Spring Data JPA - □□□□

Version 2.3.6.RELEASE, 2021-05-14

Table of Contents

1. 前言 . . . 2
    1.1. 项目元数据 . . . 2
2. 新的&值得注意的 . . . 3
    2.1. Spring Data JPA 1.11中的新特性 . . . 3
    2.2. Spring Data JPA 1.10 中的新特性 . . . 3
3. 依赖 . . . 4
    3.1. Spring Boot 的依赖管理 . . . 4
    3.2. Spring Framework . . . 5
4. 使用 Spring Data Repositories . . . 6
    4.1. 核心概念 . . . 6
    4.2. 查询方法 . . . 7
    4.3. 定义 Repository 接口 . . . 9
        4.3.1. 微调 Repository 定义 . . . 9
        4.3.2. 将 Repositories 与多个 Spring Data 模块一起使用 . . . 10
    4.4. 定义查询方法 . . . 12
        4.4.1. 查询策略 . . . 12
        4.4.2. 查询创建 . . . 12
        4.4.3. 属性表达式 . . . 13
        4.4.4. 特殊参数处理 . . . 14
            Paging 和 Sorting . . . 15
        4.4.5. 限制查询结果 . . . 15
        4.4.6. 存储库方法返回集合或可迭代对象 . . . 16
            使用 Streamable 作为查询方法返回类型 . . . 16
            返回自定义 Streamable 包装器类型 . . . 16
            支持 Vavr 集合 . . . 17
        4.4.7. 存储库方法的空处理 . . . 18
            可空性注解 . . . 18
            基于 Kotlin 的存储库中的可空性 . . . 19
        4.4.8. 流查询结果 . . . 19
        4.4.9. 异步查询结果 . . . 20
    4.5. 创建存储库实例 . . . 21
        4.5.1. XML 配置 . . . 21
            使用过滤器 . . . 21
        4.5.2. Java 配置 . . . 21
        4.5.3. 独立使用 . . . 22
    4.6. Spring Data 存储库的自定义实现 . . . 22
        4.6.1. 自定义单个存储库 . . . 22
            配置 . . . 25

4.6.2. 自定义基础存储库 . . . 26
4.7. 从聚合根发布事件 . . . 27
4.8. Spring Data 扩展 . . . 28
4.8.1. Querydsl 扩展 . . . 28
4.8.2. Web 支持 . . . 29
Basic Web 支持 . . . 30
超媒体对页面的支持 . . . 32
Web 数据绑定支持 . . . 34
Querydsl Web 支持 . . . 34
4.8.3. 存储库填充器 . . . 36
5. 参考文档 . . . 38
6. JPA 存储库 . . . 39
6.1. 简介 . . . 39
6.1.1. Spring 命名空间 . . . 39
自定义命名空间属性 . . . 39
6.1.2. 基于注解的配置 . . . 40
6.1.3. 引导模式 . . . 41
推荐建议 . . . 41
6.2. 持久化实体 . . . 41
6.2.1. 保存实体 . . . 41
实体状态检测策略 . . . 41
6.3. 查询方法 . . . 42
6.3.1. 查询策略 . . . 42
声明查询 . . . 42
6.3.2. 查询创建 . . . 43
6.3.3. 使用 JPA 命名查询 . . . 44
XML 命名查询定义 . . . 44
基于注解的配置 . . . 44
声明接口 . . . 45
6.3.4. 使用 `@Query` . . . 45
使用高级 `LIKE` 表达式 . . . 46
本地查询 . . . 46
6.3.5. 使用排序 . . . 47
6.3.6. 使用命名参数 . . . 47
6.3.7. 使用 SpEL 表达式 . . . 48
6.3.8. 修改查询 . . . 50
扩展删除查询 . . . 50
6.3.9. 应用查询提示 . . . 51
6.3.10. 配置 Fetch- 和 LoadGraphs . . . 51
6.3.11. 投影 . . . 52
基于接口的投影 . . . 53

基于类的投影 (DTO) . . . 56
动态投影 . . . 57
6.4. 存储过程 . . . 57
6.5. Specification . . . 59
6.6. 查询例子 . . . 61
6.6.1. 简介 . . . 61
6.6.2. 使用 . . . 61
6.6.3. Example 匹配 . . . 62
6.6.4. 执行一个例子 . . . 64
6.7. 事务性 . . . 65
6.7.1. 事务查询方法 . . . 66
6.8. 锁 . . . 67
6.9. 审计 . . . 68
6.9.1. 基础 . . . 68
基于注解的审核元数据 . . . 68
基于接口的审核元数据 . . . 68
AuditorAware . . . 68
6.9.2. JPA 审计 . . . 69
通用审核配置 . . . 69
6.10. 其他注意事项 . . . 70
6.10.1. 在自定义实现中使用 JpaContext . . . 70
6.10.2. 合并持久性单元 . . . 71
@Entity 类和 JPA 映射文件的类路径扫描 . . . 71
6.10.3. CDI 集成 . . . 72
7. 附录 . . . 75
Appendix A: 命名空间参考 . . . 76
<repositories /> 元素 . . . 76
Appendix B: Populators 命名空间参考 . . . 77
<populator /> element . . . 77
Appendix C: 存储库查询关键字 . . . 78
支持的查询方法主题关键字 . . . 78
支持的查询方法断言关键字和修饰符 . . . 78
Appendix D: 储存库查询返回类型 . . . 80
支持的查询返回类型 . . . 80
Appendix E: 常见问题 . . . 82
通用 . . . 82
基础 . . . 82
审计 . . . 82
Appendix F: 词汇表 . . . 83

□□□□□□□□□□□□□□□,□□□□□□□□□□,□□□□□□□□□□□□□□□□□□□,□□□□□□□□□□□□□□□□□□□(□□□□□□□□□□□□□□□□) .

Chapter 1. 前言

Spring Data JPA 为 Java 持久性 API(JPA) 提供存储库支持。它简化了需要访问 JPA 数据源的应用程序的开发。

1.1. 项目元数据

- 版本控制 - github.com/spring-projects/spring-data-jpa
- Bugtracker - jira.spring.io/browse/DATAJPA
- 发行版本库 - repo.spring.io/libs-release
- 里程碑存储库 - repo.spring.io/libs-milestone
- 快照存储库 - repo.spring.io/libs-snapshot

Chapter 2. 新的＆值得注意的

2.1. Spring Data JPA 1.11中的新特性

Spring Data JPA 1.11 增加了以下功能:

- 改进与 Hibernate 5.2 的兼容性.
- 通过示例支持任意匹配模式. .
- 分页查询执行优化.
- 支持 `exists` 存储库查询扩展中的投影.

2.2. Spring Data JPA 1.10 中的新特性

Spring Data JPA 1.10 增加了以下功能:

- 支持存储库查询方法中的 Projections(投影).
- 通过示例支持查询.
- 已启用以下注解构建组合注解: `@EntityGraph`, `@Lock`, `@Modifying`, `@Query`, `@QueryHints`, 和 `@Procedure`.
- 支持 `Contains` 集合表达式的关键字.
- JSR-310 和 ThreeTenBP `ZoneId` 的 `AttributeConverter` 实现.
- 升级到 Querydsl 4, Hibernate 5, OpenJPA 2.4, 和 EclipseLink 2.6.1.

Chapter 3. 依赖

由于各个 Spring Data 模块的初始日期不同,它们中的大多数都带有不同的主版本号和次版本号. 寻找兼容版本的最简单方法是依靠我们随定义的兼容版本提供的 Spring Data Release BOM. 在 Maven 项目中,您将在 `<dependencyManagement />` POM 的部分声明这种依赖,如下所示:

Example 1. 使用 Spring Data 发行版 BOM

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.data</groupId>
      <artifactId>spring-data-releasetrain</artifactId>
      <version>Moore-SR8</version>
      <scope>import</scope>
      <type>pom</type>
    </dependency>
  </dependencies>
</dependencyManagement>
```

目前的发行版本是 `Moore-SR8`. 名称按字母顺序上升,目前可用的 列车在 此处 列出. 版本名称遵循以下模式: `${name}-${release}`,其中发布可以是下列之一:

- `BUILD-SNAPSHOT`: 当前快照
- `M1`, `M2`, 等: 里程碑
- `RC1`, `RC2`, 发布候选
- `RELEASE`: GA 版本
- `SR1`, `SR2`, 等: 服务版本

在我们的 Spring Data 示例存储库 中可以找到使用 BOM 的一个工作示例. 有了这个,你可以在你的模块中声明 Spring Data 模块而不需要版本 `<dependencies />`,如下所示:

Example 2. 声明一个依赖 Spring Data 模块

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

3.1. Spring Boot 的依赖管理

Spring Boot 为您选择最新版本的 Spring Data 模块. 如果您仍想升级到较新版本,请将该属性配置为您要使用 `spring-data-releasetrain.version` 的 火车名称迭代 .

3.2. Spring Framework

当前版本的 Spring Data 模块需要版本 5.2.12.RELEASE 或更高版本的 Spring Framework. 这些模块也可能与该次要版本的旧版错误修复版本一起工作. 但是,强烈建议使用该代中的最新版本.

Chapter 4. 使用 Spring Data Repositories

Spring Data 存储库抽象层的目标是减少为各种持久性存储实现数据访问所需的样板代码数量.

*Spring*数据存储库文档和你的模块

本章介绍 Spring Data 存储库的核心概念和接口. 本章中的信息来自 Spring Data Commons 模块. 它使用 Java 持久性 API (JPA) 模块的配置和代码示例. 您应该将 XML 命名空间声明和要扩展的类型调整为您使用的特定模块的等同项. "命名空间参考" 涵盖了所有支持存储库API的 Spring Data 模块支持的XML配置. "存储库查询关键字" 一般涵盖了存储库抽象支持的查询方法关键字. 有关模块特定功能的详细信息,请参阅本文档的该模块章节.

4.1. 核心概念

Spring Data 存储库抽象中的中心接口是 `Repository`. 它需要 domain 类以及 domain 的 ID 类型作为类型参数. 该接口主要作为标记接口来捕获要使用的类型,并帮助您发现该接口的子接口. `CrudRepository` 实现了实体类复杂的 CRUD 功能.

Example 3. `CrudRepository` 接口

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

  <S extends T> S save(S entity);      ①

  Optional<T> findById(ID primaryKey); ②

  Iterable<T> findAll();               ③

  long count();                        ④

  void delete(T entity);               ⑤

  boolean existsById(ID primaryKey);   ⑥

  // … more functionality omitted.
}
```

① 保存给定的实体.

② 返回由给定ID标识的实体.

③ 返回所有实体.

④ 返回实体的数量.

⑤ 删除给定的实体.

⑥ 判断是否存在具有给定ID的实体.

我们还提供持久性技术的特定抽象,如 `JpaRepository` 或 `MongoRepository`. 这些接口扩展 `CrudRepository` 并暴露了持久化技术的基本功能,以及通用的持久化技术,例如 `CrudRepository`.

除此之外 `CrudRepository`,还有一个 `PagingAndSortingRepository` 的抽象的接口,来简化对实体的分页操作:

Example 4. `PagingAndSortingRepository` 接口

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

  Iterable<T> findAll(Sort sort);

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

要访问 `User` 第二页,每页 20,您可以执行以下操作:

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

除了查询方法外,count 和 delete 查询的相关扩展都是可用的. 以下列表扩展 count 查询的接口定义:

Example 5. 扩展 *count* 查询

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

  long countByLastname(String lastname);
}
```

以下列表扩展 delete 查询的接口定义:

Example 6. 扩展删除查询

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

  long deleteByLastname(String lastname);

  List<User> removeByLastname(String lastname);
}
```

4.2. 查询方法

标准CRUD功能通常会在底层数据存储上进行查询. 使用 Spring Data,声明这些查询需要四步:

1. 声明一个扩展 `Repository` 或其子接口的接口,并输入它应该处理的 domain 类和 `ID` 类型,如以下示例所示:

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

2. 在接口上声明查询方法.

```
interface PersonRepository extends Repository<Person, Long> {
  List<Person> findByLastname(String lastname);
}
```

3. 使用 Spring JavaConfig 或 XML 配置 为这些接口创建代理实例.

   a. 要使用 Java 配置,请创建类似于以下的类:

```
import
org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@EnableJpaRepositories
class Config { … }
```

   b. 要使用XML配置,请定义一个类似于以下的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.acme.repositories"/>

</beans>
```

   这个例子中使用了 JPA 命名空间. 如果您想使用其他的存储库,则需要将其更改为相应的命名空间声明. 换句话说,你应该替换 jpa,例如 mongodb. 另请注意,JavaConfig 不会显式配置包,因为默认情况下会使用注解类的包. 定制要扫描的软件包,请使用 basePackage… 特定于数据存储库的 @Enable${store}Repositories 注解的一个属性.

4. 注入资源库实例并使用它,如以下示例所示:

```
class SomeClient {

  private final PersonRepository repository;

  SomeClient(PersonRepository repository) {
    this.repository = repository;
  }

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

以下部分详细解释每一步:

- 定义 Repository 接口
- 定义查询方法
- 创建存储库实例
- Spring Data Repository 的自定义实现

4.3. 定义 Repository 接口

首先,定义一个 domain 类特定的 repository 接口. 该接口必须扩展 `Repository` 并且输入 domain 类和 `ID` 类型. 如果您想暴露该 domain 类型的 CRUD 方法,请扩展 `CrudRepository` 而不是 `Repository`.

