Spring Data Elasticsearch - 参考文档

BioMed Central Development Team, Oliver Drotbohm, Greg Turnquist, Christoph Strobl, Peter-Josef Meisch

2021-09-13

Table of Contents

前言

Spring Data Elasticsearch 项目将 Spring 的核心概念应用于使用 Elasticsearch 搜索引擎的解决方案的开发中. 它提供:

- 一个用于存储,搜索,排序以及构建聚合的高级抽象 *Templates*
- 例如,*Repositories* 允许用户自定义接口方法来传达查询方法 (有关 repositories 的基础,请查看 使用 Spring Data Repositories).

你会注意到这与 Spring Framework 中的 Spring data solr 和 mongodb 非常相似

Chapter 1. 新增功能

1.1. New in Spring Data Elasticsearch 4.1

- 使用 Spring 5.3.
- 更新版本至 Elasticsearch 7.9.3.
- 改进 别名管理 API.
- 引入用于索引管理的 `ReactiveIndexOperations` .
- Index templates support.
- Support for Geo-shape data with GeoJson.

1.2. Spring Data Elasticsearch 4.0 新增功能

- 使用 Spring 5.2.
- 更新版本至 Elasticsearch 7.6.2.
- 弃用 `TransportClient`.
- 实现大多数可用于索引映射的映射类型.
- 删除 Jackson `ObjectMapper`, 现在使用 `MappingElasticsearchConverter`
- 清理 `*Operations` interfaces API, 对方法进行分组和重命名,使他们与 Elasticsearch API 匹配, 弃用旧方法, 并与其他 Spring Data 模块版本对其.
- 引入 `SearchHit<T>` 类来表示找到的文档以及该文档的相关元数据(i.e. *sortValues*).
- 引入 `SearchHits<T>` 类来表示整个搜索结果以及完整搜索结果的元数据 (i.e. *max_score*).
- 引入 `SearchPage<T>` 类以表示包含 `SearchHits<T>` 实例的分页结果.
- 引入 `GeoDistanceOrder` 类以便能根据地理距离进行排序
- 实现审计支持
- 实现 entity 生命周期回调

1.3. Spring Data Elasticsearch 3.2 新增功能

- 提供基于 Basic Authentication 和 SSL 传输的 Elasticsearch 集群支持.
- 更新版本至 Elasticsearch 6.8.1.
- 通过 Reactive Elasticsearch 操作 和 Reactive Elasticsearch Repositories 支持 Reactive programming.
- 引入 ElasticsearchEntityMapper 作为 Jackson ObjectMapper 的替代方案.
- @Field 自定义字段名称.
- 支持按查询删除.

Chapter 2. 项目元数据

- Version Control - github.com/spring-projects/spring-data-elasticsearch
- API Documentation - docs.spring.io/spring-data/elasticsearch/docs/current/api/
- Bugtracker - jira.spring.io/browse/DATAES
- Release repository - repo.spring.io/libs-release
- Milestone repository - repo.spring.io/libs-milestone
- Snapshot repository - repo.spring.io/libs-snapshot

Chapter 3. 要求

需要安装 Elasticsearch.

3.1. 版本

下表显示了 Spring Data 发布使用的 Elasticsearch 版本以及其中包含的 Spring Data Elasticsearch 的版本,以及引用该特定 Spring Data 发布的 Spring Boot 版本:

Spring Data Release Train	Spring Data Elasticsearch	Elasticsearch	Spring Boot
2020.0.0[1]	4.1.x[1]	7.9.3	2.4.x[1]
Neumann	4.0.x	7.6.2	2.3.x
Moore	3.2.x	6.8.12	2.2.x
Lovelace	3.1.x	6.2.2	2.1.x
Kay[2]	3.0.x[2]	5.5.0	2.0.x[2]
Ingalls[2]	2.1.x[2]	2.4.0	1.5.x[2]

持续跟进对即将推出的 Elasticsearch 版本的支持,并且提供使用 高级 REST client 的兼容性

[1] Currently in development

[2] Out of maintenance

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 1. 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 2. 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 3. 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 4. 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.EnableJpaRepositor
ies;

@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 5. 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 ,存储库的定义必须区分要使用哪个. 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:

1. 如果存储库扩展了特定于指定模块的存储库,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 6. 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 7. 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 8. 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 domain 类的存储库:

Example 9. 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 的配置中的 basePackages 是必需的.

以下示例显示了基础包的注解驱动配置:

Example 10. 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 11. 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 12. 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.

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 13. 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 14. 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());
```

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 15. 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 16. 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 17. 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 18. 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 19. 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 20. 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 21. 用 *Java 8* `Stream<T>` 流查询的结果

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

Stream<User> readAllByFirstnameNotNull();

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

`Stream` 可能包装了特定于底层数据存储的资源，因此必须在使用后关闭。
您可以使用 `close()` 方法或使用 Java 7 `try-with-resources`
块来手动关闭 `Stream`，如以下示例所示：

Example 22. Stream<T> 的结果使用 try-with-resources 块

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

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

4.4.9. 异步查询结果

使用 Spring 的异步方法执行功能,可以异步运行存储库查询.
这意味着该方法在调用时立即返回,而实际查询执行发生在已提交给 Spring TaskExecutor 的任务中. 异步查询执行与响应式查询执行不同,因此不应混为一谈.
有关响应式支持的更多详细信息,请参阅存储特定的文档. 以下示例显示了许多异步查询:

```
@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 23. 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 24. 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 25. 基于注解的存储卡示例

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

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

前面的示例使用特定于 JPA 的注解 ,您将根据实际使用的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 26. 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 27. 定制 repository 功能的接口

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

Example 28. 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 29. 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 30. 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 31. 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 32. 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 33. 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 34. 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 35. 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 36. 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 37. 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 38. 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 39. 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 40. 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(…)`, `saveAll(…)`, `delete(…)` or `deleteAll(…)` 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 41. 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 42. 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

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 43. 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 44. 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.HateoasAwareSpringDataWebConfig
uration" />
```

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.
- Jackson Modules 序列化或反序列化类似 Point 和 Distance 的类型, 或者其他特定的类型, 主要由您使用的 Spring Data Module 决定.

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 45. 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 46. 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 47. 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.

Spring Data Jackson Modules

core module 和 存储库特定的模块附带了一组用于类型的 Jackson 模块, 例如 org.springframework.data.geo.Distance 和 org.springframework.data.geo.Point, 使用 Spring Data domain. 一旦启用 web support 这些模块将被导入, 并且 com.fasterxml.jackson.databind.ObjectMapper 可用.

在初始化期间, 像 SpringDataJacksonConfiguration 一样, SpringDataJacksonModules 会被自动检测, 以便声明的 com.fasterxml.jackson.databind.Module 可供 Jackson 的 ObjectMapper 使用.

Data binding mixins for the following domain types are registered by the common infrastructure.

```
org.springframework.data.geo.Distance
org.springframework.data.geo.Point
org.springframework.data.geo.Box
org.springframework.data.geo.Circle
org.springframework.data.geo.Polygon
```

各个模块可以提供附加的 SpringDataJacksonModules.
请参阅存储库特定部分以获取更多详细信息.

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 48. 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 49. 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 50. 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 51. 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>
```

4.9. 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 52. 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.

4.9.1. 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 53. 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 54. 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 55. 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 56. 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 57. 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 58. 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 59. 简单 *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 60. 简单 *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.

Nullable Wrappers

投影接口中的 getter 可以使用可为空的包装器, 以提高 null-safety 的安全性.
当前支持的包装器类型为:

- java.util.Optional
- com.google.common.base.Optional
- scala.Option
- io.vavr.control.Option

Example 61. 使用 *nullable* 包装的投影接口

```
interface NamesOnly {

  Optional<String> getFirstname();
}
```

如果基础投影值不为 null, 则使用包装器类型的 present-representation 返回值.
如果默认值为 null, 则 getter 方法将返回使用的包装器类型的空表示形式.

4.9.2. 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 62. 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。

4.9.3. 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 63. 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 64. 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);
}
```

参考文档

Chapter 5. Elasticsearch 客户端

本章介绍了如何配置和使用支持的 Elasticsearch 客户端的实现

Spring Data Elasticsearch 在连接到单个 Elasticsearch 节点或集群的 Elasticsearch 客户端上进行操作. 虽然可以使用 Elasticsearch Client 客户端与集群一起工作, 但使用 Spring Data Elasticsearch 通常使用更加高级的抽象 Elasticsearch 操作 和 Elasticsearch Repositories.

5.1. Transport Client

众所周知,从 Elasticsearch 7 开始 TransportClient 已经被弃用,并且在 Elasticsearch 8 中删除. (see the Elasticsearch documentation). Spring Data Elasticsearch 支持 TransportClient 只要它在已使用的 Elasticsearch 版本中可用,但从4.0版本开始已弃用使用它的类.

强烈建议使用 更高级别的 REST Client 而不是 TransportClient.

Example 65. Transport Client

```
@Configuration
public class TransportClientConfig extends
ElasticsearchConfigurationSupport {

    @Bean
    public Client elasticsearchClient() throws UnknownHostException {
        Settings settings = Settings.builder().put("cluster.name",
"elasticsearch").build();        ①
        TransportClient client = new PreBuiltTransportClient(settings);
        client.addTransportAddress(new
TransportAddress(InetAddress.getByName("127.0.0.1"), 9300)); ②
        return client;
    }

    @Bean(name = { "elasticsearchOperations", "elasticsearchTemplate" })
    public ElasticsearchTemplate elasticsearchTemplate() throws
UnknownHostException {
        return new ElasticsearchTemplate(elasticsearchClient());
    }
}

// ...

