周立、一号店CTO 韩军、张开涛、徐雷、林晓辉
Spring Cloud中国社区创始人 许进、DaoCloud首席架构师 王天青

联袂推荐

- 真正的从入门到精通，结合案例与工程实践，深入浅出，完整介绍Spring Data JPA
- 既是开发手册，又是实战指南，从整体到局部，深刻认识Spring Data JPA

# Spring Data JPA
## 从入门到精通

张振华 著

本书示例源代码

清華大學出版社

# Spring Data JPA
## 从入门到精通

张振华 著

清華大學出版社
北京

图书在版编目（CIP）数据

Spring Data JPA从入门到精通 / 张振华著. —北京：清华大学出版社，2018

ISBN 978-7-302-49948-0

Ⅰ. ①S… Ⅱ. ①张… Ⅲ. ①JAVA语言－程序设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字（2018）第066125号

责任编辑：夏毓彦

封面设计：王 翔

责任校对：闫秀华

责任印制：

出版发行：清华大学出版社

网 址：http://www.tup.com.cn，http://www.wqbook.com

地 址：北京清华大学学研大厦A座

邮 编：100084

社 总 机：010-62770175

邮 购：010-62786544

投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn

质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn

印 刷 者：

装 订 者：

经　　销：全国新华书店

开　　本：190mm×260mm

印　　张：15.75

字　　数：403千字

版　　次：2018年5月第1版

印　　次：2018年5月第1次印刷

印　　数：1～3000

定　　价：59.00元

产品编号：078573-01

# 内容简介

本书以Spring Boot为技术基础，从入门到精通，由浅入深地介绍Spring Data JPA的使用。有语法，有实践，有原理剖析。

本书分为12章，内容包括整体认识JPA、JPA基础查询方法、定义查询方法、注解式查询方法、@Entity实例里面常用注解详解、JpaRepository扩展详解、JPA的MVC扩展REST支持、DataSource的配置、乐观锁、SpEL表达式在Spring Data里面的应用、Spring Data Redis实现cacheable的实践、IntelliJ IDEA加快开发效率、Spring Data REST简单介绍等。

本书适合Java开发初学者、Java开发工程师、Java开发架构师阅读，也适合高等院校和培训学校相关专业的师生教学参考。

# 推荐序

Spring Data是一个伟大的项目，它为数据访问提供了一致、相对简单的编程模型，并且可用来操作几乎所有的主流存储。

Spring Data JPA是Spring Data的核心子项目之一。本书由浅入深，讲解了Spring Data JPA的常用功能与API，并结合实际工作中的场景，讲解如何扩展、如何避免踩坑等。

本书值得读者拥有。

《Spring Cloud与Docker微服务架构实战》作者 周立

随着微服务的流行，Spring Boot与Spring Cloud被广泛使用。Spring Data JPA简化了数据库的操作，本书作者从最简单的开始到复杂应用，娓娓道来，填补相关领域空白。

一号店CTO 韩军/Jason

Spring发展到现在已经是Java应用开发必备的基础设施了，而且遵循它一贯的风格，孵化出一系列优秀的解决方案，如Spring Boot、Spring Data、Spring Cloud等，每一个解决方案都完全遵循Spring的设计理念。

Spring Data JPA在开发企业级应用时有其独特的优势，能帮助开发人员快速进行各种数据库到Java模型的映射，帮我们进行快速的业务逻辑开发，而无须关心数据映射的一些细节。

我也曾经使用Spring Data JPA开发过一个JavaEE项目开发脚手架ES项目，使用Spring Data JPA能快速地帮助我完成项目DAO层的开发。强烈推荐大家在开发企业级应用时使用Spring Data，本书能让读者从入门到灵活运用，值得一读。

《亿级流量网站架构核心技术》作者　张开涛

《Spring Data JPA从入门到精通》一书以Spring Boot为基础，抛砖引玉，案例实驱，讲解了Spring Data的各种实战用法，加上对Spring Data核心源码分析，能够使读者快速驾驭Spring Data，为IT企业架构变革和发展快速赋能，产生商业价值。

Spring Cloud中国社区创始人　许进（xujin.org）

作为一个Java老程序员，2000年开始接触Java，2003年开始用Struts + MySQL做Java Web开发。刚开始的时候直接用JDBC访问数据库，影响比较深刻的是当时有大量的时间花在写SQL和处理结果集上。那个时候数据库设计是程序设计很重要的部分，一般都是先做数据库设计（例如用Power Designer做ER模型）再写程序。数据库设计除了表的设计之外，还会涉及视图、触发器和存储过程等。到了2004年，Hibernate 1.0横空出世，当时身边有个大神同学（尹俊，目前就职于美国Google）花了一个月时间通读文档和源码，给大家做了讲解，大家讨论之后决定将Hibernate引入项目中，吃一下螃蟹。从此我就开始和ORM打起了交道。

ORM最大的好处就是让程序员关注在业务本身以及对应OO（面向对象）程序设计，这个更加契合领域设计和OO设计，而不是一开始陷入数据库细节层面，影响总体设计。学过领域设计的同学都知道里面有关于Entity、Repository、Service等相关的概念，而JPA则很好地实现了这些概念。Spring Data JPA出现之后，则更加简化了我们访问数据库的方式。你只要花费1分钟，定义一个实体类（加上Entity注解）和扩展一个CrudRepository的接口，就可以具备对单表CRUD操作的基本功能。

在2016年的一个实际项目中，我们在Spring Data JPA的基础上实现了很多功能，例如字段自动加解密、字段JSON与POJO自动映射、历史表（审计功能）、自动设置创建时间/更新时间、乐观锁/悲观锁等，收获颇多。

虽然经常使用Spring Data JPA，但是基本上都是遇到问题现查文档，缺少一本提纲挈领、循序渐进、完整讲解Spring Data JPA的书。振华老弟虽年纪不大，却很爱钻研技术，算得上是Spring Data JPA的专家，并且他写的这本书正好满足了我以及广大Java程序员的需求，学习Spring Data JPA不再枯燥，同时非常翔实、完整地讲解了Spring Data JPA，并配合大量实例，兼具参考书和实战指南的作用，值得广大读者仔细研读。

DaoCloud首席架构师　王天青

Spring Data进一步简化了Java访问SQL和NoSQL数据源的复杂度。本书详细介绍了Spring Data JPA框架的知识，是一本很好的学习参考书籍。

MongoDB官方团队《MongoDB实战》第2版译者　徐雷

本书从浅入深，从原理剖析到经验结合，直观地把Spring Data JPA和周边功能展现给了读者。相信从Java初学者到经验老道的架构师阅读本书都能有所收获。

资深Java老兵　林晓辉

# 前言

## 本书初衷

随着Java技术和微服务技术逐渐广泛应用，Spring Cloud、Spring Boot逐渐统一Java的框架江湖。市场上的ORM框架也逐渐被人重视起来。Spring Data逐渐走入Java开发者的视野，被很多架构师作为ORM框架的技术选型。市场上没有对Spring Data JPA的完整介绍。资料比较零散，很难一下子全面、深入地掌握Spring Data JPA。本书注重从实际出发来提高从事Java开发者的工作效率，可以作为一本很好的自我学习手册和Spring Data JPA的查阅手册。“不仅授之以鱼，还授之以渔”，不仅告诉大家是什么、怎么用，还告诉大家学习步骤、怎么学习，以及原理、使用技巧与实践。全书以Spring Boot为技术基础，从入门到精通，由浅入深地介绍和使用Spring Data JPA，很适合Java的初学者从此弯道超车，走上Spring全家桶学习的快车道。

## “未来已经来临，只是尚未流行”

纵观市场上的ORM框架，MyBitas以灵活著称，但是要维护复杂的配置，并且不是Spring官方的天然全家桶，还得做额外的配置工作，即使是资深的架构师也得做很多封装；Hibernate以HQL和关系映射著称，但是使用起来不是特别灵活。这样Spring Data JPA来了，感觉要夺取ORM的JPA霸主地位了，它底层以Hibernate为封装，对外提供了超级灵活的使用接口，又非常符合面向对象和REST的风格，越来越多的API层面的封装都是以Spring Data JPA为基础的，感觉是架构师和开发者的福音。Spring Data JPA与Spring Boot配合起来使用具有天然的优势，你会发现越来越多的公司招聘会由传统的SSH、

Spring、MyBitas技术要求逐步地变为Spring Boot、Spring Cloud、Spring Data等Spring 全家桶技术的要求。

## 追本溯源

架构师在架构设计系统之前都要先设计各种业务模型、数据模型，其实在众多技术框架中，要掌握Spring Boot、Spring MVC、Spring Cloud、微服务架构等，都离不开底层数据库操作层，如果我们能很好地掌握Data这层的技术要领，从下往上学习，这样可能会更好掌握一些。

## 本书特色

（1）本书针对Java开发者、Spring的使用者，是Spring Data JPA开发必备书籍。

（2）本书从介绍到使用再到原理和实践，可以作为一本很好的Spring Data JPA的实战手册。

（3）本书的代码清晰，迭代完整，便于全面、完整地掌握和学习JPA。

（4）本书注重从实战经验方面进行讲解，非常实用，一点即破。

（5）本书原型PPT深受同事喜爱，并在企业内部培训的时候得到了很多Java程序员的肯定。

## 阅读指南

本书以Spring Boot为开发基础和线索，大量采用了UML释义的讲解方式。本书分为3个部分，共12章。

（1）基础部分：整体认识JPA、JPA基础查询方法、定义查询方

法、注解式查询方法、@Entity实例里面常用注解详解，了解Spring Data JPA的基本使用和语法。

（2）晋级之高级部分：JpaRepository详解、JPA的MVC扩展Rest支持、DataSource的配置、乐观锁等，了解其背后的实现动机及其原理。

（3）延展部分：SpEL表达式在Spring Data里面的应用、Spring Data Redis实现cacheable的实践、IntelliJ IDEA加快开发效率、Spring Data Rest的介绍，直至整个Spring Data的生态。

另外，由于Spring Boot 2.0的版本Spring Data JPA有了一些变化，作者对Spring Boot 2.0中的JPA也做了一些总结，作为本书的配套阅读内容。可以通过扫描如下二维码查看：

## 技术支持

本书示例源代码下载地址（注意数字与字母大小写）如下：

https://github.com/zhangzhenhuajack/spring-data-jpa-guide

如果下载有问题，请联系电子邮箱booksaga@163.com，邮件主题为“Spring Data JPA从入门到精通”。

虽然本书是以Spring Boot为配置案例的教程，但是实际工作中，我们可能用XML甚至是混合的模式，还有可能是MyBatis的方式，所以实战不免会超出本书范畴，欢迎加群进行讨论，一起进步。交流QQ群号如下：

- QQ群一：240619787。
- QQ群二：559701472。

作者本人的微信二维码如下：

# 致谢

首先，感谢清华大学出版社各位编辑的辛勤劳动，得以让此书面世。其次，感谢家人对我的支持，特别是老婆大人在我写作过程中承担了大量的家务，比较辛苦。最后，特别感谢帮我写书评的行业技术大神们，也非常感谢日常工作中提供帮助的同事们以及技术社区的技术达人们，感谢大家提供的技术资料。

著　者

2018年3月

# 目 录

推荐序
前言
第一部分　基础部分
第1章　整体认识JPA
1.1　市场上ORM框架比对
1.2　JPA的介绍以及开源实现
1.3　了解Spring Data
1.3.1　Spring Data介绍
1.3.2　Spring Data的子项目
1.3.3　Spring Data操作的主要特性
1.4　Spring Data JPA的主要类及结构图
1.5　MySQL的快速开发实例
第2章　JPA基础查询方法
2.1　Spring Data Common的Repository
2.2　Repository的类层次关系　(diagms/hierarchy/structure)
2.3　CrudRepository方法详解
2.3.1　CrudRepository interface内容
2.3.2　CrudRepository interface的使用示例
2.4　PagingAndSortingRepository方法详解
2.4.1　PagingAndSortingRepository interface内容
2.4.2　PagingAndSortingRepository使用示例
2.5　JpaRepository方法详解
2.5.1　JpaRepository详解
2.5.2　JpaRepository的使用方法
2.6　Repository的实现类SimpleJpaRepository
第3章　定义查询方法
3.1　定义查询方法的配置方法
3.2　方法的查询策略设置
3.3　查询方法的创建

3.4 关键字列表
3.5 方法的查询策略的属性表达式
3.6 查询结果的处理
3.6.1 参数选择分页和排序（Pageable/Sort）
3.6.2 查询结果的不同形式（List/Stream/Page/Future）
3.6.3 Projections对查询结果的扩展
3.7 实现机制介绍
第4章 注解式查询方法
4.1 @Query详解
4.1.1 语法及源码
4.1.2 @Query用法
4.1.3 @Query排序
4.1.4 @Query分页
4.2 @Param用法
4.3 SpEL表达式的支持
4.4 @Modifying修改查询
4.5 @QueryHints
4.6 @Procedure储存过程的查询方法
4.7 @NamedQueries预定义查询
4.7.1 简介
4.7.2 用法举例
4.7.3 @NamedQuery、@Query和方法定义查询的对比
第5章 @Entity实例里面常用注解详解
5.1 javax.persistence概况介绍
5.2 基本注解
5.2.1 @Entity
5.2.2 @Table
5.2.3 @Id
5.2.4 @IdClass
5.2.5 @GeneratedValue
5.2.6 @Basic
5.2.7 @Transient

