Spring Data MongoDB - 参考文档

Mark Pollack Thomas Risberg Oliver Gierke Costin Leau Jon Brisbin Thomas Darimont Christoph Strobl Mark Paluch 版本 1.10.6.RELEASE， 2017-07-27

目录

- 支持的查询返回类型

前言

Spring Data MongoDB 项目使用 MongoDB 文档样式数据存储将核心 Spring 概念应用于解决方案的开发。我们提供了一个“模板”作为存储和查询文档的高级抽象。您将注意到 Spring Framework 中与 JDBC 支持的相似之处。

本文档是 Spring Data - Document Support 的参考指南。它解释了文档模块的概念和语义以及各种商店命名空间的语法。

本节提供 Spring 和 Document 数据库的一些基本介绍。本文档的其余部分仅涉及 Spring Data MongoDB 功能，并假设用户熟悉 MongoDB 和 Spring 概念。

1.了解春天

Spring Data 使用 Spring 框架的核心功能，例如 IoC 容器，类型转换系统，表达式语言，JMX 集成和可移植 DAO 异常层次结构。虽然了解 Spring API 并不重要，但要理解它们背后的概念。至少，对于您选择使用的任何 IoC 容器，IoC 背后的想法应该是熟悉的。

MongoDB 支持的核心功能可以直接使用，无需调用 Spring Container 的 IoC 服务。这很像 JdbcTemplate 可以在没有 Spring 容器的任何其他服务的情况下使用 'standalone'。要利用 Spring Data MongoDB 的所有功能，例如存储库支持，您需要使用 Spring 配置库的某些部分。

要了解有关 Spring 的更多信息，可以参考详细解释 Spring Framework 的全面（有时是解除武装）文档。有很多关于此事的文章，博客文章和书籍 - 请查看 Spring 框架主页以获取更多信息。

2.了解 NoSQL 和文档数据库

NoSQL 商店风靡了存储世界。这是一个涉及众多解决方案，术语和模式的庞大领域（即使这个术语本身具有多重含义，也会使事情变得更糟）。虽然一些原则很常见，但用户在某种程度上熟悉 MongoDB 至关重要。熟悉这个解决方案的最好方法是阅读他们的文档并按照他们的示例 - 通常不需要花费 5-10 分钟来完成它们，如果你来自 RDMBS 的背景很多次这些练习可以大开眼界。

学习 MongoDB 的起点是 www.mongodb.org。以下是其他有用资源的列表：

- 该手册介绍了 MongoDB，并包含入门指南，参考文档和教程的链接。
- 在网上外壳提供了一个便捷的方式结合一个 MongoDB 实例交互与在线教程。

- MongoDB Java 语言中心
- 有几本书可供购买
- Karl Seguin 的在线书籍：The Little MongoDB Book

3.要求

Spring Data MongoDB 1.x 二进制文件需要 JDK 6.0 及以上版本，以及 Spring Framework 4.3.10.RELEASE 及以上版本。

在文档存储方面，MongoDB 至少为 2.6。

4.其他帮助资源

学习新框架并不总是直截了当。在本节中，我们尝试提供从 Spring Data MongoDB 模块开始的易于遵循的指南。但是，如果您遇到问题或者您只是在寻找建议，请随时使用以下链接之一：

4.1。支持

有一些支持选项：

4.1.1。社区论坛

Stackoverflow 上的 Spring 数据 Stackoverflow 是所有 Spring Data（不仅仅是 Document）用户共享信息和互相帮助的标记。请注意，仅发布时才需要注册。

4.1.2。专业支持

专业的，源头支持，保证响应时间，可从 Pivotal Sofware，Inc。，Spring Data and Spring 公司获得。

4.2。继发展之后

有关 Spring Data Mongo 源代码存储库，夜间构建和快照工件的信息，请参阅 Spring Data Mongo 主页。您可以通过 Stackoverflow 上的 Community 与开发人员交互，帮助 Spring Data 最好地满足 Spring 社区的需求。要关注开发人员活动，请在 Spring Data Mongo 主页上查找邮件列表信息。如果您遇到错误或想要建议改进，请在 Spring Data 问题跟踪器上创建一个票证。要及时了解 Spring eco 系统中的最新新闻和公告，请订阅 Spring 社区门户。最后，您可以在 Twitter 上关注 Spring 博客或项目团队（SpringData）。

5.新的和值得注意的

5.1。Spring Data MongoDB 1.10 中的新功能

- 兼容 MongoDB Server 3.4 和 MongoDB Java Driver 3.4。
- 新的注释@CountQuery，@DeleteQuery 和@ExistsQuery。
- 对 MongoDB 3.2 和 MongoDB 3.4 聚合运算符的扩展支持（请参阅支持的聚合操作）。
- 创建索引时支持部分过滤器表达式。
- 加载/转换时发布生命周期事件 DBRef。
- 为 Query By Example 添加了任意匹配模式。
- 支持$caseSensitive 和$diacriticSensitive 文本搜索。
- 支持带孔的 GeoJSON Polygon。
- 批量提取 DBRef 的性能改进。

5.2。Spring Data MongoDB 1.9 中的新功能

- 以下注解已启用打造自己，组成注释：@Document，@Id，@Field，@Indexed，@CompoundIndex，@GeoSpatialIndexed，@TextIndexed，@Query，@Meta。
- 支持存储库查询方法中的预测。
- 支持按示例查询。
- java.util.Currency 对象映射的开箱即用支持。
- 添加对 MongoDB 2.6 中引入的批量操作的支持。
- 升级到 Querydsl 4。
- 断言与 MongoDB 3.0 和 MongoDB Java Driver 3.2 的兼容性（参见：MongoDB 3.0 支持）。

5.3。Spring Data MongoDB 1.8 有什么新功能

- Criteria 提供创建支持$geoIntersects。
- 支持规划环境地政司表现在@Query。
- MongoMappingEvents 公开它们的发行集合名称。
- 改进了对支持<mongo:mongo-client credentials="… " />。
- 改进了索引创建失败错误消息。

5.4。Spring Data MongoDB 1.7 中有什么新功能

- 断言与 MongoDB 3.0 和 MongoDB Java Driver 3-beta3 的兼容性（参见：MongoDB 3.0 支持）。
- 支持 JSR-310 和 ThreeTen 后端口日期/时间类型。
- 允许 Stream 作为查询方法返回类型（请参阅：查询方法）。
- 在域类型和查询中添加了 GeoJSON 支持（请参阅：GeoJSON 支持）。
- QueryDslPredicateExcecutor 现在支持 findAll(OrderSpecifier<?>… orders)。
- 支持通过脚本操作调用 JavaScript 函数。
- 改善对 CONTAINS 属性等集合的关键字的支持。
- 支持$bit，$mul 和$position 运营商 Update。

6.依赖性

由于各个 Spring Data 模块的启动日期不同，因此大多数模块都带有不同的主要版本号和次要版本号。找到兼容版本的最简单方法是依靠我们附带的定义兼容版本的 Spring Data Release Train BOM。在 Maven 项目中，您将在<dependencyManagement />POM 的部分中声明此依赖项：

示例 1.使用 Spring Data 版本系列 BOM

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.data</groupId>
      <artifactId>spring-data-releasetrain</artifactId>
      <version>${release-train}</version>
      <scope>import</scope>
      <type>pom</type>
```

```
    </dependency>
  </dependencies>
</dependencyManagement>
```

目前的发布列车版本是 Ingalls-SR6。列车名称按字母顺序升序，此处列出了当前可用的列车名称。版本名称遵循以下模式：${name}-${release}其中 release 可以是以下之一：

- BUILD-SNAPSHOT - 当前快照
- M1，M2 等等-里程碑
- RC1，RC2 等等-候选发布版
- RELEASE - GA 发布
- SR1，SR2 等等-服务版本

可以在我们的 Spring Data 示例存储库中找到使用 BOM 的工作示例。如果这样就声明了你想要使用的 Spring 数据模块而没有<dependencies />块中的版本。

示例 2.声明对 Spring Data 模块的依赖关系

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

6.1。Spring Boot 的依赖管理

Spring Boot 已经为您选择了最新版本的 Spring Data 模块。如果您想要升级到更新版本，只需将属性配置为您想要使用 spring-data-releasetrain.version 的列车名称和迭代。

6.2。Spring 框架

当前版本的 Spring Data 模块需要 Spring Framework 4.3.10.RELEASE 或更高版本。这些模块也可以使用该次要版本的旧版本。但是，强烈建议使用该代中的最新版本。

7.使用 Spring Data Repositories

Spring Data 存储库抽象的目标是显着减少为各种持久性存储实现数据访问层所需的样板代码量。

Spring Data 存储库文档和您的模块

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 Persistence API（JPA）模块的配置和代码示例。调整 XML 名称空间声明和要扩展的类型以适应您正在使用的特定模块的等效项。命名空间参考涵盖支持存储库 API 的所有 Spring Data 模块支持的 XML 配置，存储库查询关键字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，请参阅本文档该模块的章节。

7.1。核心概念

Spring Data 存储库抽象的中心接口 Repository（可能不是那么令人惊讶）。它将域类和域类的 id 类型作为类型参数进行管理。此接口主要用作标记接口，用于捕获要使用的类型，并帮助您发现扩展此接口的接口。该 CrudRepository 规定对于正在管理的实体类复杂的 CRUD 功能。

示例 3. CrudRepository 接口

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

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

  T findOne(ID primaryKey);       (2)

  Iterable<T> findAll();          (3)

  Long count();                   (4)

  void delete(T entity);          (5)

  boolean exists(ID primaryKey);  (6)

  // … 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	返回实体数量。
五	删除给定的实体。
6	指示是否存在具有给定 id 的实体。

我们还提供持久性技术特定的抽象，例如 JpaRepository 或 MongoRepository。CrudRepository 除了相当通用的持久性技术无关的接口（如 CrudRepository）之外，这些接口还扩展并公开了底层持久性技术的功能。

除此之外，CrudRepository 还有一个 PagingAndSortingRepository 抽象，它添加了额外的方法来简化对实体的分页访问：

示例 4. PagingAndSortingRepository

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

  Iterable<T> findAll(Sort sort);

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

访问 User 页面大小为 20 的第二页，您可以执行以下操作：

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

除了查询方法之外，还可以使用计数和删除查询的查询派生。

示例 5.派生计数查询

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

  Long countByLastname(String lastname);
}
```

示例 6.派生的删除查询

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

  Long deleteByLastname(String lastname);

  List<User> removeByLastname(String lastname);

}
```

7.2。查询方法

标准 CRUD 功能存储库通常对基础数据存储区进行查询。使用 Spring Data，声明这些查询将分为四个步骤：

1. 声明扩展 Repository 或其子接口之一的接口，并将其键入到它将处理的域类和 ID 类型。

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

2. 在接口上声明查询方法。

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

5. 设置 Spring 以为这些接口创建代理实例。通过 JavaConfig：

6. import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
7.
8. @EnableJpaRepositories
   class Config {}

   或通过 XML 配置：

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

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

   </beans>
   ```

   在此示例中使用 JPA 名称空间。如果您使用的是存储库抽象任何其他商店，你需要更改为您储存模块，它应该被交换的适当的命名空间声明 jpa 赞成的，例如，mongodb。

   另请注意，JavaConfig 变体不会明确配置包，因为默认情况下使用带注释的类的包。要自定义要扫描的包，请使用 basePackage…数据存储特定存储库 @Enable…注释的属性之一。

9. 获取注入的存储库实例并使用它。

```
10. public class SomeClient {
11.
12.   @Autowired
13.   private PersonRepository repository;
14.
15.   public void doSomething() {
16.     List<Person> persons = repository.findByLastname("Matthews");
17.   }
    }
```

以下各节详细介绍了每个步骤。

7.3。定义存储库接口

作为第一步，您可以定义特定于域类的存储库接口。接口必须扩展 Repository 并键入域类和 ID 类型。如果要为该域类型公开 CRUD 方法，请扩展 CrudRepository 而不是 Repository。

7.3.1。微调存储库定义

通常，您的存储库接口将扩展 Repository，CrudRepository 或 PagingAndSortingRepository。或者，如果您不想扩展 Spring Data 接口，还可以使用注释来存储您的存储库接口@RepositoryDefinition。扩展 CrudRepository 公开了一整套方法来操纵您的实体。如果您希望对所公开的方法有选择性，只需将要公开的方法复制 CrudRepository 到域存储库中即可。

> This allows you to define your own abstractions on top of the provided Spring Data Repositories functionality.

Example 7. Selectively exposing CRUD methods

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

  T findOne(ID id);

  T save(T entity);
}

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

In this first step you defined a common base interface for all your domain repositories and exposed findOne(…) as well as save(…).These methods will be routed into the base repository implementation of the store of your choice provided by Spring Data ,e.g. in the case if JPA SimpleJpaRepository, because they are matching the method

signatures in CrudRepository. So the UserRepository will now be able to save users, and find single ones by id, as well as triggering a query to find Users by their email address.

> Note, that the intermediate repository interface is annotated with @NoRepositoryBean. Make sure you add that annotation to all repository interfaces that Spring Data should not create instances for at runtime.

7.3.2. Using Repositories with multiple Spring Data modules

Using a unique Spring Data module in your application makes things simple hence, all repository interfaces in the defined scope are bound to the Spring Data module. Sometimes applications require using more than one Spring Data module. In such case, it's required for a repository definition to distinguish between persistence technologies. Spring Data enters strict repository configuration mode because it detects multiple repository factories on the class path. Strict configuration requires details on the repository or the domain class to decide about Spring Data module binding for a repository definition:

1. 如果存储库定义扩展了特定于模块的存储库，那么它就是特定 Spring Data 模块的有效候选者。

2. 如果域类使用特定于模块的类型注释进行注释，那么它是特定 Spring Data 模块的有效候选者。Spring Data 模块接受第三方注释（例如 JPA @Entity）或提供自己的注释，例如@DocumentSpring Data MongoDB / Spring Data Elasticsearch。

示例 8.使用模块特定接口的存储库定义

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

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

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

MyRepository 并在其类型层次结构中 UserRepository 扩展 JpaRepository。它们是 Spring Data JPA 模块的有效候选者。

示例 9.使用通用接口的存储库定义

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

```
}

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

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

AmbiguousRepository 和 AmbiguousUserRepository 仅延伸 Repository，并 CrudRepository 在他们的类型层次。虽然使用独特的 Spring Data 模块完全没问题，但是多个模块无法区分这些存储库应该绑定到哪个特定的 Spring Data。

示例 10.使用带注释的域类的存储库定义

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

@Entity
public class Person {
  …
}

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

@Document
public class User {
  …
}
```

PersonRepositoryPerson 使用 JPA 注释注释的引用，@Entity 因此此存储库显然属于 Spring Data JPA。UserRepository 使用 User 带有 Spring Data MongoDB @Document 注释的注释。

示例 11.使用具有混合注释的域类的存储库定义

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

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

@Entity
```

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

此示例显示了使用 JPA 和 Spring Data MongoDB 注释的域类。它定义了两个存储库，JpaPersonRepository 和 MongoDBPersonRepository。一个用于 JPA，另一个用于 MongoDB 用法。Spring Data 不再能够分辨出导致未定义行为的存储库。

存储库类型详细信息和标识域类注释用于严格存储库配置，以识别特定 Spring 数据模块的存储库候选。在同一域类型上使用多个持久性技术特定的注释可以跨多个持久性技术重用域类型，但是 Spring Data 不再能够确定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基础包。基础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包中。默认情况下，注释驱动的配置使用配置类的包。基于 XML 的配置中的基础包是必需的。

示例 12.基础包的注释驱动配置

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

7.4。定义查询方法

存储库代理有两种方法可以从方法名称派生特定于商店的查询。它可以直接从方法名称派生查询，也可以使用手动定义的查询。可用选项取决于实际商店。但是，必须有一个策略来决定创建什么实际查询。我们来看看可用的选项。

7.4.1。查询查找策略

存储库基础结构可以使用以下策略来解析查询。您可以 query-lookup-strategy 在 XML 配置的情况下通过属性在命名空间配置策略，或者 queryLookupStrategy 在 Java 配置的情况下通过 Enable $ {store}存储库注释的属性配置策略。特定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商店的查询。一般方法是从方法名称中删除一组给定的已知前缀，并解析方法的其余部分。了解更多关于在查询构造创建查询。

- USE_DECLARED_QUERY 尝试查找声明的查询，并在发现异常时抛出异常。查询可以通过某处的注释来定义，也可以通过其他方式声明。查阅特定商店的文档以查找该商店 的可用选项。如果存储库基础结构在引导时未找到该方法的声明查询，则它将失败。

- CREATE_IF_NOT_FOUND（默认）组合 CREATE 和 USE_DECLARED_QUERY。它首先查找声明的查询，如果未找到声明的查询，则会创建基于自定义方法名称的查询。这是默认的查找策略，因此如果您没有明确配置任何内容，将使用它。它允许通过方法名称进行快速查询定义，还可以根据需要引入声明的查询来自定义这些查询。

7.4.2。查询创建

Spring Data 存储库基础结构中内置的查询构建器机制对于构建对存储库实体的约束查询非常有用。该机制条前缀 find…By，read…By，query…By，count…By，和 get…By 从所述方法和开始分析它的其余部分。introduction 子句可以包含其他表达式，例如 Distinct 在要创建的查询上设置不同的标志。但是，第一个 By 用作分隔符以指示实际标准的开始。在最基本的层面上，您可以在实体属性上定义条件，并将它们与 And 和它们连接起来 Or。

示例 13.从方法名称创建查询

```
public interface PersonRepository extends Repository<User, 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，有一些一般性的事情需要注意。

- 表达式通常是属性遍历与可以连接的运算符相结合。您可以将属性表达式与 AND 和组合使用 OR。您还可以得到这样的运营商为支撑 Between，LessThan，GreaterThan，Like 为属性表达式。支持的运算符可能因数据存储而异，因此请参阅参考文档的相应部分。

- 方法解析器支持 IgnoreCase 为各个属性（例如 findByLastnameIgnoreCase(…)）或支持忽略大小写的类型（String 例如，通常为实例）的所有属性设置标志 findByLastnameAndFirstnameAllIgnoreCase(…)。是否支持忽略大小写可能因商店而异，因此请参阅参考文档中有关特定于商店的查询方法的相关章节。

- 您可以通过 OrderBy 在引用属性的查询方法中附加子句并提供排序方向（Asc 或 Desc）来应用静态排序。要创建支持动态排序的查询方法，请参阅特殊参数处理。

7.4.3。属性表达式

属性表达式只能引用被管实体的直接属性，如前面的示例所示。在查询创建时，您已确保已解析的属性是托管域类的属性。但是，您也可以通过遍历嵌套属性来定义约束。假设 a Person 有 Address 一个 ZipCode。在这种情况下，方法名称为

List<Person> findByAddressZipCode(ZipCode zipCode);

创建属性遍历 x.address.zipCode。解析算法首先将整个 part（AddressZipCode）解释为属性，并检查域类中是否具有该名称的属性（未大写）。如果算法成功，则使用该属性。如果没有，算法将来自右侧的驼峰部分的源分成头部和尾部，并尝试找到相应的属性，在我们的示例中，AddressZip 和 Code。如果算法找到具有该头部的属性，则它采用尾部并继续从那里构建树，以刚刚描述的方式将尾部分开。如果第一个分割不匹配，算法会将分割点移动到左侧（Address，ZipCode）并继续。

虽然这应该适用于大多数情况，但算法可能会选择错误的属性。假设 Person 该类也具有 addressZip 属性。该算法将在第一轮拆分中匹配，并且基本上选择了错误的属性并最终失败（因为类型 addressZip 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。所以我们的方法名称最终会像这样：

List<Person> findByAddress_ZipCode(ZipCode zipCode);

由于我们将下划线视为保留字符，因此我们强烈建议遵循标准 Java 命名约定（即**不在**属性名称中使用下划线而是使用 camel case）。

7.4.4。特殊参数处理

要处理查询中的参数，只需定义方法参数，如上面的示例所示。除此之外，基础结构将识别某些特定类型 Pageable，Sort 并动态地对您的查询应用分页和排序。

示例 14.在查询方法中使用 Pageable，Slice 和 Sort
Page<User> findByLastname(String lastname, Pageable pageable);

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

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

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

The first method allows you to pass an org.springframework.data.domain.Pageable instance to the query method to dynamically add paging to your statically defined query. A Page knows about the total number of elements and pages available. It does so by the infrastructure triggering a count query to calculate the overall number. As this might be expensive depending on the store used, Slice can be used as return instead. A Slice only knows about whether there's a next Slice available which might be just sufficient when walking through a larger result set.

Sorting options are handled through the Pageable instance too. If you only need sorting, simply add an org.springframework.data.domain.Sort parameter to your method. As you also can see, simply returning a List is possible as well. In this case the additional metadata required to build the actual Page instance will not be created (which in turn means that the additional count query that would have been necessary not being issued) but rather simply restricts the query to look up only the given range of entities.

> To find out how many pages you get for a query entirely you have to trigger an additional count query. By default this query will be derived from the query you actually trigger.

7.4.5. Limiting query results

查询方法的结果可以通过关键字来限制，first 或者 top 可以互换使用。可选的数值可以附加到 top / first 以指定要返回的最大结果大小。如果省略该数字，则假定结果大小为 1。

示例 15.使用 Top 和限制查询的结果大小 First
User findFirstByOrderByLastnameAsc();

User findTopByOrderByAgeDesc();

Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);

Slice<User> findTop3ByLastname(String lastname, Pageable pageable);

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

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

限制表达式也支持 Distinct 关键字。此外，对于将结果集限制为一个实例的查询，Optional 支持将结果包装为 an 。

如果将分页或切片应用于限制查询分页（以及可用页数的计算），则将其应用于有限结果中。

请注意，通过 Sort 参数将结果与动态排序相结合，可以表示“K”最小元素以及“K”元素的查询方法。

7.4.6。流式查询结果

可以使用 Java 8 Stream<T>作为返回类型以递增方式处理查询方法的结果。不是简单地将查询结果包装在 Stream 数据存储中，而是使用特定方法来执行流式传输。

示例 16.使用 Java 8 流式传输查询结果 Stream<T>

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

Stream<User> readAllByFirstnameNotNull();

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

甲 Stream 潜在封装底层数据存储特定资源和使用后必须因此被关闭。您可以手动关闭 Stream 使用 close()方法，也可以使用 Java 7 try-with-resources 块。

示例 17. Stream<T>在 try-with-resources 块中使用结果

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

并非所有 Spring Data 模块当前都支持 Stream<T>返回类型。

7.4.7。异步查询结果

可以使用 Spring 的异步方法执行功能异步执行存储库查询。这意味着该方法将在调用时立即返回，并且实际查询将在已提交给 Spring TaskExecutor 的任务中执行。

```
@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 的返回类型。
2	使用 Java 8 java.util.concurrent.CompletableFuture 作为返回类型。
3	使用 org.springframework.util.concurrent.ListenableFuture 返回类型。

7.5。创建存储库实例

在本节中，您将为定义的存储库接口创建实例和 bean 定义。一种方法是使用随每个支持存储库机制的 Spring Data 模块一起提供的 Spring 命名空间，尽管我们通常建议使用 Java-Config 样式配置。

7.5.1。XML 配置

每个 Spring Data 模块都包含一个存储库元素，允许您简单地定义 Spring 为您扫描的基础包。

示例 18.通过 XML 启用 Spring Data 存储库

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

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

</beans:beans>
```

在前面的示例中，指示 Spring 扫描 com.acme.repositories 其所有子包以进行扩展 Repository 或其子接口之一。对于找到的每个接口，基础结构都会注册特定 FactoryBean 于持久性技术的内容，以创建处理查询方法调用的相应代理。每个 bean 都是在从接口名称派生的 bean 名称下注册的，因此 UserRepository 将在其下注册一个接口 userRepository。该 base-package 属性允许使用通配符，以便您可以定义扫描包的模式。

使用过滤器

默认情况下，基础结构会选择扩展 Repository 位于已配置的基础包下的特定于持久性技术的子接口的每个接口，并为其创建一个 bean 实例。但是，您可能希望对创建的接口 bean 实例进行更细粒度的控制。要做到这一点，你使用<include-filter />和<exclude-filter />内部元素<repositories />。语义完全等同于 Spring 的上下文命名空间中的元素。有关详细信息，请参阅有关这些元素的 Spring 参考文档

例如，要将某些接口从实例化中排除为存储库，可以使用以下配置：

示例 19.使用 exclude-filter 元素
```
<repositories base-package="com.acme.repositories">
  <context:exclude-filter type="regex" expression=".*SomeRepository" />
</repositories>
```

此示例排除了 SomeRepository 从实例化结束的所有接口。

7.5.2。JavaConfig

The repository infrastructure can also be triggered using a store-specific @Enable${store}Repositories annotation on a JavaConfig class. For an introduction into Java-based configuration of the Spring container, see the reference documentation.[1]

A sample configuration to enable Spring Data repositories looks something like this.

Example 20. Sample annotation based repository configuration
```
@Configuration
@EnableJpaRepositories("com.acme.repositories")
class ApplicationConfiguration {

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

> The sample uses the JPA-specific annotation, which you would change according to the store module you actually use. The same applies to the definition of the EntityManagerFactory bean. Consult the sections covering the store-specific configuration.

7.5.3. Standalone usage

You can also use the repository infrastructure outside of a Spring container, e.g. in CDI environments. You still need some Spring libraries in your classpath, but generally you can set up repositories programmatically as well. The Spring Data modules that provide repository support ship a persistence technology-specific RepositoryFactory that you can use as follows.

Example 21. Standalone usage of repository factory

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

7.6. Custom implementations for Spring Data repositories

Often it is necessary to provide a custom implementation for a few repository methods. Spring Data repositories easily allow you to provide custom repository code and integrate it with generic CRUD abstraction and query method functionality.

7.6.1. Adding custom behavior to single repositories

To enrich a repository with custom functionality you first define an interface and an implementation for the custom functionality. Use the repository interface you provided to extend the custom interface.

Example 22. Interface for custom repository functionality

```
interface UserRepositoryCustom {
  public void someCustomMethod(User user);
}
```

Example 23. Implementation of custom repository functionality

```
class UserRepositoryImpl implements UserRepositoryCustom {

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

> The most important bit for the class to be found is the Impl postfix of the name on it compared to the core repository interface (see below).

The implementation itself does not depend on Spring Data and can be a regular Spring bean. So you can use standard dependency injection behavior to inject references to other beans like a JdbcTemplate, take part in aspects, and so on.

