

Spring数据MongoDB - 参考文档

马克·波拉克
托马斯·里斯贝格
奥利弗·吉尔克
科斯廷·洛
乔恩布里斯宾
托马斯·达里蒙特
克里斯托夫·斯特罗布
马克·帕卢奇

版本1.10.3.RELEASE, 2017年4月19日

©2008-2017原作者。



只要您不对这些副本收取任何费用，并且进一步规定，每个副本都包含本版权声明，无论是以印刷版还是电子版分发，本文档的副本可供您自己使用并分发给他人。

目录

前言

知道春天

2.了解NoSQL和Document数据库

3.要求

4.其他帮助资源

4.1。支持

4.1.1。社区论坛

4.1.2。专业支持

4.2。发展后

5.新的和值得注意的

5.1。Spring Data MongoDB 1.10中的新功能

5.2。Spring Data MongoDB 1.9中的新功能

5.3。Spring Data MongoDB 1.8中的新功能

5.4。Spring Data MongoDB 1.7中的新功能

依赖关系

6.1。使用Spring Boot进行依赖管理

6.2。Spring框架

7.使用Spring数据存储库

7.1。核心概念

7.2。查询方式

7.3。定义存储库接口

7.3.1。微调存储库定义

7.3.2。使用多个Spring数据模块的存储库

7.4。定义查询方法

7.4.1。查询策略

7.4.2。查询创建

7.4.3。属性表达式

7.4.4。特殊参数处理

7.4.5。限制查询结果

7.4.6。流式查询结果

7.4.7。异步查询结果

7.5。创建存储库实例

7.5.1。XML配置

7.5.2。JavaConfig

7.5.3。独立使用

7.6。Spring数据存储库的自定义实现

7.6.1。将自定义行为添加到单个存储库

7.6.2。将自定义行为添加到所有存储库

7.7。从聚集根源出版事件

7.8。Spring数据扩展

7.8.1。Querydsl扩展

7.8.2。网络支持

7.8.3。存储库populator

7.8.4。传统网络支持

参考文献

8.介绍

8.1. 文件结构

9. MongoDB支持

9.1. 入门

9.2. 示例存储库

9.3. 用Spring连接到MongoDB

9.3.1. 使用基于Java的元数据注册Mongo实例

9.3.2. 使用基于XML的元数据注册Mongo实例

9.3.3. MongoClientFactory界面

9.3.4. 使用基于Java的元数据注册MongoClientFactory实例

9.3.5. 使用基于XML的元数据注册MongoClientFactory实例

9.4. MongoTemplate简介

9.4.1. 实例化MongoTemplate

9.4.2. WriteResultChecking策略

9.4.3. WriteConcern

9.4.4. WriteConcernResolver

9.5. 保存，更新和删除文档

9.5.1. 如何 _id 在映射层中处理该字段

9.5.2. 类型映射

9.5.3. 保存和插入文档的方法

9.5.4. 更新集合中的文档

9.5.5. 将文档升级到集合中

9.5.6. 在集合中查找和升级文档

9.5.7. 删除文件的方法

9.5.8. 乐观锁定

9.6. 查询文件

9.6.1. 查询集合中的文档

9.6.2. 方法查询文件

9.6.3. 地理空间查询

9.6.4. GeoJSON支持

9.6.5. 全文查询

9.7. 按示例查询

9.7.1. 介绍

9.7.2. 用法

9.7.3. 示例匹配器

9.7.4. 执行一个例子

9.8. 地图 - 减少操作

9.8.1. 使用示例

9.9. 脚本操作

9.9.1. 使用示例

9.10. 集团业务

9.10.1. 使用示例

9.11. 聚合框架支持

9.11.1. 基本概念

9.11.2. 支持的聚合操作

9.11.3. 投影表达式

9.11.4. 分面分类

9.12. 使用自定义转换器覆盖默认映射

- 9.12.1。使用注册的Spring Converter保存
- 9.12.2。阅读使用弹簧转换器
- 9.12.3。使用MongoConverter注册Spring Converters
- 9.12.4。转换器消歧
- 9.13。索引和收集管理
 - 9.13.1。创建索引的方法
 - 9.13.2。访问索引信息
 - 9.13.3。使用Collection的方法
- 9.14。执行命令
 - 9.14.1。执行命令的方法
- 9.15。生命周期活动
- 9.16。异常翻译
- 9.17。执行回调
- 9.18。GridFS支持

MongoDB存储库

- 10.1。介绍
- 10.2。用法
- 10.3。查询方式
 - 10.3.1。存储库删除查询
 - 10.3.2。地理空间存储库查询
 - 10.3.3。MongoDB基于JSON的查询方法和字段限制
 - 10.3.4。具有SpEL表达式的基于JSON的查询
 - 10.3.5。类型安全查询方法
 - 10.3.6。全文搜索查询
 - 10.3.7。预测
- 10.4。杂
 - 10.4.1。CDI整合

11.审计

- 11.1。基本
 - 11.1.1。基于注释的审计元数据
 - 11.1.2。基于接口的审计元数据
 - 11.1.3。AuditorAware
- 11.2。一般审核配置

映射

- 12.1。基于会议的映射
 - 12.1.1。如何_id 在映射层中处理该字段
- 12.2。数据映射和类型转换
- 12.3。映射配置
- 12.4。基于元数据的映射
 - 12.4.1。映射注释概述
 - 12.4.2。定制对象构造
 - 12.4.3。复合指数
 - 12.4.4。文本索引
 - 12.4.5。使用DBRefs
 - 12.4.6。映射框架事件
 - 12.4.7。用显式转换器覆盖映射

十字架支持

- 13.1。交叉存储配置

13.2. 写十字架应用程序

日志记录支持

14.1. MongoDB Log4j配置

14.1.1. 使用身份验证

JMX支持

15.1. MongoDB JMX配置

16. MongoDB 3.0支持

16.1. MongoDB使用Spring数据MongoDB 3.0

16.1.1. 配置选项

16.1.2. WriteConcern和WriteConcernChecking

16.1.3. 认证

16.1.4. 其他要注意的事情

附录

附录A：命名空间参考

<repositories />元素

附录B：Populators命名空间参考

<populator />元素

附录C：存储库查询关键字

支持的查询关键字

附录D：存储库查询返回类型

支持的查询返回类型

前言

Spring Data MongoDB项目将核心Spring概念应用于使用MongoDB文档样式数据存储开发解决方案。我们提供一个“模板”作为存储和查询文档的高级抽象。您将注意到与Spring Framework中的JDBC支持的相似之处。

本文档是Spring Data - 文档支持的参考指南。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。

本节提供了Spring和Document数据库的一些基本介绍。文档的其余部分仅指Spring Data MongoDB功能，并假定用户熟悉MongoDB和Spring概念。

知道春天

Spring Data使用Spring框架的核心

(<http://docs.spring.io/spring/docs/4.3.8.RELEASE/spring-framework-reference/html/spring-core.html>)功能，如IoC (<http://docs.spring.io/spring/docs/4.3.8.RELEASE/spring-framework-reference/html/beans.html>)容器，类型转换系统 (<http://docs.spring.io/spring/docs/4.3.8.RELEASE/spring-framework-reference/html/validation.html#core-convert>)，表达式语言 (<http://docs.spring.io/spring/docs/4.3.8.RELEASE/spring-framework-reference/html/expressions.html>)，JMX集成 (<http://docs.spring.io/spring/docs/4.3.8.RELEASE/spring-framework-reference/html/jmx.html>)和便携式DAO异常层次结构 (<http://docs.spring.io/spring/docs/4.3.8.RELEASE/spring-framework-reference/html/dao.html#dao-exceptions>)。了解Spring API并不重要，了解其背后的概念。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` 是可以使用“独立”的，而没有Spring容器的任何其他服务。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，您可以参考有关Spring Framework详细说明的全面（有时是解除武装）的文档。有很多文章，博客条目和书籍，请查看Spring框架主页 (<https://spring.io/docs>)了解更多信息。

2.了解NoSQL和Document数据库

NoSQL的商店暴风雨地占据了存储世界。这是一个广泛的领域，具有大量的解决方案，术语和模式（使事情更糟，甚至术语本身具有多重含义 (<https://www.google.com/search?q=nosql+acronym>)）。虽然一些原则是常见的，但用户在某种程度上对MongoDB熟悉至关重要。了解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例子 - 通常不需要5-10分钟的时间，如果你从RDMBS的背景中多次运行这些练习可以开眼界。

了解MongoDB的理由 (<https://www.mongodb.org/>)是www.mongodb.org (<https://www.mongodb.org/>)。以下是其他有用资源的列表：

- 本手册 (<http://docs.mongodb.org/manual/>)介绍MongoDB，并包含入门指南，参考文档和教程的链接。
- 在网上外壳 (<https://try.mongodb.org/>)提供了一个便捷的方式结合一个MongoDB实例交互与在线教程。
(<http://docs.mongodb.org/manual/tutorial/getting-started/>)
- MongoDB Java语言中心 (<http://docs.mongodb.org/ecosystem/drivers/java/>)
- 几本书 (<https://www.mongodb.org/books>)可供购买
- Karl Seguin的在线书: The Little MongoDB Book (<http://openmymind.net/mongodb.pdf>)

3.要求

Spring数据MongoDB 1.x二进制文件需要JDK 6.0及以上级别，以及Spring Framework (<https://spring.io/docs>) 4.3.8.RELEASE及以上版本。

在文档商店方面，MongoDB (<https://www.mongodb.org/>)至少要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，我们尝试提供我们认为从Spring Data MongoDB模块开始的易于遵循的指南。但是，如果您遇到问题或您只是寻求建议，请随时使用以下链接之一：

4.1。支持

有几个支持选项可用：

4.1.1。社区论坛

Stackoverflow (<https://stackoverflow.com/questions/tagged/spring-data>)上的Spring数据Stackoverflow (<https://stackoverflow.com/questions/tagged/spring-data>)是所有Spring数据（而不仅仅是Document）用户共享信息并相互帮助的标签。请注意，只需要注册才能发布。

4.1.2。专业支持

Spring Data和Spring之后的公司Pivotal Software公司 (<https://pivotal.io/>)提供了专业的源代码支持以及有保证的响应时间。

4.2。发展后

有关Spring Data Mongo源代码存储库的信息，请参阅Spring Data Mongo主页 (<http://projects.spring.io/spring-data-mongodb/>)。你可以帮助使弹簧的数据最好通过社会各界对开发人员交流服务Spring社区的需求 #1 (<https://stackoverflow.com/questions/tagged/spring-data>)。要遵循开发者活动，请查看Spring Data Mongo主页上的邮件列表信息。如果您遇到错误或想提出改进建议，请在Spring Data 问题跟踪器 (<https://jira.spring.io/browse/DATAMONGO>)上创建一张机票。要了解最新的新闻和春季生态系统公告，请订阅Spring Community Portal (<https://spring.io>)。最后，您可以在Twitter（SpringData (<https://twitter.com/SpringData>)）上关注Spring 博客 (<https://spring.io/blog>)或项目团队。（<https://twitter.com/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驱动3.4。
- 对于新的注解 `@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）。
- 在创建索引时支持部分过滤器表达式。
- 在加载/转换 `DBRef s` 时发布生命周期事件。
- 添加了查询示例的任意匹配模式。
- 支持 `$caseSensitive` 和 `$diacriticSensitive` 文本搜索。
- 支持GeoJSON多边形孔。
- 通过批量获取 `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` , `@`
- 支持存储库查询方法中的投影。
- 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。
- 声明与MongoDB 3.0和MongoDB Java Driver 3.2的兼容性（请参阅：MongoDB 3.0支持）。

5.3。 Spring Data MongoDB 1.8中的新功能

- `Criteria` 提供支持创建 `$geoIntersects` 。
- 支持规划环境地政司表现
(<http://docs.spring.io/spring/docs/4.3.8.RELEASE/spring-framework-reference/html/expressions.html>)在 `@Query` 。
- `MongoMappingEvents` 公开其发出的集合名称。
- 改进的支持 `<mongo:mongo-client credentials="..." />` 。
- 改进的索引创建失败错误消息。

5.4。 Spring Data MongoDB 1.7中的新功能

- 声明与MongoDB 3.0和MongoDB Java驱动程序3-beta3的兼容性（请参阅：MongoDB 3.0支持）。
- 支持JSR-310和ThreeTen后台日期/时间类型。
- 允许 `Stream` 作为查询方法返回类型（参见：查询方法）。
- 在域类型和查询中添加了GeoJSON (<http://geojson.org/>)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` 关键字对属性的收集的支持。
- 支持 `$bit`，`$mul` 和 `$position` 运营商 `Update`。

依赖关系

由于个别Spring数据模块的初始日期不同，其中大多数都带有不同的主要和次要版本号。找到兼容版本的最简单的方法是依靠我们随附的兼容版本定义的Spring Data Release Train BOM。在Maven项目你会声明在这种依赖 `<dependencyManagement />` 你的POM的部分：

示例1. 使用Spring Data发行列表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>
```

XML

目前发布的版本火车是 `Ingalls-SR3`。火车名称按字母顺序升序，目前可用的火车名称列在[这里](https://github.com/spring-projects/spring-data-commons/wiki/Release-planning) (<https://github.com/spring-projects/spring-data-commons/wiki/Release-planning>)。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

使用BOM的一个工作示例可以在我们的[Spring Data](https://github.com/spring-projects/spring-data-examples/tree/master/bom)示例存储库中找到 (<https://github.com/spring-projects/spring-data-examples/tree/master/bom>)。如果这样就可以声明要在 `<dependencies />` 块中使用没有版本的Spring数据模块。

示例2. 声明对Spring数据模块的依赖

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

XML

6.1. 使用Spring Boot进行依赖管理

Spring Boot已经为您选择了最新版本的Spring数据模块。如果您想要升级到较新版本，只需将该属性配置 `spring-data-releasetrain.version` 为您要使用的列车名称和迭代。

6.2。Spring框架

当前版本的Spring Data模块需要Spring Framework版本4.3.8.RELEASE或更好。这些模块也可能会使用该较小版本的旧版本错误版本。但是，强烈建议您使用该版本中的最新版本。

7.使用Spring数据仓库

Spring数据库抽象的目标是大大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 数据仓库文档和您的模块



本章介绍了Spring Data存储库的核心概念和接口。本章中的信息是从Spring Data Commons模块中获取的。它使用Java Persistence API (JPA) 模块的配置和代码示例。将XML命名空间声明和要扩展的类型调整为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数据模块支持的XML配置，Repository查询关键字涵盖了一般由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一章。

7.1。核心概念

Spring数据库抽象中的中央界面 **Repository**（可能不是什么惊喜）。管理域类以及域类的id类型作为类型参数。此接口主要作为标记接口捕获要使用的类型，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); ❶

    T findOne(ID primaryKey);      ❷

    Iterable<T> findAll();         ❸

    Long count();                  ❹

    void delete(T entity);         ❺

    boolean exists(ID primaryKey); ❻

    // ... more functionality omitted.
}
```

JAVA

- ❶ 保存给定的实体。
- ❷ 返回由给定的ID标识的实体。
- ❸ 返回所有实体。
- ❹ 返回实体数。
- ❺ 删除给定的实体。
- ❻ 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化技术不可知的接口，例如**CrudRepository**，这些接口还扩展和暴露了底层持久性技术的功能。

除此之外，`CrudRepository` 还有一个 `PagingAndSortingRepository` 抽象方法可以添加其他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);
}
```

JAVA

访问第二页的 `User` 页面大小为20，你可以简单地做这样的事情：

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

JAVA

除了查询方法之外，还可以查询计数和删除查询的推导。

示例5. 派生计数查询

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

    Long countByLastname(String lastname);
}
```

JAVA

派生删除查询

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

    Long deleteByLastname(String lastname);

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

JAVA

7.2. 查询方式

标准CRUD功能库通常在基础数据存储上有查询。使用Spring数据，声明这些查询将成为四个步骤：

1. 声明扩展Repository或其一个子接口的接口，并将其键入将要处理的域类和ID类型。

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

JAVA

2. 在界面上声明查询方法。

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

JAVA

3. 设置Spring为这些接口创建代理实例。通过JavaConfig:

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

@EnableJpaRepositories
class Config {}
```

JAVA

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>
```

XML

在这个例子中使用JPA命名空间。如果您正在为任何其他商店使用存储库抽象，则需要将其更改为商店模块的相应命名空间声明，该名称空间声明应该进行交换 `jpa`，例如 `mongodb`。

另外，请注意，JavaConfig变体不会明确地配置程序包，因为默认情况下使用注释类的程序包。要自定义要扫描的软件包，请使用 `basePackage`... 数据存储特定存储库-annotation的 `@Enable`... 属性之一。

4. 获取注册表实例并使用它。

```
public class SomeClient {

  @Autowired
  private PersonRepository repository;

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

JAVA

下面的部分详细说明每一步。

7.3. 定义存储库接口

作为第一步，您定义一个域类别的存储库接口。该接口必须扩展 `Repository` 并输入到域类和ID类型。如果要公开该域类型的CRUD方法，则扩展 `CrudRepository` 而不是 `Repository`。

7.3.1. 微调存储库定义

通常情况下，你的资料库界面将延长 `Repository`，`CrudRepository` 或 `PagingAndSortingRepository`。或者，如果您不想扩展Spring数据接口，还可以使用它来注释存储库界面 `@RepositoryDefinition`。扩展 `CrudRepository` 公开了一套完整的方法来操纵您的实体。如果您希望对所暴露的方法有选择性，只需将要暴露的方法复制 `CrudRepository` 到您的域库中即可。



这允许您在提供的Spring数据存储库功能之上定义自己的抽象。

示例7.选择性地暴露CRUD方法

```

@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);
}

```

JAVA

在此第一步中，您为所有域存储库定义了一个公共基础接口，并将其暴露出来 `findOne(...)`。 `save(...)` 这些方法将被路由到由Spring Data提供的您选择的存储库的基本存储库实现中，例如在JPA的情况下 `SimpleJpaRepository`，因为他们正在匹配方法签名 `CrudRepository`。所以 `UserRepository` 现在将能够保存用户，并通过id查找单个，以及触发查询以 `Users` 通过其电子邮件地址查找。



请注意，中间版本库接口被注释为 `@NoRepositoryBean`。确保将该注释添加到Spring Data不应在运行时创建实例的所有存储库接口。

7.3.2. 使用多个Spring数据模块的存储库

在应用程序中使用唯一的Spring数据模块使事情变得简单，因此定义范围内的所有存储库接口都绑定到Spring数据模块。有时应用程序需要使用多个Spring数据模块。在这种情况下，存储库定义需要区分持久性技术。Spring Data进入严格的存储库配置模式，因为它在类路径上检测到多个存储库工厂。严格的配置需要有关存储库或域类的详细信息来决定用于存储库定义的Spring数据模块绑定：

1. 如果存储库定义扩展了模块特定的存储库，那么它是特定 Spring数据模块的有效候选者。
2. 如果域类使用模块特定类型注释进行注释，那么它是特定Spring数据模块的有效候选项。Spring数据模块接受第三方注释（如JPA `@Entity`）或提供自己的注释，例如 `@Document` Spring 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> {
    ...
}

```

JAVA

`MyRepository` 并 `UserRepository` 延长 `JpaRepository` 他们的类型层次。它们是Spring Data JPA模块的有效候选。

示例9.使用通用接口的存储库定义

JAVA

```

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数据模块是非常好的，但是多个模块无法区分哪些特定的Spring Data这些存储库应该绑定。

示例10. 使用带有注释的域类的存储库定义

JAVA