4.3.1. 微调 Repository 定义

通常情况下,您的 Repository 接口扩展了 `Repository`,`CrudRepository` 或 `PagingAndSortingRepository`. 如果您不想扩展 Spring Data 接口,也可以使用 `@RepositoryDefinition` 注解您的 Repository 接口. 扩展 `CrudRepository` 暴露了一套完整的方法来操纵你的实体. 如果您想选择暴露的方法,请复制 `CrudRepository` 中要暴露的方法 到您的实体类 Repository 中.

这样做可以让您在提供的 Spring Data Repositories 功能之上定义自己的抽象.

以下示例显示如何选择性地暴露 CRUD方法 (`findById` 以及 `save` 在这种情况下) :

Example 7. 选择性地暴露 *CRUD* 方法

```
@NoRepositoryBean
interface MyBaseRepository<T, ID> extends Repository<T, ID> {

  Optional<T> findById(ID id);

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

interface UserRepository extends MyBaseRepository<User, Long> {
  User findByEmailAddress(EmailAddress emailAddress);
}
```

在前面的示例中,您为所有 domain Repository 定义了一个通用的基本接口,并暴露了 `findById(…)` 和 `save(…)` 方法. 这些方法被路由到 Spring Data 提供的所选存储的基本存储库实现中 (例如,如果使用JPA,则实现为 `SimpleJpaRepository`,因为它们与 `CrudRepository` 中的方法签名匹配. 因此,`UserRepository` 现在可以保存用户,通过ID查找单个用户,并触发查询以通过电子邮件地址查找 `Users`.

中间的 repository 接口用 `@NoRepositoryBean` 注解. 确保添加了该注解的 repository 接口不会在 Spring Data 运行时被创建实例.

4.3.2. 将 Repositories 与多个 Spring Data 模块一起使用

在您的应用程序中使用唯一的 Spring Data 模块很简单,因为已定义范围中的所有存储库接口均已绑定到该 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,存储库定义必须区分使用哪个. 当它在类路径上检测到多个存储库工厂时,Spring Data 进入严格的存储库配置模式. 严格的配置使用 repository 或 domain 类上的详细信息来决定有关存储库定义的 Spring Data 模块绑定:

1. 如果存储库定义扩展了特定于模块的存储库,则它是特定 Spring Data 模块的有效候选者.
2. 如果 domain 类使用模块特定的注解类型进行注解,则它是特定 Spring Data 模块的有效候选者. Spring Data 模块可以接受第三方注解 (例如JPA的 `@Entity`) ,也可以提供自己的注解 (例如 Spring Data MongoDB的 `@Document` 和 Spring Data Elasticsearch) .

以下示例显示使用特定于模块的接口 (在这种情况下为 JPA) 的存储库:

Example 8. 使用模块特定接口的存储库定义

```
interface MyRepository extends JpaRepository<User, Long> { }

@NoRepositoryBean
interface MyBaseRepository<T, ID> extends JpaRepository<T, ID> { … }

interface UserRepository extends MyBaseRepository<User, Long> { … }
```

`MyRepository` 和 `UserRepository` 继承 `JpaRepository` . 它们是 Spring Data JPA 模块的有效候选者.

下面的例子展示了一个使用通用接口的存储库:

Example 9. 使用通用接口的存储库定义

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

@NoRepositoryBean
interface MyBaseRepository<T, ID> extends CrudRepository<T, ID> { … }

interface AmbiguousUserRepository extends MyBaseRepository<User, Long> { … }
```

`AmbiguousRepository` 和 `AmbiguousUserRepository` 仅在其类型层次结构中扩展 `Repository` 和 `CrudRepository`. 尽管在使用唯一的 Spring Data 模块时没有什么问题,但是多个模块无法区分这些存储库应绑定到哪个特定的 Spring Data.

以下示例显示了使用带注解的 domain 类的存储库:

Example 10. 使用带有注解的 *domain* 类的存储库定义

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

@Entity
class Person { … }

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

@Document
class User { … }
```

`PersonRepository` 引用使用JPA `@Entity` 注解进行注解的 `Person`,因此该存储库显然属于 Spring Data JPA. `UserRepository` 引用 `User`,该 User 使用 Spring Data MongoDB 的 `@Document` 注解进行注解.

以下错误的示例显示了使用带有混和注解的 domian 类的存储库:

Example 11. 使用带有混合注解的 *domain* 类的存储库定义

```
interface JpaPersonRepository extends Repository<Person, Long> { … }

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

@Entity
@Document
class Person { … }
```

此示例显示了同时使用 JPA 和 Spring Data MongoDB 注解的 domain 类. 它定义了两个存储库,`JpaPersonRepository` 和 `MongoDBPersonRepository`.

存储库类型详细信息 和可区分的 domain 类注解 用于配置严格的存储库,以标识特定 Spring Data 模块的存储库候选者. 在同一个 domain

类型上使用多个特定于持久性技术的注解是可能的,并且可以跨多种持久性技术重用 domain 类型。但是,Spring Data 无法再确定用于绑定存储库的唯一模块。

区分存储库的最后一种方法是确定存储库 `basePackages` 的范围。 `basePackages` 包定义了扫描存储库接口定义的起点,这意味着将存储库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 的配置中中的 `basePackages` 是必需的。

以下示例显示了基础包的注解驱动配置:

Example 12. `basePackages` 的注解驱动配置

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

4.4. 定义查询方法

存储库代理有两种从方法名称扩展特定查询的方式:

- 通过直接从方法名称扩展查询。
- 通过使用手动定义的查询

可用选项取决于实际存储。但是,必须有一种策略可以决定要创建的实际查询。下一节将介绍可用的选项。

4.4.1. 查询策略

以下策略可用于存储库基础结构来解决查询。 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 `Enable${store}Repositories` 注解的 `queryLookupStrategy` 属性。某些数据存储可能不支持某些策略。

- `CREATE` 尝试从查询方法名称构造特定于存储的查询。通用方法是从方法名称中删除一组给定的前缀,然后解析该方法的其余部分。您可以在 "查询创建" 中阅读有关查询构造的更多信息。
- `USE_DECLARED_QUERY` 尝试查找已声明的查询,如果找不到则抛出异常。 该查询可以通过某处的注解定义,也可以通过其他方式声明。 请查阅特定存储的文档以找到该存储方式的可用选项。如果在查询时找不到该方法的声明查询,则它将失败。
- `CREATE_IF_NOT_FOUND` (默认) 结合 `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,则使用该策略。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调整这些查询。

4.4.2. 查询创建

Spring Data 内置的查询机制对于在存储库实体上构建查询约束很有用。 该机制的前缀 `find…By`, `read…By`, `query…By`, `count…By`, 和 `get…By` 从所述方法和开始解析它的其余部分。`Introduction` 子句可以包含其他表达式, 例如,`Distinct` 以在要创建的查询上设置不同的标志。但是,第一个 `By` 充当分隔符以指示实际查询的开始。在此级别上,您可以定义实体属性的条件,并将其与 `And` 和 `Or` 串联 。下面的示例演示如何创建许多查询:

Example 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);
}
```

解析查询方法的名称分为主题和断言.第一部分 (`find…By`, `exists…By`) 定义查询的主题,第二部分形成断言. Introduction子句 (主题) 可以包含其他表达式. 任何在 `find` (或其他 introducing 关键字) 和 `By` 之间的任何文本都被视为描述性语句,除非使用结果限制关键字之一 (例如,`Distinct`) 在要创建的查询上设置 distinct 的标记,或者 `Top` /`First` 以限制查询结果.

附录包含 查询方法主题关键字的完整列表 和 包括排序和字母大小写修饰符的查询方法断言关键字. 但是,第一个 `By` 充当分隔符,以指示实际标准断言的开始. 在此基础上,您可以定义实体属性的条件,并将其与 `And` 和 `Or` 连接起来.

解析该方法的实际结果取决于您为其创建查询的持久性存储. 但是,需要注意一些一般事项:

- 表达式通常是属性遍历,并带有可串联的运算符. 您可以将属性表达式与 `AND` 和 `OR` 结合使用. 您还将获得属性表达式的支持,例如 `between`,`LessThan`,`GreaterThan` 和 `Like`. 支持的运算符可能因数据存储而异,因此请参考参考文档的相应部分.
- 方法解析器支持为单个属性 (例如,`findByLastnameIgnoreCase(…)`) 或支持忽略大小写的类型的所有属性 (通常为 `String` 实例,例如,`findByLastnameAndFirstnameAllIgnoreCase(…)`) 设置 `IgnoreCase` 标志. 是否支持忽略大小写可能因存储而异,因此请参考参考文档中有关存储特定查询方法的相关部分.
- 您可以通过将 `OrderBy` 子句附加到引用属性的查询方法并提供排序方向 (`Asc` 或 `Desc`) 来应用静态排序. 要创建支持动态排序的查询方法,请参见 “特殊参数处理”.

4.4.3. 属性表达式

如上例所示,属性表达式只能引用实体的直接属性. 在查询创建时,您需要确保已解析的属性是被管理 domain 类的属性. 但是,您也可以通过遍历嵌套属性来定义约束. 考虑以下方法签名:

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

假设一个 `Person`（人）的 `Address`（地址）带有 `ZipCode`（邮政编码）。在这种情况下，该方法将创建遍历属性 `x.address.zipCode`。解析算法首先将整个部分（`AddressZipCode`）解释为属性，然后在 domain 类中检查具有该名称的属性（未大写）。如果算法成功，它将使用该属性。如果不是，该算法将按驼峰解析为头和尾，并尝试找到对应的属性，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，以刚才描述的方式将尾部向上拆分。如果第一个拆分不匹配，则算法会将拆分点移到左侧（`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` 类也具有 `addressZip` 属性。该算法将在第一轮拆分中匹配，选择错误的属性，然后失败（因为 `addressZip` 的类型可能没有 `code` 属性）。

要解决这种歧义，您可以在方法名称中使用 `_` 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，所以我们强烈建议您遵循以下标准 Java 命名约定（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，如前面的示例所示。除此之外，基本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您的查询。以下示例演示了这些功能：

Example 14. 在查询方法中使用 `Pageable`，`Slice`，和 `Sort`

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

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

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

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

❗ 采用 `Sort` 和 `Pageable` 的 API 期望将非 `null` 值传递到方法中。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。页面知道可用元素和页面的总数。它是通过基础结构触发计数查询来计算总数来实现的。由于这可能很耗时（取决于所使用的存储），因此您可以返回一个 `Slice`。切片仅知道下一个切片是否可用，当遍历较大的结果集时这可能就足够了。

排序选项也通过 `Pageable` 实例处理。如果只需要排序，则将 `org.springframework.data.domain.Sort` 参数添加到您的方法中。如您所见，返回列表也是可能的。在这种情况下，将不会创建构建实际的 `Page` 实例所需的其他元数据（这反过来意味着不会发出本来必要的其他计数查询）。而是，它将查询限制为仅查找给定范围的实体。

ℹ 要找出整个查询可获得多少页，您必须触发其他计数查询。默认情况下，此查询扩展自您实际触发的查询。

Paging 和 Sorting

可以使用属性名称定义简单的排序表达式. 可以将表达式连接起来,以将多个条件收集到一个表达式中.

Example 15. 定义排序表达式

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

对于排序表达式的类型安全性更高的方法,请从该类型开始为定义排序表达式,然后使用方法引用来定义要进行排序的属性.

Example 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());
```

ℹ `TypedSort.by(…)` 通过 (通常) 使用 CGlib 来使用运行时代理,这在使用 Graal VM Native 等工具时可能会影响本地镜像的编译.

