Hibernate

开放源代码的对象关系映射框架，对JDBC进行了非常轻量级的对象封装，使得Java程序员可以随心所欲地使用对象编程思维来操纵数据库，并且对象有自己的生命周期，着力对象与对象之间的关系，有自己的HQL查询语言，所以数据库的移植性很好。Hibernate是完备的ORM框架，是符合JPA规范的。Hibernate有自己的缓存机制。从上手的角度来说比较难，比较适合企业级的应用系统开发。

#### Spring Data JPA

可以理解为JPA规范的再次封装对象，底层还是使用了Hibernate的JPA技术实现，引用JPQL(Java Persistence Query Language) 查询语言，属于Spring整个生态体系的一部分，随着Spring Boot和Sping Cloud在市场上的流行，Spring Data JPA 也逐渐进入大家的视野，他们组成有机的整体，使用起来比较方便，加快了开发的效率，使得开发者不需要关心和配置更多的东西，又有很大的灵活性，市场的认可度越来越高

#### Spring Data 的子项目

Spring Data JPA 减少数据访问层的开发量  
Spring Data MongoDB 基于分布式存储的数据库MongoDB  
Spring Data Redis key-value数据库Redis

另外还有其他挺多子项目，感兴趣可以到官网获取资料：

<https://spring.io/projects/spring-data>

## 3.2 JPA介绍

JPA :Java persistence API,在Hibernate（目标：Java程序员不懂SQL也可以操作数据库）基础上发展而来，是java持久化【规范】

1.JPA是 Java Persistence AP的简称

2.JDA为 Java EE5.0平台标准的对象关系映射(FM)规范

将得到所有 Java EE服务器的支持。sun这次吸取了之E7规范怡痛失败的经历，在充分吸收现有cRM框架如 Hibernate)的基础上，得到了一个易于使用、伸编性强的oR规范。3.从目削的开发社区的反应上看

JPA受到了极大的支持和赞扬，JA作为R领域标准化整合者的目标已经实现

22.为什么需要JPA

1.这个需要从 Hibernate的起源说起， Hibernate作者oa1nKng发现以写JDBc的开发效率太低，特别在Java这种面问对象的语言在与关系型数据库之间的数据转换太了

2.于是开发出了 Hibernate,他白时的目标是，只有是ava程序员，就算不懂sL语句，也可以来操作数据库。

2.3. Hibernate和JPA的关系

Hibernate是一个开放源代码的对象关系映射(架，它对Bc进行了非第量级(相对于E这一套)的对象封禁，它将Poo(就是咱们的dman)与数据库表建立映射关系，是一个全自动的cm框架， Hibernate可以目动生成s语句，自动执行，使得7a程序员可以随心所欲的使用对象编程思维来排级跑据库

JE是sun直方提出的ava持久化规范。它为aa开发人员提供了一种对象/关系映射工具来置理Jaa应用中的关系数据。他的出现主要是力了间化现有的持久化开发工作和整合c技木，随南现在Hibernate、=pLk(现在叫Ec1F9ek)等cR果合自为营的局面

ease:

值得注意的是，JPA是在充分吸收了现有 Hibemate、 Topunk等ORM框架的基础上发展而来的，具有易于使用、伸缩性强等优点。

目标：

间化现有 Java EE和 Java Sa应用的对象持久化的开发工作

sun希望整合ORM技术，实现一统江湖。

简单一句话：JPA是持久化规范，而 Hibernate是其实现。

ease:

图：JPA的优缺点

优点与缺点都是相对的，那么讲一个东西的优缺点我们都需要一个多数对比，而咱们要来进行对比的

咱们操作数据原生的UDc

241.JDBc的优缺点

本质：处理37对除和关系型数据库表之间的换

优点操作数据库最底，性能最高(前提是你要有相应的经验，并且是一个数据库高手)

缺点：

1.使用复杂(重复代码太

2.移植数据库很麻烦，改动比较多

主键的生成方式不同(cyq1使用自增，ce使用序列

分页的q1语句也是不同(q1使用1m, oracle使用RcaN)

3.性能优化得自己处理，没有提供数据的存，需要自己实现

4面问91语句操作，不是面向对象的

JDBC 的DAO是面向SQL语句的不是面向对象，100多张表全部都要写SQL

咱们以前使用DBC最麻烦的点在什么地方呢？看下面

p页面是按属性输入，通过02后自动封装为对象，传递到层又把uer对象拆出来，变成性，所以k是面

向操作的，不是面句象(下面是伪代码

p页面

<steed rame ue

ep

suv UserC

ovate use u

gecer sete自功包装成uer对象

pub void save(user ue

sel

er cetane0 wer gete0∥又手动把ue对象分

JPA把JDBC做了封装

242PA的优缺点

本质:处理ava对象和关系型数据库表之间的转换,只是对JDBc再次做了一层封

优点:

1.程序员操作很简单,代码简单 entityManager, persist( employee)

2.直接面向持久对象操作

3.提供世界级数据存(现在几乎所有的oRM框架的存都是学的 Hibernate)

一级缓存,二级存,查询存

4.数据库移植性很强,很少的修改通过配方言搞正)

把各种数据库抽取了一个方言接口

数据库信息。

不同数据库实现一个方言接口,如果换了数据库,必须修改方言实现,驱动」ax文件,连接

缺点

1.不能干预=1语句的生成

entityManager find(User class, 1-)3

fnd万法野认查词 c user表的所有字段

目动生成

a--=e9ex0,⊥d=

se1ect让s世x0.⊥d,让e工0\_me,日出e工0,aef工mte工世x0

2.一个项目中,如果对=1语句的优化要求比较高,不适合用JFa(不过可以使用dP对原生

91的支持来解决

3如果一张来中有上亿级别拍数据量,也不适合用A,其实也不适合用1e(可以使用数据

库读写分离,分库分表方素解决

## 3.3 了解Spring Data

# 4.使用Spring Data Repository

Spring Data Repository抽象出来的目标是：显着减少实现各种数据持久化访问层所需的模板代码量.

本章要介绍Spring Data Repository的核心概念和接口.本章中的信息来自Spring Data Commons模块.它使用Java Persistence API（JPA）模块的配置和代码示例.您应该使XML中定义的命名空间声明和类型去适应特定业务模块.“ [命名空间参考](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#repositories.namespace-reference) ”（本页附录A）介绍了XML配置，所有支持repository API的Spring Data模块均支持该配置.“ repository requery [关键字](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#repository-query-keywords) ”涵盖repository抽象支持的query方法关键字.特定功能的详细信息，请参阅本文档中有关该模块的章节.

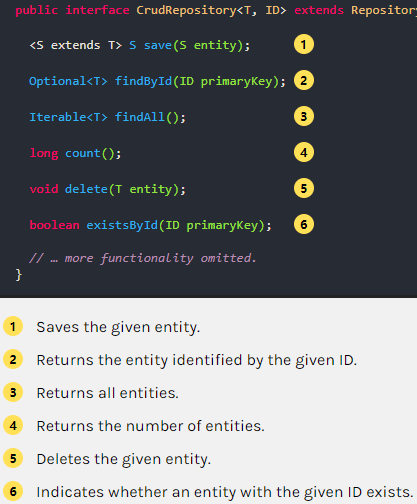
## 4.1 核心接口

* Repository中心接口

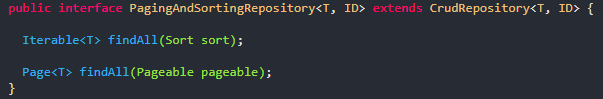
把实体类（下图中的T）以及实体类的ID类型（下图中的ID）作为类型参数来管理，该接口主要是标记接口，以捕获要使用的类型并帮助发现扩展该接口的子接口.

* CrudRepository子接口

为在Repository标记范围内的实体类提供CRUD功能.



* PagingAndSortingRepository:分页及排序

例如：访问第2页，每页20条数据：

Page<User> users = repository.findAll(PageRequest.of(1, 20));

@GetMapping*(*path="/page/{pageId:\\d+}/{sizeId:\\d+}"*)*

public @ResponseBody Page*<*User*>* getAllUsers*(*@NotNull @PathVariable*(*"pageId"*)* Integer pageId, @NotNull @PathVariable*(*"sizeId"*)* Integer sizeId*) {*

Page*<*User*>* users = userRepository.findAll*(*PageRequest.*of(*pageId, sizeId*))*;

return users;

*}*

## 4.2 query方法

以下四步为query方法的使用步骤：

* [Defining Repository Interfaces](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#repositories.definition)
* [Defining Query Methods](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#repositories.query-methods.details)
* [Creating Repository Instances](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#repositories.create-instances)
* [Custom Implementations for Spring Data Repositories](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#repositories.custom-implementations)

### 4.2.1 声明接口

public interface CustomerRepository extends CrudRepository*<*Customer, Long*> {*

### 4.2.2 在接口中声明方法

List*<*Customer*>* findByLastName*(*String lastName*)*;

### 4.2.3 创建Repository实例

增加Spring配置以支持为声明的接口生成代理实例

如果用JavaConfig配置方式则如下：

@Configuration

@EntityScan*(*"com.example.jpa"*)*

@EnableJpaRepositories*(*basePackages = "com.example.jpa"*)*

public class RepositoryConfigurations *{}*

*如果是XML配置方式则如下定义一个bean:*

**<?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 https://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/data/jpa https://www.springframework.org/schema/data/jpa/spring-jpa.xsd >**

**<jpa:repositories base-package="**com.example.jpa**"/>**

**<beans/>**

**该配置只是针对jpa,如果换成mongodb,则上述命名空间中的jpa需要替换，**@EnableJpaRepositories**也将变成**@EnableMongoRepositories

4 注入repository实例并调用实例中的query方法

@Autowired

private UserRepository userRepository;

@GetMapping*(*path = "/findUser/{name}"*)*

public @ResponseBody List*<*User*>* findUser*(*@PathVariable*(*"name"*)*String name*){*

return userRepository.findByName*(*name*)*;

*}*

## 4.3 定义repository接口

### 4.3.1 选择性暴露CRUD方法

以下例子中，继承MyBaseRepository的接口只暴露findById、save两个方法

@NoRepositoryBean

public interface MyBaseRepository*<*T, ID*>* extends Repository*<*T, ID*> {*

Optional*<*T*>* findById*(*ID id*)*;

*<*S extends T*>* S save*(*S entity*)*;

*}*

*其他需要的方法需要自己再单独定义，则*PeopleRepository *可以使用*findById、save*、*findByFullName*三个方法*

public interface PeopleRepository extends MyBaseRepository*<*People, Long*> {*

People findByFullName*(*String fullName*)*;

*}*

### 4.3.2 将Repositories与多个Spring data模块一起使用

如果您的应用程序中使用唯一的Spring Data模块，则会很简单，因为定义范围中的所有repository接口均已绑定到Spring Data模块.有时，应用程序需要使用多个Spring Data模块.在这种情况下，repository定义必须区分持久化技术.当在类路径上检测到多个repository factories时，Spring Data进入精确的repository配置模式，精确配置使用repository或domain类上的详细信息来决定哪个Spring Data模块来绑定相关的repository定义：

1. 如果repository定义[扩展了特定于模块的](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/" \l "repositories.multiple-modules.types)repository，则它是特定Spring Data模块的有效可选项.
2. 如果该domain类[使用模块特定的类型注解进行注解](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/" \l "repositories.multiple-modules.annotations)，则它是特定Spring Data模块的有效可选项.Spring Data模块可以接受第三方注解（例如JPA的注解@Entity），也可以提供自己的注解（例如为Spring Data MongoDB和Spring Data Elasticsearch提供的@Document注解）.

以下示例展示了使用特定模块的接口（此时为JPA）的repository：

**例子8.使用模块特定接口的repository定义**

interface MyRepository extends JpaRepository<User, Long> { } @NoRepositoryBean

interface MyBaseRepository<T, ID> extends JpaRepository<T, ID> { … } interface UserRepository extends MyBaseRepository<User, Long> { … }

MyRepository和UserRepository扩展JpaRepository其类型层次结构，则它们是Spring Data JPA模块的有效可选项.

以下示例显示了使用通用接口的repository：

**例子9.使用通用接口的repository定义**

interface AmbiguousRepository extends Repository<User, Long> { … } @NoRepositoryBean

interface MyBaseRepository<T, ID> extends CrudRepository<T, ID> { … } interface AmbiguousUserRepository extends MyBaseRepository<User, Long> { … }

在上述类结构层次中，AmbiguousRepository和AmbiguousUserRepository仅继承Repository和CrudRepository，尽管在使用唯一的Spring Data模块时是挺好，但是多个模块无法区分这些repository应绑定到哪个特定的Spring Data.

以下示例显示了使用带注解domain类的repository：

**例子10.**使**用带注解domain类的repository定义**

interface PersonRepository extends Repository<Person, Long> { … }

@Entity class Person { … }

interface UserRepository extends Repository<User, Long> { … }

@Document class User { … }

PersonRepository 引用 Person，它使用JPA的 @Entity注解，因此该repository显然属于Spring Data JPA.

UserRepository引用User，使用Spring Data MongoDB的@Document进行注解.

以下的反例显示了使用带有混合注解domain类的repository：

**例子11.使用带有混合注解domain类的repository定义**

interface JpaPersonRepository extends Repository<Person, Long> { … } interface MongoDBPersonRepository extends Repository<Person, Long> { … } @Entity

@Document

class Person { … }

此示例中的domain类同时使用JPA和Spring Data MongoDB的注解，定义了两个repositoryJpaPersonRepository和MongoDBPersonRepository.一个用于JPA，另一个用于MongoDB.Spring Data不再能够区分repository，这将导致不确定操作.

[repository类型详细信息](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#repositories.multiple-modules.types)和[可区分的实体类注解](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#repositories.multiple-modules.annotations)用于严格的repository配置，以标识特定Spring Data模块的repository候选对象.在同一个实体类型上使用多个特定于持久化技术的注解是可能的，并且可以跨多种持久化技术重用实体类型.不过Spring Data无法再确定用于绑定repository的唯一模块.

区分repository的最后一种方法是确定repository基本程序包的范围.基本软件包定义了扫描repository接口定义的起点，这意味着将repository定义放在适当的软件包中.默认情况下，注解驱动的配置使用配置类的包.[基于XML的配置中](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#repositories.create-instances.spring)的[基本软件包](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#repositories.create-instances.spring)是必需的.

以下示例显示了基础包的注解驱动配置：

**例子12.基础包的注解驱动配置**

@EnableJpaRepositories(basePackages = "com.acme.repositories.jpa") @EnableMongoRepositories(basePackages = "com.acme.repositories.mongo") class Configuration { … }

## 4.4.定义Query方法

repository代理有两种从特定存储方法名声明的方式：

* 直接从方法名称声明查询.
* 使用手动定义的查询.

可用选项取决于实际持久化方式.不过必须有一个策略来决定要创建什么操作的query.下一节将介绍几种方式.

### 4.4.1.query查找策略

以下策略可用于repository功能结构来解决查找.使用XML配置，您可以通过query-lookup-strategy属性在命名空间中配置策略.对于Java配置，可以使用注解@Enable${store}Repositories的**queryLookupStrategy**属性，某些数据存储方式可能不全支持这些策略.

* CREATE

从查询方法名称构造特定于持久化方式的query.通用方法是从方法名称中删除一组给定的众所周知的前缀，然后解析该方法的其余部分.您可以在“ [查询创建](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#repositories.query-methods.query-creation) ”中阅读有关查询构造的更多信息.

* USE\_DECLARED\_QUERy

查找声明的查询，如果找不到则抛出异常.该查询可以通过注解定义，也可以通过其他方式声明.请查阅特定持久化方式的文档以找到该持久化方式的可用选项.如果repository基础结构在引导时找不到该方法的声明查询，则启动失败.

* CREATE\_IF\_NOT\_FOUND（默认值）结合了CREATE和USE\_DECLARED\_QUERY.它首先查找一个声明的查询，如果找不到声明的查询，它将创建一个基于名称的自定义方法查询.这是默认的查找策略，因此，如果未显式配置任何内容，将使用它.它允许通过方法名称快速定义查询，也可以通过根据需要引入已声明的查询来自定义这些query.

