JPA □□□

Version 2.3.6.RELEASE, 2021-05-14

Table of Contents

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

Chapter 1. 简介

本节介绍通过以下两种方式配置 Spring Data JPA 的基础知识:

- “Spring 命名空间” (XML 配置)
- “基于注解的配置” (Java 配置)

1.1. Spring 命名空间

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

Example 1. 使用命名空间建立 *JPA* 存储库

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

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

</beans>
```

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

1.1.1. 自定义命名空间属性

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

Table 1. 自定义 *JPA* 特定于 `repositories` 元素的属性

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

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

1.2. 基于注解的配置

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

Example 2. 使用 *JavaConfig* 的 *Spring Data JPA* 存储库

```
@Configuration
@EnableJpaRepositories
@EnableTransactionManagement
class ApplicationConfig {

  @Bean
  public DataSource dataSource() {

    EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
    return builder.setType(EmbeddedDatabaseType.HSQL).build();
  }

  @Bean
  public LocalContainerEntityManagerFactoryBean entityManagerFactory() {

    HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
    vendorAdapter.setGenerateDdl(true);

    LocalContainerEntityManagerFactoryBean factory = new
LocalContainerEntityManagerFactoryBean();
    factory.setJpaVendorAdapter(vendorAdapter);
    factory.setPackagesToScan("com.acme.domain");
    factory.setDataSource(dataSource());
    return factory;
  }

  @Bean
  public PlatformTransactionManager transactionManager(EntityManagerFactory
entityManagerFactory) {

    JpaTransactionManager txManager = new JpaTransactionManager();
    txManager.setEntityManagerFactory(entityManagerFactory);
    return txManager;
  }
}
```

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

1.3. 引导模式

默认情况下,Spring Data JPA 存储库是默认的 Spring Bean。它们是单例作用域,并且已被初始化。在启动期间,它们已经与 JPA `EntityManager` 进行交互,以进行验证和元数据分析。Spring 框架在后台线程中支持 JPA `EntityManagerFactory` 的初始化,因为该过程通常在 Spring 应用程序中占用大量启动时间。为了有效地利用后台初始化,我们需要确保 JPA 存储库尽可能早地初始化。

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

- `DEFAULT` (默认值) — 急切地实例化存储库,除非使用 `@Lazy` 显式注解。仅当没有任何客户 Bean 需要存储库实例时,`lazification` 才有效,因为这将需要初始化存储库 bean。
- `LAZY` — 隐式地声明所有存储库 bean 都是惰性的,并且还使创建的惰性初始化代理被注入到客户端 bean 中。这意味着,如果客户端 bean 仅将实例存储在字段中并且在初始化期间不使用存储库,则不会实例化存储库。首次与存储库交互时,将初始化并验证存储库实例。
- `DEFERRED` — 基本与 `LAZY` 相同,但会响应 `ContextRefreshedEvent` 触发存储库初始化,以便在应用程序完全启动之前验证存储库。

1.3.1. 推荐建议

如果您不使用默认 Bootstrap 模式的异步 JPA 引导。

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

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

Chapter 2. 持久化实体

本节描述如何使用 Spring Data JPA 持久化 (保存) 实体.

2.1. 保存实体

可以使用 `CrudRepository.save(…)` 方法执行保存实体. 它通过使用基础 JPA `EntityManager` 持久化或合并给定实体. 如果实体还没有持久化,Spring Data JPA 会通过调用 `entityManager.persist(…)` 方法来保存实体. 否则,它将调用 `entityManager.merge(…)` 方法.

2.1.1. 实体状态检测策略

Spring Data JPA 提供以下策略来检测实体是否为新实体:

1. Version-Property 和 Id-Property 检查 (默认) : 默认情况下,Spring Data JPA 首先检查是否存在非基本类型的 Version-property. 如果存在,则将该实体视为新实体 (如果该值为 `null`) . 没有这样的版本属性,Spring Data JPA 会检查给定实体的标识符属性. 如果标识符属性为 `null`,则假定该实体为新实体. 否则,它不是新的.
2. 实现 `Persistable`: 如果实体实现 `Persistable`,则 Spring Data JPA 将新检测委托给该实体的 `isNew(…)` 方法. 有关详细信息,请参见 JavaDoc .
3. 实现 `EntityInformation`: 通过创建 `JpaRepositoryFactory` 的子类并相应地重写 `getEntityInformation(…)` 方法,可以自定义 `SimpleJpaRepository` 实现中使用的 `EntityInformation` 抽象. 然后,您必须将 JpaRepositoryFactory 的自定义实现注册为 Spring bean. 请注意,这几乎没有必要. 有关详细信息,请参见 JavaDoc .

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

Example 3. 具有手动分配的标识符的实体的基类