如果您的存储实现支持 Querydsl,则还可以使用生成的元模型类型来定义排序表达式:

Example 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` 关键字来限制查询方法的结果,这些关键字可以互换使用. 可以在 `top` 或 `first` 附加可选的数值,以指定要返回的最大结果大小. 如果省略数字,则假定结果大小为 `1`. 以下示例显示了如何限制查询大小:

Example 18. 使用 `first` 和 `top` 限制查询的结果大小

```
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);
```

限制表达式还支持 `Distinct` 关键字. 另外,对于将结果集限制为一个实例的查询,支持使用 `Optional` 关键字将结果包装到其中.

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

> 通过使用 `Sort` 参数将结果限制与动态排序结合使用,可以让您表达对最小的 "K" 元素和对 "K" 的最大元素的查询方法.

4.4.6. 存储库方法返回集合或可迭代对象

返回多个结果的查询方法可以使用标准的 Java `Iterable`,`List`,`Set`. 除此之外,我们还支持返回 Spring Data 的 `Streamable`,`Iterable` 的自定义扩展以及 Vavr 提供的集合类型.请参阅附录,了解所有可能的 查询方法返回类型.

使用 Streamable 作为查询方法返回类型

`Streamable` 可用作 `Iterable` 或任何集合类型的替代. 它提供了方便的方法来访问非并行流 (缺少 `Iterable`) ,可以直接在元素上进行 `….filter(…)` 和 `….map(…)` 并将 `Streamable` 连接到其他元素:

Example 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.findByLastnameContaining("ea"));
```

返回自定义 Streamable 包装器类型

为集合提供专用的包装器类型是一种常用的模式,用于在返回多个元素的查询执行结果上提供 API. 通常,这些类型是通过调用存储库方法来返回类似集合的类型并手动创建包装类型的实例来使用的. 如果 Spring Data 满足以下条件,则可以将这些包装器类型用作查询方法返回类型,因此可以避免执行附加步骤:

1. 该类型实现 `Streamable`.

2. 该类型以 `Streamable` 作为参数暴露构造函数或名为 `of(…)` 或 `valueOf(…)` 的静态工厂方法。

示例用例如下所示：

```
class Product {                                       ①
  MonetaryAmount getPrice() { … }
}

@RequiredArgConstructor(staticName = "of")
class Products implements Streamable<Product> {       ②

  private Streamable<Product> streamable;

  public MonetaryAmount getTotal() {                  ③
    return streamable.stream()
      .map(Priced::getPrice)
      .reduce(Money.of(0), MonetaryAmount::add);
  }


  @Override
  public Iterator<Product> iterator() {               ④
    return streamable.iterator();
  }
}

interface ProductRepository implements Repository<Product, Long> {
  Products findAllByDescriptionContaining(String text); ⑤
}
```

① 暴露 API 以访问产品价格的 `Product` 实体。

② 可以通过 `Products.of(…)` (通过 Lombok 注解创建的工厂方法) 构造的 `Streamable<Product>` 的包装器类型。

③ 包装器类型在 `Streamable<Product>` 上暴露其他用于计算新值的API。

④ 实现 `Streamable` 接口并且委托给实际结果。

⑤ 该包装器类型 `Products` 可以直接用作查询方法返回类型。无需返回 `Stremable<Product>` 并在查询之后将其手动包装在存储库客户端中。

支持 **Vavr** 集合

Vavr 是一个包含 Java 中函数式编程概念的库。它附带一组可作为查询方法返回类型使用的自定义集合类型。

Vavr 集合类型	使用 **Vavr** 实现类型	验证 **Java source** 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`（等效于 Vavr `Iterable`）,然后从实际返回值扩展实现类,即 `java.util.List` 将变成 Vavr `List`/`Seq`,而 `java.util.Set` 变为Vavr `LinkedHashSet`/`Set` 等

4.4.7. 存储库方法的空处理

从 Spring Data 2.0 开始,返回单个聚合实例的存储库 CRUD 方法使用 Java 8 的 `Optional` 来指示可能缺少值。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`

另外,查询方法可以选择不使用包装器类型。然后,通过返回 `null` 指示查询结果不存在。保证返回集合,集合替代项,包装器和流的存储库方法永远不会返回 `null`,而是会返回相应的空表示形式。有关详细信息,请参见 “存储库查询返回类型” 。

可空性注解

您可以使用 Spring Framework 的可空性注解 来表达存储库方法的可空性约束。它们提供了一种工具友好的方法,并在运行时提供了选择加入的 `null` 检查,如下所示:

- `@NonNullApi`: 在包级别用于声明参数和返回值的默认行为是不为空值。
- `@NonNull`:用于不为空的参数或返回值（`@NonNullApi` 适用的参数和返回值不需要）.
- `@Nullable`: 用于可以为空的参数或返回值。

Spring 注解使用 JSR 305 注解进行元注解。JSR 305 元注解使工具供应商（如 IDEA, Eclipse 和 Kotlin ）以通用方式提供了空安全支持,而不必对 Spring 注解进行硬编码支持。要对查询方法的可空性约束进行运行时检查,您需要使用 `package-info.java` 中的 Spring 的 `@NonNullApi` 在包级别激活非可空性,如以下示例所示:

Example 20. 在 `package-info.java` 中声明不可为空

```
@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存储库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。当方法将返回 `null` 但被声明为不可为 `null` 时（在存储库所在的包中定义了注解的默认值）,就会发生这种情况。如果要再次选择接受可为空的结果,请在各个方法上有选择地使用 `@Nullable`。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少的值。

下面的示例显示了刚才描述的许多技术:

Example 21. 使用不同的可空性约束

```
package com.acme;                                                   ①

import org.springframework.lang.Nullable;

interface UserRepository extends Repository<User, Long> {

  User getByEmailAddress(EmailAddress emailAddress);                ②

  @Nullable
  User findByEmailAddress(@Nullable EmailAddress emailAdress);      ③

  Optional<User> findOptionalByEmailAddress(EmailAddress emailAddress); ④
}
```

① 存储库位于我们上面定义的非空的包 (或子包) 中.

② 当执行的查询未产生结果时,抛出 `EmptyResultDataAccessException`. 当传递给该方法的 `emailAddress` 为 `null` 时,抛出 `IllegalArgumentException`.

③ 当执行的查询不产生结果时,返回 `null`. 还接受 `null` 作为 `emailAddress` 的值.

④ 当执行的查询不产生结果时,返回 `Optional.empty()`. 当传递给该方法的 `emailAddress` 为 `null` 时,抛出 `IllegalArgumentException`.

基于 **Kotlin** 的存储库中的可空性

Kotlin 定义了语言中包含的 可空性约束 . Kotlin 代码编译为字节码,字节码不通过方法签名来表达可空性约束,而是通过内置的元数据来表达. 请确保在您的项目中包含 `kotlin-reflect` 的JAR,以对 Kotlin 的可空性约束进行自省. Spring Data 存储库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:

Example 22. 在 *Kotlin repository* 上使用可空性约束

```
interface UserRepository : Repository<User, String> {

  fun findByUsername(username: String): User     ①

  fun findByFirstname(firstname: String?): User? ②
}
```

① 该方法将参数和结果都定义为不可为空 (Kotlin 默认值) . Kotlin 编译器拒绝将 `null` 传递给方法的方法调用. 如果查询执行产生空结果,则抛出 `EmptyResultDataAccessException`.

② 此方法的 `firstname` 参数接受 `null`,如果查询执行未产生结果,则返回 `null`.

4.4.8. 流查询结果

可以使用 Java 8 `Stream<T>` 作为返回类型来递增地处理查询方法的结果. 并非将查询结果包装在 `Stream` 中,而是使用特定于数据存储的方法来执行流传输,如以下示例所示:

Example 23. 用Java 8 Stream<T> 进行流处理

```
@Query("select u from User u")
Stream<User> findAllByCustomQueryAndStream();

Stream<User> readAllByFirstnameNotNull();

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

Stream 可能会包装底层数据存储特定的资源,因此必须关闭. 可以手动调用 close() 方法或使用 Java 7 try-with-resources 块来关闭 Stream,如下所示:

Example 24. 在 Stream<T> 结果上使用 try-with-resources 块

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

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

4.4.9. 异步查询结果

使用 Spring 的异步方法执行功能,可以异步执行存储库查询. 这意味着,在调用存储库上的方法时,将立即返回 Spring TaskExecutor 的结果. 使用异步执行时,可以使用 Java 8 的 CompletableFuture,Spring 的 ListenableFuture 或 Google Guava 的 ListenableFuture. 例如:

```
@Async
Future<User> findByFirstname(String firstname);               ①

@Async
CompletableFuture<User> findOneByFirstname(String firstname); ②

@Async
ListenableFuture<User> findOneByLastname(String lastname);    ③
```

① 使用 java.util.concurrent.Future 作为返回类型.

② 使用 Java 8 java.util.concurrent.CompletableFuture 作为返回类型.

③ 使用 org.springframework.util.concurrent.ListenableFuture 作为返回类型.

4.5. 创建存储库实例

在本部分中,将为已定义的存储库接口创建实例和 Bean 定义. 一种方法是使用支持存储库机制的每个 Spring Data 模块随附的 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,可用于定义 Spring 为其扫描的基本包,如以下示例所示:

Example 25. 通过 XML 启用 Spring Data repository

```
<?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 `Repository` 的接口或其子接口之一. 对于找到的每个接口,基础结构都会注册持久性技术特定的 `FactoryBean`,以创建处理查询方法调用的适当代理. 每个 bean 都使用从接口名称扩展的 bean 名称进行注册,因此 `UserRepository` 的接口将注册在 `userRepository` 下. `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,Spring Data 会自动扫描配置路径下的 `Repository` 子接口的每个接口,并为其创建一个 bean 实例. 但是,您可能希望更精细地控制哪些接口具有为其创建的 Bean 实例. 为此,请在 `<repositories />` 元素内使用 `<include-filter />` 和 `<exclude-filter />` 元素. 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 .

例如,要将某些接口从实例中排除为存储库 Bean,可以使用以下配置:

Example 26. 使用 exclude-filter 元素

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

前面的示例将所有以 `SomeRepository` 结尾的接口都排除在实例化之外.

4.5.2. Java 配置

还可以在 JavaConfig 类上使用特定于存储的 `@Enable${store}Repositories` 注解来触发存储库基础架构. 有关 Spring 容器的基于 Java

的配置的介绍,请参见 Spring 参考文档中的 JavaConfig.

Example 27. 基于注解的存储卡示例

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

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

ℹ 前面的示例使用特定于 JPA 的注解,您将根据实际使用的存储模块对其进行更改. 这同样适用于 `EntityManagerFactory` bean 的定义. 请参阅涵盖存储特定配置的部分.

4.5.3. 独立使用

您还可以在 Spring 容器之外使用存储库基础结构,例如在 CDI 环境中. 您的类路径中仍然需要一些 Spring 库,但是,通常,您也可以通过编程方式来设置存储库. 提供存储库支持的 Spring Data 模块附带了特定于持久性技术的 `RepositoryFactory`,您可以按以下方式使用它:

Example 28. repository 工厂的独立使用

```
RepositoryFactorySupport factory = … // Instantiate factory here
UserRepository repository = factory.getRepository(UserRepository.class);
```

4.6. Spring Data 存储库的自定义实现

本节介绍存储库定制以及片段如何形成复合存储库.

当查询方法需要不同的行为或无法通过查询扩展实现时,则有必要提供自定义实现. Spring Data 存储库使您可以提供自定义存储库代码,并将其与通用 CRUD 抽象和查询方法功能集成.