### 4.4.2.Query创建

Query构建器机制内置于Spring Data repository基础结构中,对于在repository实体构建特定查询很有用.该机制识别方法中的字符串：find…By，read…By，query…By，count…By，和get…By，并解析方法的其余部分.声明的方法名还可以包含其他表达式，例如，Distinct以在要创建的查询语句中设置distinct标志.第一个By充当分隔符，来指示实际解析字符串的开始.最基础的功能是，您可以定义实体属性的条件，并将其与And或者Or连接.下面的示例演示如何创建各种查询：

**例子13.从方法名查询创建**

interface PersonRepository extends Repository<Person, Long> {

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

*// Enables the distinct flag for the query*

List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);

List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);

*// Enabling ignoring case for an individual property*

List<Person> findByLastnameIgnoreCase(String lastname);

*// Enabling ignoring case for all suitable properties*

List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);

*// Enabling static ORDER BY for a query* List<Person> findByLastnameOrderByFirstnameAsc(String lastname);

List<Person> findByLastnameOrderByFirstnameDesc(String lastname);

}

解析该方法的实际结果取决于您为其创建查询的持久化存储方式.不过需要注意以下事项：

* 表达式通常是属性遍历，并带有可级联的运算符.您可以组合使用属性表达式AND和OR.您还可以使用数据库支持的操作符：Between，LessThan，GreaterThan，Like。支持的操作符因数据库而异，请参考参考文档的相应部分.
* 方法解析器支持单个属性的IgnoreCase（例如findByLastnameIgnoreCase(…)）或所有属性忽略大小写类型（通常是Strin类型字段，例如findByLastnameAndFirstnameAllIgnoreCase(…)）.是否支持忽略大小写可能因持久化方式而异，请参考参考文档中有关持久化方式特定查询方法的相关部分.
* 可以通过将OrderBy子句附加到查询方法，并提供排序方向（Asc或Desc）来使用静态排序.要创建支持动态排序的查询方法，请参阅“ [特殊参数处理](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#repositories.special-parameters) ”.

### 4.4.3.属性表达式

如上例所示，属性表达式只能使用repository管理实体中的属性.在查询方法创建时，确保解析出的属性是合法实体类的属性.不过您也可以通过遍历嵌套属性来定义.考虑以下方法定义：

List<Person> findByAddressZipCode(ZipCode zipCode);

假设Person的Address带有ZipCode.在这种情况下，该方法将创建属性遍历x.address.zipCode.解析算法首先将整个部分（AddressZipCode）解释为属性，然后检查实体类中该名称的属性（未大写）.如果算法成功，它将使用该属性.如果不是，该算法将驼峰部分的字符串从右往左分为头和尾，并尝试找到相应的属性-在我们的示例中为AddressZip和Code.如果该算法找到了具有该头部的属性，则它将采用该头部，并继续从那里开始构建树，以刚才描述的方式将尾部向上拆分.如果第一个分割不匹配，则算法会将分割点移到左侧（Address，ZipCode）并继续.

尽管这在大多数情况下应该可行，但是算法可能会选择错误的属性.假设Person该类也具有一个addressZip属性.该算法将在第一轮拆分中已经匹配，选择错误的属性，然后报错（因为类Person可能没有code属性）.

要解决这种歧义，您可以在方法名称内部使用下划线手动定义遍历点.因此，我们的方法名称如下：

List<Person> findByAddress\_ZipCode(ZipCode zipCode);

因为我们将下划线字符视为保留字符，所以强烈建议您遵循Java命名标准（即，在属性名称中不使用下划线，而使用驼峰大小写）.

### 4.4.4.特殊参数处理

要处理查询中的参数，请定义方法参数，就像前面的示例所示.除此之外，基础架构还可以识别某些特定类型（例如Pageable和Sort），以将分页和排序动态应用于查询.以下示例演示了这些功能：

**例14. 在查询方法中使用**Pageable**，**Slice**和**Sort

Page<User> findByLastname(String lastname, Pageable pageable);//①

Slice<User> findByLastname(String lastname, Pageable pageable) ;//②

List<User> findByLastname(String lastname, Sort sort); //③

List<User> findByLastname(String lastname, Pageable pageable);//④

**注意：使用Sort和Pageable的API只需要非null值传递给方法进行处理.**

**如果您不想应用任何排序或分页，请使用Sort.unsorted()和Pageable.unpaged().**

第一种方法使您可以将org.springframework.data.domain.Pageable实例传递给查询方法，以将分页动态添加到静态定义的查询中.使用Page就知道查询数据总数和有效页数.它是通过底层触发count查询来统计总数.这可能花销高（取决于所使用的持久化方式），因此您可以改为返回Slice.一个Slice只知道是否下一个Slice是可用的，对于数据量大的场景，用Slice比较合适.

排序选项也通过Pageable实例处理.如果只需要排序，则将一个org.springframework.data.domain.Sort参数添加到方法中.如上述返回List的③，且不想Pageable那样需要其他参数（这意味着不会触发额外count查询），只查询给定范围的数据.

要知道整个查询可获得多少页，您必须触发另一个count查询.默认情况下，此查询源自您实际触发的查询.

#### 分页和排序

可以使用属性名称定义简单的排序表达式，将表达式连接起来，使多个条件集中到一个表达式中.

**例子15.定义排序表达式**

**Sort sort = Sort.by("firstname").ascending() .and(Sort.by("lastname").descending());**

为了进一步达到类型安全，可先使用实体类型定义排序表达式，然后使用属性方法来定义要进行排序的属性.

**例子16.使用类型安全的API定义排序表达式**

TypedSort<Person> person = Sort.sort(Person.class);

TypedSort<Person> sort = person.by(Person::getFirstname).ascending() .and(person.by(Person::getLastname).descending());

如果您的持久化方式实现支持Querydsl，则还可以使用生成的元模型类型来定义排序表达式：

**例17.使用Querydsl API定义排序表达式**

QSort sort = QSort.by(QPerson.firstname.asc()) .and(QSort.by(QPerson.lastname.desc()));

### 4.4.5.限制查询结果

可以通过使用first或top关键字来限制查询结果，该关键字可以互换使用.可以将数字附加到top或first之后，也就是指定要返回的最大结果数量.如果省略该数字，则假定结果大小为1.以下示例显示了如何限制查询大小：

**例子18.用**Top**和**First**限制查询的结果数量**

User findFirstByOrderByLastnameAsc(); User findTopByOrderByAgeDesc(); Page<User> queryFirst10ByLastname(String lastname, Pageable pageable); Slice<User> findTop3ByLastname(String lastname, Pageable pageable);

List<User> findFirst10ByLastname(String lastname, Sort sort);

List<User> findTop10ByLastname(String lastname, Pageable pageable);

限制表达式也支持Distinct关键字.此外，如果需要限制结果集只为一个实例，可用Optional关键字将查询结果包装到其中.

如果将分页或切片应用于限制查询分页（以及对可用页数的计算），则会在限制结果内应用这些限制.

通过使用Sort参数来限制结果与动态排序的组合，可以实现查询最小的“ K”个元素和对最大的“ K”个元素.

### 4.4.6.repository方法返回Collections或Iterables

查询方法，返回多个结果可以使用标准的Java集合：Iterable，List，Set.除此之外，我们还支持返回Spring Data的Streamable（自定义扩展的Iterable），以及[Vavr](https://www.vavr.io/)提供的集合类型.

#### 使用Streamable作为查询方法返回类型

Streamable可用来替代Iterable或任何集合类型.它提供了便捷的方法来访问非并行项Stream（缺少Iterable），用….filter(…)和….map(…)处理元素，并将其级联到其他元素：

**例子19.使用Streamable合并查询方法结果**

interface PersonRepository extends Repository<Person, Long> {

Streamable<Person> findByFirstnameContaining(String firstname);

Streamable<Person> findByLastnameContaining(String lastname); }

Streamable<Person> result =

repository.findByFirstnameContaining("av") .and(repository.findByLastname

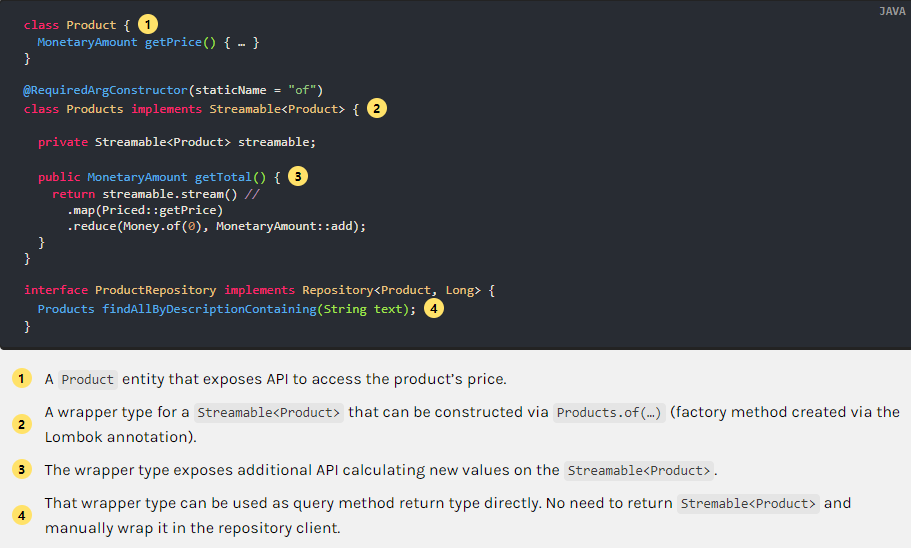
Containing("ea"));

#### 返回自定义流式包装器类型

对于返回多个元素查询结果的API，为集合提供专用的包装器类型是一种常用的模式.通常，这些类型是通过调用repository方法来返回类似集合的类型，并手动创建包装器实例.如果Spring Data允许使用这些包装器类型且满足以下条件，则会很方便：

1. 实现Streamable的实体类.
2. 该实体类存在以Streamable作为参数的构造器或静态工厂方法，如of(…)或valueOf(…).

示例用例如下所示：



上述Product提供API来访问产品的价格

|  |  |
| --- | --- |
|  |  |
|  | Streamable<Product>可以通过构建的包装类型Products.of(…)（通过Lombok注解创建的工厂方法）. |
|  | 包装器类型会公开其他API，以计算上的新值Streamable<Product>. |
|  | 该包装器类型可以直接用作查询方法返回类型.无需返回Stremable<Product>并将其手动包装在repository客户端中. |

#### 支持Vavr集合

[Vavr](https://www.vavr.io/)是一个包含Java中的函数式编程概念的库.它附带了一组自定义的集中类型，可用作查询方法返回类型.

|  |  |  |
| --- | --- | --- |
| Vavr集中类型 | 使用的Vavr实现类型 | 有效的Java源代码类型 |
| io.vavr.collection.Seq | io.vavr.collection.List | java.util.Iterable |
| io.vavr.collection.Set | io.vavr.collection.LinkedHashSet | java.util.Iterable |
| io.vavr.collection.Map | io.vavr.collection.LinkedHashMap | java.util.Map |

第一列中的类型（或其子类型）可以用作查询方法返回类型，并将根据实际查询结果的Java类型（第三列）获取第二列中的类型作为实现类型.或者，可以声明Traversable（Iterable等效于Vavr ），然后从实际返回值声明实现类，即a java.util.List将变为Vavr List/ Seq，a java.util.Set变为Vavr LinkedHashSet/ Set等.

### 4.4.7.repository方法的空参处理

从Spring Data 2.0开始，返回单个聚合实例的repositoryCRUD方法使用Java 8 Optional表示可能没有值.除此之外，Spring Data支持在查询方法上返回以下包装器类型：

* com.google.common.base.Optional
* scala.Option
* io.vavr.control.Option

或者，查询方法可以选择根本不使用包装器类型.然后，通过返回表示查询结果不存在null.保证返回集合，集合替代项，包装器和流的repository方法绝不会返回null，而只会返回相应的空表示.有关详细信息，请参见“ [repository查询返回类型](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#repository-query-return-types) ”.

可空性注解

您可以使用[Spring Framework的nullability注解](https://docs.spring.io/spring/docs/5.2.5.RELEASE/spring-framework-reference/core.html" \l "null-safety)来表达repository方法的nullability约束.它们提供了一种工具友好的方法，并且null在运行时提供了选择加入检查，如下所示：

* [@NonNullApi](https://docs.spring.io/spring/docs/5.2.5.RELEASE/javadoc-api/org/springframework/lang/NonNullApi.html)：在包级别用于声明参数和返回值的默认行为是不接受或产生null值.
* [@NonNull](https://docs.spring.io/spring/docs/5.2.5.RELEASE/javadoc-api/org/springframework/lang/NonNull.html)：用于不得包含的参数或返回值null （适用时，不需要参数和返回值@NonNullApi）.
* [@Nullable](https://docs.spring.io/spring/docs/5.2.5.RELEASE/javadoc-api/org/springframework/lang/Nullable.html)：用于可以为的参数或返回值null.

Spring注解使用[JSR 305](https://jcp.org/en/jsr/detail?id=305)注解（休眠但分布广泛的JSR）进行元注解.JSR 305元注解使工具供应商（例如[IDEA](https://www.jetbrains.com/help/idea/nullable-and-notnull-annotations.html)，[Eclipse](https://help.eclipse.org/oxygen/index.jsp?topic=/org.eclipse.jdt.doc.user/tasks/task-using_external_null_annotations.htm)和[Kotlin）](https://kotlinlang.org/docs/reference/java-interop.html" \l "null-safety-and-platform-types)以通用方式提供了空安全支持，而无需对Spring注解进行硬编码支持.为了对查询方法的可空性约束进行运行时检查，您需要使用Spring的@NonNullApiin 来在包级别激活非可空性package-info.java，如以下示例所示：

**例子20.在声明非空性**package-info.java

@org.springframework.lang.NonNullApi package com.acme;

一旦设置了非null默认值，就可以在运行时验证repository查询方法的调用是否具有可空性约束.如果查询执行结果违反了定义的约束，则会引发异常.当方法返回null但被声明为不可为空时（在repository所在的包中定义了默认值，带有注解），就会发生这种情况.如果您想再次选择接受可为空的结果，请选择性地@Nullable在各个方法上使用.使用本节开头提到的结果包装器类型可以按预期继续工作：将空结果转换为表示缺席的值.