```
@MappedSuperclass
public abstract class AbstractEntity<ID> implements Persistable<ID> {

  @Transient
  private boolean isNew = true; ①

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

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

  // More code…
}
```

① 声明一个标志以保持新状态. 暂时的,因此它不会持久化到数据库中.

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

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

Chapter 3. 查询方法

本节描述了使用 Spring Data JPA 创建查询的各种方法.

3.1. 查询策略

JPA 模块支持手动将查询定义为 String 或从方法名称扩展查询.

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

3.1.1. 声明查询

尽管从方法名扩展一个查询很方便,但可能会遇到这样一种情况,即方法名解析器不支持一个人想使用的关键字,或者方法名变的丑陋. 因此,您可以通过命名约定使用 JPA 命名查询 (有关更多信息,请参见使用JPA 命名查询) , 或者通过 `@Query` 注解您的查询方法 (有关详细信息,请参见使用 `@Query`) .

3.2. 查询创建

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

Example 4. 通过方法名称创建查询

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

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

我们从中使用 JPA 标准 API 创建查询,但是从本质上讲,这将转换为以下查询: `select u from User u where u.emailAddress = ?1 and u.lastname = ?2`. Spring Data JPA 进行属性检查并遍历嵌套的属性,如 “属性表达式” 中所述.

下表描述了 JPA 支持的关键字以及包含该关键字的方法所转换的含义:

Table 2. 方法名称中受支持的关键字

关键字	**Sample**	**JPQL snippet**
`Distinct`	`findDistinctByLastnameAndFirstname`	`select distinct … where x.lastname = ?1 and x.firstname = ?2`
`And`	`findByLastnameAndFirstname`	`… where x.lastname = ?1 and x.firstname = ?2`
`Or`	`findByLastnameOrFirstname`	`… where x.lastname = ?1 or x.firstname = ?2`
`Is`, `Equals`	`findByFirstname`,`findByFirstnameIs`,`findByFirstnameEquals`	`… where x.firstname = ?1`
`Between`	`findByStartDateBetween`	`… where x.startDate between ?1 and ?2`

关键字	Sample	JPQL snippet
LessThan	findByAgeLessThan	… where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	… where x.age <= ?1
GreaterThan	findByAgeGreaterThan	… where x.age > ?1
GreaterThanEqual	findByAgeGreaterThanEqual	… where x.age >= ?1
After	findByStartDateAfter	… where x.startDate > ?1
Before	findByStartDateBefore	… where x.startDate < ?1
IsNull, Null	findByAge(Is)Null	… where x.age is null
IsNotNull, NotNull	findByAge(Is)NotNull	… where x.age not null
Like	findByFirstnameLike	… where x.firstname like ?1
NotLike	findByFirstnameNotLike	… where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	… where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	… where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	… where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	… where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	… where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	… where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> ages)	… where x.age not in ?1
True	findByActiveTrue()	… where x.active = true
False	findByActiveFalse()	… where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	… where UPPER(x.firstname) = UPPER(?1)

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

3.3. 使用 JPA 命名查询

这些示例使用 <named-query /> 元素和 @NamedQuery 注解。这些配置元素的查询必须使用 JPA 查询语言进行定义。当然,您也可以使用 <named-native-query /> 或 @NamedNativeQuery 。这些元素使您可以通过在没有数据库平台独立性来在本地 SQL 中定义查询。

3.3.1. XML 命名查询定义

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

Example 5. XML 命名查询配置