4.6.1. 自定义单个存储库

要使用自定义功能丰富存储库,必须首先定义一个接口和自定义功能的实现,如以下示例所示:

Example 29. 定制 *repository* 功能的接口

```
interface CustomizedUserRepository {
  void someCustomMethod(User user);
}
```

Example 30. 自定义存储库功能的实现

```
class CustomizedUserRepositoryImpl implements CustomizedUserRepository {

  public void someCustomMethod(User user) {
    // Your custom implementation
  }
}
```

与这个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,可以让您的存储库接口扩展此接口,如以下示例所示:

Example 31. 更改您的存储库接口

```
interface UserRepository extends CrudRepository<User, Long>,
CustomizedUserRepository {

  // Declare query methods here
}
```

用存储库接口扩展此接口,将 CRUD 和自定义功能结合在一起,并使它可用于客户端.

Spring Data 存储库是通过使用构成存储库组成的片段来实现的. 片段是基础存储库,功能方面 (例如 QueryDsl) 以及自定义接口及其实现. 每次向存储库接口添加接口时,都通过添加片段来增强组合. 每个 Spring Data 模块都提供了基础存储库和存储库方面的实现.

以下示例显示了自定义接口及其实现:

Example 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 `CrudRepository` 的自定义存储库的接口:

Example 33. 更改您的存储库接口

```
interface UserRepository extends CrudRepository<User, Long>, HumanRepository,
ContactRepository {

  // Declare query methods here
}
```

存储库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存储库方面的优先级更高. 通过此顺序,您可以覆盖基础存储库和方面方法,并在两个片段贡献相同方法签名的情况下解决歧义. 存储库片段不限于在单个存储库界面中使用. 多个存储库可以使用片段接口,使您可以跨不同的存储库重用自定义项.

以下示例显示了存储库片段及其实现:

Example 34. 覆盖 *Fragments* `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存储库片段的存储库:

Example 35. 定制的存储库接口

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

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

配置

如果使用命名空间配置,则存储库基础结构会尝试通过扫描发现存储库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存储库和为后缀设置自定义值的存储库:

Example 36. 配置示例

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

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

上一示例中的第一个配置尝试查找一个名为 `com.acme.repository.CustomizedUserRepositoryImpl` 的类,以用作自定义存储库实现. 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.

给定前面显示的 `CustomizedUserRepository` 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`,它与片段接口 (`CustomizedUserRepository`) 加上后缀 `Impl` 的名称匹配.

Example 37. 解决歧义的实现

```
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 `@Component("specialCustom")` 注解 `UserRepository` 接口,则 Bean 名称加 `Impl` 会与 `com.acme.impl.two` 中为存储库实现匹配定义一个,并使用它代替第一个.

手动织入

如果您的自定义实现仅使用基于注解的配置和自动装配,则上述显示的方法会很好地起作用,因为它被视为其他任何 Spring Bean. 如果实现片段 bean 需要特殊的拼接,则可以声明 bean 并根据上一节中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:

Example 38. 手动织入自定义实现

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

<beans:bean id="userRepositoryImpl" class="…">
  <!-- further configuration -->
</beans:bean>
```

4.6.2. 自定义基础存储库

当您要自定义基本存储库行为时,上一节 中描述的方法需要自定义每个存储库接口,以使所有存储库均受到影响. 要改为更改所有存储库的行为,您可以创建一个实现,以扩展特定于持久性技术的存储库基类. 然后,该类充当存储库代理的自定义基类,如以下示例所示:

Example 39. 定制存储库基类

```
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存储库的存储库工厂实现使用的父类的构造函数. 如果存储库父类具有多个构造函数,则覆盖一个采用 `EntityInformation` 加上存储特定基础结构对象 (例如 `EntityManager` 或模板类) 的构造函数.

最后一步是使 Spring Data 基础结构了解定制的存储库基类. 在 Java 配置中,可以通过使用 `@Enable${store}Repositories` 注解的 `repositoryBaseClass` 属性来实现,如以下示例所示:

Example 40. 使用*JavaConfig*配置自定义存储库基类

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

XML 命名空间中有相应的属性,如下例所示:

Example 41. 使用*XML*配置自定义存储库基类

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

4.7. 从聚合根发布事件

由存储库管理的实体是聚合根. 在领域驱动设计应用程序中,这些聚合根通常发布领域事件. Spring Data 提供了一个称为 `@DomainEvents`

的注解,您可以在聚合根的方法上使用该注解,可以使发布事件变得简单,如以下示例所示:

Example 42. 从聚合根暴露领域事件

```
class AnAggregateRoot {

    @DomainEvents ①
    Collection<Object> domainEvents() {
        // … return events you want to get published here
    }

    @AfterDomainEventPublication ②
    void callbackMethod() {
       // … potentially clean up domain events list
    }
}
```

① 使用 `@DomainEvents` 的方法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.

② 在发布所有事件之后,我们有一个用 `@AfterDomainEventPublication` 注解的方法. 它可以用来潜在地清除要发布的事件列表 (以及其他用途) .

每次调用 Spring Data Repository `save(…)` 方法之一时,将调用这些方法.

4.8. Spring Data 扩展

本节记录了一组 Spring Data 扩展,这些扩展可在各种上下文中启用 Spring Data 使用. 当前,大多数集成都针对 Spring MVC.

4.8.1. Querydsl 扩展

Querydsl 是一个框架,可通过其流式的 API 来构造静态类型的类似 SQL 的查询.

几个 Spring Data 模块通过 `QuerydslPredicateExecutor` 与 `Querydsl` 集成,如以下示例所示:

Example 43. QuerydslPredicateExecutor 接口

```
public interface QuerydslPredicateExecutor<T> {

  Optional<T> findById(Predicate predicate);  ①

  Iterable<T> findAll(Predicate predicate);   ②

  long count(Predicate predicate);            ③

  boolean exists(Predicate predicate);        ④

  // … more functionality omitted.
}
```

① 查找并返回与 `Predicate` 匹配的单个实体。

② 查找并返回与 `Predicate` 匹配的所有实体。

③ 返回与 `Predicate` 匹配的实体数。

④ 返回与 `Predicate` 匹配的实体是否存在。

要使用 Querydsl 支持,请在存储库界面上扩展 `QuerydslPredicateExecutor`,如以下示例所示

Example 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 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许多变化,我们将以前行为的文档保存在[web.legacy]中。

支持存储库编程模型的 Spring Data 模块附带了各种 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的集成。通常,通过在 JavaConfig 配置类中使用 `@EnableSpringDataWebSupport` 注解来启用集成支持,如以下示例所示:

Example 45. 启用 *Spring Data web* 支持

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

`@EnableSpringDataWebSupport` 注解注册了一些我们稍后将讨论的组件。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,则将 `SpringDataWebConfiguration` 或 `HateoasAwareSpringDataWebConfiguration` 注册为 Spring Bean,如以下示例所示（对于 `SpringDataWebConfiguration`）:

Example 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"
/>
```

Basic Web 支持

上一节 中显示的配置注册了一些基本组件:

- DomainClassConverter 可让 Spring MVC 从请求参数或路径变量解析存储库管理的 domain 类的实例.
- `HandlerMethodArgumentResolver` 实现,可让 Spring MVC 从请求参数中解析 `Pageable` 和 `Sort` 实例.

使用 `DomainClassConverter` 类

`DomainClassConverter` 允许您直接在 Spring MVC 控制器方法签名中使用 domain 类型,因此您无需通过存储库手动查找实例,如以下示例所示:

Example 47. 一个在方法签名中使用 *domain* 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,该方法直接接收 `User` 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 domain 类的 `id` 类型并最终通过在为该类型注册的存储库实例上调用 `findById(…)` 来访问该实例来解析该实例。

当前,该存储库必须实现 `CrudRepository` 才有资格被发现以进行转换。

用于分页和排序的 `HandlerMethodArgumentResolvers`

上一节中显示的配置代码段还注册了 `PageableHandlerMethodArgumentResolver` 以及 `SortHandlerMethodArgumentResolver` 的实例。该注册启用了 `Pageable` 和 `Sort` 作为控制器方法参数,如以下示例所示

Example 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";
  }
}
```

前面的方法签名使 Spring MVC 尝试使用以下默认配置从请求参数扩展 `Pageable` 实例:

Table 1. 请求为 `Pageable` 实例评估后的参数

`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`	应该以格式属性 `property,property(,ASC|DESC)` 进行排序的属性。默认排序方向为升序。如果要切换排序,请使用多个排序参数。例如, `?sort=firstname&sort=lastname,asc`.

要自定义此行为,请注册分别实现 `PageableHandlerMethodArgumentResolverCustomizer` 接口或 `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 `MethodArgumentResolver` 的属性不足以满足您的目的,请扩展 `SpringDataWebConfiguration` 或启用 HATEOAS ,重写 `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` 注解将一个实例与另一个实例区分开。然后,请求参数必须以 `${qualifier}_` 为前缀。以下示例显示了生成的方法签名:

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

您必须填充 `thing1_page` 和 `thing2_page`,依此类推。

传递给该方法的默认 `Pageable` 等效于 `PageRequest.of(0, 20)`,但可以使用 `Pageable` 参数上的 `@PageableDefault` 注解注解进行自定义。

超媒体对页面的支持

Spring HATEOAS 附带了一个表示模型类 (`PagedResources`) ,该类允许使用必要的页面元数据以及链接来丰富 `Page` 实例的内容,并使客户端可以轻松浏览页面。`Page` 到 `PagedResources` 的转换是通过 Spring HATEOAS `ResourceAssembler` 接口 (称为 `PagedResourcesAssembler`) 的实现完成的。下面的示例演示如何将 `PagedResourcesAssembler` 用作控制器方法参数:

Example 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);
  }
}
```

如上例中所示启用配置，可以将 `PagedResourcesAssembler` 用作控制器方法参数。对其调用 `toResources(…)` 具有以下效果：

- `Page` 的内容成为 `PagedResources` 实例的内容。
- `PagedResources` 对象获取附加的 `PageMetadata` 实例，并使用 `Page` 和基础 `PageRequest` 的信息填充该实例。
- `PagedResources` 可能会附加上一个和下一个链接，具体取决于页面的状态。链接指向方法映射到的 URI。添加到该方法的分页参数与 `PageableHandlerMethodArgumentResolver` 的设置匹配，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 (`GET localhost:8080/persons`），并查看类似于以下内容的输出：

```
{ "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
  }
}
```

您会看到编译器生成了正确的 URI，并且还选择了默认配置以将参数解析为即将到来的请求的 `Pageable`。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，编译器指向调用它的控制器方法，但是可以通过传递自定义链接（用作构建分页链接的基础）进行自定义，这会使 `PagedResourcesAssembler.toResource(…)` 方法过载。

Web 数据绑定支持

通过使用 JSONPath 表达式 (需要 Jayway JsonPath 或 XPath表达式 (需要 XmlBeam)) ,可以使用 Spring Data 投影 (在 Projections 中描述) 来绑定传入的请求有效负载,如以下示例所示:

Example 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();
}
```

前面示例中显示的类型可以用作 Spring MVC 处理程序方法参数,也可以通过在 `RestTemplate` 的方法之一上使用 `ParameterizedTypeReference` 来使用. 前面的方法声明将尝试在给定文档中的任何位置查找名字. `lastname` XML查找是在传入文档的顶层执行的. JSON 首先尝试使用顶层 `lastname` ,但是如果前者不返回值,则还尝试嵌套在用户子文档中的 `lastname` . 这样,无需客户端调用暴露的方法即可轻松缓解源文档结构的更改 (通常是基于类的有效负载绑定的缺点) .

如 投影中所述,支持嵌套投影. 如果该方法返回复杂的非接口类型,则将使用Jackson `ObjectMapper` 映射最终值.

对于 Spring MVC,`@EnableSpringDataWebSupport` 处于活动状态并且所需的依赖在类路径上可用后,会自动自动注册必要的转换器. 要与 `RestTemplate` 一起使用,请手动注册 `ProjectingJackson2HttpMessageConverter` (JSON) 或 `XmlBeamHttpMessageConverter`.

有关更多信息,请参见规范的 Spring Data Examples repository存储库中的 web projection example .

Querydsl Web 支持

对于那些具有 QueryDSL 集成的存储,可以从 · 查询字符串中包含的属性扩展查询.

考虑以下查询字符串:

```
?firstname=Dave&lastname=Matthews
```

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

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

在类路径上找到 Querydsl 时,将自动启用该功能以及 `@EnableSpringDataWebSupport`.

将 `@QuerydslPredicate` 添加到方法签名中可提供一个现成的 `Predicate`,可以使用 `QuerydslPredicateExecutor` 来运行它.

> 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 domain 类型匹配,因此使用 `QuerydslPredicate` 的 `root` 属性可能是一个好主意.

下面的示例演示如何在方法签名中使用 `@QuerydslPredicate`:

```
@Controller
class UserController {

  @Autowired UserRepository repository;

  @RequestMapping(value = "/", method = RequestMethod.GET)
  String index(Model model, @QuerydslPredicate(root = User.class) Predicate
predicate,    ①
          Pageable pageable, @RequestParam MultiValueMap<String, String>
parameters) {

    model.addAttribute("users", repository.findAll(predicate, pageable));

    return "index";
  }
}
```

① 将查询字符串参数解析为与 `User Predicate` 匹配.

默认绑定如下:

- 简单属性上的对象如 `eq`.
- 集合上的对象,如 `contains` 的属性.
- 集合上的对象,如 `in` 的属性.

可以通过 `@QuerydslPredicate` 的 `bindings` 属性或通过使用 Java 8 `default methods` 并将 `QuerydslBinderCustomizer` 方法添加到存储库接口来自定义那些绑定.