5.2.8 @Column
5.2.9 @Temporal
5.2.10 @Enumerated
5.2.11 @Lob
5.2.12 几个注释的配合使用
5.3 关联关系注解
5.3.1 @JoinColumn定义外键关联的字段名称
5.3.2 @OneToOne关联关系
5.3.3 @OneToManyg与@ManyToOne关联关系
5.3.4 @OrderBy关联查询时排序
5.3.5 @JoinTable关联关系表
5.3.6 @ManyToMany关联关系
5.4 Left join、Inner join与@EntityGraph
5.4.1 Left join与Inner join
5.4.2 @EntityGraph
5.5 关于关系查询的一些坑
第二部分 晋级之高级部分
第6章 JpaRepository扩展详解
6.1 JpaRepository介绍
6.2 QueryByExampleExecutor的使用
6.2.1 QueryByExampleExecutor详细配置
6.2.2 QueryByExampleExecutor的使用示例
6.2.3 QueryByExampleExecutor的特点及约束
6.2.4 ExampleMatcher详解
6.2.5 QueryByExampleExecutor使用场景&实际的使用
6.2.6 QueryByExampleExecutor的原理
6.3 JpaSpecificationExecutor的详细使用
6.3.1 JpaSpecificationExecutor的使用方法
6.3.2 Criteria概念的简单介绍
6.3.3 JpaSpecificationExecutor示例
6.3.4 Specification工作中的一些扩展
6.3.5 JpaSpecificationExecutor实现原理
6.4 自定义Repository

6.4.1 EntityManager介绍
6.4.2 自定义实现Repository
6.4.3 实际工作的应用场景
第7章 Spring Data JPA的扩展
7.1 Auditing及其事件详解
7.1.1 Auditing如何配置
7.1.2 @MappedSuperclass
7.1.3 Auditing原理解析
7.1.4 Listener事件的扩展
7.2 @Version处理乐观锁的问题
7.3 对MvcWeb的支持
7.3.1 @EnableSpringDataWebSupport
7.3.2 DomainClassConverter组件
7.3.3 HandlerMethodArgumentResolvers可分页和排序
7.3.4 @PageableDefault改变默认的page和size
7.3.5 Page原理解析
7.4 @EnableJpaRepositories详解
7.4.1 Spring Data JPA加载Repositories配置简介
7.4.2 @EnableJpaRepositories详解
7.4.3 JpaRepositoriesAutoConfiguration源码解析
7.5 默认日志简单介绍
7.6 Spring Boot JPA的版本问题
第8章 DataSource的配置
8.1 默认数据源的讲解
8.1.1 通过三种方法查看默认的DataSource
8.1.2 DataSource和JPA的配置属性
8.1.3 JpaBaseConfiguration
8.1.4 Configuration思路
8.2 AliDruidDataSource的配置
8.3 事务的处理及其讲解
8.3.1 默认@Transactional注解式事务
8.3.2 声明式事务

8.4　如何配置多数据源
8.4.1　在application.properties中定义两个DataSource
8.4.2　定义两个DataSourceConfigJava类
8.5　Naming命名策略详解及其实践
8.5.1　Naming命名策略详解
8.5.2　实际工作中的一些扩展
8.6　完整的传统XML的配置方法
第三部分　延展部分
第9章　IntelliJ IDEA与Spring JPA
9.1　IntelliJ IDEA概述
9.2　DataBase插件
9.3　Persistence及JPA相关的插件介绍
9.4　IntelliJ IDEA分析源码用到的视图
第10章　Spring Data Redis详解
10.1　Redis之Jedis的使用
10.2　Spring Boot+Spring Data Redis配置
10.2.1　第1步：分析一下源码
10.2.2　第2步：配置方法
10.2.3　第3步：调用的地方
10.2.4　第4步：总结
10.2.5　主要的几个类&简单用法介绍
10.3　Spring Data Redis结合Spring Cache　配置方法
10.3.1　Spring Cache介绍
10.3.2　Spring Boot快速开始Demo
10.3.3　Spring Boot Cache实现过程解析
10.3.4　Cache和Spring Data Redis结合快速开始
10.3.5　Spring Boot实现过程
第11章　SpEL表达式讲解
11.1　SpEL介绍
11.1.1　SpEL主要特点
11.1.2　使用方法
11.2　SpEL的基础语法

11.2.1 逻辑运算操作
11.2.2 逻辑关系比较
11.2.3 逻辑关系
11.2.4 三元表达式& Elvis运算符
11.2.5 正则表达式的支持
11.2.6 Bean的引用
11.2.7 List和Map的操作
11.3 主要的类及其原理
11.3.1 ExpressionParser
11.3.2 root object
11.3.3 EvaluationContext
11.3.4 类型转换
11.3.5 SpelParserConfiguration编译器配置
11.3.6 表达式模板设置
11.3.7 主要类关系图
11.3.8 SpEL支持的一些特性
11.4 Spring的主要使用场景
11.4.1 Spring Data JPA中SpEL支持
11.4.2 Spring Cachae
11.4.3 @Value
11.4.4 Web验证应用场景
11.4.5 总结
第12章 Spring Data REST
12.1 快速入门
12.1.1 Spring Data REST介绍
12.1.2 快速开始
12.1.3 Repository资源接口介绍
12.2 Spring Data REST定制化
12.2.1 @RepositoryRestResource改变***Repository对应的Path 路径和资源名字
12.2.2 @RestResource 改变SearchPath
12.2.3 改变返回结果
12.2.4 隐藏某些Repository、Repository的查询方法

或@Entity　关系字段
12.2.5　隐藏Repository的CRUD方法
12.2.6　自定义JSON输出
12.3　Spring Boot 2.0加载原理
12.4　未来发展
附录1　Repository Query Method关键字列表
附录2　Repository Query Method返回值类型
附录3　JPA注解大全
附录4　Spring中涉及的注解
附录5　application.properties里面关于JPA的配置大全

# 第一部分
# 基础部分

通过这一基础部分的几个章节，我们可以对JPA形成完整的认识，并掌握一些必须要掌握的概念和基础知识。

# 第1章
# 整体认识JPA

“修学好古，实事求是”

——《汉书·河间献王刘德传》

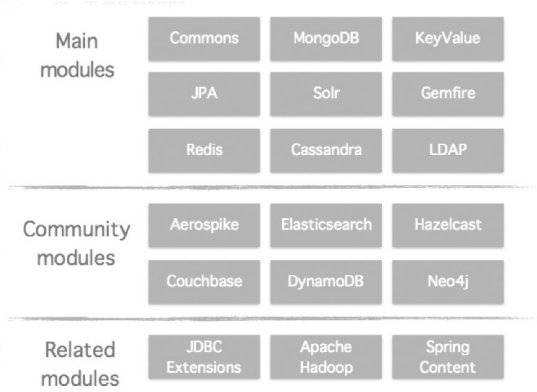



图1-1

从整体到局部，先来整体认识一下Spring Data JPA。

## 1.1 市场上ORM框架比对

### 1．MyBatis

MyBatis本是Apache的一个开源项目iBatis，2010年这个项目由Apache Software Foundation迁移到了Google Code，并且改名为MyBatis。MyBatis着力于POJO与SQL之间的映射关系，可以进行更为细致的SQL，使用起来十分灵活，上手简单，容易掌握，所以深受开发者的喜欢，目前市场占有率最高，比较适合互联应用公司的API场景。

### 2．Hibernate

Hibernate是一个开放源代码的对象关系映射框架，对JDBC进行了非常轻量级的对象封装，使得Java程序员可以随心所欲地使用对象编程思维来操纵数据库，并且对象有自己的生命周期，着力对象与对象之间的关系，有自己的HQL查询语言，所以数据库移植性很好。Hibernate是完备的ORM框架，是符合JPA规范的。Hibernate有自己的缓存机制。从上手的角度来说比较难，比较适合企业级的应用系统开发。

### 3．Spring Data JPA

可以理解为JPA规范的再次封装抽象，底层还是使用了Hibernate的JPA技术实现，引用JPQL（Java Persistence Query Language）查询语言，属于Spring整个生态体系的一部分。随着Spring Boot和Spring Cloud在市场上的流行，Spring Data JPA也逐渐进入大家的视野，它们组成有机的整体，使用起来比较方便，加快了开发的效率，使开发者不需要关心和配置更多的东西，完全可以沉浸在Spring的完整生态标准实现下。JPA上手简单，开发效率高，对对象的支持比较好，又有很大的灵活性，市场的认可度越来越高。

## 1.2 JPA的介绍以及开源实现

JPA是Java Persistence API的简称，中文名为Java持久层API，是JDK 5.0注解或XML描述对象－关系表的映射关系，并将运行期的实体对象持久化到数据库中。

Sun引入新的JPA ORM规范出于两个原因：其一，简化现有Java EE和Java SE应用开发工作；其二，Sun希望整合ORM技术，实现天下归一。

JPA包括以下3方面的内容：

（1）一套API标准。在javax.persistence的包下面，用来操作实体对象，执行CRUD操作，框架在后台替代我们完成所有的事情，开发者从烦琐的JDBC和SQL代码中解脱出来。

（2）面向对象的查询语言：Java Persistence Query Language（JPQL）。这是持久化操作中很重要的一个方面，通过面向对象而非面向数据库的查询语言查询数据，避免程序的SQL语句紧密耦合。

（3）ORM（object/relational metadata）元数据的映射。JPA支持XML和JDK5.0注解两种元数据的形式，元数据描述对象和表之间的映射关系，框架据此将实体对象持久化到数据库表中。

JPA的宗旨是为POJO提供持久化标准规范，由此可见，经过这几年的实践探索，能够脱离容器独立运行，方便开发和测试的理念已经深入人心了。Hibernate 3.2+、TopLink 10.1.3以及OpenJPA都提供了JPA的实现，以及最后的Spring的整合Spring Data JPA。目前互联网公司和传统公司大量使用了JPA的开发标准规范，如图1-2所示。

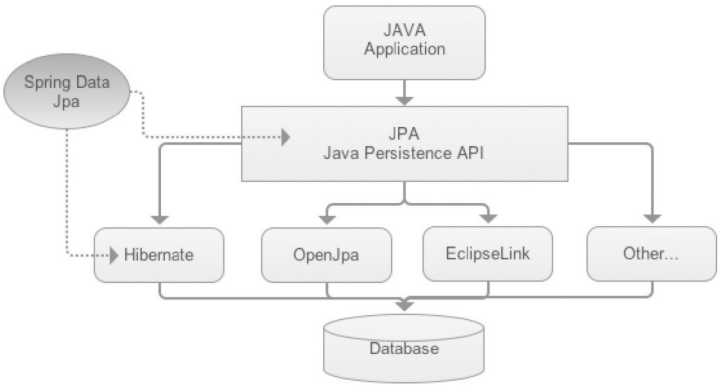



图1-2

# 1.3 了解Spring Data

## 1.3.1 Spring Data介绍

Spring Data项目是从2010年发展起来的，从创立之初Spring

Data就想提供一个大家熟悉的、一致的、基于Spring的数据访问编程模型，同时仍然保留底层数据存储的特殊特性。它可以轻松地让开发者使用数据访问技术，包括关系数据库、非关系数据库（NoSQL）和基于云的数据服务。

Spring Data Common是Spring Data所有模块的公用部分，该项目提供跨Spring数据项目的共享基础设施。它包含了技术中立的库接口以及一个坚持java类的元数据模型。

Spring Data不仅对传统的数据库访问技术JDBC、Hibernate、JDO、TopLick、JPA、Mybitas做了很好的支持、扩展、抽象、提供方便的API，还对NoSQL等非关系数据做了很好的支持，包括MongoDB、Redis、Apache Solr等。

### 1.3.2 Spring Data的子项目

主要子项目（Main modules）如下：

- Spring Data Commons
- Spring Data Gemfire
- Spring Data JPA
- Spring Data KeyValue
- Spring Data LDAP
- Spring Data MongoDB
- Spring Data REST
- Spring Data Redis
- Spring Data for Apache Cassandra
- Spring Data for Apache Solr

社区支持的子项目（Community modules）：

- Spring Data Aerospike

- Spring Data Couchbase
- Spring Data DynamoDB
- Spring Data Elasticsearch
- Spring Data Hazelcast
- Spring Data Jest
- Spring Data Neo4j
- Spring Data Vault

其他子项目（Related modules）：

- Spring Data JDBC Extensions
- Spring for Apache Hadoop
- Spring Content

当然，还有许多开源社区做出的贡献，比如Mybitas等。

市面上主要的子项目如图1-3所示。

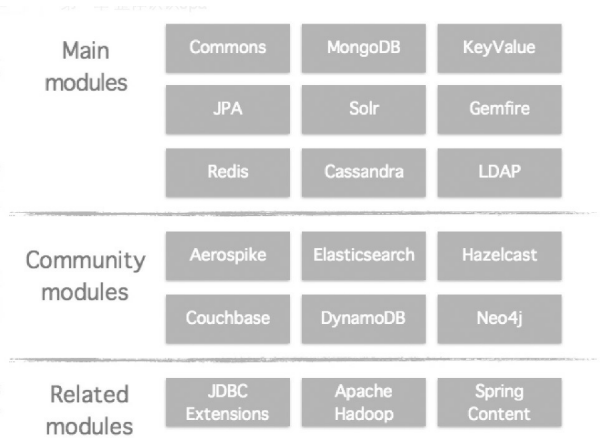



图1-3

## 1.3.3 Spring Data操作的主要特性

Spring Data项目旨在为大家提供一种通用的编码模式。数据访问对象实现了对物理数据层的抽象，为编写查询方法提供了方便。通

过对象映射，实现域对象和持续化存储之间的转换，而模板提供的是对底层存储实体的访问实现，如图1-4所示。操作上主要有如下特征：

- 提供模板操作，如Spring Data Redis和Spring Data Riak。
- 强大的Repository和定制的数据存储对象的抽象映射。
- 对数据访问对象的支持（Auting等）。