IndexRequest request = new IndexRequest("spring-data", "elasticsearch",
randomID())
 .source(someObject)
 .setRefreshPolicy(IMMEDIATE);

IndexResponse response = client.index(request);
```

① 必须使用群集名称配置 `TransportClient`.

② 连接客户端的主机和端口.

5.2. 更高级别的 REST Client

Java High Level REST Client 是 Elasticsearch 默认的客户端, 它可以直接替代 `TransportClient` 因为他们接受并返回完全相同的请求和响应.因此依赖于 Elasticsearch core 项目 异步调用是在客户端管理的线程池上进行的,并且要求在完成请求时通知回调.

Example 66. High Level REST Client

```
@Configuration
public class RestClientConfig extends AbstractElasticsearchConfiguration {

    @Override
    @Bean
    public RestHighLevelClient elasticsearchClient() {

        final ClientConfiguration clientConfiguration =
ClientConfiguration.builder()  ①
            .connectedTo("localhost:9200")
            .build();

        return RestClients.create(clientConfiguration).rest();
②
    }
}

// ...

  @Autowired
  RestHighLevelClient highLevelClient;

  RestClient lowLevelClient = highLevelClient.lowLevelClient();
③

// ...

IndexRequest request = new IndexRequest("spring-data", "elasticsearch",
randomID())
  .source(singletonMap("feature", "high-level-rest-client"))
  .setRefreshPolicy(IMMEDIATE);

IndexResponse response = highLevelClient.index(request);
```

① 使用 builder 提供集群地址，设置默认的 HttpHeaders 或 启用 SSL.

② 创建 RestHighLevelClient.

③ 也可以获取 lowLevelRest() 客户端.

5.3. Reactive Client

`ReactiveElasticsearchClient` 是一个基于 `WebClient` 的非官方驱动程序.它使用 Elasticsearch core 项目提供的请求/响应对象.调用直接在响应式堆栈上操作,而不是将异步(线程池绑定)响应包装为响应式类型.

Example 67. Reactive REST Client

```
static class Config {

  @Bean
  ReactiveElasticsearchClient client() {

    ClientConfiguration clientConfiguration =
ClientConfiguration.builder()   ①
      .connectedTo("localhost:9200", "localhost:9291")
      .withWebClientConfigurer(webClient -> {
②
        ExchangeStrategies exchangeStrategies =
ExchangeStrategies.builder()
            .codecs(configurer -> configurer.defaultCodecs()
                .maxInMemorySize(-1))
            .build();
        return
webClient.mutate().exchangeStrategies(exchangeStrategies).build();
       })
      .build();

    return ReactiveRestClients.create(clientConfiguration);
  }
}

// ...

Mono<IndexResponse> response = client.index(request ->

  request.index("spring-data")
    .type("elasticsearch")
    .id(randomID())
    .source(singletonMap("feature", "reactive-client"))
    .setRefreshPolicy(IMMEDIATE);
);
```

① 使用 `builder` 提供集群地址, 设置默认的 `HttpHeaders` 或 启用 `SSL`.

② 当配置一个响应式客户端时,可以使用 `withWebClientConfigurer` 钩子来自定义 `web` 客户端.

ReactiveClient 响应, (特别是搜索操作)绑定到请求的 from (offset) & size (limit) 选项.

5.4. Client 配置

客户端行为可以通过 ClientConfiguration 更改,该配置允许设置 SSL、connect 和 socket timeouts, headers 和其他参数的选项.

Example 68. Client Configuration

```
HttpHeaders httpHeaders = new HttpHeaders();
httpHeaders.add("some-header", "on every request")                    ①

ClientConfiguration clientConfiguration = ClientConfiguration.builder()
  .connectedTo("localhost:9200", "localhost:9291")                    ②
  .useSsl()                                                           ③
  .withProxy("localhost:8888")                                        ④
  .withPathPrefix("ela")                                              ⑤
  .withConnectTimeout(Duration.ofSeconds(5))                          ⑥
  .withSocketTimeout(Duration.ofSeconds(3))                           ⑦
  .withDefaultHeaders(defaultHeaders)                                 ⑧
  .withBasicAuth(username, password)                                  ⑨
  .withHeaders(() -> {                                                ⑩
    HttpHeaders headers = new HttpHeaders();
    headers.add("currentTime",
LocalDateTime.now().format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
    return headers;
  })
  . // ... other options
  .build();
```

① 定义默认的 headers, 如有需要,可以自定义

② 使用 builder 提供集群机制, 设置默认 HttpHeaders 或 启用 SSL.

③ 可选的,启用 SSL.

④ 可选的,设置代理.

⑤ 可选的,设置路径前缀, 主要用于不同的集群在某个反向代理后面.

⑥ 设置连接超时.默认值为 10 秒.

⑦ 设置 socket 超时.默认值为 5 秒

⑧ 可选的.设置 headers.

⑨ 添加 basic 认证.

⑩ 可以指定一个 Supplier<Header> 函数,该函数在每次请求发送到 Elasticsearch 之前都会被调用——例如,上例中,当前时间被写入 header 中.

如上例所示,可以将随时间变化请求头的注入到 Supplier 中,例如身份验证 JWT 令牌. 如果这是在响应式中使用的,supplier 函数一定不能阻塞!

5.5. Client 日志

要查看实际发送到服务器和从服务器接收的内容,请按照以下代码片段中的说明打开在 transport 级别的 Request / Response 日志记录.

Enable transport layer logging

```
<logger name="org.springframework.data.elasticsearch.client.WIRE"
level="trace"/>
```

当分别通过 RestClients 和 ReactiveRestClients 获得时,以上内容适用于 RestHighLevelClient 和 ReactiveElasticsearchClient,不适用于 TransportClient.

Chapter 6. Elasticsearch 对象映射

Spring Data Elasticsearch Object Mapping 是将 Java 对象 (domain entity)和存储在 Elasticsearch 中的 JSON 相互映射的过程

早期的 Spring Data Elasticsearch 版本使用 Jackson 进行转换,Spring Data Elasticsearch 3.2.x 引入了 Meta Model Object Mapping 对象映射,基于 Jackson 的映射不再使用,而是使用 MappingElasticsearchConverter

删除基于 Jackson 的映射的主要原因是:

- 自定义字段的映射需要使用 @JsonFormat 或 @JsonInclude 之类的注解来完成. 当在不同的基于 JSON 的数据存储中使用相同的对象或使用 JSON 的 API 发送时 .这通常会导致问题
- 自定义字段类型和格式也需要存储到 Elasticsearch 索引映射中. 基于 Jackson 的注解未完全提供表示 Elasticsearch 类型所需的所有信息..
- 不仅仅在实体之间转换需要字段映射,还必须在查询参数,返回数据以及其他地方映射字段.

使用 MappingElasticsearchConverter 现在覆盖了所有这些情况.

6.1. Meta Model Object Mapping

基于元模型的方法使用 domain 类型信息 读取/写入 Elasticsearch.这允许为特定的 domain 类型映射注册 Converter 实例.

6.1.1. 映射注解概述

MappingElasticsearchConverter 使用元数据来驱动对象到文档的映射 .元数据来自可以注解的实体属性.

有以下注解可用:

- @Document: 类级别注解,表示此类是映射到数据库的候选对象.以下为最重要的属性.:
  - indexName: 此实体的索引名称

  - type: ~~映射类型. 如果未设置,则使用类的简单小写名称.~~ (从版本4.0开始不推荐使用)
  - shards: 索引分片的数量
  - replicas: 索引副本数量
  - refreshIntervall: 索引刷新间隔.用于创建索引. 默认为 *"1s"*.
  - indexStoreType: 索引的索引存储类型.用于创建索引. 默认为 *"fs"*.
  - createIndex: 配置是否在 repository 加载时创建索引. 默认为 *true*.
  - versionType: 版本管理配置. 默认为 *EXTERNAL*.
- @Id: 字段级注解,标记字段本身的意图.
- @Transient: 默认情况下,存储或检索文档时,所有的字段都映射到文档中,此字段可以标识不映射该字段
- @PersistenceConstructor: 标记已有的构造函数 - 可以是一个 protected package - 当从数据库实例化对象时使用.按照构造函数参数名称映射到检索到的文档中的键值.
- @Field: 字段级注解,可以定义字段的属性, 大多数属性映射到各自的 Elasticsearch Mapping 定义 (以下列表不完整,请查看注解的 Javadoc 以获取完整的参考) :
  - name: 将在 Elasticsearch 文档中表示的字段名称,如果未设置,则使用 Java 字段名称.
  - type: 字段类型,可以是 *Text*, *Keyword*, *Long*, *Integer*, *Short*, *Byte*, *Double*, *Float*, *Half_Float*, *Scaled_Float*, *Date*, *Date_Nanos*, *Boolean*, *Binary*, *Integer_Range*, *Float_Range*, *Long_Range*, *Double_Range*, *Date_Range*, *Ip_Range*, *Object*, *Nested*, *Ip*, *TokenCount*, *Percolator*, *Flattened*, *Search_As_You_Type*. 请参阅 Elasticsearch 映射类型
  - 为 *Date* 类型声明 format 和 pattern . 日期类型必须指定 format .
  - store: 标记是否将原始字段值存储在 Elasticsearch 中, 默认为 *false*.
  - analyzer, searchAnalyzer, normalizer 指定自定义 analyzers 和 normalizer.

- `@GeoPoint`: 将字段标记为 *geo_point* 数据类型. 如果字段是 `GeoPoint` 类的实例,则可以省略.

从 `TemporalAccessor` 扩展 的属性必须具有类型为 `FieldType.Date` 的 `@Field` 注解,或必须为此类型注册自定义转换器.
如果使用自定义日期格式,则需要使用 *uuuu* 作为年份而不是 *yyyy*. 这是 Elasticsearch 7 中的变化.

映射元数据基础架构是在 `spring-data-commons` 项目中定义的,该项目与技术无关..

6.1.2. 映射规则

Type Hints(类型提示)

映射使用发送到服务器的文档中的类型提示来允许通用类型映射. 这些类型提示在文档中表示为 `_class` 属性,并针对每个聚合根写入.

Example 69. Type Hints

```
public class Person {                ①

  @Id String id;
  String firstname;
  String lastname;
}
```

```
{
  "_class" : "com.example.Person", ①
  "id" : "cb7bef",
  "firstname" : "Sarah",
  "lastname" : "Connor"
}
```

① 默认情况下,`domain` 类型类名用于类型提示.

类型提示可以配置为保存自定义信息.. 使用 `@TypeAlias` 注解执行此操作.

请确保将具有 `@TypeAlias` 的类型添加到初始实体集(`AbstractElasticsearchConfiguration#getInitialEntitySet`)中,以便在首次从存储中读取数据时已经具有可用的实体信息.

Example 70. Type Hints with Alias

```
@TypeAlias("human")                ①
public class Person {

  @Id String id;
  // ...
}
```

```
{
  "_class" : "human",              ①
  "id" : ...
}
```

① 编写实体时使用配置的别名.

除非属性类型为 `Object`, 接口或实际值类型与属性声明不匹配,否则不会为嵌套对象编写类型提示.

`Geospatial` 类型

`Geospatial` 类型,比如 `Point` & `GeoPoint` 将被转换为 *`lat/lon`* 对.

Example 71. Geospatial 类型

```
public class Address {

  String city, street;
  Point location;
}
```

```
{
  "city" : "Los Angeles",
  "street" : "2800 East Observatory Road",
  "location" : { "lat" : 34.118347, "lon" : -118.3026284 }
}
```

Collections(集合)

对于集合中的值,在类型提示和 自定义转换 时,与聚合根具有相同的映射规则.

Example 72. Collections

```
public class Person {

  // ...