Example 24. Changes to the your basic repository interface

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

  // Declare query methods here
}
```

Let your standard repository interface extend the custom one. Doing so combines the CRUD and custom functionality and makes it available to clients.

Configuration

If you use namespace configuration, the repository infrastructure tries to autodetect custom implementations by scanning for classes below the package we found a repository in. These classes need to follow the naming convention of appending the namespace element's attribute repository-impl-postfix to the found repository interface name. This postfix defaults to Impl.

Example 25. Configuration example
```
<repositories base-package="com.acme.repository" />

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

The first configuration example will try to look up a class com.acme.repository.UserRepositoryImpl to act as custom repository implementation, whereas the second example will try to lookup com.acme.repository.UserRepositoryFooBar.

Manual wiring

The approach just shown works well if your custom implementation uses annotation-based configuration and autowiring only, as it will be treated as any other Spring bean. If your custom implementation bean needs special wiring, you simply declare the bean and name it after the conventions just described. The infrastructure will then refer to the manually defined bean definition by name instead of creating one itself.

Example 26. Manual wiring of custom implementations
```
<repositories base-package="com.acme.repository" />

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

7.6.2. Adding custom behavior to all repositories

The preceding approach is not feasible when you want to add a single method to all your repository interfaces. To add custom behavior to all repositories, you first add an intermediate interface to declare the shared behavior.

Example 27. An interface declaring custom shared behavior
```
@NoRepositoryBean
public interface MyRepository<T, ID extends Serializable>
  extends PagingAndSortingRepository<T, ID> {

  void sharedCustomMethod(ID id);
}
```

现在，您的各个存储库接口将扩展此中间接口而不是 Repository 接口，以包含声明的功能。接下来，创建中间接口的实现，该实现扩展特定于持久性技术的存储库基类。然后，此类将充当存储库代理的自定义基类。

示例 28.自定义存储库基类

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

  private final EntityManager entityManager;

  public MyRepositoryImpl(JpaEntityInformation entityInformation,
                          EntityManager entityManager) {
    super(entityInformation, entityManager);

    // Keep the EntityManager around to used from the newly introduced methods.
    this.entityManager = entityManager;
  }

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

该类需要具有特定于商店的存储库工厂实现所使用的超类的构造函数。如果存储库基类具有多个构造函数，则覆盖采用 EntityInformation 加号存储特定基础结构对象（例如，EntityManager 模板类）的构造函数。

Spring <repositories />命名空间的默认行为是为所有接口提供实现 base-package。这意味着如果保持当前状态，MyRepositorySpring 将创建一个实现实例。这当然不是所希望的，因为它只是作为 Repository 您想要为每个实体定义的实际存储库接口之间的中介。要排除 Repository 从实例化为存储库实例扩展的接口，可以使用@NoRepositoryBean（如上所示）对其进行注释，也可以将其移出已配置的接口 base-package。

最后一步是使 Spring Data Infrastructure 了解自定义存储库基类。在 JavaConfig 中，这是通过使用注释的 repositoryBaseClass 属性来实现的@Enable…Repositories：

示例 29.使用 JavaConfig 配置自定义存储库基类

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

XML 命名空间中提供了相应的属性。

示例 30.使用 XML 配置自定义存储库基类

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

7.7。从聚合根发布事件

由存储库管理的实体是聚合根。在域驱动设计应用程序中，这些聚合根通常会发布域事件。Spring Data 提供了一个注释，@DomainEvents 您可以在聚合根的方法上使用该注释，以使该发布尽可能简单。

示例 31.从聚合根公开事件域事件

```
class AnAggregateRoot {

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

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

1	使用的方法@DomainEvents 可以返回单个事件实例或事件集合。它不能采取任何论点。
2	发布所有事件后，使用注释的方法@AfterDomainEventsPublication。它可以用于潜在地清理要发布的事件列表。

每次调用 Spring Data 存储库的 save(…)方法时，都会调用这些方法。

7.8。Spring 数据扩展

本节介绍了一组 Spring Data 扩展，它们可以在各种上下文中使用 Spring Data。目前大多数集成都针对 Spring MVC。

7.8.1。Querydsl 扩展

Querydsl 是一个框架，它可以通过其流畅的 API 构建静态类型的 SQL 类查询。

几个 Spring Data 模块提供与 Querydsl via 的集成 QueryDslPredicateExecutor。

例 32. QueryDslPredicateExecutor 接口

```
public interface QueryDslPredicateExecutor<T> {

   T findOne(Predicate predicate);           (1)

   Iterable<T> findAll(Predicate predicate);   (2)

   long count(Predicate predicate);          (3)
```

```
  boolean exists(Predicate predicate);        (4)

  // … more functionality omitted.
}
```

1	查找并返回与之匹配的单个实体 Predicate。
2	查找并返回与之匹配的所有实体 Predicate。
3	返回匹配的实体数 Predicate。
4	返回与 Predicateexists 匹配的实体。

To make use of Querydsl support simply extend QueryDslPredicateExecutor on your repository interface.

Example 33. Querydsl integration on repositories

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

}
```

The above enables to write typesafe queries using Querydsl Predicate s.

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

userRepository.findAll(predicate);
```

7.8.2. Web support

> This section contains the documentation for the Spring Data web support as it is implemented as of Spring Data Commons in the 1.6 range. As it the newly introduced support changes quite a lot of things we kept the documentation of the former behavior in Legacy web support.

如果模块支持存储库编程模型，则 Spring Data 模块附带各种 Web 支持。与 Web 相关的东西需要类路径上的 Spring MVC JAR，其中一些甚至提供与 Spring HATEOAS 的集成[2]。通常，通过@EnableSpringDataWebSupport 在 JavaConfig 配置类中使用注释来启用集成支持。

示例 34.启用 Spring Data Web 支持

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

该@EnableSpringDataWebSupport 批注注册几个组件，我们将在一个位讨论。它还将检测类路径上的 Spring HATEOAS，并为它注册集成组件（如果存在）。

另外，如果你正在使用 XML 配置，无论是注册 SpringDataWebSupport 还是 HateoasAwareSpringDataWebSupport 作为春豆：

示例 35.在 XML 中启用 Spring Data Web 支持

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

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

基本的 Web 支持

上面显示的配置设置将注册一些基本组件：

- A DomainClassConverter 使 Spring MVC 能够从请求参数或路径变量中解析存储库管理的域类的实例。

- HandlerMethodArgumentResolver 实现让 Spring MVC 从请求参数中解析 Pageable 和 Sort 实例。

DomainClassConverter

将 DomainClassConverter 让你在你的 Spring MVC 控制器方法签名直接使用域类型，这样就不必通过库手动查找实例：

示例 36.在方法签名中使用域类型的 Spring MVC 控制器

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

  @RequestMapping("/{id}")
  public 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，无需进一步查找。可以通过让 Spring MVC 首先将路径变量转换为域类的 id 类型来解析实例，并最终通过调用 findOne(…)为域类型注册的存储库实例来访问实例。

> 目前，存储库必须实现 CrudRepository 才有资格被发现进行转换。

HandlerMethodArgumentResolvers for Pageable 和 Sort

上面的配置代码段还注册了一个 PageableHandlerMethodArgumentResolver 以及一个实例 SortHandlerMethodArgumentResolver。注册启用 Pageable 并 Sort 成为有效的控制器方法参数

示例 37.使用 Pageable 作为控制器方法参数

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

  @Autowired UserRepository repository;

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

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

This method signature will cause Spring MVC try to derive a Pageable instance from the request parameters using the following default configuration:

Table 1. Request parameters evaluated for Pageable instances

page	Page you want to retrieve, 0 indexed and defaults to 0.
size	Size of the page you want to retrieve, defaults to 20.
sort	Properties that should be sorted by in the format property,property(,ASC|DESC). Default sort direction is ascending. Use multiple sort parameters if you want to switch directions,

	e.g. ?sort=firstname&sort=lastname,asc.

To customize this behavior extend either SpringDataWebConfiguration or the HATEOAS-enabled equivalent and override the pageableResolver() or sortResolver() methods and import your customized configuration file instead of using the @Enable-annotation.

In case you need multiple Pageable or Sort instances to be resolved from the request (for multiple tables, for example) you can use Spring's @Qualifier annotation to distinguish one from another. The request parameters then have to be prefixed with ${qualifier}_. So for a method signature like this:

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

you have to populate foo_page and bar_page etc.

The default Pageable handed into the method is equivalent to a new PageRequest(0, 20) but can be customized using the @PageableDefaults annotation on the Pageable parameter.

Hypermedia support for Pageables

Spring HATEOAS 附带了一个表示模型类 PagedResources，它允许 Page 使用必要的 Page 元数据丰富实例的内容，以及让客户端轻松浏览页面的链接。将 Page 转换为 a PagedResources 是通过 Spring HATEOAS ResourceAssembler 接口的实现来完成的 PagedResourcesAssembler。

例 38.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);
  }
}
```

如上所示启用配置允许将 PagedResourcesAssembler 其用作控制器方法参数。调用 toResources(…)它将导致以下情况：

- 的内容 Page 将成为内容 PagedResources 实例。
- 该 PagedResources 会得到一个 PageMetadata 附加填充信息形成的实例 Page 和基础 PageRequest。
- 在 PagedResources 得到 prev 和 next 根据页面的状态连接链路。链接将指向调用的方法映射到的 URI。添加到方法的分页参数将与设置相匹配，PageableHandlerMethodArgumentResolver 以确保稍后可以解析链接。

假设我们在数据库中有 30 个 Person 实例。您现在可以触发请求，您将看到与此类似的内容：GET http://localhost:8080/persons

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

您会看到汇编程序生成了正确的 URI，并且还选择了当前的默认配置以将参数解析 Pageable 为即将发生的请求。这意味着，如果更改该配置，链接将自动遵循更改。默认情况下，汇编程序指向它所调用的控制器方法，但可以通过交换自定义 Link 作为基础来定制，以构建 PagedResourcesAssembler.toResource(…)方法重载的分页链接。

Querydsl 网络支持

对于那些具有 QueryDSL 集成的商店，可以从 Request 查询字符串中包含的属性派生查询。

这意味着给定 User 前一个样本中的对象一个查询字符串

?firstname=Dave&lastname=Matthews

可以解决

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

使用 QuerydslPredicateArgumentResolver。

@EnableSpringDataWebSupport 在类路径中找到 Querydsl 时， 将自动启用该功能。

添加@QuerydslPredicate 到方法签名将提供可以通过使用 Predicate 执行的即用型 QueryDslPredicateExecutor。

通常从方法返回类型中解析类型信息。由于这些信息不一定与域类型匹配，因此使用该 root 属性可能是个好主意 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匹配 Predicatefor User。

默认绑定如下：

- Object 在简单的属性上 eq。
- Object 像集合一样的属性 contains。
- Collection 在简单的属性上 in。

这些绑定可以通过 Java 8 的 bindings 属性@QuerydslPredicate 或使用 Java 8 default methods 添加 QuerydslBinderCustomizer 到存储库接口来定制。

```
interface UserRepository extends CrudRepository<User, String>,
                                 QueryDslPredicateExecutor<User>,          (1)
                                 QuerydslBinderCustomizer<QUser> {         (2)
```

```
  @Override
  default public 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 提供对特定查找程序方法的访问权限 Predicate。
2	QuerydslBinderCustomizer 在存储库界面上定义的将自动拾取和快捷方式 @QuerydslPredicate(bindings=…)。
3	将 username 属性的绑定定义为简单包含绑定。
4	将 String 属性的默认绑定定义为不区分大小写包含匹配。
五	从解决方案中排除*密码*属性 Predicate。

7.8.3。存储库填充程序

如果您使用 Spring JDBC 模块，您可能熟悉 DataSource 使用 SQL 脚本填充的支持。虽然它不使用 SQL 作为数据定义语言，但它在存储库级别上可以使用类似的抽象，因为它必须与存储无关。因此，填充程序支持 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 data.json 包含以下内容的文件：

示例 39.在 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 元素轻松填充存储库。要将前面的数据填充到 PersonRepository，请执行以下操作：

示例 40.声明 Jackson 存储库 populator

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

  <repository:jackson2-populator locations="classpath:data.json" />

</beans>
```

此声明导致 data.json 文件通过 Jackson 读取和反序列化 ObjectMapper。

将通过检查_classJSON 文档的属性来确定将 JSON 对象解组到的类型。基础结构最终将选择适当的存储库来处理刚反序列化的对象。

要使用 XML 来定义存储库应该填充的数据，您可以使用该 unmarshaller-populator 元素。您将其配置为使用 Spring OXM 为您提供的 XML marshaller 选项之一。有关详细信息，请参阅 Spring 参考文档。

示例 41.声明一个解组存储库 populator（使用 JAXB）

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

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

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

</beans>
```

7.8.4。传统的网络支持

Spring MVC 的域类 Web 绑定

鉴于您正在开发 Spring MVC Web 应用程序，您通常必须从 URL 解析域类 ID。默认情况下，您的任务是将该请求参数或 URL 部分转换为域类，以将其传递给下面的层，或者直接在实体上执行业务逻辑。这看起来像这样：

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

  private final UserRepository userRepository;

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

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

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

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

首先，为每个控制器声明一个存储库依赖关系，以分别查找由控制器或存储库管理的实体。查找实体也是样板，因为它始终是一个 findOne(…)调用。幸运的是，Spring 提供了注册自定义组件的方法，这些组件允许将 String 值转换为任意类型。

属性编辑器

对于 3.0 之前的 Spring 版本 PropertyEditors，必须使用简单的 Java 。为了与之集成，Spring Data 提供了一个 DomainClassPropertyEditorRegistrar 查找注册的所有 Spring Data 存储库，ApplicationContext 并 PropertyEditor 为托管域类注册自定义。

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

```
  </property>
</bean>
```

如果您已按照前面的示例配置了 Spring MVC，则可以按如下方式配置控制器，这样可以减少大量的混乱和样板。

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

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

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

参考文档

8.简介

8.1。文件结构

这部分参考文档解释了 Spring Data MongoDB 提供的核心功能。

MongoDB 支持引入了 MongoDB 模块功能集。

MongoDB 存储库引入了 MongoDB 的存储库支持。

9. MongoDB 支持

MongoDB 支持包含各种功能，总结如下。

- Spring 配置支持使用基于 Java 的@Configuration 类或 Mongo 驱动程序实例和副本集的 XML 命名空间
- MongoTemplate 助手类，可提高执行常见 Mongo 操作的效率。包括文档和 POJO 之间的集成对象映射。
- 异常转换为 Spring 的可移植数据访问异常层次结构
- 功能丰富的对象映射与 Spring 的转换服务集成

- 基于注释的映射元数据，但可扩展以支持其他元数据格式
- 持久性和映射生命周期事件
- 基于 Java 的查询，标准和更新 DSL
- 自动实现 Repository 接口，包括支持自定义 finder 方法。
- QueryDSL 集成以支持类型安全查询。
- 跨存储持久性 - 使用 MongoDB 透明地持久保存/检索具有字段的 JPA 实体的支持
- Log4j 日志 appender
- 地理空间整合

对于您将发现自己使用的大多数任务 MongoTemplate 或利用丰富的映射功能的 Repository 支持。MongoTemplate 是寻找访问功能的地方，例如递增计数器或临时 CRUD 操作。MongoTemplate 还提供了回调方法，以便您可以轻松获取低级 API 工件，例如 com.mongo.DB 直接与 MongoDB 通信。各种 API 工件的命名约定的目标是复制基本 MongoDB Java 驱动程序中的那些，以便您可以轻松地将现有知识映射到 Spring API。

9.1。入门

Spring MongoDB 支持需要 MongoDB 2.6 或更高版本以及 Java SE 6 或更高版本。引导程序设置工作环境的简单方法是在 STS 中创建基于 Spring 的项目。

首先，您需要设置一个正在运行的 Mongodb 服务器。有关如何启动 MongoDB 实例的说明，请参阅 Mongodb 快速入门指南。一旦安装，启动 MongoDB 通常是执行以下命令的问题：MONGO_HOME/bin/mongod

要在 STS 中创建 Spring 项目，请转到文件→新建→弹簧模板项目→简单弹簧实用程序项目→在出现提示时按是。然后输入项目和包名称，例如 org.spring.mongodb.example。

然后将以下内容添加到 pom.xml 依赖项部分。

```
<dependencies>

  <!-- other dependency elements omitted -->

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

```
    <version>{version}</version>
  </dependency>

</dependencies>
```

还要更改 pom.xml 中的 Spring 版本

```
<spring.framework.version>{springVersion}</spring.framework.version>
```

您还需要将 maven 的 Spring Milestone 存储库的位置添加到 pom.xml 与 <dependencies/>元素相同的级别

```
<repositories>
  <repository>
    <id>spring-milestone</id>
    <name>Spring Maven MILESTONE Repository</name>
    <url>http://repo.spring.io/libs-milestone</url>
  </repository>
</repositories>
```

存储库也可以在这里浏览。

您可能还希望将日志记录级别设置 DEBUG 为查看一些其他信息，编辑 log4j.properties 要具有的文件

```
log4j.category.org.springframework.data.mongodb=DEBUG
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %40.40c:%
4L - %m%n
```

创建一个简单的 Person 类来保持：

```
package org.spring.mongodb.example;

public class Person {

  private String id;
  private String name;
  private int age;

  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public String getId() {
    return id;
  }
  public String getName() {
    return name;
```

```
  }
  public int getAge() {
    return age;
  }

  @Override
  public String toString() {
    return "Person [id=" + id + ", name=" + name + ", age=" + age + "]";
  }
}
```

并运行一个主要的应用程序

```
package org.spring.mongodb.example;

import static org.springframework.data.mongodb.core.query.Criteria.where;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.data.mongodb.core.MongoOperations;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.data.mongodb.core.query.Query;

import com.mongodb.Mongo;

public class MongoApp {

  private static final Log log = LogFactory.getLog(MongoApp.class);

  public static void main(String[] args) throws Exception {

    MongoOperations mongoOps = new MongoTemplate(new Mongo(), "database");
    mongoOps.insert(new Person("Joe", 34));

    log.info(mongoOps.findOne(new Query(where("name").is("Joe")), Person.class));

    mongoOps.dropCollection("person");
  }
}
```

这将产生以下输出

```
10:01:32,062 DEBUG apping.MongoPersistentEntityIndexCreator:  80 - Analyzing
class class org.spring.example.Person for index information.
10:01:32,265 DEBUG ramework.data.mongodb.core.MongoTemplate: 631 - insert
DBObject containing fields: [_class, age, name] in collection: Person
10:01:32,765 DEBUG ramework.data.mongodb.core.MongoTemplate:1243 - findOne
using query: { "name" : "Joe"} in db.collection: database.Person
10:01:32,953  INFO      org.spring.mongodb.example.MongoApp:  25 - Person
[id=4ddbba3c0be56b7e1b210166, name=Joe, age=34]
```

10:01:32,984 DEBUG ramework.data.mongodb.core.MongoTemplate: 375 - Dropped collection [database.person]

即使在这个简单的例子中，也很少有人注意到这一点

- 您可以 MongoTemplate 使用标准 com.mongodb.Mongo 对象和要使用的数据库名称来实例化 Spring Mongo 的中央帮助程序类。
- 映射器可以对照标准 POJO 对象，而无需任何其他元数据（尽管您可以选择提供该信息。请参阅此处。）。
- 约定用于处理 id 字段，将其转换为 ObjectId 存储在数据库中时的字段。
- 映射约定可以使用字段访问。请注意，Person 类只有 getter。
- 如果构造函数参数名称与存储文档的字段名称匹配，则它们将用于实例化对象

9.2。示例存储库

有一个 github 存储库，其中包含几个示例，您可以下载并使用它们来了解库的工作原理。

9.3。用 Spring 连接到 MongoDB

使用 MongoDB 和 Spring 时的首要任务之一是 com.mongodb.Mongo 使用 IoC 容器创建对象。有两种主要方法可以使用基于 Java 的 bean 元数据或基于 XML 的 bean 元数据。这些将在以下部分中讨论。

> 对于那些不熟悉如何使用基于 Java bean 的元数据，而不是 XML 来配置 Spring 容器基于元数据请参阅参考文档的高层次引进这里还有详细的文档在这里。

9.3.1。使用基于 Java 的元数据注册 Mongo 实例

下面显示了使用基于 Java 的 bean 元数据来注册 a 实例的 com.mongodb.Mongo 示例

示例 42.使用基于 Java 的 bean 元数据注册 com.mongodb.Mongo 对象

```
@Configuration
public class AppConfig {

  /*
   * Use the standard Mongo driver API to create a com.mongodb.Mongo instance.
```

```
  */
  public @Bean Mongo mongo() throws UnknownHostException {
     return new Mongo("localhost");
  }
}
```

此方法允许您使用 com.mongodb.Mongo 您可能已经习惯使用的标准 API，但也会使用 UnknownHostException 检查异常来污染代码。使用 checked 异常是不可取的，因为基于 Java 的 bean 元数据使用方法作为设置对象依赖性的手段，使调用代码变得混乱。

另一种方法是 com.mongodb.Mongo 使用 Spring 注册容器的实例实例 MongoClientFactoryBean。与 com.mongodb.Mongo 直接实例化实例相比，FactoryBean 方法不会抛出已检查的异常，并且还具有为容器提供 ExceptionTranslator 实现的附加优势，该实现将 MongoDB 异常转换为 Spring 的可移植 DataAccessException 层次结构中的异常，用于使用注释注释的数据访问类@Repository。Spring 的 DAO 支持功能中@Repository 描述了这种层次结构和使用。

@Repository 下面显示了支持带注释类的异常转换的基于 Java 的 bean 元数据的示例：

示例 43.使用 Spring 的 MongoClientFactoryBean 注册 com.mongodb.Mongo 对象并启用 Spring 的异常转换支持

```
@Configuration
public class AppConfig {

   /*
    * Factory bean that creates the com.mongodb.Mongo instance
    */
    public @Bean MongoClientFactoryBean mongo() {
        MongoClientFactoryBean mongo = new MongoClientFactoryBean();
        mongo.setHost("localhost");
        return mongo;
    }
}
```

要访问 com.mongodb.Mongo 由 MongoClientFactoryBean 其他@Configuration 或您自己的类创建的对象，请使用“private @Autowired Mongo mongo;”字段。

9.3.2。使用基于 XML 的元数据注册 Mongo 实例

虽然您可以使用 Spring 的传统<beans/>XML 命名空间来注册 com.mongodb.Mongo 容器的实例，但 XML 可能非常冗长，因为它是通用的。XML 命名空间是配置常用对象（如 Mongo 实例）的更好选择。mongo 命名空间允许您创建 Mongo 实例服务器位置，副本集和选项。

要使用 Mongo 命名空间元素，您需要引用 Mongo 架构：

示例 44.用于配置 MongoDB 的 XML 模式

```
<?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:context="http://www.springframework.org/schema/context"
       xmlns:mongo="http://www.springframework.org/schema/data/mongo"
       xsi:schemaLocation=
       "http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.0.xsd
       *http://www.springframework.org/schema/data/mongo
http://www.springframework.org/schema/data/mongo/spring-mongo-1.0.xsd*
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <!-- Default bean name is 'mongo' -->
   *<mongo:mongo host="localhost" port="27017"/>*

</beans>
```

更高级的配置 MongoOptions 如下所示（注意这些不是推荐值）

示例 45.使用 MongoOptions 配置 com.mongodb.Mongo 对象的 XML 模式

```
<beans>

 <mongo:mongo host="localhost" port="27017">
   <mongo:options connections-per-host="8"
                  threads-allowed-to-block-for-connection-multiplier="4"
                  connect-timeout="1000"
                  max-wait-time="1500}"
                  auto-connect-retry="true"
                  socket-keep-alive="true"
                  socket-timeout="1500"
                  slave-ok="true"
                  write-number="1"
                  write-timeout="0"
                  write-fsync="true"/>
 </mongo:mongo/>

</beans>
```

使用副本集的配置如下所示。

示例 46.使用副本集配置 com.mongodb.Mongo 对象的 XML 模式

```
<mongo:mongo id="replicaSetMongo" replica-
set="127.0.0.1:27017,localhost:27018"/>
```

9.3.3。MongoDbFactory 接口

虽然 com.mongodb.Mongo 是 MongoDB 驱动程序 API 的入口点，但连接到特定的 MongoDB 数据库实例需要其他信息，例如数据库名称和可选的用户名和密码。使用该信息，您可以获取 com.mongodb.DB 对象并访问特定 MongoDB 数据库实例的所有功能。Spring 提供了 org.springframework.data.mongodb.core.MongoDbFactory 下面显示的界面来引导到数据库的连接。

```
public interface MongoDbFactory {

  DB getDb() throws DataAccessException;

  DB getDb(String dbName) throws DataAccessException;
}
```

以下部分显示如何使用具有 Java 或基于 XML 的元数据的容器来配置 MongoDbFactory 接口的实例。反过来，您可以使用 MongoDbFactory 实例进行配置 MongoTemplate。

该类 org.springframework.data.mongodb.core.SimpleMongoDbFactory 提供了实现 MongoDbFactory 接口，并使用标准 com.mongodb.Mongo 实例，数据库名称和可选的 org.springframework.data.authentication.UserCredentials 构造函数参数创建。

您可以在标准 Java 代码中使用它们，而不是使用 IoC 容器来创建 MongoTemplate 实例，如下所示。

```
public class MongoApp {

  private static final Log log = LogFactory.getLog(MongoApp.class);

  public static void main(String[] args) throws Exception {

    MongoOperations mongoOps = new MongoTemplate(*new
SimpleMongoDbFactory(new Mongo(), "database")*);

    mongoOps.insert(new Person("Joe", 34));

    log.info(mongoOps.findOne(new Query(where("name").is("Joe")), Person.class));

    mongoOps.dropCollection("person");
  }
}
```

粗体代码突出显示 SimpleMongoDbFactory 的使用，并且是入门部分中显示的列表之间的唯一区别。

9.3.4。使用基于 Java 的元数据注册 MongoDbFactory 实例

要使用容器注册 MongoDbFactory 实例，您可以编写与上一代码清单中突出显示的代码类似的代码。一个简单的例子如下所示

```
@Configuration
public class MongoConfiguration {

  public @Bean MongoDbFactory mongoDbFactory() throws Exception {
    return new SimpleMongoDbFactory(new Mongo(), "database");
  }
}
```

要定义用户名和密码，请创建一个实例 org.springframework.data.authentication.UserCredentials 并将其传递给构造函数，如下所示。此列表还显示使用 MongoDbFactory 注册 MongoTemplate 的实例与容器。

```
@Configuration
public class MongoConfiguration {

  public @Bean MongoDbFactory mongoDbFactory() throws Exception {
    UserCredentials userCredentials = new UserCredentials("joe", "secret");
    return new SimpleMongoDbFactory(new Mongo(), "database", userCredentials);
  }

  public @Bean MongoTemplate mongoTemplate() throws Exception {
    return new MongoTemplate(mongoDbFactory());
  }
}
```

9.3.5。使用基于 XML 的元数据注册 MongoDbFactory 实例

SimpleMongoDbFactory 与使用<beans/>命名空间相比，mongo 命名空间提供了创建 a 的便捷方式。简单用法如下所示