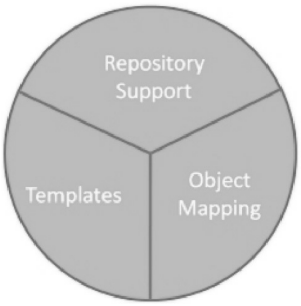



图1-4

## 1.4 Spring Data JPA的主要类及结构图

（1）我们需要掌握和使用到的类。

七个Repository接口：

- Repository (org.springframework.data.repository)
- CrudRepository (org.springframework.data.repository)
- PagingAndSortingRepository
  (org.springframework.data.repository)
- QueryByExampleExecutor
  (org.springframework.data.repository.query)
- JpaRepository
  (org.springframework.data.jpa.repository)
- JpaSpecificationExecutor
  (org.springframework.data.jpa.repository)

- QueryDslPredicateExecutor
  (org.springframework.data.querydsl)

两个实现类：

- SimpleJpaRepository
  (org.springframework.data.jpa.repository.support)
- QueryDslJpaRepository
  (org.springframework.data.jpa.repository.support)

（2）关系结构图如图1-5所示。

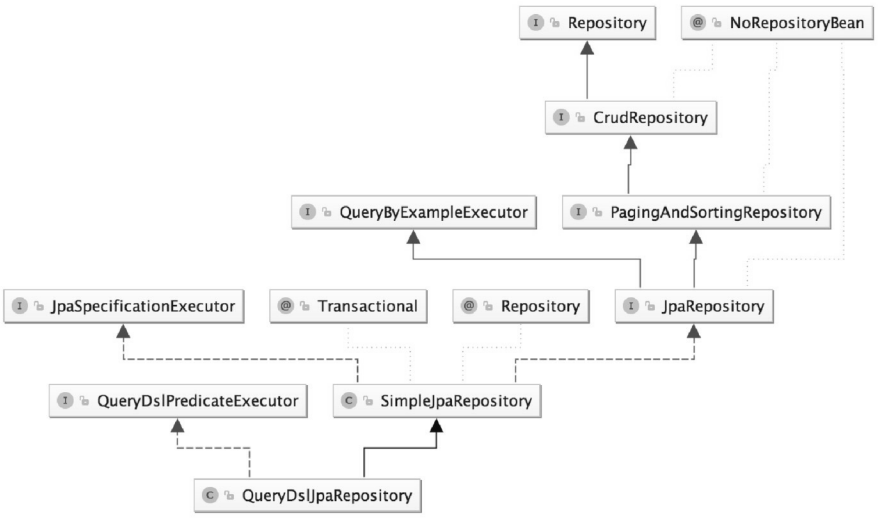



图1-5

基本上都是我们要关心的类和接口，先做到心中大体有个数，后面章节我们会一一做讲解。

（3）需要了解的类，真正的JPA的底层封装类。

- EntityManager (javax.persistence)
- EntityManagerImpl (org.hibernate.jpa.internal)

## 1.5 MySQL的快速开发实例

以Spring Boot和Spring Jdbc为技术场景，选用MySQL来做一个实例。

（1）环境要求：

- JDK 1.8
- Maven 3.0+
- IntelliJ IDEA

（2）第一步：创建数据库并建立user表。

① 创建一个数据的新用户并附上权限：

```
mysql> create database db_example;
mysql> create user 'springuser'@'localhost' identified by 'ThePassword';
mysql> grant all on db_example.* to 'springuser'@'localhost';
```

② 创建一个表：

```
CREATE TABLE `user` (
 `id` int(11) NOT NULL AUTO_INCREMENT,
 `name` varchar(50) DEFAULT NULL,
  `email` varchar(200) DEFAULT NULL,
  PRIMARY KEY ('id')
)
```

（3） 第二步：利用IntelliJ IDEA创建Example1，如图1-6、图1-7所示。

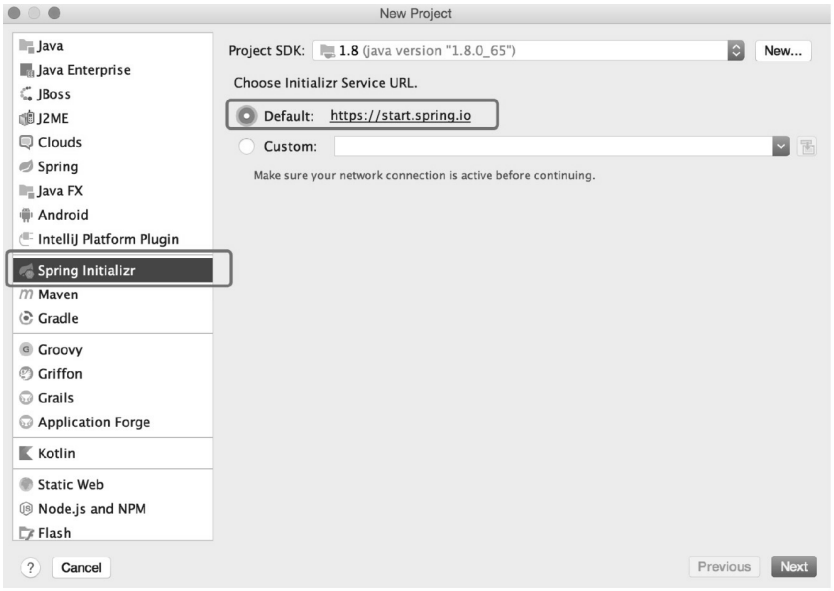



图1-6

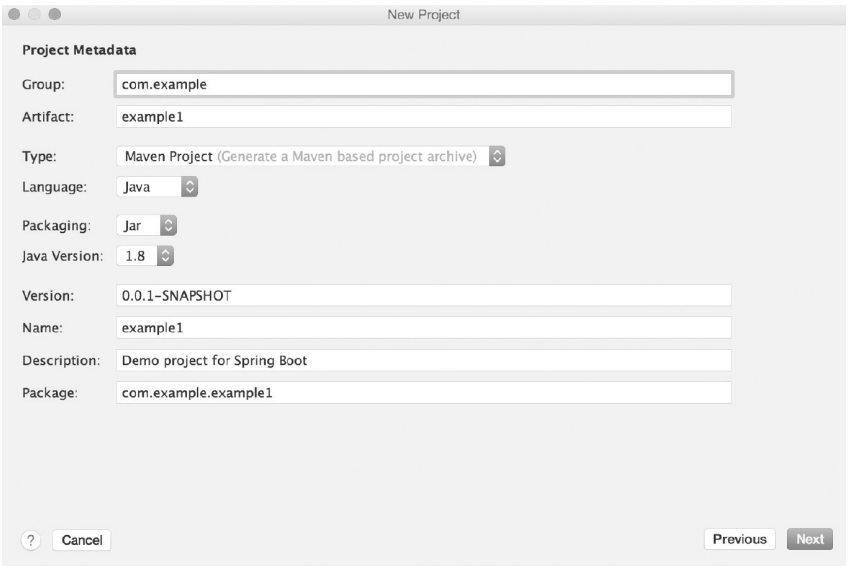



图1-7

上面的信息是Maven的pom里面所需要的，都可以修改，如图1-8所示。

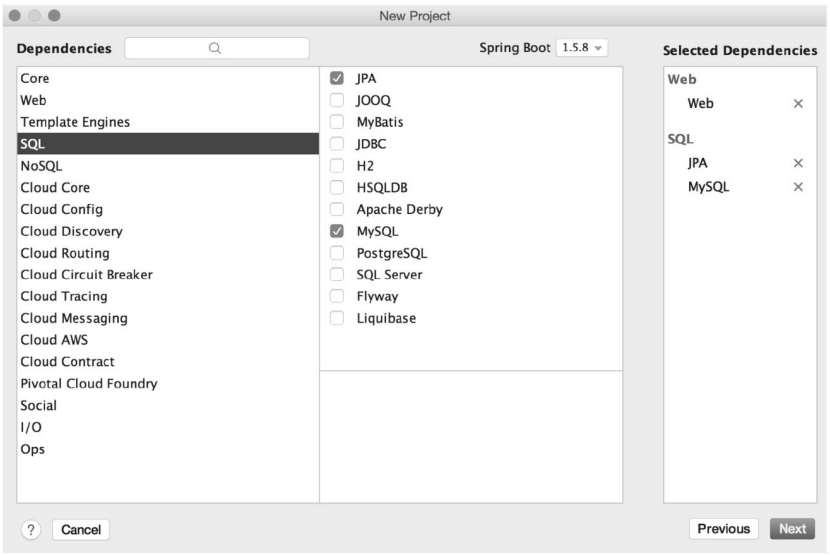



图1-8

选择JPA、MySQL和Web，一路Next然后完成得到一个工程。完成后结构如图1-9所示。


图1-9

（4）第三步：创建或者修改application.properties文件。在工程的sources下面，如src/main/resources/application.properties。内容如下：

```
spring.datasource.url=jdbc:mysql://localhost:3306/db_example
spring.datasource.username=springuser
spring.datasource.password=ThePassword
```

（5）第四步：创建一个@Entity。文件为src/main/java/example/example1/User.java。

```
package com.example.example1;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
@Entity
public class User {
   @Id
   @GeneratedValue(strategy=GenerationType.AUTO)
   private Long id;
   private String name;
   private String email;
   public Long getId() {
      return id;
   }
   public void setId(Long id) {
      this.id = id;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public String getEmail() {
      return email;
   }
   public void setEmail(String email) {
      this.email = email;
   }
}
```

（6）第五步：创建一个Repository。文件为src/main/java/example/example1/UserRepository.java。

```
package com.example.example1;
import org.springframework.data.repository.CrudRepository;
public interface UserRepository extends CrudRepository<User, Long> {
}
```

（7）第六步：创建一个Controller。

```
package com.example.example1;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;
@Controller
@RequestMapping(path = "/demo")
public class UserController {
   @Autowired
   private UserRepository userRepository;
   @GetMapping(path = "/add")
   public void addNewUser(@RequestParam String name, @RequestParam String email)
{
      User n = new User();
      n.setName(name);
      n.setEmail(email);
      userRepository.save(n);
   }
   @GetMapping(path = "/all")
   @ResponseBody
   public Iterable<User> getAllUsers() {
      return userRepository.findAll();
   }
}
```

（8）第七步：直接运行Example1Application的main函数。打开Example1Application，内容如下：

```
package com.example.example1;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class Example1Application {
   public static void main(String[] args) {
      SpringApplication.run(Example1Application.class, args);
   }
}
$ curl
'localhost:8080/demo/add?name=First&email=someemail@someemailprovider.com'
$ curl 'localhost:8080/demo/all'
```

这时已经可以看到效果了。

# 第2章
# JPA基础查询方法

学问之功，贵乎循序渐进，经久不息。

——梁启超

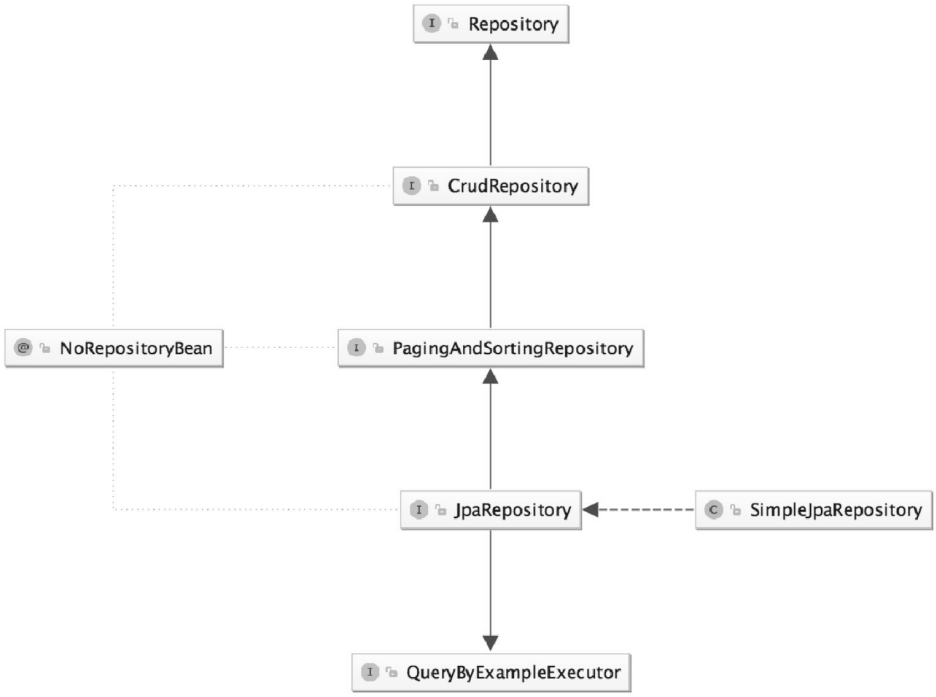



图2-1

本章学习Spring Data Common里面的公用基本方法。

## 2.1 Spring Data Common的Repository

Repository位于Spring Data Common的lib里面，是Spring Data里面做数据库操作的最底层的抽象接口、最顶级的父类，源码里面其实什么方法都没有，仅仅起到一个标识作用。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

帮助你发现扩展此接口的接口。Spring底层做动态代理的时候发现只要是它的子类或者实现类，都代表储存库操作。

Repository的源码如下：

```
package org.springframework.data.repository;
import java.io.Serializable;
public interface Repository<T, ID extends Serializable> {
}
```

有了这个类，我们就能顺藤摸瓜，找到好多Spring Data JPA提供的基本接口和操作类，及其实现方法。这个接口定义了所有Repostory操作的实体和ID两个泛型参数。我们不需要继承任何接口，只要继承这个接口，就可以使用Spring JPA里面提供的很多约定的方法查询和注解查询，后面章节会详细介绍。

## 2.2 Repository的类层次关系 (diagms/hierarchy/structure)

我们来根据Repository这个基类顺藤摸瓜，看看Spring Data JPA里面都有些什么，顺便教大家学习的方法，这样不管碰到什么框架，学习方法都类似，自己可以逐步从入门到精通，提高学习效率。