  List<Person> friends;

}
```

```
{
  // ...

  "friends" : [ { "firstname" : "Kyle", "lastname" : "Reese" } ]
}
```

Maps

对于 Maps 内的值,在类型提示和 自定义转换 时,与聚合根具有相同的映射规则.然而,Map 的

键 需要一个字符串来由 Elasticsearch 处理.

Example 73. Collections

```
public class Person {

  // ...

  Map<String, Address> knownLocations;

}
```

```
{
  // ...

  "knownLocations" : {
    "arrivedAt" : {
       "city" : "Los Angeles",
       "street" : "2800 East Observatory Road",
       "location" : { "lat" : 34.118347, "lon" : -118.3026284 }
     }
  }
}
```

6.1.3. 自定义转换

看看 上一节中的 Configuration,ElasticsearchCustomConversions 允许为 mapping domain 和简单类型注册特定规则.

Example 74. Meta Model Object Mapping Configuration

```
@Configuration
public class Config extends AbstractElasticsearchConfiguration {

  @Override
  public RestHighLevelClient elasticsearchClient() {
    return
RestClients.create(ClientConfiguration.create("localhost:9200")).rest();
  }

  @Bean
  @Override
  public ElasticsearchCustomConversions elasticsearchCustomConversions() {
    return new ElasticsearchCustomConversions(
      Arrays.asList(new AddressToMap(), new MapToAddress()));       ①
  }

  @WritingConverter                                                 ②
  static class AddressToMap implements Converter<Address, Map<String,
Object>> {

    @Override
    public Map<String, Object> convert(Address source) {

      LinkedHashMap<String, Object> target = new LinkedHashMap<>();
      target.put("ciudad", source.getCity());
      // ...

      return target;
    }
  }

  @ReadingConverter                                                 ③
  static class MapToAddress implements Converter<Map<String, Object>,
Address> {

    @Override
    public Address convert(Map<String, Object> source) {

      // ...
      return address;
    }
  }
}
```

```
{
  "ciudad" : "Los Angeles",
  "calle" : "2800 East Observatory Road",
  "localidad" : { "lat" : 34.118347, "lon" : -118.3026284 }
}
```

① 添加 `Converter` 实现.

② 设置将 `DomainType` 类型写入到 Elasticsearch 的 `Converter`.

③ 设置从搜索结果中读取到 `DomainType` 类型的 `Converter`.

Chapter 7. Elasticsearch 操作

Spring Data Elasticsearch 使用几个接口来定义对 Elasticsearch 索引调用的操作(有关响应式接口的描述,请参阅 Reactive Elasticsearch 操作).

- IndexOperations 定义索引的动作,例如创建和删除索引
- DocumentOperations 定义根据实体 id 存储,更新和检索文档实体的操作
- SearchOperations 定义使用查询来搜索多个实体的动作
- ElasticsearchOperations 结合了 DocumentOperations 和 SearchOperations 接口.

这些接口与 Elasticsearch API 相对应.

接口的默认实现提供:

- 管理索引.
- domain 类型的读写映射.
- 丰富的查询和条件 API.
- 资源管理和异常转化.

索引管理以及自动创建索引和映射.

可以从 ElasticsearchOperations 接口中获得 IndexOperations 接口和其实现类 - 例如,调用 operations.indexOps(clazz)- 可以使用户能够创建索引, put mappings 或 store template 和 Elasticsearch 集群中的别名信息.

这些操作都不是由 IndexOperations 或 ElasticsearchOperations 的实现自动完成的.
调用方法是用户的责任.

使用 Spring Data Elasticsearch repositories 时支持自动创建索引和写入映射, 请参见 自动创建具有相应映射的索引

7.1. ElasticsearchTemplate

从 4.0 版开始不推荐使用 ElasticsearchTemplate ，建议使用 ElasticsearchRestTemplate 替代.

ElasticsearchTemplate 使用了 Transport Client ,并实现了 ElasticsearchOperations 接口.

Example 75. ElasticsearchTemplate 配置

```
@Configuration
public class TransportClientConfig extends
ElasticsearchConfigurationSupport {

  @Bean
  public Client elasticsearchClient() throws UnknownHostException {
①
    Settings settings = Settings.builder().put("cluster.name",
"elasticsearch").build();
    TransportClient client = new PreBuiltTransportClient(settings);
    client.addTransportAddress(new
TransportAddress(InetAddress.getByName("127.0.0.1"), 9300));
    return client;
  }

  @Bean(name = {"elasticsearchOperations", "elasticsearchTemplate"})
  public ElasticsearchTemplate elasticsearchTemplate() throws
UnknownHostException { ②
    return new ElasticsearchTemplate(elasticsearchClient());
  }
}
```

① 设置 Transport Client. 从 4.0 版本开始废弃.

② 创建 ElasticsearchTemplate bean, 并提供两个命名, *elasticsearchOperations* 和 *elasticsearchTemplate*.

7.2. ElasticsearchRestTemplate

ElasticsearchRestTemplate 使用了 更高级别的 REST Client 并实现了

ElasticsearchOperations 接口.

Example 76. ElasticsearchRestTemplate 配置

```
@Configuration
public class RestClientConfig extends AbstractElasticsearchConfiguration {
  @Override
  public RestHighLevelClient elasticsearchClient() {       ①
    return RestClients.create(ClientConfiguration.localhost()).rest();
  }

  // no special bean creation needed                       ②
}
```

① 设置 更高级别的 REST Client.

② AbstractElasticsearchConfiguration 已经提供了 elasticsearchTemplate bean.

7.3. 使用案例

因为 ElasticsearchTemplate 和 ElasticsearchRestTemplate 都实现了 ElasticsearchOperations 接口,所以在使用它们时的代码没有什么不同 .这个例子展示了如何在 Spring REST 控制器中使用 ElasticsearchOperations 实例 .如果使用的是 TransportClient 或 RestClient,则可以通过为相应的 Bean 提供上面所示的配置之一来做出决定.

Example 77. ElasticsearchOperations usage

```
@RestController
@RequestMapping("/")
public class TestController {

  private  ElasticsearchOperations elasticsearchOperations;

  public TestController(ElasticsearchOperations elasticsearchOperations) {
①
    this.elasticsearchOperations = elasticsearchOperations;
  }

  @PostMapping("/person")
  public String save(@RequestBody Person person) {
②

    IndexQuery indexQuery = new IndexQueryBuilder()
      .withId(person.getId().toString())
      .withObject(person)
      .build();
    String documentId = elasticsearchOperations.index(indexQuery);
    return documentId;
  }

  @GetMapping("/person/{id}")
  public Person findById(@PathVariable("id")  Long id) {
③
    Person person = elasticsearchOperations
      .queryForObject(GetQuery.getById(id.toString()), Person.class);
    return person;
  }
}
```

① 在构造函数中注入 `ElasticsearchOperations` bean.

② 在 Elasticsearch 集群中存储一些 entity.

③ 通过 id 搜索 entity.

要查看 `ElasticsearchOperations` 的全部功能,请参阅 API 文档.

7.4. Reactive Elasticsearch 操作

ReactiveElasticsearchOperations 是使用 ReactiveElasticsearchClient 对 Elasticsearch 集群执行高级命令的网关..

ReactiveElasticsearchTemplate 是 ReactiveElasticsearchOperations 的默认实现.

7.4.1. Reactive Elasticsearch Template

在使用 ReactiveElasticsearchTemplate 之前,需要先了解要使用的实际客户端,有关客户端的详细信息,请参见 Reactive Client .

Reactive Template 配置

配置 ReactiveElasticsearchTemplate 最简单的方法就是通过 AbstractReactiveElasticsearchConfiguration 提供的 配置方法配置 base package, initial entity set 等.

Example 78. The AbstractReactiveElasticsearchConfiguration

```
@Configuration
public class Config extends AbstractReactiveElasticsearchConfiguration {

  @Bean ①
  @Override
  public ReactiveElasticsearchClient reactiveElasticsearchClient() {
      // ...
  }
}
```

① 配置要使用的客户端. 这可以由 ReactiveRestClients 或直接通过 DefaultReactiveElasticsearchClient 完成.

如果要适用默认的`HttpHeaders` ,可以通过 ReactiveElasticsearchClient 的 ClientConfiguration 设置. 请参考 Client 配置.

如果需要配置 ReactiveElasticsearchTemplate 默认的 RefreshPolicy 和 IndicesOptions ,可以通过覆盖 refreshPolicy() 和 indicesOptions() 方法设置.

但是,可能需要更多地控制实际组件,并使用更详细的方法.

Example 79. Configure the ReactiveElasticsearchTemplate