```
<mongo:db-factory dbname="database">
```

在上面的示例中，com.mongodb.Mongo 使用默认主机和端口号创建实例。在 SimpleMongoDbFactory 与容器注册由 id 标识“mongoDbFactory”，除非指定了 id 属性的值。

com.mongodb.Mongo 除了数据库的用户名和密码之外，您还可以为底层实例提供主机和端口，如下所示。

```
<mongo:db-factory id="anotherMongoDbFactory"
                  host="localhost"
                  port="27017"
                  dbname="database"
                  username="joe"
```

```
        password="secret"/>
```

如果 MongoDB 身份验证数据库与目标数据库不同，请使用该 authentication-dbname 属性，如下所示。

```
<mongo:db-factory id="anotherMongoDbFactory"
        host="localhost"
        port="27017"
        dbname="database"
        username="joe"
        password="secret"
        authentication-dbname="admin"
        />
```

如果需要在 com.mongodb.Mongo 用于创建的实例上配置其他选项，则 SimpleMongoDbFactory 可以使用 mongo-ref 如下所示的属性引用现有 bean 。为了显示另一种常见的使用模式，此列表显示了使用属性占位符来参数化配置和创建 MongoTemplate。

```
<context:property-placeholder
location="classpath:/com/myapp/mongodb/config/mongo.properties"/>

<mongo:mongo host="${mongo.host}" port="${mongo.port}">
  <mongo:options
    connections-per-host="${mongo.connectionsPerHost}"
    threads-allowed-to-block-for-connection-
multiplier="${mongo.threadsAllowedToBlockForConnectionMultiplier}"
    connect-timeout="${mongo.connectTimeout}"
    max-wait-time="${mongo.maxWaitTime}"
    auto-connect-retry="${mongo.autoConnectRetry}"
    socket-keep-alive="${mongo.socketKeepAlive}"
    socket-timeout="${mongo.socketTimeout}"
    slave-ok="${mongo.slaveOk}"
    write-number="1"
    write-timeout="0"
    write-fsync="true"/>
</mongo:mongo>

<mongo:db-factory dbname="database" mongo-ref="mongo"/>

<bean id="anotherMongoTemplate"
class="org.springframework.data.mongodb.core.MongoTemplate">
  <constructor-arg name="mongoDbFactory" ref="mongoDbFactory"/>
</bean>
```

9.4。MongoTemplate 简介

The class MongoTemplate, located in the package org.springframework.data.mongodb.core, is the central class of the Spring’s MongoDB

support providing a rich feature set to interact with the database. The template offers convenience operations to create, update, delete and query for MongoDB documents and provides a mapping between your domain objects and MongoDB documents.

> Once configured, MongoTemplate is thread-safe and can be reused across multiple instances.

The mapping between MongoDB documents and domain classes is done by delegating to an implementation of the interface MongoConverter. Spring provides two implementations, SimpleMappingConverter and MappingMongoConverter, but you can also write your own converter. Please refer to the section on MongoConverters for more detailed information.

The MongoTemplate class implements the interface MongoOperations. In as much as possible, the methods on MongoOperations are named after methods available on the MongoDB driver Collection object to make the API familiar to existing MongoDB developers who are used to the driver API. For example, you will find methods such as "find", "findAndModify", "findOne", "insert", "remove", "save", "update" and "updateMulti". The design goal was to make it as easy as possible to transition between the use of the base MongoDB driver and MongoOperations. A major difference in between the two APIs is that MongoOperations can be passed domain objects instead of DBObject and there are fluent APIs for Query, Criteria, and Update operations instead of populating a DBObject to specify the parameters for those operations.

> The preferred way to reference the operations on MongoTemplate instance is via its interface MongoOperations.

The default converter implementation used by MongoTemplate is MappingMongoConverter. While the MappingMongoConverter can make use of additional metadata to specify the mapping of objects to documents it is also capable of converting objects that contain no additional metadata by using some conventions for the mapping of IDs and collection names. These conventions as well as the use of mapping annotations is explained in the Mapping chapter.

> In the M2 release SimpleMappingConverter, was the default and this class is now deprecated as its functionality has been subsumed by the MappingMongoConverter.

Another central feature of MongoTemplate is exception translation of exceptions thrown in the MongoDB Java driver into Spring's portable Data Access Exception hierarchy. Refer to the section on exception translation for more information.

While there are many convenience methods on MongoTemplate to help you easily perform common tasks if you should need to access the MongoDB driver API directly

to access functionality not explicitly exposed by the MongoTemplate you can use one of several Execute callback methods to access underlying driver APIs. The execute callbacks will give you a reference to either a com.mongodb.Collection or a com.mongodb.DB object. Please see the section mongo.executioncallback[Execution Callbacks] for more information.

Now let's look at an example of how to work with the MongoTemplate in the context of the Spring container.

9.4.1. Instantiating MongoTemplate

You can use Java to create and register an instance of MongoTemplate as shown below.

Example 47. Registering a com.mongodb.Mongo object and enabling Spring's exception translation support

```
@Configuration
public class AppConfig {

  public @Bean Mongo mongo() throws Exception {
     return new Mongo("localhost");
  }

  public @Bean MongoTemplate mongoTemplate() throws Exception {
     return new MongoTemplate(mongo(), "mydatabase");
  }
}
```

There are several overloaded constructors of MongoTemplate. These are

- MongoTemplate(Mongo mongo, String databaseName) - takes the com.mongodb.Mongo object and the default database name to operate against.
- MongoTemplate(Mongo mongo, String databaseName, UserCredentials userCredentials) - adds the username and password for authenticating with the database.
- MongoTemplate(MongoDbFactory mongoDbFactory) - takes a MongoDbFactory object that encapsulated the com.mongodb.Mongo object, database name, and username and password.
- MongoTemplate(MongoDbFactory mongoDbFactory, MongoConverter mongoConverter) - adds a MongoConverter to use for mapping.

You can also configure a MongoTemplate using Spring's XML <beans/> schema.

```
<mongo:mongo host="localhost" port="27017"/>

<bean id="mongoTemplate"
class="org.springframework.data.mongodb.core.MongoTemplate">
```

```
  <constructor-arg ref="mongo"/>
  <constructor-arg name="databaseName" value="geospatial"/>
</bean>
```

Other optional properties that you might like to set when creating a MongoTemplate are the default WriteResultCheckingPolicy, WriteConcern, and ReadPreference.

> The preferred way to reference the operations on MongoTemplate instance is via its interface MongoOperations.

9.4.2. WriteResultChecking Policy

When in development it is very handy to either log or throw an exception if the com.mongodb.WriteResult returned from any MongoDB operation contains an error. It is quite common to forget to do this during development and then end up with an application that looks like it runs successfully but in fact the database was not modified according to your expectations. Set MongoTemplate's property to an enum with the following values, LOG, EXCEPTION, or NONE to either log the error, throw and exception or do nothing. The default is to use a WriteResultChecking value of NONE.

9.4.3. WriteConcern

如果尚未通过更高级别的驱动程序指定 com.mongodb.WriteConcern，则可以设置 MongoTemplate 将用于写入操作的属性，例如 com.mongodb.Mongo。如果 WriteConcern 没有设置 MongoTemplate 的属性，它将默认为 MongoDB 驱动程序的 DB 或 Collection 设置中设置的属性。

9.4.4。WriteConcernResolver

对于要 WriteConcern 在每个操作基础上设置不同值的更高级情况（用于删除，更新，插入和保存操作），WriteConcernResolver 可以配置调用的策略接口 MongoTemplate。由于 MongoTemplate 用于持久化 POJO，因此 WriteConcernResolver 可以创建可以将特定 POJO 类映射到 WriteConcern 值的策略。的 WriteConcernResolver 接口如下所示。

```
public interface WriteConcernResolver {
  WriteConcern resolve(MongoAction action);
}
```

传入的参数 MongoAction 用于确定 WriteConcern 要使用的值或将模板本身的值用作默认值。MongoAction 包含要写入的集合名称，java.lang.ClassPOJO 的集合名称，已转换的 DBObject，以及作为枚举的操作（MongoActionOperation：REMOVE，UPDATE，INSERT，INSERT_LIST，SAVE）和一些其他上下文信息。例如，

```
private class MyAppWriteConcernResolver implements WriteConcernResolver {

 public WriteConcern resolve(MongoAction action) {
   if (action.getEntityClass().getSimpleName().contains("Audit")) {
     return WriteConcern.NONE;
   } else if (action.getEntityClass().getSimpleName().contains("Metadata")) {
     return WriteConcern.JOURNAL_SAFE;
   }
   return action.getDefaultWriteConcern();
 }
}
```

9.5。保存，更新和删除文档

MongoTemplate 提供了一种简单的方法来保存，更新和删除域对象，并将这些对象映射到 MongoDB 中存储的文档。

给出一个像 Person 这样的简单类

```
public class Person {

 private String id;
 private String name;
 private int age;

 public Person(String name, int age) {
   this.name = name;
   this.age = age;
 }

 public String getId() {
   return id;
 }
 public String getName() {
   return name;
 }
 public int getAge() {
   return age;
 }

 @Override
 public String toString() {
   return "Person [id=" + id + ", name=" + name + ", age=" + age + "]";
 }

}
```

您可以保存，更新和删除对象，如下所示。

MongoOperations 是 MongoTemplate 实现的接口。

```
package org.spring.example;

import static org.springframework.data.mongodb.core.query.Criteria.where;
import static org.springframework.data.mongodb.core.query.Update.update;
import static org.springframework.data.mongodb.core.query.Query.query;

import java.util.List;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.data.mongodb.core.MongoOperations;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.data.mongodb.core.SimpleMongoDbFactory;

import com.mongodb.Mongo;

public class MongoApp {

  private static final Log log = LogFactory.getLog(MongoApp.class);

  public static void main(String[] args) throws Exception {

    MongoOperations mongoOps = new MongoTemplate(new
SimpleMongoDbFactory(new Mongo(), "database"));

    Person p = new Person("Joe", 34);

    // Insert is used to initially store the object into the database.
    mongoOps.insert(p);
    log.info("Insert: " + p);

    // Find
    p = mongoOps.findById(p.getId(), Person.class);
    log.info("Found: " + p);

    // Update
    mongoOps.updateFirst(query(where("name").is("Joe")), update("age", 35),
Person.class);
    p = mongoOps.findOne(query(where("name").is("Joe")), Person.class);
    log.info("Updated: " + p);

    // Delete
    mongoOps.remove(p);

    // Check that deletion worked
    List<Person> people =  mongoOps.findAll(Person.class);
    log.info("Number of people = : " + people.size());
```

```
    mongoOps.dropCollection(Person.class);
  }
}
```

这将产生以下日志输出（包括 MongoTemplate 自身的调试消息）

```
DEBUG apping.MongoPersistentEntityIndexCreator:  80 - Analyzing class class
org.spring.example.Person for index information.
DEBUG work.data.mongodb.core.MongoTemplate: 632 - insert DBObject containing
fields: [_class, age, name] in collection: person
INFO            org.spring.example.MongoApp:  30 - Insert: Person
[id=4ddc6e784ce5b1eba3ceaf5c, name=Joe, age=34]
DEBUG work.data.mongodb.core.MongoTemplate:1246 - findOne using query:
{ "_id" : { "$oid" : "4ddc6e784ce5b1eba3ceaf5c"}} in db.collection: database.person
INFO            org.spring.example.MongoApp:  34 - Found: Person
[id=4ddc6e784ce5b1eba3ceaf5c, name=Joe, age=34]
DEBUG work.data.mongodb.core.MongoTemplate: 778 - calling update using query:
{ "name" : "Joe"} and update: { "$set" : { "age" : 35}} in collection: person
DEBUG work.data.mongodb.core.MongoTemplate:1246 - findOne using query:
{ "name" : "Joe"} in db.collection: database.person
INFO            org.spring.example.MongoApp:  39 - Updated: Person
[id=4ddc6e784ce5b1eba3ceaf5c, name=Joe, age=35]
DEBUG work.data.mongodb.core.MongoTemplate: 823 - remove using query: { "id" :
"4ddc6e784ce5b1eba3ceaf5c"} in collection: person
INFO            org.spring.example.MongoApp:  46 - Number of people = : 0
DEBUG work.data.mongodb.core.MongoTemplate: 376 - Dropped collection
[database.person]
```

There was implicit conversion using the MongoConverter between a String and ObjectId as stored in the database and recognizing a convention of the property "Id" name.

> This example is meant to show the use of save, update and remove operations on MongoTemplate and not to show complex mapping functionality

The query syntax used in the example is explained in more detail in the section Querying Documents.

9.5.1. How the _id field is handled in the mapping layer

MongoDB requires that you have an _id field for all documents. If you don’t provide one the driver will assign a ObjectId with a generated value. When using the MappingMongoConverter there are certain rules that govern how properties from the Java class is mapped to this _id field.

以下概述了将哪些属性映射到_id 文档字段：

- 用@Id（org.springframework.data.annotation.Id）注释的属性或字段将映射到该_id 字段。

- 没有注释但已命名的属性或字段 id 将映射到该_id 字段。

下面概述了在使用 MappingMongoConverter 默认值时，将在映射到_id 文档字段的属性上执行的类型转换（如果有）MongoTemplate。

- 在 Java 类中声明为 String 的 id 属性或字段将 ObjectId 使用 Spring 转换为并尽可能存储 Converter<String, ObjectId>。有效的转换规则被委托给 MongoDB Java 驱动程序。如果无法将其转换为 ObjectId，则该值将作为字符串存储在数据库中。

- BigInteger 在 Java 类中声明的 id 属性或字段将转换为并 ObjectId 使用 Spring 存储 Converter<BigInteger, ObjectId>。

如果 Java 类中不存在上面指定的字段或属性，则_id 驱动程序将生成隐式文件，但不会映射到 Java 类的属性或字段。

查询和更新时，MongoTemplate 将使用转换器处理与上述保存文档规则对应的对象 Query 和 Update 对象的转换，以便查询中使用的字段名称和类型能够与域类中的字段名称和类型相匹配。

9.5.2。类型映射

由于 MongoDB 集合可以包含表示各种类型实例的文档。这里一个很好的例子是，如果您存储类的层次结构，或者只是具有类型属性的类 Object。在后一种情况下，在检索对象时必须正确读取该属性内的值。因此，我们需要一种机制来存储类型信息和实际文档。

为了实现这一点，MappingMongoConverter 使用 MongoTypeMapper 抽象 DefaultMongoTypeMapper 作为它的主要实现。它的默认行为是_class 在顶层文档的文档内部以及每个值存储完全限定的类名，如果它是复杂类型和声明的属性类型的子类型。

示例 48.类型映射

```
public class Sample {
  Contact value;
}

public abstract class Contact { … }

public class Person extends Contact { … }
```

```
Sample sample = new Sample();
sample.value = new Person();

mongoTemplate.save(sample);

{ "_class" : "com.acme.Sample",
  "value" : { "_class" : "com.acme.Person" }
}
```

正如您所看到的，我们存储实际根类持久性的类型信息以及嵌套类型，因为它是复杂的，并且是子类型 Contact。因此，如果您现在正在使用 mongoTemplate.findAll(Object.class, "sample")我们能够发现存储的文档应该是一个 Sample 实例。我们还能够发现 value 属性应该是 Person 实际的。

自定义类型映射

如果您想避免将整个 Java 类名称作为类型信息编写，而是想使用某些键，则可以使用@TypeAlias 持久化的实体类中的注释。如果您需要自定义映射，请查看 TypeInformationMapper 界面。可以在 DefaultMongoTypeMapper 可以依次配置的接口上配置该接口的实例 MappingMongoConverter。

例 49.为实体定义 TypeAlias

```
@TypeAlias("pers")
class Person {

}
```

Note that the resulting document will contain "pers" as the value in the _class Field.

Configuring custom type mapping

The following example demonstrates how to configure a custom MongoTypeMapper in MappingMongoConverter.

Example 50. Configuring a custom MongoTypeMapper via Spring Java Config

```
class CustomMongoTypeMapper extends DefaultMongoTypeMapper {
  //implement custom type mapping here
}
@Configuration
class SampleMongoConfiguration extends AbstractMongoConfiguration {

  @Override
  protected String getDatabaseName() {
    return "database";
  }

  @Override
  public Mongo mongo() throws Exception {
```

```
    return new Mongo();
  }

  @Bean
  @Override
  public MappingMongoConverter mappingMongoConverter() throws Exception {
    MappingMongoConverter mmc = super.mappingMongoConverter();
    mmc.setTypeMapper(customTypeMapper());
    return mmc;
  }

  @Bean
  public MongoTypeMapper customTypeMapper() {
    return new CustomMongoTypeMapper();
  }
}
```

Note that we are extending the AbstractMongoConfiguration class and override the bean definition of the MappingMongoConverter where we configure our custom MongoTypeMapper.

Example 51. Configuring a custom MongoTypeMapper via XML

```
<mongo:mapping-converter type-mapper-ref="customMongoTypeMapper"/>

<bean name="customMongoTypeMapper"
class="com.bubu.mongo.CustomMongoTypeMapper"/>
```

9.5.3. Methods for saving and inserting documents

There are several convenient methods on MongoTemplate for saving and inserting your objects. To have more fine-grained control over the conversion process you can register Spring converters with the MappingMongoConverter, for example Converter<Person, DBObject> and Converter<DBObject, Person>.

> The difference between insert and save operations is that a save operation will perform an insert if the object is not already present.

The simple case of using the save operation is to save a POJO. In this case the collection name will be determined by name (not fully qualified) of the class. You may also call the save operation with a specific collection name. The collection to store the object can be overridden using mapping metadata.

When inserting or saving, if the Id property is not set, the assumption is that its value will be auto-generated by the database. As such, for auto-generation of an ObjectId to succeed the type of the Id property/field in your class must be either a String, ObjectId, or BigInteger.

Here is a basic example of using the save operation and retrieving its contents.

Example 52. Inserting and retrieving documents using the MongoTemplate

```
import static org.springframework.data.mongodb.core.query.Criteria.where;
import static org.springframework.data.mongodb.core.query.Criteria.query;
…

Person p = new Person("Bob", 33);
mongoTemplate.insert(p);

Person qp = mongoTemplate.findOne(query(where("age").is(33)), Person.class);
```

The insert/save operations available to you are listed below.

- void **save** (Object objectToSave) Save the object to the default collection.
- void **save** (Object objectToSave, String collectionName) Save the object to the specified collection.

A similar set of insert operations is listed below

- void **insert** (Object objectToSave) Insert the object to the default collection.
- void **insert** (Object objectToSave, String collectionName) Insert the object to the specified collection.

Which collection will my documents be saved into?

There are two ways to manage the collection name that is used for operating on the documents. The default collection name that is used is the class name changed to start with a lower-case letter. So a com.test.Person class would be stored in the "person" collection. You can customize this by providing a different collection name using the @Document annotation. You can also override the collection name by providing your own collection name as the last parameter for the selected MongoTemplate method calls.

Inserting or saving individual objects

The MongoDB driver supports inserting a collection of documents in one operation. The methods in the MongoOperations interface that support this functionality are listed below

- **insert** inserts an object. If there is an existing document with the same id then an error is generated.
- **insertAll** takes a Collection of objects as the first parameter. This method inspects each object and inserts it to the appropriate collection based on the rules specified above.

- **save** saves the object overwriting any object that might exist with the same id.

Inserting several objects in a batch

The MongoDB driver supports inserting a collection of documents in one operation. The methods in the MongoOperations interface that support this functionality are listed below

- **insert** methods that take a Collection as the first argument. This inserts a list of objects in a single batch write to the database.

9.5.4. Updating documents in a collection

For updates we can elect to update the first document found using MongoOperation 's method updateFirst or we can update all documents that were found to match the query using the method updateMulti. Here is an example of an update of all SAVINGS accounts where we are adding a one-time $50.00 bonus to the balance using the $inc operator.

Example 53. Updating documents using the MongoTemplate

```
import static org.springframework.data.mongodb.core.query.Criteria.where;
import static org.springframework.data.mongodb.core.query.Query;
import static org.springframework.data.mongodb.core.query.Update;

...

WriteResult wr = mongoTemplate.updateMulti(new
Query(where("accounts.accountType").is(Account.Type.SAVINGS)),
  new Update().inc("accounts.$.balance", 50.00), Account.class);
```

In addition to the Query discussed above we provide the update definition using an Update object. The Update class has methods that match the update modifiers available for MongoDB.

As you can see most methods return the Update object to provide a fluent style for the API.

Methods for executing updates for documents

- **updateFirst** Updates the first document that matches the query document criteria with the provided updated document.

- **updateMulti** Updates all objects that match the query document criteria with the provided updated document.

Methods for the Update class

The Update class can be used with a little 'syntax sugar' as its methods are meant to be chained together and you can kick-start the creation of a new Update instance via the

static method public static Update update(String key, Object value) and using static imports.

Here is a listing of methods on the Update class

- Update **addToSet** (String key, Object value) Update using the $addToSet update modifier
- Update **currentDate** (String key) Update using the $currentDate update modifier
- Update **currentTimestamp** (String key) Update using the $currentDate update modifier with $type timestamp
- Update **inc** (String key, Number inc) Update using the $inc update modifier
- Update **max** (String key, Object max) Update using the $max update modifier
- Update **min** (String key, Object min) Update using the $min update modifier
- Update **multiply** (String key, Number multiplier) Update using the $mul update modifier
- Update **pop** (String key, Update.Position pos) Update using the $pop update modifier
- Update **pull** (String key, Object value) Update using the $pull update modifier
- Update **pullAll** (String key, Object[] values) Update using the $pullAll update modifier
- Update **push** (String key, Object value) Update using the $push update modifier
- Update **pushAll** (String key, Object[] values) Update using the $pushAll update modifier
- Update **rename** (String oldName, String newName) Update using the $rename update modifier
- Update **set** (String key, Object value) Update using the $set update modifier
- Update **setOnInsert** (String key, Object value) Update using the $setOnInsert update modifier
- Update **unset** (String key) Update using the $unset update modifier

Some update modifiers like $push and $addToSet allow nesting of additional operators.

// { $push : { "category" : { "$each" : ["spring" , "data"] } } }

```
new Update().push("category").each("spring", "data")

// { $push : { "key" : { "$position" : 0 , "$each" : [ "Arya" , "Arry" , "Weasel" ] } } }
new Update().push("key").atPosition(Position.FIRST).each(Arrays.asList("Arya",
"Arry", "Weasel"));

// { $push : { "key" : { "$slice" : 5 , "$each" : [ "Arya" , "Arry" , "Weasel" ] } } }
new Update().push("key").slice(5).each(Arrays.asList("Arya", "Arry", "Weasel"));
// { $addToSet : { "values" : { "$each" : [ "spring" , "data" , "mongodb" ] } } }
new Update().addToSet("values").each("spring", "data", "mongodb");
```

9.5.5. Upserting documents in a collection

Related to performing an updateFirst operations, you can also perform an upsert operation which will perform an insert if no document is found that matches the query. The document that is inserted is a combination of the query document and the update document. Here is an example

```
template.upsert(query(where("ssn").is(1111).and("firstName").is("Joe").and("Fraizer")
.is("Update")), update("address", addr), Person.class);
```

9.5.6. Finding and Upserting documents in a collection

The findAndModify(…) method on DBCollection can update a document and return either the old or newly updated document in a single operation. MongoTemplate provides a findAndModify method that takes Query and Update classes and converts from DBObject to your POJOs. Here are the methods

```
<T> T findAndModify(Query query, Update update, Class<T> entityClass);

<T> T findAndModify(Query query, Update update, Class<T> entityClass, String
collectionName);

<T> T findAndModify(Query query, Update update, FindAndModifyOptions options,
Class<T> entityClass);

<T> T findAndModify(Query query, Update update, FindAndModifyOptions options,
Class<T> entityClass, String collectionName);
```

作为示例用法，我们将向 Person 容器中插入少量对象并执行简单的 findAndUpdate 操作

```
mongoTemplate.insert(new Person("Tom", 21));
mongoTemplate.insert(new Person("Dick", 22));
mongoTemplate.insert(new Person("Harry", 23));

Query query = new Query(Criteria.where("firstName").is("Harry"));
Update update = new Update().inc("age", 1);
```

```
Person p = mongoTemplate.findAndModify(query, update, Person.class); // return's
old person object

assertThat(p.getFirstName(), is("Harry"));
assertThat(p.getAge(), is(23));
p = mongoTemplate.findOne(query, Person.class);
assertThat(p.getAge(), is(24));

// Now return the newly updated document when updating
p = template.findAndModify(query, update, new
FindAndModifyOptions().returnNew(true), Person.class);
assertThat(p.getAge(), is(25));
```

将 FindAndModifyOptions 允许您设置 returnNew，UPSERT 的选项，并删除。扩展前一代码片段的示例如下所示

```
Query query2 = new Query(Criteria.where("firstName").is("Mary"));
p = mongoTemplate.findAndModify(query2, update, new
FindAndModifyOptions().returnNew(true).upsert(true), Person.class);
assertThat(p.getFirstName(), is("Mary"));
assertThat(p.getAge(), is(1));
```

9.5.7。删除文档的方法

您可以使用多个重载方法从数据库中删除对象。

- **remove** 基于以下内容之一删除给定文档：特定对象实例，与类或查询文档条件相结合的查询文档条件以及特定集合名称。

9.5.8。乐观锁定

该@Version 注释提供了类似的语义 MongoDB 中的背景下，JPA 并确保更新只适用于具有匹配版本的文件。因此，版本属性的实际值将添加到更新查询中，如果另一个操作在其间更改了文档，则更新将不会产生任何影响。在这种情况下，OptimisticLockingFailureException 抛出一个。

```
@Document
class Person {

  @Id String id;
  String firstname;
  String lastname;
  @Version Long version;
}

Person daenerys = template.insert(new Person("Daenerys"));                 (1)
```

```
Person tmp = teplate.findOne(query(where("id").is(daenerys.getId())), Person.class);
(2)

daenerys.setLastname("Targaryen");
template.save(daenerys);                                    (3)

template.save(tmp); // throws OptimisticLockingFailureException          (4)
```

1	最初插入文档。version 设置为 0。
2	加载刚插入的文件 version 仍然是 0。
3	用的更新文档 version = 0。设置 lastname 和碰撞 version 到 1。
4	尝试更新以前加载的文档窗台 version = 0，OptimisticLockingFailureException 因为当前 version 是失败的 1。 使用 MongoDB 驱动程序版本 3 需要设置 WriteConcern 为 ACKNOWLEDGED。否则 OptimisticLockingFailureException 可以默默吞噬。

9.6。查询文件

你可以表达使用您的疑问 Query 和 Criteria 它具有反映本地的 MongoDB 运营商的名称，如方法名称类 lt，lte，is，等。在 Query 和 Criteria 类遵循流畅 API 风格，让您可以轻松地串联多个方法标准，同时具有易于理解的代码查询到一起。Java 中的静态导入用于帮助消除为创建 Query 和 Criteria 实例而查看“new”关键字的需要，从而提高可读性。如果您想 Query 从普通的 JSON String 使用创建实例 BasicQuery。

示例 54.从普通 JSON 字符串创建 Query 实例