```
<named-query name="User.findByLastname">
  <query>select u from User u where u.lastname = ?1</query>
</named-query>
```

该查询具有一个特殊名称,该名称用于在运行时解析它.

3.3.2. 基于注解的配置

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

Example 6. 基于注解的命名查询配置

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

}
```

3.3.3. 声明接口

要允许执行这些命名查询,请按以下方式指定 `UserRepository`:

*Example 7. UserRepository*中的查询方法声明

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

  List<User> findByLastname(String lastname);

  User findByEmailAddress(String emailAddress);
}
```

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

3.4. 使用 `@Query`

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

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

以下示例显示使用 `@Query` 注解创建的查询:

Example 8. 使用 `@Query` 在查询方法中声明查询

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

  @Query("select u from User u where u.emailAddress = ?1")
  User findByEmailAddress(String emailAddress);
}
```

3.4.1. 使用高级 `LIKE` 表达式

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

*Example 9. @Query*中的高级 `like` 表达式

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

  @Query("select u from User u where u.firstname like %?1")
  List<User> findByFirstnameEndsWith(String firstname);
}
```

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

3.4.2. 本地查询

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

Example 10. 使用 `@Query` 在查询方法中声明一个本地查询

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

  @Query(value = "SELECT * FROM USERS WHERE EMAIL_ADDRESS = ?1", nativeQuery =
true)
  User findByEmailAddress(String emailAddress);
}
```

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

Example 11. 使用 `@Query` 在查询方法中声明本地 *count* 查询以进行分页

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

  @Query(value = "SELECT * FROM USERS WHERE LASTNAME = ?1",
    countQuery = "SELECT count(*) FROM USERS WHERE LASTNAME = ?1",
    nativeQuery = true)
  Page<User> findByLastname(String lastname, Pageable pageable);
}
```

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

3.5. 使用排序

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

使用任何不可引用的路径表达式都会导致异常.

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

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

Example 12. 使用 `Sort` 和 `JpaSort`

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

  @Query("select u from User u where u.lastname like ?1%")
  List<User> findByAndSort(String lastname, Sort sort);

  @Query("select u.id, LENGTH(u.firstname) as fn_len from User u where u.lastname
like ?1%")
  List<Object[]> findByAsArrayAndSort(String lastname, Sort sort);
}

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

① 指向 domain 模型中属性的有效 `Sort` 表达式.

② 包含函数调用的无效 `Sort` Throws 异常.

③ 有效 `Sort` 包含明显不安全的 `Order`.

④ 指向别名函数的有效 `Sort` 表达式.

3.6. 使用命名参数

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

Example 13. 使用命名参数

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

  @Query("select u from User u where u.firstname = :firstname or u.lastname =
:lastname")
  User findByLastnameOrFirstname(@Param("lastname") String lastname,
                                 @Param("firstname") String firstname);
}
```

方法参数根据其在定义的查询中的顺序进行切换。

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

3.7. 使用 SpEL 表达式

从 Spring Data JPA 1.4 版开始,我们支持在使用 `@Query` 定义的手动定义的查询中使用受限的 SpEL 模板表达式。查询执行后,将根据一组预定义的变量对这些表达式进行求值。Spring Data JPA 支持一个名为 `entityName` 的变量。它的用法是 `select x from #{#entityName} x`。它插入与给定存储库关联的 domain 类型的 `entityName`。实体名称的解析如下:如果 domain 类型已在 `@Entity` 注解上设置了名称属性,则将其使用。否则,将使用 domain 类型的简单类名。

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

Example 14. 在存储库查询方法中使用*SpEL*表达式*-entityName*

```
@Entity
public class User {

  @Id
  @GeneratedValue
  Long id;

  String lastname;
}

public interface UserRepository extends JpaRepository<User,Long> {

  @Query("select u from #{#entityName} u where u.lastname = ?1")
  List<User> findByLastname(String lastname);
}
```