```
@Configuration
public class Config {

  @Bean ①
  public ReactiveElasticsearchClient reactiveElasticsearchClient() {
    // ...
  }
  @Bean ②
  public ElasticsearchConverter elasticsearchConverter() {
    return new
MappingElasticsearchConverter(elasticsearchMappingContext());
  }
  @Bean ③
  public SimpleElasticsearchMappingContext elasticsearchMappingContext() {
    return new SimpleElasticsearchMappingContext();
  }
  @Bean ④
  public ReactiveElasticsearchOperations reactiveElasticsearchOperations()
{
    return new
ReactiveElasticsearchTemplate(reactiveElasticsearchClient(),
elasticsearchConverter());
  }
}
```

① 配置要使用的客户端. 这可以由 ReactiveRestClients 或直接通过 DefaultReactiveElasticsearchClient 完成.

② 使用映射上下文提供的元数据设置来提供 doamin 类型映射的 ElasticsearchConverter.

③ 特定于 Elasticsearch 的 domain 类型元数据映射上下文.

④ 基于 client 和 conversion 结构的真实 template.

Reactive Template 使用

ReactiveElasticsearchTemplate 使您可以保存,查找和删除 domain 对象,并将这些对象映射到存储在 Elasticsearch 中的文档..

如下:

Example 80. Use the ReactiveElasticsearchTemplate

```
@Document(indexName = "marvel")
public class Person {

  private @Id String id;
  private String name;
  private int age;
  // Getter/Setter omitted...
}
```

```
template.save(new Person("Bruce Banner", 42))                    ①
  .doOnNext(System.out::println)
  .flatMap(person -> template.findById(person.id, Person.class)) ②
  .doOnNext(System.out::println)
  .flatMap(person -> template.delete(person))                    ③
  .doOnNext(System.out::println)
  .flatMap(id -> template.count(Person.class))                    ④
  .doOnNext(System.out::println)
  .subscribe(); ⑤
```

上面的输出如下.

```
> Person(id=QjWCWWcBXiLAnp77ksfR, name=Bruce Banner, age=42)
> Person(id=QjWCWWcBXiLAnp77ksfR, name=Bruce Banner, age=42)
> QjWCWWcBXiLAnp77ksfR
> 0
```

① 将一个新的 `Person` 文档插入到 *marvel* 索引下,类型为字符串. `id` 由服务端自动生成并返回实例.

② 在 *marvel* 索引中查找匹配 `id` 的 `Person`.

③ 在 *marvel* 索引中删除匹配 `id` 的 `Person`.

④ 计算 *marvel* 索引下的文档总数.

⑤ 不要忘记了 *subscribe()*.

7.5. 搜索结果类型

使用 DocumentOperations 接口的方法搜索文档时,只返回匹配的实体. 使用 SearchOperations 接口的方法进行搜索时,每个实体都有额外的其他附加信息,例如 ,找到的的实体具有 *score* 或 *sortValues*.

为了返回此信息,每个实体都包装在一个 SearchHit 对象中 ,该对象包含该特定于实体的附加信息. 这些 SearchHit 对象本身在 SearchHits 对象中返回,该对象还包含有关整个搜索的信息,例如 *maxScore* 或请求的聚合.
现在可以使用以下类和接口:

SearchHit<T>

包含以下信息:

- Id
- Score
- Sort Values
- Highlight fields
- Inner hits (this is an embedded SearchHits object containing eventually returned inner hits)
- The retrieved entity of type <T>

SearchHits<T>

包含以下信息:

- Number of total hits
- Total hits relation
- Maximum score
- A list of SearchHit<T> objects
- Returned aggregations

SearchPage<T>

定义一个包含 SearchHits<T> 的 Spring Data Page
可以使用存储库方法进行分页访问..

SearchScrollHits<T>

由 ElasticsearchRestTemplate 中的低级 scroll API 函数返回,它使用 Elasticsearch scroll ID 丰富了 SearchHits<T>.

SearchHitsIterator<T>

由 SearchOperations 接口返回一个可迭代的流(Stream).

7.6. Queries

几乎在 SearchOperations 和 ReactiveSearchOperations 接口中定义的所有方法都采用 Query 参数, 该参数定义要执行的查询以进行搜索. Query 是一个接口, Spring Data Elasticsearch 提供了三种实现: CriteriaQuery, StringQuery 和 NativeSearchQuery.

7.6.1. CriteriaQuery

基于 CriteriaQuery 的查询允许创建查询来搜索数据, 而无需了解 Elasticsearch 查询的语法或基础知识. 它们允许用户通过简单地链接和组合 Criteria 对象来构建查询, 这些对象指定了搜索文档必须满足的条件.

在谈论 AND 或 OR 时, 请牢记组合条件 在 Elasticsearch 中, AND 会被转换为 **must** 条件, OR 会被转换为 **should**

最好通过示例来说明 Criteria 的用法.(假设我们有一个具有 price 属性的 Book 实体):

Example 81. Get books with a given price

```
Criteria criteria = new Criteria("price").is(42.0);
Query query = new CriteriaQuery(criteria);
```

可以链接同一字段的条件, 这些条件将与逻辑 AND 相结合:

Example 82. Get books with a given price

```
Criteria criteria = new
Criteria("price").greaterThan(42.0).lessThan(34.0L);
Query query = new CriteriaQuery(criteria);
```

当链接 `Criteria` 时，默认情况下使用 `AND` 逻辑：

Example 83. Get all persons with first name James and last name Miller:

```
Criteria criteria = new Criteria("lastname").is("Miller") ①
  .and("firstname").is("James")                           ②
Query query = new CriteriaQuery(criteria);
```

① 第一个 `Criteria`

② `and()` 创建一个新的 `Criteria` 并且将其链接到第一个.

如果要创建嵌套查询，则需要为此使用子查询. 假设我们要查找 `last name` 为 *Miller* 且 `first name` 为 *Jack* 或 *John* 的所有人：

Example 84. Nested subqueries

```
Criteria miller = new Criteria("lastName").is("Miller")  ①
  .subCriteria(                                          ②
    new Criteria().or("firstName").is("John")            ③
      .or("firstName").is("Jack")                        ④
  );
Query query = new CriteriaQuery(criteria);
```

① 为 `last name` 创建第一个 `Criteria`

② 这与 `AND` 合并为一个子条件

③ This sub Criteria is an OR combination for the first name *John*

④ and the first name Jack

请参阅 Criteria 类的 API 文档以获取有关各种可用操作的完整概述.

7.6.2. StringQuery

此类将 Elasticsearch 查询作为 JSON 字符串. 以下代码显示了一个查询, 该查询搜索 first name 为 "Jack" 的人:

```
Query query = new SearchQuery("{ \"match\": { \"firstname\": { \"query\": \"Jack\" } } } ");
SearchHits<Person> searchHits = operations.search(query, Person.class);
```

如果您已经有要使用的 Elasticsearch 查询, 则可以使用 StringQuery.

7.6.3. NativeSearchQuery

NativeSearchQuery 是当您有复杂查询或无法使用 Criteria API 表示的查询(例如, 在构建查询和使用聚合时)时使用的类. 它允许使用来自 Elasticsearch 库的所有不同的 QueryBuilder 实现, 因此命名为 "native"..

下面的代码显示了如何搜索具有给定名字的人, 并且找到的文档具有 terms 聚合, 这些 terms 对这些人的 lastnames 的出现次数进行了计数:

```
Query query = new NativeSearchQueryBuilder()
    .addAggregation(terms("lastnames").field("lastname").size(10)) //
    .withQuery(QueryBuilders.matchQuery("firstname", firstName))
    .build();

SearchHits<Person> searchHits = operations.search(query, Person.class);
```

Chapter 8. Elasticsearch Repositories

本章包括 Elasticsearch 存储库实现的详细信息.

1. Book 实体

```
@Document(indexName="books")
class Book {
    @Id
    private String id;

    @Field(type = FieldType.text)
    private String name;

    @Field(type = FieldType.text)
    private String summary;

    @Field(type = FieldType.Integer)
    private Integer price;

    // getter/setter ...
}
```

8.1. 自动创建具有相应映射的索引

@Document 注解有一个 createIndex 参数. 如果此参数设置为 true (这是默认值), 则 Spring Data Elasticsearch 将在引导应用程序启动时引导存储库支持期间检查是否存在由 @Document 注解定义的索引.

如果不存在, 则将创建索引, 并且将从实体的注解派生的映射(请参见 Elasticsearch 对象映射)写入新创建的索引.

8.2. 查询方法

8.2.1. 查询方法查找策略

Elasticsearch 模块支持所有基本的查询构建特性
,如字符串查询、本地搜索查询、基于条件的查询或从方法名扩展的查询.

声明查询

从方法名扩展的查询还远远不够,而且还可能导致不可读的方法名.在这种情况下,可以使用 @Query 注解(请参阅使用 使用 @Query 注解).

8.2.2. 创建查询

通常,Elasticsearch 的查询创建机制按 查询方法 中所述运行. 这是 Elasticsearch 查询方法的转换简短示例:

Example 85. 从方法名创建查询

```
interface BookRepository extends Repository<Book, String> {
  List<Book> findByNameAndPrice(String name, Integer price);
}
```

上面的方法名称将被转换成以下 Elasticsearch json 查询