以下示例显示了刚才描述的各种技术：

**例子21.使用不同的可空性约束**

package com.acme; import org.springframework.lang.Nullable; interface UserRepository extends Repository<User, Long> { User getByEmailAddress(EmailAddress emailAddress); @Nullable User findByEmailAddress(@Nullable EmailAddress emailAdress); Optional<User> findOptionalByEmailAddress(EmailAddress emailAddress); }

|  |  |
| --- | --- |
|  | repository位于我们已为其定义非空行为的包（或子包）中. |
|  | EmptyResultDataAccessException当执行查询不产生结果时抛出.传递给方法IllegalArgumentException时抛出.emailAddressnull |
|  | null当执行的查询不产生结果时返回.也接受null作为的值emailAddress. |
|  | Optional.empty()当执行的查询不产生结果时返回.传递给方法IllegalArgumentException时抛出.emailAddressnull |

基于Kotlin的repository中的可空性

Kotlin定义了语言中包含的可[空性约束](https://kotlinlang.org/docs/reference/null-safety.html).Kotlin代码编译为字节码，字节码不通过方法定义来表达可空性约束，而是通过内置的元数据来表达.确保kotlin-reflect在您的项目中包含JAR，以对Kotlin的可空性约束进行自省.Spring Datarepository使用语言机制来定义这些约束以应用相同的运行时检查，如下所示：

**例子22.在Kotlin仓库上使用可空性约束**

interface UserRepository : Repository<User, String> { fun findByUsername(username: String): User fun findByFirstname(firstname: String?): User? }

|  |  |
| --- | --- |
|  | 该方法将参数和结果都定义为不可为空（Kotlin默认值）.Kotlin编译器拒绝传递null给该方法的方法调用.如果查询执行产生空结果，EmptyResultDataAccessException则抛出. |
|  | 该方法接受null的firstname参数，并返回null，如果查询执行不产生结果. |

### 4.4.8.查询结果流化处理

通过使用Java 8 Stream<T>作为返回类型，可以逐步处理查询方法的结果.Stream并非将查询结果包装在数据存储区中，而是使用特定于方法的方法来执行流传输，如以下示例所示：

**例子23.用Java 8流式查询的结果**Stream<T>

@Query("select u from User u") Stream<User> findAllByCustomQueryAndStream(); Stream<User> readAllByFirstnameNotNull(); @Query("select u from User u") Stream<User> streamAllPaged(Pageable pageable);

|  |  |
| --- | --- |
|  | 一个Stream潜在包装特定于底层数据存储的资源，因此必须在使用后关闭.您可以Stream使用close()方法或使用Java 7 try-with-resources块来手动关闭，如以下示例所示： |

**例子24.**Stream<T>**在try-with-resources块中处理结果**

try (Stream<User> stream = repository.findAllByCustomQueryAndStream()) { stream.forEach(…); }

|  |  |
| --- | --- |
|  | 当前，并非所有的Spring Data模块都支持Stream<T>作为返回类型. |

### 4.4.9.异步查询处理

仓库查询可以通过使用[Spring的异步方法执行能力](https://docs.spring.io/spring/docs/5.2.5.RELEASE/spring-framework-reference/integration.html" \l "scheduling)来异步运行.这意味着该方法在调用时立即返回，而实际查询执行发生在已提交给Spring的任务中TaskExecutor.异步查询执行与反应式查询执行不同，因此不应混为一谈.有关响应式支持的更多详细信息，请参阅持久化方式特定的文档.以下示例显示了各种异步查询：

@Async Future<User> findByFirstname(String firstname); @Async CompletableFuture<User> findOneByFirstname(String firstname); @Async ListenableFuture<User> findOneByLastname(String lastname);

|  |  |
| --- | --- |
|  | 使用java.util.concurrent.Future作为返回类型. |
|  | 使用Java 8 java.util.concurrent.CompletableFuture作为返回类型. |
|  | 使用a org.springframework.util.concurrent.ListenableFuture作为返回类型. |

## 4.5.创建repository实例

在本部分中，将为已定义的repository接口创建实例和Bean定义.一种方法是使用支持repository机制的每个Spring Data模块附带的Spring命名空间，尽管我们通常建议使用Java配置.

### 4.5.1.XML配置

每个Spring Data模块都包含一个repositories元素，可让您定义Spring会为您扫描的基本软件包，如以下示例所示：

**例子25.通过XML启用Spring Data仓库**

<?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 https://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/data/jpa https://www.springframework.org/schema/data/jpa/spring-jpa.xsd"> <repositories base-package="com.acme.repositories" /> </beans:beans>

在前面的示例中，指示Spring扫描com.acme.repositories及其所有子包以查找扩展的接口Repository或其中一个子接口.对于找到的每个接口，基础结构都会注册特定FactoryBean于持久化技术的内容，以创建处理查询方法调用的适当代理.每个Bean都使用从接口名称声明的Bean名称UserRepository进行注册，因此的接口将在之下注册userRepository.该base-package属性允许使用通配符，以便您可以定义扫描程序包的模式.

使用过滤器

默认情况下，基础架构会拾取扩展Repository位于配置的基本程序包下的特定于持久化技术的子接口的每个接口，并为其创建一个bean实例.不过您可能希望更精细地控制哪些接口为其创建了bean实例.为此，请在元素内部使用<include-filter />和<exclude-filter />元素<repositories />.语义完全等同于Spring的上下文命名空间中的元素.有关详细信息，请参见这些元素的[Spring参考文档](https://docs.spring.io/spring/docs/5.2.5.RELEASE/spring-framework-reference/core.html" \l "beans-scanning-filters).

例如，要将某些接口从实例中排除为repositoryBean，可以使用以下配置：

**例子26.使用exclude-filter元素**

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

前面的示例排除了所有以结尾的接口SomeRepository.

### 4.5.2.JavaConfig

还可以通过@Enable${store}Repositories在JavaConfig类上使用持久化方式特定的注解来触发repository基础结构.有关Spring容器的基于Java的配置的简介，请参见[Spring参考文档中的JavaConfig](https://docs.spring.io/spring/docs/5.2.5.RELEASE/spring-framework-reference/core.html" \l "beans-java).

启用S​​pring数据repository的示例配置类似于以下内容：

**例子27.基于样本注解的repository配置**

@Configuration @EnableJpaRepositories("com.acme.repositories") class ApplicationConfiguration { @Bean EntityManagerFactory entityManagerFactory() { *// …* } }

|  |  |
| --- | --- |
|  | 前面的示例使用特定于JPA的注解，您将根据实际使用的持久化方式模块对其进行更改.EntityManagerFactoryBean 的定义也是如此.请参阅有关持久化方式特定配置的部分. |

### 4.5.3独立使用(独立于Spring容器)

您还可以在Spring容器之外使用repository基础结构，例如在CDI环境中.您的类路径中仍然需要一些Spring库，不过通常，您也可以通过编程方式来设置repository.提供repository支持的Spring Data模块附带了特定RepositoryFactory于持久化技术的代码，您可以按以下方式使用它：

**例子28.仓库工厂的独立使用**

RepositoryFactorySupport factory = … *// Instantiate factory here*

UserRepository repository = factory.getRepository(UserRepository.class);

## 4.6.Spring data repository的定制化实现

本节介绍repository定制以及片段如何形成复合repository.

当查询方法需要不同的行为或无法通过查询声明实现时，则有必要提供自定义实现.Spring Datarepository使您可以提供自定义repository代码，并将其与通用CRUD抽象和查询方法功能集成.

### 4.6.1.自定义单个repository

要使用自定义功能丰富repository，必须首先定义片段接口和自定义功能的实现，如以下示例所示：

**例子29.定制仓库功能的接口**

interface CustomizedUserRepository { void someCustomMethod(User user); }

**例子30.定制仓库功能的实现**

class CustomizedUserRepositoryImpl implements CustomizedUserRepository { public void someCustomMethod(User user) { *// Your custom implementation* } }

|  |  |
| --- | --- |
|  | 与片段接口相对应的类名称中最重要的部分是Impl后缀. |

实现本身不依赖于Spring Data，可以是常规的Spring bean.因此，您可以使用标准的依赖项注入行为来注入对其他bean（例如JdbcTemplate）的引用，参与各个方面，等等.

然后，可以让您的repository接口扩展片段接口，如以下示例所示：

**示例31.对您的repository界面的更改**

interface UserRepository extends CrudRepository<User, Long>, CustomizedUserRepository { *// Declare query methods here* }

用您的repository接口扩展片段接口，将CRUD和自定义功能结合在一起，并使它可用于客户端.

Spring Datarepository是通过使用构成repository组成的片段来实现的.片段是基础repository，功能方面（例如[QueryDsl](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/" \l "core.extensions.querydsl)）以及自定义接口及其实现.每次向repository接口添加接口时，都通过添加片段来增强组合.每个Spring Data模块都提供了基础repository和repository方面的实现.

以下示例显示了自定义接口及其实现：

**例子32.片段及其实现**

interface HumanRepository { void someHumanMethod(User user); } class HumanRepositoryImpl implements HumanRepository { public void someHumanMethod(User user) { *// Your custom implementation* } } interface ContactRepository { void someContactMethod(User user); User anotherContactMethod(User user); } class ContactRepositoryImpl implements ContactRepository { public void someContactMethod(User user) { *// Your custom implementation* } public User anotherContactMethod(User user) { *// Your custom implementation* } }

以下示例显示了扩展的自定义repository的界面CrudRepository：

**示例33.对您的repository界面的更改**

interface UserRepository extends CrudRepository<User, Long>, HumanRepository, ContactRepository { *// Declare query methods here* }

repository可能由多个自定义实现组成，这些自定义实现按其声明顺序导入.自定义实现比基础实现和repository方面的优先级更高.通过此排序，可以覆盖基本repository和方面方法，并在两个片段贡献相同方法定义的情况下解决歧义.repository片段不限于在单个repository界面中使用.多个repository可以使用片段接口，使您可以跨不同的repository重用自定义项.

以下示例显示了repository片段及其实现：

**例子34.片段覆盖**save(…)

interface CustomizedSave<T> { <S extends T> S save(S entity); } class CustomizedSaveImpl<T> implements CustomizedSave<T> { public <S extends T> S save(S entity) { *// Your custom implementation* } }

以下示例显示了使用上述repository片段的repository：

**例子35.定制的repository界面**

interface UserRepository extends CrudRepository<User, Long>, CustomizedSave<User> { } interface PersonRepository extends CrudRepository<Person, Long>, CustomizedSave<Person> { }

组态

如果使用命名空间配置，则repository基础结构会尝试通过扫描发现repository的包下方的类来自动检测自定义实现片段.这些类需要遵循将命名空间元素的repository-impl-postfix属性附加到片段接口名称的命名约定.此后缀默认为Impl.以下示例显示了使用默认后缀的repository和为后缀设置自定义值的repository：

**例子36.配置例子**

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

上一个示例中的第一个配置尝试查找一个称为com.acme.repository.CustomizedUserRepositoryImpl自定义repository实现的类.第二个示例尝试查找com.acme.repository.CustomizedUserRepositoryMyPostfix.

解决歧义

如果在不同的包中找到具有匹配类名的多个实现，Spring Data将使用Bean名称来标识要使用的那个.

给定CustomizedUserRepository前面显示的以下两个自定义实现，将使用第一个实现.它的bean名称是customizedUserRepositoryImpl，与片段接口（CustomizedUserRepository）和后缀的名称匹配Impl.

**例子37.歧义实现的解决**

package com.acme.impl.one; class CustomizedUserRepositoryImpl implements CustomizedUserRepository { *// Your custom implementation* }

package com.acme.impl.two; @Component("specialCustomImpl") class CustomizedUserRepositoryImpl implements CustomizedUserRepository { *// Your custom implementation* }

如果使用注解UserRepository接口@Component("specialCustom")，则Bean名称plus Impl将与在中为repository实现定义的名称匹配com.acme.impl.two，并且将使用它代替第一个.

手动接线

如果您的自定义实现仅使用基于注解的配置和自动装配，则显示的上述方法会很好地起作用，因为它被视为其他任何Spring bean.如果您的实现片段bean需要特殊的接线，则可以声明bean并根据上[一节中](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/" \l "repositories.single-repository-behaviour.ambiguity)描述的约定对其进行命名.然后，基础结构通过名称引用手动定义的bean定义，而不是自己创建一个.以下示例显示如何手动连接自定义实现：

**例子38.手工连接定制实现**

<repositories base-package="com.acme.repository" /> <beans:bean id="userRepositoryImpl" class="…"> *<!-- further configuration -->* </beans:bean>

### 4.6.2.自定义基础repository

当您要自定义基本repository行为以使所有repository受到影响时，上[一节中](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/" \l "repositories.manual-wiring)描述的方法要求自定义每个repository接口.要改为更改所有repository的行为，可以创建一个实现，以扩展特定于持久化技术的repository基类.然后，该类充当repository代理的自定义基类，如以下示例所示：

**例子39.定制repository基类**

class MyRepositoryImpl<T, ID> extends SimpleJpaRepository<T, ID> { private final EntityManager entityManager; MyRepositoryImpl(JpaEntityInformation entityInformation, EntityManager entityManager) { super(entityInformation, entityManager); *// Keep the EntityManager around to used from the newly introduced methods.* this.entityManager = entityManager; } @Transactional public <S extends T> S save(S entity) { *// implementation goes here* } }

|  |  |
| --- | --- |
|  | 该类需要具有特定于repository的repository工厂实现使用的超类的构造函数.如果repository基类具有多个构造函数，请重写一个构造函数，并EntityInformation加上一个持久化方式特定的基础结构对象（例如EntityManager或模板类）. |

最后一步是使Spring Data基础结构了解定制的repository基类.在Java配置中，可以通过使用注解的repositoryBaseClass属性来执行此操作@Enable${store}Repositories，如以下示例所示：

**例子40.使用JavaConfig配置一个定制的仓库基类**

@Configuration @EnableJpaRepositories(repositoryBaseClass = MyRepositoryImpl.class) class ApplicationConfiguration { … }

XML命名空间中提供了相应的属性，如以下示例所示：

**例子41.使用XML配置一个定制的repository基类**

<repositories base-package="com.acme.repository" base-class="….MyRepositoryImpl" />

## 4.7.从聚合根发布事件

由repository管理的实体是聚合根.在域驱动的设计应用程序中，这些聚合根通常发布域事件.Spring Data提供了一个称为的注解@DomainEvents，您可以在聚合根的方法上使用该注解，以使该发布尽可能容易，如以下示例所示：

**例子42.从聚合根公开域事件**

class AnAggregateRoot { @DomainEvents Collection<Object> domainEvents() { *// … return events you want to get published here* } @AfterDomainEventPublication void callbackMethod() { *// … potentially clean up domain events list* } }

|  |  |
| --- | --- |
|  | 使用的方法@DomainEvents可以返回单个事件实例或事件的集合.它不能接受任何参数. |
|  | 在发布所有事件之后，我们将使用注解方法@AfterDomainEventPublication.它可以用于潜在地清除要发布的事件列表（以及其他用途）. |

每次调用Spring Datarepository的save(…)一种方法时，就会调用这些方法.

### 4.8.扩展Spring Data

本部分记录了一组Spring Data扩展，这些扩展允许在各种上下文中使用Spring Data.当前，大多数集成都针对Spring MVC.

### 4.8.1.Querydsl扩展

[Querydsl](http://www.querydsl.com/)是一个框架，可通过其流畅的API来构造静态类型的类似SQL的查询.

几个Spring Data模块通过提供与Querydsl的集成QuerydslPredicateExecutor，如以下示例所示：

**例子43. QuerydslPredicateExecutor接口**

public interface QuerydslPredicateExecutor<T> {

Optional<T> findById(Predicate predicate);

Iterable<T> findAll(Predicate predicate);

long count(Predicate predicate);

boolean exists(Predicate predicate); *// … more functionality omitted.* }

|  |  |
| --- | --- |
|  | 查找并返回与匹配的单个实体Predicate. |
|  | 查找并返回与匹配的所有实体Predicate. |
|  | 返回与匹配的实体数Predicate. |
|  | 返回匹配Predicate存在的实体. |

要使用Querydsl支持，请扩展QuerydslPredicateExecutorrepository界面，如以下示例所示

**例子44.repository上的Querydsl集成**

interface UserRepository extends CrudRepository<User, Long>, QuerydslPredicateExecutor<User> { }

前面的示例使您可以使用Querydsl Predicate实例编写类型安全查询，如以下示例所示：

Predicate predicate = user.firstname.equalsIgnoreCase("dave") .and(user.lastname.startsWithIgnoreCase("mathews")); userRepository.findAll(predicate);

### 4.8.2.Web支持

|  |  |
| --- | --- |
|  | 本部分包含有关Spring Data Web支持的文档，该文档在Spring Data Commons的当前（和更高版本）中实现.随着新引入的支持发生了很多变化，我们将以前行为的文档保存在[[web.legacy]中](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/" \l "web.legacy). |

支持repository编程模型的Spring Data模块附带各种Web支持.与Web相关的组件要求Spring MVC JAR位于类路径上.其中一些甚至提供与[Spring HATEOAS的](https://github.com/SpringSource/spring-hateoas)集成.通常，通过使用@EnableSpringDataWebSupport

JavaConfig配置类中的注解来启用集成支持，如以下示例所示：

**例子45.启用Spring Data Web支持**

@Configuration @EnableWebMvc @EnableSpringDataWebSupport class WebConfiguration {}

该@EnableSpringDataWebSupport注解注册几个组件，我们将在一个位讨论.它还将在类路径上检测Spring HATEOAS，并为其注册集成组件（如果存在）.