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

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

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

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

Example 15. 在 *repository* 查询方法中使用 *SpEL* 表达式-具有继承的 `entityName`

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

@Entity
public class ConcreteType extends AbstractMappedType { … }

@NoRepositoryBean
public interface MappedTypeRepository<T extends AbstractMappedType>
  extends Repository<T, Long> {

  @Query("select t from #{#entityName} t where t.attribute = ?1")
  List<T> findAllByAttribute(String attribute);
}

public interface ConcreteRepository
  extends MappedTypeRepository<ConcreteType> { … }
```

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

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

Example 16. 在存储库查询方法中使用 *SpEL* 表达式-访问参数.

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

对于 `like`,通常需要将 % 附加到 `String` 值参数的开头或结尾. 这可以通过在绑定参数标记或 SpEL 表达式上附加或前缀 % 来完成. 以下示例再次说明了这一点.

Example 17. 在 *repository* 查询方法中使用 *SpEL* 表达式-通配符快捷方式.

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

如果使用 `like` 条件的值来自不安全来源,则应清除这些值,以使它们不能包含任何通配符,从而使攻击者可以选择比其应有的能力更多的数据. 为此,在 SpEL 上下文中可以使用 `escape(String)` 方法. 它在第一个参数中的 `_` 和 % 的所有实例之前加上第二个参数中的单个字符. 与 JPQL 中提供的 `like` 表达式的转义子句和标准 SQL

结合使用,可以轻松清除绑定参数.

Example 18. 在存储库查询方法中使用 *SpEL* 表达式-清理输入值.

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

在存储库接口中给出此方法声明后, `findContainingEscaped("Peter_")` 将找到 `Peter_Parker` 而不是 `Peter Parker`. 可以通过设置 `@EnableJpaRepositories` 注解的 `escapeCharacter` 来配置所使用的转义字符. 请注意,该方法 `escape(String)` 可用 在SpEL上下文中,仅将转义 SQL 和 JPQL 标准通配符 `_` 和 %,如果基础数据库或 JPA 实现支持其他通配符,则将不会转义这些通配符.

3.8. 修改查询

前面所有部分均描述了如何声明查询以访问给定实体或实体集合. 您可以使用 “Spring数据存储库的自定义实现” 中介绍的功能来添加自定义修改行为. 由于此方法对于全面的定制功能是可行的,因此可以通过使用 `@Modifying` 注解查询方法来修改仅需要参数绑定的查询,如以下示例所示:

Example 19. 声明操作查询

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

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

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

3.8.1. 扩展删除查询

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

Example 20. 使用扩展的删除查询

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

  void deleteByRoleId(long roleId);

  @Modifying
  @Query("delete from User u where u.role.id = ?1")
  void deleteInBulkByRoleId(long roleId);
}
```

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

为了确保生命周期查询被实际调用,调用 `deleteByRoleId(…)` 会执行一个查询,然后逐个删除返回的实例,以便持久性提供程序实际上可以在这些实体上调用 `@PreRemove` 回调。

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

3.9. 应用查询提示

要将 JPA 查询提示应用于在存储库接口中声明的查询,可以使用 `@QueryHints` 注解。 它需要一个JPA `@QueryHint` 注解加上一个布尔标志,以潜在地禁用应用于应用分页时触发的附加计数查询的提示,如以下示例所示:

Example 21. 将 `QueryHints` 与存储库方法一起使用

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

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

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

3.10. 配置 Fetch- 和 LoadGraphs

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

Example 22. 在一个实体上定义一个命名实体图。

```
@Entity
@NamedEntityGraph(name = "GroupInfo.detail",
  attributeNodes = @NamedAttributeNode("members"))
public class GroupInfo {

  // default fetch mode is lazy.
  @ManyToMany
  List<GroupMember> members = new ArrayList<GroupMember>();

  …
}
```

以下示例显示如何在存储库查询方法上引用命名实体图:

Example 23. 在存储库查询方法上引用命名实体图定义

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

  @EntityGraph(value = "GroupInfo.detail", type = EntityGraphType.LOAD)
  GroupInfo getByGroupName(String name);

}
```

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

Example 24. 在存储库查询方法上使用 *AD-HOC* 实体图定义。

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

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

}
```

3.11. 投影

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

想象一下一个存储库和聚合根类型,例如以下示例:

Example 25. 一个示例集合和存储库

```
class Person {

  @Id UUID id;
  String firstname, lastname;
  Address address;

  static class Address {
    String zipCode, city, street;
  }
}

interface PersonRepository extends Repository<Person, UUID> {

  Collection<Person> findByLastname(String lastname);
}
```

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

3.11.1. 基于接口的投影

将查询结果限制为仅 `name` 属性的最简单方法是声明一个接口,该接口暴露要读取的属性的 get 方法,如以下示例所示:

Example 26. 一个投影接口来检索属性的子集

```
interface NamesOnly {

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

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

Example 27. 使用基于接口的投影和查询方法的存储库

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

  Collection<NamesOnly> findByLastname(String lastname);
}
```

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

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

Example 28. 一个投影接口来检索属性的子集

```
interface PersonSummary {

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

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

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

封闭投影

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

Example 29. 一个封闭的投影

```
interface NamesOnly {

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

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

打开投影

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

Example 30. 一个 打开投影

```
interface NamesOnly {

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

在 `target` 变量中提供了支持投影的合计根. 使用 `@Value` 的投影接口是开放式投影. 在这种情况下,Spring Data 无法应用查询执行优化,因为SpEL表达式可以使用聚合根的任何属性.

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

Example 31. 使用默认方法自定义逻辑的投影接口

```
interface NamesOnly {

  String getFirstname();
  String getLastname();

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

这种方法要求您能够完全基于投影接口上暴露的其他 get 方法来实现逻辑。第二个更灵活的选择是在 Spring bean 中实现自定义逻辑,然后从 SpEL 表达式中调用该自定义逻辑,如以下示例所示:

Example 32. Sample Person 对象

```
@Component
class MyBean {

  String getFullName(Person person) {
    …
  }
}

interface NamesOnly {

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

请注意 SpEL 表达式如何引用 `myBean` 并调用 `getFullName(…)` 方法,并将投影目标作为方法参数转发。SpEL 表达式评估支持的方法也可以使用方法参数,然后可以从表达式中引用这些参数。方法参数可通过名为 `args` 的对象数组获得。下面的示例演示如何从 `args` 数组获取方法参数:

Example 33. Sample Person 对象

```
interface NamesOnly {

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

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

3.11.2. 基于类的投影 (DTO)

定义投影的另一种方法是使用值类型DTO (数据传输对象) ,该类型DTO保留应该被检索的字段的属性. 这些DTO类型可以以与使用投影接口完全相同的方式使用,除了没有代理发生和不能应用嵌套投影之外.

如果存储通过限制要加载的字段来优化查询执行,则要加载的字段由暴露的构造函数的参数名称确定.

以下示例显示了一个预计的DTO:

Example 34. 一个投影的DTO

```
class NamesOnly {

  private final String firstname, lastname;

  NamesOnly(String firstname, String lastname) {

    this.firstname = firstname;
    this.lastname = lastname;
  }

  String getFirstname() {
    return this.firstname;
  }

  String getLastname() {
    return this.lastname;
  }

  // equals(…) and hashCode() implementations
}
```

避免投影*DTO*的样板代码

您可以使用 Project Lombok 大大简化DTO的代码,该项目提供了 `@Value` 注解 (不要与前面的界面示例中显示的Spring的 `@Value` 注解混淆) . 如果您使用 Project Lombok 的 `@Value` 注解,则前面显示的示例DTO将变为以下内容:

```
@Value
class NamesOnly {
    String firstname, lastname;
}
```

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

3.11.3. 动态投影

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

Example 35. 使用动态投影参数的存储库

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

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

通过这种方式,该方法可以按原样或应用投影来获得聚合,如下例所示:

Example 36. 使用带有动态投影的存储库

```
void someMethod(PersonRepository people) {

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

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

Chapter 4. 存储过程

JPA 2.1 规范引入了对使用 JPA 标准查询 API 调用存储过程的支持。我们引入了 `@Procedure` 注解,用于在存储库方法上声明存储过程元数据。

下面的示例使用以下存储过程:

Example 37. HSQL DB中 `plus1inout` 过程的定义。

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

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

Example 38. 实体上的 `StoredProcedure` 元数据定义

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

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

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

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

下面的示例演示如何引用显式映射的过程:

Example 39. 在数据库中引用名称为 "plus1inout" 的显式映射过程。

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

以下示例与上一个示例等效,但是使用了 `procedureName` 别名:

Example 40. 通过 `procedureName` 别名在数据库中引用名称为 *"plus1inout"* 的隐式映射过程

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

下面再次与前两个等效,但是使用方法名称而不是显式注解属性。

Example 41. 使用方法名称在 `EntityManager` 中引用隐式映射的命名存储过程 *"User.plus1"*.

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

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

Example 42. 在 *EntityManager* 中引用显式映射的命名存储过程 *"User.plus1IO"*.

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

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

Chapter 5. Specification

JPA 2 引入了一个标准 API,您可以使用它来以编程方式构建查询. 通过编写条件,可以定义域类查询的 `where` 子句. 再往前一步,这些标准可以视为 JPA 标准 API 约束所描述的实体的断言.

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

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

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

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

`Specification` 接口定义如下:

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

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

Example 43. 自定义 Specifications

```
public class CustomerSpecs {


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

  public static Specification<Customer> hasSalesOfMoreThan(MonetaryAmount value) {
    return (root, query, builder) -> {
      // build query here
    };
  }
}
```

诚然,样板文件的数量尚待改进（最终可能会因 Java 8 闭包而减少）,但是客户端会变得更好,正如您将在本节后面看到的那样。`Customer_` 类型是使用 JPA 元模型生成器生成的元模型类型（有关示例,参见Hibernate实现的文档）. 因此,表达式 `Customer_.createdAt` 假定客户具有类型为 `Date` 的 `createdAt` 属性。除此之外,我们在业务需求抽象级别上表达了一些标准,并创建了可执行的 `Specifications`。因此,客户端可以使用以下 `Specifications`:

Example 44. 使用一个简单的 *Specification*

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

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

Example 45. 组合 *Specifications*

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

`Specification` 提供了一些 “glue-code” 默认方法来链接和组合 `Specification` 实例,这些方法使您可以通过创建新的 `Specification` 实现并将它们与现有的实现组合来扩展数据访问层。

Chapter 6. 查询例子

6.1. 简介

本章对 "按示例查询" 进行了介绍,并说明了如何使用它.

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

6.2. 使用

示例查询由三部分组成:

- Probe: 带有填充字段的 domain 对象的实际示例.
- `ExampleMatcher`: `ExampleMatcher` 包含有关如何匹配特定字段的详细信息. 可以在多个示例中重复使用它.
- `Example`: 示例包括探针和 `ExampleMatcher`. 它用于创建查询.

示例查询非常适合几种用例:

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

示例查询也有一些限制:

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

在开始使用示例查询之前,您需要具有一个 domain 对象. 首先,为您的存储库创建一个接口,如以下示例所示:

Example 46. Sample Person 对象

```
public class Person {

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

  // … getters and setters omitted
}
```

前面的示例显示了一个简单的域对象. 您可以使用它来创建一个 `Example`. 默认情况下,具有 `null` 的字段将被忽略,并且使用存储特定的默认值来匹配字符串. 可以使用工厂方法或使用 `ExampleMatcher` 构建示例. 例子是一成不变的. 以下清单显示了一个简单的示例:

Example 47. Simple Example

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

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

① 创建 domain 对象的新实例.

② 设置要查询的属性.

③ 创建 `Example`.

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

Example 48. `QueryByExampleExecutor`