（1）我们用工具IntelliJ IDEA，打开类Repository.class，单击Navigate→Type Hierarchy。然后我们会得到如图2-2所示的视图。

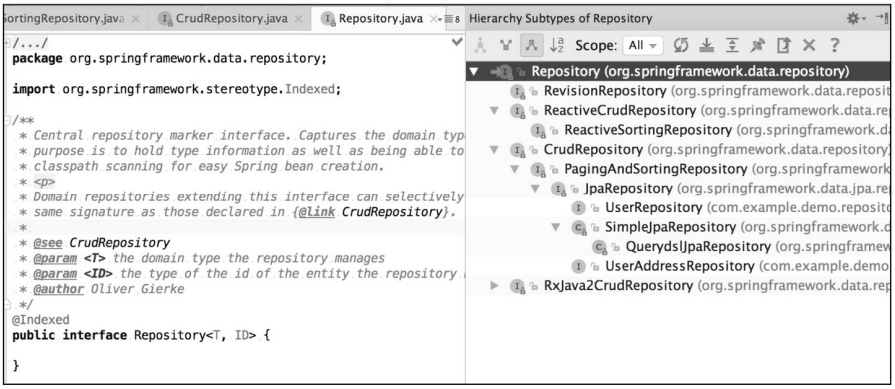


图2-2

通过这个层次结构视图，我们就会明白基类Repository的实现，对工程里面的所有Repository了如指掌，我们项目里面有哪些、Spring的项目里面有哪些也会一目了然。

（2）通过IntelliJ IDEA打开类Example1里面的UserRepository.java，右击选择show diagrams，用图表的方式查看类的层次关系，如图2-3所示。

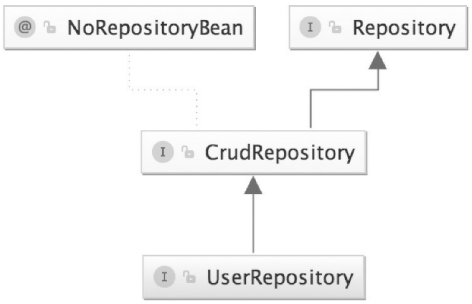



图2-3

（3）通过IntelliJ IDEA打开类QueryDslJpaRepository，右击，选择show diagrams，用图表的方式查看类的关系层次。打开的界面如图2-4所示。

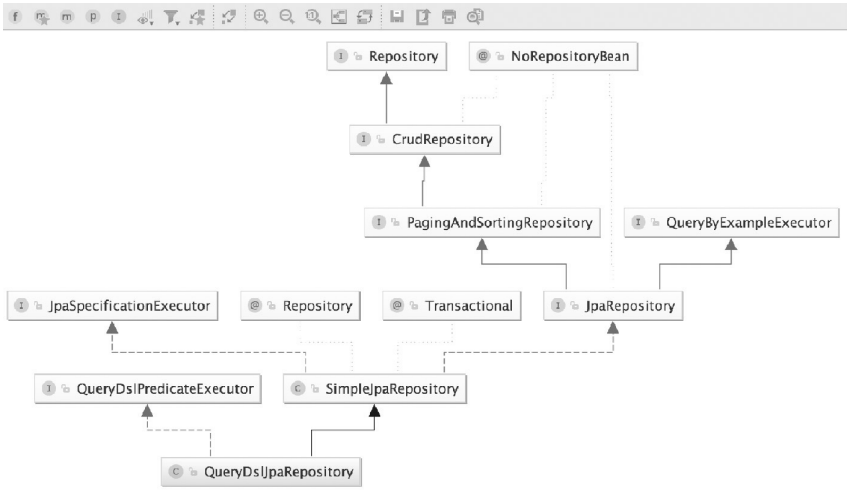



图2-4

（4）通过IntelliJ IDEA打开类Example1里面的

UserRepository.java，打开Navigate→File Structure，可以查看此类的结构以及有哪些方法。以此类推到其他类上。打开的界面如图2-5所示。

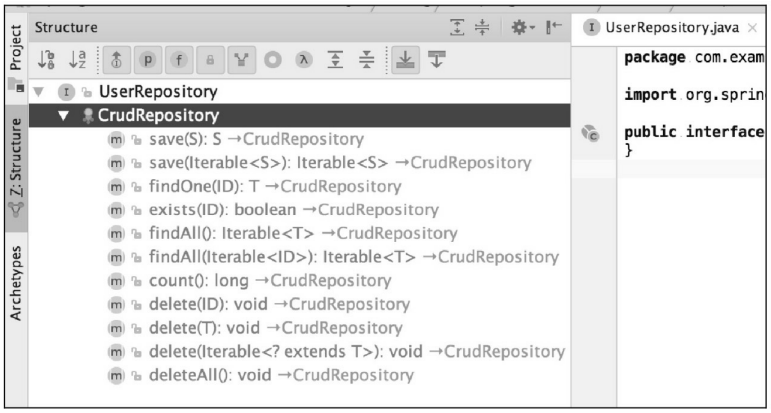



图2-5

以上三种视图是开发过程中经常用到的视图。

我们来看一个Repository的实例：

```
package com.example.example2;
import org.springframework.data.domain.Page;
import org.springframework.data.repository.Repository;
import java.util.List;
public interface UserRepository extends Repository<User, Long> {
   /**
    * 根据名称查询用户列表
    * @param name
    * @return
    */
   List<User> findByName(String name);
   /**
    * 根据用户的邮箱和名称查询
    *
    * @param email
    * @param name
    * @return
    */
   List<User> findByEmailAndName(String email,String name);
}
```

## 2.3 CrudRepository方法详解

通过类关系图可以看到CrudRepository提供了公共的通用的CRUD方法。

## 2.3.1 CrudRepository interface内容

```
package org.springframework.data.repository;
import java.io.Serializable;
@NoRepositoryBean
public interface CrudRepository<T, ID extends Serializable> extends Repository<T,
ID> {
    <S extends T> S save(S entity); (1)
    <S extends T> Iterable<S> save(Iterable<S> entities); (2)
    T findOne(ID id); (3)
    boolean exists(ID id); (4)
    Iterable<T> findAll(); (5)
    Iterable<T> findAll(Iterable<ID> ids); (6)
    long count(); (7)
    void delete(ID id); (8)
    void delete(T entity); (9)
    void delete(Iterable<? extends T> entities); (10)
    void deleteAll(); (11)
}
```

（1）保存实体方法。我们通过刚才的类关系查看其他实现类。

SimpleJpaRepository里面的实现方法：

```
public <S extends T> S save(S entity) {
  if (entityInformation.isNew(entity)) {
     em.persist(entity);
     return entity;
   } else {
     return em.merge(entity);
   }
}
```

我们发现它是先检查传进去的实体是不是存在，然后判断是新增还是更新；是不是存在两种根据机制，一种是根据主键来判断，另一种是根据Version来判断（后面讲解Version的时候详解）。如果我们去看JPA控制台打印出来的SQL，最少会有两条，一条是查询，一条是insert或者update。

（2）批量保存。原理和步骤（1）相同。实现方法就是for循环调用上面的save方法。

（3）根据主键查询实体。

（4）根据主键判断实体是否存在。

（5）查询实体的所有列表。

（6）根据主键列表查询实体列表。

（7）查询总数。

（8）根据主键删除。我们通过刚才的类关系查看其他实现类。

SimpleJpaRepository里面的实现方法：

```
   @Transactional
   public void delete(ID id) {
     Assert.notNull(id, ID_MUST_NOT_BE_NULL);
     T entity = findOne(id);
     if (entity == null) {
         throw new EmptyResultDataAccessException(String.format("No %s entity with
id %s exists!", entityInformation.getJavaType(), id), 1);
      }
     delete(entity);
   }
```

我们看到JPA会先去查询一下，再做保存，不存在抛出异常。

这里特别强调一下delete和save方法，因为在实际工作中有的人会画蛇添足，自己先去查询再做判断处理，其实Spring JPA底层都已经考虑到了。

## 2.3.2 CrudRepository interface的使用示例

使用也很简单，只需要自己的Repository继承CrudRepository即可。

第1章的示例我们修改如下：UserCrudRepository继承CrudRepository。

```
import com.example.example2.User;
import org.springframework.data.repository.CrudRepository;
public interface UserCrudRepository extends CrudRepository<User,Long> {
}
```

第1章的示例UserController修改如下：

```
@Controller
@RequestMapping(path = "/demo")
public class UserController {
   @Autowired
   private UserCrudRepository userRepository;
   @GetMapping(path = "/add")
   public void addNewUser(@RequestParam String name, @RequestParam String email)
{
      User n = new User();
      n.setName(name);
      n.setEmail(email);
      userRepository.save(n);
   }
   @GetMapping(path = "/all")
   @ResponseBody
   public Iterable<User> getAllUsers() {
      return userRepository.findAll();
   }
   @GetMapping(path = "/info")
   @ResponseBody
   public User findOne(@RequestParam Long id){
      return userRepository.findOne(id);
   }
   @GetMapping(path = "/delete")
   public void delete(@RequestParam Long id){
      userRepository.delete(id);
   }
}
```

然后启动运行就可以直接看效果了。

## 2.4 PagingAndSortingRepository方法详解

通过类的关系图，我们可以看到PagingAndSortingRepository继承CrudRepository所有的基本方法，它增加了分页和排序等对查询结果进行限制的基本的、常用的、通用的一些分页方法。

## 2.4.1 PagingAndSortingRepository interface 内容

```
package org.springframework.data.repository;
import java.io.Serializable;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;
@NoRepositoryBean
public interface PagingAndSortingRepository<T, ID extends Serializable> extends
CrudRepository<T, ID> {
    Iterable<T> findAll(Sort sort);(1)
    Page<T> findAll(Pageable pageable);(2)
}
```

（1）根据排序取所有对象的集合。

（2）根据分页和排序进行查询，并用Page对象封装。Pageable对象包含分页和Sort对象。

PagingAndSortingRepository和CrudRepository都是Spring Data Common的标准接口，如果我们采用JPA，那它对应的实现类就是Spring Data JPA的model里面的SimpleJpaRepository。如果是其他NoSQL的实现Mongodb，那它的实现就在Spring Data Mongodb的model里面。

实现内容如下：

```
public Page<T> findAll(Pageable pageable) {
   if (null == pageable) {
      return new PageImpl<T>(findAll());
   }
  return findAll((Specification<T>) null, pageable);
}
```

通过上面的源码我们可以发现这些查询都会用到后面章节要讲的Specification查询方法。

## 2.4.2 PagingAndSortingRepository使用示例

只需要继承PagingAndSortingRepository的接口即可，其他不用做任何改动。UserPagingAndSortingRepository修改如下：

```
    import com.example.example2.User;
    import org.springframework.data.repository.PagingAndSortingRepository;
    public interface UserPagingAndSortingRepository extends
PagingAndSortingRepository<User,Long> {
    }
```

UserController修改如下：

```
    /**
     * 验证排序和分页查询方法
     * @return
     */
    @GetMapping(path = "/page")
    @ResponseBody
    public Page<User> getAllUserByPage() {
       return userPagingAndSortingRepository.findAll(
             new PageRequest(1, 20,new Sort(new
Sort.Order(Sort.Direction.ASC,"name"))));
    }
    /**
     * 排序查询方法
     * @return
     */
    @GetMapping(path = "/sort")
    @ResponseBody
    public Iterable<User> getAllUsersWithSort() {
       return userPagingAndSortingRepository.findAll(new Sort(new
Sort.Order(Sort.Direction.ASC,"name")));
    }
```

# 2.5 JpaRepository方法详解

## 2.5.1 JpaRepository详解

JpaRepository到这里可以进入分水岭了，上面的那些都是Spring Data为了兼容NoSQL而进行的一些抽象封装，从JpaRepository开始是对关系型数据库进行抽象封装。从类图可以看得出来它继承了PagingAndSortingRepository类，也就继承了其所有

方法，并且实现类也是SimpleJpaRepository。从类图上还可以看出JpaRepository继承和拥有了QueryByExampleExecutor的相关方法。

```
package org.springframework.data.jpa.repository;
import java.io.Serializable;
import java.util.List;import javax.persistence.EntityManager;
import org.springframework.data.domain.Example;
import org.springframework.data.domain.Sort;
import org.springframework.data.repository.NoRepositoryBean;
import org.springframework.data.repository.PagingAndSortingRepository;
import org.springframework.data.repository.query.QueryByExampleExecutor;
@NoRepositoryBean
public interface JpaRepository<T, ID extends Serializable> extends
PagingAndSortingRepository<T, ID>,
    QueryByExampleExecutor<T> {
    List<T> findAll();
    List<T> findAll(Sort sort);
    List<T> findAll(Iterable<ID> ids);
    <S extends T> List<S> save(Iterable<S> entities);
    void flush();
    <S extends T> S saveAndFlush(S entity);
    void deleteInBatch(Iterable<T> entities);
    void deleteAllInBatch();
    T getOne(ID id);
    <S extends T> List<S> findAll(Example<S> example, Sort sort);}
```

通过源码和CrudRepository相比较，它支持Query By Example，批量删除，提高删除效率，手动刷新数据库的更改方法，并将默认实现的查询结果变成了List。

### 2.5.2 JpaRepository的使用方法

JpaRepository的使用方法也一样，只需要继承它即可，比如：

```
import com.example.example2.User;
import org.springframework.data.jpa.repository.JpaRepository;
public interface UserJpaRepository extends JpaRepository<User,Long> {
}
```

## 2.6 Repository的实现类SimpleJpaRepository

SimpleJpaRepository是JPA整个关联数据库的所有Repository的

接口实现类。如果想进行扩展，可以继承此类，如QueryDsl的扩展，还有默认的处理机制。如果将此类里面的实现方法看透了，基本上JPA的API就能掌握大部分。同时也是Spring JPA动态代理的实现类，包括我们后面讲的Query Method。