```
BasicQuery query = new BasicQuery("{ age : { $lt : 50 }, accounts.balance : { $gt : 1000.00 }}");
List<Person> result = mongoTemplate.find(query, Person.class);
```

还支持地理空间查询，并在地理空间查询部分中进行了更详细的描述。

还支持 Map-Reduce 操作，并在 Map-Reduce 一节中进行了更详细的描述。

9.6.1。查询集合中的文档

我们在前面的部分中看到了如何使用 MongoTemplate 上的 findOne 和 findById 方法检索单个文档，这些方法返回单个域对象。我们还可以查询要作为域对象列表返回的文档集合。假设我们有一些 Person 对象，其名称和年龄存储为集合中的文档，并且每个人都有一个带有余额的嵌入式帐户文档。我们现在可以使用以下代码运行查询。

例 55.使用 MongoTemplate 查询文档

```
import static org.springframework.data.mongodb.core.query.Criteria.where;
import static org.springframework.data.mongodb.core.query.Query.query;

…

List<Person> result = mongoTemplate.find(query(where("age").lt(50)
  .and("accounts.balance").gt(1000.00d)), Person.class);
```

所有 find 方法都将 Query 对象作为参数。此对象定义用于执行查询的条件和选项。使用 Criteria 具有名为 where 用于实例化新 Criteria 对象的静态工厂方法的对象来指定条件。我们建议使用静态导入 org.springframework.data.mongodb.core.query.Criteria.where，Query.query 以使查询更具可读性。

This query should return a list of Person objects that meet the specified criteria. The Criteria class has the following methods that correspond to the operators provided in MongoDB.

As you can see most methods return the Criteria object to provide a fluent style for the API.

Methods for the Criteria class

- Criteria **all** (Object o) Creates a criterion using the $all operator
- Criteria **and** (String key) Adds a chained Criteria with the specified key to the current Criteria and returns the newly created one
- Criteria **andOperator** (Criteria… criteria) Creates an and query using the $and operator for all of the provided criteria (requires MongoDB 2.0 or later)
- Criteria **elemMatch** (Criteria c) Creates a criterion using the $elemMatch operator
- Criteria **exists** (boolean b) Creates a criterion using the $exists operator
- Criteria **gt** (Object o) Creates a criterion using the $gt operator
- Criteria **gte** (Object o) Creates a criterion using the $gte operator
- Criteria **in** (Object… o) Creates a criterion using the $in operator for a varargs argument.

- Criteria **in** (Collection<?> collection) Creates a criterion using the $in operator using a collection
- Criteria **is** (Object o) Creates a criterion using the $is operator
- Criteria **lt** (Object o) Creates a criterion using the $lt operator
- Criteria **lte** (Object o) Creates a criterion using the $lte operator
- Criteria **mod** (Number value, Number remainder) Creates a criterion using the $mod operator
- Criteria **ne** (Object o) Creates a criterion using the $ne operator
- Criteria **nin** (Object… o) Creates a criterion using the $nin operator
- Criteria **norOperator** (Criteria… criteria) Creates an nor query using the $nor operator for all of the provided criteria
- Criteria **not** () Creates a criterion using the $not meta operator which affects the clause directly following
- Criteria **orOperator** (Criteria… criteria) Creates an or query using the $or operator for all of the provided criteria
- Criteria **regex** (String re) Creates a criterion using a $regex
- Criteria **size** (int s) Creates a criterion using the $size operator
- Criteria **type** (int t) Creates a criterion using the $type operator

There are also methods on the Criteria class for geospatial queries. Here is a listing but look at the section on GeoSpatial Queries to see them in action.

- Criteria **within** (Circle circle) Creates a geospatial criterion using $geoWithin $center operators.
- Criteria **within** (Box box) Creates a geospatial criterion using a $geoWithin $box operation.
- Criteria **withinSphere** (Circle circle) Creates a geospatial criterion using $geoWithin $center operators.
- Criteria **near** (Point point) Creates a geospatial criterion using a $near operation
- Criteria **nearSphere** (Point point) Creates a geospatial criterion using $nearSphere$center operations. This is only available for MongoDB 1.7 and higher.

- Criteria **minDistance** (double minDistance) Creates a geospatial criterion using the $minDistance operation, for use with $near.

- Criteria **maxDistance** (double maxDistance) Creates a geospatial criterion using the $maxDistance operation, for use with $near.

The Query class has some additional methods used to provide options for the query.

Methods for the Query class

- Query **addCriteria** (Criteria criteria) used to add additional criteria to the query

- Field **fields** () used to define fields to be included in the query results

- Query **limit** (int limit) used to limit the size of the returned results to the provided limit (used for paging)

- Query **skip** (int skip) used to skip the provided number of documents in the results (used for paging)

- Query **with** (Sort sort) used to provide sort definition for the results

9.6.2. Methods for querying for documents

The query methods need to specify the target type T that will be returned and they are also overloaded with an explicit collection name for queries that should operate on a collection other than the one indicated by the return type.

- **findAll** Query for a list of objects of type T from the collection.

- **findOne** 将集合上的即席查询的结果映射到指定类型的对象的单个实例。

- **findById** 返回给定 id 和目标类的对象。

- **find** 将集合上的即席查询的结果映射到指定类型的 List。

- **findAndRemove** 将集合上的即席查询的结果映射到指定类型的对象的单个实例。将返回与查询匹配的第一个文档，并将其从数据库中的集合中删除。

9.6.3。地理空间查询

MongoDB 的支持通过使用等运营商的地理空间查询$near，$within，geoWithin 和$nearSphere。Criteria 该类可以使用特定于地理空间查询的方法。也有一些形状类，Box，Circle，和 Point 那些与地理信息相关的配合使用 Criteria 方法。

要了解如何执行 GeoSpatial 查询，我们将使用以下 Venue 类来自依赖于使用 rich 的集成测试 MappingMongoConverter。

```
@Document(collection="newyork")
public class Venue {

  @Id
  private String id;
  private String name;
  private double[] location;

  @PersistenceConstructor
  Venue(String name, double[] location) {
    super();
    this.name = name;
    this.location = location;
  }

  public Venue(String name, double x, double y) {
    super();
    this.name = name;
    this.location = new double[] { x, y };
  }

  public String getName() {
    return name;
  }

  public double[] getLocation() {
    return location;
  }

  @Override
  public String toString() {
    return "Venue [id=" + id + ", name=" + name + ", location="
        + Arrays.toString(location) + "]";
  }
}
```

要查找 a 中的位置 Circle，可以使用以下查询。

```
Circle circle = new Circle(-73.99171, 40.738868, 0.01);
List<Venue> venues =
    template.find(new Query(Criteria.where("location").within(circle)), Venue.class);
```

为了在 Circle 使用球面坐标内找到场地，可以使用以下查询

```
Circle circle = new Circle(-73.99171, 40.738868, 0.003712240453784);
List<Venue> venues =
    template.find(new Query(Criteria.where("location").withinSphere(circle)),
Venue.class);
```

要查找 Box 以下查询中的场地，可以使用

```
//lower-left then upper-right
Box box = new Box(new Point(-73.99756, 40.73083), new Point(-73.988135,
40.741404));
List<Venue> venues =
   template.find(new Query(Criteria.where("location").within(box)), Venue.class);
```

要查找 a 附近的场所 Point，可以使用以下查询

```
Point point = new Point(-73.99171, 40.738868);
List<Venue> venues =
   template.find(new
Query(Criteria.where("location").near(point).maxDistance(0.01)), Venue.class);
Point point = new Point(-73.99171, 40.738868);
List<Venue> venues =
   template.find(new
Query(Criteria.where("location").near(point).minDistance(0.01).maxDistance(100)),
Venue.class);
```

为了找到 Point 使用球面坐标附近的场地，可以使用以下查询

```
Point point = new Point(-73.99171, 40.738868);
List<Venue> venues =
   template.find(new Query(

Criteria.where("location").nearSphere(point).maxDistance(0.003712240453784)),
      Venue.class);
```

地理附近的查询

MongoDB 支持在数据库中查询地理位置，并在同一时间计算与给定原点的距离。通过地理附近查询，可以表达如下查询：“查找周围 10 英里内的所有餐馆”。为此，我们 MongoOperations 提供 geoNear(…)了采用 NearQueryas 参数的方法以及已经熟悉的实体类型和集合

```
Point location = new Point(-73.99171, 40.738868);
NearQuery query = NearQuery.near(location).maxDistance(new Distance(10,
Metrics.MILES));

GeoResults<Restaurant> = operations.geoNear(query, Restaurant.class);
```

正如您所看到的，我们使用 NearQuery 构建器 API 来设置查询，以便 Restaurant 将给定周围的所有实例返回 Point 最多 10 英里。Metrics 这里使用的枚举实际上实现了一个接口，以便其他指标也可以插入一个距离。A Metric 由乘数支持，以将给定度量的距离值转换为本机距离。此处显示的示例将 10 视为英里。使用其中一个预先构建的度量标准（英里和公里）将自动触发在查询上设置的球形标

记。如果你想避免这种情况，只需将普通 double 值输入即可 maxDistance(…)。欲了解更多信息，请参阅的 JavaDoc 中 NearQuery 和 Distance。

geo near 操作返回一个 GeoResults 封装 GeoResult 实例的包装器对象。包装 GeoResults 允许访问所有结果的平均距离。单个 GeoResult 对象只是携带找到的实体加上它与原点的距离。

9.6.4。GeoJSON 支持

MongoDB 支持地理空间数据的 GeoJSON 和简单（传统）坐标对。这些格式既可用于存储也可用于查询数据。

> 请参阅 GeoJSON 支持的 MongoDB 手册以了解要求和限制。

域类中的 GeoJSON 类型

在域类中使用 GeoJSON 类型是直截了当的。该 org.springframework.data.mongodb.core.geo 软件包包含类型，如 GeoJsonPoint，GeoJsonPolygon 等。这些是现有 org.springframework.data.geo 类型的扩展。

```
public class Store {

		String id;

		/**
		 * location is stored in GeoJSON format.
		 * {
		 *   "type" : "Point",
		 *   "coordinates" : [ x, y ]
		 * }
		 */
		GeoJsonPoint location;
}
```

存储库查询方法中的 GeoJSON 类型

使用 GeoJSON 类型作为存储库查询参数会$geometry 在创建查询时强制使用运算符。

```
public interface StoreRepository extends CrudRepository<Store, String> {

		List<Store> findByLocationWithin(Polygon polygon);  (1)

}
```

```
/*
 * {
 *   "location": {
 *     "$geoWithin": {
 *       "$geometry": {
 *         "type": "Polygon",
 *         "coordinates": [
 *           [
 *             [-73.992514,40.758934],
 *             [-73.961138,40.760348],
 *             [-73.991658,40.730006],
 *             [-73.992514,40.758934]
 *           ]
 *         ]
 *       }
 *     }
 *   }
 * }
 */
repo.findByLocationWithin(                    (2)
  new GeoJsonPolygon(
    new Point(-73.992514, 40.758934),
    new Point(-73.961138, 40.760348),
    new Point(-73.991658, 40.730006),
    new Point(-73.992514, 40.758934)));          (3)

/*
 * {
 *   "location" : {
 *     "$geoWithin" : {
 *       "$polygon" : [ [-73.992514,40.758934] , [-73.961138,40.760348] , [-
73.991658,40.730006] ]
 *     }
 *   }
 * }
 */
repo.findByLocationWithin(                    (4)
  new Polygon(
    new Point(-73.992514, 40.758934),
    new Point(-73.961138, 40.760348),
    new Point(-73.991658, 40.730006));
```

1	使用 commons 类型的存储库方法定义允许使用 GeoJSON 和传统格式调用它。
2	使用 GeoJSON 类型使用$geometry 运算符。

3	请注意，GeoJSON 多边形需要定义一个闭环。
4	使用遗留格式$polygon 运算符。

9.6.5。全文查询

由于可以使用$text 运算符执行 MongoDB 2.6 全文查询。特定于全文查询的方法和操作可在 TextQuery 和中使用 TextCriteria。在进行全文搜索时，请参阅 MongoDB 参考资料，了解其行为和限制。

全文检索

在我们实际能够使用全文搜索之前，我们必须确保正确设置搜索索引。请参阅文本索引部分以创建索引结构。

```
db.foo.createIndex(
{
 title : "text",
 content : "text"
},
{
 weights : {
            title : 3
           }
}
)
```

搜索 coffee cake，根据相关性排序的查询 weights 可以定义并执行为：

```
Query query = TextQuery.searching(new TextCriteria().matchingAny("coffee",
"cake")).sortByScore();
List<Document> page = template.find(query, Document.class);
```

可以通过在术语前加上-或使用前缀来直接排除搜索术语 notMatching

```
// search for 'coffee' and not 'cake'
TextQuery.searching(new TextCriteria().matching("coffee").matching("-cake"));
TextQuery.searching(new TextCriteria().matching("coffee").notMatching("cake"));
```

按 TextCriteria.matching 原样提供的术语。因此，短语可以通过将它们放在双引号之间来定义（例如\"coffee cake\")或使用 TextCriteria.phrase.

```
// search for phrase 'coffee cake'
TextQuery.searching(new TextCriteria().matching("\"coffee cake\""));
```

TextQuery.searching(new TextCriteria().phrase("coffee cake"));

可以通过相应的方法设置$caseSensitive 和的标志。请注意，MongoDB 3.2 中引入了这两个可选标志，除非明确设置，否则不会包含在查询中。
$diacriticSensitiveTextCriteria

9.7。按示例查询

9.7.1。介绍

本章将向您介绍 Query by Example 并解释如何使用 Examples。

按示例查询（QBE）是一种用户友好的查询技术，具有简单的界面。它允许动态创建查询，不需要编写包含字段名称的查询。实际上，Query by Example 不需要使用特定于商店的查询语言来编写查询。

9.7.2。用法

Query by Example API 由三部分组成：

- 探测器：这是具有填充字段的域对象的实际示例。
- ExampleMatcher：ExampleMatcher 携带有关如何匹配特定字段的详细信息。它可以在多个示例中重用。
- Example：一个 Example 由探针和 ExampleMatcher。它用于创建查询。

按示例查询适用于多个用例，但也有限制：

何时使用

- 使用一组静态或动态约束查询数据存储
- 频繁重构域对象，而不必担心破坏现有查询
- 独立于底层数据存储 API 工作

限制

- 不支持嵌套/分组属性约束 firstname = ?0 or (firstname = ?1 and lastname = ?2)
- 仅支持字符串的开始/包含/结束/正则表达式匹配以及其他属性类型的精确匹配

在开始使用 Query by Example 之前，您需要拥有一个域对象。要开始，只需为您的存储库创建一个接口：

例 56.示例 Person 对象

```
public class Person {

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

  // … getters and setters omitted
}
```

这是一个简单的域对象。你可以用它来创建一个 Example。默认情况下，null 将忽略具有值的字段，并使用特定于商店的默认值匹配字符串。可以使用 of 工厂方法或使用来构建示例 ExampleMatcher。Example 是不可改变的。

例 57.简单例子

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

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

1	创建域对象的新实例
2	设置要查询的属性
3	创建 Example

理想情况下，可以使用存储库执行示例。为此，请让您的存储库接口扩展 QueryByExampleExecutor<T>。这是 QueryByExampleExecutor 界面的摘录：

例 58. QueryByExampleExecutor

```
public interface QueryByExampleExecutor<T> {

  <S extends T> S findOne(Example<S> example);

  <S extends T> Iterable<S> findAll(Example<S> example);

  // … more functionality omitted.
}
```

您可以在下面阅读有关按示例执行查询的更多信息。

9.7.3。示例匹配器

示例不限于默认设置。您可以使用以下命令为字符串匹配，空值处理和特定于属性的设置指定自己的默认值 ExampleMatcher。

例 59.具有自定义匹配的示例匹配器

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

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

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

1	创建域对象的新实例。
2	设置属性。
3	创建一个 ExampleMatcher 以期望所有值匹配。即使没有进一步配置，它也可以在此阶段使用。
4	构造一个新的 ExampleMatcher 以忽略属性路径 lastname。
五	构造一个 new ExampleMatcher 来忽略属性路径 lastname 并包含空值。
6	构造一个 new ExampleMatcher 来忽略属性路径 lastname，包含空值，并使用 perform 后缀字符串匹配。
7	Example 根据域对象和配置创建新的 ExampleMatcher。

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

您可以为单个属性指定行为（例如，对于嵌套属性，“firstname”和“lastname”，“address.city”）。您可以使用匹配选项和区分大小写来调整它。

示例 60.配置匹配器选项

```
ExampleMatcher matcher = ExampleMatcher.matching()
```

```
  .withMatcher("firstname", endsWith())
  .withMatcher("lastname", startsWith().ignoreCase());
}
```

配置匹配器选项的另一种方式是使用 Java 8 lambdas。此方法是一个回调，要求实现者修改匹配器。由于配置选项保存在匹配器实例中，因此不需要返回匹配器。

例 61.使用 lambdas 配置 matcher 选项

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

通过 Example 使用配置的合并视图创建的查询。可以在 ExampleMatcher 级别设置默认匹配设置，而可以将单个设置应用于特定属性路径。已设置上的设置 ExampleMatcher 由属性路径设置继承，除非它们被明确定义。属性修补程序上的设置优先于默认设置。

表 2. ExampleMatcher 设置范围

设置	范围
空操作	ExampleMatcher
字符串匹配	ExampleMatcher 和财产路径
忽略属性	物业路径
区分大小写	ExampleMatcher 和财产路径
价值转变	物业路径

9.7.4。执行一个例子

示例 62.使用存储库按示例查询

```
public interface PersonRepository extends QueryByExampleExecutor<Person> {

}
```

```
public class PersonService {

  @Autowired PersonRepository personRepository;

  public List<Person> findPeople(Person probe) {
    return personRepository.findAll(Example.of(probe));
  }
}
```

一种 Example 包含非类型化的 ExampleSpec 使用存储库类型及其集合名称。键入 ExampleSpec 使用其类型作为结果类型和存储库中的集合名称。

> 当 null 在 ExampleSpecSpring Data 中包含值时，Mongo 使用嵌入式文档匹配而不是点符号属性匹配。这会强制对嵌入文档中的所有属性值和属性顺序进行精确的文档匹配。

Spring Data MongoDB 支持以下匹配选项：

表 3. StringMatcher 选项

匹配	逻辑结果
DEFAULT （区分大小写）	{"firstname" : firstname}
DEFAULT （不区分大小写）	{"firstname" : { $regex: firstname, $options: 'i'}}
EXACT （区分大小写）	{"firstname" : { $regex: /^firstname$/}}
EXACT （不区分大小写）	{"firstname" : { $regex: /^firstname$/, $options: 'i'}}
STARTING （区分大小写）	{"firstname" : { $regex: /^firstname/}}
STARTING （不区分大小写）	{"firstname" : { $regex: /^firstname/, $options: 'i'}}

表 3. StringMatcher 选项

匹配	逻辑结果
ENDING （区分大小写）	{"firstname" : { $regex: /firstname$/}}
ENDING （不区分大小写）	{"firstname" : { $regex: /firstname$/, $options: 'i'}}
CONTAINING （区分大小写）	{"firstname" : { $regex: /.*firstname.*/}}
CONTAINING （不区分大小写）	{"firstname" : { $regex: /.*firstname.*/, $options: 'i'}}
REGEX （区分大小写）	{"firstname" : { $regex: /firstname/}}
REGEX （不区分大小写）	{"firstname" : { $regex: /firstname/, $options: 'i'}}

9.8。地图减少操作

您可以使用 Map-Reduce 查询 MongoDB，这对批处理，数据聚合以及查询语言无法满足您的需求非常有用。

Spring 通过在 MongoOperations 上提供方法来简化 Map-Reduce 操作的创建和执行，从而提供与 MongoDB map reduce 的集成。它可以将 Map-Reduce 操作的结果转换为 POJO，还可以与 Spring 的 Resource 抽象抽象集成。这将允许您将 JavaScript 文件放在文件系统，类路径，http 服务器或任何其他 Spring Resource 实现上，然后通过简单的 URI 样式语法引用 JavaScript 资源，例如'classpath：reduce.js;。在文件中外化 JavaScript 代码通常比在代码中将它们嵌入 Java 字符串更可取。请注意，如果您愿意，仍可以将 JavaScript 代码作为 Java 字符串传递。

9.8.1。示例用法

要了解如何执行 Map-Reduce 操作，请使用“MongoDB - 权威指南”一书中的示例。在这个例子中，我们将创建三个文档，它们分别具有值[a，b]，[b，c]和[c，d]。每个文档中的值与键“x”相关联，如下所示。对于此示例，假设这些文档位于名为“jmr1”的集合中。

```
{ "_id" : ObjectId("4e5ff893c0277826074ec533"), "x" : [ "a", "b" ] }
{ "_id" : ObjectId("4e5ff893c0277826074ec534"), "x" : [ "b", "c" ] }
{ "_id" : ObjectId("4e5ff893c0277826074ec535"), "x" : [ "c", "d" ] }
```

下面将显示一个 map 函数，该函数将计算每个文档中数组中每个字母的出现次数

```
function () {
   for (var i = 0; i < this.x.length; i++) {
      emit(this.x[i], 1);
   }
}
```

reduce 函数将总结所有文档中每个字母的出现，如下所示

```
function (key, values) {
   var sum = 0;
   for (var i = 0; i < values.length; i++)
      sum += values[i];
   return sum;
}
```

执行此操作将导致如下所示的集合。

```
{ "_id" : "a", "value" : 1 }
{ "_id" : "b", "value" : 2 }
{ "_id" : "c", "value" : 2 }
{ "_id" : "d", "value" : 1 }
```

假设地图和减少功能位于 map.js 和 reduce.js 在你的罐子，使他们可在类路径捆绑，你可以执行地图，减少运行，如下图所示获得的结果

```
MapReduceResults<ValueObject> results = mongoOperations.mapReduce("jmr1",
"classpath:map.js", "classpath:reduce.js", ValueObject.class);
for (ValueObject valueObject : results) {
  System.out.println(valueObject);
}
```

上面代码的输出是

```
ValueObject [id=a, value=1.0]
ValueObject [id=b, value=2.0]
ValueObject [id=c, value=2.0]
ValueObject [id=d, value=1.0]
```

MapReduceResults 类实现 Iterable 并提供对原始输出的访问，以及计时和计数统计信息。这个 ValueObject 班很简单

```
public class ValueObject {

  private String id;
  private float value;

  public String getId() {
    return id;
  }

  public float getValue() {
    return value;
  }

  public void setValue(float value) {
    this.value = value;
  }

  @Override
  public String toString() {
    return "ValueObject [id=" + id + ", value=" + value + "]";
  }
}
```

默认情况下，使用 INLINE 的输出类型，因此您不必指定输出集合。要指定其他 map-reduce 选项，请使用带有附加 MapReduceOptions 参数的重载方法。该类 MapReduceOptions 具有流畅的 API，因此可以使用非常紧凑的语法添加其他选项。这是一个将输出集合设置为“jmr1_out”的示例。请注意，仅设置输出集合采用默认输出类型 REPLACE。

```
MapReduceResults<ValueObject> results = mongoOperations.mapReduce("jmr1",
"classpath:map.js", "classpath:reduce.js",
                                                new
MapReduceOptions().outputCollection("jmr1_out"), ValueObject.class);
```

还有一个静态导入 import static org.springframework.data.mongodb.core.mapreduce.MapReduceOptions.options;可用于使语法稍微紧凑

```
MapReduceResults<ValueObject> results = mongoOperations.mapReduce("jmr1",
"classpath:map.js", "classpath:reduce.js",
                                                options().outputCollection("jmr1_out"),
ValueObject.class);
```

您还可以指定查询以减少将用于提供给 map-reduce 操作的数据集。这将从考虑 map-reduce 操作中删除包含[a，b]的文档。

```
Query query = new Query(where("x").ne(new String[] { "a", "b" }));
```

```
MapReduceResults<ValueObject> results = mongoOperations.mapReduce(query,
"jmr1", "classpath:map.js", "classpath:reduce.js",
                                                options().outputCollection("jmr1_out"),
ValueObject.class);
```

请注意，您还可以在查询中指定其他限制和排序值，但不能跳过值。

9.9。脚本操作

MongoDB 允许通过直接发送脚本或调用存储脚本来在服务器上执行 JavaScript 函数。ScriptOperations 可以通过访问 MongoTemplate 并提供基本的抽象 JavaScript 使用。

9.9.1。示例用法

```
ScriptOperations scriptOps = template.scriptOps();

ExecutableMongoScript echoScript = new ExecutableMongoScript("function(x)
{ return x; }");
scriptOps.execute(echoScript, "directly execute script");     (1)

scriptOps.register(new NamedMongoScript("echo", echoScript)); (2)
scriptOps.call("echo", "execute script via name");            (3)
```

1	直接执行脚本而不在服务器端存储该功能。
2	使用'echo'作为名称存储脚本。给定名称标识脚本并允许稍后调用它。
3	使用提供的参数执行名为'echo'的脚本。

9.10。集团运营

作为使用 Map-Reduce 执行数据聚合的替代方法，您可以使用与查询样式使用 SQL 组相似的 group 操作，因此与使用 Map-Reduce 相比，它可能更容易接近。使用组操作确实有一些限制，例如它在共享环境中不受支持，并且它在单个 BSON 对象中返回完整的结果集，因此结果应该很小，小于 10,000 个键。

Spring 通过在 MongoOperations 上提供方法来简化组操作的创建和执行，从而提供与 MongoDB 组操作的集成。它可以将组操作的结果转换为 POJO，还可以与 Spring 的资源抽象抽象集成。这将允许您将 JavaScript 文件放在文件系统，类路径，http 服务器或任何其他 Spring Resource 实现上，然后通过简单的 URI 样式语法引用 JavaScript 资源，例如'classpath：reduce.js;。将 JavaScript 代码外部化为

文件，如果通常更喜欢将它们作为 Java 字符串嵌入代码中。请注意，如果您愿意，仍可以将 JavaScript 代码作为 Java 字符串传递。

9.10.1。示例用法

为了理解组操作如何工作，使用了以下示例，这有点人为。有关更现实的示例，请参阅“MongoDB - 权威指南”一书。名为 group_test_collection 使用以下行创建的集合。