```
interface UserRepository extends CrudRepository<User, String>,
                                 QuerydslPredicateExecutor<User>,
①
                                 QuerydslBinderCustomizer<QUser> {
②

  @Override
  default void customize(QuerydslBindings bindings, QUser user) {

    bindings.bind(user.username).first((path, value) -> path.contains(value))
③
    bindings.bind(String.class)
      .first((StringPath path, String value) -> path.containsIgnoreCase(value));
④
    bindings.excluding(user.password);
⑤
  }
}
```

① `QuerydslPredicateExecutor` 提供对断言的特定查找器方法的访问

② 在存储库界面上定义的 `QuerydslBinderCustomizer` 会被自动提取,并提供 `@QuerydslPredicate(bindings=…)`.

③ 将 `username` 属性的绑定定义为简单的 `contains` 绑定.

④ 将 `String` 属性的默认绑定定义为不区分大小写的 `contains` 匹配项.

⑤ 从 `Predicate` 解析中排除 `password` 属性.

4.8.3. 存储库填充器

如果您使用 Spring JDBC 模块,则可能熟悉使用 SQL 脚本填充 `DataSource` 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,因此可以在存储库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) 来定义用于填充存储库的数据.

假设您有一个包含以下内容的 `data.json` 文件:

*Example 51. JSON*中定义的数据

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

您可以使用 Spring Data Commons 中提供的存储库命名空间的 `populator` 元素来填充存储库. 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:

Example 52. 声明一个*Jackson*存储库填充器

```
<?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 `Jackson.ObjectMapper` 读取并反序列化 `data.json` 文件.

通过检查JSON文档的 `_class` 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存储库来处理反序列化的对象.

要改为使用 XML 定义应使用存储库填充的数据,可以使用 `unmarshaller-populator` 元素. 您可以将其配置为使用 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. 以下示例显示如何使用JAXB解组存储库填充器:

Example 53. 声明一个解组存储库填充器 *(*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>
```

Chapter 5. □□□□

Chapter 6. JPA 存储库

本章指出了 JPA 存储库支持的特性。这建立在“使用 Spring 数据存储库”中解释的核心存储库支持的基础上。确保您对此处介绍的基本概念有很好的了解。

6.1. 简介

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” (XML 配置)
- “基于注解的配置” (Java 配置)

6.1.1. Spring 命名空间

Spring Data 的 JPA 模块包含一个自定义命名空间,允许定义存储库 bean。它还包含 JPA 特有的某些功能和元素属性。通常,可以通过使用 `repositories` 元素来设置 JPA 存储库,如以下示例所示:

Example 54. 使用命名空间建立 JPA 存储库

```xml
<?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 `repositories` 元素可按“创建存储库实例”中所述查找 Spring Data 存储库。除此之外,它还为所有使用 `@Repository` 注解的 bean 激活持久性异常转换,以将 JPA 持久性导致的异常转换为 Spring 的 `DataAccessException` 层次结构。

自定义命名空间属性

除了 `repositories` 元素的默认属性之外,JPA 命名空间还提供了其他属性,使您可以更详细地控制存储库的设置:

Table 2. 自定义 JPA 特定于 `repositories` 元素的属性

`entity-manager-factory-ref`	显式地将 `EntityManagerFactory` 与要使用的 `repositories` 元素所检测到的存储库连接。通常在应用程序中使用多个 `EntityManagerFactory` bean 的情况下使用。如果未配置,Spring Data 会在 `ApplicationContext` 中自动查找名称为 `EntityManagerFactory` 的 `EntityManagerFactory` bean.
`transaction-manager-ref`	明确地将 `PlatformTransactionManager` 与要由 `repositories` 元素检测到的存储库进行连线。通常仅在配置了多个事务管理器或 `EntityManagerFactory` bean 时才需要。默认为当前 `ApplicationContext` 中单个定义的 `PlatformTransactionManager`.

如果显式未定义的 `transaction-manager-ref`,Spring Data JPA 要求提供一个名为 `transactionManager` 的 `PlatformTransactionManager` bean.

6.1.2. 基于注解的配置

Spring Data JPA 存储库支持不仅可以通过 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:

Example 55. 使用 JavaConfig 的 Spring Data JPA 存储库

```
@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;
  }
}
```

您必须直接创建 `LocalContainerEntityManagerFactoryBean` 而不是 `EntityManagerFactory`,因为前者除了创建 `EntityManagerFactory` 之外,还参与异常转换机制.

6.1.3. 引导模式

默认情况下,Spring Data JPA 存储库是默认的 Spring Bean. 它们是单例作用域,并且已被初始化. 在启动期间,它们已经与 JPA `EntityManager` 进行交互,以进行验证和元数据分析. Spring 框架在后台线程中支持 JPA `EntityManagerFactory` 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. 为了有效地利用后台初始化,我们需要确保 JPA 存储库尽可能早地初始化.

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) ,该 `BootstrapMode` 采用以下值:

- `DEFAULT` (默认值) — 急切地实例化存储库,除非使用 `@Lazy` 显式注解. 仅当没有任何客户 Bean 需要存储库实例时,`lazification` 才有效,因为这将需要初始化存储库 bean.
- `LAZY` — 隐式地声明所有存储库 bean 都是惰性的,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存储库,则不会实例化存储库. 首次与存储库交互时,将初始化并验证存储库实例.
- `DEFERRED` — 基本与 `LAZY` 相同,但会响应 `ContextRefreshedEvent` 触发存储库初始化,以便在应用程序完全启动之前验证存储库.

推荐建议

如果您不使用默认 Bootstrap 模式的异步 JPA 引导.

如果您以异步方式引导 JPA,则 `DEFERRED` 是一个合理的默认值,因为它可以确保 Spring Data JPA 引导仅在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 `EntityManagerFactory` 安装. 尽管如此,它仍可以确保在应用程序发出信号之前,对存储库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. 一旦确定了存储库将正确引导后,或者在测试应用程序的其他部分时,对所有存储库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存储库即可.

6.2. 持久化实体

本节描述如何使用 Spring Data JPA 持久化 (保存) 实体.

6.2.1. 保存实体

可以使用 `CrudRepository.save(…)` 方法执行保存实体. 它通过使用基础 JPA `EntityManager` 持久化或合并给定实体. 如果实体还没有持久化,Spring Data JPA 会通过调用 `entityManager.persist(…)` 方法来保存实体. 否则,它将调用 `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,则将该实体视为新实体 (如果该值为 `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. 如果标识符属性为 `null`,则假定该实体为新实体. 否则,它不是新的.
2. 实现 `Persistable`: 如果实体实现 `Persistable`,则 Spring Data JPA 将新检测委托给该实体的 `isNew(…)` 方法. 有关详细信息,请参见 JavaDoc .
3. 实现 `EntityInformation`: 通过创建 `JpaRepositoryFactory` 的子类并相应地重写 `getEntityInformation(…)` 方法,可以自定义 `SimpleJpaRepository` 实现中使用的 `EntityInformation` 抽象. 然后,您必须将 JpaRepositoryFactory 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 .

对于使用手动分配的标识符的实体,选项 1 不是选项,因为标识符将始终为非 `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反转该标志：

Example 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; ①

  @Override
  public boolean isNew() {
    return isNew; ②
  }

  @PrePersist ③
  @PostLoad
  void markNotNew() {
    this.isNew = false;
  }

  // More code…
}
```

① 声明一个标志以保持新状态。暂时的，因此它不会持久化到数据库中。

② 在 `Persistable.isNew()` 的实现中返回标志，以便 Spring Data 存储库知道是调用 `EntityManager.persist()` 还是 `….merge()`。

③ 声明一个使用 JPA 实体回调的方法，以便在存储库调用 `save(…)` 或持久性提供程序创建实例之后，将标志切换为指示现有实体。

6.3. 查询方法

本节描述了使用 Spring Data JPA 创建查询的各种方法。

6.3.1. 查询策略

JPA 模块支持手动将查询定义为 String 或从方法名称扩展查询。

断言为 `IsStartingWith`,`StartingWith`,`StartsWith`,`IsEndingWith`,`EndingWith`,`EndsWith`,`IsNotContaining`,`NotContaining`,`NotContains`,`IsContaining`,`Containing` 的扩展查询将包含这些查询的各自参数。 这意味着，如果参数实际包含 `LIKE` 识别为通配符的字符，则这些字符将被转义，因此它们仅作为文字匹配。 可以通过设置 `@EnableJpaRepositories` 注解的 `escapeCharacter` 来配置使用的转义字符。 与使用 SpEL 表达式进行比较。

声明查询

尽管从方法名扩展一个查询很方便，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，或者方法名变的丑陋。 因此，您可以通过命名约定使用 JPA 命名查询 (有关更多信息，请参见使用JPA 命名查询) ，或者通过 `@Query` 注解您的查询方法 (有关详细信息，请参见使用 `@Query`) .

6.3.2. 查询创建

通常,JPA 的查询创建机制按 “查询方法” 中所述运行. 以下示例显示了 JPA 查询方法转换为的内容:

Example 57. 通过方法名称创建查询

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

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

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” 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:

Table 3. 方法名称中受支持的关键字

关键字	Sample	JPQL snippet
`Distinct`	`findDistinctByLastnameAndFirstname`	`select distinct … where x.lastname = ?1 and x.firstname = ?2`
`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` (parameter bound with appended `%`)

关键字	**Sample**	**JPQL snippet**
`EndingWith`	`findByFirstnameEndingWith`	`… where x.firstname like ?1` (parameter bound with prepended `%`)
`Containing`	`findByFirstnameContaining`	`… where x.firstname like ?1` (parameter bound wrapped in `%`)
`OrderBy`	`findByAgeOrderByLastnameDesc`	`… where x.age = ?1 order by x.lastname desc`
`Not`	`findByLastnameNot`	`… where x.lastname <> ?1`
`In`	`findByAgeIn(Collection<Age> ages)`	`… where x.age in ?1`
`NotIn`	`findByAgeNotIn(Collection<Age> ages)`	`… where x.age not in ?1`
`True`	`findByActiveTrue()`	`… where x.active = true`
`False`	`findByActiveFalse()`	`… where x.active = false`
`IgnoreCase`	`findByFirstnameIgnoreCase`	`… where UPPER(x.firstname) = UPPER(?1)`

`In` 和 `NotIn` 也将 `Collection` 的任何子类作为参数以及 `varargs` 的数组. 对于同一逻辑运算符的其他语法版本,请选中 "存储库查询关键字".

6.3.3. 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,您也可以使用 `<named-native-query />` 或 `@NamedNativeQuery` . 这些元素使您可以通过在没有数据库平台独立性来在本地 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 />` 元素添加到位于类路径的 `META-INF` 文件夹中的 `orm.xml` 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.

Example 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 domain 类,从而为此付出了代价.

Example 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,请按以下方式指定 `UserRepository`:

*Example 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 domain 类的简单名称开始,然后是由点分隔的方法名称。因此,前面的示例将使用示例中定义的命名查询,而不是尝试从方法名称创建查询。

6.3.4. 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,因此您实际上可以通过使用 Spring Data JPA `@Query` 注解直接绑定它们,而不是将它们注解到 domain 类。这样可以将 domain 类从持久性特定的信息中释放出来,并将查询放置在存储库接口中。

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

以下示例显示使用 `@Query` 注解创建的查询:

Example 61. 使用 `@Query` 在查询方法中声明查询