另外，如果您使用XML配置，请注册SpringDataWebConfiguration或注册HateoasAwareSpringDataWebConfiguration为Spring Bean，如以下示例（针对SpringDataWebConfiguration）所示：

**例子46.在XML中启用Spring Data Web支持**

<bean class="org.springframework.data.web.config.SpringDataWebConfiguration" /> *<!-- If you use Spring HATEOAS, register this one \*instead\* of the former -->* <bean class="org.springframework.data.web.config.HateoasAwareSpringDataWebConfiguration" />

基本网络支持

[上一节中](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#core.web)显示的配置注册了一些基本组件：

* 一个[DomainClassConverter](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/" \l "core.web.basic.domain-class-converter)让仓库管理领实体类的Spring MVC的决心情况下，从请求参数或路径变量.
* [HandlerMethodArgumentResolver](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#core.web.basic.paging-and-sorting)Spring MVC实现的解析实现Pageable以及Sort来自请求参数的实例.

DomainClassConverter

使用DomainClassConverter，您可以直接在Spring MVC控制器方法定义中使用实体类型，从而无需通过repository手动查找实例，如以下示例所示：

**例子47.一个在方法定义中使用实体类型的Spring MVC控制器**

@Controller @RequestMapping("/users") class UserController { @RequestMapping("/{id}") String showUserForm(@PathVariable("id") User user, Model model) { model.addAttribute("user", user); return "userForm"; } }

如您所见，该方法User直接接收一个实例，不需要进一步的查找.可以通过让Spring MVC首先将路径变量转换为id实体类的类型并最终通过调用findById(…)为该实体类型注册的repository实例来访问该实例来解决该实例.

|  |  |
| --- | --- |
|  | 当前，该repository必须实现CrudRepository才有资格被发现以进行转换. |

用于分页和排序的HandlerMethodArgumentResolvers

[上一节中](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#core.web.basic.domain-class-converter)显示的配置代码[段](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#core.web.basic.domain-class-converter)还注册PageableHandlerMethodArgumentResolver以及的实例SortHandlerMethodArgumentResolver.注册启用Pageable并Sort作为有效的控制器方法参数，如以下示例所示：

**例子48.使用Pageable作为控制器方法参数**

@Controller @RequestMapping("/users") class UserController { private final UserRepository repository; UserController(UserRepository repository) { this.repository = repository; } @RequestMapping String showUsers(Model model, Pageable pageable) { model.addAttribute("users", repository.findAll(pageable)); return "users"; } }

前面的方法定义使Spring MVC尝试Pageable使用以下默认配置从请求参数声明实例：

|  |  |
| --- | --- |
| page | 您要检索的页面.0索引，默认为0. |
| size | 您要检索的页面大小.默认为20 |
| sort | 应该以格式排序的属性property,property(,ASC|DESC).默认排序方向为升序.sort如果要切换方向，请使用多个参数，例如?sort=firstname&sort=lastname,asc. |

要自定义此行为，请分别注册一个实现该PageableHandlerMethodArgumentResolverCustomizer接口或该SortHandlerMethodArgumentResolverCustomizer接口的bean .它的customize()方法被调用，让您更改设置，如以下示例所示：

@Bean SortHandlerMethodArgumentResolverCustomizer sortCustomizer() { return s -> s.setPropertyDelimiter("<-->"); }

如果设置现有属性MethodArgumentResolver不足以满足您的目的，请扩展SpringDataWebConfiguration或扩展启用HATEOAS的等效项，覆盖pageableResolver()或sortResolver()方法，然后导入自定义的配置文件，而不使用@Enable注解.

如果您需要从请求中解析多个Pageable或Sort实例（例如，对于多个表），则可以使用Spring的@Qualifier注解将一个或另一个区分.然后，请求参数必须以开头${qualifier}\_.以下示例显示了生成的方法定义：

String showUsers(Model model, @Qualifier("thing1") Pageable first, @Qualifier("thing2") Pageable second) { … }

你有填充thing1\_page和thing2\_page等.

Pageable传递给该方法的默认值等效于a，PageRequest.of(0, 20)但可以使用参数@PageableDefault上的注解进行自定义Pageable.

超媒体对页面的支持

Spring HATEOAS附带一个表示模型类（PagedResources），该类允许Page使用必要的Page元数据和链接丰富实例的内容，以使客户端可以轻松浏览页面.Page到a的转换PagedResources是通过Spring HATEOAS ResourceAssembler接口（称为）的实现完成的PagedResourcesAssembler.下面的示例演示如何使用a PagedResourcesAssembler作为控制器方法参数：

**例子49.使用PagedResourcesAssembler作为控制器方法参数**

@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); } }

如上例所示启用配置，可以将PagedResourcesAssembler其用作控制器方法参数.调用toResources(…)它具有以下效果：

* 的内容Page成为PagedResources实例的内容.
* 所述PagedResources对象获得一个PageMetadata附加实例，并且它填充了从信息Page和底层PageRequest.
* 将PagedResources可能会prev和next连接链路，根据页面的状态.链接指向方法映射到的URI.添加到该方法的分页参数与的设置匹配，PageableHandlerMethodArgumentResolver以确保以后可以解析链接.

假设数据库中有30个Person实例.现在，您可以触发请求（）并查看类似于以下内容的输出：GET <http://localhost:8080/persons>

{ "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 } }

您会看到汇编器生成了正确的URI，并且还选择了默认配置以将参数解析Pageable为即将到来的请求.这意味着，如果您更改该配置，则链接将自动遵循更改.默认情况下，汇编器指向调用它的控制器方法，但是可以通过传递一个自定义项Link（用于构建分页链接的基础）来对其进行自定义，这会使该PagedResourcesAssembler.toResource(…)方法过载.

Web数据绑定支持

通过使用[JSONPath](https://goessner.net/articles/JsonPath/)表达式（需要[Jayway JsonPath](https://github.com/json-path/JsonPath)或[XPath](https://www.w3.org/TR/xpath-31/)表达式（需要[XmlBeam](https://xmlbeam.org/)）），可以使用Spring Data映射（在[Projections中](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/" \l "projections)描述）来绑定传入的请求有效负载，如以下示例所示：

**例子50.使用JSONPath或XPath表达式的HTTP有效负载绑定**

@ProjectedPayload public interface UserPayload { @XBRead("//firstname") @JsonPath("$..firstname") String getFirstname(); @XBRead("/lastname") @JsonPath({ "$.lastname", "$.user.lastname" }) String getLastname(); }

在前面的示例中示出的类型可以用作一个Spring MVC处理程序方法参数或通过使用ParameterizedTypeReference上的一个RestTemplate的方法.前面的方法声明将尝试firstname在给定文档中找到任何地方.该lastnameXML查询是对输入文档的顶层进行.的JSON变体lastname首先尝试顶层，但如果前者未返回值，则还尝试lastname嵌套在user子文档中.这样，可以轻松缓解源文档结构的更改，而无需客户端调用公开的方法（通常是基于类的有效负载绑定的缺点）.

如映射中所述，支持嵌套[映射](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/" \l "projections).如果该方法返回复杂的非接口类型，ObjectMapper则使用Jackson 来映射最终值.

对于Spring MVC，必要的转换器在@EnableSpringDataWebSupport活动时会立即自动注册，并且所需的依赖项在类路径上可用.要用于RestTemplate，请注册ProjectingJackson2HttpMessageConverter（JSON）或XmlBeamHttpMessageConverter手动.

有关更多信息，请参阅规范的[Spring Data Examplesrepository中](https://github.com/spring-projects/spring-data-examples)的[Web映射示例](https://github.com/spring-projects/spring-data-examples/tree/master/web/projection).

Querydsl Web支持

对于那些具有[QueryDSL](http://www.querydsl.com/)集成的持久化方式，可以从Request查询字符串中包含的属性声明查询.

考虑以下查询字符串：

?firstname=Dave&lastname=Matthews

给定User前面示例中的对象，可以使用来将查询字符串解析为以下值QuerydslPredicateArgumentResolver.

QUser.user.firstname.eq("Dave").and(QUser.user.lastname.eq("Matthews"))

|  |  |
| --- | --- |
|  | @EnableSpringDataWebSupport当在类路径上找到Querydsl时 ，该功能将与一起自动启用. |

@QuerydslPredicate在方法定义中添加一个即可使用Predicate，可以使用来运行QuerydslPredicateExecutor.

|  |  |
| --- | --- |
|  | 类型信息通常从方法的返回类型中解析.由于该信息不一定与实体类型匹配，因此使用的root属性可能是一个好主意QuerydslPredicate. |

以下示例显示了如何@QuerydslPredicate在方法定义中使用：

@Controller class UserController { @Autowired UserRepository repository; @RequestMapping(value = "/", method = RequestMethod.GET) String index(Model model, @QuerydslPredicate(root = User.class) Predicate predicate, Pageable pageable, @RequestParam MultiValueMap<String, String> parameters) { model.addAttribute("users", repository.findAll(predicate, pageable)); return "index"; } }

|  |  |
| --- | --- |
|  | 解析查询字符串参数匹配Predicate的User. |

默认绑定如下：

* Object关于的简单属性eq.
* Object像属性一样的集合contains.
* Collection关于的简单属性in.

可以通过Java 8 的bindings属性@QuerydslPredicate或通过使用Java 8 default methods并将这些QuerydslBinderCustomizer方法添加到repository接口来定制这些绑定.

interface UserRepository extends CrudRepository<User, String>, QuerydslPredicateExecutor<User>, QuerydslBinderCustomizer<QUser> { @Override default void customize(QuerydslBindings bindings, QUser user) { bindings.bind(user.username).first((path, value) -> path.contains(value)) bindings.bind(String.class) .first((StringPath path, String value) -> path.containsIgnoreCase(value)); bindings.excluding(user.password); } }

|  |  |
| --- | --- |
|  | QuerydslPredicateExecutor提供对的特定查找器方法的访问Predicate. |
|  | QuerydslBinderCustomizer系统会自动获取在repository界面上定义的定义和快捷方式@QuerydslPredicate(bindings=…​). |
|  | 将username属性的绑定定义为简单contains绑定. |
|  | 将String属性的默认绑定定义为不区分大小写的contains匹配. |
|  | password从Predicate分辨率中排除该属性. |

### 4.8.3.repository填充器

如果您使用Spring JDBC模块，则可能熟悉DataSource使用SQL脚本填充a的支持.尽管它不使用SQL作为数据定义语言，因为它必须独立于存储，因此可以在repository级别使用类似的抽象.因此，填充器支持XML（通过Spring的OXM抽象）和JSON（通过Jackson）来定义用于填充repository的数据.

假设您有一个data.json包含以下内容的文件：

**例子51.用JSON定义的数据**

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

您可以使用Spring Data Commons中提供的repository命名空间的populator元素来填充repository.要将前面的数据填充到PersonRepository中，请声明类似于以下内容的填充器：

**例子52.声明一个Jacksonrepository填充器**

<?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 https://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/data/repository https://www.springframework.org/schema/data/repository/spring-repository.xsd"> <repository:jackson2-populator locations="classpath:data.json" /> </beans>

前面的声明使data.json文件由Jackson读取并反序列化ObjectMapper.

通过检查\_classJSON文档的属性来确定将JSON对象解组到的类型.基础结构最终选择适当的repository来处理反序列化的对象.

要改为使用XML定义应使用repository填充的数据，可以使用unmarshaller-populator元素.您可以配置它以使用Spring OXM中可用的XML marshaller选项之一.有关详细信息，请参见[Spring参考文档](https://docs.spring.io/spring/docs/5.2.5.RELEASE/spring-framework-reference/data-access.html" \l "oxm).以下示例显示如何使用JAXB解组repository填充器：

**例子53.声明一个解组repository填充器（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 https://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/data/repository https://www.springframework.org/schema/data/repository/spring-repository.xsd http://www.springframework.org/schema/oxm https://www.springframework.org/schema/oxm/spring-oxm.xsd"> <repository:unmarshaller-populator locations="classpath:data.json" unmarshaller-ref="unmarshaller" /> <oxm:jaxb2-marshaller contextPath="com.acme" /> </beans>

参考文件

# 5. JPA repository

本章指出了JPA信息库支持的特长.这基于“ [使用Spring数据repository](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#repositories) ”中解释的核心repository支持.确保您对此处介绍的基本概念有很好的理解.

## 5.1.介绍

本节介绍通过以下两种方式配置Spring Data JPA的基础知识：

* “ [Spring命名空间](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#jpa.namespace) ”（XML配置）
* “ [基于注解的配置](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#jpa.java-config) ”（Java配置）

### 5.1.1.Spring命名空间

Spring Data的JPA模块包含一个自定义命名空间，允许定义repositorybean.它还包含某些JPA特有的功能和元素属性.通常，可以通过使用repositories元素来设置JPArepository，如以下示例所示：

**例子54.使用命名空间设置JPArepository**

<?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 https://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/data/jpa https://www.springframework.org/schema/data/jpa/spring-jpa.xsd"> <jpa:repositories base-package="com.acme.repositories" /> </beans>

使用该repositories元素可按“ [创建repository实例](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#repositories.create-instances) ”中所述查找Spring Data存储[库](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/" \l "repositories.create-instances).除此之外，它@Repository还会激活所有注解了的bean的持久化异常转换，以使JPA持久化提供程序引发的异常转换为Spring的DataAccessException层次结构.

自定义命名空间属性

除了repositories元素的默认属性外，JPA命名空间还提供了其他属性，使您可以更详细地控制repository的设置：

|  |  |
| --- | --- |
| entity-manager-factory-ref | 明确EntityManagerFactory将要使用的链接到repositories元素检测到的repository.通常EntityManagerFactory在应用程序中使用多个bean的情况下使用.如果未配置，春数据自动查找了EntityManagerFactory名为豆entityManagerFactory中ApplicationContext. |
| transaction-manager-ref | 明确PlatformTransactionManager将要使用的链接到repositories元素检测到的repository.通常仅EntityManagerFactory在配置了多个事务管理器或bean 时才需要.默认为PlatformTransactionManagercurrent中定义的单个ApplicationContext. |
|  | 如果未定义显式 数据，Spring Data JPA要求提供一个PlatformTransactionManager名为的bean . transactionManagertransaction-manager-ref |

### 5.1.2.基于注解的配置

Spring Data JPArepository支持不仅可以通过XML命名空间来激活，还可以通过JavaConfig使用注解来激活，如以下示例所示：

**例子55.使用JavaConfig的Spring Data JPArepository**

@Configuration @EnableJpaRepositories @EnableTransactionManagement class ApplicationConfig { @Bean public DataSource dataSource() { EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder(); return builder.setType(EmbeddedDatabaseType.HSQL).build(); } @Bean public LocalContainerEntityManagerFactoryBean entityManagerFactory() { HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter(); vendorAdapter.setGenerateDdl(true); LocalContainerEntityManagerFactoryBean factory = new LocalContainerEntityManagerFactoryBean(); factory.setJpaVendorAdapter(vendorAdapter); factory.setPackagesToScan("com.acme.domain"); factory.setDataSource(dataSource()); return factory; } @Bean public PlatformTransactionManager transactionManager(EntityManagerFactory entityManagerFactory) { JpaTransactionManager txManager = new JpaTransactionManager(); txManager.setEntityManagerFactory(entityManagerFactory); return txManager; } }

|  |  |
| --- | --- |
|  | 您必须创建LocalContainerEntityManagerFactoryBean而不是EntityManagerFactory直接创建，因为前者除了创建外还参与异常转换机制EntityManagerFactory. |

前面的配置类使用的EmbeddedDatabaseBuilderAPI 建立了嵌入式HSQL数据库spring-jdbc.然后，Spring Data会设置一个，EntityManagerFactory并使用Hibernate作为示例持久化提供程序.在此声明的最后一个基础结构组件是JpaTransactionManager.最后，该示例通过使用@EnableJpaRepositories注解激活Spring Data JPArepository，该注解实质上具有与XML命名空间相同的属性.如果未配置任何基本程序包，它将使用配置类所在的程序包.