```
{
    "query": {
        "bool" : {
            "must" : [
                { "query_string" : { "query" : "?", "fields" : [ "name" ] } },
                { "query_string" : { "query" : "?", "fields" : [ "price" ] } }
            ]
        }
    }
}
```

Elasticsearch 支持的关键字列表如下.

Table 2. 方法名中支持的关键字

关键字	示例	**Elasticsearch Query String**
And	findByNameAndPrice	{ "query" : { "bool" : { "must" : [{ "query_string" : { "query" : "?", "fields" : ["name"] } }, { "query_string" : { "query" : "?", "fields" : ["price"] } }] } }}
Or	findByNameOrPrice	{ "query" : { "bool" : { "should" : [{ "query_string" : { "query" : "?", "fields" : ["name"] } }, { "query_string" : { "query" : "?", "fields" : ["price"] } }] } }}
Is	findByName	{ "query" : { "bool" : { "must" : [{ "query_string" : { "query" : "?", "fields" : ["name"] } }] } }}
Not	findByNameNot	{ "query" : { "bool" : { "must_not" : [{ "query_string" : { "query" : "?", "fields" : ["name"] } }] } }}
Between	findByPriceBetween	{ "query" : { "bool" : { "must" : [{"range" : {"price" : {"from" : ?, "to" : ?, "include_lower" : true, "include_upper" : true } } }] } }}
LessThan	findByPriceLessThan	{ "query" : { "bool" : { "must" : [{"range" : {"price" : {"from" : null, "to" : ?, "include_lower" : true, "include_upper" : false } } }] } }}
LessThanEqual	findByPriceLessThanEqual	{ "query" : { "bool" : { "must" : [{"range" : {"price" : {"from" : null, "to" : ?, "include_lower" : true, "include_upper" : true } } }] } }}

关键字	示例	Elasticsearch Query String
GreaterThan	findByPriceGreaterThan	{ "query" : { "bool" : { "must" : [{"range" : {"price" : {"from" : ?, "to" : null, "include_lower" : false, "include_upper" : true } } }] } }}
GreaterThanEqual	findByPriceGreaterThan	{ "query" : { "bool" : { "must" : [{"range" : {"price" : {"from" : ?, "to" : null, "include_lower" : true, "include_upper" : true } } }] } }}
Before	findByPriceBefore	{ "query" : { "bool" : { "must" : [{"range" : {"price" : {"from" : null, "to" : ?, "include_lower" : true, "include_upper" : true } } }] } }}
After	findByPriceAfter	{ "query" : { "bool" : { "must" : [{"range" : {"price" : {"from" : ?, "to" : null, "include_lower" : true, "include_upper" : true } } }] } }}
Like	findByNameLike	{ "query" : { "bool" : { "must" : [{ "query_string" : { "query" : "?*", "fields" : ["name"] }, "analyze_wildcard": true }] } }}
StartingWith	findByNameStartingWith	{ "query" : { "bool" : { "must" : [{ "query_string" : { "query" : "?*", "fields" : ["name"] }, "analyze_wildcard": true }] } }}
EndingWith	findByNameEndingWith	{ "query" : { "bool" : { "must" : [{ "query_string" : { "query" : "*?", "fields" : ["name"] }, "analyze_wildcard": true }] } }}

关键字	示例	Elasticsearch Query String
Contains/Containing	findByNameContaining	{ "query" : { "bool" : { "must" : [{ "query_string" : { "query" : "*?*", "fields" : ["name"] }, "analyze_wildcard": true }] } }}
In (when annotated as FieldType.Keyword)	findByNameIn(Collection<String>names)	{ "query" : { "bool" : { "must" : [{"bool" : {"must" : [{"terms" : {"name" : ["?","?"]}}] } }] } }}
In	findByNameIn(Collection<String>names)	{ "query": {"bool": {"must": [{"query_string":{"query": "\"?\" \"?\"", "fields": ["name"]}}]}}}
NotIn (when annotated as FieldType.Keyword)	findByNameNotIn(Collection<String>names)	{ "query" : { "bool" : { "must" : [{"bool" : {"must_not" : [{"terms" : {"name" : ["?","?"]}}] } }] } }}
NotIn	findByNameNotIn(Collection<String>names)	{"query": {"bool": {"must": [{"query_string": {"query": "NOT(\"?\" \"?\")", "fields": ["name"]}}]}}}
Near	findByStoreNear	Not Supported Yet !
True	findByAvailableTrue	{ "query" : { "bool" : { "must" : [{ "query_string" : { "query" : "true", "fields" : ["available"] } }] } }}
False	findByAvailableFalse	{ "query" : { "bool" : { "must" : [{ "query_string" : { "query" : "false", "fields" : ["available"] } }] } }}

关键字	示例	Elasticsearch Query String
OrderBy	findByAvailableTrueOrderByNameDesc	{ "query" : { "bool" : { "must" : [{ "query_string" : { "query" : "true", "fields" : ["available"] } }] } }, "sort":[{"name":{"order":"desc"}}] }

不支持使用 GeoJson 参数构建 Geo-shape 查询的方法名称.
如果需要在存储库中具有这样的功能，请在自定义存储库实现中将 ElasticsearchOperations 与 CriteriaQuery 一起使用.

8.2.3. 方法返回类型

Repository 可以定义为以下类型来返回多个元素:

- List<T>
- Stream<T>
- SearchHits<T>
- List<SearchHit<T>>
- Stream<SearchHit<T>>
- SearchPage<T>

8.2.4. 使用 @Query 注解

Example 86. 在方法上使用 `@Query` 注解声明查询.

```
interface BookRepository extends ElasticsearchRepository<Book, String> {
    @Query("{\"match\": {\"name\": {\"query\": \"?0\"}}}")
    Page<Book> findByName(String name,Pageable pageable);
}
```

注解参数 `String` 必须是一个有效的 Elasticsearch JSON 查询.它将会作为 `query` 元素的 `value` 发送到 Elasticsearch 中; 例如,如果使用参数 *John* 调用该函数,它将产生以下查询内容:

```
{
  "query": {
    "match": {
      "name": {
        "query": "John"
      }
    }
  }
}
```

8.3. Reactive Elasticsearch Repositories

响应式 Elasticsearch 存储库支持建立在核心存储库支持的基础上,在使用 使用 Spring Data Repositories 时解释了如何利用 Reactive Client 执行的 Reactive Elasticsearch 操作 提供的操作.

Spring Data Elasticsearch 响应式存储库支持使用 Project Reactor 作为其响应式组合库的选择.

主要使用 3 个接口:

- ReactiveRepository
- ReactiveCrudRepository
- ReactiveSortingRepository

8.3.1. 使用

要使用 Repository 访问存储在 Elasticsearch 中的 domain 对象,只需为其创建一个接口.另外,您将需要一个实体.

Example 87. Person 实体

```
public class Person {

  @Id
  private String id;
  private String firstname;
  private String lastname;
  private Address address;

  // … getters and setters omitted
}
```

请注意 id 属性的类型必须是 String.

Example 88. 创建 *repository interface* 来持久化 *Person* 实体

```
interface ReactivePersonRepository extends
ReactiveSortingRepository<Person, String> {

  Flux<Person> findByFirstname(String firstname);
①

  Flux<Person> findByFirstname(Publisher<String> firstname);
②

  Flux<Person> findByFirstnameOrderByLastname(String firstname);
③

  Flux<Person> findByFirstname(String firstname, Sort sort);
④

  Flux<Person> findByFirstname(String firstname, Pageable page);
⑤

  Mono<Person> findByFirstnameAndLastname(String firstname, String
lastname);       ⑥

  Mono<Person> findFirstByLastname(String lastname);
⑦

  @Query("{ \"bool\" : { \"must\" : { \"term\" : { \"lastname\" : \"?0\" }
} } }")
  Flux<Person> findByLastname(String lastname);
⑧

  Mono<Long> countByFirstname(String firstname)
⑨

  Mono<Boolean> existsByFirstname(String firstname)
⑩

  Mono<Long> deleteByFirstname(String firstname)
⑪
}
```

① 根据 `lastname` 查询匹配的所有人.

② 等待 `Publisher` 输入,为 `firstname` 绑定参数值.

③ 按 `lastname` 进行排序.

④ 按照 Sort 定义的排序规则进行排序.

⑤ 使用 Pageable 进行分页.

⑥ 使用 And / Or 关键字进行条件查询.

⑦ 查找第一个匹配的实体.

⑧ 根据 lastname 查询匹配的所有人, 方法使用 @Query 注解参数进行查询.

⑨ 计算所有匹配 firstname 的实体.

⑩ 检查是否至少存在一个与 firstname 匹配的实体.

⑪ 删除所有与 firstname 匹配的实体.

8.3.2. 配置

对于 Java 配置,请使用 @EnableReactiveElasticsearchRepositories 注解. 如果未配置 base package,则 SpringBoor 将扫描带 configuration 注解的类所在的包.

下面展示了如何使用 Java 配置:

Example 89. Java configuration for repositories

```
@Configuration
@EnableReactiveElasticsearchRepositories
public class Config extends AbstractReactiveElasticsearchConfiguration {

  @Override
  public ReactiveElasticsearchClient reactiveElasticsearchClient() {
    return ReactiveRestClients.create(ClientConfiguration.localhost());
  }
}
```

由于上一个示例中的存储库扩展了 ReactiveSortingRepository,因此所有 CRUD 操作以及对实体的排序方法都可用. 使用存储库实例是将其注入客户端的依赖,如以下示例所示:

Example 90. Sorted access to Person entities

```
public class PersonRepositoryTests {

  @Autowired ReactivePersonRepository repository;

  @Test
  public void sortsElementsCorrectly() {

    Flux<Person> persons = repository.findAll(Sort.by(new Order(ASC,
"lastname")));

    // ...
  }
}
```

8.4. repository 方法注解

8.4.1. @Highlight

`@Highlight` 注解定义了返回实体中那些字段需要高亮显示. 假如要在 `Book` 中将一些文本的 `name` 和 `summary` 高亮显示,可以使用一下存储库方法.