```

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

@Entity
public class Person {
    ...
}

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

@Document
public class User {
    ...
}

```

PersonRepository 引用 Person 用JPA注释注释 @Entity，因此这个存储库显然属于Spring Data JPA。UserRepository 使用 User 注释与Spring数据MongoDB的 @Document 注释。

示例11. 使用具有混合注释的域类的存储库定义

JAVA

```

interface JpaPersonRepository extends Repository<Person, Long> {
    ...
}

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

@Entity
@Document
public class Person {
    ...
}

```

此示例显示使用JPA和Spring Data MongoDB注释的域类。它定义了两个仓库，`JpaPersonRepository` 和 `MongoDBPersonRepository`。一个用于JPA，另一个用于MongoDB使用。Spring数据不再能够分辨出存储库导致未定义的行为。

存储库类型详细信息和标识域类注释用于严格的存储库配置，以识别特定Spring数据模块的存储库候选。在同一个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。

区分资源库的最后一个方法是定义库基础包。基本包定义了扫描存储库接口定义的起点，这意味着存储库定义位于相应的包中。默认情况下，注释驱动的配置使用配置类的包。基于XML的配置中的基本包是强制性的。

示例12. 基本包的注释驱动配置

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

JAVA

7.4. 定义查询方法

存储库代理有两种方法从方法名称中导出特定于存储的查询。它可以直接从方法名称导出查询，或通过使用手动定义的查询。可用选项取决于实际存储。但是，必须有一个策略来决定创建什么实际的查询。我们来看看可用的选项。

7.4.1. 查询策略

以下策略可用于存储库基础架构来解决查询。在配置 `query-lookup-strategy` XML的情况下，您可以通过属性配置命名空间中的策略，也可以通过 `queryLookupStrategy` Java配置中启用 `$ {store}` 存储库注释的属性来配置策略。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。一般的方法是从方法名称中删除一组已知的前缀，并解析该方法的其余部分。详细了解查询创建中的查询构造。
- **USE_DECLARED_QUERY** 尝试找到一个声明的查询，并将抛出一个异常，以防万一找不到它。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。请参阅特定商店的文档以查找该商店的可用选项。如果存储库基础架构在引导时没有找到方法的声明查询，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，如果没有找到声明的查询，它将创建一个基于名称的自定义查询。这是默认的查找策略，因此如果您没有明确配置任何内容。它允许通过方法名称快速查询定义，但也可以根据需要引入声明的查询来定制这些查询。

7.4.2. 查询创建

构建在Spring数据存储库基础架构中的查询生成器机制对于在存储库的实体上构建约束查询很有用。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。引入子句可以包含其他表达式，例如在 `Distinct` 要创建的查询上设置不同的标志。然而，第一个 `By` 作为分隔符来指示实际标准的开始。在一个非常基本的水平，你可以定义实体性条件，并与它们串联 `And` 和 `Or`。

示例13. 从方法名称创建查询

JAVA

```
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创建查询的持久性存储。但是，有一些一般的事情要注意。

- 表达式通常是可以连接的运算符的属性遍历。您可以使用组合属性表达式 **AND** 和 **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。受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。
- 该方法解析器支持设置一个 **IgnoreCase** 标志个别特性（例如，**findByLastnameIgnoreCase(...)**）或对于支持忽略大小写（通常是一个类型的所有属性 **String** 情况下，例如，**findByLastnameAndFirstnameAllIgnoreCase(...)**）。是否支持忽略案例可能会因存储而异，因此请参阅参考文档中相关章节，了解特定于商店的查询方法。
- 您可以通过 **OrderBy** 在引用属性和提供排序方向（**Asc** 或 **Desc**）的查询方法中附加一个子句来应用静态排序。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，如前面的例子所示。在查询创建时，您已经确保已解析属性是受管域类的属性。但是，您还可以通过遍历嵌套属性来定义约束。假设一个 **Person** 有 **Address** 一个 **ZipCode**。在这种情况下，方法名称为

JAVA

```
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，并使用该名称（**uncapitalized**）检查域类的属性。如果算法成功，则使用该属性。如果不是，则算法拆分了从右侧的驼峰部分的信号源到头部和尾部，并试图找出相应的属性，在我们的例子，**AddressZip** 和 **Code**。如果算法找到一个具有该头部的属性，那么它需要尾部，并从那里继续构建树，然后按照刚刚描述的方式将尾部分割。如果第一个分割不匹配，则算法将分割点移动到左（**Address**，**ZipCode**），然后继续。

虽然这在大多数情况下都适用，但算法可能会选择错误的属性。假设 **Person** 该类也有一个 **addressZip** 属性。该算法将在第一个分割轮中匹配，并且基本上选择了错误的属性，最终失败（因为该类型 **addressZip** 可能没有 **code** 属性）。

要解决这个歧义，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如此：

JAVA

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


当我们将下划线视为保留字符时，我们强烈建议遵循标准的Java命名约定（即不使用属性名称中的下划线，而是使用骆驼案例）。

7.4.4. 特殊参数处理

要处理查询中的参数，您只需定义方法参数，如上述示例中所示。此外，基础设施将会识别某些特定类型，`Pageable` 并动态 `Sort` 地将分页和排序应用于查询。

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);
```

JAVA

第一种方法允许您将 `org.springframework.data.domain.Pageable` 实例传递给查询方法，以动态地将分页添加到静态定义的查询中。A `Page` 知道可用的元素和页面的总数。它通过基础设施触发计数查询来计算总数。由于这可能是昂贵的，这取决于所使用的商店，`Slice` 可以用作返回。A `Slice` 只知道是否有下一个 `Slice` 可用的，当走过较大的结果集时可能只是足够的。

排序选项也通过 `Pageable` 实例处理。如果只需要排序，只需在 `org.springframework.data.domain.Sort` 参数中添加一个参数即可。正如你也可以看到的，只需返回一个 `List` 也是可能的。在这种情况下，`Page` 不会创建构建实际实例所需的附加元数据（这反过来意味着必须不发布的附加计数查询），而只是简单地限制查询仅查找给定范围的实体。



要查找完整查询的页面数量，您必须触发额外的计数查询。默认情况下，此查询将从您实际触发的查询派生。

7.4.5. 限制查询结果

的查询方法的结果可以通过关键字来限制 `first` 或 `top`，其可以被可互换地使用。可选的数值可以追加到顶部/第一个以指定要返回的最大结果大小。如果数字被省略，则假设结果大小为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);
```

JAVA

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，支持将结果包装到一个实例中 `Optional`。

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），则在限制结果中应用。



请注意，通过参数将结果限制为动态排序，`Sort` 可以表示最小的“K”以及“K”最大元素的查询方法。

7.4.6. 流式查询结果

可以通过使用Java 8 `Stream<T>` 作为返回类型来逐步处理查询方法的结果。而不是简单地将查询结果包装在 `Stream` 数据存储中，特定的方法用于执行流。

示例16. 使用Java 8流式传输查询的结果 `Stream<T>`

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

Stream<User> readAllByFirstnameNotNull();

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

JAVA



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` 使用该 `close()` 方法或使用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(...);
}
```

JAVA



目前并不是所有的Spring数据模块都支持 `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 (<http://docs.spring.io/spring/docs/current/spring-framework-reference/html#scheduling>)执行功能异步执行存储库查询。这意味着方法将在调用时立即返回，并且实际的查询执行将发生在已提交给Spring `TaskExecutor`的任务中。

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

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

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

JAVA

- ❶ 使用 `java.util.concurrent.Future` 的返回类型。
- ❷ 使用Java 8 `java.util.concurrent.CompletableFuture` 作为返回类型。
- ❸ 使用Java 8 `java.util.concurrent.ListenableFuture` 作为返回类型。

- 使用 `org.springframework.util.concurrent.ListenableFuture` 作为返回类型。

7.5. 创建存储库实例

在本节中，您将定义的存储库接口创建实例和bean定义。一种方法是使用支持存储库机制的每个Spring数据模块附带的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，它允许您简单地定义Spring为您扫描的基础包。

示例18. 通过XML启用Spring数据存储库

```
<?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>
```

在上述示例中，指示Spring扫描 `com.acme.repositories` 其所有接口扩展的子包 `Repository` 或其子接口之一。对于发现的每个接口，基础架构注册特定 `FactoryBean` 于持久性技术，以创建处理查询方法调用的适当代理。每个bean都是从接口名称导出的bean名称下注册的，所以接口 `UserRepository` 将被注册 `userRepository`。该 `base-package` 属性允许通配符，以便您可以定义扫描包的模式。

使用过滤器

默认情况下，基础设施拾取每个接口 `Repository`，扩展位于配置的基础包下面的持久性技术特定子接口，并为其创建一个bean实例。但是，您可能需要对要为其创建哪些接口bean实例进行更细粒度的控制。要做到这一点，您使用 `<include-filter />` 和 `<exclude-filter />` 元素里面 `<repositories />`。语义与Spring的上下文命名空间中的元素完全相同。有关详细信息，请参阅有关这些元素的[Spring参考文档](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html#beans-scanning-filters)

(<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html#beans-scanning-filters>)。

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>
```

此示例排除所有 `SomeRepository` 从实例化开始的接口。

7.5.2. JavaConfig

也可以使用 `@Enable${store}Repositories` `JavaConfig`类上的特定于商店的注释触发存储库基础架构。有关Spring容器的基于Java的配置的介绍，请参阅参考文档。^[1]

启用Spring Data存储库的示例配置看起来像这样。

示例20. 基于样本注释的存储库配置

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

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

JAVA



该示例使用JPA特定的注释，您可以根据实际使用的存储模块进行更改。这同样适用于 `EntityManagerFactory` bean的定义。请参阅涵盖商店特定配置的部分。

7.5.3. 独立使用

您还可以使用Spring容器外部的存储库基础架构，例如在CDI环境中。您在类路径中仍然需要一些Spring库，但通常可以通过编程方式设置存储库。提供存储库支持的Spring数据模块提供了一个持久性技术特定的RepositoryFactory，可以使用如下所示的RepositoryFactory。

示例21. 存储库工厂的独立使用

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

JAVA

7.6. Spring数据存储库的自定义实现

通常有必要为几个存储库方法提供自定义实现。Spring数据存储库可以轻松地允许您提供自定义存储库代码，并将其与通用CRUD抽象和查询方法功能集成。

7.6.1. 将自定义行为添加到单个存储库

要使用自定义功能丰富资源库，您首先要定义一个界面和自定义功能的实现。使用您提供的存储库接口来扩展自定义界面。

示例22. 自定义存储库功能的接口

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

JAVA

示例23. 自定义存储库功能的实现

```
class UserRepositoryImpl implements UserRepositoryCustom {  
  
    public void someCustomMethod(User user) {  
        // Your custom implementation  
    }  
}
```

JAVA



找到的类的最重要的一点是 `Impl` 与核心存储库接口相比的名称的后缀（见下文）。

实现本身并不依赖于Spring Data，而且可以是一个常规的Spring bean。因此，您可以使用标准依赖注入行为来注入其他bean的引用，如a `JdbcTemplate`，参与方面等等。

示例24.更改您的基本存储库界面

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

JAVA

让您的标准存储库界面扩展自定义库。这样做结合了CRUD和自定义功能，并将其提供给客户端。

组态

如果使用命名空间配置，存储库基础设施会尝试自动检测自定义实现，方法是扫描我们找到存储库的包下面的类。这些类需要遵循将命名空间元素的属性附加 `repository-impl-postfix` 到找到的存储库接口名称的命名约定。此后缀默认为 `Impl`。

示例25.配置示例

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

XML

第一个配置示例将尝试查找一个类 `com.acme.repository.UserRepositoryImpl` 作为自定义存储库实现，而第二个示例将尝试查找 `com.acme.repository.UserRepositoryFooBar`。

手动接线

如果您的自定义实现仅使用基于注释的配置和自动布线，那么刚才显示的方法效果很好，因为它将被视为任何其他Spring bean。如果您的自定义实现bean需要特殊布线，那么您只需简单地声明该bean并将其命名为刚刚描述的约定。然后，基础设施将通过名称引用手动定义的bean定义，而不是创建一个本身。

示例26.自定义实现的手动接线

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

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

XML

7.6.2. 将自定义行为添加到所有存储库

当您想将一个方法添加到所有的存储库接口时，上述方法是不可行的。要将自定义行为添加到所有存储库，您首先添加一个中间界面来声明共享行为。

示例27. 声明定制共享行为的界面

```
@NoRepositoryBean
public interface MyRepository<T, ID extends Serializable>
    extends PagingAndSortingRepository<T, ID> {

    void sharedCustomMethod(ID id);
}
```

JAVA

现在，您的各个存储库接口将扩展此中间接口，而不是扩展 `Repository` 接口以包含声明的功能。接下来，创建扩展了持久性技术特定的存储库基类的中间接口的实现。然后，该类将用作存储库代理的自定义基类。

自定义库基础类

```
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
    }
}
```

JAVA



该类需要具有专门的存储库工厂实现使用的超级类的构造函数。如果存储库基类有多个构造函数，则覆盖一个 `EntityInformation` 加上特定于存储的基础架构对象（例如，一个 `EntityManager` 或一个模板类）。

Spring `<repositories />` 命名空间的默认行为是为所有接下来的接口提供一个实现 `base-package`。这意味着如果保持当前状态，`MyRepository` Spring将创建一个实现实例。这当然是不希望的，因为它只是作为一个中介，`Repository` 以及您想为每个实体定义的实际存储库接口。要排除 `Repository` 从被实例化为存储库实例的接口，您可以使用 `@NoRepositoryBean`（如上所示）注释它，或将其移动到已配置的外部 `base-package`。

最后一步是使Spring数据基础架构了解定制的库基类。在JavaConfig中，这是通过使用 `repositoryBaseClass` 注释的 `@Enable...Repositories` 属性来实现的：

示例29. 使用JavaConfig配置自定义存储库基类

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

JAVA

相应的属性在XML命名空间中可用。

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" />
```

XML

7.7. 从聚集根源出版事件

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` 您可以使用聚合根的方法来使该发布尽可能简单。

示例31. 从聚合根中暴露域事件

```
class AnAggregateRoot {

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

    @AfterDomainEventsPublication ❷
    void callbackMethod() {
        // ... potentially clean up domain events list
    }

}
```

JAVA

- ❶ 使用的方法 `@DomainEvents` 可以返回单个事件实例或事件集合。它不能采取任何论据。
- ❷ 在所有事件发布之后，注释了一个方法 `@AfterDomainEventsPublication`。它可以用于潜在地清理要发布的事件列表。

每次调用Spring数据存储库的 `save(...)` 方法之一时，都会调用这些方法。

7.8. Spring数据扩展

本节介绍一组Spring数据扩展，可以在各种上下文中启用Spring数据使用。目前大部分的集成针对的是Spring MVC。

7.8.1. Querydsl扩展

Querydsl (<http://www.querydsl.com/>)是一个框架，可以通过流畅的API构建静态类型的类SQL查询。

几个Spring数据模块提供与Querydsl的集成 `QueryDslPredicateExecutor`。

示例32. QueryDslPredicateExecutor 接口

```
public interface QueryDslPredicateExecutor<T> {  
  
    T findOne(Predicate predicate);           ❶  
  
    Iterable<T> findAll(Predicate predicate);  ❷  
  
    long count(Predicate predicate);          ❸  
  
    boolean exists(Predicate predicate);       ❹  
  
    // ... more functionality omitted.  
}
```

JAVA

- ❶ 查找并返回一个匹配的实体 Predicate 。
- ❷ 查找并返回匹配的所有实体 Predicate 。
- ❸ 返回匹配的实体数 Predicate 。
- ❹ 如果匹配的实体 Predicate 存在则返回。

要使用Querydsl支持，只需 QueryDslPredicateExecutor 在您的存储库界面上扩展。

实例33. 在存储库上进行Querydsl整合

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

JAVA

以上使用Querydsl可以编写类型安全的查询 Predicate 。

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

JAVA

7.8.2. 网络支持



本部分包含Spring数据Web支持的文档，因为它在1.6范围内与Spring Data Commons相同。由于新引入的支持更改了很多事情，因此我们保留了旧版Web支持中前一行文的文档。

如果模块支持仓库编程模型，Spring数据模块将附带各种Web支持。Web相关的东西在类路径上需要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 { }
```

JAVA

该 `@EnableSpringDataWebSupport` 批注册几个组件，我们将在一个位讨论。它还将在类路径上检测Spring HATEOAS，并注册集成组件（如果存在）。

或者，如果您使用XML配置，请注册 `SpringDataWebSupport` 或 `HateoasAwareSpringDataWebSupport` 作为Spring bean:

示例35. 启用XML中的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.HateoasAwareSpringDataWebConfiguration" />
```

XML

基本网络支持

上述配置设置将注册几个基本组件:

- `DomainClassConverter` 使Spring MVC能够从请求参数或路径变量解析存储库管理域类的实例。
- `HandlerMethodArgumentResolver` 实现让Spring MVC从请求参数中解析Pageable和Sort实例。

DomainClassConverter

将 `DomainClassConverter` 让你在您的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. Spring MVC控制器在方法签名中使用域类型

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

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

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

JAVA

您可以看到该方法直接接收User实例，不需要进一步的查找。实例可以通过让Spring MVC首先将路径变量转换为域类型，最终通过调用 `findOne(...)` 注册为域类型的存储库实例来访问该实例来解决。



目前，资源库 `CrudRepository` 必须实施才能被发现以进行转换。

HandlerMethodArgumentResolvers for Pageable和Sort

上面的配置代码片段也注册了一个 `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

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";
    }
}

```

此方法签名将导致Spring MVC尝试使用以下默认配置从请求参数派生一个Pageable实例:

表1. 为Pageable实例评估的请求参数

page	您要检索的页面，0已编入索引并默认为0。
size	您要检索的页面的大小，默认为20。
sort	应以格式排序的属性 property,property(,ASC DESC) 。默认排序方向是上升。 sort 如果要切换路线，请使用多个参数，例如 ?sort=firstname&sort=lastname,asc 。

要自定义此行为可扩展 SpringDataWebConfiguration 或启用HATEOAS启用的等效项，并覆盖 pageableResolver() 或 sortResolver() 方法并导入自定义配置文件，而不是使用 @Enable -annotation。

如果您需要从请求中解析多个 Pageable 或 Sort 实例（例如，对于多个表），则可以使用Spring的 @Qualifier 注释来区分出来。然后请求参数必须加上前缀 \${qualifier}_ 。所以对于像这样的方法签名:

```

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

```

你有填充 foo_page 和 bar_page 等。

该 Pageable 方法的默认值相当于一个， new PageRequest(0, 20) 但可以使用 @PageableDefaults 参数上的 Pageable 注释进行自定义。

超媒体支持页面

Spring HATEOAS提供了一个表示模型类 PagedResources ，它允许 Page 使用必要的 Page 元数据丰富实例的内容，以及链接，让客户端轻松浏览页面。 PagedResources 通过Spring HATEOAS ResourceAssembler 接口的实现来完成页面的转换 PagedResourcesAssembler 。

示例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)
    ResponseEntity<PagedResources<Person>> persons(Pageable pageable,
        PagedResourcesAssembler assembler) {

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

```

JAVA

启用如上所示的配置允许将 `PagedResourcesAssembler` 其用作控制器方法参数。打电话 `toResources(...)` 会导致以下情况：

- 将内容的内容 `Page` 变成 `PagedResources` 实例的内容。
- 该 `PagedResources` 会得到一个 `PageMetadata` 附加填充信息形成的实例 `Page` 和基础 `PageRequest`。
- 根据页面的状态 `PagedResources` 获取 `prev` 和 `next` 链接。链接将指向被调用的方法映射到的URI。添加到方法中的分页参数将匹配设置，`PageableHandlerMethodArgumentResolver` 以确保以后可以解析链接。