```
public interface QueryByExampleExecutor<T> {

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

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

  // … more functionality omitted.
}
```

6.3. Example 匹配

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

Example 49. 具有定制匹配的例子匹配器

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

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

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

① 创建域对象的新实例.

② 设置属性.

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

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

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

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

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

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

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

Example 50. 配置匹配器选项

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

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

Example 51. 用*lambdas*配置匹配器选项

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

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

`ExampleMatcher` 上设置的设置将由属性路径设置继承。属性修补程序上的设置优先于默认设置。下表描述了各种 `ExampleMatcher` 设置的范围:

表4. `ExampleMatcher` 设置的范围

Table 3. Scope of `ExampleMatcher` *settings*

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

6.4. 执行一个例子

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

Example 52. 使用存储库按示例查询

```
public interface PersonRepository extends JpaRepository<Person, String> { … }

public class PersonService {

  @Autowired PersonRepository personRepository;

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

当前,仅 `SingularAttribute` 属性可用于属性匹配。

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

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

Table 4. `StringMatcher` *options*

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

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

Chapter 7. 事务性

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

Example 53. CRUD 的自定义事务配置

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

  @Override
  @Transactional(timeout = 10)
  public List<User> findAll();

  // Further query method declarations
}
```

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

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

Example 54. 使用外观定义多个存储库调用的事务

```
@Service
class UserManagementImpl implements UserManagement {

  private final UserRepository userRepository;
  private final RoleRepository roleRepository;

  @Autowired
  public UserManagementImpl(UserRepository userRepository,
    RoleRepository roleRepository) {
    this.userRepository = userRepository;
    this.roleRepository = roleRepository;
  }

  @Transactional
  public void addRoleToAllUsers(String roleName) {

    Role role = roleRepository.findByName(roleName);

    for (User user : userRepository.findAll()) {
      user.addRole(role);
      userRepository.save(user);
    }
}
```

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

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

7.1. 事务查询方法

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

Example 55. 在查询方法上使用 `@Transactional`

```
@Transactional(readOnly = true)
public interface UserRepository extends JpaRepository<User, Long> {

  List<User> findByLastname(String lastname);

  @Modifying
  @Transactional
  @Query("delete from User u where u.active = false")
  void deleteInactiveUsers();
}
```

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

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

Chapter 8. 锁

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

Example 56. 在查询方法上定义锁元数据

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

  // Plain query method
  @Lock(LockModeType.READ)
  List<User> findByLastname(String lastname);
}
```

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

Example 57. 在 *CRUD* 方法上定义锁元数据

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

  // Redeclaration of a CRUD method
  @Lock(LockModeType.READ)
  List<User> findAll();
}
```

Chapter 9. 审计

9.1. 基础

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

9.1.1. 基于注解的审核元数据

我们提供 `@CreatedBy` 和 `@LastModifiedBy` 来捕获创建或修改实体的用户,并提供 `@CreatedDate` 和 `@LastModifiedDate` 来捕获更改发生的时间。

Example 58. 被审计实体

```
class Customer {

  @CreatedBy
  private User user;

  @CreatedDate
  private DateTime createdDate;

  // … further properties omitted
}
```

如您所见,可以根据要捕获的信息有选择地应用注解。捕获更改时捕获的注解可用于类型 `Joda-Time`,`DateTime`,旧版Java `Date` 和 `Calendar`,JDK8日期和时间类型以及 `long` 或 `Long` 的属性。

9.1.2. 基于接口的审核元数据

如果您不想使用注解来定义审核元数据,则可以让您的 domain 类实现 `Auditable` 接口。它为所有审核属性暴露了setter方法。

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

9.1.3. `AuditorAware`

如果使用 `@CreatedBy` 或 `@LastModifiedBy`,则审计基础结构需要以某种方式了解当前的主体。为此,我们提供了 `AuditorAware<T>` SPI接口,您必须实现该接口以告知基础结构与应用程序交互的当前用户或系统是谁。通用类型T定义必须使用 `@CreatedBy` 或 `@LastModifiedBy` 注解的属性的类型。

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

Example 59. 基于Spring Security的AuditorAware的实现