```
{ "_id" : ObjectId("4ec1d25d41421e2015da64f1"), "x" : 1 }
{ "_id" : ObjectId("4ec1d25d41421e2015da64f2"), "x" : 1 }
{ "_id" : ObjectId("4ec1d25d41421e2015da64f3"), "x" : 2 }
{ "_id" : ObjectId("4ec1d25d41421e2015da64f4"), "x" : 3 }
{ "_id" : ObjectId("4ec1d25d41421e2015da64f5"), "x" : 3 }
{ "_id" : ObjectId("4ec1d25d41421e2015da64f6"), "x" : 3 }
```

我们希望按每行中的唯一字段进行分组，x 字段和聚合每个特定值 x 发生的次数。为此，我们需要创建一个包含 count 变量的初始文档，以及一个 reduce 函数，它会在每次遇到它时递增它。执行组操作的 Java 代码如下所示

```
GroupByResults<XObject> results = mongoTemplate.group("group_test_collection",
                                        GroupBy.key("x").initialDocument("{ count:
0 }").reduceFunction("function(doc, prev) { prev.count += 1 }"),
                                        XObject.class);
```

第一个参数是用于运行组操作的集合的名称，第二个参数是一个流畅的 API，它通过 GroupBy 类指定组操作的属性。在这个例子中，我们只使用 intialDocument 和 reduceFunction 方法。您还可以指定键功能，以及作为 Fluent API 的一部分的终结器。如果您有多个要分组的键，则可以传入逗号分隔的键列表。

组操作的原始结果是一个类似于此的 JSON 文档

```
{
 "retval" : [ { "x" : 1.0 , "count" : 2.0} ,
              { "x" : 2.0 , "count" : 1.0} ,
              { "x" : 3.0 , "count" : 3.0} ] ,
 "count" : 6.0 ,
 "keys" : 3 ,
 "ok" : 1.0
}
```

“retval”字段下的文档被映射到 group 方法中的第三个参数，在本例中为 XObject，如下所示。

```
public class XObject {

  private float x;
```

```
  private float count;


  public float getX() {
    return x;
  }

  public void setX(float x) {
    this.x = x;
  }

  public float getCount() {
    return count;
  }

  public void setCount(float count) {
    this.count = count;
  }

  @Override
  public String toString() {
    return "XObject [x=" + x + " count = " + count + "]";
  }
}
```

您还可以 DbObject 通过调用类 getRawResults 上的方法获取原始结果 GroupByResults。

组方法还有一个额外的方法重载 MongoOperations，允许您指定 Criteria 用于选择行子集的对象。下面显示了一个使用 Criteria 对象，使用静态导入的一些语法糖，以及通过 Spring 资源字符串引用键函数和减少函数 javascript 文件的示例。

```
import static
org.springframework.data.mongodb.core.mapreduce.GroupBy.keyFunction;
import static org.springframework.data.mongodb.core.query.Criteria.where;

GroupByResults<XObject> results = mongoTemplate.group(where("x").gt(0),
                                  "group_test_collection",

keyFunction("classpath:keyFunction.js").initialDocument("{ count:
0 }").reduceFunction("classpath:groupReduce.js"), XObject.class);
```

9.11。聚合框架支持

Spring Data MongoDB 支持在 2.2 版中引入 MongoDB 的聚合框架。

MongoDB 文档描述了聚合框架，如下所示：

有关详细信息，请参阅 MongoDB 的聚合框架和其他数据聚合工具的完整参考文档。

9.11.1。基本概念

Spring Data MongoDB 中的聚合框架支持基于以下关键抽象 Aggregation，AggregationOperation 以及 AggregationResults。

- Aggregation

  Aggregation 表示 MongoDB aggregate 操作，并保存聚合管道指令的描述。通过调用类的相应 newAggregation(…)静态工厂方法来创建聚合，该方法 Aggregation 将列表 AggregateOperation 作为可选输入类旁边的参数。

  实际的聚合操作由其 aggregate 方法执行，该方法 MongoTemplate 也将所需的输出类作为参数。

- AggregationOperation

  An AggregationOperation 表示 MongoDB 聚合管道操作，并描述了应在此聚合步骤中执行的处理。虽然可以手动创建 AggregationOperation 构建的推荐方法 AggregateOperation 是使用 Aggregate 类提供的静态工厂方法。

- AggregationResults

  AggregationResults 是聚合操作结果的容器。它 DBObject 以映射对象和执行聚合的信息的形式提供对原始聚合结果的访问。

  使用 Spring Data MongoDB 对 MongoDB 聚合框架的支持的规范示例如下所示：

```
import static org.springframework.data.mongodb.core.aggregation.Aggregation.*;

Aggregation agg = newAggregation(
   pipelineOP1(),
   pipelineOP2(),
   pipelineOPn()
);

AggregationResults<OutputType> results = mongoTemplate.aggregate(agg,
"INPUT_COLLECTION_NAME", OutputType.class);
List<OutputType> mappedResult = results.getMappedResults();
```

请注意，如果您提供的输入类作为第一个参数的 newAggregation 方法 MongoTemplate 从这个类派生的输入集合的名称。否则，如果您未指定输入类，则必须显式提供输入集合的名称。如果提供输入类和输入集合，则后者优先。

9.11.2。支持的聚合操作

MongoDB 聚合框架提供以下类型的聚合操作：

- 管道聚合运算符
- 组聚合运算符
- 布尔聚合运算符
- 比较聚合运算符
- 算术聚合运算符
- 字符串聚合运算符
- 日期聚合运算符
- 数组聚合运算符
- 条件聚合运算符
- 查找聚合运算符

在撰写本文时，我们为 Spring Data MongoDB 中的以下聚合操作提供支持。

表 4. Spring Data MongoDB 当前支持的聚合操作

管道聚合运算符	bucket，bucketAuto，count，facet，geoNear，graphLookup，group，limit，lookup，match，project，replaceRoot，skip，sort，unwind
设置聚合运算符	setEquals，setIntersection，setUnion，setDifference，setIsSubset，anyElementTrue，allElementsTrue
组聚合运算符	addToSet，first，last，max，min，avg，push，sum，（* count），stdDevPop，stdDevSamp
算术聚	abs，add（* via plus），ceil，divide，exp，floor，ln，log，log10，

合运算符	mod，multiply，pow，sqrt，subtract（* via minus），trunc
字符串聚合运算符	concat，substr，toLower，toUpper，stcasecmp，indexOfBytes，indexOfCP，split，strLenBytes，strLenCP，substrCP，
比较聚合运算符	eq（* via：is），gt，gte，lt，lte，ne
数组聚合运算符	arrayElementAt，concatArrays，filter，in，indexOfArray，isArray，range，reverseArray，reduce，size，slice，zip
文字运营商	文字
日期聚合运算符	dayOfYear，dayOfMonth，dayOfWeek，year，month，week，hour，minute，second，millisecond，dateToString，isoDayOfWeek，isoWeek，isoWeekYear
变量运算符	地图
条件聚合运算符	cond，ifNull，switch
类型聚合运算符	类型

请注意，Spring Data MongoDB 当前不支持此处未列出的聚合操作。比较聚合运算符表示为 Criteria 表达式。

*）Spring Data MongoDB 映射或添加操作。

9.11.3。投影表达

投影表达式用于定义作为特定聚合步骤的结果的字段。可以通过传递列表或聚合框架对象，通过类的 project 方法定义投影表达式。可以通过方法通过流畅的 API 使用附加字段扩展投影，并通过该方法使用别名。请注意，还可以通过聚合框架的静态工厂方法定义具有别名的字段，然后可以使用该方法构造新的 AggregationStringFieldsand(String)as(String)Fields.fieldFields 实例。在以后的聚合阶段中对投影字段的引用仅通过使用包含字段的字段名称或其别名或新定义字段的别名来有效。未包含在投影中的字段不能在以后的聚合阶段中引用。

实施例 63.投影表达实施例

```
// will generate {$project: {name: 1, netPrice: 1}}
project("name", "netPrice")

// will generate {$project: {bar: $foo}}
project().and("foo").as("bar")

// will generate {$project: {a: 1, b: 1, bar: $foo}}
project("a","b").and("foo").as("bar")
```

例 64.使用投影和排序的多阶段聚合

```
// will generate {$project: {name: 1, netPrice: 1}}, {$sort: {name: 1}}
project("name", "netPrice"), sort(ASC, "name")

// will generate {$project: {bar: $foo}}, {$sort: {bar: 1}}
project().and("foo").as("bar"), sort(ASC, "bar")

// this will not work
project().and("foo").as("bar"), sort(ASC, "foo")
```

可以在 AggregationTests 课堂上找到更多项目操作示例。请注意，有关投影表达式的更多详细信息，请参阅 MongoDB 聚合框架参考文档的相应部分。

9.11.4。分面分类

MongoDB 支持使用聚合框架进行版本 3.4 分面分类。分面分类使用一般或特定主题的语义类别，这些语义类别被组合以创建完整的分类条目。流经聚合管道的文档被分类为存储桶。多面分类允许在同一组输入文档上进行各种聚合，而无需多次检索输入文档。

水桶

存储桶操作根据指定的表达式和存储区边界将传入文档分组（称为存储桶）。存储桶操作需要分组字段或分组表达式。它们可以通过类的 bucket()/bucketAuto()方法定义 Aggregate。BucketOperation 并且 BucketAutoOperation 可以

基于输入文档的聚合表达式公开累积。可以通过 with…()方法，andOutput(String)方法和通过 as(String)方法别名的流程 API 通过附加参数扩展桶操作。每个存储桶在输出中表示为文档。

BucketOperation 采用一组定义的边界将传入的文档分组到这些类别中。边界需要进行排序。

例 65.铲斗操作实例
```
// will generate {$bucket: {groupBy: $price, boundaries: [0, 100, 400]}}
bucket("price").withBoundaries(0, 100, 400);

// will generate {$bucket: {groupBy: $price, default: "Other" boundaries: [0, 100]}}
bucket("price").withBoundaries(0, 100).withDefault("Other");

// will generate {$bucket: {groupBy: $price, boundaries: [0, 100], output: { count: { $sum: 1}}}}
bucket("price").withBoundaries(0, 100).andOutputCount().as("count");

// will generate {$bucket: {groupBy: $price, boundaries: [0, 100], 5, output: { titles: { $push: "$title"}}}
bucket("price").withBoundaries(0, 100).andOutput("title").push().as("titles");
```

BucketAutoOperation 确定边界本身，以尝试将文档均匀分布到指定数量的存储桶中。BucketAutoOperation 可选地采用粒度指定要使用的首选数字序列，以确保计算的边界边缘以首选圆整数或其幂 10 结束。

例 66.铲斗操作实例
```
// will generate {$bucketAuto: {groupBy: $price, buckets: 5}}
bucketAuto("price", 5)

// will generate {$bucketAuto: {groupBy: $price, buckets: 5, granularity: "E24"}}
bucketAuto("price", 5).withGranularity(Granularities.E24).withDefault("Other");

// will generate {$bucketAuto: {groupBy: $price, buckets: 5, output: { titles: { $push: "$title"}}}
bucketAuto("price", 5).andOutput("title").push().as("titles");
```

铲斗操作可以使用 AggregationExpression 通过 andOutput()与规划环境地政司表达式通过 andOutputExpression()创造桶输出领域。

请注意，有关存储桶表达式的更多详细信息，请参阅 MongoDB 聚合框架参考文档的$bucket 部分和 $bucketAuto 部分。

多方面聚合

可以使用多个聚合管道来创建多面聚合，这些聚合在单个聚合阶段内表征跨多个维度或构面的数据。多面聚合提供多个过滤器和分类，以指导数据浏览和分

析。分面的一个常见实现是，有多少在线零售商通过对产品价格，制造商，尺寸等应用过滤器来提供缩小搜索结果范围的方法。

A FacetOperation 可以通过类的 facet()方法定义 Aggregation。可以通过该 and()方法使用多个聚合管道进行自定义。每个子管道在输出文档中都有自己的字段，其结果存储为文档数组。

子管道可以在分组之前投影和过滤输入文档。常见的情况是在分类之前提取日期部分或计算。

例 67.方面操作示例

```
// will generate {$facet: {categorizedByPrice: [ { $match: { price: {$exists : true}}},
{ $bucketAuto: {groupBy: $price, buckets: 5}}]}}
facet(match(Criteria.where("price").exists(true)), bucketAuto("price",
5)).as("categorizedByPrice"))

// will generate {$facet: {categorizedByYear: [
//                       { $project: { title: 1, publicationYear: { $year:
"publicationDate"}}},
//                       { $bucketAuto: {groupBy: $price, buckets: 5, output: { titles:
{$push:"$title"}}}
//                     ]}}
facet(project("title").and("publicationDate").extractYear().as("publicationYear"),
     bucketAuto("publicationYear", 5).andOutput("title").push().as("titles"))
  .as("categorizedByYear"))
```

需要注意的是可以在中找到有关操作方面的进一步细节$facet 部分 MongoDB 的聚合框架参考文档。

投影表达式中的 Spring 表达式支持

我们支持通过和类的 andExpression 方法在投影表达式中使用 SpEL 表达式。这允许您将所需表达式定义为 SpEL 表达式，该表达式在查询执行时转换为相应的 MongoDB 投影表达式部分。这使得表达复杂计算变得更加容易。
ProjectionOperationBucketOperation

使用 SpEL 表达式进行复杂计算

以下 SpEL 表达式：

```
1 + (q + 1) / (q - 1)
```

将被翻译成以下投影表达式部分：

```
{ "$add" : [ 1, {
   "$divide" : [ {
      "$add":["$q", 1]}, {
```

```
        "$subtract":[ "$q", 1]}
    ]
}]}
```

在聚合框架示例 5 和聚合框架示例 6 中查看更多上下文中的示例。您可以在中找到支持的 SpEL 表达式构造的更多用法示例 SpelExpressionTransformerUnitTests。

表 5.支持的 SpEL 转换

a == b	{$ eq：[$ a，$ b]}
a！= b	{$ ne：[$ a，$ b]}
a> b	{$ gt：[$ a，$ b]}
a> = b	{$ gte：[$ a，$ b]}
a <b	{$ lt：[$ a，$ b]}
a⇐b	{$ lte：[$ a，$ b]}
a + b	{$ add：[$ a，$ b]}
a - b	{$减：[$ a，$ b]}
a * b	{$ multiply：[$ a，$ b]}
a / b	{$ divide：[$ a，$ b]}
—^ B	{$ pow：[$ a，$ b]}
a％b	{$ mod：[$ a，$ b]}

a && b	{$和：[$ a，$ b]}
a || b	{$或：[$ a，$ b]}
！一个	{$ not：[$ a]}

在[支持的 SpEL 转换]中显示的转换旁边，可以使用标准的 SpEL 操作 new，例如。通过名称创建数组和引用表达式，然后在括号中使用参数。

```
// { $setEquals : [$a, [5, 8, 13] ] }
.andExpression("setEquals(a, new int[]{5, 8, 13})");
```

聚合框架示例

以下示例演示了使用 Spring Data MongoDB 的 MongoDB 聚合框架的使用模式。

聚合框架示例 1

在这个介绍性示例中，我们希望聚合一个标记列表，以获取 MongoDB 集合中特定标记的出现次数，该集合"tags"按出现次数降序排序。此示例演示了分组，排序，投影（选择）和展开（结果拆分）的用法。

```
class TagCount {
 String tag;
 int n;
}
import static org.springframework.data.mongodb.core.aggregation.Aggregation.*;

Aggregation agg = newAggregation(
    project("tags"),
    unwind("tags"),
    group("tags").count().as("n"),
    project("n").and("tag").previousOperation(),
    sort(DESC, "n")
);

AggregationResults<TagCount> results = mongoTemplate.aggregate(agg, "tags",
TagCount.class);
List<TagCount> tagCount = results.getMappedResults();
```

- 为了做到这一点，我们首先通过 newAggregation 静态工厂方法创建一个新的聚合，我们传递一个聚合操作列表。这些聚合操作定义了我们的聚合管道 Aggregation。

- 作为第二步，我们"tags"使用 project 操作从输入集合中选择字段（这是一个字符串数组）。
- 在第三步中，我们使用该 unwind 操作为"tags"数组中的每个标记生成一个新文档。
- 在第四步中，我们使用 group 操作为每个"tags"值定义一个组，我们通过 count 聚合运算符聚合发生计数，并在一个名为的新字段中收集结果"n"。
- 作为第五步，我们选择字段"n"并为前一个组操作（因此调用 previousOperation()）生成的 id 字段创建一个别名，其名称为"tag"。
- 作为第六步，我们通过 sort 操作按降序排列生成计数的结果列表。
- 最后，我们 aggregate 在 MongoTemplate 上调用 Method，以便让 MongoDB 使用创建 Aggregation 的参数执行实际的聚合操作。

请注意，输入集合被明确指定为 Method 的"tags"参数 aggregate。如果未明确指定输入集合的名称，则它将从作为第一个参数传递给 newAggreationMethod 的输入类派生。

聚合框架示例 2

此示例基于 MongoDB 聚合框架文档中的“ 最大和最小城市”示例。我们添加了额外的排序以使用不同的 MongoDB 版本生成稳定的结果。在这里，我们希望使用聚合框架按每个州的人口返回最小和最大的城市。此示例演示了分组，排序和投影（选择）的用法。

```
class ZipInfo {
  String id;
  String city;
  String state;
  @Field("pop") int population;
  @Field("loc") double[] location;
}

class City {
  String name;
  int population;
}

class ZipInfoStats {
  String id;
  String state;
  City biggestCity;
  City smallestCity;
}
```

```
import static org.springframework.data.mongodb.core.aggregation.Aggregation.*;

TypedAggregation<ZipInfo> aggregation = newAggregation(ZipInfo.class,
  group("state", "city")
    .sum("population").as("pop"),
  sort(ASC, "pop", "state", "city"),
  group("state")
    .last("city").as("biggestCity")
    .last("pop").as("biggestPop")
    .first("city").as("smallestCity")
    .first("pop").as("smallestPop"),
  project()
    .and("state").previousOperation()
    .and("biggestCity")
      .nested(bind("name", "biggestCity").and("population", "biggestPop"))
    .and("smallestCity")
      .nested(bind("name", "smallestCity").and("population", "smallestPop")),
  sort(ASC, "state")
);

AggregationResults<ZipInfoStats> result = mongoTemplate.aggregate(aggregation,
ZipInfoStats.class);
ZipInfoStats firstZipInfoStats = result.getMappedResults().get(0);
```

- 该类 ZipInfo 映射给定输入集合的结构。该类 ZipInfoStats 以所需的输出格式定义结构。

- 作为第一步，我们使用 group 操作从输入集合中定义组。分组标准是字段的组合，"state"并"city"形成组的 id 结构。我们"population"通过使用 sum 运算符将结果保存在字段中来聚合分组元素中的属性值"pop"。

- 在第二步中，我们使用 sort 操作按字段对中间结果进行排序"pop"，"state"并按"city"升序排序，使最小的城市位于顶部，最大的城市位于结果的底部。请注意，对 Spring Data MongoDB 负责的组 ID 字段进行排序"state"和"city"隐式执行。

- 在第三步中，我们 group 再次使用操作将中间结果分组"state"。请注意，"state"再次隐式引用 group-id 字段。我们分别通过操作选择 last(…)和 first(…)操作员来选择最大和最小城市的名称和人口数 project。

- 作为第四步，我们"state"从上一个 group 操作中选择字段。请注意，"state"再次隐式引用 group-id 字段。由于我们不希望出现隐式生成的 id，因此我们从前一个操作中排除了 id and(previousOperation()).exclude()。因为我们想 City 在输出类中填充嵌套结构，所以我们必须使用嵌套方法发出适当的子文档。

- 最后，作为第五步，我们 StateStats 通过 sort 操作按升序对其结果列表进行排序。

请注意，我们从 ZipInfo 作为第一个参数传递给 newAggregation-Method 的-class 派生输入集合的名称。

聚合框架示例 3

此示例基于 MongoDB 聚合框架文档中的人口超过 1000 万的示例。我们添加了额外的排序以使用不同的 MongoDB 版本生成稳定的结果。在这里，我们希望使用聚合框架返回人口超过 1000 万的所有州。此示例演示了分组，排序和匹配（过滤）的用法。

```
class StateStats {
  @Id String id;
  String state;
  @Field("totalPop") int totalPopulation;
}
import static org.springframework.data.mongodb.core.aggregation.Aggregation.*;

TypedAggregation<ZipInfo> agg = newAggregation(ZipInfo.class,
   group("state").sum("population").as("totalPop"),
   sort(ASC, previousOperation(), "totalPop"),
   match(where("totalPop").gte(10 * 1000 * 1000))
);

AggregationResults<StateStats> result = mongoTemplate.aggregate(agg,
StateStats.class);
List<StateStats> stateStatsList = result.getMappedResults();
```

- 作为第一步，我们按"state"字段对输入集合进行分组，并计算字段的总和"population"并将结果存储在新字段中"totalPop"。
- 在第二步中，除了"totalPop"字段的升序之外，我们还通过前一组操作的 id-reference 对中间结果进行排序。
- 最后，在第三步中，我们使用 match 接受 Criteria 查询作为参数的操作来过滤中间结果。

请注意，我们从 ZipInfo 作为第一个参数传递给 newAggregation-Method 的-class 派生输入集合的名称。

聚合框架示例 4

此示例演示了在投影操作中使用简单算术运算。

```
class Product {
   String id;
   String name;
   double netPrice;
```

```
   int spaceUnits;
}
import static org.springframework.data.mongodb.core.aggregation.Aggregation.*;

TypedAggregation<Product> agg = newAggregation(Product.class,
   project("name", "netPrice")
      .and("netPrice").plus(1).as("netPricePlus1")
      .and("netPrice").minus(1).as("netPriceMinus1")
      .and("netPrice").multiply(1.19).as("grossPrice")
      .and("netPrice").divide(2).as("netPriceDiv2")
      .and("spaceUnits").mod(2).as("spaceUnitsMod2")
);

AggregationResults<DBObject> result = mongoTemplate.aggregate(agg,
DBObject.class);
List<DBObject> resultList = result.getMappedResults();
```

请注意，我们从 Product 作为第一个参数传递给 newAggregation-Method 的-class 派生输入集合的名称。

聚合框架示例 5

此示例演示了在投影操作中使用从 SpEL 表达式派生的简单算术运算。

```
class Product {
   String id;
   String name;
   double netPrice;
   int spaceUnits;
}
import static org.springframework.data.mongodb.core.aggregation.Aggregation.*;

TypedAggregation<Product> agg = newAggregation(Product.class,
   project("name", "netPrice")
      .andExpression("netPrice + 1").as("netPricePlus1")
      .andExpression("netPrice - 1").as("netPriceMinus1")
      .andExpression("netPrice / 2").as("netPriceDiv2")
      .andExpression("netPrice * 1.19").as("grossPrice")
      .andExpression("spaceUnits % 2").as("spaceUnitsMod2")
      .andExpression("(netPrice * 0.8  + 1.2) *
1.19").as("grossPriceIncludingDiscountAndCharge")

);

AggregationResults<DBObject> result = mongoTemplate.aggregate(agg,
DBObject.class);
List<DBObject> resultList = result.getMappedResults();
```

聚合框架示例 6

此示例演示了在投影操作中使用从 SpEL 表达式派生的复杂算术运算。

注意：传递给 addExpressionMethod 的附加参数可以通过索引器表达式根据其位置引用。在这个例子中，我们引用参数，它是参数数组 via 的第一个参数[0]。当 SpEL 表达式转换为 MongoDB 聚合框架表达式时，外部参数表达式将替换为各自的值。

```
class Product {
   String id;
   String name;
   double netPrice;
   int spaceUnits;
}
import static org.springframework.data.mongodb.core.aggregation.Aggregation.*;

double shippingCosts = 1.2;

TypedAggregation<Product> agg = newAggregation(Product.class,
   project("name", "netPrice")
       .andExpression("(netPrice * (1-discountRate)  + [0]) * (1+taxRate)",
shippingCosts).as("salesPrice")
);

AggregationResults<DBObject> result = mongoTemplate.aggregate(agg,
DBObject.class);
List<DBObject> resultList = result.getMappedResults();
```

请注意，我们还可以在 SpEL 表达式中引用文档的其他字段。

聚合框架示例 7

此示例使用条件投影。它来自$ cond 参考文档。

```
public class InventoryItem {

  @Id int id;
  String item;
  String description;
  int qty;
}

public class InventoryItemProjection {

  @Id int id;
  String item;
  String description;
  int qty;
  int discount
}
```

```
import static org.springframework.data.mongodb.core.aggregation.Aggregation.*;

TypedAggregation<InventoryItem> agg = newAggregation(InventoryItem.class,
  project("item").and("discount")
    .applyCondition(ConditionalOperator.newBuilder().when(Criteria.where("qty").gte
(250))
      .then(30)
      .otherwise(20))
    .and(ifNull("description", "Unspecified")).as("description")
);

AggregationResults<InventoryItemProjection> result =
mongoTemplate.aggregate(agg, "inventory", InventoryItemProjection.class);
List<InventoryItemProjection> stateStatsList = result.getMappedResults();
```

- 此一步聚合使用与 inventory 集合的投影操作。我们 discount 使用条件操作为具有 qty 大于或等于的所有库存项目投影该字段 250。对该 description 字段执行第二条件投影。我们将描述 Unspecified 应用于所有没有具有描述 description 的项目字段的项目 null。

9.12。使用自定义转换器覆盖默认映射

为了对映射过程进行更细粒度的控制，您可以使用 MongoConverter 诸如之类的实现来注册 Spring 转换器 MappingMongoConverter。

MappingMongoConverter 在尝试映射对象本身之前，检查是否有任何可以处理特定类的 Spring 转换器。要“劫持”正常的映射策略 MappingMongoConverter，可能是为了提高性能或其他自定义映射需求，首先需要创建 Spring Converter 接口的实现，然后将其注册到 MappingConverter。

> 有关 Spring 类型转换服务的更多信息，请参阅此处的参考文档。

9.12.1。使用已注册的 Spring Converter 进行保存

Converter 从 Person 对象转换为 a 的示例实现 com.mongodb.DBObject 如下所示

```
import org.springframework.core.convert.converter.Converter;

import com.mongodb.BasicDBObject;
import com.mongodb.DBObject;

public class PersonWriteConverter implements Converter<Person, DBObject> {

  public DBObject convert(Person source) {
    DBObject dbo = new BasicDBObject();
```

```
    dbo.put("_id", source.getId());
    dbo.put("name", source.getFirstName());
    dbo.put("age", source.getAge());
    return dbo;
  }
}
```

9.12.2。使用 Spring Converter 阅读

从 DBObject 转换为 Person 对象的 Converter 的示例实现如下所示。