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
  }
}

```

JAVASCRIPT

您会看到汇编程序产生了正确的URI，并且还提取了存在的默认配置，以将参数解析 `Pageable` 为即将到来的请求。这意味着，如果您更改该配置，链接将自动遵守更改。默认情况下，汇编程序指向被调用的控制器方法，但可以通过将自定义 `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

对于具有[QueryDSL](http://www.querydsl.com/) (http://www.querydsl.com/)集成的商店，可以从 `Request` 查询字符串中包含的属性中导出查询。

这意味着给定 `User` 前一个样本的对象一个查询字符串

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

TEXT

可以解决

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

使用 `QuerydslPredicateArgumentResolver` 。



`@EnableSpringDataWebSupport` 当类路径上找到 `Querydsl` 时，该功能将自动启用。

添加 `@QuerydslPredicate` 到方法签名将提供一个可以使用的 `Predicate`，可以通过执行 `QueryDslPredicateExecutor` 。



类型信息通常从方法返回类型中解析出来。由于这些信息不一定与域类型相匹配，所以使用 `root` 属性可能是一个好主意 `QuerydslPredicate` 。

JAVA

```
@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 解析查询字符串参数匹配 `Predicate` 的 `User` 。

默认绑定如下：

- `Object` 简单的属性为 `eq` 。
- `Object` 收集像属性一样 `contains` 。
- `Collection` 简单的属性为 `in` 。

这些绑定可以通过 `bindings` 属性 `@QuerydslPredicate` 或通过使用Java 8 `default methods` 添加 `QuerydslBinderCustomizer` 到存储库接口进行定制。

JAVA

```

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

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

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

```

- ❶ QueryDslPredicateExecutor 提供对特定查找器方法的访问 Predicate 。
- ❷ QuerydslBinderCustomizer 在存储库界面上定义将自动拾取和快捷方式 @QuerydslPredicate(bindings=...) 。
- ❸ 将 username 属性的绑定定义为简单的包含绑定。
- ❹ 将属性的默认绑定定义 String 为不区分大小写的包含匹配。
- ❺ 从解决方案中排除 密码属性 Predicate 。

7.8.3. 存储库populator

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。尽管它不使用SQL作为数据定义语言，但存储库级别可以使用类似的抽象，因为它必须与存储无关。因此，populator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。

假设你有一个 data.json 包含以下内容的文件：

示例39. JSON中定义的数据

JAVASCRIPT

```

[ { "_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元素轻松填充您的存储库。要将上述数据填充到您的PersonRepository，请执行以下操作：

示例40. 声明Jackson存储库填充程序

```
<?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>
```

XML

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

JSON对象将被解组的类型将通过检查 `_class` JSON文档的属性来确定。基础设施将最终选择适当的存储库来处理刚被反序列化的对象。

要使用XML来定义数据库，应该使用这些 `unmarshaller-populator` 元素来填充这些数据库。您可以将其配置为使用Spring OXM为您提供的一种XML编组器选项。有关详细信息，请参阅[Spring参考文档](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/oxm.html) (<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/oxm.html>)。

示例41. 声明一个解组的存储库填充程序（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>
```

XML

7.8.4. 传统网络支持

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，您的任务是将请求参数或URL部分转换为域类，将其转交到下面的层，然后直接对实体执行业务逻辑。这看起来像这样：

JAVA

```

@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，您声明每个控制器的存储库依赖关系，以分别查找由控制器或存储库管理的实体。查看实体也是样板，因为它总是 `findOne(...)` 打电话。幸运的是，Spring提供了注册自定义组件的方法，允许在 `String` 值与任意类型之间进行转换。

属性编辑器

对于3.0之前的Spring版本 `PropertyEditors`，必须使用简单的Java。要与之进行整合，Spring Data提供了一个 `DomainClassPropertyEditorRegistrar` 查找所有在其中注册的Spring数据存储库 `ApplicationContext` 并注册 `PropertyEditor` 管理域类的自定义。

XML

```

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

```

如果您已经配置了上述示例中的Spring MVC，则可以按如下方式配置控制器，从而减少了大量杂乱和样板。

JAVA

```

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

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

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

```

参考文献

8.介绍

8.1。文件结构

参考文档的这一部分解释了Spring Data MongoDB提供的核心功能。

MongoDB支持介绍MongoDB模块功能集。

MongoDB存储库介绍了MongoDB的存储库支持。

9. MongoDB支持

MongoDB支持包含以下概述的各种功能。

- 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助手类可以提高执行常见Mongo操作的效率。包括文档和POJO之间的集成对象映射。
- 异常翻译成Spring的便携式数据访问异常层次结构
- 与Spring的转换服务集成功能丰富的对象映射
- 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
- 自动实现存储库界面，包括支持自定义查找器方法。
- QueryDSL集成支持类型安全查询。
- 跨存储持久性 - 支持使用MongoDB透明地保留/检索的字段JPA实体
- Log4j日志追加器
- 地理空间整合

对于大多数任务，您将发现自己使用 `MongoTemplate` 或者使用 `Repository` 支持来利用丰富的映射功能。 `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` 还提供回调方法，以便您轻松掌握低级API工件， `com.mongodb.DB` 以便直接与MongoDB通信。在各种API工件上使用命名约定的目标是复制基础MongoDB Java驱动程序中的内容，以便您可以轻松将现有知识映射到Spring API上。

9.1. 入门

Spring MongoDB支持需要MongoDB 2.6或更高版本，Java SE 6或更高版本。引导设置工作环境的简单方法是在[STS](https://spring.io/tools/sts) (<https://spring.io/tools/sts>)中创建一个基于Spring的项目。

首先你需要设置一个运行的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](http://docs.mongodb.org/manual/core/introduction/) (<http://docs.mongodb.org/manual/core/introduction/>)。一旦安装，MongoDB通常会执行以下命令：

令： `MONGO_HOME/bin/mongod`

要在STS中创建一个Spring项目，请转到File→New→Spring Template Project→Simple Spring Utility Project→按提示时按Yes。然后输入项目和包名称，如`org.springframework.mongodb.example`。

然后将以下内容添加到pom.xml依赖关系部分。

```
<dependencies>

  <!-- other dependency elements omitted -->

  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-mongodb</artifactId>
    <version>{version}</version>
  </dependency>

</dependencies>
```

XML

还要改变pom.xml中的Spring版本

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

XML

您还需要将maven的Spring Milestone存储库的位置添加 pom.xml 到与 <dependencies/> 元素位于同一级别的位置

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

XML

存储库也可以[在这里浏览](http://repo.spring.io/milestone/org/springframework/data/) (http://repo.spring.io/milestone/org/springframework/data/)。

您可能还需要将日志记录级别设置 DEBUG 为查看一些其他信息，编辑该 log4j.properties 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类来持久化:

```
package org.springframework.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 + "];"
    }
}
```

JAVA

并运行一个主要应用程序

```

package org.springframework.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.springframework.example.Person for index information.
10:01:32,265 DEBUG ramework.data.mongodb.core.MongoTemplate: 631 - insertDBObject 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.springframework.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，也有几件事情要注意

- 您可以 `MongoTemplate` 使用标准 `com.mongodb.Mongo` 对象和要使用的数据库的名称实例化Spring Mongo的中心助手类。
- 映射程序可以针对标准的POJO对象，而不需要任何额外的元数据（尽管可以选择提供该信息，参见此处）。
- 约定用于处理id字段，将其转换为 `ObjectId` 存储在数据库中时。
- 映射约定可以使用字段访问。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相匹配，则它们将用于实例化对象

9.2. 示例存储库

有一个[github仓库](https://github.com/spring-projects/spring-data-examples)，有几个例子 (<https://github.com/spring-projects/spring-data-examples>)，您可以下载和播放，以了解图书馆的工作原理。

9.3. 用Spring连接到MongoDB

使用MongoDB和Spring的首要任务之一是 `com.mongodb.Mongo` 使用IoC容器创建一个对象。使用基于Java的bean元数据或基于XML的bean元数据有两种主要方法。这些将在以下部分中讨论。



对于不熟悉如何使用基于Java的bean元数据而不是基于XML的元数据配置Spring容器的人员，请参阅[此处](http://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/new-in-3.0.html#new-java-configuration) (http://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/new-in-3.0.html#new-java-configuration) 参考文档中的高级介绍以及[此处](http://docs.spring.io/spring/docs/4.3.8.RELEASE/spring-framework-reference/html/beans.html#beans-java-instantiating-container) (http://docs.spring.io/spring/docs/4.3.8.RELEASE/spring-framework-reference/html/beans.html#beans-java-instantiating-container) 的详细文档。

9.3.1. 使用基于Java的元数据注册Mongo实例

使用基于Java的bean元数据来注册一个实例的 `com.mongodb.Mongo` 例子如下所示

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");
    }
}
```

JAVA

这种方法允许您使用 `com.mongodb.Mongo` 可能已被使用的标准API，但也会使用UnknownHostException检查异常来污染代码。使用检查的异常是不可取的，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，使调用代码混乱。

另一种方法是 `com.mongodb.Mongo` 使用Spring 来注册一个实例的容器 `MongoClientFactoryBean`。与 `com.mongodb.Mongo` 直接实例化实例相比，FactoryBean方法不会抛出被检查的异常，并且还具有向容器提供ExceptionTranslator实现的另外一个优点，该实现将MongoDB异常转换为Spring的便携式DataAccessException层次结构中的异常，用于注释注释的数据访问类@Repository。@Repository在Spring的DAO支持功能 (http://docs.spring.io/spring/docs/4.3.8.RELEASE/spring-framework-reference/html/dao.html)中描述了这种层次结构和使用。

在@Repository注释类上支持异常翻译的基于Java的bean元数据的示例如下所示：

示例43. 使用Spring的MongoClientFactoryBean注册com.mongodb.Mongo对象，并启用Spring的异常翻译支持

```
@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;
    }
}
```

JAVA

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命名空间来注册容器的实例 `com.mongodb.Mongo`，但XML通常可以是冗长的。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模式：

示例44. 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"/>
```

XML

9.3.3. MongoClientFactory界面

当 `com.mongodb.Mongo` MongoDB驱动程序API的入口点时，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，您可以获取一个`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.MongoClientFactory` 如下所示的界面来引导与数据库的连接。

```
public interface MongoClientFactory {  
  
    DB getDb() throws DataAccessException;  
  
    DB getDb(String dbName) throws DataAccessException;  
}
```

JAVA

以下部分显示如何使用容器与Java或基于XML的元数据配置接口的实例 `MongoClientFactory`。反过来，您可以使用 `MongoClientFactory` 实例进行配置 `MongoTemplate`。

该类 `org.springframework.data.mongodb.core.SimpleMongoClientFactory` 提供实现 `MongoClientFactory` 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。

不用使用IoC容器创建`MongoTemplate`的实例，您可以使用标准的Java代码，如下所示。

```
public class MongoApp {  
  
    private static final Log log = LoggerFactory.getLog(MongoApp.class);  
  
    public static void main(String[] args) throws Exception {  
  
        MongoOperations mongoOps = new MongoTemplate(*new SimpleMongoClientFactory(new Mongo(), "database");  
  
        mongoOps.insert(new Person("Joe", 34));  
  
        log.info(mongoOps.findOne(new Query(where("name").is("Joe")), Person.class));  
  
        mongoOps.dropCollection("person");  
    }  
}
```

JAVA

粗体中的代码强调了`SimpleMongoClientFactory`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MongoClientFactory实例

要使用容器注册`MongoClientFactory`实例，您可以像前面的代码列表中突出显示的那样编写代码。一个简单的例子如下所示

JAVA

```
@Configuration
public class MongoConfiguration {

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

要定义用户名和密码，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。此列表还显示了使用 `MongoDbFactory` 注册一个 `MongoTemplate` 实例与容器。

JAVA

```
@Configuration
public class MongoConfiguration {

    public @Bean MongoClientFactory mongoDbFactory() throws Exception {
        UserCredentials userCredentials = new UserCredentials("joe", "secret");
        return new SimpleMongoDbFactory(new MongoClient(), "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`命名空间提供了一种创建方便的方法。使用简单如下

XML

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

在上述示例 `com.mongodb.Mongo` 中，使用默认主机和端口号创建实例。使用 `SimpleMongoDbFactory` `id='mongoDbFactory'`来标识容器的注册，除非指定了`id`属性的值。

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

XML

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

如果您的MongoDB认证数据库与目标数据库不同，请使用该 `authentication-database` 属性，如下所示。

XML

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

如果需要在 `com.mongodb.Mongo` 用于创建的实例上配置其他选项，`SimpleMongoDbFactory` 则可以使用 `mongo-ref` 如下所示的属性引用现有的bean。为了显示另一个常见的使用模式，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>
```

XML

9.4. MongoTemplate简介

该类 `MongoTemplate` 位于包中 `org.springframework.data.mongodb.core`，是Spring的MongoDB支持的中心类，提供了与数据库交互的丰富功能集。该模板提供了方便的操作来创建，更新，删除和查询MongoDB文档，并提供域对象和MongoDB文档之间的映射。



一旦配置，`MongoTemplate` 线程安全，可以跨多个实例重用。

MongoDB文档和域类之间的映射是通过委托接口的实现完成的 `MongoConverter`。Spring提供了两种实现方式，`SimpleMappingConverter` 以及 `MappingMongoConverter`，但您也可以编写自己的转换器。有关详细信息，请参阅MongoConverters部分。

本 `MongoTemplate` 类实现了接口 `MongoOperations`。尽可能地，这些方法以 `MongoOperations` MongoDB驱动程序 `Collection` 对象上可用的方法命名，使API熟悉现有的用于驱动程序API的MongoDB开发人员。例如，您可以找到诸如“find”，“findAndModify”，“findOne”，“insert”，“remove”，“save”，“update”和“updateMulti”等方法。设计目标是尽可能简单地在基本的MongoDB驱动程序之间进行转换 `MongoOperations`。在两个API之间的主要区别在于，`MongoOperations`可以通过域对象，而不是 `DBObject` 和有助于流利的API `Query`，`Criteria`，`Update` `DBObject`



引用操作的首选方式 `MongoTemplate` 是通过它的接口 `MongoOperations`。

使用的默认转换器实现 `MongoTemplate` 是 `MappingMongoConverter`。虽然 `MappingMongoConverter` 可以使用额外的元数据来指定对象到文档的映射，但是它也能够通过使用一些用于映射ID和集合名称的约定来转换不包含附加元数据的对象。映射章节将说明这些约定以及映射注释的使用。



在M2版本中 `SimpleMappingConverter`，是默认的，现在这个类已经被弃用了，因为它的功能被归纳了 `MappingMongoConverter`。

`MongoTemplate`的另一个核心功能是将MongoDB Java驱动程序中抛出的异常异常转换为Spring的便携式数据访问异常层次结构。有关更多信息，请参阅异常翻译部分。

尽管 `MongoTemplate` 如果您需要直接访问MongoDB驱动程序API以访问`MongoTemplate`未显式公开的功能，那么您可以使用多种方便的方法来轻松执行常见任务，您可以使用多个执行回调方法之一访问底层驱动程序API。执行回调函数将给你一个 `com.mongodb.Collection` 或一个 `com.mongodb.DB` 对象的引用。有关详细信息，请参阅部分 `mongo.executioncallback` [执行回调]。

现在来看一下如何 `MongoTemplate` 在Spring容器的上下文中使用的例子。

9.4.1. 实例化MongoTemplate

您可以使用Java创建和注册一个 `MongoTemplate` 如下所示的实例。

示例47. 注册com.mongodb.Mongo对象并启用Spring的异常翻译支持

```
@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");
    }
}
```

JAVA

`MongoTemplate`有几个重载的构造函数。这些是

- `MongoTemplate(Mongo mongo, String databaseName)` - 将 `com.mongodb.Mongo` 对象和默认数据库名称对齐。
- `MongoTemplate(Mongo mongo, String databaseName, UserCredentials userCredentials)` - 添加用于使用数据库进行身份验证的用户名和密码。
- `MongoTemplate(MongoDbFactory mongoDbFactory)` - 使用一个`MongoDbFactory`对象来封装 `com.mongodb.Mongo` 对象，数据库名称以及用户名和密码。
- `MongoTemplate(MongoDbFactory mongoDbFactory, MongoConverter mongoConverter)` - 添加一个 `MongoConverter`用于映射。

您还可以使用Spring的XML `<beans />`模式配置`MongoTemplate`。

```
<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>
```

JAVA

其他的，你可能会喜欢创建时设置可选属性 `MongoTemplate` 是默认 `WriteResultCheckingPolicy`，`WriteConcern` 和 `ReadPreference`。



引用操作的首选方式 `MongoTemplate` 是通过它的接口 `MongoOperations`。

9.4.2. WriteResultChecking策略

在开发过程中，如果 `com.mongodb.WriteResult` 从任何MongoDB操作返回的包含错误，记录或抛出异常非常方便。在开发过程中忘记这样做是很常见的，然后最终会出现一个似乎运行成功的应用程序，但实际上数据库没有根据您的期望进行修改。`MongoTemplate`的属性设置为具有以下值的枚举 `LOG`，`EXCEPTION` 或者 `NONE` 要么在错误日志中抛出和异常或者什么都不做。默认是使用一个 `WriteResultChecking` 值 `NONE`。

9.4.3. WriteConcern

如果尚未通过较高级别的驱动程序 `com.mongodb.WriteConcern` 指定，则可以设置 `MongoTemplate` 将用于写入操作的属性 `com.mongodb.Mongo`。如果 `WriteConcern` 未设置 `MongoTemplate`的属性，则默认为MongoDB驱动程序的数据库或集合设置中的一个。

9.4.4. WriteConcernResolver

对于要设置不同的更先进的情况下，`WriteConcern` 对每个操作值（删除，更新，插入和保存操作），称为战略界面 `WriteConcernResolver` 可以上配置 `MongoTemplate`。由于 `MongoTemplate` 用于保留POJO，所以 `WriteConcernResolver` 允许您创建一个可将特定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);  
}
```

JAVA

传入的参数 `MongoAction` `WriteConcern` 是用于确定要使用的值或将模板本身的值用作默认值。`MongoAction` 包含正在写入的集合名称，`java.lang.Class` POJO，转换的集合名称 `DBObject` 以及作为枚举的操作（`MongoActionOperation`: `REMOVE`, `UPDATE`, `INSERT`, `INSERT_LIST`, `SAVE`）和其他一些上下文信息。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。

给出一个简单的类，如 `Person`

```
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` 是实现 `MongoTemplate` 接口。

```
package org.springframework.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` 本身的调试消息）

```

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

```

有 `MongoConverter` 一个使用 `a String` 和 `ObjectId` 存储在数据库中的隐式转换，并识别属性“`Id`”名称的约定。



此示例旨在显示在 `MongoTemplate` 上使用保存，更新和删除操作，而不显示复杂的映射功能

在示例中使用的查询语法进行更详细的部分中解释查询文件。

9.5.1. 如何 `_id` 在映射层中处理该字段

MongoDB 要求您具有 `_id` 所有文档的字段。如果您不提供一个驱动程序将分配 `ObjectId` 一个生成的值。在使用时，`MappingMongoConverter` 有一些规则可以控制 Java 类的属性如何映射到此 `_id` 字段。

以下概述什么属性将映射到 `_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。
- 没有注释但命名的属性或字段 `id` 将映射到 `_id` 字段。