```
interface BookRepository extends Repository<Book, String> {

    @Highlight(fields = {
        @HighlightField(name = "name"),
        @HighlightField(name = "summary")
    })
    List<SearchHit<Book>> findByNameOrSummary(String text, String
summary);
}
```

可以定义多个像上面那样高亮显示的字段,并且 `@Highlight` 和 `@HighlightField` 注解都可以通过 `@HighlightParameters` 注解进一步定制.查看 `Javadocs` 以获得可能的配置选项.

在搜索结果中,可以从 SearchHit 类中检索高亮显示的数据.

8.5. 基于注解的配置

Spring Data Elasticsearch repositories 可以通过 JavaConfig 使用注解来启用.

Example 91. Spring Data Elasticsearch repositories using JavaConfig

```
@Configuration
@EnableElasticsearchRepositories(                              ①
  basePackages = "org.springframework.data.elasticsearch.repositories"
  )
static class Config {

  @Bean
  public ElasticsearchOperations elasticsearchTemplate() {     ②
      // ...
  }
}

class ProductService {

  private ProductRepository repository;                        ③

  public ProductService(ProductRepository repository) {
    this.repository = repository;
  }

  public Page<Product> findAvailableBookByName(String name, Pageable
pageable) {
    return repository.findByAvailableTrueAndNameStartingWith(name,
pageable);
  }
}
```

① EnableElasticsearchRepositories 启用 Repository 支持. 如果没有配置 base package, 它将使用放在上面的配置类之一.

② 使用 Elasticsearch 操作 一章中显示的配置之一 提供一个名为 elasticsearchTemplate 的 ElasticsearchOperations bean

③ 让 Spring 将 Repository bean 注入到您的类中.

8.6. Elasticsearch Repositories 使用 CDI

还可以使用 CDI 功能设置 Spring Data Elasticsearch 存储库.

Example 92. Spring Data Elasticsearch repositories using CDI

```
class ElasticsearchTemplateProducer {

  @Produces
  @ApplicationScoped
  public ElasticsearchOperations createElasticsearchTemplate() {
    // ...                                    ①
  }
}

class ProductService {

  private ProductRepository repository;  ②
  public Page<Product> findAvailableBookByName(String name, Pageable
pageable) {
    return repository.findByAvailableTrueAndNameStartingWith(name,
pageable);
  }
  @Inject
  public void setRepository(ProductRepository repository) {
    this.repository = repository;
  }
}
```

① 通过使用与 Elasticsearch 操作 一章中相同的方式来创建组件.

② 让 CDI 框架将存储库注入到您的类中.

8.7. Spring 命名空间

Spring Data Elasticsearch 模块包含一个自定义命名空间,允许定义存储库 bean 以及用于实例化 ElasticsearchServer 的元素.

如 创建存储库实例 中所述,使用 repositories 元素查找 Spring Data 存储库.

Example 93. Setting up Elasticsearch repositories using Namespace

```
<?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:elasticsearch="http://www.springframework.org/schema/data/elasticsea
rch"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       https://www.springframework.org/schema/beans/spring-beans-3.1.xsd
       http://www.springframework.org/schema/data/elasticsearch
       https://www.springframework.org/schema/data/elasticsearch/spring-
elasticsearch-1.0.xsd">

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

</beans>
```

使用 `Transport Client` 或 `Rest Client` 元素在上下文注册 `Elasticsearch Server` 实例.

Example 94. Transport Client using Namespace

```
<?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:elasticsearch="http://www.springframework.org/schema/data/elasticsea
rch"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       https://www.springframework.org/schema/beans/spring-beans-3.1.xsd
       http://www.springframework.org/schema/data/elasticsearch
       https://www.springframework.org/schema/data/elasticsearch/spring-
elasticsearch-1.0.xsd">

  <elasticsearch:transport-client id="client" cluster-
nodes="localhost:9300,someip:9300" />

</beans>
```

Example 95. Rest Client using Namespace

```
<?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:elasticsearch="http://www.springframework.org/schema/data/elasticsea
rch"

xsi:schemaLocation="http://www.springframework.org/schema/data/elasticsear
ch
       https://www.springframework.org/schema/data/elasticsearch/spring-
elasticsearch.xsd
       http://www.springframework.org/schema/beans
       https://www.springframework.org/schema/beans/spring-beans.xsd">

  <elasticsearch:rest-client id="restClient"
hosts="http://localhost:9200">

</beans>
```

Chapter 9. 审计

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.

此外, 必须通过注解配置或 XML 配置启用审核, 以注册所需的基础结构组件.
请参阅特定的存储库部分来获取帮助.

仅跟踪创建和修改日期的应用程序不需要指定 AuditorAware.

9.1.1. 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 96. 被审计实体

```
class Customer {

  @CreatedBy
  private User user;

  @CreatedDate
  private Instant 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.

审计的元数据并不一定要存在于根级实体中, 而是可以添加内嵌的元数据

(取决于所使用的实际存储)， 如下面的片段所示.

Example 97. Audit metadata in embedded entity

```
class Customer {

  private AuditMetadata auditingMetadata;

  // … further properties omitted
}

class AuditMetadata {

  @CreatedBy
  private User user;

  @CreatedDate
  private Instant createdDate;

}
```

9.1.2. 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.

9.1.3. 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 98. 基于 *Spring Security* 的 AuditorAware 的实现

```
class SpringSecurityAuditorAware implements AuditorAware<User> {

  @Override
  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 domain 用户,但是根据找到的 Authentication ,您还可以从任何地方查找它.

9.1.4. ReactiveAuditorAware

当使用响应式时, 您可能想利用上下文信息来提供 @CreatedBy 或 @LastModifiedBy 信息. 我们提供了一个 ReactiveAuditorAware<T> SPI 接口, 您必须实现该接口通知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响应式 Spring Security 的 Authentication 对象的接口的实现:

Example 99. 基于 *Spring Security* 的 ReactiveAuditorAware 实现

```
class SpringSecurityAuditorAware implements ReactiveAuditorAware<User> {

  @Override
  public Mono<User> getCurrentAuditor() {

    return ReactiveSecurityContextHolder.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 domain 用户,但是根据找到的 Authentication ,您还可以从任何地方查找它.

9.2. Elasticsearch 审计

9.2.1. 准备实体

为了让审计代码能够判断一个实体实例是否是最新的,这个实体必须实现 Persistable<ID> 接口,定义如下:

```
package org.springframework.data.domain;

import org.springframework.lang.Nullable;

public interface Persistable<ID> {
    @Nullable
    ID getId();

    boolean isNew();
}
```

由于 Id 的存在不能够确定在 Elasticsearch 中实体是否是最新实体

,所以我们还需一些附加信息. 一种方法是使用与创建审计相关的字段进行确定:

一个 Person 实体可能如下所示 - 为简洁起见,省略了 getter 和 setter 方法:

```
@Document(indexName = "person")
public class Person implements Persistable<Long> {
    @Id private Long id;
    private String lastName;
    private String firstName;
    @Field(type = FieldType.Date, format = DateFormat.basic_date_time)
    private Instant createdDate;
    private String createdBy
    @Field(type = FieldType.Date, format = DateFormat.basic_date_time)
    private Instant lastModifiedDate;
    private String lastModifiedBy;

    public Long getId() {                                                    ①
        return id;
    }

    @Override
    public boolean isNew() {
        return id == null || (createdDate == null && createdBy == null);    ②
    }
}
```

① 这个 getter 方法是此接口必须实现的

② 如果对象没有 id,或者没有设置包含创建属性的字段,那么该对象就是新的.

9.2.2. 启用审计

在实体设置好并提供 AuditorAware - 或 ReactiveAuditorAware 之后
,审计必须通过在配置类上设置 @EnableElasticsearchAuditing 注解来启用:

```
@Configuration
@EnableElasticsearchRepositories
@EnableElasticsearchAuditing
class MyConfiguration {
   // configuration code
}
```

当使用 reactive 技术栈时必须按如下操作:

```
@Configuration
@EnableReactiveElasticsearchRepositories
@EnableReactiveElasticsearchAuditing
class MyConfiguration {
   // configuration code
}
```

如果您的代码包含多个针对不同类型的 `AuditorAware` bean,则必须提供该 bean 的名称,以用作 `@EnableElasticsearchAuditing` 注解的 `auditAwareRef` 参数的参数.

Chapter 10. Entity Callbacks (实体回调)

Spring Data 基础结构提供了用于在调用某些方法之前和之后修改实体的钩子.

这些所谓的 EntityCallback 实例提供了一种方便的方式来检查并可能以回调方式修改实体.

EntityCallback 看起来很像 ApplicationListener. 一些 Spring Data 模块发布特定的存储库事件 (例如 BeforeSaveEvent) ,这些事件允许修改给定实体. 在某些情况下,例如使用不可变类型时,这些事件可能会引起麻烦. 同样,事件发布依赖于 ApplicationEventMulticaster. 如果使用异步 TaskExecutor 进行配置 ,则可能导致不可预测的结果,因为事件处理可以在不同的线程中执行.

实体回调为同步和响应式 API 提供集成点,以确保在处理链中定义明确的检查点处按顺序执行 ,并返回可能修改的实体或响应式包装器类型.

实体回调通常按 API 类型分开. 这种分离意味着同步 API 仅考虑同步实体回调 ,而响应式实现仅考虑响应式实体回调.

实体回调 API 在 Spring Data Commons 2.2 引入. 这是应用实体修改的推荐方法. 在调用可能已注册的 EntityCallback 实例之前,仍将发布现有的特定于存储库的 ApplicationEvents.

10.1. 实现 Entity Callbacks

EntityCallback 通过其泛型类型参数直接与 domain 类型相关联. 每个 Spring Data 模块通常附带一组涵盖实体生命周期的预定义 EntityCallback 接口.

Example 100. Anatomy of an EntityCallback

```
@FunctionalInterface
public interface BeforeSaveCallback<T> extends EntityCallback<T> { ①

    /**
     * Entity callback method invoked before a domain object is saved.
     * Can return either the same or a modified instance.
     *
     * @return the domain object to be persisted.
     */
    T onBeforeSave(T entity , ②
                String collection ); ③
}
```

① 保存实体之前要调用的 `BeforeSaveCallback` 方法. 返回一个可能被修改的实例.

② 实体被持久化之前.

③ 许多 `store` 特定的参数,例如持久化的实体集合.