```
class SpringSecurityAuditorAware implements AuditorAware<User> {

  public Optional<User> getCurrentAuditor() {

    return Optional.ofNullable(SecurityContextHolder.getContext())
              .map(SecurityContext::getAuthentication)
              .filter(Authentication::isAuthenticated)
              .map(Authentication::getPrincipal)
              .map(User.class::cast);
  }
}
```

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

9.2. JPA 审计

9.2.1. 通用审核配置

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

Example 60. Auditing configuration orm.xml

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

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

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

}
```

审计功能要求 `spring-aspects.jar` 位于类路径中。

通过适当地修改 `orm.xml` 并在类路径上使用 `spring-aspects.jar`，激活审核功能只需将 Spring Data JPA `auditing` 命名空间元素添加到您的配置中，如下所示：

Example 61. 使用 XML 配置激活审计

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

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

Example 62. 用 Java 配置激活审计

```
@Configuration
@EnableJpaAuditing
class Config {

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

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

Chapter 10. 其他注意事项

10.1. 在自定义实现中使用 `JpaContext`

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

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

Example 63. 在自定义存储库实现中使用 `JpaContext`

```
class UserRepositoryImpl implements UserRepositoryCustom {

  private final EntityManager em;

  @Autowired
  public UserRepositoryImpl(JpaContext context) {
    this.em = context.getEntityManagerByManagedType(User.class);
  }

  …
}
```

这种方法的优点是,如果将 domain 类型分配给其他持久性单元,则无需触摸存储库即可更改对持久性单元的引用.

10.2. 合并持久性单元

Spring 支持具有多个持久性单元. 但是,有时您可能希望对应用程序进行模块化,但仍要确保所有这些模块都在单个持久性单元中运行. 为了实现这种行为,Spring Data JPA 提供了一个 `PersistenceUnitManager` 实现,该实现会根据其名称自动合并持久性单元,如以下示例所示:

Example 64. 使用 *MergingPersistenceUnitmanager*

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

10.2.1. @Entity 类和 JPA 映射文件的类路径扫描

普通的JPA设置要求所有注解映射的实体类在 `orm.xml` 中列出. XML映射文件也是如此. Spring Data JPA 提供了一个 `ClasspathScanningPersistenceUnitPostProcessor`,它配置了一个基本包,并可以选择采用映射文件名模式. 然后,它在给定的软件包中扫描以 `@Entity` 或 `@MappedSuperclass` 注解的类,加载与文件名模式匹配的配置文件,并将其交给JPA配置. 后处理器必须配置如下:

Example 65. 使用 ClasspathScanningPersistenceUnitPostProcessor

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

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

10.3. CDI 集成

存储库接口的实例通常由容器创建,在使用 Spring Data 时,Spring 是最自然的选择。如创建存储库实例中所述,Spring 为创建 bean 实例提供了复杂的支持。从 1.1.0 版本开始,Spring Data JPA 附带了一个自定义 CDI 扩展名,该扩展名允许在 CDI 环境中使用存储库抽象。该扩展是 JAR 的一部分。要激活它,请将 Spring Data JPA JAR 包含在类路径中。

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

```
class EntityManagerFactoryProducer {

  @Produces
  @ApplicationScoped
  public EntityManagerFactory createEntityManagerFactory() {
    return Persistence.createEntityManagerFactory("my-persistence-unit");
  }

  public void close(@Disposes EntityManagerFactory entityManagerFactory) {
    entityManagerFactory.close();
  }

  @Produces
  @RequestScoped
  public EntityManager createEntityManager(EntityManagerFactory entityManagerFactory)
{
    return entityManagerFactory.createEntityManager();
  }

  public void close(@Disposes EntityManager entityManager) {
    entityManager.close();
  }
}
```

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

```
class CdiConfig {

  @Produces
  @RequestScoped
  @PersistenceContext
  public EntityManager entityManager;
}
```

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

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

```
class RepositoryClient {

  @Inject
  PersonRepository repository;

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