```
public class PersonReadConverter implements Converter<DBObject, Person> {

  public Person convert(DBObject source) {
    Person p = new Person((ObjectId) source.get("_id"), (String) source.get("name"));
    p.setAge((Integer) source.get("age"));
    return p;
  }
}
```

9.12.3。使用 MongoConverter 注册 Spring 转换器

蒙戈春天命名空间提供了便利的方式来春登记 Converters 的了 MappingMongoConverter。下面的配置片段显示了如何手动注册转换器 bean 以及如何将包装配置 MappingMongoConverter 为 MongoTemplate。

```
<mongo:db-factory dbname="database"/>

<mongo:mapping-converter>
  <mongo:custom-converters>
    <mongo:converter ref="readConverter"/>
    <mongo:converter>
      <bean class="org.springframework.data.mongodb.test.PersonWriteConverter"/>
    </mongo:converter>
  </mongo:custom-converters>
</mongo:mapping-converter>

<bean id="readConverter"
class="org.springframework.data.mongodb.test.PersonReadConverter"/>

<bean id="mongoTemplate"
class="org.springframework.data.mongodb.core.MongoTemplate">
  <constructor-arg name="mongoDbFactory" ref="mongoDbFactory"/>
  <constructor-arg name="mongoConverter" ref="mappingConverter"/>
</bean>
```

您还可以使用 custom-converters 元素的 base-package 属性为给定包下的所有实现 Converter 和 GenericConverter 实现启用类路径扫描。

```
<mongo:mapping-converter>
  <mongo:custom-converters base-package="com.acme.**.converters" />
</mongo:mapping-converter>
```

9.12.4。转换器消除歧义

通常，我们会检查 Converter 它们转换的源类型和目标类型的实现。根据其中一个是 MongoDB 本身可以处理的类型，我们将转换器实例注册为读取或写入。看看以下样本：

```
// Write converter as only the target type is one Mongo can handle natively
class MyConverter implements Converter<Person, String> { … }

// Read converter as only the source type is one Mongo can handle natively
class MyConverter implements Converter<String, Person> { … }
```

如果您编写 Converter 的源和目标类型是本机 Mongo 类型，我们无法确定是否应将其视为读取或写入转换器。将转换器实例注册为两者可能会导致不必要的结果。例如 a Converter<String, Long>是不明确的，尽管在编写时尝试将所有 String 实例转换为 Long 实例可能没有意义。通常能够强制基础设施以单向方式注册转换器，我们提供@ReadingConverter 以及@WritingConverter 在转换器实现中使用。

9.13。索引和收集管理

MongoTemplate 提供了一些管理索引和集合的方法。这些被收集到一个名为的辅助接口中 IndexOperations。您可以通过调用方法来访问这些操作，indexOps 并传入集合名称或 java.lang.Class 实体（集合名称将通过名称或注释元数据从.class 派生）。

的 IndexOperations 界面如下图所示

```
public interface IndexOperations {

  void ensureIndex(IndexDefinition indexDefinition);

  void dropIndex(String name);

  void dropAllIndexes();

  void resetIndexCache();

  List<IndexInfo> getIndexInfo();
}
```

9.13.1。创建索引的方法

我们可以在集合上创建索引以提高查询性能。

使用 MongoTemplate 创建索引

```
mongoTemplate.indexOps(Person.class).ensureIndex(new
Index().on("name",Order.ASCENDING));
```

- **ensureIndex** 确保集合中存在所提供的 IndexDefinition 的索引。

您可以使用类标准，地理空间和文本索引 IndexDefinition，GeoSpatialIndex 和 TextIndexDefinition。例如，给定上一节中定义的 Venue 类，您将声明一个地理空间查询，如下所示。

```
mongoTemplate.indexOps(Venue.class).ensureIndex(new
GeospatialIndex("location"));
```

9.13.2。访问索引信息

IndexOperations 接口具有 getIndexInfo 方法，该方法返回 IndexInfo 对象的列表。它包含集合上定义的所有索引。下面是一个在 Person 类上定义具有 age 属性的索引的示例。

```
template.indexOps(Person.class).ensureIndex(new Index().on("age",
Order.DESCENDING).unique(Duplicates.DROP));

List<IndexInfo> indexInfoList = template.indexOps(Person.class).getIndexInfo();

// Contains
// [IndexInfo [fieldSpec={_id=ASCENDING}, name=_id_, unique=false,
dropDuplicates=false, sparse=false],
//  IndexInfo [fieldSpec={age=DESCENDING}, name=age_-1, unique=true,
dropDuplicates=true, sparse=false]]
```

9.13.3。使用 Collection 的方法

现在是时候看一些代码示例，展示如何使用 MongoTemplate。首先我们来看看创建我们的第一个系列。

例子 68.使用 MongoTemplate 处理集合

```
DBCollection collection = null;
if (!mongoTemplate.getCollectionNames().contains("MyNewCollection")) {
   collection = mongoTemplate.createCollection("MyNewCollection");
}

mongoTemplate.dropCollection("MyNewCollection");
```

- **getCollectionNames** 返回一组集合名称。

- **collectionExists** 检查是否存在具有给定名称的集合。

- **createCollection** 创建一个无上限的集合
- **dropCollection** 删除集合
- **getCollection** 按名称获取集合，如果它不存在则创建它。

9.14。执行命令

您还可以 DB.command( )使用 executeCommand(…)方法获取 MongoDB 驱动程序的方法 MongoTemplate。这些也将执行异常转换到 Spring 的 DataAccessException 层次结构。

9.14.1。执行命令的方法

- CommandResult **executeCommand** (DBObject command)执行 MongoDB 命令。
- CommandResult **executeCommand** (String jsonCommand)执行表示为 JSON 字符串的 MongoDB 命令。

9.15。生命周期事件

MongoDB 映射框架内置了几个 org.springframework.context.ApplicationEvent 事件，您的应用程序可以通过在其中注册特殊 bean 来响应 ApplicationContext。通过基于 Spring 的 ApplicationContext 事件基础结构，这使得其他产品（如 Spring Integration）可以轻松接收这些事件，因为它们是基于 Spring 的应用程序中众所周知的事件机制。

要在对象经历转换过程（将您的域对象转换为 a com.mongodb.DBObject）之前拦截它，您将注册一个子类 AbstractMongoEventListener 来覆盖该 onBeforeConvert 方法。调度事件时，将调用侦听器并在进入转换器之前传递域对象。

```
public class BeforeConvertListener extends AbstractMongoEventListener<Person> {
  @Override
  public void onBeforeConvert(BeforeConvertEvent<Person> event) {
    ... does some auditing manipulation, set timestamps, whatever ...
  }
}
```

要在对象进入数据库之前拦截它，您需要注册一个子类 org.springframework.data.mongodb.core.mapping.event.AbstractMongoEventListener 来覆盖该 onBeforeSave 方法。调度事件时，将调用侦听器并传递域对象和转换后的对象 com.mongodb.DBObject。

```
public class BeforeSaveListener extends AbstractMongoEventListener<Person> {
  @Override
```

```
  public void onBeforeSave(BeforeSaveEvent<Person> event) {
    … change values, delete them, whatever …
  }
}
```

只需在 Spring ApplicationContext 中声明这些 bean，就会在调度事件时调用它们。

AbstractMappingEventListener 中存在的回调方法列表是

- onBeforeConvert - 在使用 MongoConveter 将对象转换为 DBObject 之前调用 MongoTemplate insert，insertList 和 save 操作。
- onBeforeSave- 在数据库中插入/保存 DBObject **之前**调用 MongoTemplate insert，insertList 和 save 操作。
- onAfterSave- 在数据库中插入/保存 DBObject **后**调用 MongoTemplate insert，insertList 和 save 操作。
- onAfterLoad - 在从数据库中检索 DBObject 之后调用 MongoTemplate find，findAndRemove，findOne 和 getCollection 方法。
- onAfterConvert - 在从数据库检索到的 DBObject 转换为 POJO 之后，在 MongoTemplate 中调用 find，findAndRemove，findOne 和 getCollection 方法。

> 生命周期事件仅针对根级别类型发出。在文档根目录中用作属性的复杂类型不是事件发布的主题，除非它们是带有注释的文档引用@DBRef。

9.16。例外翻译

Spring 框架为各种数据库和映射技术提供异常转换。传统上这适用于 JDBC 和 JPA。MongoDB 的 Spring 支持通过提供 org.springframework.dao.support.PersistenceExceptionTranslator 接口的实现将此功能扩展到 MongoDB 数据库。

映射到 Spring 的一致数据访问异常层次结构背后的动机是，您可以编写可移植和描述性的异常处理代码，而无需对 MongoDB 错误代码进行编码。Spring 的所有数据访问异常都是从根 DataAccessException 类继承的，因此您可以确保在一个 try-catch 块中捕获所有与数据库相关的异常。请注意，并非 MongoDB 驱动程序抛出的所有异常都从 MongoException 类继承。保留内部异常和消息，因此不会丢失任何信息。

执行的一些映射 MongoExceptionTranslator 是：com.mongodb.Network 到 DataAccessResourceFailureException 和 MongoException 错误代码

1003,12001,12010,12011,12012 到 InvalidDataAccessApiUsageException。查看实现以获取有关映射的更多详细信息。

9.17。执行回调

所有 Spring 模板类的一个常见设计特性是所有功能都被路由到其中一个模板执行回调方法。这有助于确保可能需要的异常和任何资源管理都执行一致性。虽然对于 JDBC 和 JMS 而言，这比 MongoDB 需要更多，但它仍然提供了一个用于异常转换和日志记录的单点。因此，使用这些执行回调是访问 MongoDB 驱动程序 DB 和 DBCollection 对象的首选方法，以执行未作为方法公开的非常见操作 MongoTemplate。

以下是执行回调方法的列表。

- <T> T **execute** (Class<?> entityClass, CollectionCallback<T> action)为指定类的实体集合执行给定的 CollectionCallback。

- <T> T **execute** (String collectionName, CollectionCallback<T> action)对给定名称的集合执行给定的 CollectionCallback。

- <T> T **execute** (DbCallback<T> action) Spring Data MongoDB provides support for the Aggregation Framework introduced to MongoDB in version 2.2.执行 DbCallback，根据需要转换任何异常。

- <T> T **execute** (String collectionName, DbCallback<T> action)对给定名称的集合执行 DbCallback，根据需要转换任何异常。

- <T> T **executeInSession** (DbCallback<T> action)在与数据库相同的连接中执行给定的 DbCallback，以确保在写入密集环境中的一致性，您可以在其中读取您编写的数据。

这是一个使用 CollectionCallback 返回索引信息的示例

```
boolean hasIndex = template.execute("geolocation", new
CollectionCallbackBoolean>() {
  public Boolean doInCollection(Venue.class, DBCollection collection) throws
MongoException, DataAccessException {
    List<DBObject> indexes = collection.getIndexInfo();
    for (DBObject dbo : indexes) {
      if ("location_2d".equals(dbo.get("name"))) {
        return true;
      }
    }
    return false;
  }
});
```

9.18。GridFS 支持

MongoDB 支持在其文件系统 GridFS 中存储二进制文件。Spring Data MongoDB 提供了一个 GridFsOperations 接口以及相应的实现，GridFsTemplate 可以轻松地与文件系统进行交互。你可以设置一个 GridFsTemplate 实例被交给它 MongoDbFactory 还有一个 MongoConverter：

例子 69. GridFsTemplate 的 JavaConfig 设置

```
class GridFsConfiguration extends AbstractMongoConfiguration {

  // … further configuration omitted

  @Bean
  public GridFsTemplate gridFsTemplate() {
    return new GridFsTemplate(mongoDbFactory(), mappingMongoConverter());
  }
}
```

相应的 XML 配置如下所示：

例子 70. GridFsTemplate 的 XML 配置

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

  <mongo:db-factory id="mongoDbFactory" dbname="database" />
  <mongo:mapping-converter id="converter" />

  <bean class="org.springframework.data.mongodb.gridfs.GridFsTemplate">
    <constructor-arg ref="mongoDbFactory" />
    <constructor-arg ref="converter" />
  </bean>

</beans>
```

现在可以注入模板并用于执行存储和检索操作。

例 71.使用 GridFsTemplate 存储文件

```
class GridFsClient {

  @Autowired
  GridFsOperations operations;
```

```
  @Test
  public void storeFileToGridFs {

    FileMetadata metadata = new FileMetadata();
    // populate metadata
    Resource file = … // lookup File or Resource

    operations.store(file.getInputStream(), "filename.txt", metadata);
  }
}
```

的 store(…)操作采取 InputStream，一个关于文件来存储文件名和任选的元数据信息。元数据可以是任意对象，它将由 MongoConverter 配置的编组 GridFsTemplate。或者您也可以提供一个 DBObject。

从文件系统读取文件可以通过 find(…)或 getResources(…)方法实现。我们先来看看这些 find(…)方法。您可以找到匹配一个 Query 或多个文件的单个文件。为了轻松定义文件查询，我们提供了 GridFsCriteria 帮助程序类。它提供静态工厂方法来封装默认元数据字段（例如 whereFilename()，whereContentType()）或定制的一个通 whereMetaData()。

例子 72.使用 GridFsTemplate 查询文件

```
class GridFsClient {

  @Autowired
  GridFsOperations operations;

  @Test
  public void findFilesInGridFs {
    List<GridFSDBFile> result =
operations.find(query(whereFilename().is("filename.txt")))
  }
}
```

> Currently MongoDB does not support defining sort criteria when retrieving files from GridFS. Thus any sort criteria defined on the Query instance handed into the find(…) method will be disregarded.

The other option to read files from the GridFs is using the methods introduced by the ResourcePatternResolver interface. They allow handing an Ant path into the method ar thus retrieve files matching the given pattern.

Example 73. Using GridFsTemplate to read files

```
class GridFsClient {

  @Autowired
  GridFsOperations operations;
```

```
  @Test
  public void readFilesFromGridFs {
    GridFsResources[] txtFiles = operations.getResources("*.txt");
  }
}
```

GridFsOperations extending ResourcePatternResolver allows the GridFsTemplate e.g. to be plugged into an ApplicationContext to read Spring Config files from a MongoDB.

10. MongoDB repositories

10.1. Introduction

This chapter will point out the specialties for repository support for MongoDB. This builds on the core repository support explained in Working with Spring Data Repositories. So make sure you've got a sound understanding of the basic concepts explained there.

10.2. Usage

To access domain entities stored in a MongoDB you can leverage our sophisticated repository support that eases implementing those quite significantly. To do so, simply create an interface for your repository:

Example 74. Sample Person entity

```
public class Person {

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

  // … getters and setters omitted
}
```

我们这里有一个非常简单的域对象。请注意，它具有名为 idtype 的属性 ObjectId。MongoTemplate（支持存储库支持）中使用的默认序列化机制将名为 id 的属性视为文档 ID。目前我们支持 String，ObjectId 并 BigInteger 作为 id 类型。

示例 75.用于持久化 Person 实体的基本存储库接口

```
public interface PersonRepository extends PagingAndSortingRepository<Person,
Long> {

  // additional custom finder methods go here
}
```

现在这个界面只是用于打字目的，但我们稍后会添加其他方法。在 Spring 配置中添加

例 76.一般的 MongoDB 存储库 Spring 配置

```
<?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:mongo="http://www.springframework.org/schema/data/mongo"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/data/mongo
    http://www.springframework.org/schema/data/mongo/spring-mongo-1.0.xsd">

  <mongo:mongo id="mongo" />

  <bean id="mongoTemplate"
class="org.springframework.data.mongodb.core.MongoTemplate">
    <constructor-arg ref="mongo" />
    <constructor-arg value="databaseName" />
  </bean>

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

</beans>
```

此命名空间元素将导致扫描基础包以查找扩展的接口，MongoRepository 并为找到的每个创建 Spring bean。默认情况下，存储库将获得一个 MongoTemplate 被调用的 Spring bean 连接 mongoTemplate，因此 mongo-template-ref 如果您偏离此约定，则只需要显式配置。

如果您更愿意使用 JavaConfig，请使用@EnableMongoRepositories 注释。注释带有与命名空间元素完全相同的属性。如果未配置基本软件包，则基础结构将扫描带注释的配置类的软件包。

例 77.存储库的 JavaConfig

```
@Configuration
@EnableMongoRepositories
class ApplicationConfig extends AbstractMongoConfiguration {

  @Override
  protected String getDatabaseName() {
    return "e-store";
  }

  @Override
  public Mongo mongo() throws Exception {
    return new Mongo();
  }
```

```
  @Override
  protected String getMappingBasePackage() {
    return "com.oreilly.springdata.mongodb"
  }
}
```

随着我们的域存储库的扩展，PagingAndSortingRepository 它为您提供了 CRUD 操作以及对实体进行分页和排序访问的方法。使用存储库实例只是将其注入客户端的依赖性问题。因此，访问 Person 页面大小为 10 的 s 的第二页看起来就像这样：

示例 78.对 Person 实体的分页访问

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class PersonRepositoryTests {

   @Autowired PersonRepository repository;

   @Test
   public void readsFirstPageCorrectly() {

     Page<Person> persons = repository.findAll(new PageRequest(0, 10));
     assertThat(persons.isFirstPage(), is(true));
   }
}
```

该示例使用 Spring 的单元测试支持创建应用程序上下文，该支持将执行基于注释的依赖注入到测试用例中。在测试方法中，我们只需使用存储库来查询数据存储区。我们向存储库提供一个 PageRequest 请求页面大小为 10 的第一页人员的实例。

10.3。查询方法

您通常在存储库上触发的大多数数据访问操作都会导致对 MongoDB 数据库执行查询。定义这样的查询只是在存储库接口上声明一个方法

例子 79.带有查询方法的 PersonRepository

```
public interface PersonRepository extends PagingAndSortingRepository<Person,
String> {

   List<Person> findByLastname(String lastname);                    (1)

   Page<Person> findByFirstname(String firstname, Pageable pageable); (2)

   Person findByShippingAddresses(Address address);                 (3)

   Stream<Person> findAllBy();                                      (4)
```

}

1	该方法显示具有给定姓氏的所有人的查询。将派生查询解析可以与 And 和连接的约束的方法名称 Or。因此，方法名称将导致查询表达式为 {"lastname" : lastname}。
2	将分页应用于查询。只需为方法签名配备一个 Pageable 参数，让方法返回一个 Page 实例，我们将自动相应地分页查询。
3	显示您可以基于非基本类型的属性进行查询。
4	使用 Java 8 Stream，它在迭代流时读取和转换单个元素。

请注意，对于 1.0 版，我们目前不支持引用 DBRef 在域类中映射的参数。

表 6.查询方法支持的关键字

关键词	样品	逻辑结果
After	findByBirthdateAfter(Date date)	{"birthdate" : {"$gt" : date}}
GreaterThan	findByAgeGreaterThan(int age)	{"age" : {"$gt" : age}}
GreaterThanEqual	findByAgeGreaterThanEqual(int age)	{"age" : {"$gte" : age}}
Before	findByBirthdateBefore(Date date)	{"birthdate" : {"$lt" : date}}
LessThan	findByAgeLessThan(int age)	{"age" : {"$lt" : age}}

表 6.查询方法支持的关键字

关键词	样品	逻辑结果
LessThanEqual	findByAgeLessThanEqual(int age)	{"age" : {"$lte" : age}}
Between	findByAgeBetween(int from, int to)	{"age" : {"$gt" : from, "$lt" : to}}
In	findByAgeIn(Collection ages)	{"age" : {"$in" : [ages…]}}
NotIn	findByAgeNotIn(Collection ages)	{"age" : {"$nin" : [ages…]}}
IsNotNull，NotNull	findByFirstnameNotNull()	{"firstname" : {"$ne" : null}}
IsNull， Null	findByFirstnameNull()	{"firstname" : null}
Like，StartingWith，EndingWith	findByFirstnameLike(String name)	{"firstname" : name} (name as regex)
NotLike，IsNotLike	findByFirstnameNotLike(String name)	{"firstname" : { "$not" : name }} (name as regex)
Containing 在字符串上	findByFirstnameContaining(String name)	{"firstname" : name} (name as regex)
NotContaining 在字符串上	findByFirstnameNotContaining(String name)	{"firstname" : { "$not" : name}}

表 6.查询方法支持的关键字

关键词	样品	逻辑结果
		(name as regex)
Containing 关于收藏	findByAddressesContaining(Address address)	{"addresses" : { "$in" : address}}
NotContaining 关于收藏	findByAddressesNotContaining(Address address)	{"addresses" : { "$not" : { "$in" : address}}}
Regex	findByFirstnameRegex(String firstname)	{"firstname" : {"$regex" : firstname }}
(No keyword)	findByFirstname(String name)	{"firstname" : name}
Not	findByFirstnameNot(String name)	{"firstname" : {"$ne" : name}}
Near	findByLocationNear(Point point)	{"location" : {"$near" : [x,y]}}
Near	findByLocationNear(Point point, Distance max)	{"location" : {"$near" : [x,y], "$maxDistance" : max}}
Near	findByLocationNear(Point point, Distance min, Distance max)	{"location" : {"$near" : [x,y], "$minDistance" : min, "$maxDistance" : max}}

表 6.查询方法支持的关键字

关键词	样品	逻辑结果
Within	findByLocationWithin(Circle circle)	{"location" : {"$geoWithin" : {"$center" : [[x, y], distance]}}}
Within	findByLocationWithin(Box box)	{"location" : {"$geoWithin" : {"$box" : [[x1, y1], x2, y2]}}}
IsTrue， True	findByActiveIsTrue()	{"active" : true}
IsFalse， False	findByActiveIsFalse()	{"active" : false}
Exists	findByLocationExists(boolean exists)	{"location" : {"$exists" : exists }}

10.3.1。存储库删除查询

上述关键字可以与删除匹配文档一起使用 delete…By 或 remove…By 创建查询。

例 80. Delete…By 查询

```
public interface PersonRepository extends MongoRepository<Person, String> {

  List <Person> deleteByLastname(String lastname);

  Long deletePersonByLastname(String lastname);
}
```

使用返回类型 List 将在实际删除之前检索并返回所有匹配的文档。数字返回类型直接删除匹配的文档，返回删除的文档总数。

10.3.2。地理空间存储库查询

正如您刚才看到的，有几个关键字在 MongoDB 查询中触发地理空间操作。的 Near 关键字允许一些进一步修改。我们来看看一些例子：

例子 81.高级 Near 查询

```
public interface PersonRepository extends MongoRepository<Person, String>

  // { 'location' : { '$near' : [point.x, point.y], '$maxDistance' : distance}}
  List<Person> findByLocationNear(Point location, Distance distance);
}
```

向 Distance 查询方法添加参数允许将结果限制为给定距离内的结果。如果 Distance 设置包含一个 Metric 我们将透明使用$nearSphere 而不是$ code。

实施例 82.使用 Distance 与 Metrics

```
Point point = new Point(43.7, 48.8);
Distance distance = new Distance(200, Metrics.KILOMETERS);
… = repository.findByLocationNear(point, distance);
// {'location' : {'$nearSphere' : [43.7, 48.8], '$maxDistance' : 0.03135711885774796}}
```

正如您所看到的那样，使用 Distance 配备的 Metriccause $nearSphere 子句而不是普通的子句$near。除此之外，实际距离根据 Metrics 使用情况计算得出。

> 使用@GeoSpatialIndexed(type = GeoSpatialIndexType.GEO_2DSPHERE)上的目标属性力量的使用$nearSphere 操作。

地理附近查询

```
public interface PersonRepository extends MongoRepository<Person, String>

  // {'geoNear' : 'location', 'near' : [x, y] }
  GeoResults<Person> findByLocationNear(Point location);

  // No metric: {'geoNear' : 'person', 'near' : [x, y], maxDistance : distance }
  // Metric: {'geoNear' : 'person', 'near' : [x, y], 'maxDistance' : distance,
  //          'distanceMultiplier' : metric.multiplier, 'spherical' : true }
  GeoResults<Person> findByLocationNear(Point location, Distance distance);

  // Metric: {'geoNear' : 'person', 'near' : [x, y], 'minDistance' : min,
  //          'maxDistance' : max, 'distanceMultiplier' : metric.multiplier,
  //          'spherical' : true }
  GeoResults<Person> findByLocationNear(Point location, Distance min, Distance
max);

  // {'geoNear' : 'location', 'near' : [x, y] }
  GeoResults<Person> findByLocationNear(Point location);
}
```

10.3.3。基于 MongoDB JSON 的查询方法和字段限制

通过添加注释 org.springframework.data.mongodb.repository.Query 存储库查找器方法，您可以指定要使用的 MongoDB JSON 查询字符串，而不是从方法名称派生查询。例如

```
public interface PersonRepository extends MongoRepository<Person, String>

  @Query("{ 'firstname' : ?0 }")
  List<Person> findByThePersonsFirstname(String firstname);

}
```

占位符?0 允许您将方法参数中的值替换为 JSON 查询字符串。

> String 参数值在绑定过程中被转义，这意味着无法通过参数添加 MongoDB 特定的运算符。

您还可以使用 filter 属性来限制将映射到 Java 对象的属性集。例如，

```
public interface PersonRepository extends MongoRepository<Person, String>

  @Query(value="{ 'firstname' : ?0 }", fields="{ 'firstname' : 1, 'lastname' : 1}")
  List<Person> findByThePersonsFirstname(String firstname);

}
```

这将仅返回 Person 对象的 firstname，lastname 和 Id 属性。age 属性 java.lang.Integer 将不会被设置，因此其值将为 null。

10.3.4。使用 SpEL 表达式的基于 JSON 的查询

查询字符串和字段定义可与 SpEL 表达式一起使用，以在运行时创建动态查询。SpEL 表达式可以提供谓词值，并可用于扩展带有子文档的谓词。

表达式通过包含所有参数的数组公开方法参数。以下查询用于[0] 声明谓词值，lastname 该值等效于?0 参数绑定。

```
public interface PersonRepository extends MongoRepository<Person, String>

  @Query("{'lastname': ?#{[0]} }")
  List<Person> findByQueryWithExpression(String param0);
}
```

表达式可用于调用函数，评估条件和构造值。SpEL 表达式与 JSON 一起显示副作用，因为 SpEL 内部的类似 Map 的声明就像 JSON 一样。

```
public interface PersonRepository extends MongoRepository<Person, String>

  @Query("{'id': ?#{ [0] ? {$exists :true} : [1] }}")
  List<Person> findByQueryWithExpressionAndNestedObject(boolean param0,
String param1);
}
```

查询字符串中的 SpEL 可以是增强查询的强大方法，并且可以接受各种不需要的参数。您应确保在将字符串传递给查询之前清理字符串，以避免对查询进行不必要的更改。

表达式支持可以通过查询 SPI 进行扩展，而 org.springframework.data.repository.query.spi.EvaluationContextExtension 不是提供属性，函数和自定义根对象。在构建查询时，在 SpEL 评估时从应用程序上下文中检索扩展。