Example 101. Anatomy of a reactive `EntityCallback`

```
@FunctionalInterface
public interface ReactiveBeforeSaveCallback<T> extends EntityCallback<T> {
①

    /**
     * Entity callback method invoked on subscription, before a domain
object is saved.
     * The returned Publisher can emit either the same or a modified
instance.
     *
     * @return Publisher emitting the domain object to be persisted.
     */
    Publisher<T> onBeforeSave(T entity ,  ②
                              String collection ); ③
}
```

① 保存实体之前,要在订阅上调用的 `BeforeSaveCallback` 特定方法.
发出可能已修改的实例.

② 实体在持久化之前.

③ 许多 `store` 特定的参数,例如持久化的实体集合.

可选的实体回调参数由实现中的 `Spring Data` 模块定义,并从 `EntityCallback.callback()` 的调用站点中推断出来.

实现适合您的应用程序需求的接口,如下例所示:

Example 102. BeforeSaveCallback 示例

```
class DefaultingEntityCallback implements BeforeSaveCallback<Person>,
Ordered {      ②

    @Override
    public Object onBeforeSave(Person entity, String collection) {
①

        if(collection == "user") {
            return // ...
        }

        return // ...
    }

    @Override
    public int getOrder() {
        return 100;
②
    }
}
```

① 根据您的要求实现回调.

② 如果存在相同 domain 类型的多个实体回调,则可能对实体回调进行排序. 排序遵循最低优先级.

10.2. 注册 Entity Callbacks

如果实体的实现在 ApplicationContext 中注册,则由实体的实现获取 EntityCallback Bean. 大多数 template API 已经实现了 ApplicationContextAware,因此可以访问 ApplicationContext

以下示例说明了有效的实体回调注册的集合:

Example 103. EntityCallback Bean 注册示例

```
@Order(1)                                                                  ①
@Component
class First implements BeforeSaveCallback<Person> {

    @Override
    public Person onBeforeSave(Person person) {
        return // ...
    }
}

@Component
class DefaultingEntityCallback implements BeforeSaveCallback<Person>,
                                                                Ordered { ②

    @Override
    public Object onBeforeSave(Person entity, String collection) {
        // ...
    }

    @Override
    public int getOrder() {
        return 100;                                                        ②
    }
}

@Configuration
public class EntityCallbackConfiguration {

    @Bean
    BeforeSaveCallback<Person> unorderedLambdaReceiverCallback() {    ③
        return (BeforeSaveCallback<Person>) it -> // ...
    }
}

@Component
class UserCallbacks implements BeforeConvertCallback<User>,
                                        BeforeSaveCallback<User> {    ④

    @Override
    public Person onBeforeConvert(User user) {
        return // ...
    }

    @Override
```

```
    public Person onBeforeSave(User user) {
        return // ...
    }
}
```

① `BeforeSaveCallback` 可以从 `@Order` 注解进行排序.

② `BeforeSaveCallback` 可以实现 `Ordered` 接口排序.

③ `BeforeSaveCallback` 使用 `lambda` 表达式. 默认情况下无序,最后调用.

④ 将多个实体回调接口组合在一个实现类中.

10.3. Elasticsearch EntityCallbacks

Spring Data Elasticsearch 在内部将 `EntityCallback` API 用于审计支持,并对以下回调进行响应:

Table 3. Supported Entity Callbacks

Callback	Method	Description	Order
Reactive/BeforeConvertCallback	`onBeforeConvert(T entity, IndexCoordinates index)`	Invoked before a domain object is converted to `org.springframework.data.elasticsearch.core.document.Document`. Can return the `entity` or a modified entity which then will be converted.	`Ordered.LOWEST_PRECEDENCE`

Callback	Method	Description	Order
Reactive/AfterConvertCallback	`onAfterConvert(T entity, Document document, IndexCoordinates indexCoordinates)`	Invoked after a domain object is converted from `org.springframework.data.elasticsearch.core.document.Document` on reading result data from Elasticsearch.	`Ordered.LOWEST_PRECEDENCE`
Reactive/AuditingEntityCallback	`onBeforeConvert(Object entity, IndexCoordinates index)`	Marks an auditable entity *created* or *modified*	100
Reactive/AfterSaveCallback	`T onAfterSave(T entity, IndexCoordinates index)`	Invoked after a domain object is saved.	`Ordered.LOWEST_PRECEDENCE`

Chapter 11. 其他 Elasticsearch 操作支持

本章涵盖了对不能通过 repository 接口直接访问的 Elasticsearch 操作的额外支持.建议将这些操作添加为自定义实现,如 Spring Data 存储库的自定义实现 中所述.

11.1. Filter Builder

Filter Builder 提高查询速度.

```
private ElasticsearchOperations operations;

IndexCoordinates index = IndexCoordinates.of("sample-index");

SearchQuery searchQuery = new NativeSearchQueryBuilder()
  .withQuery(matchAllQuery())
  .withFilter(boolFilter().must(termFilter("id", documentId)))
  .build();

Page<SampleEntity> sampleEntities = operations.searchForPage(searchQuery,
SampleEntity.class, index);
```

11.2. 对的数据量大的结果集使用 Scroll

Elasticsearch 有一个 scroll API,用于以块为单位获取较大的结果集. Spring Data Elasticsearch 在内部使用此方法来提供 `<T> SearchHitsIterator<T> SearchOperations.searchForStream(Query query, Class<T> clazz, IndexCoordinates index)` 方法的实现.

```
IndexCoordinates index = IndexCoordinates.of("sample-index");

SearchQuery searchQuery = new NativeSearchQueryBuilder()
  .withQuery(matchAllQuery())
  .withFields("message")
  .withPageable(PageRequest.of(0, 10))
  .build();

SearchHitsIterator<SampleEntity> stream =
elasticsearchTemplate.searchForStream(searchQuery, SampleEntity.class, index);

List<SampleEntity> sampleEntities = new ArrayList<>();
while (stream.hasNext()) {
  sampleEntities.add(stream.next());
}

stream.close();
```

SearchOperations API 中没有方法访问 scroll ID,如果必须访问 scroll ID,则可以使用 ElasticsearchRestTemplate 的以下方法:

```
@Autowired ElasticsearchRestTemplate template;

IndexCoordinates index = IndexCoordinates.of("sample-index");

SearchQuery searchQuery = new NativeSearchQueryBuilder()
  .withQuery(matchAllQuery())
  .withFields("message")
  .withPageable(PageRequest.of(0, 10))
  .build();

SearchScrollHits<SampleEntity> scroll = template.searchScrollStart(1000,
searchQuery, SampleEntity.class, index);

String scrollId = scroll.getScrollId();
List<SampleEntity> sampleEntities = new ArrayList<>();
while (scroll.hasSearchHits()) {
  sampleEntities.addAll(scroll.getSearchHits());
  scrollId = scroll.getScrollId();
  scroll = template.searchScrollContinue(scrollId, 1000, SampleEntity.class);
}
template.searchScrollClear(scrollId);
```

要将 Scroll API 与存储库方法一起使用,返回类型必须在 Elasticsearch 存储库中定义为

Stream. 然后,该方法的实现将使用 ElasticsearchTemplate 中的 scroll 方法.

```
interface SampleEntityRepository extends Repository<SampleEntity, String> {

    Stream<SampleEntity> findBy();

}
```

11.3. Sort 选项

除了描述的默认排序选项之外,Paging 和 Sorting Spring Data Elasticsearch 还具有 GeoDistanceOrder 类,该类可以将搜索结果按地理距离排序.

如果要搜索的类有 *location* 的 GeoPoint 属性,则下面的 Sort
将按到指定点的距离对结果进行排序:

```
Sort.by(new GeoDistanceOrder("location", new GeoPoint(48.137154, 11.5761247)))
```

11.4. Join-Type 实现

Spring Data Elasticsearch 支持 Join data type,
用于创建相应的索引映射并存储相关信息.

11.4.1. 设置数据

对于具有父子关系的实体中, 它必须具有 JoinField 属性, 并且对其注解. 假设有一个 Statement 实体, 其中的语句可能是 *question*, *answer*, *comment* 或 *vote*
(在此示例中也显示了 *Builder*, 但这不是必需的, 但稍后将在示例代码中使用):

```
@Document(indexName = "statements")
public class Statement {
    @Id
    private String id;

    @Field(type = FieldType.Text)
    private String text;

    @JoinTypeRelations(
        relations =
            {
                @JoinTypeRelation(parent = "question", children =
{"answer", "comment"}), ①
                @JoinTypeRelation(parent = "answer", children = "vote")
②
            }
    )
    private JoinField<String> relation;
③

    private Statement() {
    }

    public static StatementBuilder builder() {
        return new StatementBuilder();
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }

    public JoinField<String> getRelation() {
        return relation;
    }
```

```
    public void setRelation(JoinField<String> relation) {
        this.relation = relation;
    }

    public static final class StatementBuilder {
        private String id;
        private String text;
        private JoinField<String> relation;

        private StatementBuilder() {
        }

        public StatementBuilder withId(String id) {
            this.id = id;
            return this;
        }

        public StatementBuilder withText(String text) {
            this.text = text;
            return this;
        }

        public StatementBuilder withRelation(JoinField<String> relation) {
            this.relation = relation;
            return this;
        }

        public Statement build() {
            Statement statement = new Statement();
            statement.setId(id);
            statement.setText(text);
            statement.setRelation(relation);
            return statement;
        }
    }
}
```

① 一个 question 可以有 answers 和 comments

② 一个 answer 可以有 votes

③ JoinField 属性用于将相关联的名称 (*question*, *answer*, *comment* or *vote*) 和父 ID 组合在一起. 泛型类型必须与带注解的 @Id 属性相同.