以下概述了在使用 `MappingMongoConverter` 默认值时映射到 `_id` 文档字段的属性将执行什么类型转换（如果有）`MongoTemplate`。

- 在 Java 类中声明为 `String` 的 `id` 属性或字段将被转换并存储为 `ObjectId` 尽可能使用 `Spring Converter<String, ObjectId>`。有效的转换规则被委派给 MongoDB Java 驱动程序。如果不能将其转换为 `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` 属性或字段将被转换并 `ObjectId` 使用 `Spring 存储 Converter<BigInteger, ObjectId>`。

如果 Java 类中没有上面指定的字段或属性 `_id`，那么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。

当查询和更新 `MongoTemplate` 将使用转换器来处理转换的 `Query`，并 `Update` 对应于上述规则保存文档，以便在查询的字段名称和类型将能够匹配什么是在你的领域类的对象。

9.5.2. 类型映射

由于MongoDB集合可以包含代表各种类型的实例的文档。这里的一个很好的例子是，如果您存储一个类的层次结构，或者只需要一个具有类型属性的类 `Object`。在后一种情况下，检索对象时必须正确读取该属性中保存的值。因此，我们需要一种在实际文档旁边存储类型信息的机制。

为了实现这一点，`MappingMongoConverter` 使用 `MongoTypeMapper` 抽象与 `DefaultMongoTypeMapper` 它的主要实现。它的默认行为是将完全限定的类名存储在 `_class` 顶级文档的文档内部，如果它是一个复杂的类型和声明的属性类型的子类型，那么它将为每个值存储。

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" }
}
```

JAVA

正如你可以看到，我们存储实际的根类持久性以及嵌套类型的类型信息，因为它是复杂的，并且是一个子类型 `Contact`。所以如果你现在正在使用，`mongoTemplate.findAll(Object.class, "sample")` 我们可以发现存储的文件应该是一个 `Sample` 实例。我们也可以发现，价值属性应该是一个 `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，而是使用某些键，则可以 `@TypeAlias` 在实体类中持久化使用注释。如果您需要自定义映射，甚至可以看看 `TypeInformationMapper` 界面。该接口的实例可以配置在 `DefaultMongoTypeMapper` 可以配置的情况下 `MappingMongoConverter`。

示例49. 定义实体的类型库

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

JAVA

请注意，生成的文档将包含 `"pers"` 作为 `_class` 字段中的值。

配置自定义类型映射

下面的例子演示了如何配置自定义 `MongoTypeMapper` 在 `MappingMongoConverter`。

示例50. 通过Spring Java Config配置自定义MongoTypeMapper

```
class CustomMongoTypeMapper extends DefaultMongoTypeMapper {
    //implement custom type mapping here
}
```

JAVA

```
@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();
    }
}
```

JAVA

请注意，我们正在扩展 `AbstractMongoConfiguration` 类并覆盖 `MappingMongoConverter` 我们配置自定义的位置的 bean 定义 `MongoTypeMapper`。

示例51. 通过XML配置自定义MongoTypeMapper

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

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

XML

9.5.3. 保存和插入文档的方法

`MongoTemplate` 存储和插入对象有几种方便的方法。要对转换过程进行更细粒度的控制，您可以注册Spring转换器 `MappingMongoConverter`，例如 `Converter<Person, DBObject>` 和 `Converter<DBObject, Person>`。



插入和保存操作之间的区别是如果对象不存在，则保存操作将执行插入。

使用保存操作的简单情况是保存POJO。在这种情况下，集合名称将由类的名称（不完全限定）确定。您也可以使用特定的集合名称调用保存操作。可以使用映射元数据覆盖存储对象的集合。

插入或保存时，如果Id属性未设置，则假定其值将由数据库自动生成。因此，对于自动生成的ObjectId的成功在您的类Id属性/字段的类型必须是 `String`，`ObjectId` 或 `BigInteger`。

这是使用保存操作和检索其内容的基本示例。

示例52. 使用MongoTemplate插入和检索文档

```
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);
```

JAVA

下面列出了可用的插入/保存操作。

- `void` 将 `(Object objectToSave)` 对象保存到默认集合。
- `void` 将 `(Object objectToSave, String collectionName)` 对象保存到指定的集合。

下面列出了类似的插入操作

- `void insert (Object objectToSave)` 将对象插入默认集合。
- `void insert (Object objectToSave, String collectionName)` 将对象插入指定的集合。

我的文件将保存在哪个集合中？

管理用于在文档上操作的集合名称有两种方法。使用的默认集合名称是更改为以小写字母开头的类名称。所以一个 `com.test.Person` 类将被存储在“人”集合中。您可以通过使用 `@Document` 注释提供不同的集合名称来进行自定义。您还可以通过提供您自己的集合名称作为所选 `MongoTemplate` 方法调用的最后一个参数来覆盖集合名称。

插入或保存单个对象

MongoDB驱动程序支持在一个操作中插入文档集合。MongoOperations界面中支持此功能的方法如下所示

- 插入一个对象。如果存在具有相同ID的现有文档，则会生成错误。
- `insertAll` 将一个 `Collection` 对象作为第一个参数。该方法根据上述规则检查每个对象并将其插入相应的集合。
- `save` 保存对象覆盖可能存在相同id的任何对象。

批量插入几个对象

MongoDB驱动程序支持在一个操作中插入文档集合。MongoOperations界面中支持此功能的方法如下所示

- 插入该采取的方法 `Collection` 的第一个参数。这将在对数据库的单次批量写入中插入对象列表。

9.5.4. 更新集合中的文档

对于更新，我们可以选择更新使用 `MongoOperation` 方法找到的第一个文档，`updateFirst` 或者我们可以使用该方法更新发现与查询匹配的所有文档 `updateMulti`。以下是所有SAVINGS帐户更新的示例，我们将使用 `$inc` 操作员添加一次\$50.00的余额。

示例53. 使用MongoTemplate更新文档

JAVA

```
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);
```

除了上述 `Query` 讨论之外，我们还提供使用 `Update` 对象的更新定义。该 `Update` 类有匹配供MongoDB的更新改进剂的方法。

您可以看到大多数方法返回 `Update` 对象以为API提供流畅的样式。

用于执行文档更新的方法

- `updateFirst` 使用提供的更新文档更新与查询文档标准匹配的第一个文档。
- `updateMulti` 使用提供的更新文档更新与查询文档标准匹配的所有对象。

Update类的方法

`Update` 类可以使用一些“语法糖”，因为它的方法意在链接在一起，您可以通过静态方法 `public static Update update(String key, Object value)` 和静态导入来启动创建新的 `Update` 实例。

以下是 `Update` 类上的方法列表

- `Update addToSet (String key, Object value)` 更新使用 `$addToSet` 更新修饰符
- `Update currentDate (String key)` 更新使用 `$currentDate` 更新修饰符
- `Update currentTimestamp (String key)` 更新使用 `$currentDate` 更新修饰符与 `$type timestamp`
- `Update inc (String key, Number inc)` 使用 `$inc` 更新修饰符更新
- `Update max (String key, Object max)` 更新使用 `$max` 更新修饰符
- `Update min (String key, Object min)` 更新使用 `$min` 更新修饰符
- `Update 乘以 (String key, Number multiplier)` 更新使用 `$mul` 更新修饰符
- `Update pop (String key, Update.Position pos)` 更新使用 `$pop` 更新修饰符
- `Update (String key, Object value)` 使用 `$pull` 更新修饰符拉取更新
- `Update pullAll (String key, Object[] values)` `Update` 使用 `$pullAll` 更新修饰符
- `Update (String key, Object value)` 使用 `$push` 更新修饰符推送更新
- `Update (String key, Object[] values)` 使用 `$pushAll` 更新修饰符的 `pushAll` 更新
- `Update (String oldName, String newName)` 使用 `$rename` 更新修饰符重命名更新
- `Update (String key, Object value)` 使用 `$set` 更新修饰符设置更新
- `Update setOnInsert (String key, Object value)` `Update` 使用 `$setOnInsert` 更新修饰符
- `Update (String key)` 使用 `$unset` 更新修饰符取消设置更新

一些更新修饰符喜欢 `$push` 和 `$addToSet` 允许更多的运营商的嵌套。

```
// { $push : { "category" : { "$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. 将文档升级到集合中

与 `updateFirst` 执行操作相关，您还可以执行一个 `upsert` 操作，如果没有找到与查询匹配的文档，它将执行插入。插入的文档是查询文档和更新文档的组合。这是一个例子

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

9.5.6. 在集合中查找和升级文档

`findAndModify(...)` `DBCollection` 中的方法可以更新文档，并在单个操作中返回旧的或新更新的文档。`MongoTemplate` 提供了一个 `findAndModify` 方法，该方法接受 `Query` 并将 `Update` 类转换 `DBObject` 为 `POJO`。以下是方法

```
<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);
```

作为示例使用，我们将几个 `Person` 对象插入容器并执行一个简单的 `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。以下示例展示了以前的代码段

```
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));
```

JAVA

9.5.7. 删除文件的方法

您可以使用多个重载方法从数据库中删除对象。

- 删除基于以下之一删除给定的文档: 特定对象实例, 查询文档标准与类或查询文档标准相结合的特定集合名称。

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以更新查询的方式添加到更新查询中, 如果另一个操作在两者之间更改了文档, 则更新将不会有任何影响。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"));

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

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

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

JAVA

- 1 直接插入文件。 `version` 被设置为 0。
- 2 加载刚才插入的文件 `version` 还是 0。
- 3 更新文件 `version = 0`。设置 `lastname` 和碰撞 `version` 到 1。
- 4 尝试更新之前加载的文件窗台有 `version = 0` 失败, `OptimisticLockingFailureException` 因为当前 `version` 是 1。



使用MongoDB驱动程序版本3需要设置 `WriteConcern` 为 `ACKNOWLEDGED`。否则 `OptimisticLockingFailureException` 可以静静地吞咽。

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创建 `Query` 和 `Criteria` 实例的“new”关键字的需要, 从而提高

可读性。如果您喜欢 `Query` 从纯JSON字符串使用创建实例 `BasicQuery`。

示例54. 从纯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);
```

JAVA

GeoSpatial查询也受支持，并在“地理空间查询”部分中进行了更多描述。

Map-Reduce系列还支持Map-Reduce操作。

9.6.1. 查询集合中的文档

我们看到如何使用 `MongoTemplate` 上的 `findOne` 和 `findById` 方法检索单个文档，以前的部分返回单个域对象。我们还可以查询要作为域对象列表返回的文档的集合。假设我们有一些具有名称和年龄的 `Person` 对象作为文档存储在集合中，并且每个人都有一个具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。

示例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);
```

JAVA

所有查找方法都将 `Query` 对象作为参数。此对象定义用于执行查询的标准和选项。使用 `Criteria` 具有静态工厂方法的对象来指定条件，该对象名称 `where` 用于实例化新 `Criteria` 对象。我们建议使用静态导入的 `org.springframework.data.mongodb.core.query.Criteria.where` 和 `Query.query`，使查询更具可读性。

此查询应返回 `Person` 符合指定条件的对象列表。所述 `Criteria` 类具有对应于MongoDB中提供的运营商使用下列方法。

您可以看到大多数方法返回 `Criteria` 对象以为API提供流畅的样式。

标准类的方法

- `Criteria all` (`Object o`) 使用 `$all` 运算符创建标准
- `Criteria 并` (`String key`) 增加了一个链接 `Criteria` 与指定 `key` 为当前 `Criteria` 和返回新创建的一个
- `Criteria andOperator` (`Criteria... criteria`) 使用 `$and` 运算符创建并查询所有提供的条件（需要MongoDB 2.0或更高版本）
- `Criteria elemMatch` (`Criteria c`) 使用 `$elemMatch` 运算符创建标准
- `Criteria exists` (`boolean b`) 使用 `$exists` 运算符创建条件
- `Criteria gt` (`Object o`) 使用 `$gt` 运算符创建标准
- `Criteria gte` (`Object o`) 使用 `$gte` 运算符创建标准
- `Criteria in` (`Object... o`) 使用 `$in` 运算符为varargs参数创建标准。
- `Criteria in` (`Collection<?> collection`) 使用 `$in` 运算符使用集合创建标准

- `Criteria` 是 (`Object o`) 使用 `$is` 运算符创建标准
- `Criteria lt` (`Object o`) 使用 `$lt` 操作符创建标准
- `Criteria lte` (`Object o`) 使用 `$lte` 操作符创建标准
- `Criteria mod` (`Number value`, `Number remainder`) 使用 `$mod` 运算符创建标准
- `Criteria ne` (`Object o`) 使用 `$ne` 运算符创建标准
- `Criteria nin` (`Object... o`) 使用 `$nin` 操作符创建标准
- `Criteria norOperator` (`Criteria... criteria`) 使用 `$nor` 运算符为所有提供的条件创建或查询
- `Criteria` 不 (`)` 使用 `$not` 影响直接跟随的子句的元运算符创建标准
- `Criteria orOperator` (`Criteria... criteria`) 使用 `$or` 运算符创建或查询所有提供的条件
- `Criteria` 正则表达式 (`String re`) 使用 `$regex` 创建标准
- `Criteria size` (`int s`) 使用 `$size` 运算符创建标准
- `Criteria type` (`int t`) 使用 `$type` 运算符创建条件

地理空间查询的`Criteria`类也有方法。以下是列表，但请查看“地理空间查询”部分，以查看它们的行动。

- `Criteria` 在 (`Circle circle`) 使用 `$geoWithin $center` 运算符创建地理空间标准。
- `Criteria` 在使用操作 (`Box box`) 创建地理空间 `$geoWithin $box` 标准。
- `Criteria inSphere` (`Circle circle`) 使用 `$geoWithin $center` 运算符创建地理空间标准。
- `Criteria` 附近使用操作 (`Point point`) 创建地理空间 `$near` 标准
- `Criteria nearSphere` (`Point point`) 使用 `$nearSphere $center` 操作创建地理空间标准。这仅适用于 MongoDB 1.7及更高版本。
- `Criteria minDistance` (`double minDistance`) 使用该操作创建一个地理空间 `$minDistance` 标准，用于 `$near`。
- `Criteria maxDistance` (`double maxDistance`) 使用该操作创建一个地理空间 `$maxDistance` 标准，用于 `$near`。

该 `Query` 类有用于提供查询选项的一些其他方法。

Query类的方法

- `Query addCriteria` (`Criteria criteria`) 用于向查询添加附加条件
- `Field` 字段 (`)` 用来定义字段被包括在查询结果中
- `Query` 限制 (`int limit`) 用于将返回结果的大小限制为提供的限制（用于分页）
- `Query` 跳过 (`int skip`) 用于跳过结果中提供的文档数量（用于分页）
- `Query` 与 (`Sort sort`) 用于为结果提供定义排序

9.6.2. 方法查询文件

查询方法需要指定将返回的目标类型T，并且它们也将重载一个明确的集合名称，以便对除了返回类型指示的集合以外的集合进行操作的查询。

- 从集合中查找类型T的对象列表的 `findAll` `Query`。

- **findOne**将集合上的临时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。
- 找到将集合上的即席查询的结果映射到指定类型的列表。
- **findAndRemove**将集合上的临时查询的结果映射到指定类型的对象的单个实例。与查询匹配的文档将被返回，并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`。课程中提供了具体的地理空间查询方法 `Criteria`。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，我们将使用从依赖于使用富集的集成测试获取的以下Venue类 `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) + "];"
    }
}
```

JAVA

要查找a内的位置 `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);
```

JAVA

要 `Circle` 使用球面坐标查找场地，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);
```

JAVA

`Box` 可以在以下查询中查找场地

```
//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);
```

JAVA

要查找附近的场地 `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);
```

JAVA

```
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);
```

JAVA

要查找 `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);
```

JAVA

地理附近查询

MongoDB支持查询数据库的地理位置，并在同一时间计算与给定来源的距离。使用地理附近的查询，可以表达如下所示的查询：“查找周围10英里的所有餐馆”。这样做 `MongoOperations` 提供 `geoNear(...)` 了一个 `NearQuery` 参数作为参数以及已经熟悉的实体类型和集合的方法

```
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);
```

JAVA

您可以看到，我们使用 `NearQuery` 构建器API设置查询，以 `Restaurant` 使 `Point` 最大值为10英里的给定周围的所有实例返回。`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段距离。A `Metric` 由乘法器支持，以将给定度量的距离值转换为本地距离。这里显示的样本将考虑10为英里。使用预先制定的度量（英里和公里）之一将自动触发在查询上设置的球形标志。如果你想避免这种情况，只需将普通 `double` 价值换成 `maxDistance(...)`。有关更多信息，请参阅JavaDoc `NearQuery` 和 `Distance`。

地理附近操作返回 `GeoResults` 封装实例的包装器对象 `GeoResult`。包装 `GeoResults` 允许访问所有结果的平均距离。单个 `GeoResult` 对象简单地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 (<http://geojson.org/>)空间数据的GeoJSON (<http://geojson.org/>)和简单（遗留）坐标对。这些格式都可以用于存储以及查询数据。



请参考[MongoDB关于GeoJSON支持的手册](#)

(<http://docs.mongodb.org/manual/core/2dsphere/#geospatial-indexes-store-geojson/>)，了解需求和限制。

GeoJSON类型在域类中

的使用以GeoJSON (<http://geojson.org/>)在域类类型是直线前进。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;  
}
```

JAVA

GeoJSON键入存储库查询方法

使用GeoJSON类型作为资源库查询参数会 `$geometry` 在创建查询时强制运算符的使用。


```

public interface StoreRepository extends CrudRepository<Store, String> {

    List<Store> findByLocationWithin(Polygon polygon); ❶

}

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

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

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

```

- ❶ 使用commons类型的Repository方法定义允许使用GeoJSON和旧格式进行调用。
- ❷ 使用GeoJSON键入 `$geometry` 运算符。
- ❸ 请注意，GeoJSON多边形需要定义一个闭环。
- ❹ 使用旧格式 `$polygon` 运算符。

9.6.5. 全文查询

由于MongoDB 2.6全文查询可以使用 `$text` 运算符执行。方法和具体操作对全文查询是可用的 `TextQuery` 和 `TextCriteria`。在进行全文搜索时，请参考[MongoDB参考资料](http://docs.mongodb.org/manual/reference/operator/query/text/#behavior) (<http://docs.mongodb.org/manual/reference/operator/query/text/#behavior>)的行为和限制。

全文检索

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

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

JAVASCRIPT

coffee cake 根据相关性排序的查询搜索 weights 可以定义和执行为:

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

JAVA

排除搜索词可以直接通过使用术语前缀 - 或使用来完成 notMatching

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

JAVA

按照 TextCriteria.matching 所提供的术语。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"));
```

JAVA

标志 \$caseSensitive 和 \$diacriticSensitive 可以通过相关方法设置 TextCriteria。请注意，这两个可选标志已在MongoDB 3.2中引入，除非明确设置，否则不会将其包含在查询中。

9.7. 按示例查询

9.7.1. 介绍

本章将为您提供“按示例查询”的介绍，并说明如何使用示例。

示例查询（QBE）是一种用户友好的查询技术，具有简单的界面。它允许动态查询创建，并且不需要编写包含字段名称的查询。实际上，按示例查询，根本不需要使用商店特定的查询语言编写查询。

9.7.2. 用法

由示例API查询由三部分组成:

- Probe: 这是具有填充字段的域对象的实际示例。
- ExampleMatcher: ExampleMatcher 载有关于如何匹配特定字段的详细信息。它可以重复使用在多个示例。
- Example: 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来查询数据存储
- 频繁重构域对象，而不用担心破坏现有查询
- 独立于底层数据存储API

限制

- 不支持嵌套/分组属性约束，如 `firstname = ?0 or (firstname = ?1 and lastname = ?2)`
- 只支持对字符串进行启动/包含/结束/正则表达式匹配以及其他属性类型的精确匹配

在开始使用按示例查询之前，您需要有一个域对象。要开始，只需为您的存储库创建一个界面：

示例56. *Sample Person* 对象

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

JAVA

这是一个简单的域对象。你可以用它创建一个 `Example`。默认情况下，具有 `null` 值的字段将被忽略，字符串将使用存储特定的默认值进行匹配。示例可以通过使用 `of` 工厂方法或使用来构建 `ExampleMatcher`。`Example` 是不可变的

示例57. 简单示例

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

JAVA

- 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，使用存储库执行示例。为此，让您的存储库界面扩展 `QueryByExampleExecutor<T>`。以下是该 `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.  
}
```