```
public class SampleEvaluationContextExtension extends
EvaluationContextExtensionSupport {

  @Override
  public String getExtensionId() {
    return "security";
  }

  @Override
  public Map<String, Object> getProperties() {
    return Collections.singletonMap("principal",
SecurityContextHolder.getCurrent().getPrincipal());
  }
}
```

> 引导 MongoRepositoryFactory 自己不是应用程序上下文感知，需要进一步配置才能获取查询 SPI 扩展。

10.3.5。类型安全的查询方法

MongoDB 存储库支持与 QueryDSL 项目集成，后者提供了一种在 Java 中执行类型安全查询的方法。引用项目描述，“不是将查询编写为内联字符串或将它们外部化为 XML 文件，而是通过流畅的 API 构建它们。” 它提供以下功能

- IDE 中的代码完成（所有属性，方法和操作都可以在您喜欢的 Java IDE 中扩展）

- 几乎没有语法上无效的查询允许（所有级别的类型安全）
- 可以安全地引用域类型和属性（不涉及字符串！）
- 通过更好地重构域类型的更改
- 增量查询定义更容易

请参阅 QueryDSL 文档，该文档描述了如何使用 Maven 或 Ant 为基于 APT 的代码生成引导环境。

使用 QueryDSL，您将能够编写查询，如下所示

```
QPerson person = new QPerson("person");
List<Person> result = repository.findAll(person.address.zipCode.eq("C0123"));

Page<Person> page = repository.findAll(person.lastname.contains("a"),
                                       new PageRequest(0, 2, Direction.ASC, "lastname"));
```

QPerson 是一个生成的类（通过 Java 注释后处理工具），Predicate 它允许您编写类型安全的查询。请注意，除了值“C0123”之外，查询中没有字符串。

您可以 Predicate 通过 QueryDslPredicateExecutor 如下所示的界面使用生成的类

```
public interface QueryDslPredicateExecutor<T> {

  T findOne(Predicate predicate);

  List<T> findAll(Predicate predicate);

  List<T> findAll(Predicate predicate, OrderSpecifier<?>... orders);

  Page<T> findAll(Predicate predicate, Pageable pageable);

  Long count(Predicate predicate);
}
```

要在存储库实现中使用它，除了其他存储库接口之外，只需继承它。如下所示

```
public interface PersonRepository extends MongoRepository<Person, String>,
QueryDslPredicateExecutor<Person> {

   // additional finder methods go here
}
```

我们认为您会发现这是一个非常强大的编写 MongoDB 查询的工具。

10.3.6。全文搜索查询

MongoDBs 全文搜索功能是特定于商店的，因此可以找到而 MongoRepository 不是更一般的 CrudRepository。我们需要的是一个定义了全文索引的文档（请参阅创建文本索引一节）。

上的其他方法 MongoRepository 取出 TextCriteria 作为输入参数。除了那些显式方法之外，还可以添加 TextCriteria 派生的存储库方法。该标准将作为附加 AND 标准添加。一旦实体包含带@TextScore 注释的属性，将检索文档全文分数。此外，带@TextScore 注释的属性还可以按文档分数进行排序。

```
@Document
class FullTextDocument {

  @Id String id;
  @TextIndexed String title;
  @TextIndexed String content;
  @TextScore Float score;
}

interface FullTextRepository extends Repository<FullTextDocument, String> {

  // Execute a full-text search and define sorting dynamically
  List<FullTextDocument> findAllBy(TextCriteria criteria, Sort sort);

  // Paginate over a full-text search result
  Page<FullTextDocument> findAllBy(TextCriteria criteria, Pageable pageable);

  // Combine a derived query with a full-text search
  List<FullTextDocument> findByTitleOrderByScoreDesc(String title, TextCriteria
criteria);
}


Sort sort = new Sort("score");
TextCriteria criteria = TextCriteria.forDefaultLanguage().matchingAny("spring",
"data");
List<FullTextDocument> result = repository.findAllBy(criteria, sort);

criteria = TextCriteria.forDefaultLanguage().matching("film");
Page<FullTextDocument> page = repository.findAllBy(criteria, new PageRequest(1,
1, sort));
List<FullTextDocument> result =
repository.findByTitleOrderByScoreDesc("mongodb", criteria);
```

10.3.7。预测

Spring Data Repositories 通常在使用查询方法时返回域模型。但是，有时，您可能需要根据各种原因更改该模型的视图。在本节中，您将学习如何定义投影以提供简化和简化的资源视图。

查看以下域模型：

```
@Entity
public class Person {

  @Id @GeneratedValue
  private Long id;
  private String firstName, lastName;

  @OneToOne
  private Address address;
  …
}

@Entity
public class Address {

  @Id @GeneratedValue
  private Long id;
  private String street, state, country;

  …
}
```

这 Person 有几个属性：

- id 是主键
- firstName 并且 lastName 是数据属性
- address 是另一个域对象的链接

现在假设我们按如下方式创建相应的存储库：

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

  Person findPersonByFirstName(String firstName);
}
```

Spring Data 将返回包含其所有属性的域对象。检索 address 属性有两个选项。一种选择是为这样的 Address 对象定义一个存储库：

```
interface AddressRepository extends CrudRepository<Address, Long> {}
```

在这种情况下，使用 PersonRepository 仍将返回整个 Person 对象。使用 AddressRepository 将只返回 Address。

但是，如果您不想公开 address 细节怎么办？您可以通过定义一个或多个投影为您的存储库服务的使用者提供替代方案。

例 83.简单投影

```
interface NoAddresses {  (1)

  String getFirstName(); (2)

  String getLastName();  (3)
}
```

此预测具有以下详细信息：

1	一个普通的 Java 接口，使其具有声明性。
2	出口 firstName。
3	出口 lastName。

该 NoAddresses 投影仅具有 getter firstName，lastName 意味着它不会提供任何地址信息。在这种情况下，查询方法定义返回 NoAdresses 而不是 Person。

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

  NoAddresses findByFirstName(String firstName);
}
```

预测声明基础类型和与公开属性相关的方法签名之间的合同。因此，需要根据基础类型的属性名称来命名 getter 方法。如果底层属性已命名 firstName，则必须命名 getter 方法，getFirstName 否则 Spring Data 无法查找 source 属性。这种类型的投影也称为*闭合投影*。封闭投影公开属性的子集，因此它们可用于优化查询，以减少数据存储中的选定字段。正如您可能想象的那样，另一种类型是*开放式投影*。

重塑数据

到目前为止，您已经了解了如何使用投影来减少呈现给用户的信息。投影可用于调整公开的数据模型。您可以为投影添加虚拟属性。请看以下投影界面：

例子 84.重命名一个属性

```
interface RenamedProperty {    (1)

  String getFirstName();       (2)

  @Value("#{target.lastName}")
  String getName();            (3)
}
```

此预测具有以下详细信息：

1	一个普通的 Java 接口，使其具有声明性。
2	出口 firstName。
3	出口 name 物业。由于此属性是虚拟的，因此需要 @Value("#{target.lastName}")指定属性源。

支持域模型没有此属性，因此我们需要告诉 Spring Data 从何处获取此属性。虚拟财产是@Value 发挥作用的地方。使用指向支持属性的 SpEL 表达式对 namegetter 进行注释。您可能已经注意到前缀是指向后备对象的变量名称。使用 on 方法可以定义获取值的位置和方式。@ValuelastNamelastNametarget@Value

某些应用程序需要一个人的全名。连接字符串 String.format("%s %s", person.getFirstName(), person.getLastName())是一种可能性，但需要在需要全名的每个地方调用这段代码。投影上的虚拟属性利用了重复该代码的需要。

```
interface FullNameAndCountry {

  @Value("#{target.firstName} #{target.lastName}")
  String getFullName();

  @Value("#{target.address.country}")
  String getCountry();
}
```

实际上，@Value 它提供了对目标对象及其嵌套属性的完全访问权限。SpEL 表达式非常强大，因为定义始终应用于投影方法。让我们将投影中的 SpEL 表达式提升到一个新的水平。

想象一下，您有以下域模型定义：

```
@Entity
```

```
public class User {

  @Id @GeneratedValue
  private Long id;
  private String name;

  private String password;
  …
}
```

这个例子似乎有点人为，但是有可能通过更丰富的域模型和许多预测来意外泄露这些细节。由于 Spring Data 无法识别此类数据的敏感性，因此开发人员可以避免此类情况。不建议将密码存储为纯文本。你真的不应该这样做。对于此示例，您还可以替换 password 其他任何秘密的内容。

在某些情况下，您可能会 password 尽可能地保密，并且不要将其暴露得更多。解决方案是使用@ValueSpEL 表达式创建投影。

```
interface PasswordProjection {
  @Value("#{(target.password == null || target.password.empty) ? null : '******'}")
  String getPassword();
}
```

表达式检查密码是否为 null 空并 null 在此情况下返回，否则设置六个星号表示密码。

10.4。杂

10.4.1。CDI 集成

存储库接口的实例通常由容器创建，在使用 Spring Data 时，Spring 是最自然的选择。从版本 1.3.0 开始，Spring Data MongoDB 附带了一个自定义 CDI 扩展，允许在 CDI 环境中使用存储库抽象。扩展是 JAR 的一部分，因此您需要做的就是将 Spring Data MongoDB JAR 放入类路径中。您现在可以通过为以下内容实施 CDI Producer 来设置基础结构 MongoTemplate：

```
class MongoTemplateProducer {

   @Produces
   @ApplicationScoped
   public MongoOperations createMongoTemplate() throws UnknownHostException,
MongoException {

      MongoDbFactory factory = new SimpleMongoDbFactory(new Mongo(),
"database");
      return new MongoTemplate(factory);
```

```
  }
}
```

Spring Data MongoDB CDI 扩展将 MongoTemplate 获取可用的 CDI bean，并在容器请求存储库类型的 bean 时为 Spring Data 存储库创建代理。因此，获取 Spring Data 存储库的实例是声明@Inject-ed 属性的问题：

```
class RepositoryClient {

 @Inject
 PersonRepository repository;

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

11.审计

11.1。基本

Spring Data 提供了复杂的支持，可以透明地跟踪创建或更改实体的人员以及发生这种情况的时间点。要从该功能中受益，您必须为实体类配备审计元数据，该元数据可以使用注释或通过实现接口来定义。

11.1.1。基于注释的审计元数据

我们提供@CreatedBy，@LastModifiedBy 捕捉谁创建或修改的实体以及用户@CreatedDate 和@LastModifiedDate 捕捉一次发生这种情况的地步。

例子 85.一个被审计的实体

```
class Customer {

 @CreatedBy
 private User user;

 @CreatedDate
 private DateTime createdDate;

 // … further properties omitted
}
```

如您所见，可以有选择地应用注释，具体取决于您要捕获的信息。对于捕获时间点的注释，可以用于 JodaTimes 类型 DateTime，遗留 Java Date 和 CalendarJDK8 日期/时间类型以及 long/的属性 Long。

11.1.2。基于接口的审计元数据

如果您不想使用注释来定义审核元数据，可以让您的域类实现该 Auditable 接口。它公开了所有审计属性的 setter 方法。

还有一个便捷基类 AbstractAuditable，您可以扩展它以避免手动实现接口方法。请注意，这会增加域类与 Spring Data 的耦合，这可能是您想要避免的。通常，基于注释的定义审计元数据的方式是优选的，因为它具有较小的侵入性和更灵活性。

11.1.3。AuditorAware

如果你使用@CreatedBy 或者@LastModifiedBy，审计基础设施需要知道当前的主体。为此，我们提供了一个 AuditorAware<T>SPI 接口，您必须实现该接口，以告知基础架构当前用户或系统与应用程序交互的人员。泛型类型 T 定义了带注释@CreatedBy 或@LastModifiedBy 必须注释的属性的类型。

这是使用 Spring Security Authentication 对象的接口的示例实现：

例子 86.基于 Spring Security 的 AuditorAware 的实现

```
class SpringSecurityAuditorAware implements AuditorAware<User> {

  public User getCurrentAuditor() {

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

    if (authentication == null || !authentication.isAuthenticated()) {
      return null;
    }

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

该实现正在访问 AuthenticationSpring Security 提供的对象，并 UserDetails 从您在 UserDetailsService 实现中创建的查找自定义实例。我们假设您通过该 UserDetails 实现公开域用户，但您也可以根据 Authentication 找到的内容从任何地方查找。

11.2。一般审计配置

激活审计功能只需将 Spring Data Mongo auditing 命名空间元素添加到您的配置中：

示例 87.使用 XML 配置激活审计

```
<mongo:auditing mapping-context-ref="customMappingContext" auditor-aware-
ref="yourAuditorAwareImpl"/>
```

Since Spring Data MongoDB 1.4 auditing can be enabled by annotating a configuration class with the @EnableMongoAuditing annotation.

Example 88. Activating auditing using JavaConfig

```
@Configuration
@EnableMongoAuditing
class Config {

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

If you expose a bean of type AuditorAware to the ApplicationContext, the auditing infrastructure will pick it up automatically and use it to determine the current user to be set on domain types. If you have multiple implementations registered in the ApplicationContext, you can select the one to be used by explicitly setting the auditorAwareRef attribute of @EnableJpaAuditing.

12. Mapping

Rich mapping support is provided by the MappingMongoConverter. MappingMongoConverter has a rich metadata model that provides a full feature set of functionality to map domain objects to MongoDB documents.The mapping metadata model is populated using annotations on your domain objects. However, the infrastructure is not limited to using annotations as the only source of metadata information. The MappingMongoConverter also allows you to map objects to documents without providing any additional metadata, by following a set of conventions.

In this section we will describe the features of the MappingMongoConverter. How to use conventions for mapping objects to documents and how to override those conventions with annotation based mapping metadata.

> SimpleMongoConverter has been deprecated in Spring Data MongoDB M3 as all of its functionality has been subsumed into MappingMongoConverter.

12.1. Convention based Mapping

MappingMongoConverter has a few conventions for mapping objects to documents when no additional mapping metadata is provided. The conventions are:

- The short Java class name is mapped to the collection name in the following manner. The class com.bigbank.SavingsAccount maps to savingsAccount collection name.

- All nested objects are stored as nested objects in the document and **not** as DBRefs
- The converter will use any Spring Converters registered with it to override the default mapping of object properties to document field/values.
- The fields of an object are used to convert to and from fields in the document. Public JavaBean properties are not used.
- You can have a single non-zero argument constructor whose constructor argument names match top level field names of document, that constructor will be used. Otherwise the zero arg constructor will be used. if there is more than one non-zero argument constructor an exception will be thrown.

12.1.1. How the _id field is handled in the mapping layer

MongoDB requires that you have an _id field for all documents. If you don't provide one the driver will assign a ObjectId with a generated value. The "_id" field can be of any type the, other than arrays, so long as it is unique. The driver naturally supports all primitive types and Dates. When using the MappingMongoConverter there are certain rules that govern how properties from the Java class is mapped to this _id field.

The following outlines what field will be mapped to the _id document field:

- A field annotated with @Id (org.springframework.data.annotation.Id) will be mapped to the _id field.
- A field without an annotation but named id will be mapped to the _id field.
- The default field name for identifiers is _id and can be customized via the @Field annotation.

Table 7. Examples for the translation of _id field definitions

Field definition	Resulting Id-Fieldname in MongoDB
String id	_id
@Field String id	_id
@Field("x") String id	x
@Id String x	_id

Table 7. Examples for the translation of _id field definitions

Field definition	Resulting Id-Fieldname in MongoDB
@Field("x") @Id String x	_id

The following outlines what type conversion, if any, will be done on the property mapped to the _id document field.

- If a field named id is declared as a String or BigInteger in the Java class it will be converted to and stored as an ObjectId if possible. ObjectId as a field type is also valid. If you specify a value for id in your application, the conversion to an ObjectId is detected to the MongoDBdriver. If the specified id value cannot be converted to an ObjectId, then the value will be stored as is in the document's _id field.

- If a field named id id field is not declared as a String, BigInteger, or ObjectID in the Java class then you should assign it a value in your application so it can be stored 'as-is' in the document's _id field.

- If no field named id is present in the Java class then an implicit _id file will be generated by the driver but not mapped to a property or field of the Java class.

When querying and updating MongoTemplate will use the converter to handle conversions of the Query and Update objects that correspond to the above rules for saving documents so field names and types used in your queries will be able to match what is in your domain classes.

12.2. Data mapping and type conversion

This section explain how types are mapped to a MongoDB representation and vice versa. Spring Data MongoDB supports all types that can be represented as BSON, MongoDB's internal document format. In addition to these types, Spring Data MongoDB provides a set of built-in converters to map additional types. You can provide your own converters to adjust type conversion, see Overriding Mapping with explicit Converters for further details.

Table 8. Type

Type	Type conversion	Sample

Table 8. Type

Type	Type conversion	Sample
String	native	{"firstname" : "Dave"}
double, Double, float, Float	native	{"weight" : 42.5}
int, Integer, short, Short	native 32-bit integer	{"height" : 42}
long, Long	native 64-bit integer	{"height" : 42}
Date, Timestamp	native	{"date" : ISODate("2019-11-12T23:00:00.809Z")}
byte[]	native	{"bin" : { "$binary" : "AQIDBA==", "$type" : "00" }}
java.util.UUID (Legacy UUID)	native	{"uuid" : { "$binary" : "MEaf1CFQ6lSphaa3b9AtlA==", "$type" : "03" }}
Date	native	{"date" : ISODate("2019-11-12T23:00:00.809Z")}
ObjectId	native	{"_id" : ObjectId("5707a2690364aba3136ab870")}
Array, List, BasicDBList	native	{"cookies" : […]}

Table 8. Type

Type	Type conversion	Sample
boolean, Boolean	native	{"active" : true}
null	native	{"value" : null}
DBObject	native	{"value" : { … }}
Decimal128	native	{"value" : NumberDecimal(…)}
AtomicInteger calling get() before the actual conversion	converter 32-bit integer	{"value" : "741" }
AtomicLong calling get() before the actual conversion	converter 64-bit integer	{"value" : "741" }
BigInteger	converter String	{"value" : "741" }
BigDecimal	converter String	{"value" : "741.99" }
URL	converter	{"website" : "http://projects.spring.io/spring-data-mongodb/" }
Locale	converter	{"locale : "en_US" }
char, Character	converter	{"char" : "a" }

Table 8. Type

Type	Type conversion	Sample
NamedMongoScript	converter Code	{"_id" : "script name", value: (some javascript code)}
java.util.Currency	converter	{"currencyCode" : "EUR"}
LocalDate （Joda，Java 8，JSR310-BackPort）	变流器	{"date" : ISODate("2019-11-12T00:00:00.000Z")}
LocalDateTime，LocalTime，Instant （约达，爪哇 8，JSR310-反向移植）	变流器	{"date" : ISODate("2019-11-12T23:00:00.809Z")}
DateTime （约达）	变流器	{"date" : ISODate("2019-11-12T23:00:00.809Z")}
DateMidnight （约达）	变流器	{"date" : ISODate("2019-11-12T00:00:00.000Z")}
ZoneId （Java 8，JSR310-BackPort）	变流器	{"zoneId" : "ECT - Europe/Paris"}
Box	变流器	{"box" : { "first" : { "x" : 1.0 , "y" : 2.0} , "second" : { "x" : 3.0 , "y" : 4.0}}
Polygon	变流器	{"polygon" : { "points" : [{ "x" : 1.0 , "y" : 2.0} , { "x" : 3.0 , "y" : 4.0} , { "x" : 4.0 , "y" : 5.0}]}}

Table 8. Type

Type	Type conversion	Sample
Circle	变流器	{"circle" : { "center" : { "x" : 1.0 , "y" : 2.0} , "radius" : 3.0 , "metric" : "NEUTRAL"}}
Point	变流器	{"point" : { "x" : 1.0 , "y" : 2.0}}
GeoJsonPoint	变流器	{"point" : { "type" : "Point" , "coordinates" : [3.0 , 4.0] }}
GeoJsonMultiPoint	变流器	{"geoJsonLineString" : {"type":"MultiPoint", "coordinates": [[0 , 0], [0 , 1], [1 , 1]] }}
Sphere	变流器	{"sphere" : { "center" : { "x" : 1.0 , "y" : 2.0} , "radius" : 3.0 , "metric" : "NEUTRAL"}}
GeoJsonPolygon	变流器	{"polygon" : { "type" : "Polygon", "coordinates" : [[[0 , 0], [3 , 6], [6 , 1], [0 , 0]]] }}
GeoJsonMultiPolygon	变流器	{"geoJsonMultiPolygon" : { "type" : "MultiPolygon", "coordinates" : [[[[-73.958 , 40.8003] , [-73.9498 , 40.7968]]], [[[-73.973 , 40.7648] , [-73.9588 , 40.8003]]]] }}
GeoJsonLineString	变流器	{ "geoJsonLineString" : { "type" : "LineString", "coordinates" : [[40 , 5], [41 , 6]] }}
GeoJsonMultiLineString	变流器	{"geoJsonLineString" : { "type" : "MultiLineString", coordinates: [[[-

Table 8. Type

Type	Type conversion	Sample
		73.97162 , 40.78205], [-73.96374 , 40.77715]], [[-73.97880 , 40.77247], [-73.97036 , 40.76811]]] }}

12.3。映射配置

除非明确配置，否则 MappingMongoConverter 在创建时默认创建实例 MongoTemplate。您可以创建自己的实例，MappingMongoConverter 以便在启动域类时告诉它在哪里扫描类路径，以便提取元数据和构造索引。此外，通过创建自己的实例，您可以注册 Spring 转换器以用于将特定类映射到数据库或从数据库映射。

您可以使用基于 Java 或 XML 的元数据配置 MappingMongoConverter 以及 com.mongodb.MongoMongoTemplate。以下是使用 Spring 基于 Java 的配置的示例

例 89. @Configuration 类用于配置 MongoDB 映射支持

```
@Configuration
public class GeoSpatialAppConfig extends AbstractMongoConfiguration {

 @Bean
 public Mongo mongo() throws Exception {
  return new Mongo("localhost");
 }

 @Override
 public String getDatabaseName() {
  return "database";
 }

 @Override
 public String getMappingBasePackage() {
  return "com.bigbank.domain";
 }

 // the following are optional


 @Bean
 @Override
```

```
  public CustomConversions customConversions() throws Exception {
    List<Converter<?, ?>> converterList = new ArrayList<Converter<?, ?>>();
    converterList.add(new
org.springframework.data.mongodb.test.PersonReadConverter());
    converterList.add(new
org.springframework.data.mongodb.test.PersonWriteConverter());
    return new CustomConversions(converterList);
  }

  @Bean
  public LoggingEventListener<MongoMappingEvent> mappingEventsListener() {
    return new LoggingEventListener<MongoMappingEvent>();
  }
}
```

AbstractMongoConfiguration 要求您实现定义 a com.mongodb.Mongo 以及提供数据库名称的方法。AbstractMongoConfiguration 还有一个方法可以覆盖 named getMappingBasePackage(…)，告诉转换器在哪里扫描注释@Document 注释的类。

您可以通过覆盖 afterMappingMongoConverterCreation 之后的方法将其他转换器添加到转换器。上面的示例中还显示了一个 LoggingEventListener 记录 MongoMappingEvent 在 Spring 的 ApplicationContextEvent 基础结构上的日志。

> AbstractMongoConfiguration 将创建一个 MongoTemplate 实例，并在名称下注册容器 mongoTemplate。

您还可以覆盖该方法 UserCredentials getUserCredentials()以提供用于连接到数据库的用户名和密码信息。

Spring’s MongoDB namespace enables you to easily enable mapping functionality in XML

Example 90. XML schema to configure MongoDB mapping support

```
<?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:context="http://www.springframework.org/schema/context"
  xmlns:mongo="http://www.springframework.org/schema/data/mongo"
  xsi:schemaLocation="http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
    http://www.springframework.org/schema/data/mongo
http://www.springframework.org/schema/data/mongo/spring-mongo-1.0.xsd
    http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

  <!-- Default bean name is 'mongo' -->
```

```
  <mongo:mongo host="localhost" port="27017"/>

  <mongo:db-factory dbname="database" mongo-ref="mongo"/>

  <!-- by default look for a Mongo object named 'mongo' - default name used for the
converter is 'mappingConverter' -->
  <mongo:mapping-converter base-package="com.bigbank.domain">
    <mongo:custom-converters>
      <mongo:converter ref="readConverter"/>
      <mongo:converter>
        <bean class="org.springframework.data.mongodb.test.PersonWriteConverter"/>
      </mongo:converter>
    </mongo:custom-converters>
  </mongo:mapping-converter>

  <bean id="readConverter"
class="org.springframework.data.mongodb.test.PersonReadConverter"/>

  <!-- set the mapping converter to be used by the MongoTemplate -->
  <bean id="mongoTemplate"
class="org.springframework.data.mongodb.core.MongoTemplate">
    <constructor-arg name="mongoDbFactory" ref="mongoDbFactory"/>
    <constructor-arg name="mongoConverter" ref="mappingConverter"/>
  </bean>

  <bean
class="org.springframework.data.mongodb.core.mapping.event.LoggingEventListene
r"/>

</beans>
```

The base-package property tells it where to scan for classes annotated with the @org.springframework.data.mongodb.core.mapping.Document annotation.

12.4. Metadata based Mapping

To take full advantage of the object mapping functionality inside the Spring Data/MongoDB support, you should annotate your mapped objects with the @Document annotation. Although it is not necessary for the mapping framework to have this annotation (your POJOs will be mapped correctly, even without any annotations), it allows the classpath scanner to find and pre-process your domain objects to extract the necessary metadata. If you don’t use this annotation, your application will take a slight performance hit the first time you store a domain object because the mapping framework needs to build up its internal metadata model so it knows about the properties of your domain object and how to persist them.