```
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 @Query 创建的手动定义查询的查询执行机制允许在查询定义中定义高级 LIKE 表达式,如以下示例所示:

*Example 62. @Query*中的高级 like 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

@Query 注解允许通过将 nativeQuery 标志设置为 true 来运行本地查询,如以下示例所示:

Example 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 当前不支持对本地查询进行动态排序,因为它必须声明操作的实际查询,而这对于本地 SQL 无法可靠地进行. 但是,您可以自己指定 count 查询,从而将本地查询用于分页,如以下示例所示:

Example 64. 使用 @Query 在查询方法中声明本地 *count* 查询以进行分页

```
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.

6.3.5. 使用排序

可以通过提供 `PageRequest` 或直接使用 `Sort` 来完成排序。 在 `Sort` 的 `Order` 实例中实际使用的属性需要与您的 domain 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异常。

但是,将 `Sort` 与 `@Query` 一起使用,可以让您潜入包含 `ORDER BY` 子句中的函数的未经路径检查的 `Order` 实例。 这是可能的,因为 `Order` 附加到给定的查询字符串。 默认情况下,Spring Data JPA 拒绝任何包含函数调用的 Order 实例,但是您可以使用 `JpaSort.unsafe` 添加可能不安全的排序。

以下示例使用 `Sort` 和 `JpaSort`,在 `JpaSort` 上包括一个不安全的选项:

Example 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", Sort.by("firstname"));                ①
repo.findByAndSort("stark", Sort.by("LENGTH(firstname)"));            ②
repo.findByAndSort("targaryen", JpaSort.unsafe("LENGTH(firstname)")); ③
repo.findByAsArrayAndSort("bolton", Sort.by("fn_len"));               ④
```

① 指向 domain 模型中属性的有效 `Sort` 表达式。

② 包含函数调用的无效 `Sort` Throws 异常。

③ 有效 `Sort` 包含明显不安全的 `Order`。

④ 指向别名函数的有效 `Sort` 表达式。

6.3.6. 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,并在查询中绑定该名称,如以下示例所示:

Example 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);
}
```

方法参数根据其在定义的查询中的顺序进行切换.

从版本 4 开始,Spring 完全基于 `-parameters` 编译器标志支持 Java 8 的参数名称发现. 通过在构建中使用此注解作为调试信息的替代方法,可以省略 `@Param` 注解中的命名参数.

6.3.7. 使用 SpEL 表达式

从 Spring Data JPA 1.4 版开始,我们支持在使用 `@Query` 定义的手动定义的查询中使用受限的 SpEL 模板表达式. 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 支持一个名为 `entityName` 的变量. 它的用法是 `select x from #{#entityName} x`. 它插入与给定存储库关联的 domain 类型的 `entityName`. 实体名称的解析如下: 如果 domain 类型已在 `@Entity` 注解上设置了名称属性,则将其使用. 否则,将使用 domain 类型的简单类名.

以下示例演示了查询字符串中 `#{#entityName}` 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存储库接口:

Example 67. 在存储库查询方法中使用*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.

可以使用 `@Entity` 注解来自定义 `entityName`. SpEL 表达式不支持 `orm.xml` 中的自定义.

当然,您可能只在查询声明中直接使用了 `User`,但这也需要您更改查询. 对 `#entityName` 的引用将 `User` 类将来可能的重新映射选择为另一个实体名称 (例如,通过使用 `@Entity(name = "MyUser")`.

查询字符串中 `#{#entityName}` 表达式的另一个用例是,如果您想为特定的 domain 类型定义一个带有专用存储库接口的通用存储库接口. 要不在具体接口上重复定义自定义查询方法,可以在通用存储库接口的 `@Query` 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:

Example 68. 在 *repository* 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,`MappedTypeRepository` 接口是扩展 `AbstractMappedType` 的一些 domain 类型的公共父接口. 它还定义了通用的 `findAllByAttribute(…)` 方法,该方法可用于专用存储库接口的实例. 如果现在在 `ConcreteRepository` 上调用 `findByAllAttribute(…)`,则查询 `select t from ConcreteType t where t.attribute = ?1`.

SpEL 表达式可用于操作参数,也可用于操作方法参数. 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.

Example 69. 在存储库查询方法中使用 *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`,通常需要将 % 附加到 `String` 值参数的开头或结尾. 这可以通过在绑定参数标记或 SpEL 表达式上附加或前缀 % 来完成. 以下示例再次说明了这一点.

Example 70. 在 *repository* 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);
```

如果使用 `like` 条件的值来自不安全来源,则应清除这些值,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,在 SpEL 上下文中可以使用 `escape(String)` 方法. 它在第一个参数中的 `_` 和 % 的所有实例之前加上第二个参数中的单个字符. 与 JPQL 中提供的 `like` 表达式的转义子句和标准 SQL 结合使用,可以轻松清除绑定参数.

Example 71. 在存储库查询方法中使用 *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);
```

在存储库接口中给出此方法声明后, `findContainingEscaped("Peter_")` 将找到 `Peter_Parker` 而不是 `Peter Parker`. 可以通过设置 `@EnableJpaRepositories` 注解的 `escapeCharacter` 来配置所使用的转义字符. 请注意,该方法 `escape(String)` 可用 在SpEL上下文中,仅将转义 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,则将不会转义这些通配符.

6.3.8. 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数据存储库的自定义实现" 中介绍的功能来添加自定义修改行为. 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 `@Modifying` 注解查询方法来修改仅需要参数绑定的查询,如以下示例所示:

Example 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);
```

这样做会触发注解该方法的查询作为更新查询,而不是选择查询. 由于 `EntityManager` 在执行修改查询后可能包含之前的实体,因此我们不会自动清除它 (有关详细信息,请参阅 `EntityManager.clear()` 的 JavaDoc) ,因为这会有效地将所有尚未刷新的更新丢弃在 `EntityManager` 中. 如果您希望自动清除 `EntityManager`,则可以将 `@Modifying` 注解的 `clearAutomatically` 属性设置为 `true`.

`@Modifying` 注解仅与 `@Query` 注解结合使用. 扩展的查询方法或自定义方法不需要此注解.

扩展删除查询

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

Example 73. 使用扩展的删除查询

```
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 `deleteByRoleId(…)` 方法看起来基本上与 `deleteInBulkByRoleId(…)` 产生相同的结果,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 `User` 实例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 `deleteByRoleId(…)` 会执行一个查询,然后逐个删除返回的实例,以便持久性提供程序实际上可以在这些实体上调用 `@PreRemove` 回调.

实际上,扩展的删除查询是执行查询,然后对结果调用 `CrudRepository.delete(Iterable<User> users)` 并使行为与 `CrudRepository` 中其他 `delete(…)` 方法的实现保持同步的快捷方式.

6.3.9. 应用查询提示

要将 JPA 查询提示应用于在存储库接口中声明的查询,可以使用 `@QueryHints` 注解. 它需要一个JPA `@QueryHint` 注解加上一个布尔标志,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:

Example 74. 将 `QueryHints` 与存储库方法一起使用

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

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

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

6.3.10. 配置 Fetch- 和 LoadGraphs

JPA 2.1 规范引入了对指定 Fetch- 和 LoadGraphs 的支持,我们也支持 `@EntityGraph` 注解,该注解使您可以引用 `@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 `@EntityGraph` 注解上的 `type` 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.

Example 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存储库查询方法上引用命名实体图:

Example 76. 在存储库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);

}
```

也可以使用 `@EntityGraph` 定义临时实体图. 提供的 `attributePaths` 转换为相应的 `EntityGraph`,而无需将 `@NamedEntityGraph` 显式添加到您的 domain 类型,如以下示例所示:

Example 77. 在存储库查询方法上使用 *AD-HOC* 实体图定义.

```
@Repository
public interface GroupRepository extends CrudRepository<GroupInfo, String> {

  @EntityGraph(attributePaths = { "members" })
  GroupInfo getByGroupName(String name);

}
```

6.3.11. 投影

Spring Data 查询方法通常返回存储库管理的聚合根的一个或多个实例. 但是,有时可能需要根据这些类型的某些属性创建投影. 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.

想象一下一个存储库和聚合根类型，例如以下示例：

Example 78. 一个示例集合和存储库

```
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。

基于接口的投影

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，该接口暴露要读取的属性的 get 方法，如以下示例所示：

Example 79. 一个投影接口来检索属性的子集

```
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：

Example 80. 使用基于接口的投影和查询方法的存储库

```
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，并将对暴露方法的调用转发给目标对象。

投影可以递归使用. 如果还希望包括一些 `Address` 信息,则为此创建一个投影接口,并从 `getAddress()` 的声明返回该接口,如以下示例所示:

Example 81. 一个投影接口来检索属性的子集

```
interface PersonSummary {

  String getFirstname();
  String getLastname();
  AddressSummary getAddress();

  interface AddressSummary {
    String getCity();
  }
}
```

在方法调用时,将获得目标实例的 `address` 属性,并将其包装到投影代理中.

封闭投影

其 get 方法均与目标集合的属性完全匹配的投影接口被视为封闭投影. 下面的示例 (也在本章前面使用过) 是一个封闭的投影:

Example 82. 一个封闭的投影

```
interface NamesOnly {

  String getFirstname();
  String getLastname();
}
```

如果您使用封闭式投影,Spring Data 可以优化查询执行,因为我们知道支持投影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.

打开投影

投影接口中的 get 方法也可以通过使用 `@Value` 注解来计算新值,如以下示例所示:

Example 83. 一个 打开投影

```
interface NamesOnly {

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

在 `target` 变量中提供了支持投影的合计根. 使用 `@Value` 的投影接口是开放式投影. 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.

`@Value` 中使用的表达式应该不太复杂-您要避免在 `String` 变量中进行编程. 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:

Example 84. 使用默认方法自定义逻辑的投影接口

```
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投影接口上暴露的其他 get 方法来实现逻辑. 第二个更灵活的选择是在 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:

Example 85. Sample Person 对象

```
@Component
class MyBean {

  String getFullName(Person person) {
    …
  }
}

interface NamesOnly {

  @Value("#{@myBean.getFullName(target)}")
  String getFullName();
  …
}
```

请注意 SpEL 表达式如何引用 `myBean` 并调用 `getFullName(…)` 方法,并将投影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. 方法参数可通过名为 `args` 的对象数组获得. 下面的示例演示如何从 `args` 数组获取方法参数:

Example 86. Sample Person 对象

```
interface NamesOnly {

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

同样,对于更复杂的表达式,您应该使用Spring bean并让该表达式调用方法,如前所述.

基于类的投影 (DTO)

定义投影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类型可以以与使用投影接口完全相同的方式使用,除了没有代理发生和不能应用嵌套投影之外.

如果存储通过限制要加载的字段来优化查询执行,则要加载的字段由暴露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:

Example 87. 一个投影的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
}
```

避免投影*DTO*的样板代码

您可以使用 Project Lombok 大大简化DTO的代码,该项目提供了 `@Value` 注解（不要与前面的界面示例中显示的Spring的 `@Value` 注解混淆）. 如果您使用 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;
}
```

默认情况下,字段是 `private` `final` 的,并且该类暴露了一个构造函数,该构造函数接受所有字段并自动获取实现的 `equals(…)` 和 `hashCode()` 方法.

动态投影

到目前为止,我们已经将投影类型用作集合的返回类型或元素类型. 但是,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）. 要应用动态投影,请使用查询方法,如以下示例中所示:

Example 88. 使用动态投影参数的存储库

```
interface PersonRepository extends Repository<Person, UUID> {

  <T> Collection<T> findByLastname(String lastname, Class<T> type);
}
```

通过这种方式,该方法可以按原样或应用投影来获得聚合,如下例所示:

Example 89. 使用带有动态投影的存储库

```
void someMethod(PersonRepository people) {

  Collection<Person> aggregates =
    people.findByLastname("Matthews", Person.class);

  Collection<NamesOnly> aggregates =
    people.findByLastname("Matthews", NamesOnly.class);
}
```

6.4. 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. 我们引入了 `@Procedure` 注解,用于在存储库方法上声明存储过程元数据.