我们可以通过Debug视图看一下动态代理过程，如图2-6所示。

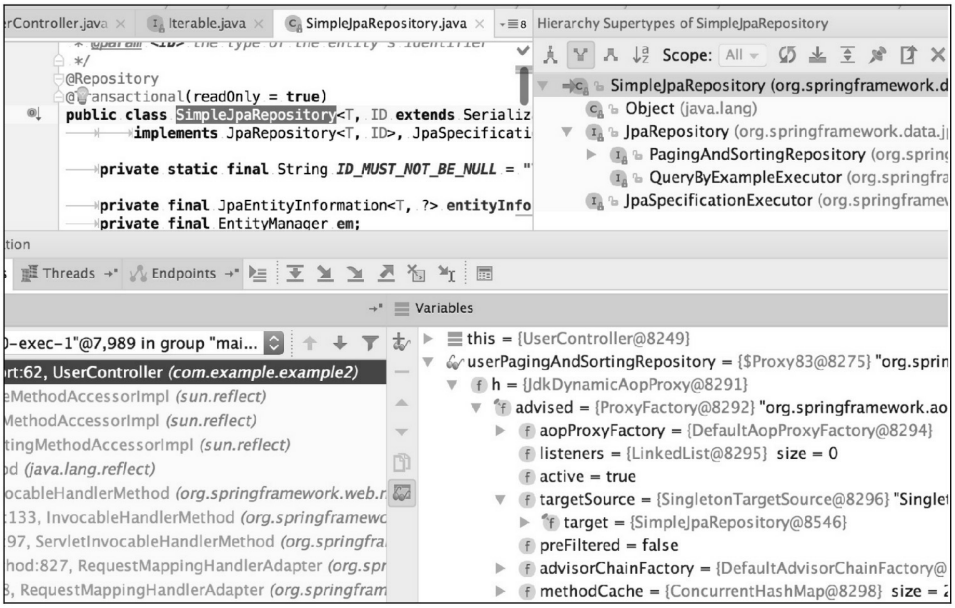



图2-6

SimpleJpaRepository的部分源码如下：

```
@Repository
@Transactional(readOnly = true)
public class SimpleJpaRepository<T, ID extends Serializable>
      implements JpaRepository<T, ID>, JpaSpecificationExecutor<T> {
   private final JpaEntityInformation<T, ?> entityInformation;
   private final EntityManager em;
   private final PersistenceProvider provider;
   private CrudMethodMetadata metadata;
......
   @Transactional
   public void deleteAllInBatch() {
       em.createQuery(getDeleteAllQueryString()).executeUpdate();
   }
......
}
```

可以看出SimpleJpaRepository的实现机制还挺清晰的，通过

EntityManger进行实体的操作，JpaEntityInforMation里面保存着实体的相关信息以及crud方法的元数据等，后面章节会经常提到此类，到时再慢慢讲解。

# 第3章
# 定义查询方法

“千里之行，始于足下。不积跬步，无以致千里。”

——荀子《劝学篇》

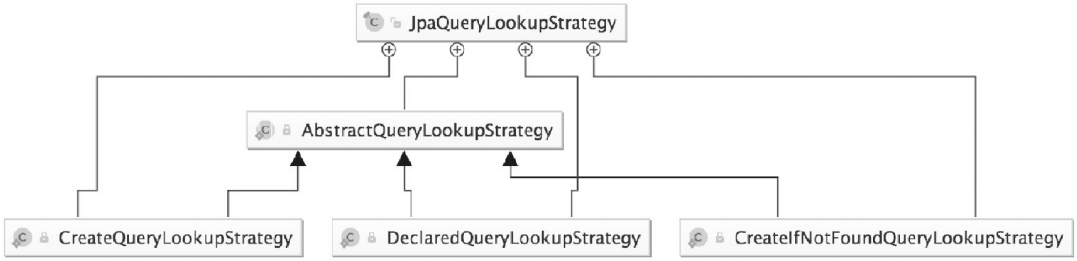



图3-1

本章详细讲解如何利用方法名定义查询方法（Defining Query Methods）。

## 3.1 定义查询方法的配置方法

由于Spring JPA Repository的实现原理是采用动态代理的机制，所以我们介绍两种定义查询方法：从方法名称中可以指定特定用于存储的查询和更新，或通过使用@Query手动定义的查询，这个取决于实际存储操作。只需要实体Repository继承Spring Data Common里面的Repository接口即可，就像前面我们讲的一样。如果你想有其他更多默认通用方法的实现，可以选择JpaRepository、PagingAndSortingRepository、CrudRepository等接口，也可以直接继承我们后面要讲的JpaSpecificationExecutor、QueryByExampleExecutor和自定义Response，都可以达到同样的效果。

如果你不想扩展Spring数据接口，还可以使用它来注解存储库接口@RepositoryDefinition。扩展CrudRepository公开了一套完整的方法来操纵实体。如果你希望对所暴露的方法有选择性，只需要将暴露的方法复制CrudRepository到域库中即可。其实也是自定义Repository的一种。

看下面的示例，选择性地暴露CRUD方法：

```
@NoRepositoryBeaninterface
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);
}
```

在这个示例的第一步中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(…)和save(…)方法将被路由到由Spring Data提供的、你选择的存储库的基本存储库实现中，例如JPA中的SimpleJpaRepository。因为它们正在匹配方法签名CrudRepository，所以UserRepository将能够保存用户，并通过id查找单个用户信息，以及触发查询以通过其电子邮件地址查找Users。

## 3.2 方法的查询策略设置

通过@EnableJpaRepositories(queryLookupStrategy=QueryLookupStrategy.Key.CREATE_IF_NOT_FOUND)可以配置方法的查询策略，其中QueryLookupStrategy.Key的值一共有三个。

- CREATE：直接根据方法名进行创建。规则是根据方法名称的构造进行尝试，一般的方法是从方法名中删除给定的一组已知前缀，并解析该方法的其余部分。如果方法名不符合规则，启动的时候就会报异常。
- USE_DECLARED_QUERY：声明方式创建，即本书说的注解方式。

启动的时候会尝试找到一个声明的查询，如果没有找到就将抛出一个异常。查询可以由某处注释或其他方法声明。

- CREATE_IF_NOT_FOUND：这个是默认的，以上两种方式的结合版。先用声明方式进行查找，如果没有找到与方法相匹配的查询，就用create的方法名创建规则创建一个查询。

除非有特殊需求，一般直接用默认的，不用管。配置示例如下：

```
   @EnableJpaRepositories(queryLookupStrategy=
QueryLookupStrategy.Key.CREATE_IF_NOT_FOUND)
   public class Example1Application {
      public static void main(String[] args) {
         SpringApplication.run(Example1Application.class, args);
      }
   }
```

QueryLookupStrategy是策略的定义接口，JpaQueryLookupStrategy是具体策略的实现类。

类图如图3-2所示。

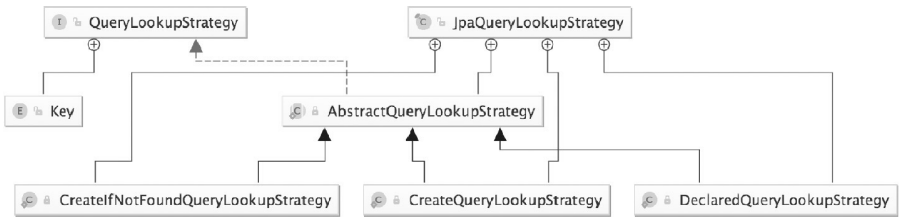



图3-2

## 3.3 查询方法的创建

内部基础架构中有个根据方法名的查询生成器机制，对于在存储库的实体上构建约束查询很有用。该机制方法的前缀有find…By、read…By、query…By、count…By和get…By，从这些方法可以分析它的其余部分（实体里面的字段）。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上，你可以

定义实体性条件，并与它们串联（And和Or）。

用一句话概括，待查询功能的方法名由查询策略（关键字）、查询字段和一些限制性条件组成。在如下例子中，可以直接在controller里面进行调用以查看效果：

```
    interface PersonRepository extends Repository<User, Long> {
       // and 的查询关系
       List<User> findByEmailAddressAndLastname(EmailAddress emailAddress, String
lastname);
       // 包含 distinct 去重、or 的 SQL 语法
       List<User> findDistinctPeopleByLastnameOrFirstname(String lastname, String
firstname);
       List<User> findPeopleDistinctByLastnameOrFirstname(String lastname, String
firstname);
       // 根据 lastname 字段查询忽略大小写
       List<User> findByLastnameIgnoreCase(String lastname);
       // 根据 lastname 和 firstname 查询 equal 并且忽略大小写
       List<User> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String
firstname);
      // 对查询结果根据 lastname 排序
       List<User> findByLastnameOrderByFirstnameAsc(String lastname);
       List<User> 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，有一些常见的事项需要注意：

- 表达式通常是可以连接的运算符的属性遍历。你可以使用组合属性表达式AND和OR。你还可以将运算关键字Between、LessThan、GreaterThan、Like作为属性表达式。受支持的操作员可能因数据存储而异，因此请参阅官方参考文档的相应部分内容。
- 该方法解析器支持设置一个IgnoreCase标志个别特性（例如，findByLastnameIgnoreCase(…)）或支持忽略大小写（通常是一个类型的所有属性为String的情况下，例如，findByLastnameAndFirstnameAllIgnoreCase(…)）。是否支持忽略示例可能会因存储而异，因此请参阅参考文档中的相关章节，了解特定于场景的查询方法。

- 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来影响查询结果。

## 3.4 关键字列表

关键字列表如表3-1所示。

表3-1 关键字列表

| 关键字 | 示例 | JPQL 表达 |
| --- | --- | --- |
| And | findByLastnameAndFirstname | … where x.lastname = ?1 and x.firstname = ?2 |
| Or | findByLastnameOrFirstname | … where x.lastname = ?1 or x.firstname = ?2 |
| Is、Equals | findByFirstname、findByFirstnameIs、findByFirstnameEquals | … where x.firstname = ?1 |
| Between | findByStartDateBetween | … where x.startDate between ?1 and ?2 |
| LessThan | findByAgeLessThan | … where x.age < ?1 |
| LessThanEqual | findByAgeLessThanEqual | … where x.age <= ?1 |
| GreaterThan | findByAgeGreaterThan | … where x.age > ?1 |
| GreaterThanEqual | findByAgeGreaterThanEqual | … where x.age >= ?1 |
| After | findByStartDateAfter | … where x.startDate > ?1 |

（续表）

| 关键字 | 示例 | JPQL 表达 |
| --- | --- | --- |
| Before | findByStartDateBefore | … where x.startDate < ?1 |
| IsNull | findByAgeIsNull | … where x.age is null |
| IsNotNull,<br>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<br>（参数增加前缀 %） |
| EndingWith | findByFirstnameEndingWith | … where x.firstname like ?1<br>（参数增加后缀 %） |
| Containing | findByFirstnameContaining | … where x.firstname like ?1<br>（参数被 % 包裹） |
| OrderBy | findByAgeOrderByLastnameDesc | … where x.age = ?1<br>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.firstame) = UPPER(?1) |

注意，除了find的前缀之外，我们查看PartTree的源码，还有如下几种前缀：

```
private static final String QUERY_PATTERN = "find|read|get|query|stream";
private static final String COUNT_PATTERN = "count";
private static final String EXISTS_PATTERN = "exists";
private static final String DELETE_PATTERN = "delete|remove";
```

使用的时候要配合不同的返回结果进行使用，例如：

```
interface UserRepository extends CrudRepository<User, Long> {
     long countByLastname(String lastname);//查询总数
     long deleteByLastname(String lastname);//根据一个字段进行删除操作
     List<User> removeByLastname(String lastname);
}
```

## 3.5 方法的查询策略的属性表达式

属性表达式（Property Expressions）只能引用托管（泛化）实体的直接属性，如前一个示例所示。在查询创建时，你已经确保解析的属性是托管实体的属性。同时，还可以通过遍历嵌套属性定义约束。假设一个Person实体对象里面有一个Address属性里面包含一个ZipCode属性。

在这种情况下，方法名为：

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

创建及其查找的过程是：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，就使用该属性。如果不是，就拆分右侧驼峰部分的信号源到头部和尾部，并试图找出相应的属性，在我们的例子中是AddressZip和Code。如果算法找到一个具有头部的属性，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，就将分割点移动到左边（Address、ZipCode），然后继续。

虽然这在大多数情况下应该起作用，但是算法可能会选择错误的属性。假设Person类也有一个addressZip属性，该算法将在第一个分割轮中匹配，并且基本上会选择错误的属性，最后失败（因为该类型addressZip可能没有code属性）。

要解决这个歧义，可以在方法名称中手动定义遍历点，所以我们的方法名称最终会是：

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

当然Spring JPA里面是将下划线视为保留字符，但是强烈建议遵循标准Java命名约定（不使用属性名称中的下划线，而是使用骆驼示例）。命名属性的时候注意一下这个特性。

可以到PartTreeJpaQuery.class查询一下相关的method的name拆分和实现逻辑，也可以利用开发工具的Search anywhere视图输入PropertyExpression，然后Find Used就可以跟出很多源码，再设置一个断点，就可以进行代码分析了。

## 3.6 查询结果的处理

### 3.6.1 参数选择分页和排序（Pageable/Sort）

#### 1．特定类型的参数，动态地将分页和排序应用于查询

示例：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);
```

第一种方法允许将org.springframework.data.domain.Pageable实例传递给查询方法，以便动态地将分页添加到静态定义的查询中。Page知道可用的元素和页面的总数。它通过基础框架里面触发计数查询来计算总数。由于这可能是昂贵的，具体取决于所使用的场景，说白了，当用到Pageable的时候会默认执行一条cout语句。而Slice的作用是，只知道是否有下一个Slice可用，不会执行count，所以当查询较大的结果集时，只知道数据是足够的就可以了，而且相关的业务场景也不用关心一共有多少页。