Example 91. Example domain object

```
package com.mycompany.domain;

@Document
```

```
public class Person {

  @Id
  private ObjectId id;

  @Indexed
  private Integer ssn;

  private String firstName;

  @Indexed
  private String lastName;
}
```

该@Id 注解告诉你要使用 MongoDB 的哪个属性映射器_id 属性和@Indexed 注解告诉映射框架调用 createIndex(…)你的文档的那个属性，使得搜索速度更快。

仅对带注释的类型执行自动索引创建@Document。

12.4.1。映射注释概述

MappingMongoConverter 可以使用元数据来驱动对象到文档的映射。下面提供了注释的概述

- @Id - 在字段级别应用以标记用于标识目的的字段。
- @Document - 在类级别应用以指示此类是映射到数据库的候选者。您可以指定将存储数据库的集合的名称。
- @DBRef - 在现场应用以指示使用 com.mongodb.DBRef 存储它。
- @Indexed - 在字段级别应用以描述如何索引字段。
- @CompoundIndex - 在类型级别应用以声明复合索引
- @GeoSpatialIndexed - 在现场级别应用以描述如何对该字段进行地理索引。
- @TextIndexed - 在字段级别应用以标记要包含在文本索引中的字段。
- @Language - 在字段级别应用以设置文本索引的语言覆盖属性。
- @Transient - 默认情况下，所有私有字段都映射到文档，此批注将排除应用它的字段排除在数据库中

- @PersistenceConstructor - 标记给定的构造函数 - 甚至是受保护的构造函数 - 在从数据库实例化对象时使用。构造函数参数按名称映射到检索到的 DBObject 中的键值。

- @Value - 此注释是 Spring Framework 的一部分。在映射框架中，它可以应用于构造函数参数。这使您可以使用 Spring Expression Language 语句在用于构造域对象之前转换在数据库中检索的键值。为了引用给定文档的属性，必须使用如下表达式：@Value("#root.myProperty")where root 指给定文档的根。

- @Field - 在字段级别应用并描述字段的名称，因为它将在 MongoDB BSON 文档中表示，从而允许名称与类的字段名不同。

- @Version - 在字段级别应用用于乐观锁定并检查保存操作的修改。初始值是 zero 每次更新时自动发生的。

映射元数据基础结构在与技术无关的单独的 spring-data-commons 项目中定义。特定的子类在 MongoDB 支持中用于支持基于注释的元数据。如果有需求，也可以采取其他策略。

以下是更复杂映射的示例。

```
@Document
@CompoundIndexes({
   @CompoundIndex(name = "age_idx", def = "{'lastName': 1, 'age': -1}")
})
public class Person<T extends Address> {

  @Id
  private String id;

  @Indexed(unique = true)
  private Integer ssn;

  @Field("fName")
  private String firstName;

  @Indexed
  private String lastName;

  private Integer age;

  @Transient
  private Integer accountTotal;

  @DBRef
  private List<Account> accounts;

  private T address;
```

```
  public Person(Integer ssn) {
    this.ssn = ssn;
  }

  @PersistenceConstructor
  public Person(Integer ssn, String firstName, String lastName, Integer age, T address)
{
    this.ssn = ssn;
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
    this.address = address;
  }

  public String getId() {
    return id;
  }

  // no setter for Id.  (getter is only exposed for some unit testing)

  public Integer getSsn() {
    return ssn;
  }

// other getters/setters omitted
```

12.4.2。定制对象构造

映射子系统允许通过使用注释注释构造函数来定制对象构造
@PersistenceConstructor。用于构造函数参数的值按以下方式解析：

- 如果使用注释注释参数@Value，则计算给定表达式并将结果用作参数值。
- 如果 Java 类型具有名称与输入文档的给定字段匹配的属性，则它的属性信息用于选择适当的构造函数参数以将输入字段值传递给。这只有在 java .class 文件中存在参数名称信息时才有效，这可以通过使用调试信息编译源代码或-parameters 在 Java 8 中使用 javac 的新命令行开关来实现。
- 否则 MappingException 将抛出一个指示无法绑定给定构造函数参数的 a。

```
class OrderItem {

  private @Id String id;
  private int quantity;
  private double unitPrice;
```

```
  OrderItem(String id, @Value("#root.qty ?: 0") int quantity, double unitPrice) {
    this.id = id;
    this.quantity = quantity;
    this.unitPrice = unitPrice;
  }

  // getters/setters ommitted
}

DBObject input = new BasicDBObject("id", "4711");
input.put("unitPrice", 2.5);
input.put("qty",5);
OrderItem item = converter.read(OrderItem.class, input);
```

如果无法解析给定的属性路径@Value，则 quantity 参数注释中的 SpEL 表达式将回退到该值 0。

@PersistenceConstructor 可以在 MappingMongoConverterUnitTests 测试套件中找到使用注释的其他示例。

12.4.3。复合指数

还支持复合索引。它们是在类级别定义的，而不是在单个属性上定义的。

复合索引对于提高涉及多个字段的条件的查询的性能非常重要

这是一个 lastName 按升序和 age 降序创建复合索引的示例：

实施例 92.实施例化合物索引用法

```
package com.mycompany.domain;

@Document
@CompoundIndexes({
    @CompoundIndex(name = "age_idx", def = "{'lastName': 1, 'age': -1}")
})
public class Person {

  @Id
  private ObjectId id;
  private Integer age;
  private String firstName;
  private String lastName;

}
```

12.4.4。文字索引

mongodb v.2.4 默认禁用文本索引功能。

创建文本索引允许将多个字段累积到可搜索的全文索引中。每个集合只能有一个文本索引，因此标记的所有字段@TextIndexed 都合并到此索引中。可以对属性进行加权以影响排名结果的文档分数。文本索引的默认语言是英语，用于将默认语言集更改为@Document(language="spanish")您想要的任何语言。使用名为 language 或@Language 允许在每个文档库上定义语言覆盖的属性。

例 93.示例文本索引用法

```
@Document(language = "spanish")
class SomeEntity {

   @TextIndexed String foo;

   @Language String lang;

   Nested nested;
}

class Nested {

   @TextIndexed(weight=5) String bar;
   String roo;
}
```

12.4.5。使用 DBRefs

映射框架不必存储嵌入在文档中的子对象。您也可以单独存储它们并使用 DBRef 来引用该文档。当从 MongoDB 加载对象时，将急切地解析这些引用，并且您将获得一个映射对象，该对象看起来与嵌入在主文档中的对象相同。

下面是一个使用 DBRef 引用特定文档的示例，该文档独立于引用它的对象而存在（为了简洁起见，这两个类都以内联方式显示）：

```
@Document
public class Account {

  @Id
  private ObjectId id;
  private Float total;
}

@Document
public class Person {
```

```
  @Id
  private ObjectId id;
  @Indexed
  private Integer ssn;
  @DBRef
  private List<Account> accounts;
}
```

没有必要使用类似的东西，@OneToMany 因为映射框架看到你想要一对多关系，因为有一个对象列表。当对象存储在 MongoDB 中时，会有一个 DBRef 列表而不是 Account 对象本身。

> 映射框架不处理级联保存。如果更改 Account 对象引用的 Person 对象，则必须单独保存 Account 对象。调用 save 该 Person 对象不会自动将 Account 对象保存在属性中 accounts。

12.4.6。映射框架事件

事件在映射过程的整个生命周期中触发。“ 生命周期事件”部分对此进行了描述。

只需在 Spring ApplicationContext 中声明这些 bean，就会在调度事件时调用它们。

12.4.7。使用显式转换器覆盖映射

在存储和查询对象时，让 MongoConverter 实例处理所有 Java 类型到 DBObjects 的映射是很方便的。但是，有时您可能希望 MongoConverters 执行大部分工作，但允许您有选择地处理特定类型的转换或优化性能。

要自己选择性地处理转换，请 org.springframework.core.convert.converter.Converter 使用 MongoConverter 注册一个或多个实例。

> Spring 3.0 引入了一个 core.convert 包，它提供了一个通用的类型转换系统。这在名为 Spring Type Conversion 的 Spring 参考文档部分中有详细描述。

该方法 customConversions 的 AbstractMongoConfiguration 可用于配置转换器。这些例子在这里本章开头显示了如何进行使用 Java 和 XML 的配置。

下面是一个 Spring Converter 实现的示例，它从 DBObject 转换为 Person POJO。

```
@ReadingConverter
```

```
public class PersonReadConverter implements Converter<DBObject, Person> {

  public Person convert(DBObject source) {
    Person p = new Person((ObjectId) source.get("_id"), (String) source.get("name"));
    p.setAge((Integer) source.get("age"));
    return p;
  }
}
```

这是一个从 Person 转换为 DBObject 的示例。

```
@WritingConverter
public class PersonWriteConverter implements Converter<Person, DBObject> {

  public DBObject convert(Person source) {
    DBObject dbo = new BasicDBObject();
    dbo.put("_id", source.getId());
    dbo.put("name", source.getFirstName());
    dbo.put("age", source.getAge());
    return dbo;
  }
}
```

13. Cross Store 支持

Sometimes you need to store data in multiple data stores and these data stores can be of different types. One might be relational while the other a document store. For this use case we have created a separate module in the MongoDB support that handles what we call cross-store support. The current implementation is based on JPA as the driver for the relational database and we allow select fields in the Entities to be stored in a Mongo database. In addition to allowing you to store your data in two stores we also coordinate persistence operations for the non-transactional MongoDB store with the transaction life-cycle for the relational database.

13.1. Cross Store Configuration

Assuming that you have a working JPA application and would like to add some cross-store persistence for MongoDB. What do you have to add to your configuration?

首先，您需要在模块上添加依赖项。使用 Maven，可以通过向 pom 添加依赖项来完成：

例子 94.具有 spring-data-mongodb-cross-store 依赖性的示例 Maven pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
```

```
  ...

    <!-- Spring Data -->
    <dependency>
      <groupId>org.springframework.data</groupId>
      <artifactId>spring-data-mongodb-cross-store</artifactId>
      <version>${spring.data.mongo.version}</version>
    </dependency>

  ...

</project>
```

完成此操作后，我们需要为项目启用 AspectJ。跨存储支持是使用 AspectJ 方面实现的，因此通过启用编译时 AspectJ 支持，跨项存储功能将可用于您的项目。在 Maven 中，您可以在 pom 的<build>部分添加一个额外的插件：

例子 95.启用了 AspectJ 插件的示例 Maven pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  ...

  <build>
    <plugins>

      …

      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>aspectj-maven-plugin</artifactId>
        <version>1.0</version>
        <dependencies>
          <!-- NB: You must use Maven 2.0.9 or above or these are ignored (see MNG-
2972) -->
          <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjrt</artifactId>
            <version>${aspectj.version}</version>
          </dependency>
          <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjtools</artifactId>
            <version>${aspectj.version}</version>
          </dependency>
```

```
      </dependencies>
      <executions>
        <execution>
          <goals>
            <goal>compile</goal>
            <goal>test-compile</goal>
          </goals>
        </execution>
      </executions>
      <configuration>
        <outxml>true</outxml>
        <aspectLibraries>
          <aspectLibrary>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aspects</artifactId>
          </aspectLibrary>
          <aspectLibrary>
            <groupId>org.springframework.data</groupId>
            <artifactId>spring-data-mongodb-cross-store</artifactId>
          </aspectLibrary>
        </aspectLibraries>
        <source>1.6</source>
        <target>1.6</target>
      </configuration>
    </plugin>

    ...

  </plugins>
 </build>

...

</project>
```

最后，您需要将项目配置为使用 MongoDB 并配置所使用的方面。应将以下 XML 代码段添加到您的应用程序上下文中：

示例 96.具有 MongoDB 和跨存储方面支持的示例应用程序上下文

```
<?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:jdbc="http://www.springframework.org/schema/jdbc"
  xmlns:jpa="http://www.springframework.org/schema/data/jpa"
  xmlns:mongo="http://www.springframework.org/schema/data/mongo"
  xsi:schemaLocation="http://www.springframework.org/schema/data/mongo
    http://www.springframework.org/schema/data/mongo/spring-mongo.xsd
    http://www.springframework.org/schema/jdbc
    http://www.springframework.org/schema/jdbc/spring-jdbc-3.0.xsd
    http://www.springframework.org/schema/beans
```

```
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/data/jpa
    http://www.springframework.org/schema/data/jpa/spring-jpa-1.0.xsd">

  ...

  <!--  Mongo config -->
  <mongo:mongo host="localhost" port="27017"/>

  <bean id="mongoTemplate"
class="org.springframework.data.mongodb.core.MongoTemplate">
    <constructor-arg name="mongo" ref="mongo"/>
    <constructor-arg name="databaseName" value="test"/>
    <constructor-arg name="defaultCollectionName" value="cross-store"/>
  </bean>

  <bean
class="org.springframework.data.mongodb.core.MongoExceptionTranslator"/>

  <!--  Mongo cross-store aspect config -->
  <bean
class="org.springframework.data.persistence.document.mongo.MongoDocumentBack
ing"
        factory-method="aspectOf">
    <property name="changeSetPersister" ref="mongoChangeSetPersister"/>
  </bean>
  <bean id="mongoChangeSetPersister"

class="org.springframework.data.persistence.document.mongo.MongoChangeSetPers
ister">
    <property name="mongoTemplate" ref="mongoTemplate"/>
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
  </bean>

  ...

</beans>
```

13.2。编写跨存储应用程序

We are assuming that you have a working JPA application so we will only cover the additional steps needed to persist part of your Entity in your Mongo database. First you need to identify the field you want persisted. It should be a domain class and follow the general rules for the Mongo mapping support covered in previous chapters. The field you want persisted in MongoDB should be annotated using the @RelatedDocument annotation. That is really all you need to do!. The cross-store aspects take care of the rest. This includes marking the field with @Transient 所以它不会持久化使用 JPA，跟踪对字段值所做的任何更改并在成功完成事务时将它们写入数据库，在第一

次在应用程序中使用该值时从 MongoDB 加载文档。以下是一个带有注释字段的简单实体的示例@RelatedDocument。

例 97.带有@RelatedDocument 的实体示例

```
@Entity
public class Customer {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String firstName;

 private String lastName;

 @RelatedDocument
 private SurveyInfo surveyInfo;

 // getters and setters omitted
}
```

例 98.要存储为文档的域类的示例

```
public class SurveyInfo {

 private Map<String, String> questionsAndAnswers;

 public SurveyInfo() {
  this.questionsAndAnswers = new HashMap<String, String>();
 }

 public SurveyInfo(Map<String, String> questionsAndAnswers) {
  this.questionsAndAnswers = questionsAndAnswers;
 }

 public Map<String, String> getQuestionsAndAnswers() {
  return questionsAndAnswers;
 }

 public void setQuestionsAndAnswers(Map<String, String> questionsAndAnswers) {
  this.questionsAndAnswers = questionsAndAnswers;
 }

 public SurveyInfo addQuestionAndAnswer(String question, String answer) {
  this.questionsAndAnswers.put(question, answer);
  return this;
 }
}
```

一旦在 Customer 对象上设置了 SurveyInfo，上面配置的 MongoTemplate 用于保存 SurveyInfo 以及有关 JPA 实体的一些元数据存储在以 JPA Entity 类的完全限定名称命名的 MongoDB 集合中。以下代码：

示例 99.使用为跨存储持久性配置的 JPA 实体的代码示例

```
Customer customer = new Customer();
customer.setFirstName("Sven");
customer.setLastName("Olafsen");
SurveyInfo surveyInfo = new SurveyInfo()
  .addQuestionAndAnswer("age", "22")
  .addQuestionAndAnswer("married", "Yes")
  .addQuestionAndAnswer("citizenship", "Norwegian");
customer.setSurveyInfo(surveyInfo);
customerRepository.save(customer);
```

执行上面的代码会导致以下 JSON 文档存储在 MongoDB 中。

示例 100.存储在 MongoDB 中的 JSON 文档的示例

```
{ "_id" : ObjectId( "4d9e8b6e3c55287f87d4b79e" ),
  "_entity_id" : 1,
  "_entity_class" :
"org.springframework.data.mongodb.examples.custsvc.domain.Customer",
  "_entity_field_name" : "surveyInfo",
  "questionsAndAnswers" : { "married" : "Yes",
    "age" : "22",
    "citizenship" : "Norwegian" },
  "_entity_field_class" :
"org.springframework.data.mongodb.examples.custsvc.domain.SurveyInfo" }
```

14.记录支持

在 maven 模块“spring-data-mongodb-log4j”中提供了 Log4j 的 appender。注意，不依赖于其他 Spring Mongo 模块，只有 MongoDB 驱动程序。

14.1。MongoDB Log4j 配置

这是一个示例配置

```
log4j.rootCategory=INFO, mongo

log4j.appender.mongo=org.springframework.data.document.mongodb.log4j.MongoL
og4jAppender
log4j.appender.mongo.layout=org.apache.log4j.PatternLayout
log4j.appender.mongo.layout.ConversionPattern=%d %p [%c] - <%m>%n
log4j.appender.mongo.host = localhost
log4j.appender.mongo.port = 27017
log4j.appender.mongo.database = logs
```

```
log4j.appender.mongo.collectionPattern = %X{year}%X{month}
log4j.appender.mongo.applicationId = my.application
log4j.appender.mongo.warnOrHigherWriteConcern = FSYNC_SAFE

log4j.category.org.apache.activemq=ERROR
log4j.category.org.springframework.batch=DEBUG
log4j.category.org.springframework.data.document.mongodb=DEBUG
log4j.category.org.springframework.transaction=INFO
```

除主机和端口外，重要的配置是数据库和 collectionPattern。变量 year，month，day 和 hour 可供您在形成集合名称使用。这是为了支持在对应于特定时间段的集合中对日志信息进行分组的通用约定，例如每天的集合。

还有一个 applicationId 放入存储的消息中。：将文档从登录为下列密钥存储 level，name，applicationId，timestamp，properties，traceback，和 message。

14.1.1。使用身份验证

可以将 MongoDB Log4j appender 配置为使用用户名/密码身份验证。使用指定的数据库执行身份验证。不同的 authenticationDatabase 可以指定覆盖默认行为。

```
# ...
log4j.appender.mongo.username = admin
log4j.appender.mongo.password = test
log4j.appender.mongo.authenticationDatabase = logs
# ...
```

> 身份验证失败会在日志记录期间导致异常，并传播到日志记录方法的调用方。

15. JMX 支持

对 MongoDB 的 JMX 支持公开了在单个 MongoDB 服务器实例的 admin 数据库上执行'serverStatus'命令的结果。它还公开了一个管理 MBean，MongoAdmin，它允许您执行管理操作，如删除或创建数据库。JMX 功能基于 Spring Framework 中提供的 JMX 功能集。有关详细信息，请参见此处

15.1。MongoDB JMX 配置

Spring 的 Mongo 命名空间使您可以轻松启用 JMX 功能

例 101.用于配置 MongoDB 的 XML 模式

```
<?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:context="http://www.springframework.org/schema/context"
  xmlns:mongo="http://www.springframework.org/schema/data/mongo"
  xsi:schemaLocation="
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd
    http://www.springframework.org/schema/data/mongo
    http://www.springframework.org/schema/data/mongo/spring-mongo-1.0.xsd
    http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <!-- Default bean name is 'mongo' -->
    <mongo:mongo host="localhost" port="27017"/>

    <!-- by default look for a Mongo object named 'mongo' -->
    <mongo:jmx/>

    <context:mbean-export/>

    <!-- To translate any MongoExceptions thrown in @Repository annotated classes --
>
    <context:annotation-config/>

    <bean id="registry"
class="org.springframework.remoting.rmi.RmiRegistryFactoryBean" p:port="1099"
/>

    <!-- Expose JMX over RMI -->
    <bean id="serverConnector"
class="org.springframework.jmx.support.ConnectorServerFactoryBean"
        depends-on="registry"
        p:objectName="connector:name=rmi"

p:serviceUrl="service:jmx:rmi://localhost/jndi/rmi://localhost:1099/myconnector" />

</beans>
```

这将暴露几个 MBean

- AssertMetrics
- BackgroundFlushingMetrics
- BtreeIndexCounters
- ConnectionMetrics
- GlobalLoclMetrics
- MemoryMetrics

- OperationCounters
- ServerInfo
- MongoAdmin

这在 JConsole 的屏幕截图中显示如下

16. MongoDB 3.0 支持

Spring Data MongoDB allows usage of both MongoDB Java driver generations 2 and 3 when connecting to a MongoDB 2.6/3.0 server running *MMap.v1* or a MongoDB server 3.0 using *MMap.v1* or the *WiredTiger* storage engine.

> Please refer to the driver and database specific documentation for major differences between those.

Operations that are no longer valid using a 3.x MongoDB Java driver have been deprecated within Spring Data and will be removed in a subsequent release.

16.1. Using Spring Data MongoDB with MongoDB 3.0

16.1.1. Configuration Options

Some of the configuration options have been changed / removed for the *mongo-java-driver*. The following options will be ignored using the generation 3 driver:

- autoConnectRetry
- maxAutoConnectRetryTime
- slaveOk

Generally it is recommended to use the <mongo:mongo-client … /> and <mongo:client-options … /> elements instead of <mongo:mongo … /> when doing XML based configuration, since those elements will only provide you with attributes valid for the 3 generation java driver.

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

  <mongo:mongo-client host="127.0.0.1" port="27017">
    <mongo:client-options write-concern="NORMAL" />
  </mongo:mongo-client>

</beans>
```

16.1.2. WriteConcern and WriteConcernChecking

The WriteConcern.NONE, which had been used as default by Spring Data MongoDB, was removed in 3.0. Therefore in a MongoDB 3 environment the WriteConcern will be defaulted to WriteConcern.UNACKNOWLEGED. In case WriteResultChecking.EXCEPTION is enabled the WriteConcern will be altered to WriteConcern.ACKNOWLEDGED for write operations, as otherwise errors during execution would not be throw correctly, since simply not raised by the driver.

16.1.3. Authentication

MongoDB Server generation 3 changed the authentication model when connecting to the DB. Therefore some of the configuration options available for authentication are no longer valid. Please use the MongoClient specific options for setting credentials via MongoCredential to provide authentication data.

```
@Configuration
public class ApplicationContextEventTestsAppConfig extends
AbstractMongoConfiguration {

  @Override
  public String getDatabaseName() {
    return "database";
  }

  @Override
  @Bean
  public Mongo mongo() throws Exception {
    return new MongoClient(singletonList(new ServerAddress("127.0.0.1", 27017)),
      singletonList(MongoCredential.createCredential("name", "db",
"pwd".toCharArray())));
  }
}
```

In order to use authentication with XML configuration use the credentials attribue on <mongo-client>.

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

  <mongo:mongo-client credentials="user:password@database" />

</beans>
```

16.1.4. Other things to be aware of

This section covers additional things to keep in mind when using the 3.0 driver.

- IndexOperations.resetIndexCache() is no longer supported.
- Any MapReduceOptions.extraOption is silently ignored.
- WriteResult does not longer hold error information but throws an Exception.

- MongoOperations.executeInSession(…) no longer calls requestStart / requestDone.
- Index name generation has become a driver internal operations, still we use the 2.x schema to generate names.
- Some Exception messages differ between the generation 2 and 3 servers as well as between *MMap.v1* and *WiredTiger* storage engine.

Appendix

Appendix A: Namespace reference

The <repositories /> element

The <repositories /> element triggers the setup of the Spring Data repository infrastructure. The most important attribute is base-package which defines the package to scan for Spring Data repository interfaces.[3]

Table 9. Attributes

Name	Description
base-package	Defines the package to be used to be scanned for repository interfaces extending *Repository (actual interface is determined by specific Spring Data module) in auto detection mode. All packages below the configured package will be scanned, too. Wildcards are allowed.
repository-impl-postfix	Defines the postfix to autodetect custom repository implementations. Classes whose names end with the configured postfix will be considered as candidates. Defaults to Impl.
query-lookup-strategy	Determines the strategy to be used to create finder queries. See Query lookup strategies for details. Defaults to create-if-not-found.
named-queries-location	Defines the location to look for a Properties file containing externally defined queries.
consider-nested-	Controls whether nested repository interface definitions should be

Table 9. Attributes

Name	Description
repositories	considered. Defaults to false.

Appendix B: Populators namespace reference

The <populator /> element

The <populator /> element allows to populate the a data store via the Spring Data repository infrastructure.[4]

Table 10. Attributes

Name	Description
locations	Where to find the files to read the objects from the repository shall be populated with.

Appendix C: Repository query keywords

Supported query keywords

The following table lists the keywords generally supported by the Spring Data repository query derivation mechanism. However, consult the store-specific documentation for the exact list of supported keywords, because some listed here might not be supported in a particular store.

Table 11. Query keywords

Logical keyword	Keyword expressions
AND	And

Table 11. Query keywords

Logical keyword	Keyword expressions
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_NOT_NULL	NotNull, IsNotNull
IS_NULL	Null, IsNull

Table 11. Query keywords

Logical keyword	Keyword expressions
LESS_THAN	LessThan, IsLessThan
LESS_THAN_EQUAL	LessThanEqual, IsLessThanEqual
LIKE	Like, IsLike
NEAR	Near, IsNear
NOT	Not, IsNot
NOT_IN	NotIn, IsNotIn
NOT_LIKE	NotLike, IsNotLike
REGEX	Regex, MatchesRegex, Matches
STARTING_WITH	StartingWith, IsStartingWith, StartsWith
TRUE	True, IsTrue
WITHIN	Within, IsWithin

Appendix D: Repository query return types

Supported query return types

The following table lists the return types generally supported by Spring Data repositories. However, consult the store-specific documentation for the exact list of supported return types, because some listed here might not be supported in a particular store.

	Geospatial types like (GeoResult, GeoResults, GeoPage) are only available for data stores that support geospatial queries.

Table 12. Query return types

Return type	**Description**
void	表示没有返回值。
基元	Java 原语。
包装类型	Java 包装器类型。
T	一个独特的实体。期望查询方法最多返回一个结果。如果没有找到结果，null 则返回。不止一个结果会触发一个 IncorrectResultSizeDataAccessException。
Iterator<T>	安 Iterator。
Collection<T>	一 Collection。
List<T>	一 List。
Optional<T>	Java 8 或 Guava Optional。期望查询方法最多返回一个结果。如果没有找到结果 Optional.empty()/ Optional.absent()返回。不止一个结果会触发一个 IncorrectResultSizeDataAccessException。
Option<T>	Scala 或 JavaSlang Option 类型。与 Optional 上面描述的 Java 8 语义相同的行为。
Stream<T>	一个 Java 8 Stream。

Future<T>	一 Future。预计要注释的方法，@Async 并要求启用 Spring 的异步方法执行功能。
CompletableFuture<T>	一个 Java 8 CompletableFuture。预计要注释的方法，@Async 并要求启用 Spring 的异步方法执行功能。
ListenableFuture	一 org.springframework.util.concurrent.ListenableFuture。预计要注释的方法，@Async 并要求启用 Spring 的异步方法执行功能。
Slice	一个大小的数据块，包含是否有更多可用数据的信息。需要 Pageable 方法参数。
Page<T>	A Slice 附加信息，例如结果总数。需要 Pageable 方法参数。
GeoResult<T>	带有附加信息的结果条目，例如到参考位置的距离。
GeoResults<T>	GeoResult<T>带有附加信息的列表，例如到参考位置的平均距离。
GeoPage<T>	甲 Page 带 GeoResult<T>，例如平均距离的参考位置。

1。Spring 参考文档中的 JavaConfig

2。春天的 HATEOAS - https://github.com/SpringSource/spring-hateoas

3。请参阅 XML 配置

4。请参阅 XML 配置

版本 1.10.6.RELEASE
上次更新时间：2017-07-26 23:44:29 MESZ