下面的示例使用以下存储过程:

*Example 90. HSQL DB*中 `plus1inout` 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
/;
```

可以通过在实体类型上使用 `NamedStoredProcedureQuery` 注解来配置存储过程的元数据.

Example 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 {}
```

请注意,`@NamedStoredProcedureQuery` 具有两个不同的存储过程名称. 名称是 JPA 使用的名称. `procedureName` 是存储过程在数据库中具有的名称.

您可以通过多种方式从存储库方法引用存储过程. 可以使用 `@Procedure` 注解的 `value` 或 `procedureName` 属性直接定义要调用的存储过程. 这直接引用数据库中的存储过程,并忽略通过 `@NamedStoredProcedureQuery` 进行的任何配置.

或者,您可以将 `@NamedStoredProcedureQuery.name` 属性指定为 `@Procedure.name` 属性. 如果未配置 `value`,`procedureName` 或 `name`,则将存储库方法的名称用作 `name` 属性.

下面的示例演示如何引用显式映射的过程:

Example 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,但是使用了 `procedureName` 别名:

Example 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,但是使用方法名称而不是显式注解属性.

Example 94. 使用方法名称在 `EntityManager` 中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.

Example 95. 在 *EntityManager* 中引用显式映射的命名存储过程 *"User.plus1IO"*.

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

如果被调用的存储过程只有一个 `out` 参数,则该参数可以作为方法的返回值返回. 如果在 `@NamedStoredProcedureQuery` 注解中指定了多个 `out` 参数,则这些参数可以作为 `Map` 返回,其键为 `@NamedStoredProcedureQuery` 注解中给出的参数名称.

6.5. Specification

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. 通过编写条件,可以定义域类查询的 `where` 子句. 再往前一步,这些标准可以视为 JPA 标准 API 约束所描述的实体的断言.

Spring Data JPA 遵循 Eric Evans 的书 “领域驱动设计” 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,可以使用 `JpaSpecificationExecutor` 接口扩展存储库接口,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,`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);
```

`Specification` 接口定义如下:

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

Specifications 可以轻松地用于在实体之上构建可扩展的断言集合,然后可以将其组合并与 `JpaRepository` 一起使用,而无需为每个所需的组合声明查询 (方法) ,如以下示例所示:

Example 96. 自定义 *Specifications*

```
public class CustomerSpecs {


  public static Specification<Customer> isLongTermCustomer() {
    return (root, query, builder) -> {
      LocalDate date = LocalDate.now().minusYears(2);
      return builder.lessThan(root.get(Customer_.createdAt), date);
    };
  }

  public static Specification<Customer> hasSalesOfMoreThan(MonetaryAmount value) {
    return (root, query, 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. `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,参见Hibernate实现的文档) . 因此,表达式 `Customer_.createdAt` 假定客户具有类型为 `Date` 的 `createdAt` 属性. 除此之外,我们在业务需求抽象级别上表达了一些标准,并创建了可执行的 `Specifications`. 因此,客户端可以使用以下 `Specifications`:

Example 97. 使用一个简单的 *Specification*

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

为什么不为这种数据访问创建查询? 与纯查询声明相比,使用单个 `Specification` 不会带来很多好处. 将 `specifications` 组合在一起以创建新的 `specifications` 对象时,`specifications` 的力量真正发挥了作用. 您可以通过我们提供的用于构建类似于以下内容的表达式的默认 `Specification` 方法来实现此目的:

Example 98. 组合 Specifications

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

`Specification` 提供了一些 “glue-code” 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` 实现并将它们与现有的实现组合来扩展数据访问层.

6.6. 查询例子

6.6.1. 简介

本章对 "按示例查询" 进行了介绍,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,"示例查询" 根本不需要您使用存储库特定的查询语言编写查询.

6.6.2. 使用

示例查询由三部分组成:

- Probe: 带有填充字段的 domain 对象的实际示例.
- `ExampleMatcher`: `ExampleMatcher` 包含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`: 示例包括探针和 `ExampleMatcher`. 它用于创建查询.

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 domain 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.

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 domain 对象. 首先,为您的存储库创建一个接口,如以下示例所示:

Example 99. Sample Person 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,具有 `null` 的字段将被忽略,并且使用存储特定的默认值来匹配字符串。 可以使用工厂方法或使用 `ExampleMatcher` 构建示例。例子是一成不变的。以下清单显示了一个简单的示例:

Example 100. Simple Example

```
Person person = new Person();                         ①
person.setFirstname("Dave");                          ②

Example<Person> example = Example.of(person);         ③
```

① 创建 domain 对象的新实例。

② 设置要查询的属性。

③ 创建 `Example`。

最好在存储库中执行示例。为此,让您的存储库接口扩展 `QueryByExampleExecutor<T>`。以下清单显示了 `QueryByExampleExecutor` 接口:

Example 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.
}
```

6.6.3. Example 匹配

示例不限于默认设置。您可以使用 `ExampleMatcher` 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,如以下示例所示:

Example 102. 具有定制匹配的例子匹配器

```
Person person = new Person();                          ①
person.setFirstname("Dave");                           ②

ExampleMatcher matcher = ExampleMatcher.matching()     ③
  .withIgnorePaths("lastname")                         ④
  .withIncludeNullValues()                             ⑤
  .withStringMatcherEnding();                          ⑥

Example<Person> example = Example.of(person, matcher); ⑦
```

① 创建域对象的新实例。

② 设置属性。

③ 创建一个 `ExampleMatcher` 以期望所有值都匹配。即使没有进一步的配置,它也可以在此阶段使用。

④ 构造一个新的 `ExampleMatcher` 以忽略 `lastname` 属性路径。

⑤ 构造一个新的 `ExampleMatcher` 以忽略 `lastname` 属性路径并包含空值。

⑥ 构造一个新的 `ExampleMatcher` 以忽略 `lastname` 属性路径,包括空值,并执行后缀字符串匹配。

⑦ 基于域对象和配置的 `ExampleMatcher` 创建一个新的 `Example`。

默认情况下,`ExampleMatcher` 期望设置的所有值都匹配。如果要获取与隐式定义的任何断言匹配的结果,请使用 `ExampleMatcher.matchingAny()`。

您可以为单个属性 (例如 "firstname" 和 "lastname",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:

Example 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:

Example 104. 用*lambdas*配置匹配器选项

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

由 `Example` 创建的查询使用配置的合并视图。 可以在 `ExampleMatcher` 级别上设置默认的匹配设置,而可以将单个设置应用于特定的属性路径。 除非明确定义,否则

`ExampleMatcher` 上设置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:

表4. `ExampleMatcher` 设置的范围

Table 4. Scope of `ExampleMatcher` *settings*

Setting	Scope
Null-handling	`ExampleMatcher`
String matching	`ExampleMatcher` and property path
Ignoring properties	Property path
Case sensitivity	`ExampleMatcher` and property path
Value transformation	Property path

6.6.4. 执行一个例子

在 Spring Data JPA 中,您可以对存储库使用按示例查询,如以下示例所示:

Example 105. 使用存储库按示例查询

```
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,仅 `SingularAttribute` 属性可用于属性匹配。

属性说明符接受属性名称(例如 `firstname` and `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。

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

Table 5. `StringMatcher` *options*

Matching	Logical result
`DEFAULT` (case-sensitive)	`firstname = ?0`
`DEFAULT` (case-insensitive)	`LOWER(firstname) = LOWER(?0)`
`EXACT` (case-sensitive)	`firstname = ?0`
`EXACT` (case-insensitive)	`LOWER(firstname) = LOWER(?0)`
`STARTING` (case-sensitive)	`firstname like ?0 + '%'`

Matching	**Logical result**
`STARTING` (case-insensitive)	`LOWER(firstname) like LOWER(?0) + '%'`
`ENDING` (case-sensitive)	`firstname like '%' + ?0`
`ENDING` (case-insensitive)	`LOWER(firstname) like '%' + LOWER(?0)`
`CONTAINING` (case-sensitive)	`firstname like '%' + ?0 + '%'`
`CONTAINING` (case-insensitive)	`LOWER(firstname) like '%' + LOWER(?0) + '%'`

6.7. 事务性

默认情况下,存储库实例上的 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`. 所有其他文件都配置有简单的 `@Transactional`,以便应用默认事务配置. 有关详细信息,请参见 `SimpleJpaRepository` 的JavaDoc. 如果需要调整存储库中声明的方法之一的事务配置,请在存储库接口中重新声明该方法,如下所示:

Example 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 `findAll()` 方法以 10 秒的超时运行,并且没有 `readOnly` 标志.

更改事务行为的另一种方法是使用 facade 或 service 实现 (通常) 覆盖多个存储库. 其目的是为非 CRUD 操作定义事务边界. 以下示例使用了 facade 用于多个存储库:

Example 107. 使用外观定义多个存储库调用的事务

```
@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);
    }
}
```

此示例使对 `addRoleToAllUsers(…)` 的调用在事务内运行 (参与现有事务或在没有事务的情况下创建新事务) . 然后忽略存储库中的事务配置,因为外部事务配置确定了实际使用的事务配置. 请注意,必须激活 `<tx:annotation-driven />` 或显式使用 `@EnableTransactionManagement` 才能使立面的基于注解的配置生效. 本示例假定您使用组件扫描.

请注意, 从 JPA 的角度来看, 对 `save` 的调用不是绝对必要的, 但为了与 Spring Data 提供的存储库抽象保持一致, 仍然应该存在.

6.7.1. 事务查询方法

要使查询方法具有事务性,请在您定义的存储库接口上使用 `@Transactional`,如以下示例所示:

Example 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()` 使用 `@Modifying` 注解并覆盖事务配置。因此,该方法在 `readOnly` 标志设置为 `false` 的情况下运行。

ℹ 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。但是,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只读事务中的 `INSERT` 和 `UPDATE` 语句)。相反,将 `readOnly` 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 一起使用时,将事务配置为 `readOnly` 时,刷新模式将设置为 `NEVER`,这将导致 Hibernate 跳过脏检查(对大型对象树的显着改进)。

6.8. 锁

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

Example 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);
}
```

此方法声明使触发的查询配备有 `READ` 的 `LockModeType`。您还可以通过在存储库界面中重新声明 CRUD 方法并为它们添加 `@Lock` 注解来定义 CRUD 方法的锁定,如以下示例所示:

Example 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();
}
```

6.9. 审计

6.9.1. 基础

Spring Data 提供了完善的支持,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` 来捕获创建或修改实体的用户,并提供 `@CreatedDate` 和 `@LastModifiedDate` 来捕获更改发生的时间。

Example 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。 捕获更改时捕获的注解可用于类型 `Joda-Time`,`DateTime`,旧版Java `Date` 和 `Calendar`,JDK8日期和时间类型以及 `long` 或 `Long` 的属性。

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 domain 类实现 `Auditable` 接口。它为所有审核属性暴露了setter方法。

还有一个便捷的基类 `AbstractAuditable`,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 domain 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 `AuditorAware<T>` SPI接口,您必须实现该接口以告知基础结构与应用程序交互的当前用户或系统是谁。通用类型T定义必须使用 `@CreatedBy` 或 `@LastModifiedBy` 注解的属性的类型。

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

Example 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Spring Security提供的 `Authentication` 对象,并查找您在 `UserDetailsService` 实现中创建的自定义 `UserDetails` 实例. 我们在这里假设您通过 `UserDetails` 实现暴露域用户,但是根据找到的身份验证,您还可以从任何地方查找它. :leveloffset: -1

6.9.2. JPA 审计

通用审核配置

Spring Data JPA 附带了一个实体监听器,该监听器可用于触发捕获审计信息. 首先,必须在 `orm.xml` 文件内的持久性上下文中注册要用于所有实体的 `AuditingEntityListener`,如以下示例所示:

Example 113. Auditing configuration 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>
```

您还可以使用 `@EntityListeners` 注解按每个实体启用 `AuditingEntityListener`,如下所示:

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

}
```

ℹ 审计功能要求 `spring-aspects.jar` 位于类路径中.