JAVA

您可以在下面阅读更多关于按示例执行查询。

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，空处理和特定于属性的设置指定自己的默认值 `ExampleMatcher`。

示例59. 具有自定义匹配的示例匹配器

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

JAVA

- ❶ 创建域对象的新实例。
- ❷ 设置属性。
- ❸ 创建一个 `ExampleMatcher` 期望所有值匹配。即使没有进一步的配置，在这个阶段也可以使用。
- ❹ 构造一个新的 `ExampleMatcher` 来忽略属性路径 `lastname`。
- ❺ 构造一个新的 `ExampleMatcher` 来忽略属性路径 `lastname` 并包含空值。
- ❻ 构造一个新 `ExampleMatcher` 的忽略属性路径 `lastname`，包含空值，并使用 `perform suffix` 字符串匹配。
- ❼ `Example` 根据域对象和配置创建新的 `ExampleMatcher`。

默认情况下，`ExampleMatcher` 将期望探针上设置的所有值都匹配。如果要获取匹配任何隐含定义的谓词的结果，请使用 `ExampleMatcher.matchingAny()`。

您可以为各个属性指定行为（例如嵌套属性的“`firstname`”和“`lastname`”，“`address.city`”）。您可以使用匹配的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());  
}
```

JAVA

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

示例61. 使用lambdas配置匹配器选项

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

JAVA

通过 `Example` 使用合并的配置视图创建的查询。默认匹配设置可以在 `ExampleMatcher` 级别设置，而个别设置可以应用于特定的属性路径。设置的设置 `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的优先级高于默认设置。

表2. `ExampleMatcher` 设置范围

设置	范围
空处理	<code>ExampleMatcher</code>
字符串匹配	<code>ExampleMatcher</code> 和物业路径
忽略属性	物业路径
区分大小写	<code>ExampleMatcher</code> 和物业路径
价值转型	物业路径

9.7.4. 执行一个例子

示例62. 使用存储库查询示例

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

public class PersonService {

    @Autowired PersonRepository personRepository;

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

JAVA

一种 `Example` 包含非类型化的 `ExampleSpec` 使用存储库类型及其集合名称。类型 `ExampleSpec` 使用它们的类型作为结果类型和从存储库的集合名称。



当 `null` 在 `ExampleSpec` Spring数据中包含值Mongo使用嵌入式文档匹配而不是点符号属性匹配。这将强制所有属性值的精确文档匹配以及嵌入文档中的属性顺序。

Spring数据MongoDB提供对以下匹配选项的支持：

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' }}
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，这对于批处理，数据聚合以及查询语言不能满足您的需求很有用。

Spring通过为MongoOperations提供方法来简化Map-Reduce操作的创建和执行，从而提供与MongoDB Map的集成。它可以将Map-Reduce操作的结果转换为POJO，还可以与Spring的[资源抽象](#) (<http://docs.spring.io/spring/docs/4.3.8.RELEASE/spring-framework-reference/html/resources.html>)抽象集成。这将让您将JavaScript文件放在文件系统，classpath，http服务器或任何其他Spring Resource实现中，然后通过简单的URI样式语法（例如'classpath: reduce.js ;' 外部化JavaScript代码在文件中通常比嵌入到代码中的Java字符串更为可取。请注意，如果您愿意，您仍然可以将JavaScript代码作为Java字符串传递。

9.8.1. 使用示例

要了解如何执行Map-Reduce操作，请使用本书“MongoDB - 定义指南”中的示例。在这个例子中，我们将创建三个文档，值为[a, b], [b, c]和[c, d]。每个文档中的值与密钥“x”相关联，如下所示。对于这个例子，假设这些文档在名为“jmr1”的集合中。

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

将对每个文档的阵列中每个字母的出现进行计数的地图函数如下所示

JAVA

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

所有文件中每个字母的出现的缩写函数如下所示

JAVA

```
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收集。

```
{ "_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

JAVA

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

上面的代码的输出是

```
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，以及定时和计数统计信息。该 `ValueObject` 班是根本

JAVA

```

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，请使用重载方法，该方法需要一个附加 `MapReduceOptions` 参数。该类 `MapReduceOptions` 具有流畅的API，因此可以以非常紧凑的语法完成附加选项。这里是一个将输出集合设置为“jmr1_out”的示例。请注意，仅设置输出集合将采用REPLACE的默认输出类型。

JAVA

```

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;` 可以用来使语法稍微更紧凑

JAVA

```

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

```

您还可以指定一个查询，以减少将用于进入map-reduce操作的数据集。这将从map-reduce操作的考虑中删除包含[a, b]的文档。

JAVA

```

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，您可以在查询上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的脚本来在服务器上执行JavaScript函数。 `ScriptOperations` 可以访问 `MongoTemplate` 并提供基本的抽象 JavaScript 使用。

9.9.1. 使用示例

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

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

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

JAVA

- ❶ 直接执行脚本，而不在服务器端存储功能。
- ❷ 使用“echo”作为其名称存储脚本。给定的名称标识脚本，并允许稍后调用。
- ❸ 使用提供的参数执行名称为“echo”的脚本。

9.10. 集团业务

作为替代使用的map-reduce进行数据汇总，您可以使用 **group 操作**

(<https://www.mongodb.org/display/DOCS/Aggregation#Aggregation-Group>)这感觉类似于使用SQL的group by查询的风格，所以它可能会觉得更加平易近人与使用的map-reduce。使用组操作确实有一些限制，例如在共享环境中不支持它，并且它将单个BSON对象中的完整结果集返回，因此结果应该小于10,000个密钥。

Spring通过在MongoOperations上提供方法来简化组操作的创建和执行，来与MongoDB的组操作进行集成。它可以将组操作的结果转换为POJO，并且还Spring的**资源抽象**

(<http://docs.spring.io/spring/docs/4.3.8.RELEASE/spring-framework-reference/html/resources.html>)抽象集成。这将让您将JavaScript文件放在文件系统，classpath，http服务器或任何其他Spring Resource实现中，然后通过简单的URI样式语法（例如'classpath: reduce.js'；外部化JavaScript代码在文件中，如果通常更喜欢将它们嵌入到代码中的Java字符串。请注意，如果您愿意，您仍然可以将JavaScript代码作为Java字符串传递。

9.10.1. 使用示例

为了了解组操作如何工作，使用以下示例，这是有些人为了。更实际的例子，请参考“MongoDB - 最终指南”一书。命名集合 **group_test_collection** 与以下行创建。

```
{ "_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 }
```

我们希望通过每一行的唯一字段进行分组，该 **x** 字段将聚合 **x** 出现每个特定值的次数。为了做到这一点，我们需要创建一个包含我们的count变量的初始文档，还有一个reduce函数，它会在遇到每一个变量时增加它。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);
```

JAVA

第一个参数是运行组操作的集合的名称，第二个参数是一个流畅的API，用于通过 `GroupBy` 类指定组操作的属性。在这个例子中，我们只使用 `initialDocument` 和 `reduceFunction` 方法。您还可以指定一个键功能，以及一个终结器作为流畅API的一部分。如果您有多个分组按键，您可以传递一个逗号分隔的键列表。

组操作的原始结果是一个看起来像这样的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”字段下的文档映射到组方法的第三个参数，在这种情况下，`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 + "];"
    }
}
```

JAVA

您还可以 `DBObject` 通过调用类 `getRawResults` 上的方法来获取原始结果 `GroupByResults`。

有一个额外的方法重载的组方法 `MongoOperations` 可以让您指定一个 `Criteria` 对象来选择一个子集的行。使用一个 `Criteria` 对象的示例，使用静态导入的一些语法糖，以及通过Spring资源字符串引用键功能和减少函数javascript文件如下所示。

```
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数据MongoDB为在2.2版本中引入MongoDB的聚合框架提供了支持。

MongoDB文档描述了聚合框架 (<http://docs.mongodb.org/manual/core/aggregation/>)如下：

有关更多信息，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 (<http://docs.mongodb.org/manual/aggregation/>)。

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`

聚合表示MongoDB `aggregate` 操作，并保存汇总管道指令的描述。聚合是通过调用适当的 `newAggregation(...)` 静态工厂方法创建的，该方法将可选输入类旁边的参数 `Aggregation` 列表 `AggregateOperation` 作为参数。

实际的聚合操作是通过将所需输出类作为参数的 `aggregate` 方法执行的 `MongoTemplate`。

- `AggregationOperation`

An `AggregationOperation` 表示MongoDB汇管道操作，并描述了在此聚合步骤中应执行的处理。虽然可以手动创建一个 `AggregationOperation` 建议的方法来构造一个 `AggregateOperation` 是使用 `Aggregate` 该类提供的静态工厂方法。

- `AggregationResults`

`AggregationResults` 是集合操作结果的容器。它 `DBObject` 以对映射的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。

为MongoDB Aggregation框架使用Spring Data MongoDB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();
```

JAVA

请注意，如果您提供一个输入类作为该 `newAggregation` 方法的第一个参数，`MongoTemplate` 将从该类派生出输入集合的名称。否则，如果您不指定输入类，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，则优先级为。

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：

- 管道聚集运营商
- 集群运营商
- 布尔聚合算子
- 比较聚合算子
- 算术聚合算子
- 字符串汇总算子
- 日期汇总算子
- 数组聚合算子
- 条件聚合算子
- 查询聚合运算符

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 目前支持的聚合操作

管道聚集运营商	bucket, bucketAuto, count, facet, geoNear, graphLookup, group, limit, lookup, match, project, replaceRoot, skip, sort, unwind
设置汇总算子	setEquals, setIntersection, setUnion, setDifference, setIsSubset, anyElementTrue, allElementsTrue
集群运营商	addToSet, first, last, max, min, avg, push, sum, (* count) , stdDevPop, stdDevSamp
算术聚合算子	abs, add (* via plus) , ceil, divide, exp, floor, ln, log, log10, mod, multiply, pow, sqrt, subtract (* via minus) , trunc
字符串汇总算子	concat, substr, toLower, toUpper, stcasecmp, indexOfBytes, indexOfCP, split, strLenBytes, strLenCP, substrCP,
比较聚合算子	eq (* via: is) , gt, gte, lt, lte, ne
数组聚合算子	arrayElementAt, concatArrays, filter, in, indexOfArray, isArray, range, reverseArray, reduce, size, slice, zip
文字运算符	文字

日期汇总算子	dayOfYear, dayOfMonth, dayOfWeek, year, month, week, hour, minute, second, millisecond, dateToString, isoDayOfWeek, isoWeek, isoWeekYear
变量运算符	地图
条件聚合算子	cond, ifNull, switch
类型聚合运算符	类型

请注意，此处未列出的聚合操作目前不受Spring Data MongoDB的支持。比较聚合算子表达为 `Criteria` 表达式。

*) 操作由Spring Data MongoDB映射或添加。

9.11.3. 投影表达式

投影表达式用于定义作为特定聚合步骤的结果的字段。可以通过类的 `project` 方法来定义投影表达式，方法 `Aggregation` 是通过传递列表 `String` 或聚合框架 `Fields` 对象。该投影可以通过流畅的API通过该 `and(String)` 方法和通过该方法进行别名扩展 `as(String)`。请注意，还可以通过 `Fields.field` 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建新的 `Fields` 实例。对后期聚合阶段中的投影字段的引用仅通过使用包含的字段的字段名称或其别名或新定义字段的别名才有效。

示例63. 投影表达示例

```

// 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")

```

示例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")

```

`AggregationTests` 类中可以找到更多的项目操作示例。请注意，有关投影表达式的更多详细信息可以在MongoDB汇总框架参考文档的相应部分 (http://docs.mongodb.org/manual/reference/operator/aggregation/project/#pipe._S_project)找到。

9.11.4. 分面分类

MongoDB支持使用聚合框架的版本3.4分面分类。分面分类使用通用或主题特定的语义类别，它们被组合以创建完整的分类条目。流经聚合管道的文件被划分为桶。多面分类可以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。

桶

Bucket操作根据指定的表达式和存储桶边界将传入文档分为组，称为存储桶。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来显示累加。可以通过流畅的API通过 `with...`() 方法， `andOutput(String)` 方法和通过方法进行别名来扩展桶操作 `as(String)`。每个桶在输出中表示为文档。

`BucketOperation` 采用一组定义的界限将传入的文档分组到这些类别中。边界需要排序。

示例65. 桶操作示例

```
// 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");
```

JAVA

`BucketAutoOperation` 确定边界本身以试图将文档均匀地分配到指定数量的桶中。 `BucketAutoOperation` 可选地采取粒度来指定要使用的首选数字 (https://en.wikipedia.org/wiki/Preferred_number)系列，以确保所计算的边界边缘以优选的回合号码或其10的幂数结束。

示例66. 桶操作示例

```
// 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");
```

JAVA

桶操作可以使用 `AggregationExpression` via `andOutput()` 和SpEL表达式通过 `andOutputExpression()` 在桶中创建输出字段。

请注意，有关bucket表达式的更多详细信息，请参见MongoDB Aggregation Framework参考文档的 `$bucket` 部分 (<http://docs.mongodb.org/manual/reference/operator/aggregation/bucket/>)和 `$bucketAuto` 部分 (<http://docs.mongodb.org/manual/reference/operator/aggregation/bucketAuto/>)。

多面聚集

多个聚合流水线可用于创建多面聚集，其在单个聚合阶段中跨多个维度或小平面表征数据。多面聚集提供多个过滤器和分类，以指导数据浏览和分析。一个常见的方法是通过在产品价格，制造商，尺寸等上应用过滤器，有多少在线零售商提供缩小搜索结果的方法。

A `FacetOperation` 可以通过类的 `facet()` 方法定义 `Aggregation` 。可以通过该 `and()` 方法定制多个聚合管道。每个子流水线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。

示例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
(<http://docs.mongodb.org/manual/reference/operator/aggregation/facet/>) 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。这允许您将所需的表达式定义为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。这使得更容易表达复杂的计算。`ProjectionOperation` `BucketOperation`

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。您可以在其中找到更多支持的Spel表达式构造的使用示例 `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	{ \$subtract: [\$a, \$b] }
a * b	{ \$multiply: [\$a, \$b] }
a / b	{ \$divide: [\$a, \$b] }
a ^ b	{ \$pow: [\$a, \$b] }
a % b	{ \$mod: [\$a, \$b] }
a && b	{ \$and: [\$a, \$b] }
一个 b	{ \$or: [\$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。通过它们的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})");
```

JAVA

聚合框架示例

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

聚合框架示例1

在这个介绍性的例子中，我们想聚合一个标签列表，以从降序中 `"tags"` 按发生计数排序的MongoDB集合获取特定标签的发生次数。此示例演示了分组，排序，投影（选择）和展开（结果分割）的用法。

```
class TagCount {  
    String tag;  
    int n;  
}
```

JAVA

```
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。这些聚合操作定义我们的聚合管道 `Aggregation`。
- 作为第二步，我们 `"tags"` 从输入集合中选择该字段（这是一个字符串数组） `project`。
- 在第三步中，我们使用该 `unwind` 操作为 `"tags"` 数组中的每个标签生成一个新文档。
- 在第四步中，我们使用操作 `group` 为每个 `"tags"` 值定义一个组，通过 `count` 聚合运算符聚合发生次数，并将结果收集到一个新的字段中 `"n"`。
- 作为第五步，我们选择该字段 `"n"`，并为从上一个组操作生成的 `id` 字段创建一个别名（因此调用 `previousOperation()`） `"tag"`。
- 作为第六步，我们通过操作按降序排列生成的标签列表 `sort`。
- 最后，我们调用 `aggregate` `MongoTemplate` 上的方法，以使 `MongoDB` 使用创建 `Aggregation` 的参数来执行实际的聚合操作。

需要注意的是输入集合明确指定为 `"tags"` 参数的 `aggregate` 方法。如果输入集合的名称未被明确指定，则从作为 `newAggregation` `Method` 的第一个参数传入的输入类派生。

聚合框架示例2

这个例子是基于 `MongoDB` 汇总框架文档中的 最大和最小城市

(<http://docs.mongodb.org/manual/tutorial/aggregation-examples/#largest-and-smallest-cities-by-state>) 例子。我们添加了额外的排序，以使用不同的 `MongoDB` 版本产生稳定的结果。在这里，我们要使用汇总框架，为每个国家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` 定义了所需输出格式的结构。
- 作为第一步，我们使用该 `group` 操作从输入集合中定义一个组。分组标准是字段的组合 `"state"`，并且 `"city"` 组成组的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来自分组元素的属性值 `"pop"`。
- 在第二步骤中，我们使用的 `sort` 操作由字段到中间结果进行排序 `"pop"`，`"state"` 并 `"city"` 以升序，使得最小的城市是在顶部和最大的城市是在结果的底部。请注意，对Spring Data MongoDB所关心的组ID字段进行排序 `"state"` 并 `"city"` 隐式执行。
- 在第三步中，我们 `group` 再次使用一个操作来对中间结果进行分组 `"state"`。请注意，`"state"` 再次隐式引用 `group-id` 字段。我们通过操作分别选择最大和最小城市的名称和人口数量，分别通过呼叫 `last(...)` 和 `first(...)` 操作 `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，所以我们通过以前的操作排除了id `and(previousOperation()).exclude()`。因为我们想 `City` 在我们的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` 通过操作按升序对其结果列表进行排序 `sort`。

请注意，我们从 `ZipInfo` -class 中将输入集合的名称作为第一个参数传递给-Method `newAggregation`。

聚合框架示例3

该示例基于MongoDB汇总框架文档中的具有超过1000万

(<http://docs.mongodb.org/manual/tutorial/aggregation-examples/#states-with-populations-over-10-million>)人口的州份。我们添加了额外的排序，以使用不同的MongoDB版本产生稳定的结果。在这里，我们要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，我们将 "state" 字段中的输入集合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" 按照升序排列的字段之外，我们还可以通过上一个组操作的id引用对中间结果进行排序。
- 最后在第三步中，我们通过使用 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 -class 中将输入集合的名称作为第一个参数传递给-Method newAggregation。

聚合框架示例4

该示例演示了在投影操作中使用简单的算术运算。

```
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 -class 中将输入集合的名称作为第一个参数传递给-Method newAggregation。

聚合框架示例5

本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

本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 `addExpression` 方法的附加参数可以通过索引器表达式根据其位置进行引用。在这个例子中，我们引用参数，它是参数数组的第一个参数 `[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表达式中引用该文档的其他字段。

聚合框架示例7

此示例使用条件投影。它来自于[\\$ cond参考文档](https://docs.mongodb.com/manual/reference/operator/aggregation/cond/) (https://docs.mongodb.com/manual/reference/operator/aggregation/cond/)。