排序选项也通过Pageable实例处理。如果只需要排序，那么在org.springframework.data.domain.Sort参数中添加一个参数即可。正如你可以看到的，只返回一个List也是可能的。在这种情况下，Page将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被发布的附加计数查询），而仅仅是限制查询仅查找给定范围的

实体。

### 2．限制查询结果

示例：在查询方法上加限制查询结果的关键字first和top。

```
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);
```

查询方法的结果可以通过关键字来限制first或top，其可以被互换地使用。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。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的Optional中。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。

## 3.6.2 查询结果的不同形式（List/Stream/Page/Future）

Page和List在上面的示例中都有涉及，下面介绍几种特殊的。

### 1．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：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);
```

**提示**

流的关闭问题，try cache是一种关闭方法。

```
Stream<User> stream;
try {
   stream = repository.findAllByCustomQueryAndStream()
   stream.forEach(…);
} catch (Exception e) {
   e.printStackTrace();
} finally {
   if (stream!=null){
      stream.close();
   }
}
```

## 2．异步查询结果

可以使用Spring的异步方法执行功能异步的存储库查询。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，比较适合定时任务的实际场景。

```
@Async
Future<User> findByFirstname(String firstname); (1)
@Async
CompletableFuture<User> findOneByFirstname(String firstname); (2)
@Async
ListenableFuture<User> findOneByLastname(String lastname);(3)
```

（1）使用java.util.concurrent.Future的返回类型。

（2）使用java.util.concurrent.CompletableFuture作为返回类型。

（3）使用

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

所支持的返回结果类型远不止这些（可参考附录2：Repository Query Method返回值类型）。可以根据实际的使用场景灵活选择。其中，Map和Object[]的返回结果也支持，这种方法不太推荐使用，因为没有用到对象思维，不知道结果里面装的是什么。

### 3.6.3 Projections对查询结果的扩展

Spring JPA对Projections扩展的支持是非常好的。从字面意思上理解就是映射，指的是和DB查询结果的字段映射关系。一般情况下，返回的字段和DB查询结果的字段是一一对应的，但有的时候，我们需要返回一些指定的字段，不需要全部返回，或者只返回一些复合型的字段，还要自己写逻辑。Spring Data正是考虑到了这一点，允许对专用返回类型进行建模，以便我们有更多的选择，将部分字段显示成视图对象。

假设Person是一个正常的实体，和数据表Person一一对应，正常的写法如下：

```
@Entity
class Person {
   @Id
   UUID id;
   String firstname, lastname;
   Address address;
   @Entity
   static class Address {
      String zipCode, city, street;
   }
}
interface PersonRepository extends Repository<Person, UUID> {
   Collection<Person> findByLastname(String lastname);
}
```

（1）我们想仅仅返回其中与name相关的字段，应该怎么做呢？基于projections的思路，其实是比较容易的。我们只需要声明一个接口，包含要返回的属性的方法即可，例如：

```
interface NamesOnly {
String getFirstname();
String getLastname();
}
```

Repository里面的写法如下，直接用这个对象接收结果即可：

```
interface PersonRepository extends Repository<Person, UUID> {
Collection<NamesOnly> findByLastname(String lastname);
}
```

在Ctroller里面直接调用对象可以查看结果。原理是运行时底层会有动态代理机制为这个接口生成一个实现实体类。

（2）查询关联的子对象，例如：

```
interface PersonSummary {
    String getFirstname();
    String getLastname();
    AddressSummary getAddress();
    interface AddressSummary {
        String getCity();
    }
}
```

（3）@Value和SPEL也支持：

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

PersonRepository里面保持不变，这样会返回一个firstname和lastname相加的只有fullName的结果集合。

（4）对Spel表达式的支持远不止这些：

```
@Component
class MyBean {
    String getFullName(Person person) {
        …//自定义的运算
    }
}

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

（5）还可以通过Spel表达式取到方法里面的参数值。

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

（6）这时可能有人会想，只能用interface吗？Dto支持吗？其实也是可以的，我们也可以定义自己的Dto实体类。需要哪些字段，我们直接在Dto类中使用get/set属性即可。例如：

```
class NamesOnlyDto {
    private final String firstname, lastname;
       //注意构造方法
    NamesOnlyDto(String firstname, String lastname) {
       this.firstname = firstname;
       this.lastname = lastname;
    }

    String getFirstname() {
        return this.firstname;
    }

    String getLastname() {
        return this.lastname;
    }
}
```

（7）支持动态projections。通过泛化，可以根据不同的业务情况返回不同的字段集合。可以对PersonRepository做一定的变化，例如：

```
interface PersonRepository extends Repository<Person, UUID> {
   Collection<T> findByLastname(String lastname, Class<T> type);
}

//调用方可以通过 class 类型动态指定返回不同字段的结果集合
void someMethod(PersonRepository people) {
   //想包含全字段，直接用原始 entity（Person.class）接收即可
   Collection<Person> aggregates =
   people.findByLastname("Matthews", Person.class);
   //如果想仅仅返回名称，只需要指定 Dto 即可

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

Projections的应用场景还是挺多的，希望大家好好体会，以利用更优雅的代码实现不同的场景，不必用数组、冗余的对象去接收查询结果。

## 3.7 实现机制介绍

通过QueryExecutorMethodInterceptor这个类的源代码，我们发现这个类实现了MethodInterceptor接口。也就是说它是一个方法调用的拦截器，当一个Repository上的查询方法（譬如findByEmailAndLastname方法）被调用时，Advice拦截器会在方法真正地实现调用前先执行MethodInterceptor的invoke方法。这样我们就有机会在真正方法实现执行前执行其他的代码了。

对于QueryExecutorMethodInterceptor来说，最重要的代码并不在invoke方法中，而是在它的构造器QueryExecutorMethodInterceptor(RepositoryInformation r, Object customImplementation, Object target)中。

最重要的一段代码如下：

```
for (Method method : queryMethods) {
     // 使用 lookupStrategy，针对 Repository 接口上的方法查询 Query
     RepositoryQuery query=lookupStrategy.resolveQuery(method,repositoryInformation,
factory, namedQueries); invokeListeners(query);
     queries.put(method, query);
}
```

通过这个思路我们就可以找到很多具体的实现方法，其中有一个重要类，即PartTree，它包含了主要的算法逻辑。

一图胜千言，可以直接看图3-3。

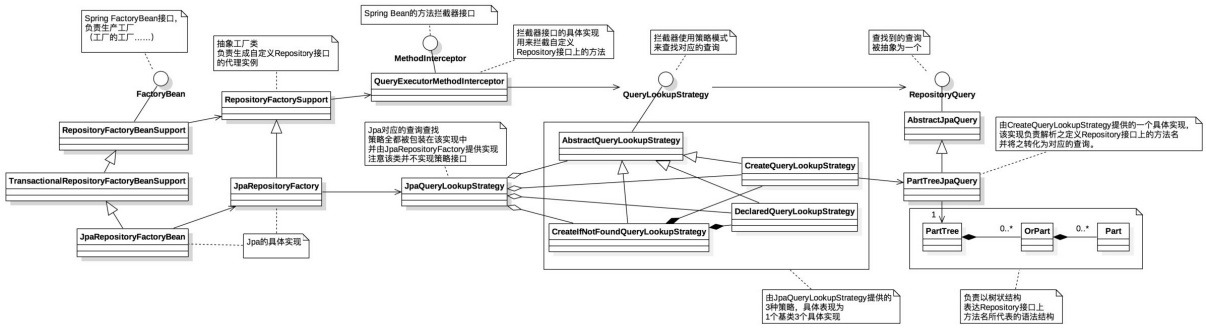



图3-3

# 第4章
# 注解式查询方法

“合抱之木，生于毫末；九层之台，起于累土；”

——老子

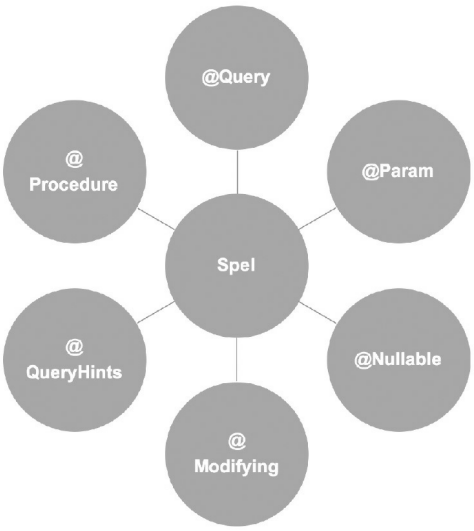



图4-1

本章详细讲解声明式的查询方法，即注解的查询用法大全。

## 4.1 @Query详解

### 4.1.1 语法及源码

先看一下语法及其源码：

```
public @interface Query {
/**
* 指定 JPQL 的查询语句。（nativeQuery=true 的时候，是原生的 SQL 语句）
*/
String value() default "";
/**
* 指定 count 的 JPQL 语句，如果不指定将根据 query 自动生成。
* （nativeQuery=true 的时候，是原生的 SQL 语句）
*/
String countQuery() default "";
/**
* 根据哪个字段来 count，一般默认即可。
*/
String countProjection() default "";
/**
* 默认是 false，表示 value 里面是不是原生的 Sql 语句
*/
boolean nativeQuery() default false;
/**
* 可以指定一个 query 的名字，必须是唯一的。
* 如果不指定，默认的生成规则是：
* {$domainClass}.${queryMethodName}
*/
String name() default "";
/*
* 可以指定一个 count 的 query 名字，必须是唯一的。
* 如果不指定，默认的生成规则是：
* {$domainClass}.${queryMethodName}.count
*/
String countName() default "";
}
```

### 4.1.2 @Query用法

使用命名查询为实体声明查询是一种有效的方法，对于少量查询很有效。一般只需要关心@Query里面的value和nativeQuery的值。使用声明式JPQL查询有一个好处，就是启动的时候就知道语法正确与否。

【示例4.1】声明一个注解在Repository的查询方法上。

```
public interface UserRepository extends JpaRepository<User, Long> {
    @Query("select u from User u where u.emailAddress = ?1")
    User findByEmailAddress(String emailAddress);
}
```

【示例4.2】Like查询，注意firstname不会自动加上%关键字的。

```
public interface UserRepository extends JpaRepository<User, Long> {
    @Query("select u from User u where u.firstname like %?1")
    List<User> findByFirstnameEndsWith(String firstname);
}
```

【示例4.3】直接用原始SQL。

```
public interface UserRepository extends JpaRepository<User, Long> {
    @Query(value = "SELECT * FROM USERS WHERE EMAIL_ADDRESS = ?1", nativeQuery
= true)
    User findByEmailAddress(String emailAddress);
}
```

**提示**

nativeQuery不支持直接Sort的参数查询。

【示例4.4】nativeQuery排序的错误写法，下面这个是启动不起来的。

```
public interface UserRepository extends JpaRepository<User, Long> {
    @Query(value = "select * from user_info where first_name=?1", nativeQuery
= true)
    List<UserInfoEntity> findByFirstName(String firstName, Sort sort);
}
```

【示例4.5】nativeQuery排序的正确写法。

```
    @Query(value = "select * from user_info where first_name=?1 order by ?2",
nativeQuery = true)
    List<UserInfoEntity> findByFirstName(String firstName,String sort);
    //调用的地方 last_name 是数据里面的字段名，不是对象的字段名
    repository.findByFirstName("jackzhang","last_name");
```

### 4.1.3 @Query排序

@Query在JPQL下想实现排序，直接用PageRequest或者直接用Sort参数都可以。

在排序实例中实际使用的属性需要与实体模型里面的字段相匹配，这意味着它们需要解析为查询中使用的属性或别名。这是一个state_field_path_expression JPQL定义，并且Sort的对象支持一些特定的函数。

【示例4.6】Sort and JpaSort的使用。

```
    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);
}

//调用方的写法，如下：
repo.findByAndSort("lannister",new Sort("firstname"));
repo.findByAndSort("stark",new Sort("LENGTH(firstname)"));
repo.findByAndSort("targaryen",JpaSort.unsafe("LENGTH(firstname)"));
repo.findByAsArrayAndSort("bolton",new Sort("fn_len"));
```

### 4.1.4 @Query分页

【示例4.7】直接用Page对象接收接口，参数直接用Pageable的实现类即可。

```
   public interface UserRepository extends JpaRepository<User, Long> {
  @Query(value = "select u from User u where u.lastname = ?1")
  Page<User> findByLastname(String lastname, Pageable pageable);
}
   //调用者的写法
   repository.findByFirstName("jackzhang",new PageRequest(1,10));
```

【示例4.8】对原生SQL（以MySQL为例）的分页支持示例，但是支持得不是特别友好。

```
   import org.springframework.data.jpa.repository.Query;

   public interface UserRepository extends JpaRepository<UserInfoEntity, Integer>,
JpaSpecificationExecutor<UserInfoEntity> {
       @Query(value = "select * from user_info where first_name=?1 /* #pageable#
*/", countQuery = "select count(*) from user_info where first_name=?1", nativeQuery
= true)
       Page<UserInfoEntity> findByFirstName(String firstName, Pageable pageable);
   }
   //调用者的写法
   return userRepository.findByFirstName("jackzhang",
   new PageRequest(1,10, Sort.Direction.DESC,"last_name"));
   //打印出来的 sql
   select * from user_info where first_name=? /* #pageable# */ order by last_name
desc limit ?, ?
```

**提示**

注释/* #pageable# */必须有。估计随着版本的变化这个会做优化。另外一种实现方法就是自己写两个查询方法，手动分页。

## 4.2 @Param用法

默认情况下，参数是通过顺序绑定在查询语句上的。这使得查询方法对参数位置的重构容易出错。为了解决这个问题，你可以使用@Param注解指定方法参数的具体名称，通过绑定的参数名字做查询条件。

【示例4.9】根据参数进行查询。