### 5.1.3.引导模式

默认情况下，Spring Data JPArepository是默认的Spring Bean.它们是单例作用域的，并且急切地初始化.在启动过程中，它们已经与JPA交互以EntityManager进行验证和元数据分析.Spring Framework EntityManagerFactory在后台线程中支持JPA的初始化，因为该过程通常在Spring应用程序中占用大量启动时间.为了有效地利用该后台初始化，我们需要确保JPArepository尽可能早地初始化.

从Spring Data JPA 2.1开始，您现在可以配置BootstrapMode（通过@EnableJpaRepositories注解或XML命名空间）采用以下值的：

* DEFAULT（默认）—急切地实例化repository，除非用显式注解@Lazy.仅当没有任何客户Bean需要repository实例时，lazification才有效，因为这将需要初始化repositorybean.
* LAZY —隐式地将所有repositorybean声明为惰性的，并且还使创建的惰性初始化代理被注入到客户端bean中.这意味着，如果客户端bean仅将实例存储在字段中并且在初始化期间不使用repository，则不会实例化repository.首次与repository交互时，将初始化并验证repository实例.
* DEFERRED —与基本上具有相同的操作模式LAZY，但响应触发repository初始化，ContextRefreshedEvent以便在应用程序完全启动之前验证repository.

推荐建议

如果您不使用默认引导方式的异步JPA引导棒.

如果您以异步方式引导JPA，这DEFERRED是一个合理的默认值，因为它将确保Spring Data JPA引导仅在EntityManagerFactory其花费比初始化所有其他应用程序组件更长的时间时才等待安装.尽管如此，它仍可以确保在应用程序发出信号之前，对repository进行了正确的初始化和验证.

LAZY是测试方案和本地开发的不错选择.一旦确定了repository将正确引导后，或者在测试应用程序的其他部分的情况下，对所有repository执行验证可能只会不必要地增加启动时间.这同样适用于本地开发，在本地开发中，您仅访问应用程序的某些部分，而这些部分可能只需要初始化一个repository即可.

## 5.2.持久实体

本节描述如何使用Spring Data JPA持久化（保存）实体.

### 5.2.1.保存实体

可以使用该CrudRepository.save(…)方法执行保存实体.它通过使用基础JPA持久化或合并给定实体EntityManager.如果实体尚未持久化，Spring Data JPA将通过调用该entityManager.persist(…)方法保存该实体.否则，它将调用该entityManager.merge(…)方法.

实体状态检测策略

Spring Data JPA提供以下策略来检测实体是否为新实体：

1. Version-Property和Id-Property检查（**默认**）：默认情况下，Spring Data JPA首先检查是否存在非基本类型的Version-property.如果存在，则将实体视为新实体（如果值为）null.没有这样的版本属性，Spring Data JPA会检查给定实体的标识符属性.如果identifier属性为null，则假定该实体为新实体.否则，假定它不是新的.
2. 实现Persistable：如果实体实现Persistable，Spring Data JPA会将新的检测委托给isNew(…)实体的方法.有关详细信息，请参见[JavaDoc](https://docs.spring.io/spring-data/data-commons/docs/current/api/index.html?org/springframework/data/domain/Persistable.html).
3. 实施EntityInformation：您可以通过创建的子类并相应地覆盖方法来自定义实施中EntityInformation使用的抽象.然后，您必须将的自定义实现注册为Spring bean.请注意，这几乎没有必要.有关详细信息，请参见[JavaDoc](https://docs.spring.io/spring-data/data-jpa/docs/current/api/index.html?org/springframework/data/jpa/repository/support/JpaRepositoryFactory.html).SimpleJpaRepositoryJpaRepositoryFactorygetEntityInformation(…)JpaRepositoryFactory

对于使用手动分配的标识符的实体，选项1不是选项，因为标识符始终为非标识符null.在这种情况下，一种常见的模式是使用一个公共基类，该基类的过渡标志默认表示一个新实例，并使用JPA生命周期回调在持久化操作上翻转该标志：

**例子56.具有手动分配的标识符的实体的基类**

@MappedSuperclass public abstract class AbstractEntity<ID> implements Persistable<ID> { @Transient private boolean isNew = true; @Override public boolean isNew() { return isNew; } @PrePersist @PostLoad void markNotNew() { this.isNew = false; } *// More code…* }

|  |  |
| --- | --- |
|  | 声明一个标志以保持新状态.临时的，因此它不会持久化到数据库中. |
|  | 返回的实现中的标志，Persistable.isNew()以便Spring Datarepository知道是调用EntityManager.persist()还是….merge(). |
|  | 声明使用JPA实体回调的方法，以便save(…)在持久化提供程序对repository进行调用或创建实例后，将标志切换为指示现有实体. |

## 5.3.查询方法

本节描述了使用Spring Data JPA创建查询的各种方法.

### 5.3.1.查询查询策略

JPA模块支持手动将查询定义为String或从方法名称声明查询.

与谓词声明查询IsStartingWith，StartingWith，StartsWith，IsEndingWith，EndingWith，EndsWith， IsNotContaining，NotContaining，NotContains，IsContaining，Containing，Contains这些查询的各个参数将得到消毒.这意味着，如果参数实际包含由LIKE通配符识别的字符，则这些字符将被转义，因此它们仅作为文字匹配.所使用的转义字符可以通过设置来配置escapeCharacter所述的@EnableJpaRepositories注解.与[使用SpEL表达式](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/" \l "jpa.query.spel-expressions)进行比较.

声明查询

尽管从方法名声明一个查询很方便，但可能会遇到这样一种情况，即方法名解析器不支持一个人想使用的关键字，或者方法名不必要地变得丑陋.因此，您可以通过命名约定[使用JPA命名查询（](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/" \l "jpa.query-methods.named-queries)有关更多信息，请参见[使用JPA命名查询](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#jpa.query-methods.named-queries)），也@Query可以[通过@Query](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/" \l "jpa.query-methods.at-query)以下方式为您的查询方法添加注解（有关详细信息，请参见[使用](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/" \l "jpa.query-methods.at-query)）.

### 5.3.2.查询创建

通常，JPA的查询创建机制按“ [查询方法](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#repositories.query-methods) ”中所述工作.以下示例显示了JPA查询方法转换为的内容：

**例子57.从方法名查询创建**

公共接口UserRepository扩展了Repository <User，Long> { List <用户> findByEmailAddressAndLastname（字符串emailAddress，字符串姓氏）; }

我们从中使用JPA标准API创建查询，但是从本质上讲，这将转换为以下查询：select u from User u where u.emailAddress = ?1 and u.lastname = ?2.Spring Data JPA进行属性检查并遍历嵌套的属性，如“ [属性表达式](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#repositories.query-methods.query-property-expressions) ”中所述.

下表描述了JPA支持的关键字以及包含该关键字的方法所转换的含义：

|  |  |  |
| --- | --- | --- |
| 关键词 | 样品 | JPQL片段 |
| And | findByLastnameAndFirstname | … where x.lastname = ?1 and x.firstname = ?2 |
| Or | findByLastnameOrFirstname | … where x.lastname = ?1 or x.firstname = ?2 |
| Is， Equals | findByFirstname，findByFirstnameIs，findByFirstnameEquals | … where x.firstname = ?1 |
| Between | findByStartDateBetween | … where x.startDate between ?1 and ?2 |
| LessThan | findByAgeLessThan | … where x.age < ?1 |
| LessThanEqual | findByAgeLessThanEqual | … where x.age <= ?1 |
| GreaterThan | findByAgeGreaterThan | … where x.age > ?1 |
| GreaterThanEqual | findByAgeGreaterThanEqual | … where x.age >= ?1 |
| After | findByStartDateAfter | … where x.startDate > ?1 |
| Before | findByStartDateBefore | … where x.startDate < ?1 |
| IsNull， Null | findByAge(Is)Null | … 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（参数附后%） |
| EndingWith | findByFirstnameEndingWith | … where x.firstname like ?1（参数加前缀%） |
| Containing | findByFirstnameContaining | … where x.firstname like ?1（参数绑定在中%） |
| 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> ages) | … where x.age not in ?1 |
| True | findByActiveTrue() | … where x.active = true |
| False | findByActiveFalse() | … where x.active = false |
| IgnoreCase | findByFirstnameIgnoreCase | … where UPPER(x.firstame) = UPPER(?1) |
|  | In并NotIn接受Collectionas参数的任何子类以及数组或varargs.对于同一逻辑运算符的其他语法版本，请选中“ [repository查询关键字](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#repository-query-keywords) ”. | |

### 5.3.3.使用JPA命名查询

|  |  |
| --- | --- |
|  | 这些示例使用<named-query />元素和@NamedQuery注解.这些配置元素的查询必须使用JPA查询语言进行定义.当然，你可以使用<named-native-query />或@NamedNativeQuery过.这些元素使您可以通过失去数据库平台独立性来在本机SQL中定义查询. |

XML命名查询定义

要使用XML配置，请将必要的<named-query />元素添加到orm.xml位于META-INF类路径文件夹中的JPA配置文件中.通过使用一些定义的命名约定，可以自动调用命名查询.有关更多详细信息，请参见下文.

**例子58. 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>

该查询具有一个特殊名称，该名称用于在运行时解析它.

基于注解的配置

基于注解的配置的优点是不需要编辑另一个配置文件，从而减少了维护工作.您需要为每个新的查询声明重新编译实体类，从而为此付出了代价.

**例子59.基于注解的命名查询配置**

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

声明接口

要允许执行这些命名查询，请指定UserRepository如下：

**例子60. UserRepository中的查询方法声明**

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

Spring Data尝试将对这些方法的调用解析为对命名查询的调用，从已配置的实体类的简单名称开始，然后是由句点分隔的方法名称.因此，前面的示例将使用在示例中定义的命名查询，而不是尝试从方法名称创建查询.

### 5.3.4.使用@Query

使用命名查询声明对实体的查询是一种有效的方法，并且对于少量查询也可以正常工作.由于查询本身与执行它们的Java方法相关联，因此实际上您可以通过使用Spring Data JPA @Query注解直接绑定它们，而不是将它们注解到实体类.这样可以将实体类从持久化特定的信息中释放出来，并将查询放置在repository接口中.

注解查询方法的查询优先于使用定义的@NamedQuery查询或在中声明的命名查询orm.xml.

以下示例显示了使用@Query注解创建的查询：

**例子61.使用以下方法声明查询**@Query

public interface UserRepository extends JpaRepository<User, Long> { @Query("select u from User u where u.emailAddress = ?1") User findByEmailAddress(String emailAddress); }

使用高级LIKE表达式

使用创建的手动定义查询的查询执行机制@Query允许LIKE在查询定义内定义高级表达式，如以下示例所示：

**例子62.**like**@Query中的高级表达式**

public interface UserRepository extends JpaRepository<User, Long> { @Query("select u from User u where u.firstname like %?1") List<User> findByFirstnameEndsWith(String firstname); }

在前面的示例中，识别了LIKE分隔符（%），并将查询转换为有效的JPQL查询（删除了%）.查询执行后，传递给方法调用的参数将使用先前识别的LIKE模式进行扩充.

本地查询

该@Query注解允许通过设定运行的原生查询nativeQuery标志设置为true，如图以下示例：

**例子63.使用@Query在查询方法中声明一个本地查询**

public interface UserRepository extends JpaRepository<User, Long> { @Query(value = "SELECT \* FROM USERS WHERE EMAIL\_ADDRESS = ?1", nativeQuery = true) User findByEmailAddress(String emailAddress); }

|  |  |
| --- | --- |
|  | Spring Data JPA当前不支持对本机查询进行动态排序，因为它必须操纵声明的实际查询，而这对于本机SQL无法可靠地完成.不过您可以通过自己指定count查询来使用本机查询进行分页，如以下示例所示： |

**例子64.通过使用以下方法在查询方法中声明本地计数查询以进行分页**@Query

public interface UserRepository extends JpaRepository<User, Long> { @Query(value = "SELECT \* FROM USERS WHERE LASTNAME = ?1", countQuery = "SELECT count(\*) FROM USERS WHERE LASTNAME = ?1", nativeQuery = true) Page<User> findByLastname(String lastname, Pageable pageable); }

通过将.count后缀添加到查询的副本中，类似的方法也可用于命名本地查询.不过您可能需要为计数查询注册结果集映射.

### 5.3.5.使用排序

排序可以通过提供PageRequest或Sort直接使用来完成.Order实例中实际使用的属性Sort需要与您的域模型匹配，这意味着它们需要解析为查询中使用的属性或别名.JPQL将此定义为状态字段路径表达式.

|  |  |
| --- | --- |
|  | 使用任何不可引用的路径表达式都会导致Exception. |

不过Sort与一起使用[@Query](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/" \l "jpa.query-methods.at-query)可以让您潜入Order包含在ORDER BY子句中的函数的未经路径检查的实例.这是可能的，因为Order会附加到给定的查询字符串中.默认情况下，Spring Data JPA拒绝任何Order包含函数调用的实例，但是您可以JpaSort.unsafe用来添加潜在的不安全排序.

以下示例使用Sort和JpaSort，其中包括一个不安全的选项JpaSort：

**例子65.使用**Sort**和**JpaSort

public interface UserRepository extends JpaRepository<User, Long> { @Query("select u from User u where u.lastname like ?1%") List<User> findByAndSort(String lastname, Sort sort); @Query("select u.id, LENGTH(u.firstname) as fn\_len from User u where u.lastname like ?1%") List<Object[]> findByAsArrayAndSort(String lastname, Sort sort); } repo.findByAndSort("lannister", new Sort("firstname")); repo.findByAndSort("stark", new Sort("LENGTH(firstname)")); repo.findByAndSort("targaryen", JpaSort.unsafe("LENGTH(firstname)")); repo.findByAsArrayAndSort("bolton", new Sort("fn\_len"));

|  |  |
| --- | --- |
|  | Sort指向域模型中属性的有效表达式. |
|  | 无效的Sort包含函数调用.Thows异常. |
|  | 有效，Sort包含显式*不安全* Order. |
|  | Sort指向别名函数的有效表达式. |

### 5.3.6.使用命名参数

默认情况下，Spring Data JPA使用基于位置的参数绑定，如前面所有示例中所述.当重构关于参数位置的查询方法时，这会使查询方法容易出错.要解决此问题，可以使用@Param注解为方法参数指定一个具体名称，然后在查询中绑定该名称，如以下示例所示：

**例子66.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); }

|  |  |
| --- | --- |
|  | 方法参数根据它们在已定义查询中的顺序进行切换. |
|  | 从版本4开始，Spring完全支持基于-parameters编译器标志的Java 8参数名称发现.通过在构建中将此标志用作调试信息的替代方法，可以省略@Param命名参数的注解. |

### 5.3.7.使用SpEL表达式

从Spring Data JPA 1.4版开始，我们支持在定义的手动定义的查询中使用受限的SpEL模板表达式@Query.查询执行后，将根据一组预定义的变量对这些表达式进行求值.Spring Data JPA支持名为的变量entityName.它的用法是select x from #{#entityName} x.插入entityName与给定repository关联的实体类型的.该entityName解决如下：如果实体类型已设置的name属性@Entity的注解，它被使用.否则，将使用实体类型的简单类名.

以下示例演示#{#entityName}了查询字符串中表达式的一种用例，您想在其中使用查询方法和手动定义的查询来定义repository接口：

**例67.在repository查询方法中使用SpEL表达式-EntityName**

@Entity public class User {

@Id @GeneratedValue Long id;

String lastname; }

public interface UserRepository extends JpaRepository<User,Long> {

@Query("select u from #{#entityName} u where u.lastname = ?1")

List<User> findByLastname(String lastname); }

为了避免在@Query注解的查询字符串中声明实际的实体名称，可以使用#{#entityName}变量.

|  |  |
| --- | --- |
|  | 该entityName可以通过使用定制@Entity的注解.orm.xmlSpEL表达式不支持in 的自定义. |

当然，您可能只User在查询声明中直接使用过，但这也需要您更改查询.对的引用可以#entityName将User类将来可能的重新映射选择为其他实体名称（例如，使用）@Entity(name = "MyUser").