```
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();
```

- 此一步聚合使用集合的投影 `inventory` 操作。我们 `discount` 使用具有 `qty` 大于或等于的所有库存料品的条件操作来投放该库 250。对该 `description` 字段执行第二个条件投影。我们将描述应用于 `Unspecified` 不具有 `description` 描述项目的字段的所有项目 `null`。

9.12. 使用自定义转换器覆盖默认映射

为了让在映射过程中，你可以用注册春季转换器更细粒度的控制 `MongoConverter` 实现如 `MappingMongoConverter`。

该 `MappingMongoConverter` 检查，看看是否有任何春转换器，可以尝试映射对象本身前，处理特定的类。为了“劫持”正常的映射策略 `MappingMongoConverter`，也许为了提高性能或其他自定义映射需求，您首先需要创建 `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](#)

(<http://docs.spring.io/spring/docs/4.3.8.RELEASE/spring-framework-reference/html/validation.html#core-convert>) 的参考文档。

9.12.1. 使用注册的Spring Converter保存

的一示例实现中 `Converter`，从一个 `Person` 对象到一个转换 `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. 阅读使用弹簧转换器

一个从DBObject转换为Person对象的转换器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 Converters

Mongo Spring命名空间提供了一种方便的注册Spring Converter 的方式 MappingMongoConverter 。下面的配置片段显示了如何手动注册转换器bean以及将包装配置 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>
```

您还可以使用自定义转换器元素的base-package属性来为给定包下面的所有 Converter 和实现启用类路径扫描 GenericConverter 。

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

9.12.4. 转换器消歧

一般来说，我们检查 `Converter` 他们转换的源和目标类型的实现。根据其中一个是MongoDB类型可以自己处理，我们将注册转换器实例作为读或写一个。看看下面的样品：

```
// 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> { ... }
```

JAVA

如果你写一个 `Converter` 源和目标类型是本地Mongo类型，我们无法确定我们是否应该将其视为读写转换器。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>` 是模糊的，尽管在写入时尝试将所有 `String` 实例转换为 `Long` 实例可能是没有意义的。通常能够强制基础设施以单向注册转换器，只有我们提供 `@ReadingConverter` 以及在 `@WritingConverter` 转换器实现中使用。

9.13. 索引和收集管理

`MongoTemplate` 提供了几种管理索引和集合的方法。这些被收集到一个叫做“帮助”界面中 `IndexOperations`。您可以通过调用方法访问这些操作，`indexOps` 并传入集合名称或 `java.lang.Class` 实体（集合名称将通过名称或通过注释元数据从.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();
}
```

JAVA

9.13.1. 创建索引的方法

我们可以在集合上创建一个索引，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));
```

JAVA

- **ensureIndex**确保为集合存在所提供的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"));
```

JAVA

9.13.2. 访问索引信息

`IndexOperations`接口具有返回`IndexInfo`对象列表的方法`getIndexInfo`。这包含集合上定义的所有索引。这是一个例子，它定义了具有`age`属性的`Person`类的索引。

```
template.indexOps(Person.class).ensureIndex(new Index().on("age",
Order.DESENDING).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显示如何使用 `MongoTemplate` 。首先我们来看看创建我们的第一个集合。

示例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`创建一个未封顶的集合
- `dropCollection`删除集合
- `getCollection`通过名称获取集合，如果不存在则创建它。

9.14. 执行命令

您也可以在MongoDB的驱动程序的 `DB.command()` 方法使用 `executeCommand(...)` 的方法 `MongoTemplate` 。这些也将执行到Spring的 `DataAccessException` 层次结构的异常翻译。

9.14.1. 执行命令的方法

- `CommandResult executeCommand (DBObject command)` 执行MongoDB命令。
- `CommandResult executeCommand (String jsonCommand)` 执行以JSON字符串表示的MongoDB命令。

9.15. 生命周期活动

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的bean来响应 `ApplicationContext` 。通过基于Spring的`ApplicationContext`事件基础架构，这使得其他产品（如Spring Integration）能够容易地接收这些事件，因为它们是基于Spring的应用程序中众所周知的事件机制。

要在通过转换过程（将您的域对象变成一个 `com.mongodb.DBObject` ）之前拦截对象，您将注册该对象的子类 `AbstractMongoEventListener` 覆盖该 `onBeforeConvert` 方法。当调度事件时，您的监听器将被调用并传递给域对象，然后才能进入转换器。

JAVA

```
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一个对象，你需要注册一个子

类 `org.springframework.data.mongodb.core.mapping.event.AbstractMongoEventListener` 来覆盖该 `onBeforeSave` 方法。当调度事件时，您的监听器将被调用并传递给域对象和转换 `com.mongodb.DBObject`。

JAVA

```
public class BeforeSaveListener extends AbstractMongoEventListener<Person> {  
    @Override  
    public void onBeforeSave(BeforeSaveEvent<Person> event) {  
        ... change values, delete them, whatever ...  
    }  
}
```

在Spring ApplicationContext中简单地声明这些bean将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MongoTemplate中使用MongoConveter将对象转换为DBObject之前，在MongoTemplate中插入，insertList和保存操作。
- `onBeforeSave` - 在数据库中插入/ 保存DBObject 之前，在MongoTemplate中调用insert，insertList和save操作。
- `onAfterSave` - 在数据库中插入/ 保存DBObject 后，在MongoTemplate中调用insert，insertList和save操作。
- `onAfterLoad` 在从数据库检索DBObject后，在MongoTemplate中调用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 find，findAndRemove，findOne和getCollection方法中调用。



生命周期事件仅针对根级别类型发出。用作文档根目录中的属性的复杂类型不是事件发布的主体，除非它们是注释的文档引用 `@DBRef`。

9.16. 异常翻译

Spring框架为各种数据库和映射技术提供异常翻译。这传统上是用于JDBC和JPA。MongoDB的Spring支持通过提供接口的实现将此功能扩展到MongoDB数据库 `org.springframework.dao.support.PersistenceExceptionTranslator`。

映射到Spring 一致的数据访问异常层次结构的动机

(<http://docs.spring.io/spring/docs/4.3.8.RELEASE/spring-framework-reference/html/dao.html#dao-exceptions>)是，您可以编写可移植和描述性的异常处理代码，而不需要对MongoDB错误代码进行编码

(<https://www.mongodb.org/about/contributors/error-codes/>)。Spring的所有数据访问异常都从根 `DataAccessException` 类继承，所以您可以确保在单个try-catch块中可以捕获与数据库相关的所有异常。请注意，MongoDB驱动程序抛出的并非所有异常都从MongoException类继承。内部异常和消息被保留，所以没有信息丢失。

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` 。了解更多关于映射的细节的实现。

9.17. 执行回调

所有Spring模板类的一个常见设计特征是所有功能都被路由到一个模板中执行回调方法。这有助于确保可能需要的异常和任何资源管理的一致性。虽然在JDBC和JMS的情况下,与使用MongoDB相比,这需要更大的需求,但它仍然提供了异常转换和日志记录发生的单一位置。因此,使用这些执行回调是访问MongoDB驱动程序 `DB` 和 `DBCollection` 对象以执行未作为方法公开的不常见操作的首选方式 `MongoTemplate` 。

这是一个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`, 根据需要翻译任何异常。
- `<T> T execute (String collectionName, DbCallback<T> action)` 对给定名称的集合执行`DbCallback`, 根据需要翻译任何异常。
- `<T> T executeInSession (DbCallback<T> action)` 在与数据库相同的连接中执行给定的`DbCallback`, 以确保写入沉重环境中的一致性, 您可以在其中读取所写入的数据。

这是一个使用该 `CollectionCallback` 函数返回有关索引的信息的示例

```
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;  
    }  
});
```

JAVA

9.18. GridFS支持

MongoDB支持在其文件系统GridFS中存储二进制文件。Spring数据MongoDB提供了一个 `GridFsOperations` 接口以及相应的实现 `GridFsTemplate` , 可轻松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` :

示例69.GroupFemplate的JavaConfig设置

```
class GridFsConfiguration extends AbstractMongoConfiguration {

    // ... further configuration omitted

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

JAVA

根据XML配置看起来像这样:

示例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>
```

XML

该模板现在可以被注入并用于执行存储和检索操作。

示例71. 使用GridFsTemplate来存储文件

```
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);
    }
}
```

JAVA

这些 `store(...)` 操作需要一个 `InputStream` 文件名和可选的元数据信息来存储文件。元数据可以是一个任意的对象，它将被配置 `MongoConverter` 为 `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(...)` 或者 `getResources(...)` 方法来实现。我们先来看一下 `find(...)` 方法。您可以找到一个匹配一个 `Query` 或多个文件的文件。为了方便地定义文件查询，我们提供了 `GridFsCriteria` 帮助类。它提供静态工厂方法来封装默认元数据字段（例如 `whereFilename()`，`whereContentType()`）或自定义元数据字段 `whereMetaData()`。

示例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")))  
    }  
}
```

JAVA



目前，MongoDB不支持在从GridFS检索文件时定义排序条件。因此，任何在 `Query` 递交该 `find(...)` 方法的实例上定义的排序标准将被忽略。

从GridFs读取文件的其他选项是使用接口介绍的方法 `ResourcePatternResolver`。它们允许将Ant路径传递到方法 `ar` 中，从而检索与给定模式匹配的文件。

示例73. 使用GridFsTemplate读取文件

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

JAVA

`GridFsOperations` 扩展 `ResourcePatternResolver` 允许将 `GridFsTemplate` 例如插入到 `ApplicationContext` MongoDB中读取Spring Config文件。

MongoDB存储库

10.1. 介绍

本章将指出MongoDB的存储库支持的特色。这建立在使用Spring数据存储库中解释的核心存储库支持。所以，确保你对这里解释的基本概念有了很好的了解。

10.2. 用法

要访问存储在MongoDB中的域实体，您可以利用我们复杂的存储库支持，轻松实现这些实体。为此，只需为您的存储库创建一个界面：

示例74. 示例人员实体

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

JAVA

我们在这里有一个非常简单的域对象。注意它有一个名为 `id type` 的属性 `ObjectId`。用于 `MongoTemplate`（它支持存储库支持）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` 为ID类型。

示例75. 持久化的基本存储库接口Person实体

```
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的目的，但是我们稍后会添加其他方法。在你的Spring配置中，只需添加一下

示例76. 通用MongoDB存储库Spring配置

XML

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

这个命名空间元素将导致基础包被扫描以扩展 `MongoRepository` 并为每个发现的Spring bean创建Spring bean。默认情况下，存储库将获得一个 `MongoTemplate` 被调用的Spring bean `mongoTemplate`，因此 `mongo-template-ref` 如果偏离此约定，则只需要显式配置。

如果您想要使用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。如果没有配置基础包，基础设施将扫描带注释的配置类的包。

示例77. 存储库的JavaConfig

JAVA

```
@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操作以及分页和分类访问实体的方法。使用存储库实例只是将其注入客户端的依赖问题。所以访问 `Person` 页面大小为10的第二页将只是看起来像这样：

示例78. 寻呼访问个人实体

```

@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));
    }
}

```

JAVA

该示例使用Spring的单元测试支持创建一个应用程序上下文，该支持将基于注解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。我们将存储库 `PageRequest` 提交给页面大小为10的人的第一页的实例。

10.3. 查询方式

您通常在存储库上触发的大多数数据访问操作会导致针对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的问题

示例79. 具有查询方法的PersonRepository

```

public interface PersonRepository extends PagingAndSortingRepository<Person, String> {

    List<Person> findByLastname(String lastname);

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

    Person findByShippingAddresses(Address address);

    Stream<Person> findAllBy();
}

```

JAVA

- ❶ 该方法显示对具有给定姓氏的所有人的查询。该查询将被导出可与被连结约束解析方法名称 `And` 和 `Or` 。因此，方法名称将导致查询表达式 `{"lastname" : lastname}` 。
- ❷ 将分页应用于查询。只需将方法签名与 `Pageable` 参数配合，并让方法返回一个 `Page` 实例，我们将自动对该查询进行相应的页面查询。
- ❸ 显示您可以根据不是原始类型的属性进行查询。
- ❹ 使用Java 8 `Stream` ，它在迭代流时读取和转换单个元素。



请注意，对于版本1.0，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	<code>findByBirthdateAfter(Date date)</code>	<code>{"birthdate" : {"\$gt" : date}}</code>

关键词	样品	逻辑结果
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}}
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...]}}
IsNull, 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}} (name as regex)
Containing 集合	findByAddressesContaining(Address address)	{"addresses" : {"\$in" : address}}
NotContaining 集合	findByAddressesNotContaining(Address address)	{"addresses" : {"\$not" : {"\$in" : address}}}

关键词	样品	逻辑结果
Regex	<code>findByFirstnameRegex(String firstname)</code>	<code>{"firstname" : {"\$regex" : firstname }}</code>
(No keyword)	<code>findByFirstname(String name)</code>	<code>{"firstname" : name}</code>
Not	<code>findByFirstnameNot(String name)</code>	<code>{"firstname" : {"\$ne" : name}}</code>
Near	<code>findByLocationNear(Point point)</code>	<code>{"location" : {"\$near" : [x,y]}}</code>
Near	<code>findByLocationNear(Point point, Distance max)</code>	<code>{"location" : {"\$near" : [x,y], "\$maxDistance" : max}}</code>
Near	<code>findByLocationNear(Point point, Distance min, Distance max)</code>	<code>{"location" : {"\$near" : [x,y], "\$minDistance" : min, "\$maxDistance" : max}}</code>
Within	<code>findByLocationWithin(Circle circle)</code>	<code>{"location" : {"\$geoWithin" : {"\$center" : [[x, y], distance]}}</code>
Within	<code>findByLocationWithin(Box box)</code>	<code>{"location" : {"\$geoWithin" : {"\$box" : [[x1, y1], x2, y2]}}</code>
IsActive, True	<code>findByIsActiveTrue()</code>	<code>{"active" : true}</code>
IsActive, False	<code>findByIsActiveFalse()</code>	<code>{"active" : false}</code>
Exists	<code>findByLocationExists(boolean exists)</code>	<code>{"location" : {"\$exists" : exists }}</code>

10.3.1. 存储库删除查询

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

实施例80 `Delete...By` 的查询


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

    List<Person> deleteByLastname(String lastname);

    Long deletePersonByLastname(String lastname);
}
```

JAVA

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

正如您刚刚看到的，有一些关键字在MongoDB查询中触发地理空间操作。该 `Near` 关键字允许进一步修改。让我们来看一些例子：

示例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);
}
```

JAVA

将参数添加 `Distance` 到查询方法允许将结果限制在给定距离内的结果。如果 `Distance` 设置包含 `Metric` 我们将透明地使用 `$nearSphere` 而不是 `$` 代码。

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}}
```

JAVA

正如你可以看到使用 `Distance` 配备了一个 `Metric` cause `$nearSphere` 子句而不是一个简单的 `$near`。除此之外，根据使用的实际距离进行计算 `Metrics`。



使用 `@GeoSpatialIndexed(type = GeoSpatialIndexType.GEO_2DSPHERE)` 目标属性强制 `$nearSphere` 运算符的使用。

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. MongoDB基于JSON的查询方法和字段限制

通过添加注释存储 `org.springframework.data.mongodb.repository.Query` 库查找器方法，您可以指定使用 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` 可以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转义，这意味着无法通过参数添加MongoDB特定的操作符。

您还可以使用 `filter` 属性来限制将映射到Java对象的一组属性。例如，

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

    @Query(value="{ 'firstname' : ?0 }", fields="{ 'firstname' : 1, 'lastname' : 1}")
    List<Person> findByThePersonsFirstname(String firstname);

}
```

这将只返回Person对象的 `firstname`、`lastname` 和 `Id` 属性。`age` 属性，一个 `java.lang.Integer`，不会被设置，因此它的值将为 `null`。

10.3.4. 具有SpEL表达式的基于JSON的查询

查询字符串和字段定义可以与 `Spel` 表达式一起使用，以便在运行时创建动态查询。`SpEL` 表达式可以提供谓词值，并可用于使用子文档扩展谓词。

表达式通过包含所有参数的数组来公开方法参数。以下查询用于 `[0]` 声明该谓词的 `lastname` 等价于 `?0` 参数绑定的谓词值。

JAVA

```
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结合显示副作用，如SpEL中的类似Map的声明，像JSON一样。

JAVA

```
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可以成为强大的查询方式，可以接受广泛的不必要的参数。在将这些信息传递给查询之前，您应该确保对字符串进行清理，以避免对查询进行不必要的更改。

通过Query SPI `org.springframework.data.repository.query.spi.EvaluationContextExtension` 可以扩展表达式支持，可以贡献属性，函数和自定义根对象。当查询构建时，在SpEL评估时从应用程序上下文中检索扩展。

JAVA

```
public class SampleEvaluationContextExtension extends EvaluationContextExtensionSupport {

    @Override
    public String getExtensionId() {
        return "security";
    }

    @Override
    public Map<String, Object> getProperties() {
        return Collections.singletonMap("principal", SecurityContextHolder.getCurrent().getPrincipal());
    }
}
```



Bootstrapping `MongoRepositoryFactory` 自己不是应用程序上下文感知，需要进一步配置来接收Query SPI扩展。

10.3.5. 类型安全查询方法

MongoDB存储库支持与[QueryDSL](http://www.querydsl.com/) (http://www.querydsl.com/)项目集成，该项目提供了一种在Java中执行类型安全查询的方法。引用项目描述，“而不是将查询作为内联字符串或外部化为XML文件，而不是通过流畅的API构建。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）
- 几乎没有语法无效的查询被允许（所有级别都是安全的）
- 可以安全地引用域类型和属性（不涉及字符串！）
- 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](http://www.querydsl.com/static/querydsl/latest/reference/html/) (http://www.querydsl.com/static/querydsl/latest/reference/html/)，该文档介绍如何使用Maven或Ant为基于APT的代码生成引导您的环境。

使用QueryDSL，您将能够编写如下所示的查询

```
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` 是通过Java注释后处理工具生成的一个类 `Predicate`，它允许您编写类型安全的查询。请注意查询中没有字符串，而不是值“C0123”。

您可以 `Predicate` 通过 `QueryDslPredicateExecutor` 下面显示的接口使用生成的类

```
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，只需从其中继承其他存储库接口即可。如下图所示

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

    // additional finder methods go here
}
```

我们认为您会发现这是编写MongoDB查询的非常强大的工具。

10.3.6. 全文搜索查询

MongoDBs全文搜索功能是非常专门的商店，因此可以找到 `MongoRepository` 比更通用的 `CrudRepository`。我们需要的是一个全文索引定义的文档（请参阅文本索引创建）。

上的其他方法 `MongoRepository` 取出 `TextCriteria` 作为输入参数。除了这些显式方法之外，还可以添加 `TextCriteria` 派生的存储库方法。标准将作为附加标准添加 `AND`。一旦实体包含 `@TextScore` 注释属性，将检索文档全文分数。此外，`@TextScore` 注释属性还可以通过文档分数进行排序。

```

@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数据库通常在使用查询方法时返回域模型。但是，有时，由于各种原因，您可能需要更改该模型的视图。在本节中，您将学习如何定义预测以提供资源的简化和简化视图。

看下面的域名模型：

```

@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` 是主要的键

- `firstName` 并且 `lastName` 是数据属性
- `address` 是指向另一个域对象的链接

现在假设我们创建一个相应的仓库，如下所示：

```
interface PersonRepository extends CrudRepository<Person, Long> {  
  
    Person findPersonByFirstName(String firstName);  
}
```

JAVA

Spring Data将返回包含其所有属性的域对象。有两个选项只是检索 `address` 属性。一个选择是为这样的 `Address` 对象定义一个存储库：