```
   import org.springframework.data.jpa.repository.Query;

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);
}
```

## 4.3 SpEL表达式的支持

在Spring Data JPA 1.4以后，支持在@Query中使用SpEL表达式（简介）来接收变量。

SpEL支持的变量如表4-1所示。

表4-1 SpEL支持的变量

| 变量名 | 使用方式 | 描述 |
| --- | --- | --- |
| entityName | select x from #{#entityName} x | 根据指定的 Repository 自动插入相关的 entityName。<br>有两种方式能被解析出来：<br>（1）如果定义了@Entity 注解，直接用其属性名<br>（2）如果没定义，直接用实体类的名称 |

在以下的例子中，我们在查询语句中插入表达式：

```
   @Entity("User")
   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);
   }
```

这个SpEL的支持比较适合自定义的Repository，如果想写一个通

用的Repository接口，那么可以用这个表达式来处理：

```
@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> {
    …
}
```

MappedTypeRepository作为一个公用的父类，自己的Repository可以继承它，当调用ConcreteRepository执行findAllByAttribute方法的时候执行结果如下：

```
select t from ConcreteType t where t.attribute = ?1
```

## 4.4 @Modifying修改查询

先看源码：

```
public @interface Modifying {
    //如果配置了一级缓存，这个时候用 clearAutomatically=true，就会刷新 hibernate 的一级缓存，不然你在同一接口中更新一个对象，接着查询这个对象，查出来的对象就是没有更新之前的状态。
    boolean clearAutomatically() default false;
}
```

可以通过在@Modifying注解实现只需要参数绑定的update查询的执行：

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

简单地针对某些特定属性的更新也可以直接用基类里面提供的通用save。还有第三种方法，就是自定义Repository，使用EntityManager来进行更新操作。

对删除操作的支持如下：

```
interface UserRepository extends Repository<User, Long> {
   void deleteByRoleId(long roleId);

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

两种方式：一种是通过方法表达式（参见第3章），另一种是@Modifying和@Query注解。

## 4.5 @QueryHints

有很多数据库支持Hint Query的语法，不过这种查询支持比较老旧，感觉应该会慢慢被淘汰，工作中很少有人使用。Spring Data JPA还是做了很好的支持，它只支持一些固定的HintValue值，用来优化Query的作用。有两个注解需要了解和知道一下@QueryHints，value等于多个@QueryHint。

用法如下：

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

@QueryHint中的name是固定在类QueryHints中的，只能到这里选。接着我们看一下QueryHints的源码：

```
package org.hibernate.jpa;
.......
public class QueryHints {
//指定此处查询的超时时间，毫秒
public static final String SPEC_HINT_TIMEOUT = TIMEOUT_JPA;
//支持数据的 comment 的 hint 提示语法
public static final String HINT_COMMENT = COMMENT;
//每次 fetch 的大小
public static final String HINT_FETCH_SIZE = FETCH_SIZE;
//是否开启缓存，需要配合一级缓存使用，不建议用
public static final String HINT_CACHEABLE = CACHEABLE;
public static final String HINT_CACHE_REGION = CACHE_REGION;
//是否只读
public static final String HINT_READONLY = READ_ONLY;
public static final String HINT_CACHE_MODE = CACHE_MODE;
public static final String HINT_FLUSH_MODE = FLUSH_MODE;
public static final String HINT_NATIVE_LOCKMODE = NATIVE_LOCKMODE;
public static final String HINT_FETCHGRAPH = FETCHGRAPH;
//配置 EntityGraph 的两种值 FetchType.LAZY 或者 FetchType.EAGER
public static final String HINT_LOADGRAPH = LOADGRAPH;
.......
}
```

QueryHint仅仅了解一下即可，一般的业务场景基本不用。

## 4.6 @Procedure储存过程的查询方法

我们通过@Procedure来介绍一下JPA对储存过程的支持。

（1）@Procedure源码如下：

```
public @interface Procedure {
    // 数据库里面储存过程的名称
    String value() default "";
    // 数据库里面储存过程的名称
    String procedureName() default "";
    //在 EntityManager 中的名字，NamedStoredProcedureQuery 使用
    String name() default "";
    //输出参数的名字
    String outputParameterName() default "";
}
```

（2）首先创建一个储存过程名字plus1inout，有两个参数、两个结果。

```
CREATE PROCEDURE plus1inout(IN arg int, OUT res int)
BEGIN
SELECT (arg+10) into res;
END
```

（3）使用@NamedStoredProcedureQueries注释来调用存储过程。这个必须定义在一个实体上面。

```
   @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 {
   //这是一个 Procedure 实体类，可以通过 NamedStoredProcedureQueries 在这个类里面定义多个储存过程的查询
   }
```

关键要点：

- 存储过程使用了注释@NamedStoredProcedureQuery，并绑定到一个JPA表。
- procedureName是存储过程的名字。
- name是JPA中存储过程的名字。
- 使用注释@StoredProcedureParameter来定义存储过程使用的IN/OUT参数。

（4）直接通过自定义过的Repository完成储存过程的调用。

```
   public interface MyUserRepository extends CrudRepository<User, Long> {
      @Procedure("plus1inout")
      //通过储存过程的名字
      Integer explicitlyNamedPlus1inout(Integer arg);

      //通过储存过程的名字
      @Procedure(procedureName = "plus1inout")
      Integer plus1inout(Integer arg);

      @Procedure(name = "User.plus1IO")
      //自定义的储存过程的名字
      Integer entityAnnotatedCustomNamedProcedurePlus1IO(@Param("arg") Integer arg);
   }
```

关键要点：

- @Procedure的procedureName参数必须匹配

@NamedStoredProcedureQuery的procedureName。

- @Procedure的name参数必须匹配@NamedStoredProcedureQuery的name。
- @Param必须匹配@StoredProcedureParameter注释的name参数。
- 返回类型必须匹配：in_only_test存储过程返回是void，in_and_out_test存储过程必须返回String。

# 4.7 @NamedQueries预定义查询

## 4.7.1 简介

这是预定义查询的一种形式。

（1）在@Entity下增加@NamedQuery定义。

```
public @interface NamedQuery {
    //query 的名称，规则：实体.方法名；
    String name();

    //具体的 JPQL 查询语法
    String query();
}
```

需要注意，query里面的值也是JPQL。查询参数也要和实体对应起来。因为实际场景中这种破坏Entity的侵入式很不美，也不方便，所以这种方式容易遗忘，工作中也很少推荐。

（2）与之相对应的还有@NamedNativeQuery。用法一样，唯一不一样的是，query里面放置的是原生SQL语句，而非实体的字段名字。

## 4.7.2 用法举例

（1）实体里面的写法：

```
@Entity
@NamedQuery(name="Customer.findByFirstName",
query = "select c from Customer c where c.firstName = ?1")
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String firstName;
    private String lastName;
    ......
}
```

（2）CustomerRepository里面的写法：

```
Customer findByFirstName(String bauer);
```

（3）调用者的写法：

```
Customer customer = repository.findByFirstName("Bauer");
```

### 4.7.3 @NamedQuery、@Query和方法定义查询的对比

（1）Spring JPA里面的优先级，@Query > @NameQuery > 方法定义查询。

（2）推荐使用的优先级：@Query > 方法定义查询 > @NameQuery。

（3）相同点是都不支持动态条件查询。

# 第5章
# @Entity实例里面常用注解详解

工作的最重要的动力是工作中的乐趣，是工作获得结果时的乐趣以及对这个结果的社会价值的认识。

—— 爱因斯坦

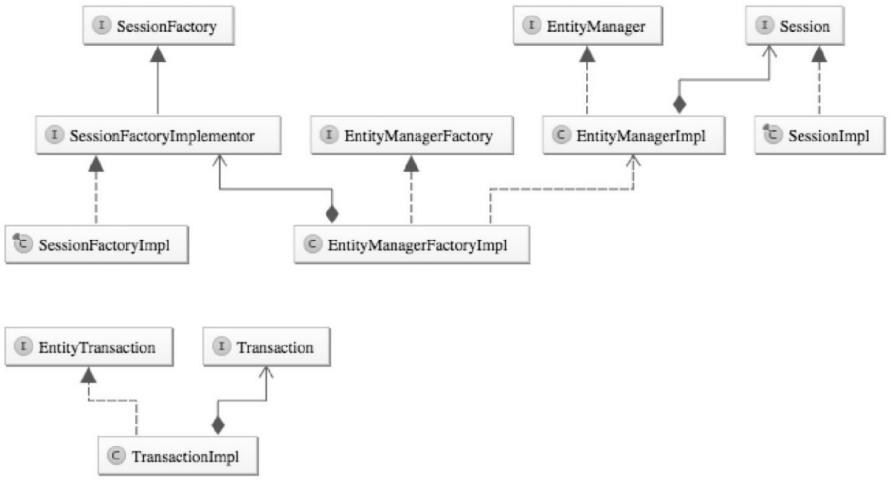



图5-1

本章详细介绍javax.persistence下面的Entity中常用的注解。本章学习的基本条件是要对Java的注解有基本的了解。

## 5.1 javax.persistence概况介绍

虽然Spring Data JPA已经帮我们对数据的操作封装得很好了，约定大于配置思想，帮我们默认了很多东西。JPA（Java持久性API）是存储业务实体关联的实体来源。它显示了如何定义一个面向普通Java对象（POJO）作为一个实体，以及如何与管理关系实体提供一套标准。因此，javax.persistence下面的有些注解还是必须要去了解的，以便于更好地提高工作效率。

（1）javax.persistence位于hibernate-jpa-**.jar包里面，可以通过Intellij Idea的maven插件直接分析一下maven的依赖，也可以用$ mvn dependency:tree分析，例如：

```
   com.jackzhang.example:quick_start:jar:0.0.1-SNAPSHOT
   +- org.springframework.boot:spring-boot-starter-data-jpa:jar:1.5.8.RELEASE
   |  +- org.hibernate:hibernate-core:jar:5.0.12.Final
   |  |  +-
org.hibernate.javax.persistence:hibernate-jpa-2.1-api:jar:1.0.0.Final
   |  +- org.hibernate:hibernate-entitymanager:jar:5.0.12.Final
```

（2）通过Intellij Idea的Diagram来看一下此模块类的关键关系，如图5-2所示。

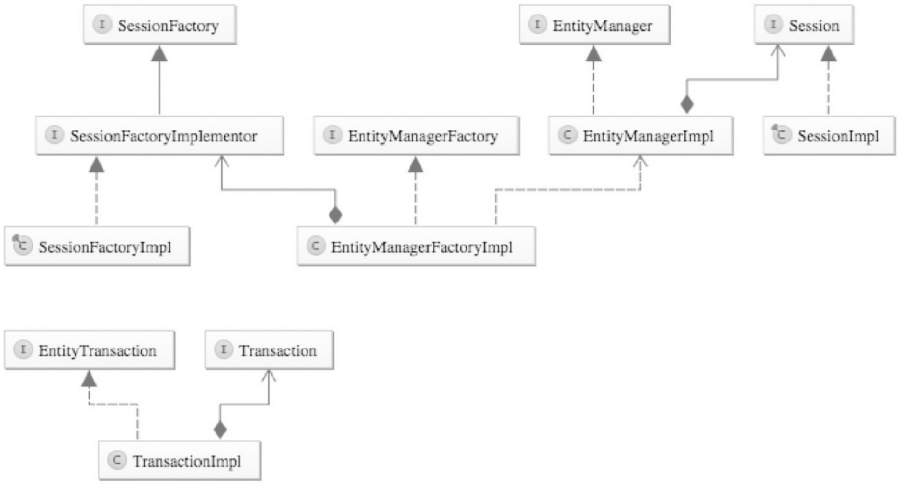



图5-2

（3）图5-3显示了JPA的类的层次结构，包括核心类和JPA接口。

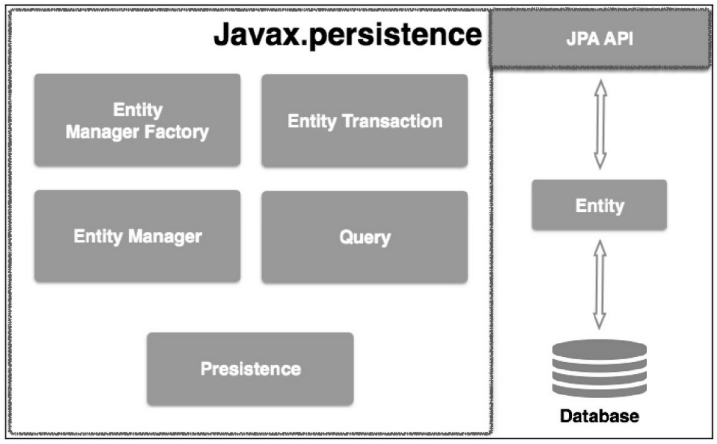



图5-3

表5-1描述了每个在上述架构的显示单元。

表5-1　JPA类层次结构的显示单元

| 单元 | 描述 |
| --- | --- |
| EntityManagerFactory | 一个 EntityManager 的工厂类，创建并管理多个 EntityManager 实例 |
| EntityManager | 一个接口，管理持久化操作的对象，工作原理类似工厂的查询实例 |
| Entity | 实体是持久性对象，是存储在数据库中的记录 |
| EntityTransaction | 与 EntityManager 是一对一的关系。对于每一个 EntityManager，操作是由 EntityTransaction 类维护的 |
| Persistence | 这个类包含静态方法来获取 EntityManagerFactory 实例 |
| Query | 该接口由每个 JPA 供应商实现，能够获得符合标准的关系对象 |

上述的类和接口用于存储实体到数据库的一个记录，帮助程序员通过减少自己编写的代码将数据存储到数据库中，使他们能够专注于更重要的业务活动代码，如数据库表映射的类编写代码。