#{#entityName}查询字符串中表达式的另一个用例是，如果您想为特定实体类型定义带有专用repository接口的通用repository接口.要不在具体接口上重复定义自定义查询方法，可以@Query在通用repository接口的注解的查询字符串中使用实体名称表达式，如以下示例所示：

**例子68.在仓库查询方法中使用SpEL表达式-具有继承的entityName**

@MappedSuperclass public abstract class AbstractMappedType { … String attribute } @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> { … }

在前面的示例中，该MappedTypeRepository接口是用于几个实体类型extension的公共父接口AbstractMappedType.它还定义了通用findAllByAttribute(…)方法，可以在专用repository接口的实例上使用.如果现在调用findByAllAttribute(…)on ConcreteRepository，则查询变为select t from ConcreteType t where t.attribute = ?1.

SpEL表达式用于操作参数，也可以用于操作方法参数.在这些SpEL表达式中，实体名称不可用，但自变量可用.可以通过名称或索引访问它们，如以下示例所示.

**示例69.在repository查询方法中使用SpEL表达式-访问参数.**

@Query("select u from User u where u.firstname = ?1 and u.firstname=?#{[0]} and u.emailAddress = ?#{principal.emailAddress}") List<User> findByFirstnameAndCurrentUserWithCustomQuery(String firstname);

对于like-condition %，人们通常希望将其附加到String值参数的开头或结尾.这可以通过将绑定参数标记或SpEL表达式附加或前缀为来完成%.以下示例再次说明了这一点.

**例子70.在仓库查询方法中使用SpEL表达式-通配符快捷方式.**

@Query("select u from User u where u.lastname like %:#{[0]}% and u.lastname like %:lastname%") List<User> findByLastnameWithSpelExpression(@Param("lastname") String lastname);

当使用like-conditions的值来自不安全来源时，应该对这些值进行清理，以使它们不能包含任何通配符，从而使攻击者可以选择比其应有的能力更多的数据.为此目的，该escape(String)方法在SpEL上下文中可用.它在第一个参数的\_和的所有实例之前%加上第二个参数的单个字符.与JPQL中可用escape的like表达式子句和标准SQL结合使用，可以轻松清除绑定参数.

**示例71.在repository查询方法中使用SpEL表达式-清理输入值.**

@Query("select u from User u where u.firstname like %?#{escape([0])}% escape ?#{escapeCharacter()}") List<User> findContainingEscaped(String namePart);

在repository接口中给出此方法声明，findContainingEscaped("Peter\_")" will find `Peter\_Parker但不能Peter Parker.所使用的转义字符可以通过设置来配置escapeCharacter所述的@EnableJpaRepositories注解.请注意，escape(String)SpEL上下文中可用的方法将仅转义SQL和JPQL标准通配符\_和%.如果基础数据库或JPA实现支持其他通配符，则这些通配符将无法转义.

### 5.3.8.修改查询

前面所有部分均描述了如何声明查询以访问给定实体或实体集合.您可以使用“ [Spring数据repository的定制实现](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#repositories.custom-implementations) ”中介绍的功能来添加定制修改行为.由于此方法对于全面的定制功能是可行的，因此可以通过使用注解查询方法来修改仅需要参数绑定的查询@Modifying，如以下示例所示：

**例子72.声明操作查询**

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

这样做将触发注解该方法的查询作为更新查询，而不是选择查询.作为EntityManager可能的修改查询执行后包含过时的实体，我们不会自动清除它（见[的JavaDoc](https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html)的EntityManager.clear()详细内容），因为这有效地删除所有非刷新的变化仍悬而未决的EntityManager.如果您希望EntityManager自动清除，可以将@Modifying注解的clearAutomatically属性设置为true.

该@Modifying注解是唯一与组合相关的@Query注解.声明的查询方法或自定义方法不需要此注解.

声明删除查询

Spring Data JPA还支持声明的删除查询，使您避免显式声明JPQL查询，如以下示例所示：

**例子73.使用声明的删除查询**

interface UserRepository extends Repository<User, Long> { void deleteByRoleId(long roleId); @Modifying @Query("delete from User u where u.role.id = ?1") void deleteInBulkByRoleId(long roleId); }

尽管该deleteByRoleId(…)方法看起来基本上与产生相同的结果deleteInBulkByRoleId(…)，但是在执行方法方面，这两个方法声明之间存在重要区别.顾名思义，后一种方法针对数据库发出单个JPQL查询（在注解中定义的查询）.这意味着即使当前加载的实例User也看不到生命周期回调.

为了确保生命周期查询被实际调用，调用会deleteByRoleId(…)执行一个查询，然后一个接一个地删除返回的实例，以便持久化提供程序实际上可以@PreRemove在这些实体上调用回调.

实际上，声明的删除查询是执行查询，然后调用CrudRepository.delete(Iterable<User> users)结果并使其行为与中其他delete(…)方法的实现保持同步的快捷方式CrudRepository.

### 5.3.9.应用查询提示

要将JPA查询提示应用于在repository接口中声明的查询，可以使用@QueryHints注解.它需要一个JPA @QueryHint注解数组和一个布尔标志，以潜在地禁用应用于应用分页时触发的附加计数查询的提示，如以下示例所示：

**例子74.将QueryHints与repository方法一起使用**

public interface UserRepository extends Repository<User, Long> { @QueryHints(value = { @QueryHint(name = "name", value = "value")}, forCounting = false) Page<User> findByLastname(String lastname, Pageable pageable); }

前面的声明将为@QueryHint实际查询应用已配置的内容，但省略了将其应用于为计算总页数而触发的计数查询.

### 5.3.10.配置提取和加载图

JPA 2.1规范引入了对指定Fetch-和LoadGraphs的支持，@EntityGraph注解也支持该功能，使您可以引用@NamedEntityGraph定义.您可以在实体上使用该注解来配置结果查询的获取计划.可以通过使用注解上的属性来配置获取的类型（Fetch或Load）.有关更多参考，请参见JPA 2.1 Spec 3.7.4.type@EntityGraph

以下示例显示如何在实体上定义命名实体图：

**例子75.在一个实体上定义一个命名实体图.**

@Entity @NamedEntityGraph(name = "GroupInfo.detail", attributeNodes = @NamedAttributeNode("members"))

public class GroupInfo {

*// default fetch mode is lazy.*

@ManyToMany

List<GroupMember> members = new ArrayList<GroupMember>(); … }

以下示例显示如何在repository查询方法上引用命名实体图：

**示例76.在repository查询方法上引用命名实体图定义.**

@Repository public interface GroupRepository extends CrudRepository<GroupInfo, String> { @EntityGraph(value = "GroupInfo.detail", type = EntityGraphType.LOAD) GroupInfo getByGroupName(String name); }

也可以通过使用定义临时实体图@EntityGraph.提供的内容attributePaths将转换为，EntityGraph而无需显式添加@NamedEntityGraph到您的实体类型，如以下示例所示：

**示例77.在repository查询方法上使用AD-HOC实体图定义.**

@Repository public interface GroupRepository extends CrudRepository<GroupInfo, String> { @EntityGraph(attributePaths = { "members" }) GroupInfo getByGroupName(String name); }

### 5.3.11.映射

Spring Data查询方法通常返回repository管理的聚合根的一个或多个实例.不过有时可能需要根据这些类型的某些属性创建映射.Spring Data允许对专用的返回类型进行建模，以更选择性地检索托管聚合的部分视图.

想象一下一个repository和聚合根类型，例如以下示例：

**例子78.一个样本集合和repository**

class Person { @Id UUID id; String firstname, lastname; Address address; static class Address { String zipCode, city, street; } } interface PersonRepository extends Repository<Person, UUID> { Collection<Person> findByLastname(String lastname); }

现在假设我们只想检索此人的姓名属性.Spring Data提供什么手段来实现这一目标？本章的其余部分将回答该问题.

基于界面的映射

将查询结果限制为仅name属性的最简单方法是声明一个接口，该接口公开要读取的属性的访问器方法，如以下示例所示：

**例子79.一个映射界面来检索属性的子集**

interface NamesOnly { String getFirstname(); String getLastname(); }

此处重要的一点是，此处定义的属性与聚合根中的属性完全匹配.这样做可以使查询方法添加如下：

**例子80.使用基于接口的映射和查询方法的repository**

interface PersonRepository extends Repository<Person, UUID> { Collection<NamesOnly> findByLastname(String lastname); }

查询执行引擎在运行时为返回的每个元素创建该接口的代理实例，并将对公开方法的调用转发给目标对象.

可以递归使用映射.如果您还想包括一些Address信息，请为此创建一个映射接口，并从的声明中返回该接口getAddress()，如以下示例所示：

**例子81.一个映射界面来检索属性的子集**

interface PersonSummary { String getFirstname(); String getLastname(); AddressSummary getAddress(); interface AddressSummary { String getCity(); } }

在方法调用时，将address获取目标实例的属性并将其包装到一个映射代理中.

封闭映射

其访问者方法均与目标集合的属性完全匹配的映射接口被视为封闭映射.下面的示例（也在本章前面使用过）是一个封闭的映射：

**例子82.一个封闭的映射**

interface NamesOnly { String getFirstname(); String getLastname(); }

如果您使用封闭式映射，Spring Data可以优化查询执行，因为我们知道支持映射代理所需的所有属性.有关更多信息，请参见参考文档中特定于模块的部分.

开放式映射

映射接口中的访问器方法也可以通过使用@Value注解来计算新值，如以下示例所示：

**例子83.一个开放的映射**

interface NamesOnly { @Value("#{target.firstname + ' ' + target.lastname}") String getFullName(); … }

target变量中提供了支持映射的聚合根.使用的映射界面@Value是开放式映射.在这种情况下，Spring Data无法应用查询执行优化，因为SpEL表达式可以使用聚合根的任何属性.

中使用的表达式@Value应该不太复杂-您要避免使用String变量进行编程.对于非常简单的表达式，一种选择可能是求助于默认方法（在Java 8中引入），如以下示例所示：

**例子84.使用默认方法自定义逻辑的映射接口**

interface NamesOnly { String getFirstname(); String getLastname(); default String getFullName() { return getFirstname().concat(" ").concat(getLastname()); } }

这种方法要求您能够完全基于映射接口上公开的其他访问器方法来实现逻辑.第二种更灵活的选择是在Spring bean中实现自定义逻辑，然后从SpEL表达式中调用该自定义逻辑，如以下示例所示：

**例子85.采样人对象**

@Component class MyBean { String getFullName(Person person) { … } } interface NamesOnly { @Value("#{@myBean.getFullName(target)}") String getFullName(); … }

注意SpEL表达式如何引用myBean和调用getFullName(…)方法，以及将映射目标作为方法参数转发.SpEL表达式评估支持的方法也可以使用方法参数，然后可以从表达式中引用这些参数.方法参数可通过Object名为的数组获得args.下面的示例演示如何从args数组中获取方法参数：

**例子86.采样人对象**

interface NamesOnly { @Value("#{args[0] + ' ' + target.firstname + '!'}") String getSalutation(String prefix); }

同样，对于更复杂的表达式，应该使用Spring bean并让该表达式调用方法，[如前所述](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/" \l "projections.interfaces.open.bean-reference).

基于类的映射（DTO）

定义映射的另一种方法是使用值类型DTO（数据传输对象），该类型DTO保留应该被检索的字段的属性.这些DTO类型的使用方式与映射接口的使用方式完全相同，不同之处在于，不会发生任何代理并且不能应用嵌套的映射.

如果持久化方式通过限制要加载的字段来优化查询执行，则要加载的字段由公开的构造函数的参数名称确定.

以下示例显示了一个预计的DTO：

**例子87.一个映射的DTO**

class NamesOnly { private final String firstname, lastname; NamesOnly(String firstname, String lastname) { this.firstname = firstname; this.lastname = lastname; } String getFirstname() { return this.firstname; } String getLastname() { return this.lastname; } *// equals(…) and hashCode() implementations* }

|  |  |
| --- | --- |
|  | **避免映射DTO的样板代码**  您可以使用[Project Lombok](https://projectlombok.org/)大大简化DTO的代码，该[项目](https://projectlombok.org/)提供了@Value注解（不要与@Value前面的界面示例中显示的Spring 注解混淆）.如果使用Project Lombok的@Value注解，则前面显示的示例DTO将变为以下内容：  @Value  class NamesOnly {  String firstname, lastname;  }  字段是private final默认设置，并且该类公开了一个构造函数，该构造函数接受所有字段并自动获取equals(…)和hashCode()实现方法. |

动态映射

到目前为止，我们已经将映射类型用作集合的返回类型或元素类型.不过您可能想要选择在调用时要使用的类型（这使它成为动态的）.要应用动态映射，请使用一种查询方法，如以下示例中所示：

**例子88.使用动态映射参数的repository**

interface PersonRepository extends Repository<Person, UUID> { <T> Collection<T> findByLastname(String lastname, Class<T> type); }

这样，该方法可用于按原样或应用映射来获取聚合，如以下示例所示：

**例子89.使用带有动态映射的repository**

void someMethod(PersonRepository people) { Collection<Person> aggregates = people.findByLastname("Matthews", Person.class); Collection<NamesOnly> aggregates = people.findByLastname("Matthews", NamesOnly.class); }

## 5.4.存储过程

JPA 2.1规范引入了对使用JPA标准查询API调用存储过程的支持.我们介绍了@Procedure用于在repository方法中声明存储过程元数据的注解.

下面的示例使用以下存储过程：

**例子90.**plus1inout**HSQL DB中过程的定义**

/; DROP procedure IF EXISTS plus1inout /; CREATE procedure plus1inout (IN arg int, OUT res int) BEGIN ATOMIC set res = arg + 1; END /;

可以通过使用NamedStoredProcedureQuery实体类型上的注解来配置存储过程的元数据.

**例子91.实体上的StoredProcedure元数据定义.**

@Entity @NamedStoredProcedureQuery(name = "User.plus1", procedureName = "plus1inout", parameters = { @StoredProcedureParameter(mode = ParameterMode.IN, name = "arg", type = Integer.class), @StoredProcedureParameter(mode = ParameterMode.OUT, name = "res", type = Integer.class) }) public class User {}

注意，@NamedStoredProcedureQuery存储过程有两个不同的名称. name是JPA使用的名称.procedureName是存储过程在数据库中具有的名称.

您可以通过多种方法从repository方法引用存储过程.可以使用注解的value或procedureName属性直接定义要调用的存储过程@Procedure.这直接引用数据库中的存储过程，并通过忽略任何配置@NamedStoredProcedureQuery.

或者，您可以将@NamedStoredProcedureQuery.name属性指定为@Procedure.name属性.如果既未配置value，procedureName也未name配置，则将repository方法的名称用作name属性.

下面的示例演示如何引用显式映射的过程：

**示例92.在数据库中引用名称为“ plus1inout”的显式映射过程.**

@Procedure("plus1inout") Integer explicitlyNamedPlus1inout(Integer arg);

以下示例与上一个示例等效，但使用procedureName别名：

**例子93.通过**procedureName**别名在数据库中引用名称为“ plus1inout”的隐式映射过程.**

@Procedure(procedureName = "plus1inout") Integer callPlus1InOut(Integer arg);

下面再次与前两个等效，只是使用方法名称而不是显式注解属性.

**示例94.**EntityManager**使用方法名称引用隐式映射的命名存储过程“ User.plus1” .**

@Procedure Integer plus1inout(@Param("arg") Integer arg);

下面的示例演示如何通过引用@NamedStoredProcedureQuery.name属性来引用存储过程.

**例子95.在中引用显式映射的命名存储过程“ User.plus1IO”**EntityManager**.**

@Procedure(name = "User.plus1IO") Integer entityAnnotatedCustomNamedProcedurePlus1IO(@Param("arg") Integer arg);

如果要调用的存储过程具有单个out参数，则该参数可以作为方法的返回值返回.如果@NamedStoredProcedureQuery注解中指定了多个out参数，则可以将这些参数作为a返回Map，其键为@NamedStoredProcedureQuery注解中给定的参数名称.