Spring Data Elasticsearch 将为此类构建以下映射:

```
{
  "statements": {
    "mappings": {
      "properties": {
        "_class": {
          "type": "text",
          "fields": {
            "keyword": {
              "type": "keyword",
              "ignore_above": 256
            }
          }
        },
        "relation": {
          "type": "join",
          "eager_global_ordinals": true,
          "relations": {
            "question": [
              "answer",
              "comment"
            ],
            "answer": "vote"
          }
        },
        "text": {
          "type": "text"
        }
      }
    }
  }
}
```

11.4.2. 存储数据

给定该类的存储库，以下代码插入一个 question，两个 answers，一个 comment 和一个 vote：

```
void init() {
    repository.deleteAll();

    Statement savedWeather = repository.save(
        Statement.builder()
            .withText("How is the weather?")
            .withRelation(new JoinField<>("question"))
①
            .build());

    Statement sunnyAnswer = repository.save(
        Statement.builder()
            .withText("sunny")
            .withRelation(new JoinField<>("answer", savedWeather.getId()))
②
            .build());

    repository.save(
        Statement.builder()
            .withText("rainy")
            .withRelation(new JoinField<>("answer", savedWeather.getId()))
③
            .build());

    repository.save(
        Statement.builder()
            .withText("I don't like the rain")
            .withRelation(new JoinField<>("comment",
savedWeather.getId()))     ④
            .build());

    repository.save(
        Statement.builder()
            .withText("+1 for the sun")
            .withRelation(new JoinField<>("vote", sunnyAnswer.getId()))
⑤
            .build());
}
```

① create a question statement

② the first answer to the question

③ the second answer

④ a comment to the question

⑤ a vote for the first answer

11.4.3. 检索数据

当前必须使用本地搜索查询来查询数据，因此标准存储库方法不提供支持。可以使用Spring Data 存储库的自定义实现 代替。

下面的代码示例展示了如何使用 `ElasticsearchOperations` 实例检索所有具有 *vote*（必须为 *answers*，因为只有 answers 才可以 vote）的所有条目：

```
SearchHits<Statement> hasVotes() {
    NativeSearchQuery query = new NativeSearchQueryBuilder()
        .withQuery(hasChildQuery("vote", matchAllQuery(), ScoreMode.None))
        .build();

    return operations.search(query, Statement.class);
}
```

Appendix

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 4. 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. [3]

Table 5. 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.

[3] 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 6. 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 7. 查询断言关键字

逻辑关键字	关键字表达
AND	And

逻辑关键字	关键字表达
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)
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

逻辑关键字	关键字表达
`TRUE`	`True, IsTrue`
`WITHIN`	`Within, IsWithin`

除过滤断言外,还支持以下修饰符列表:

Table 8. 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 9. 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.

返回类型	描述
`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.
`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`.

返回类型	描述
`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: Migration Guides

Upgrading from 3.2.x to 4.0.x

This section describes breaking changes from version 3.2.x to 4.0.x and how removed features can be replaced by new introduced features.

Removal of the used Jackson Mapper

One of the changes in version 4.0.x is that Spring Data Elasticsearch does not use the Jackson Mapper anymore to map an entity to the JSON representation needed for Elasticsearch (see Elasticsearch 对象映射). In version 3.2.x the Jackson Mapper was the default that was used. It was possible to switch to the meta-model based converter (named ElasticsearchEntityMapper) by explicitly configuring it (Meta Model Object Mapping).

In version 4.0.x the meta-model based converter is the only one that is available and does not need to be configured explicitly. If you had a custom configuration to enable the meta-model converter by providing a bean like this:

```
@Bean
@Override
public EntityMapper entityMapper() {

  ElasticsearchEntityMapper entityMapper = new ElasticsearchEntityMapper(
    elasticsearchMappingContext(), new DefaultConversionService()
  );
  entityMapper.setConversions(elasticsearchCustomConversions());

  return entityMapper;
}
```

You now have to remove this bean, the ElasticsearchEntityMapper interface has been removed.

Entity configuration

Some users had custom Jackson annotations on the entity class, for example in order to define a custom name for the mapped document in Elasticsearch or to configure date conversions. These are not taken into account anymore. The needed functionality is now provided with Spring Data Elasticsearch's `@Field` annotation. Please see 映射注解概述 for detailed information.

Removal of implicit index name from query objects

In 3.2.x the different query classes like `IndexQuery` or `SearchQuery` had properties that were taking the index name or index names that they were operating upon. If these were not set, the passed in entity was inspected to retrieve the index name that was set in the `@Document` annotation.
In 4.0.x the index name(s) must now be provided in an additional parameter of type `IndexCoordinates`. By separating this, it now is possible to use one query object against different indices.

So for example the following code:

```
IndexQuery indexQuery = new IndexQueryBuilder()
  .withId(person.getId().toString())
  .withObject(person)
  .build();

String documentId = elasticsearchOperations.index(indexQuery);
```

must be changed to:

```
IndexCoordinates indexCoordinates =
elasticsearchOperations.getIndexCoordinatesFor(person.getClass());

IndexQuery indexQuery = new IndexQueryBuilder()
  .withId(person.getId().toString())
  .withObject(person)
  .build();

String documentId = elasticsearchOperations.index(indexQuery, indexCoordinates);
```

To make it easier to work with entities and use the index name that is contained in the entitie's `@Document` annotation, new methods have been added like `DocumentOperations.save(T entity)`;

The new Operations interfaces

In version 3.2 there was the `ElasticsearchOperations` interface that defined all the methods for the `ElasticsearchTemplate` class. In version 4 the functions have been split into different interfaces, aligning these interfaces with the Elasticsearch API:

- `DocumentOperations` are the functions related documents like saving, or deleting
- `SearchOperations` contains the functions to search in Elasticsearch
- `IndexOperations` define the functions to operate on indexes, like index creation or mappings creation.

`ElasticsearchOperations` now extends `DocumentOperations` and `SearchOperations` and has methods get access to an `IndexOperations` instance.

All the functions from the `ElasticsearchOperations` interface in version 3.2 that are now moved to the `IndexOperations` interface are still available, they are marked as deprecated and have default implementations that delegate to the new implementation:

```
/**
 * Create an index for given indexName.
 *
 * @param indexName the name of the index
 * @return {@literal true} if the index was created
 * @deprecated since 4.0, use {@link IndexOperations#create()}
 */
@Deprecated
default boolean createIndex(String indexName) {
    return indexOps(IndexCoordinates.of(indexName)).create();
}
```

Deprecations

Methods and classes

Many functions and classes have been deprecated. These functions still work, but the Javadocs show with what they should be replaced.

Example from ElasticsearchOperations

```
/*
 * Retrieves an object from an index.
 *
 * @param query the query defining the id of the object to get
 * @param clazz the type of the object to be returned
 * @return the found object
 * @deprecated since 4.0, use {@link #get(String, Class, IndexCoordinates)}
 */
@Deprecated
@Nullable
<T> T queryForObject(GetQuery query, Class<T> clazz);
```

Elasticsearch deprecations

Since version 7 the Elasticsearch `TransportClient` is deprecated, it will be removed with Elasticsearch version 8. Spring Data Elasticsearch deprecates the `ElasticsearchTemplate` class which uses the `TransportClient` in version 4.0.

Mapping types were removed from Elasticsearch 7, they still exist as

deprecated values in the Spring Data `@Document` annotation and the `IndexCoordinates` class but they are not used anymore internally.

Removals

- As already described, the `ElasticsearchEntityMapper` interface has been removed.
- The `SearchQuery` interface has been merged into it's base interface `Query`, so it's occurrences can just be replaced with `Query`.
- The method `org.springframework.data.elasticsearch.core.ElasticsearchOperations.query(SearchQuery query, ResultsExtractor<T> resultsExtractor);` and the `org.springframework.data.elasticsearch.core.ResultsExtractor` interface have been removed. These could be used to parse the result from Elasticsearch for cases in which the response mapping done with the Jackson based mapper was not enough. Since version 4.0, there are the new 搜索结果类型 to return the information from an Elasticsearch response, so there is no need to expose this low level functionality.
- The low level methods `startScroll`, `continueScroll` and `clearScroll` have been removed from the `ElasticsearchOperations` interface. For low level scroll API access, there now are `searchScrollStart`, `searchScrollContinue` and `searchScrollClear` methods on the `ElasticsearchRestTemplate` class.

从 4.0.x 升级到 4.1.x

本节介绍了从版本 4.0.x 到 4.1.x 的重大更改,
以及如何用新引入的功能替换已删除的功能.

弃用

定义 *id* 属性

通过使用 id 或 document 命名, 可以将实体的属性定义为 id 属性. 现在已弃用此行为,

并将产生警告. 请向我们提供 @Id 注解, 以将某个属性标记为 id 属性.

索引映射

在 ReactiveElasticsearchClient.Indices 接口中, 不赞成使用 updateMapping 方法, 而建议使用 putMapping 方法. 它们执行相同的操作, 但是 putMapping 与 Elasticsearch API 中的命名一致:

Alias handling

在 IndexOperations 接口中, 不赞成使用 addAlias(AliasQuery), removeAlias(AliasQuery) 和 queryForAlias() 方法. 请使用的新的 alias(AliasAction), getAliases(String…) 和 getAliasesForIndex(String…) 方法, 他们提供了更多功能和更简洁的 API.

Parent-ID

从版本 6 开始, 已从 Elasticsearch 中删除了父代 ID 的用法.
我们现在弃用相应的字段和方法.

删除

Type mappings

删除了 @Document 注解的 *type mappings* 参数和 IndexCoordinates 对象. 它们在 Spring Data Elasticsearch 4.0 中已弃用, 并且不再使用它们的值.

重大变化

ReactiveElasticsearchClient.Indices 方法的返回类型

到目前为止, 尚未使用 ReactiveElasticsearchClient.Indices 中的方法. 随着 ReactiveIndexOperations 的引入, 有必要更改一些返回类型:

- createIndex 将返回 Mono<Boolean> 而不是 Mono<Void> 来指示成功创建索引.
- updateMapping 现在返回 Mono<Boolean> 而不是 Mono<Void> 来表示成功的映射存储.

DocumentOperartions.bulkIndex 方法的返回类型

这些方法正在重现包含新索引记录的 ID 的 List<String>. 现在他们返回一个 List<IndexedObjectInformation>; 这些对象包含 id 和有关乐观锁的信息(seq_no 和primary_term) :leveloffset: -1 :leveloffset: -1