```
interface AddressRepository extends CrudRepository<Address, Long> {}
```

JAVA

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，如果不想公开 `address` 细节呢？您可以通过定义一个或多个投影来为存储库服务的消费者提供替代方案。

示例83.简单投影

```
interface NoAddresses {  
  
    String getFirstName();  
  
    String getLastName();  
}
```

JAVA

此投影具有以下细节：

- ❶ 一个简单的Java界面，使其声明。
- ❷ 导出 `firstName`。
- ❸ 导出 `lastName`。

该 `NoAddresses` 投影仅拥有干将 `firstName` 和 `lastName` 意义，它不会成为的任何地址信息。查询方法定义返回在这种情况下 `NoAddresses`，而不是 `Person`。

```
interface PersonRepository extends CrudRepository<Person, Long> {  
  
    NoAddresses findByFirstName(String firstName);  
}
```

JAVA

投影声明了底层类型和与暴露属性相关的方法签名之间的合同。因此，需要根据底层类型的属性名称命名getter方法。如果底层属性被命名 `firstName`，那么必须命名getter方法，`getFirstName` 否则Spring Data不能查找source属性。这种投影也称为*封闭投影*。封闭的投影公开了属性的一个子集，因此它们可以用来优化查询，以减少数据存储中选定的字段。你可能想像的另一种类型是一个*公开的投射*。

重塑数据

到目前为止，您已经看到如何使用投影来减少向用户呈现的信息。投影可用于调整曝光的数据模型。您可以向投影添加虚拟属性。看下面的投影界面：

示例84.重命名属性

```
interface RenamedProperty {  
    String getFirstName();  
    @Value("#{target.lastName}")  
    String getName();  
}
```

JAVA

此投影具有以下细节：

- 1 一个简单的Java界面，使其声明。
- 2 导出 `firstName`。
- 3 导出 `name` 财产。由于此属性是虚拟的，因此需要 `@Value("#{target.lastName}")` 指定属性源。

后台域模型没有这个属性，所以我们需要告诉Spring Data从中获取这个属性。虚拟属性是 `@Value` 发挥作用的地方。该 `name` 属性与注释 `@Value` 使用 规划环境地政司表情 (<http://docs.spring.io/spring/docs/4.3.8.RELEASE/spring-framework-reference/html/expressions.html>)指着后盾财产 `lastName`。您可能已经注意到 `lastName` 前缀 `target` 是指向后备对象的变量名。使用 `@Value` 方法允许定义获取值的位置和方式。

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())` 可能是一种可能性，但是这个代码需要在每个地方都需要调用。预测的虚拟属性充分考虑了重复该代码的需要。

```
interface FullNameAndCountry {  
    @Value("#{target.firstName} #{target.lastName}")  
    String getFullName();  
    @Value("#{target.address.country}")  
    String getCountry();  
}
```

JAVA

实际上，`@Value` 可以完全访问目标对象及其嵌套属性。`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 `Spel`表达式的预测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;  
    ...  
}
```

JAVA



这个例子可能看起来有点诡异，但是可以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无法识别这些数据的敏感性，所以由开发人员来避免这种情况。不要将密码存储为纯文本。你真的不应该这样做 对于这个例子，你也可以 `password` 用其他任何秘密替代。

在某些情况下，您可能会保持 `password` 尽可能的秘密，而不会使其暴露更多。解决方案是创建一个 `@Value` 与Spel表达式一起使用的投影。

```
interface PasswordProjection {
    @Value("#{(target.password == null || target.password.empty) ? null : '*****'}")
    String getPassword();
}
```

JAVA

该表达式检查密码是否为 `null` 空，并 `null` 在这种情况下返回，否则设置了六个星号以指示密码。

10.4. 杂

10.4.1. CDI整合

存储库接口的实例通常由容器创建，Spring在使用Spring Data时是最自然的选择。从版本1.3.0起，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，所以您需要做的是激活它，将Spring Data MongoDB JAR删除到您的类路径中。您现在可以通过实施CDI Producer来设置基础架构 `MongoTemplate`：

```
class MongoTemplateProducer {

    @Produces
    @ApplicationScoped
    public MongoOperations createMongoTemplate() throws UnknownHostException, MongoException {

        MongoClientFactory factory = new SimpleMongoDbFactory(new MongoClient(), "database");
        return new MongoTemplate(factory);
    }
}
```

JAVA

Spring数据MongoDB CDI扩展将拾取 `MongoTemplate` 作为CDI bean可用，并在容器请求存储库类型的bean时为Spring数据存储库创建一个代理。因此，获取Spring数据存储库的一个实例是声明一个 `@Inject` 属性的问题：

```
class RepositoryClient {

    @Inject
    PersonRepository repository;

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

JAVA

11. 审计

11.1. 基本

Spring Data提供了复杂的支持，以透明地跟踪创建或更改实体的人员以及发生的时间点。为了从该功能中受益，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实现一个接口进行定义。

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。

实例85. 被审计单位

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

JAVA

您可以看到，可以选择性地应用注释，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 DateTime`，旧 `Java Date` 和 `Calendar JDK8` 日期/时间类型以及 `long` / 的类型的属性 `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，则可以让您的域类实现该 `Auditable` 接口。它暴露了所有审核属性的setter方法。

还有一个方便的基类 `AbstractAuditable`，您可以扩展，以避免手动实现接口方法的需要。请注意，这增加了您的域类与Spring数据的耦合，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首选的，因为它具有较少的侵入性和更灵活性。

11.1.3. AuditorAware

如果您使用任何一个 `@CreatedBy` 或者 `@LastModifiedBy`，审计基础设施需要了解当前的主体。为此，我们提供了一个 `AuditorAware<T>` SPI接口，您必须实现该接口来告知基础架构当前用户或系统与应用程序交互的情况。通用类型 `T` 定义了使用 `@CreatedBy` 或 `@LastModifiedBy` 必须注释的属性类型。

以下是使用Spring Security Authentication 对象的接口示例实现：

示例86. 基于Spring Security的AuditorAware的实现

JAVA

```
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();
    }
}
```

该实现是访问 `Authentication` Spring Security提供的对象，并查找 `UserDetails` 您在实现中创建的自定义实例 `UserDetailsService`。我们在这里假设您通过该 `UserDetails` 实现暴露域用户，但您也可以根据 `Authentication` 发现从任何地方查找。

11.2. 一般审核配置

激活审计功能只是将Spring Data Mongo `auditing` 命名空间元素添加到您的配置中：

示例87. 使用XML配置激活审核

XML

```
<mongo:auditing mapping-context-ref="customMappingContext" auditor-aware-
ref="yourAuditorAwareImpl"/>
```

由于可以通过使用注释注释配置类来启用Spring Data MongoDB 1.4审核 `@EnableMongoAuditing`。

示例88. 使用JavaConfig激活审核

JAVA

```
@Configuration
@EnableMongoAuditing
class Config {

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

如果你暴露类型的豆 `AuditorAware` 到 `ApplicationContext`，审计基础设施会自动把它捡起来，并用它来确定当前用户要在域类型设置。如果您有多个实现注册 `ApplicationContext`，您可以通过显式设置 `auditorAwareRef` 属性来选择要使用的实现 `@EnableJpaAuditing`。

映射

丰富的地图支持由...提供 `MappingMongoConverter` 。 `MappingMongoConverter` 具有丰富的元数据模型，提供功能集功能，将域对象映射到MongoDB文档。映射元数据模型使用域对象上的注释填充。然而，基础设施不限于使用注释作为元数据信息的唯一来源。该 `MappingMongoConverter` 还允许在不提供任何额外的元数据，通过以下一组约定的映射对象的文件。

在这一节中我们将介绍一下这些特性 `MappingMongoConverter` 。如何使用约定将对象映射到文档，以及如何使用基于注释的映射元数据覆盖这些约定。



`SimpleMongoConverter` 已经在Spring Data MongoDB M3中弃用，因为它的所有功能都已被归入 `MappingMongoConverter` 。

12.1。基于会议的映射

`MappingMongoConverter` 在不提供附加的映射元数据的情况下，有几个约定将对象映射到文档。公约是：

- 短Java类名称以以下方式映射到集合名称。该类 `com.bigbank.SavingsAccount` 映射到 `savingsAccount` 集合名称。
- 所有嵌套对象都作为嵌套对象存储在文档中，而不是作为DBRefs存储
- 转换器将使用其注册的任何Spring转换器来覆盖对象属性到文档字段/值的默认映射。
- 对象的字段用于转换文档中的字段和从字段转换。不使用公共JavaBean属性。
- 您可以拥有一个非零参数构造函数，其构造函数参数名称与文档的顶级字段名称匹配，该构造函数将被使用。否则将使用零arg构造函数。如果有多个非零参数构造函数将抛出异常。

12.1.1。如何 `_id` 在映射层中处理该字段

MongoDB要求您具有 `_id` 所有文档的字段。如果您没有提供一个驱动程序，则会为ObjectId分配一个生成的值。只要它是唯一的，“`_id`”字段可以是除数组之外的任何类型。司机自然支持所有原始类型和日期。在使用时， `MappingMongoConverter` 有一些规则可以控制Java类的属性如何映射到此 `_id` 字段。

以下概述将映射到 `_id` 文档字段的字段：

- 用 `@Id` （ `org.springframework.data.annotation.Id` ）注释的字段将映射到 `_id` 字段。
- 没有注释但命名 `id` 的 `_id` 字段将被映射到字段。
- 标识符的默认字段名称 `_id` 可以通过注释进行自 `@Field` 定义。

表7. `_id` 字段定义的翻译示例

字段定义	MongoDB中产生的Id字段名
String ID	<code>_id</code>
@Field String ID	<code>_id</code>
@Field("x") String ID	<code>x</code>
@Id String X	<code>_id</code>

字段定义	MongoDB中产生的Id字段名
@Field("x") @Id String X	_id

以下概述了在映射到_id文档字段的属性上将执行什么类型转换（如果有）。

- 如果一个名为的字段 `id` 在Java类中声明为String或BigInteger，那么它将被转换并存储为ObjectId（如果可能）。ObjectId作为字段类型也是有效的。如果您在应用程序中指定了一个值，`id` 则会向MongoDBdriver检测到ObjectId的转换。如果指定的 `id` 值不能转换为ObjectId，则该值将按照文档的_id字段存储。
- 如果一个名为 `id` id字段的字段未被声明为Java类中的String，BigInteger或ObjectID，那么您应该在应用程序中为其分配一个值，以便它可以原样存储在文档的_id字段中。
- 如果 `id` Java类中没有命名的字段 `_id`，那么驱动程序将生成隐式文件，但不映射到Java类的属性或字段。

当查询和更新 `MongoTemplate` 将使用转换器来处理转换的 `Query`，并 `Update` 对应于上述规则保存文档，以便在查询的字段名称和类型将能够匹配什么是在你的领域类的对象。

12.2。数据映射和类型转换

本节介绍如何将类型映射到MongoDB表示形式，反之亦然。Spring数据MongoDB支持可以表示为BSON，MongoDB内部文档格式的所有类型。除了这些类型，Spring Data MongoDB还提供了一组内置转换器来映射其他类型。您可以提供自己的转换器来调整类型转换，有关详细信息，请参阅使用显式转换器覆盖映射。

表8. 类型

类型	类型转换	样品
String	本地人	{"firstname" : "Dave"}
double， Double， float， Float	本地人	{"weight" : 42.5}
int， Integer， short， Short	本机 32位整数	{"height" : 42}
long， Long	本机 64位整数	{"height" : 42}
Date， Timestamp	本地人	{"date" : ISODate("2019-11-12T23:00:00.809Z")}
byte[]	本地人	{"bin" : { "\$binary" : "AQIDBA==", "\$type" : "00" }}
java.util.UUID （旧版UUID）	本地人	{"uuid" : { "\$binary" : "MEaf1CFQ6lSphaa3b9AtlA==", "\$type" : "03" }}
Date	本地人	{"date" : ISODate("2019-11-12T23:00:00.809Z")}
ObjectId	本地人	{"_id" : ObjectId("5707a2690364aba3136ab870")}

类型	类型转换	样品
数组 List , BasicDBList	本地人	{"cookies" : [...]}
boolean , Boolean	本地人	{"active" : true}
null	本地人	{"value" : null}
DBObject	本地人	{"value" : { ... }}
Decimal128	本地人	{"value" : NumberDecimal(...)}
AtomicInteger get() 在实际转换前 拨打电话	转换器 32位整数	{"value" : "741" }
AtomicLong get() 在实际转换前 拨打电话	转换器 64位整数	{"value" : "741" }
BigInteger	转换器 String	{"value" : "741" }
BigDecimal	转换器 String	{"value" : "741.99" }
URL	转换器	{"website" : "http://projects.spring.io/spring-data-mongodb/" }
Locale	转换器	{"locale" : "en_US" }
char , Character	转换器	{"char" : "a" }
NamedMongoScript	转换器 Code	{"_id" : "script name", value: (some javascript code) }
java.util.Currency	转换器	{"currencyCode" : "EUR"}
LocalDate (Joda, Java 8, JSR310-BackPort)	转换器	{"date" : ISODate("2019-11-12T00:00:00.000Z")}
LocalDateTime , LocalTime , Instant (约达, 爪哇8, JSR310-反向移植)	转换器	{"date" : ISODate("2019-11-12T23:00:00.809Z")}
DateTime (雅达)	转换器	{"date" : ISODate("2019-11-12T23:00:00.809Z")}
DateMidnight (雅达)	转换器	{"date" : ISODate("2019-11-12T00:00:00.000Z")}
ZoneId (Java 8, JSR310-BackPort)	转换器	{"zoneId" : "ECT - Europe/Paris"}

类型	类型转换	样品
Box	转换器	<code>{"box" : { "first" : { "x" : 1.0 , "y" : 2.0} , "second" : { "x" : 3.0 , "y" : 4.0}}}</code>
Polygon	转换器	<code>{"polygon" : { "points" : [{ "x" : 1.0 , "y" : 2.0} , { "x" : 3.0 , "y" : 4.0} , { "x" : 4.0 , "y" : 5.0}] }}</code>
Circle	转换器	<code>{"circle" : { "center" : { "x" : 1.0 , "y" : 2.0} , "radius" : 3.0 , "metric" : "NEUTRAL"}}</code>
Point	转换器	<code>{"point" : { "x" : 1.0 , "y" : 2.0}}</code>
GeoJsonPoint	转换器	<code>{"point" : { "type" : "Point" , "coordinates" : [3.0 , 4.0] }}</code>
GeoJsonMultiPoint	转换器	<code>{"geoJsonLineString" : { "type": "MultiPoint", "coordinates": [[0 , 0] , [0 , 1] , [1 , 1]] }}</code>
Sphere	转换器	<code>{"sphere" : { "center" : { "x" : 1.0 , "y" : 2.0} , "radius" : 3.0 , "metric" : "NEUTRAL"}}</code>
GeoJsonPolygon	转换器	<code>{"polygon" : { "type" : "Polygon", "coordinates" : [[[0 , 0] , [3 , 6] , [6 , 1] , [0 , 0]]] }}</code>
GeoJsonMultiPolygon	转换器	<code>{"geoJsonMultiPolygon" : { "type" : "MultiPolygon", "coordinates" : [[[[-73.958 , 40.8003] , [-73.9498 , 40.7968]]] , [[[-73.973 , 40.7648] , [-73.9588 , 40.8003]]]] }}</code>
GeoJsonLineString	转换器	<code>{ "geoJsonLineString" : { "type" : "LineString", "coordinates" : [[40 , 5] , [41 , 6]] }}</code>
GeoJsonMultiLineString	转换器	<code>{"geoJsonLineString" : { "type" : "MultiLineString", coordinates: [[[-73.97162 , 40.78205] , [-73.96374 , 40.77715]] , [[-73.97880 , 40.77247] , [-73.97036 , 40.76811]]] }}</code>

12.3. 映射配置

除非明确配置，否则 `MappingMongoConverter` 在创建时将默认创建一个实例 `MongoTemplate`。您可以创建自己的实例，`MappingMongoConverter` 以便在启动您的域类时告诉它在哪里扫描类路径，以便提取元数据并构造索引。另外，通过创建自己的实例，您可以注册Spring转换器，用于将数据库的特定类映射到数据库。

您可以使用基于Java或XML的元数据 `MappingMongoConverter` 来配置 `com.mongodb.Mongo` 以及 `MongoTemplate`。以下是使用Spring基于Java的配置的示例

示例89. `@Configuration`类配置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` 要求您实现定义一个 `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(...)`，它告诉转换器在哪里扫描注释注释的 `@Document` 类。

您可以通过覆盖 `MappingMongoConverterCreation` 之后的方法向转换器添加额外的转换器。在上面的例子中还示出了 `LoggingEventListener` 其中记录 `MongoMappingEvent` 被张贴到春季的 `S- ApplicationContextEvent` 基础设施。



`AbstractMongoConfiguration`将创建一个 `MongoTemplate`实例，并在该名称下注册到该容器 `mongoTemplate`。

您还可以覆盖该方法 `UserCredentials` `getUserCredentials()` 以提供连接到数据库的用户名和密码信息。

Spring的MongoDB命名空间使您能够轻松地启用XML中的映射功能

示例90.配置MongoDB映射支持的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"/>

  <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.LoggingEventListener"/>

</beans>
```

该 `base-package` 属性告诉它在哪里扫描用注释注释的

`@org.springframework.data.mongodb.core.mapping.Document` 类。

12.4. 基于元数据的映射

要充分利用Spring Data / MongoDB支持中的对象映射功能，您应该使用注释来注释映射的对象 `@Document`。虽然映射框架没有必要具有此注释（即使没有任何注释也能正确映射您的POJO），它允许类路径扫描程序查找并预处理您的域对象以提取必要的元数据。如果您不使用此注释，那么您的应用程序在第一次存储域对象时会受到轻微的性能影响，因为映射框架需要构建其内部元数据模型，以便了解域对象的属性以及如何坚持他们

示例91. 域对象示例

JAVA

```
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。



自动创建索引仅适用于注释的类型 `@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` - 应用于外地一级以标示用于身份目的的领域。
- `@Document` - 应用于类级别，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` - 应用于该字段以指示它将使用`com.mongodb.DBRef`进行存储。
- `@Indexed` - 应用于现场一级，以描述如何对该领域进行索引。
- `@CompoundIndex` - 应用于类型级别以声明复合索引
- `@GeoSpatialIndexed` - 应用于现场一级，以描述如何对该领域进行地理指标。
- `@TextIndexed` - 应用于字段级标记要包含在文本索引中的字段。
- `@Language` - 应用于字段级别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，此注释将将其应用的字段从存储在数据库中排除
- `@PersistenceConstructor` - 标记一个给定的构造函数 - 即使是一个包保护的 - 在从数据库实例化对象时使用。构造函数参数通过名称映射到检索的DBObject中的键值。
- `@Value` - 这个注释是Spring框架的一部分。在映射框架内，它可以应用于构造函数参数。这允许您使用Spring表达式语言语句来转换在数据库中检索的键值，然后再用它来构造一个域对象。为了引用给定文档的属性，必须使用以下表达式: `@Value("#root.myProperty")` where `root` 指向给定文档的根。
- `@Field` - 应用在字段级别，并描述字段的名称，因为它将在MongoDB BSON文档中表示，从而允许名称与该类的字段名不同。
- `@Version` - 应用于现场级用于乐观锁定，并检查保存操作的修改。初始值是 `zero` 每次更新时自动触发的。

映射元数据基础设施在一个独立的，与数据技术无关的spring-data-commons项目中定义。MongoDB支持中使用的特定子类支持基于注释的元数据。如果有需求，也可以采取其他策略。