通过适当地修改 `orm.xml` 并在类路径上使用 `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:

Example 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 `@EnableJpaAuditing` 注解对配置类进行注解来启用审核. 您仍然必须修改 `orm.xml` 文件,并在类路径上具有 `spring-aspects.jar`. 以下示例显示了如何使用 `@EnableJpaAuditing` 注解:

Example 115. 用 *Java* 配置激活审计

```
@Configuration
@EnableJpaAuditing
class Config {

  @Bean
  public AuditorAware<AuditableUser> auditorProvider() {
    return new AuditorAwareImpl();
  }
}
```

如果将类型 `AuditorAware` 的 bean 暴露给 `ApplicationContext`,则审计基础结构会自动选择它并使用它来确定要在 domain 类型上设置的当前用户. 如果您在 `ApplicationContext` 中注册了多个实现,则可以通过显式设置 `@EnableJpaAuditing` 的 `auditAwareRef` 属性来选择要使用的实现.

6.10. 其他注意事项

6.10.1. 在自定义实现中使用 `JpaContext`

当使用多个 `EntityManager` 实例和自定义存储库实现实现时,您需要将正确的 `EntityManager` 连接到存储库实现类中. 您可以通过在 `@PersistenceContext` 注解中显式命名 `EntityManager` 来实现, 或者,如果 `EntityManager` 是 `@Autowired`,则可以使用 `@Qualifier` 来实现.

从 Spring Data JPA 1.9 开始,Spring Data JPA 包含一个名为 `JpaContext` 的类,假定您只由应用程序中的 `EntityManager` 实例之一进行管理,该类使您可以通过被管理 domain 类获取 `EntityManager`. 以下示例显示如何在自定义存储库中使用 `JpaContext`:

Example 116. 在自定义存储库实现中使用 `JpaContext`

```
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 domain 类型分配给其他持久性单元,则无需触摸存储库即可更改对持久性单元的引用.

6.10.2. 合并持久性单元

Spring 支持具有多个持久性单元. 但是,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性单元中运行. 为了实现这种行为,Spring Data JPA 提供了一个 `PersistenceUnitManager` 实现,该实现会根据其名称自动合并持久性单元,如以下示例所示:

Example 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设置要求所有注解映射的实体类在 `orm.xml` 中列出. XML映射文件也是如此. Spring Data JPA 提供了一个 `ClasspathScanningPersistenceUnitPostProcessor`,它配置了一个基本包,并可以选择采用映射文件名模式. 然后,它在给定的软件包中扫描以 `@Entity` 或 `@MappedSuperclass` 注解的类,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:

Example 118. 使用 ClasspathScanningPersistenceUnitPostProcessor

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

从 Spring 3.1 开始,可以直接在 `LocalContainerEntityManagerFactoryBean` 上配置要扫描的程序包,以对实体类启用类路径扫描。有关详细信息,请参见 JavaDoc

6.10.3. CDI 集成

存储库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。如创建存储库实例中所述,Spring 为创建 bean 实例提供了复杂的支持。从 1.1.0 版本开始,Spring Data JPA 附带了一个自定义 CDI 扩展名,该扩展名允许在 CDI 环境中使用存储库抽象。该扩展是 JAR 的一部分。要激活它,请将 Spring Data JPA JAR 包含在类路径中。

现在,您可以通过为 `EntityManagerFactory` 和 `EntityManager` 实现 CDI 生产者来设置基础结构,如以下示例所示:

```
class EntityManagerFactoryProducer {

  @Produces
  @ApplicationScoped
  public EntityManagerFactory createEntityManagerFactory() {
    return Persistence.createEntityManagerFactory("my-persistence-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将 `EntityManager` 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 `EntityManager` 重新导出为 CDI bean.

每当容器请求存储库类型的 bean 时,Spring Data JPA CDI 扩展都将所有可用的 `EntityManager` 实例作为 CDI bean 进行选择,并为 Spring Data 存储库创建代理. 因此,获取 Spring Data 存储库的实例只需声明一个 `@Injected` 属性即可,如以下示例所示:

```
class RepositoryClient {

  @Inject
  PersonRepository repository;

  public void businessMethod() {
    List<Person> people = repository.findAll();
  }
}
```

Chapter 7. □□

Appendix A: 命名空间参考

`<repositories />` 元素

`<repositories />` 元素触发Spring Data存储库基础结构的设置。最重要的属性是 `base-package`,它定义了要扫描Spring Data 存储库接口的软件包。请参阅 “XML配置”。下表描述了 `<repositories />` 元素的属性:

Table 6. 属性

名称	描述
`base-package`	定义要扫描的软件包,以查找在自动检测模式下扩展 `*Repository`(实际接口由特定的Spring Data模块确定) 的存储库接口。 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存储库实现。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”。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`	是否应考虑嵌套的存储库接口定义。默认为 `false`。

Appendix B: Populators 命名空间参考

<populator /> element

`<populator />` 元素允许通过Spring数据存储库基础结构填充数据存储. [1]

Table 7. 属性

名称	描述
`locations`	从哪里可以找到要从存储库读取对象的文件,应在其中填充.

[1] 参阅 XML 配置

Appendix C: 存储库查询关键字

支持的查询方法主题关键字

下表列出了 Spring Data 存储库查询扩展机制通常支持的表示断言的主题关键字.但是,请参阅 store-specific 的文档以获取受支持关键字的确切列表,因为 store-specific 可能不支持此处列出的某些关键字.

Table 8. Query 主题关键字

关键字	描述
`find…By`, `read…By`, `get…By`, `query…By`, `search…By`, `stream…By`	一般查询方法通常返回存储库类型, `Collection` 或 `Streamable` 的子类型或包装类型 `Page`, `GeoResults` 或任何其他 store-specific 的结果包装器. 可以用作 `findBy…`, `findMyDomainTypeBy…` 或其他关键字结合使用.
`exists…By`	是否存在, 通常返回 `boolean` 类型.
`count…By`	计算返回的结果数字
`delete…By`, `remove…By`	删除查询方法,不返回结果 (`void`) 或 delete count.
`…First<number>…`, `…Top<number>…`	返回查询结果的第一个 `<number>` . 此关键字可以出现在主题 `find` (或其他关键字) 和 `by` 之间.
`…Distinct…`	使用 distinct 查询返回唯一的结果. 请查阅特定的文档以了解是否支持该功能. 此关键字可以出现在主题 `find` (或其他关键字) 和 `by` 之间.

支持的查询方法断言关键字和修饰符

下表列出了 Spring Data 存储库查询扩展机制通常支持的断言关键字. 但是,请参阅 store-specific 的文档以获取受支持关键字的确切列表,因为 store-specific 可能不支持此处列出的某些关键字.

Table 9. 查询断言关键字

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`, (or no keyword)

逻辑关键字	关键字表达
`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`

除过滤断言外,还支持以下修饰符列表:

Table 10. 查询断言修饰符关键字

关键字	描述
`IgnoreCase`, `IgnoringCase`	与福安艳关键字一起使用,不区分大小写的比较.
`AllIgnoreCase`, `AllIgnoringCase`	忽略所有属性的大小写. 在查询方法断言中的某处使用.
`OrderBy…`	指定一个静态的排序顺序,后面跟属性的 path 和 方向 (例如. `OrderByFirstnameAscLastnameDesc`).

Appendix 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 Data 存储库通常支持的返回类型。但是,请参阅 store-specific 的文档以获取受支持的退货类型的确切列表,因为特定 存储 可能不支持此处列出的某些类型。

地理空间类型（例如 `GeoResult`,`GeoResults` 和 `GeoPage`）仅适用于支持地理空间查询的数据存储。

某些存储模块可能会定义自己的结果包装器类型。

Table 11. 查询返回类型

返回类型	描述
`void`	表示没有返回值。
Primitives	Java 原语。
Wrapper types	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,则返回 `null`。一个以上的结果触发一个 `IncorrectResultSizeDataAccessException`。
`Iterator<T>`	`Iterator`。
`Collection<T>`	`Collection`。
`List<T>`	`List`。
`Optional<T>`	Java 8或 `Guava` 可选。期望查询方法最多返回一个结果。如果未找到结果,则返回 `Optional.empty()` 或 `Optional.absent()`。一个以上的结果触发一个 `IncorrectResultSizeDataAccessException`。
`Option<T>`	Scala或Vavr `Option` 类型。语义上与前面描述的Java 8的 `Optional` 行为相同。
`Stream<T>`	Java 8 `Stream`。
`Streamable<T>`	`Iterable` 的便捷扩展,直接将方法暴露以流式处理,映射和过滤结果,将其串联等。
Types that implement `Streamable` and take a `Streamable` constructor or factory method argument	暴露构造函数或使用Streamable作为参数的 `….of(…)`/`….valueOf(…)` 工厂方法的类型。有关详细信息,请参见返回自定义流式包装器类型。
Vavr `Seq`, `List`, `Map`, `Set`	Vavr集合类型。有关详细信息,请参见 支持Vavr集合
`Future<T>`	`Future`。期望使用 `@Async` 注解方法,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 `@Async` 注解方法,并且需要启用Spring的异步方法执行功能。
`ListenableFuture`	`org.springframework.util.concurrent.ListenableFuture`。期望使用 `@Async` 注解方法,并且需要启用Spring的异步方法执行功能。
`Slice<T>`	一定大小的数据块,用于指示是否有更多可用数据。需要 `Pageable` 方法参数。
`Page<T>`	具有附加信息（例如结果总数）的 `Slice`。需要 `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>`	包含其他信息的 `GeoResult<T>` 列表,例如到参考位置的平均距离。

返回类型	描述
`GeoPage<T>`	具有 `GeoResult<T>` 的页面,例如到参考位置的平均距离.
`Mono<T>`	使用 Reactor 储存库发射零或一个元素的Project Reactor Mono. 期望查询方法最多返回一个结果. 如果未找到结果,则返回 `Mono.empty()`. 一个以上的结果触发一个 `IncorrectResultSizeDataAccessException`.
`Flux<T>`	使用 Reactor 存储库发射零,一个或多个元素的Project Reactor通量. 返回 `Flux` 的查询也可以发出无限数量的元素.
`Single<T>`	使用 Reactor 存储库发出 `Single` 元素的RxJava Single. 期望查询方法最多返回一个结果. 如果未找到结果,则返回 `Mono.empty()`. 一个以上的结果触发一个 `IncorrectResultSizeDataAccessException`.
`Maybe<T>`	RxJava可能使用 Reactor 存储库发出零个或一个元素. 期望查询方法最多返回一个结果. 如果未找到结果,则返回 `Mono.empty()` . 一个以上的结果触发一个 `IncorrectResultSizeDataAccessException`.
`Flowable<T>`	RxJava `Flowable` 使用响应式存储库发出零个,一个或多个元素. 返回 `Flowable` 的查询也可以发出无限数量的元素.

Appendix E: 常见问题

通用

1. 我想获得更详细的日志记录信息，例如有关在 JpaRepository 内部调用哪些方法的信息。我如何获得他们？

   您可以使用 Spring 提供的 CustomizableTraceInterceptor，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

1. 目前，我已经基于 *HibernateDaoSupport* 实现了一个存储库层。我使用 Spring 的AnnotationSessionFactoryBean 创建一个 SessionFactory。如何在这种环境中使用 *Spring Data* 存储库？

   您必须使用 HibernateJpaSessionFactoryBean 替换 AnnotationSessionFactoryBean，如下所示：

   Example 119. 从一个 HibernateEntityManagerFactory 查找一个 SessionFactory

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

审计

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 auditing 命名空间元素的 set-dates 属性设置为 false。

Appendix F: 词汇表

AOP	面向切面的编程
Commons DBCP	Commons DataBase Connection Pools-来自Apache基础的库,提供DataSource接口的池实现.
CRUD	创建,读取,更新,删除-基本持久性操作.
DAO	数据访问对象-用于将持久逻辑与要持久的对象分离的模式
Dependency Injection	从外部将组件的依赖传递给组件的模式,以释放组件以查找依赖本身. 有关更多信息,请参见en.wikipedia.org/wiki/Dependency_Injection.
EclipseLink	实现JPA的对象关系映射器- www.eclipse.org/eclipselink/
Hibernate	实现JPA的对象关系映射器 - hibernate.org/
JPA	Java持久性API
Spring	Java应用程序框架 - projects.spring.io/spring-framework