## 5.5.技术指标

JPA 2引入了一个标准API，您可以使用它来以编程方式构建查询.通过编写criteria，您可以定义实体类查询的where子句.再往前退一步，这些标准可以视为JPA标准API约束所描述的实体的谓词.

Spring Data JPA遵循Eric Evans的书“域驱动设计”中的规范概念，遵循相同的语义，并提供了使用JPA标准API定义此类规范的API.为了支持规范，您可以使用该JpaSpecificationExecutor接口扩展repository接口，如下所示：

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

附加接口具有使您能够以各种方式执行规范的方法.例如，该findAll方法返回与规范匹配的所有实体，如以下示例所示：

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

的Specification接口被定义为如下：

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

规范可以轻松地用于在实体之上构建可扩展的谓词集，然后可以对其进行组合和使用，JpaRepository而无需为每个所需的组合声明查询（方法），如以下示例所示：

**例子96.客户的规格**

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(MonetaryAmount value) { return new Specification<Customer>() { public Predicate toPredicate(Root<T> root, CriteriaQuery<?> query, CriteriaBuilder builder) { *// build query here* } }; } }

诚然，样板文件的数量尚待改进（最终可能会因Java 8闭包而减少），但是客户端会变得更好，正如您将在本节后面看到的那样.该Customer\_类型是使用JPA元模型生成器生成的元模型类型（[有关示例，](https://docs.jboss.org/hibernate/jpamodelgen/1.0/reference/en-US/html_single/" \l "whatisit)请参见[Hibernate实现的文档](https://docs.jboss.org/hibernate/jpamodelgen/1.0/reference/en-US/html_single/#whatisit)）.因此，表达式Customer\_.createdAt假设Customer拥有createdAttype属性Date.除此之外，我们在业务需求抽象级别上表达了一些标准，并创建了可执行文件Specifications.因此，客户端可以使用Specification以下代码：

**例子97.使用一个简单的规范**

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

为什么不为这种数据访问创建查询？与单纯的Specification查询声明相比，使用单个不会带来很多好处.当您将它们组合在一起以创建新Specification对象时，规范的威力才真正发挥出作用.您可以通过提供的默认方法Specification来构建类似于以下内容的表达式来实现此目的：

**例子98.组合规格**

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

Specification提供一些“胶水代码”默认方法来链接和组合Specification实例.这些方法使您可以通过创建新的Specification实现并将其与现有的实现组合来扩展数据访问层.

## 5.6.实例查询

### 5.6.1.介绍

本章介绍了“按示例查询”，并说明了如何使用它.

示例查询（QBE）是一种具有简单界面的用户友好查询技术.它允许动态查询创建，并且不需要您编写包含字段名称的查询.实际上，“按示例查询”根本不需要您使用持久化方式特定的查询语言编写查询.

### 5.6.2.用法

按示例查询API包含三部分：

* 探针：带有填充字段的域对象的实际示例.
* ExampleMatcher：ExampleMatcher附带有关如何匹配特定字段的详细信息.可以在多个示例中重复使用它.
* Example：Example由探头和探头组成ExampleMatcher.它用于创建查询.

按示例查询非常适合几种用例：

* 使用一组静态或动态约束来查询数据存储.
* 频繁重构域对象，而不必担心破坏现有查询.
* 独立于基础数据存储API进行工作.

按示例查询也有一些限制：

* 不支持嵌套或分组属性约束，例如firstname = ?0 or (firstname = ?1 and lastname = ?2).
* 仅支持字符串的开始/包含/结束/正则表达式匹配，以及其他属性类型的完全匹配.

在开始使用“示例查询”之前，您需要具有一个域对象.首先，为您的repository创建一个接口，如以下示例所示：

**例子99.采样人对象**

public class Person { @Id private String id; private String firstname; private String lastname; private Address address; *// … getters and setters omitted* }

前面的示例显示了一个简单的域对象.您可以使用它来创建一个Example.默认情况下，null将忽略具有值的字段，并使用持久化方式特定的默认值来匹配字符串.可以使用of工厂方法或使用来构建示例[ExampleMatcher](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/" \l "query-by-example.matchers).Example是一成不变的.以下清单显示了一个简单的示例：

**例子100.简单的例子**

Person person = new Person(); person.setFirstname("Dave"); Example<Person> example = Example.of(person);

|  |  |
| --- | --- |
|  | 创建域对象的新实例. |
|  | 设置要查询的属性. |
|  | 创建Example. |

可以使用repository执行示例查询.为此，让您的repository接口扩展QueryByExampleExecutor<T>.以下清单显示了该QueryByExampleExecutor界面的摘录：

**例子101.**QueryByExampleExecutor

public interface QueryByExampleExecutor<T> { <S extends T> S findOne(Example<S> example); <S extends T> Iterable<S> findAll(Example<S> example); *// … more functionality omitted.* }

### 5.6.3.示例匹配器

示例不限于默认设置.您可以使用来为字符串匹配，空值处理和特定于属性的设置指定自己的默认值ExampleMatcher，如以下示例所示：

**例子102.具有定制匹配的例子匹配器**

Person person = new Person(); person.setFirstname("Dave"); ExampleMatcher matcher = ExampleMatcher.matching() .withIgnorePaths("lastname") .withIncludeNullValues() .withStringMatcherEnding(); Example<Person> example = Example.of(person, matcher);

|  |  |
| --- | --- |
|  | 创建域对象的新实例. |
|  | 设置属性. |
|  | 创建一个ExampleMatcher期望所有值都匹配.即使没有进一步的配置，它也可以在此阶段使用. |
|  | 构造一个新ExampleMatcher的忽略lastname属性路径. |
|  | 构造一个新ExampleMatcher的lastname属性以忽略属性路径并包含空值. |
|  | 构造一个新ExampleMatcher的lastname属性，以忽略属性路径，包含空值并执行后缀字符串匹配. |
|  | 创建一个新的Example基于域对象和配置ExampleMatcher. |

默认情况下，ExampleMatcher期望探针上设置的所有值都匹配.如果要获得与隐式定义的任何谓词匹配的结果，请使用ExampleMatcher.matchingAny().

您可以为单个属性（例如“名字”和“姓氏”，或者对于嵌套属性，“ address.city”）指定行为.您可以使用匹配选项和区分大小写对其进行调整，如以下示例所示：

**例子103.配置匹配器选项**

ExampleMatcher matcher = ExampleMatcher.matching() .withMatcher("firstname", endsWith()) .withMatcher("lastname", startsWith().ignoreCase()); }

配置匹配器选项的另一种方法是使用lambda（在Java 8中引入）.此方法创建一个回调，要求实现者修改匹配器.您无需返回匹配器，因为配置选项保存在匹配器实例中.以下示例显示了使用lambda的匹配器：

**例子104.用lambdas配置匹配器选项**

ExampleMatcher matcher = ExampleMatcher.matching() .withMatcher("firstname", match -> match.endsWith()) .withMatcher("firstname", match -> match.startsWith()); }

通过Example使用配置的合并视图创建的查询.可以在ExampleMatcher级别上设置默认匹配设置，而可以将单个设置应用于特定的属性路径.ExampleMatcher除非明确定义，否则设置为on 的设置将由属性路径设置继承.属性修补程序上的设置优先于默认设置.下表描述了各种ExampleMatcher设置的范围：

|  |  |
| --- | --- |
| 设置 | 范围 |
| 空处理 | ExampleMatcher |
| 字符串匹配 | ExampleMatcher 和财产路径 |
| 忽略属性 | 属性路径 |
| 区分大小写 | ExampleMatcher 和财产路径 |
| 价值转换 | 属性路径 |

### 5.6.4.执行一个例子

在Spring Data JPA中，您可以对repository使用示例查询，如以下示例所示：

**示例105.使用repository按示例查询**

public interface PersonRepository extends JpaRepository<Person, String> { … } public class PersonService { @Autowired PersonRepository personRepository; public List<Person> findPeople(Person probe) { return personRepository.findAll(Example.of(probe)); } }

|  |  |
| --- | --- |
|  | 当前，只有SingularAttribute属性可以用于属性匹配. |

属性说明符接受属性名称（例如firstname和lastname）.您可以通过将属性与点（address.city）链接在一起进行导航.您还可以使用匹配选项和区分大小写对其进行调整.

下表显示了StringMatcher可以使用的各种选项以及在名为的字段上使用它们的结果firstname：

|  |  |
| --- | --- |
| 匹配 | 逻辑结果 |
| DEFAULT （区分大小写） | firstname = ?0 |
| DEFAULT （不区分大小写） | LOWER(firstname) = LOWER(?0) |
| EXACT （区分大小写） | firstname = ?0 |
| EXACT （不区分大小写） | LOWER(firstname) = LOWER(?0) |
| STARTING （区分大小写） | firstname like ?0 + '%' |
| STARTING （不区分大小写） | LOWER(firstname) like LOWER(?0) + '%' |
| ENDING （区分大小写） | firstname like '%' + ?0 |
| ENDING （不区分大小写） | LOWER(firstname) like '%' + LOWER(?0) |
| CONTAINING （区分大小写） | firstname like '%' + ?0 + '%' |
| CONTAINING （不区分大小写） | LOWER(firstname) like '%' + LOWER(?0) + '%' |

## 5.7.交易性

默认情况下，repository实例上的CRUD方法是事务性的.对于读取操作，事务配置readOnly标志设置为true.所有其他文件均配置为普通格式，@Transactional以便应用默认事务配置.有关详细信息，请参见的JavaDoc [SimpleJpaRepository](https://docs.spring.io/spring-data/data-jpa/docs/current/api/index.html?org/springframework/data/jpa/repository/support/SimpleJpaRepository.html).如果需要调整repository中声明的方法之一的事务配置，请在repository接口中重新声明该方法，如下所示：

**例子106. CRUD的自定义事务配置**

public interface UserRepository extends CrudRepository<User, Long> { @Override @Transactional(timeout = 10) public List<User> findAll(); *// Further query method declarations* }

这样做会使该findAll()方法以10秒的超时运行并且没有该readOnly标志.

更改事务行为的另一种方法是使用外观或服务实现（通常）覆盖多个repository.其目的是为非CRUD操作定义事务边界.以下示例显示了如何将这样的外观用于多个repository：

**例子107.使用外观定义多个repository调用的事务**

@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); } }

此示例使调用addRoleToAllUsers(…)在事务内部运行（参与现有事务或在没有事务的情况下创建新事务）.然后忽略repository中的事务配置，因为外部事务配置确定了实际使用的事务配置.请注意，您必须激活<tx:annotation-driven />或@EnableTransactionManagement显式使用它才能使外立面基于注解的配置起作用.本示例假定您使用组件扫描.

### 5.7.1.事务查询方法

为了使查询方法具有事务性，请@Transactional在您定义的repository接口上使用，如以下示例所示：

**例子108.在查询方法上使用@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(); }

通常，您希望将readOnly标志设置为true，因为大多数查询方法仅读取数据.与此相反，deleteInactiveUsers()利用@Modifying注解并覆盖事务配置.因此，该方法在readOnly标志设置为的情况下运行false.

您可以将事务用于只读查询，并通过设置readOnly标志将其标记为事务.不过这样做并不表示您不会触发操作查询（尽管某些数据库拒绝INSERT和UPDATE只读事务中的语句）.readOnly而是将标志作为提示传播到底层JDBC驱动程序，以进行性能优化.此外，Spring在基础JPA提供程序上执行了一些优化.例如，当与Hibernate一起使用NEVER时，将事务配置为时，刷新模式将设置为readOnly，这将导致Hibernate跳过脏检查（对大对象树的显着改进）.

## 5.8.锁定

要指定要使用的锁定模式，可以@Lock在查询方法上使用注解，如以下示例所示：

**例子109.在查询方法上定义锁元数据**

interface UserRepository extends Repository<User, Long> { *// Plain query method* @Lock(LockModeType.READ) List<User> findByLastname(String lastname); }

此方法声明使被触发的查询配备LockModeType的READ.您还可以通过在repository界面中重新声明CRUD方法并@Lock为其添加注解来定义CRUD方法的锁定，如以下示例所示：

**例子110.在CRUD方法上定义锁元数据**

interface UserRepository extends Repository<User, Long> { *// Redeclaration of a CRUD method* @Lock(LockModeType.READ); List<User> findAll(); }

## 5.9.稽核

### 5.9.1.基本

Spring Data提供了复杂的支持，可以透明地跟踪创建或更改实体的人员以及更改发生的时间.要利用该功能，您必须为实体类配备审核元数据，该审核元数据可以使用注解或通过实现接口来定义.

基于注解的审核元数据

我们提供@CreatedBy和@LastModifiedBy捕捉谁创建或修改的实体以及用户@CreatedDate和@LastModifiedDate以捕获时的变化发生了.

**例子111.被审计实体**

class Customer { @CreatedBy private User user; @CreatedDate private DateTime createdDate; *// … further properties omitted* }

如您所见，可以根据要捕获的信息有选择地应用注解.更改时间捕捉注解可以在类型乔达，时间，性质使用DateTime，遗留Java Date和Calendar，JDK8日期和时间类型，以及long或Long.

基于接口的审核元数据

如果您不想使用注解来定义审核元数据，则可以让您的实体类实现该Auditable接口.它公开了所有审核属性的设置器方法.

还有一个便捷的基类，AbstractAuditable您可以对其进行扩展，以避免需要手动实现接口方法.这样做会增加您的实体类与Spring Data的耦合，这可能是您要避免的事情.通常，首选基于注解的方式来定义审核元数据，因为它侵入性较小且更灵活.

AuditorAware

如果您使用@CreatedBy或@LastModifiedBy，则审计基础结构需要以某种方式了解当前的主体.为此，我们提供了一个AuditorAware<T>SPI接口，您必须实现该接口以告诉基础结构与应用程序交互的当前用户或系统是谁.泛型类型T定义使用@CreatedBy或@LastModifiedBy必须注解的属性的类型.

以下示例显示了使用Spring Security Authentication对象的接口的实现：

**例子112.基于Spring Security的AuditorAware的实现**

class SpringSecurityAuditorAware implements AuditorAware<User> { public Optional<User> getCurrentAuditor() { return Optional.ofNullable(SecurityContextHolder.getContext()) .map(SecurityContext::getAuthentication) .filter(Authentication::isAuthenticated) .map(Authentication::getPrincipal) .map(User.class::cast); } }

该实现访问AuthenticationSpring Security提供的对象，并查找UserDetails您在UserDetailsService实现中创建的自定义实例.我们在这里假设您正在通过UserDetails实现公开域用户，但是根据Authentication找到的内容，您还可以从任何地方查找它.：leveloffset：-1

### 5.9.2.JPA审核

常规审核配置

Spring Data JPA附带了一个实体侦听器，该侦听器可用于触发捕获审计信息.首先，必须AuditingEntityListener在orm.xml文件内部的持久化上下文中注册要用于所有实体的，如以下示例所示：

**例子113.审计配置orm.xml**

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

您还可以AuditingEntityListener使用@EntityListeners注解按实体启用，如下所示：

@Entity @EntityListeners(AuditingEntityListener.class) public class MyEntity { }

|  |  |
| --- | --- |
|  | 审核功能必须spring-aspects.jar位于类路径上. |

在类路径上进行orm.xml适当修改后spring-aspects.jar，激活审核功能只需将Spring Data JPA auditing命名空间元素添加到您的配置中，如下所示：

**例子114.使用XML配置激活审计**

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

从Spring Data JPA 1.5开始，您可以通过使用注解对配置类进行@EnableJpaAuditing注解来启用审核.您仍然必须修改orm.xml文件，并spring-aspects.jar在类路径上.以下示例显示了如何使用@EnableJpaAuditing注解：

**例子115.使用Java配置激活审计**

@Configuration @EnableJpaAuditing class Config { @Bean public AuditorAware<AuditableUser> auditorProvider() { return new AuditorAwareImpl(); } }

如果您将类型的Bean暴露AuditorAware给ApplicationContext，则审核基础结构会自动选择它并使用它来确定要在实体类型上设置的当前用户.如果您在中注册了多个实现，则ApplicationContext可以通过显式设置的auditorAwareRef属性来选择要使用的实现@EnableJpaAuditing.