这是一个更复杂的映射的例子。

```
JAVA
@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 `@PersistenceConstructor` 构造。用于构造函数参数的值以下列方式解决：

- 如果使用注释对参数进行注释，`@Value` 则会给出给定的表达式，并将结果用作参数值。
- 如果Java类型的名称与输入文档的给定字段匹配，则其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。仅当参数名称信息存在于Java `.class` 文件中时才可以使用，该文件可以通过使用调试信息编译源或 `-parameters` 在Java 8中使用javac 的新命令行开关来实现。
- 否则 `MappingException` 将抛出一个指示给定的构造函数参数无法绑定的信号。

```
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 omitted  
}  
  
DBObject input = new BasicDBObject("id", "4711");  
input.put("unitPrice", 2.5);  
input.put("qty", 5);  
OrderItem item = converter.read(OrderItem.class, input);
```

JAVA



如果给定的属性路径无法解析，参数注释中的 `@Value` Spel表达式将 `quantity` 返回到该值 `0`。

使用注释的其他示例 `@PersistenceConstructor` 可以在 [MappingMongoConverterUnitTests](https://github.com/spring-projects/spring-data-mongodb/blob/master/spring-data-mongodb/src/test/java/org/springframework/data/mongodb/core/convert/MappingMongoConverterUnitTests.java) (https://github.com/spring-projects/spring-data-mongodb/blob/master/spring-data-mongodb/src/test/java/org/springframework/data/mongodb/core/convert/MappingMongoConverterUnitTests.java) 测试套件中找到。

12.4.3. 复合指数

还支持复合索引。它们在类级别定义，而不是单个属性。



复合索引对于提高涉及多个字段标准的查询的性能非常重要

这是一个 `lastName` 以升序和 `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;

}
```

JAVA

12.4.4. 文本索引



默认情况下，mongodb v.2.4禁用文本索引功能。

创建文本索引可以将多个字段累加到可搜索的全文索引中。每个集合只能有一个文本索引，所以所有标记的字段 `@TextIndexed` 都被合并到这个索引中。可以对属性进行加权，以影响文档分数的排名结果。文本索引的默认语言是英语，将默认语言设置 `@Document(language="spanish")` 为所需的任何语言。使用一个名为 `language` 或 `@Language` 允许在每个文档基础上定义语言覆盖的属性。

示例93. 示例文本索引使用

```
@Document(language = "spanish")
class SomeEntity {

    @TextIndexed String foo;

    @Language String lang;

    Nested nested;
}

class Nested {

    @TextIndexed(weight=5) String bar;
    String roo;
}
```

JAVA

12.4.5. 使用DBRefs

映射框架不必存储嵌入在文档中的子对象。您还可以单独存储它们，并使用DBRef来引用该文档。当从MongoDB加载该对象时，这些引用将被热切地解决，您将获得一个映射的对象，看起来和它们已被存储在主文档中一样。

以下是使用DBRef引用独立于引用的对象存在的特定文档的示例（为简洁起见，这两个类都显示在内）：

JAVA

```

@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;
}

```

没有必要使用这样的东西 `@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，将会有 `DBRefs` 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。 `save` 在 `Person` 对象上调用不会自动将 `Account` 对象保存在属性 `accounts` 中。

12.4.6. 映射框架事件

在映射过程的整个生命周期中触发事件。这在“生命周期事件”部分中有描述。

在 `Spring ApplicationContext` 中简单地声明这些 `bean` 将会在调度事件时调用它们。

12.4.7. 用显式转换器覆盖映射

存储和查询对象时，让一个 `MongoConverter` 实例处理所有 `Java` 类型到 `DBObject` 的映射是很方便的。但是，有时您可能希望 `MongoConverter` 做大部分工作，但允许您选择性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，请使用 `MongoConverter` 注册一个或多个 `org.springframework.core.convert.converter.Converter` 实例。



Spring 3.0 引入了一个提供一般类型转换系统的 `core.convert` 包。这在 `Spring` 参考文档部分标题为“[Spring Type Conversion](http://docs.spring.io/spring/docs/4.3.8.RELEASE/spring-framework-reference/html/validation.html#core-convert)”

(<http://docs.spring.io/spring/docs/4.3.8.RELEASE/spring-framework-reference/html/validation.html#core-convert>) 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。

下面是一个从 `DBObject` 转换为 `Person POJO` 的 `Spring Converter` 实现的例子。

```
@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;
    }
}
```

十字架支持

有时您需要将数据存储多个数据存储中，并且这些数据存储可以是不同的类型。一个可能是关系，而另一个文档存储。对于这种用例，我们在MongoDB支持中创建了一个单独的模块，用于处理我们称之为跨店支持。目前的实现是基于JPA作为关系数据库的驱动程序，我们允许将实体中的选择字段存储在Mongo数据库中。除了允许您将数据存储两个存储中，我们还将使用关系数据库的事务生命周期来协调非事务性MongoDB存储的持久性操作。

13.1. 交叉存储配置

假设您有一个工作的JPA应用程序，并希望为MongoDB添加一些跨存储持久性。您需要添加什么配置？

首先，您需要在模块上添加依赖关系。使用Maven这是通过添加一个依赖关系到你的pom:

示例94. 具有spring-data-mongodb-cross-store依赖关系的Maven pom.xml示例

```
<?xml version="1.0" encoding="UTF-8"?>
<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>
```

一旦完成，我们需要为项目启用AspectJ。跨店支持使用AspectJ方面实现，因此通过启用编译时间AspectJ支持跨商店功能将可用于您的项目。在Maven中，您可以向pom的<build>部分添加一个附加插件：

示例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。以下XML代码段应该添加到您的应用程序上下文中：

示例96. 使用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.MongoDocumentBacking"
    factory-method="aspectOf">
    <property name="changeSetPersister" ref="mongoChangeSetPersister"/>
  </bean>
  <bean id="mongoChangeSetPersister"
    class="org.springframework.data.persistence.document.mongo.MongoChangeSetPersister">
    <property name="mongoTemplate" ref="mongoTemplate"/>
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
  </bean>

  ...

</beans>
```

13.2. 写十字架应用程序

我们假设你有一个工作的JPA应用程序，所以我们只会涵盖在Mongo数据库中保留部分实体所需的额外步骤。首先，您需要确定要保留的字段。它应该是域类，并遵循前面章节中涵盖的Mongo映射支持的一般规则。您要保存在MongoDB中的字段应使用 `@RelatedDocument` 注释注释。这真的是你需要做的！十字架方面的照顾其余。这包括标记字段，`@Transient` 因此不会使用JPA持久化，跟踪对字段值进行的任何更改，并在成功的事务完成时将其写入数据库，在首次使用该值时从MongoDB加载文档你的申请。`@RelatedDocument`

示例97. 使用@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
}
```

JAVA

示例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;
    }
}
```

JAVA

一旦SurveyInfo设置在上面配置的MongoTemplate上面的Customer对象用于保存SurveyInfo，并且有关JPA Entity的一些元数据存储在以完全限定名称的JPA Entity类命名的MongoDB集合中。以下代码：

示例99 使用JPA实体的代码示例配置为跨存储持久性的实体

JAVA

```
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);
```

执行上述代码会导致存储在MongoDB中的以下JSON文档。

示例100. 存储在MongoDB中的JSON文档示例

JAVASCRIPT

```
{ "_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" }
```

日志记录支持

在maven模块“spring-data-mongodb-log4j”中提供了Log4j的追加器。注意，没有依赖于其他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.MongoLog4jAppender
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
```

除了主机和端口，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。这是为了支持将日志信息分组到与特定时间段对应的集合中的常规约定，例如每天的集合。

还有一个 `applicationId` 被放入存储的消息中。：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

MongoDB Log4j appender可以配置为使用用户名/密码认证。使用指定的数据库执行认证。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
# ...
```



身份验证失败会导致日志记录中的异常，并传播到记录方法的调用者。

JMX支持

MongoDB的JMX支持在单个MongoDB服务器实例的管理数据库上公布执行“serverStatus”命令的结果。它还暴露了一个管理MBean MongoAdmin，它将允许您执行管理操作，如删除或创建数据库。JMX功能基于Spring框架中提供的JMX功能集。有关详细信息，请参阅[此处](http://docs.spring.io/spring/docs/4.3.8.RELEASE/spring-framework-reference/html/jmx.html) (http://docs.spring.io/spring/docs/4.3.8.RELEASE/spring-framework-reference/html/jmx.html)。

15.1。MongoDB JMX配置

Spring的Mongo命名空间使您能够轻松启用JMX功能

示例101.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" />

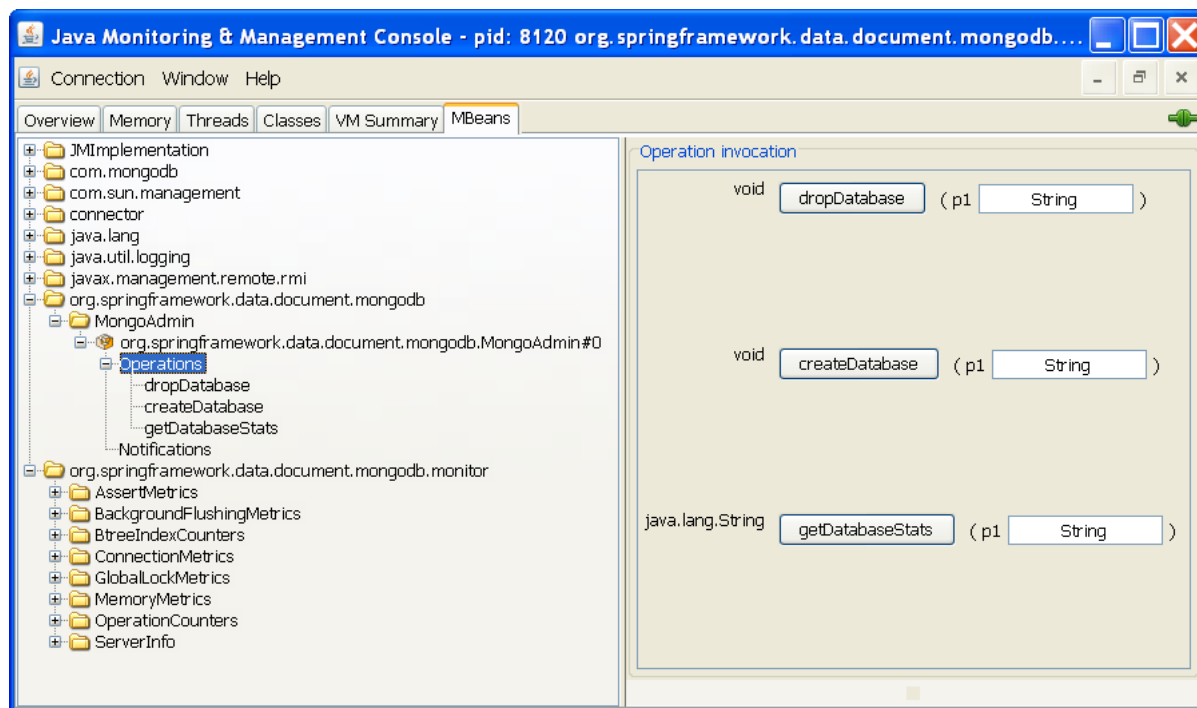
</beans>
```

这将暴露几个MBean

- AssertMetrics
- BackgroundFlushingMetrics
- BtreeIndexCounters
- ConnectionMetrics

- GlobalLockMetrics
- MemoryMetrics
- OperationCounters
- ServerInfo
- MongoAdmin

这在JConsole的屏幕快照中显示



16. MongoDB 3.0支持

Spring数据MongoDB允许在连接到运行*MMAP.v1*的MongoDB 2.6 / 3.0服务器或使用*MMAP.v1*或*WiredTiger*存储引擎的MongoDB服务器3.0时使用MongoDB Java驱动程序第2代和第3代。



请参阅驱动程序和数据库特定文档，了解这些文档之间的主要区别。



使用3.x MongoDB Java驱动程序不再有效的操作已在Spring Data中被弃用，并将在随后的版本中删除。

16.1. MongoDB使用Spring数据MongoDB 3.0

16.1.1. 配置选项

某些配置选项已被更改/删除为*mongo-java*驱动程序。以下选项将被忽略使用第3代驱动程序：

- `autoConnectRetry`
- `maxAutoConnectRetryTime`
- `slaveOk`

通常建议使用 `<mongo:mongo-client ... />` 和 `<mongo:client-options ... />` 元素，而不是 `<mongo:mongo ... />` 在进行基于XML的配置时，因为这些元素将仅为您提供对3代java驱动程序有效的属性。

```
<?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>
```

XML

16.1.2. WriteConcern和WriteConcernChecking

的 `WriteConcern.NONE`，其中已经使用如春的数据MongoDB的默认情况下，在3.0被删除。因此在MongoDB 3环境 `WriteConcern` 中将被默认 `WriteConcern.UNACKNOWLEDGED`。在 `WriteResultChecking.EXCEPTION` 启用的情况下，`WriteConcern` 将被更改 `WriteConcern.ACKNOWLEDGED` 为写入操作，否则执行期间的错误将不会被正确抛出，因为驱动程序根本没有提出。

16.1.3. 认证

MongoDB Server 3在连接到DB时更改了认证模型。因此，可用于身份验证的某些配置选项不再有效。请使用 `MongoClient` 特定选项设置凭据，`MongoCredential` 以提供认证数据。

```

@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())));
    }
}

```

为了使用身份验证与XML配置使用attribute `credentials` 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. 其他要注意的事情

本节介绍使用3.0驱动程序时要注意的其他事项。

- `IndexOperations.resetIndexCache()` 不再受支持。
- 任何一个 `MapReduceOptions.extraOption` 都被忽略了。
- `WriteResult` 不再保存错误信息，但会抛出异常。
- `MongoOperations.executeInSession(...)` 不再叫 `requestStart / requestDone`。
- 索引名称生成已经成为驱动程序的内部操作，我们仍然使用2.x模式来生成名称。
- 一些异常消息在第2代和第3代服务器之间以及 *MMap.v1* 和 *WiredTiger* 存储引擎之间有所不同。

附录

附录A：命名空间参考

<repositories />元素

该 `<repositories />` 元素触发了Spring Data存储库基础结构的设置。最重要的属性是 `base-package` 定义要扫描Spring数据存储库接口的包。^[3]

表9. 属性

名称	描述
<code>base-package</code>	定义用于在自动检测模式下扩展*存储库（实际接口由特定的Spring数据模块确定）的存储库接口进行扫描的软件包。所配置的软件包以下的所有软件包也将被扫描。通配符是允许的。
<code>repository-impl-postfix</code>	定义后缀自动检测自定义存储库实现。其名称以配置的后缀结尾的类将被视为候选。默认为 <code>Impl</code> 。
<code>query-lookup-strategy</code>	确定用于创建查找器查询的策略。有关详细信息，请参阅查询查询策略。默认为 <code>create-if-not-found</code> 。
<code>named-queries-location</code>	定义查找包含外部定义查询的“属性”文件的位置。
<code>consider-nested-repositories</code>	控制是否应考虑是否嵌套存储库接口定义。默认为 <code>false</code> 。

附录B : Populators命名空间参考

<populator />元素

该 <populator /> 元素允许通过Spring数据库基础架构填充数据存储。^[4]

属性

名称	描述
locations	在哪里可以找到从存储库读取对象的文件。

附录C：存储库查询关键字

支持的查询关键字

下表列出了Spring数据库查询推导机制通常支持的关键字。但是，请查阅特定于商店的文档，了解支持的关键字的确切列表，因为某些商店中可能不支持这些列表。

表11. 查询关键字


逻辑关键字	关键词表达式
AND	And
OR	Or
AFTER	After , IsAfter
BEFORE	Before , IsBefore
CONTAINING	Containing , IsContaining , Contains
BETWEEN	Between , IsBetween
ENDING_WITH	EndingWith , IsEndingWith , EndsWith
EXISTS	Exists
FALSE	False , IsFalse
GREATER_THAN	GreaterThan , IsGreaterThan
GREATER_THAN_EQUALS	GreaterThanOrEqualTo , IsGreaterThanOrEqualTo
IN	In , IsIn
IS	Is , Equals （或没有关键词）
IS_NOT_NULL	NotNull , IsNotNull
IS_NULL	Null , IsNull
LESS_THAN	LessThan , IsLessThan
LESS_THAN_EQUAL	LessThanOrEqualTo , IsLessThanOrEqualTo
LIKE	Like , IsLike
NEAR	Near , IsNear
NOT	Not , IsNot
NOT_IN	NotIn , IsNotIn
NOT_LIKE	NotLike , IsNotLike

逻辑关键字	关键词表达式
REGEX	Regex , MatchesRegex , Matches
STARTING_WITH	StartingWith , IsStartingWith , StartsWith
TRUE	True , IsTrue
WITHIN	Within , IsWithin

附录D：存储库查询返回类型

支持的查询返回类型

下表列出了Spring Data仓库通常支持的返回类型。但是，请查阅特定于商店的文档，以获取支持的返回类型的确切列表，因为某些商店中可能不支持这些列表。



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

表12. 查询返回类型

返回类型	描述
<code>void</code>	不表示返回值。
原语	Java原语。
包装类型	Java包装器类型。
<code>T</code>	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 <code>null</code> 返回。多个结果将触发 <code>IncorrectResultSizeDataAccessException</code> 。
<code>Iterator<T></code>	的 <code>Iterator</code> 。
<code>Collection<T></code>	一 <code>Collection</code> 。
<code>List<T></code>	一 <code>List</code> 。
<code>Optional<T></code>	Java 8或番石榴 <code>Optional</code> 。期望查询方法最多返回一个结果。如果没有找到 <code>Optional.empty()</code> / <code>Optional.absent()</code> 返回结果。多个结果将触发 <code>IncorrectResultSizeDataAccessException</code> 。
<code>Option<T></code>	Scala或JavaSlang <code>Option</code> 类型。与 <code>Optional</code> 上述Java 8相似的行为。
<code>Stream<T></code>	Java 8 <code>Stream</code> 。
<code>Future<T></code>	一 <code>Future</code> 。期待使用 <code>@Async</code> 注释的方法，并且需要启用Spring的异步方法执行功能。
<code>CompletableFuture<T></code>	Java 8 <code>CompletableFuture</code> 。期待使用 <code>@Async</code> 注释的方法，并且需要启用Spring的异步方法执行功能。
<code>ListenableFuture</code>	一 <code>org.springframework.util.concurrent.ListenableFuture</code> 。期待使用 <code>@Async</code> 注释的方法，并且需要启用Spring的异步方法执行功能。
<code>Slice</code>	大小的数据块与信息是否有更多的数据可用。需要一个 <code>Pageable</code> 方法参数。
<code>Page<T></code>	A <code>Slice</code> 附加信息，例如总结果数。需要一个 <code>Pageable</code> 方法参数。
<code>GeoResult<T></code>	带有附加信息的结果条目，例如到参考位置的距离。
<code>GeoResults<T></code>	的列表 <code>GeoResult<T></code> 与其他信息，到参考位置例如平均距离。

返回类型	描述
GeoPage<T>	甲 Page 带 GeoResult<T> ， 例如平均距离的参考位置。

-
- 1. [JavaConfig在Spring参考文档](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html#beans-java)(http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html#beans-java)
 - 2. 春天HATEOAS(<https://github.com/SpringSource/spring-hateoas>)<https://github.com/SpringSource/spring-hateoas>
 - 3. 请参阅XML配置
 - 4. 请参阅XML配置

版本1.10.3.RELEASE
最后更新2017-04-19 20:01:47 MESZ