## 5.10.其他注意事项

### 5.10.1.使用JpaContext在自定义实现

当使用多个EntityManager实例和[自定义repository实现时](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/" \l "repositories.custom-implementations)，您需要将正确的代码连接EntityManager到repository实现类中.您可以通过EntityManager在@PersistenceContext注解中显式命名或（如果EntityManager是@Autowired）使用来实现@Qualifier.

从Spring Data JPA 1.9开始，Spring Data JPA包含一个名为的类，假定您仅由应用程序中的一个实例管理该类，该类JpaContext可让您EntityManager通过托管实体类获取EntityManager.以下示例显示了如何JpaContext在自定义repository中使用：

**例子116.**JpaContext**在一个定制的仓库实现中使用**

class UserRepositoryImpl implements UserRepositoryCustom { private final EntityManager em; @Autowired public UserRepositoryImpl(JpaContext context) { this.em = context.getEntityManagerByManagedType(User.class); } … }

这种方法的优点是，如果将实体类型分配给其他持久化单元，则无需触摸repository即可更改对持久化单元的引用.

### 5.10.2.合并持久化单元

Spring支持具有多个持久化单元.不过有时您可能希望对应用程序进行模块化，但仍要确保所有这些模块都在单个持久化单元中运行.为了实现该行为，Spring Data JPA提供了一个PersistenceUnitManager实现，该实现根据持久化单元的名称自动合并它们，如以下示例所示：

**例子117.使用MergingPersistenceUnitmanager**

<bean class="….LocalContainerEntityManagerFactoryBean"> <property name="persistenceUnitManager"> <bean class="….MergingPersistenceUnitManager" /> </property> </bean>

@Entity类和JPA映射文件的类路径扫描

普通的JPA设置需要在中列出所有注解映射的实体类orm.xml.XML映射文件也是如此.Spring Data JPA提供了一个ClasspathScanningPersistenceUnitPostProcessor，它可以配置基本软件包，还可以选择采用映射文件名模式.然后，它将扫描给定的软件包以查找带有@Entity或注解的类@MappedSuperclass，并加载与文件名模式匹配的配置文件，并将其交给JPA配置.后处理器必须配置如下：

**例子118.使用ClasspathScanningPersistenceUnitPostProcessor**

<bean class="….LocalContainerEntityManagerFactoryBean"> <property name="persistenceUnitPostProcessors"> <list> <bean class="org.springframework.data.jpa.support.ClasspathScanningPersistenceUnitPostProcessor"> <constructor-arg value="com.acme.domain" /> <property name="mappingFileNamePattern" value="\*\*/\*Mapping.xml" /> </bean> </list> </property> </bean>

|  |  |
| --- | --- |
|  | 从Spring 3.1开始，可以LocalContainerEntityManagerFactoryBean直接在扫描包上配置要扫描的包，以对实体类启用类路径扫描.有关详细信息，请参见[JavaDoc](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/orm/jpa/LocalContainerEntityManagerFactoryBean.html" \l "setPackagesToScan(java.lang.String...)). |

### 5.10.3.CDI整合

repository接口的实例通常由容器创建，在使用Spring Data时，Spring是最自然的选择.如[创建repository](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/" \l "repositories.create-instances)实例中所述，Spring为创建bean实例提供了复杂的支持.从1.1.0版开始，Spring Data JPA附带了一个自定义CDI扩展，该扩展允许在CDI环境中使用repository抽象.该扩展是JAR的一部分.要激活它，请在类路径中包含Spring Data JPA JAR.

现在，您可以通过为EntityManagerFactory和实现CDI生产者来设置基础架构EntityManager，如以下示例所示：

class EntityManagerFactoryProducer { @Produces @ApplicationScoped public EntityManagerFactory createEntityManagerFactory() { return Persistence.createEntityManagerFactory("my-presistence-unit"); } public void close(@Disposes EntityManagerFactory entityManagerFactory) { entityManagerFactory.close(); } @Produces @RequestScoped public EntityManager createEntityManager(EntityManagerFactory entityManagerFactory) { return entityManagerFactory.createEntityManager(); } public void close(@Disposes EntityManager entityManager) { entityManager.close(); } }

必要的设置可能会因JavaEE环境而异.您可能需要做的只是EntityManager将a 重新声明为CDI bean，如下所示：

class CdiConfig { @Produces @RequestScoped @PersistenceContext public EntityManager entityManager; }

在前面的示例中，容器必须能够创建JPA EntityManagers本身.所有配置所做的就是将JPA重新导出EntityManager为CDI bean.

EntityManager每当容器请求repository类型的bean时，Spring Data JPA CDI扩展都将所有可用的实例作为CDI bean 拾取，并为Spring Datarepository创建代理.因此，获取Spring Datarepository的一个实例就是声明一个@Injected属性，如下面的示例所示：

class RepositoryClient { @Inject PersonRepository repository; public void businessMethod() { List<Person> people = repository.findAll(); } }

# 6.JPA之STARiBOSS运用

## 6.1使用场景

什么时候用jpa什么时候用mybatis举例说明

记录创建人

分页

JPA底层原理

查询 持久化 事务管理器 数据源

二级缓存

druid

dsl 替换mybatis复杂查询

sql打印

超时时间

读写分离

枚举值、时间戳转换处理

sequence

# 附录

## 附录A：命名空间参考

该<repositories />元素

该<repositories />元素触发Spring Datarepository基础结构的设置.最重要的属性是base-package，它定义了要扫描Spring Datarepository接口的包.请参阅“ [XML配置](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#repositories.create-instances.spring) ”.下表描述了<repositories />元素的属性：

|  |  |
| --- | --- |
| 名称 | 描述 |
| base-package | 定义要扫描的包，以查找\*Repository在自动检测模式下扩展的repository接口（实际接口由特定的Spring Data模块确定）.配置包下面的所有包也将被扫描.允许使用通配符. |
| repository-impl-postfix | 定义后缀以自动检测自定义repository实现.名称以配置的后缀结尾的类被视为候选.默认为Impl. |
| query-lookup-strategy | 确定用于创建查找器查询的策略.有关详细信息，请参见“ [查询查找策略](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#repositories.query-methods.query-lookup-strategies) ”.默认为create-if-not-found. |
| named-queries-location | 定义搜索包含外部定义查询的属性文件的位置. |
| consider-nested-repositories | 是否应考虑嵌套的repository接口定义.默认为false. |

## 附录B：填充器命名空间参考

<populator />元素

该<populator />元素允许通过Spring数据repository基础结构填充数据存储.[ [1](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/#_footnotedef_1) ]

|  |  |
| --- | --- |
| 名称 | 描述 |
| locations | 从哪里可以找到要从repository读取对象的文件，应在其中填充. |

## 附录C：repository查询关键字

支持的查询关键字

下表列出了Spring Datarepository查询声明机制通常支持的关键字.不过请查阅持久化方式特定的文档以获取受支持关键字的确切列表，因为特定持久化方式可能不支持此处列出的某些关键字.

|  |  |
| --- | --- |
| 逻辑关键字 | 关键字表达 |
| AND | And |
| OR | Or |
| AFTER | After， IsAfter |
| BEFORE | Before， IsBefore |
| CONTAINING | Containing，IsContaining，Contains |
| BETWEEN | Between， IsBetween |
| ENDING\_WITH | EndingWith，IsEndingWith，EndsWith |
| EXISTS | Exists |
| FALSE | False， IsFalse |
| GREATER\_THAN | GreaterThan， IsGreaterThan |
| GREATER\_THAN\_EQUALS | GreaterThanEqual， IsGreaterThanEqual |
| IN | In， IsIn |
| IS | Is、、Equals（或没有关键字） |
| IS\_EMPTY | IsEmpty， Empty |
| IS\_NOT\_EMPTY | IsNotEmpty， NotEmpty |
| IS\_NOT\_NULL | NotNull， IsNotNull |
| IS\_NULL | Null， IsNull |
| LESS\_THAN | LessThan， IsLessThan |
| LESS\_THAN\_EQUAL | LessThanEqual， IsLessThanEqual |
| LIKE | Like， IsLike |
| NEAR | Near， IsNear |
| NOT | Not， IsNot |
| NOT\_IN | NotIn， IsNotIn |
| NOT\_LIKE | NotLike， IsNotLike |
| REGEX | Regex，MatchesRegex，Matches |
| STARTING\_WITH | StartingWith，IsStartingWith，StartsWith |
| TRUE | True， IsTrue |
| WITHIN | Within， IsWithin |

## 附录D：储存库查询返回类型

支持的查询返回类型

下表列出了Spring Datarepository通常支持的返回类型.不过请查阅持久化方式特定的文档以获取受支持的返回类型的确切列表，因为特定持久化方式可能不支持此处列出的某些类型.

|  |  |
| --- | --- |
|  | 地理空间类型（如GeoResult，GeoResults，和GeoPage）是仅适用于支持地理空间查询的数据存储. |
| 返回类型 | 描述 |
| void | 表示没有返回值. |
| 原语 | Java基元. |
| 包装类型 | Java包装器类型. |
| T | 唯一实体.期望查询方法最多返回一个结果.如果未找到结果，null则返回.多个结果触发IncorrectResultSizeDataAccessException. |
| Iterator<T> | 的Iterator. |
| Collection<T> | 一Collection. |
| List<T> | 一List. |
| Optional<T> | Java 8或Guava Optional.期望查询方法最多返回一个结果.如果没有找到结果，Optional.empty()或者Optional.absent()返回结果.多个结果触发IncorrectResultSizeDataAccessException. |
| Option<T> | Scala或Vavr Option类型.语义上与Optional前面描述的Java 8相同. |
| Stream<T> | Java 8 Stream. |
| Streamable<T> | 该便利性的便捷扩展Iterable使方法可以流式处理，映射和过滤结果，将其级联等. |
| 实现Streamable并接受Streamable构造函数或工厂方法参数的类型 | 公开以a 作为参数的构造函数或….of(…)/ ….valueOf(…)工厂方法的类型Streamable.有关详细信息，请参见[返回自定义可流包装器类型](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/" \l "repositories.collections-and-iterables.streamable-wrapper). |
| Vavr Seq，List，Map，Set | Vavr集合类型.有关详细信息，请参见[支持Vavr集合](https://docs.spring.io/spring-data/jpa/docs/2.2.6.RELEASE/reference/html/" \l "repositories.collections-and-iterables.vavr). |
| Future<T> | 一Future.期望使用方法进行注解，@Async并且需要启用Spring的异步方法执行功能. |
| CompletableFuture<T> | Java 8 CompletableFuture.期望使用方法进行注解，@Async并且需要启用Spring的异步方法执行功能. |
| ListenableFuture | 一org.springframework.util.concurrent.ListenableFuture.期望使用方法进行注解，@Async并且需要启用Spring的异步方法执行功能. |
| Slice | 一定大小的数据块，用于指示是否有更多可用数据.需要一个Pageable方法参数. |
| Page<T> | 阿Slice与其他信息，例如结果的总数.需要一个Pageable方法参数. |
| GeoResult<T> | 具有附加信息（例如到参考位置的距离）的结果条目. |
| GeoResults<T> | GeoResult<T>具有其他信息的列表，例如到参考位置的平均距离. |
| GeoPage<T> | 甲Page带GeoResult<T>，如到参考位置的平均距离. |
| Mono<T> | Mono使用反应性储存库发射零或一个元素的项目反应堆.期望查询方法最多返回一个结果.如果未找到结果，Mono.empty()则返回.多个结果触发IncorrectResultSizeDataAccessException. |
| Flux<T> | Flux使用反应性储存库发射零，一个或多个元素的项目反应堆.返回的查询Flux也可以发出无限数量的元素. |
| Single<T> | Single使用反应性repository发出单个元素的RxJava .期望查询方法最多返回一个结果.如果未找到结果，Mono.empty()则返回.多个结果触发IncorrectResultSizeDataAccessException. |
| Maybe<T> | Maybe使用反应性repository发出零或一个元素的RxJava .期望查询方法最多返回一个结果.如果未找到结果，Mono.empty()则返回.多个结果触发IncorrectResultSizeDataAccessException. |
| Flowable<T> | RxJava Flowable使用反应性repository发出零个，一个或多个元素.返回的查询Flowable也可以发出无限数量的元素. |

## 附录E：常见问题

共同

1. *我想获得JpaRepository有关例如内部调用什么方法的更详细的日志记录信息.我如何获得他们？*

您可以利用CustomizableTraceInterceptorSpring提供的，如以下示例所示：

<bean id="customizableTraceInterceptor" class=" org.springframework.aop.interceptor.CustomizableTraceInterceptor"> <property name="enterMessage" value="Entering $[methodName]($[arguments])"/> <property name="exitMessage" value="Leaving $[methodName](): $[returnValue]"/> </bean> <aop:config> <aop:advisor advice-ref="customizableTraceInterceptor" pointcut="execution(public \* org.springframework.data.jpa.repository.JpaRepository+.\*(..))"/> </aop:config>

基础设施

1. *目前，我已经实现了基于的repository层HibernateDaoSupport.我SessionFactory使用Spring的创建了一个AnnotationSessionFactoryBean.如何在这种环境中使用Spring Datarepository？*

您必须将替换AnnotationSessionFactoryBean为HibernateJpaSessionFactoryBean，如下所示：

**例子119.**SessionFactory**从一个a 查找一个**HibernateEntityManagerFactory

<bean id="sessionFactory" class="org.springframework.orm.jpa.vendor.HibernateJpaSessionFactoryBean"> <property name="entityManagerFactory" ref="entityManagerFactory"/> </bean>

稽核

1. *我想使用Spring Data JPA审计功能，但是已经将我的数据库配置为在实体上设置修改和创建日期.如何防止Spring Data以编程方式设置日期.*

将命名空间元素的set-dates属性设置auditing为false.

## 附录F：词汇表

**行动计划**

面向方面的编程

**公用DBCP**

Commons DataBase Connection Pools-来自Apache基础的库，提供DataSource接口的池实现.

**欺诈**

创建，读取，更新，删除-基本持久化操作.

**道**

数据访问对象-用于将持久化逻辑与要持久化的对象分离的模式

**依赖注入**

从外部将组件的依赖关系传递给组件的模式，以释放组件以查找依赖项本身.有关更多信息，请参见[https://zh.wikipedia.org/wiki/Dependency\_Injection](https://en.wikipedia.org/wiki/Dependency_Injection).

**EclipseLink**

实现JPA的对象关系映射器- <https://www.eclipse.org/eclipselink/>

**冬眠**

实现JPA的对象关系映射器- <https://hibernate.org/>

**JPA**

Java持久化API

**弹簧**

Java应用程序框架- <https://projects.spring.io/spring-framework>

JPA :Java persistence API,在Hibernate（目标：Java程序员不懂SQL也可以操作数据库）基础上发展而来

是java持久化【规范】

JDBC 的DAO是面向SQL语句的不是面向对象，100多张表全部都要写SQL

JPA把JDBC做了封装

不论JDBC还是JPA都不适合搞定大数据量处理

JPA有跟多坑，用不好就性能很慢

有20张表则有20个类，20种实例对象，每个对象对应表里的一行数据

JDBC需要考虑缓存数据更新/清理的问题

在国内，是Hibernate和MyBatis的天下

jboss收购了hibernate，所以有的jar以jboss开头命名

方言配置，从这个文件搜索

持久单元名称cn.itsource.jpa在指定实体管理工厂EntityManagerFactory时使用

oracle的第一页分页

自动生成表：

运行结果：

remove:

映射配置

核心API

张三/李四/王五各自属于一个线程

entityManagerFactory创建耗时，而EntityManager不耗时，且非线程安全，每个线程各自创建自己的

一级缓存命中，只查询一次：

一级缓存没有命中，再次查询：

脏数据更新

二级缓存