下面我们主要介绍一下在Entity里面常用的注解，对于其他没有介绍到的注解，建议读者直接到包的源码里面进行查找和分析。

## 5.2　基本注解

基本注解包括@Entity、@Table、@Id、@IdClass、@GeneratedValue、@Basic、@Transient、@Column、@Temporal、@Enumerated、@Lob。

### 5.2.1　@Entity

先看一个Blog的示例，其中实体的配置如下：

```
@Entity
@Table(name = "user_blog", schema = "test")
public class UserBlogEntity {
   @Id
   @Column(name = "id", nullable = false)
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Integer id;
   @Column(name = "title", nullable = true, length = 200)
   private String title;
   @Basic
   @Column(name = "create_user_id", nullable = true)
   private Integer createUserId;
   @Basic
   @Column(name = "blog_content", nullable = true, length = -1)
   @Lob
   private String blogContent;
   @Basic(fetch = FetchType.LAZY)
   @Column(name = "image", nullable = true)
   @Lob
   private byte[] image;
   @Basic
   @Column(name = "create_time", nullable = true)
   @Temporal(TemporalType.TIMESTAMP)
   private Date createTime;
   @Basic
   @Column(name = "create_date", nullable = true)
   @Temporal(TemporalType.DATE)
   private Date createDate;
   @Transient
   private String transientSimple;
......
}
```

@Entity定义对象将会成为被JPA管理的实体，将映射到指定的数据库表。

```
public @interface Entity {
   //可选，默认是此实体类的名字，全局唯一
   String name() default "";
}
```

## 5.2.2 @Table

@Table指定数据库的表名。

```
public @interface Table {
    //表的名字，可选。如果不填写，系统认为好实体的名字一样为表名
    String name() default "";
    //此表的 catalog，可选
    String catalog() default "";
    //此表所在的 schema，可选
    String schema() default "";
    //唯一性约束，只有创建表的时候有用，默认不需要
    UniqueConstraint[] uniqueConstraints() default { };
    //索引，只有创建表的时候使用，默认不需要
    Index[] indexes() default {};
}
```

### 5.2.3 @Id

@Id定义属性为数据库的主键，一个实体里面必须有一个。

### 5.2.4 @IdClass

@IdClass利用外部类的联合主键。

（1）源码：

```
public @interface IdClass {
    //联合主键的类
    Class value();
}
```

作为符合主键类，要满足以下几点要求。

- 必须实现Serializable接口。
- 必须有默认的public无参数的构造方法。
- 必须覆盖equals和hashCode方法。equals方法用于判断两个对象是否相同，EntityManger通过find方法来查找Entity时是根据equals的返回值来判断的。在本例中，只有对象的name和email值完全相同或同一个对象时才返回true，否则返回false。hashCode方法返回当前对象的哈希码，生成的

hashCode相同的概率越小越好，算法可以进行优化。

（2） 用法：

① 我们假设UserBlog的联合主键是createUserId和title，新增一个UserBlogKey的类。UserBlogKey.class代码如下：

```
import java.io.Serializable;
public class UserBlogKey implements Serializable {
   private String title;
   private Integer createUserId;
   public UserBlogKey() {
   }
   public UserBlogKey(String title, Integer createUserId) {
      this.title = title;
      this.createUserId = createUserId;
   }
   ...../get set 方法略
}
```

② UserBlogEntity.java要稍加改动：实体类上需要加@IdClass注解，主键上都得加@Id。

```
@Entity
@Table(name = "user_blog", schema = "test")
@IdClass(value = UserBlogKey.class)
public class UserBlogEntity {
   @Column(name = "id", nullable = false)
   private Integer id;
   @Id
   @Column(name = "title", nullable = true, length = 200)
   private String title;
   @Id
   @Column(name = "create_user_id", nullable = true)
   private Integer createUserId;
......//不变的部分省略
}
```

③ UserBlogRepository中的改动如下：

```
public interface UserBlogRepository extends
JpaRepository<UserBlogEntity,UserBlogKey>{
}
```

④ 使用的时候：

```
    @RequestMapping(path = "/blog/{title}/{createUserId}")
    @ResponseBody
    public UserBlogEntity showBlogs(@PathVariable(value = "createUserId") Integer
createUserId,@PathVariable("title") String title) {
        return userBlogRepository.findOne(new UserBlogKey(title,createUserId));
    }
```

## 5.2.5 @GeneratedValue

@GeneratedValue为主键生成策略，例如：

```
    public @interface GeneratedValue {
        //Id 的生成策略
        GenerationType strategy() default AUTO;
        //通过 Sequences 生成 Id，常见的是 Orcale 数据库 ID 生成规则，需要配合
@SequenceGenerator 使用
        String generator() default "";
    }
```

GenerationType一共有以下4个值：

```
    public enum GenerationType {
        //通过表产生主键，框架由表模拟序列产生主键，使用该策略可以使应用更易于数据库移植
        TABLE,
        //通过序列产生主键，通过 @SequenceGenerator 注解指定序列名， MySQL 不支持这种方式
        SEQUENCE,
        //采用数据库 ID 自增长， 一般用于 MySQL 数据库
        IDENTITY,
        //JPA 自动选择合适的策略，是默认选项
        AUTO
    }
```

## 5.2.6 @Basic

@Basic表示属性是到数据库表的字段的映射。如果实体的字段上没有任何注解，默认即为@Basic。

```
public @interface Basic {
    //可选，EAGER（默认）为立即加载，LAZY 为延迟加载（LAZY 主要应用在大字段上面）
    FetchType fetch() default EAGER;
    //可选，设置这个字段是否可以为 null，默认是 true
    boolean optional() default true;
}
```

## 5.2.7 @Transient

@Transient表示该属性并非一个到数据库表的字段的映射，表示非持久化属性，与@Basic作用相反。JPA映射数据库的时候忽略它。

## 5.2.8 @Column

@Column定义该属性对应数据库中的列名。

```
public @interface Column {
    //数据库中表的列名。可选，如果不填写认为字段名和实体属性名一样
    String name() default "";
    //是否唯一，默认 flase，可选
    boolean unique() default false
    //数据字段是否允许空。可选，默认 true
    boolean nullable() default true;
    //执行 insert 操作的时候是否包含此字段，默认 true，可选
    boolean insertable() default true;
    //执行 update 的时候是否包含此字段，默认 true，可选
    boolean updatable() default true;
    //表示该字段在数据库中的实际类型
    String columnDefinition() default "";
    //数据库字段的长度，可选，默认255
    int length() default 255;
}
```

## 5.2.9 @Temporal

@Temporal用来设置Date类型的属性映射到对应精度的字段。

（1）@Temporal(TemporalType.DATE)映射为日期//date（只有日期）。

（2）@Temporal(TemporalType.TIME)映射为日期//time（只有时间）。

（3）@Temporal(TemporalType.TIMESTAMP)映射为日期//datetime（日期+时间）。

## 5.2.10 @Enumerated

@Enumerated很好用，直接映射enum枚举类型的字段。

（1）看源码：

```
public @interface Enumerated {
//枚举映射的类型，默认是ORDINAL（枚举字段的下标）
    EnumType value() default ORDINAL;
}
public enum EnumType {
     //映射枚举字段的下标
    ORDINAL,
     //映射枚举的Name
    STRING
}
```

（2）看例子：

```
 //有一个枚举类，用户的性别
 public enum Gender {
     MAIL("男性"), FMAIL("女性");
    private String value;
    private Gender(String value) {
        this.value = value;
    }
 }
 //实体类@Enumerated的写法如下
 @Entity
 @Table(name = "tb_user")
 public class User implements Serializable {
    @Enumerated(EnumType.STRING)
    @Column(name = "user_gender")
    private Gender gender;
    ......................
 }
```

这时插入两条数据，数据库里面的值是MAIL/FMAIL，而不是“男性”/“女性”。如果我们用@Enumerated(EnumType.ORDINAL)，那么这时数据库里面的值是0,1。但是实际工作中，不建议用数字下标，因为枚举里面的属性值是会不断新增的，如果新增一个，位置变化了就惨了。

## 5.2.11 @Lob

@Lob 将属性映射成数据库支持的大对象类型，支持以下两种数据库类型的字段。

（1）Clob（Character Large Ojects）类型是长字符串类型，java.sql.Clob、Character[]、char[]和String将被映射为Clob类型。

（2）Blob（Binary Large Objects）类型是字节类型，java.sql.Blob、Byte[]、byte[]和实现了Serializable接口的类型将被映射为Blob类型。

（3）Clob、Blob占用内存空间较大，一般配合@Basic(fetch=FetchType.LAZY)将其设置为延迟加载。

## 5.2.12 几个注释的配合使用

@SqlResultSetMapping、@EntityResult、@ColumnResult可以配合@NamedNativeQuery一起使用，但是实际工作中不建议这样配置。

下面看一个简单的示例：

```
@NamedNativeQueries({
       @NamedNativeQuery(name = "getUsers",
       query="select id,username,usertype from t_xfw_operator order by id desc",
       resultSetMapping = "usersMap")
})
@SqlResultSetMappings({
       @SqlResultSetMapping(name = "usersMap",
       entities = {},
       columns = {
              @ColumnResult(name = "id"),
              @ColumnResult(name="username"),
              @ColumnResult(name="usertype")
       })
})
@Entity
@Table(name = "operator")
public class Operator {
......
}
```

## 5.3 关联关系注解

关联关系注解包括@JoinColumn、@OneToOne、@OneToMany、@ManyToOne、@ManyToMany、@JoinTable、@OrderBy。

### 5.3.1 @JoinColumn定义外键关联的字段名称

（1）源码语法如下：

```
public @interface JoinColumn {
    //目标表的字段名,必填
    String name() default "";
    //本实体的字段名，非必填，默认是本表 ID
    String referencedColumnName() default "";
    //外键字段是否唯一
    boolean unique() default false;
    //外键字段是否允许为空
    boolean nullable() default true;
    //是否跟随一起新增
    boolean insertable() default true;
    //是否跟随一起更新
    boolean updatable() default true;
}
```

（2）用法：@JoinColumn主要配合@OneToOne、@ManyToOne、@OneToMany一起使用，单独使用没有意义。

（3）@JoinColumns定义多个字段的关联关系。

## 5.3.2 @OneToOne关联关系

（1）源码语法如下：

```
public @interface OneToOne {
    //关系目标实体，非必填，默认该字段的类型
    Class targetEntity() default void.class;
    //cascade 级联操作策略
  1. CascadeType.PERSIST 级联新建
  2. CascadeType.REMOVE 级联删除
  3. CascadeType.REFRESH 级联刷新
  4. CascadeType.MERGE 级联更新
  5. CascadeType.ALL 四项全选
  6. 默认，关系表不会产生任何影响
    CascadeType[] cascade() default {};
    //数据获取方式，EAGER(立即加载)/LAZY(延迟加载)
    FetchType fetch() default EAGER;
    //是否允许为空
    boolean optional() default true;
    //关联关系被谁维护，非必填，一般不需要特别指定
   //注意：只有关系维护方才能操作两者的关系，被维护方即使设置了维护方属性进行存储也不会更新外键关联。（1）mappedBy 不能与@JoinColumn 或者@JoinTable 同时使用。（2）mappedBy 的值是指另一方的实体里面属性的字段，而不是数据库字段，也不是实体的对象的名字。即另一方配置了@JoinColumn 或者@JoinTable 注解的属性的字段名称。
    String mappedBy() default "";
    //是否级联删除，和 CascadeType.REMOVE 的效果一样，只要配置了两种中的一种就会自动级联删除
    boolean orphanRemoval() default false;
}
```

（2）用法：@OneToOne需要配合@JoinColumn一起使用。注意：可以双向关联，也可以只配置一方，需要视实际需求而定。

【示例5.1】假设一个部门只有一个员工。Department的内容如下：

```
@OneToOne
@JoinColumn(name="employee_id",referencedColumnName="id")
private Employee employeeAttribute = new Employee();
```

**提示**

employee_id指的是Department里面的字段，而referencedColumnName="id"指的是Employee表里面的字段。

如果需要双向关联，Employee的内容如下：

```
@OneToOne(mappedBy="employeeAttribute")
private Department department;
```

当然也可以不选用mappedBy，和下面效果是一样的：

```
@OneToOne
@JoinColumn(name="id",referencedColumnName="employee_id")
private Department department;
```

### 5.3.3 @OneToManyg与@ManyToOne关联关系

@OneToMany与@ManyToOne可以相对存在，也可只存在一方。

（1）@OneToMany源码语法如下：

```
    public @interface OneToMany {
        Class targetEntity() default void.class;
        //cascade 级联操作策略：(CascadeType.PERSIST、CascadeType.REMOVE、
CascadeType.REFRESH、CascadeType.MERGE、CascadeType.ALL)。如果不填，默认关系表不会产
生任何影响
        CascadeType[] cascade() default {};
        //数据获取方式，EAGER(立即加载)/LAZY(延迟加载)
        FetchType fetch() default LAZY;
         //关系被谁维护，单项的。注意：只有关系维护方才能操作两者的关系
        String mappedBy() default "";
        //是否级联删除，和 CascadeType.REMOVE 的效果一样，配置了两种中的一种就会自动级联删除
        boolean orphanRemoval() default false;
    }
    public @interface ManyToOne {
        Class targetEntity() default void.class;
        CascadeType[] cascade() default {};
        FetchType fetch() default EAGER;
        boolean optional() default true;
    }
```

@ManyToOne与OneToMany的源码稍有区别，仔细体会。

（2）使用示例，必须和@JoinColumn配合使用才有效。

```
    @Entity
    @Table(name="user")
    public class User implements Serializable{
    @OneToMany(cascade=CascadeType.ALL,fetch=FetchType.LAZY,mappedBy="user")
        private Set<role> setRole;
    ......}
    @Entity
    @Table(name="role")
    public class Role {
    @ManyToOne(cascade=CascadeType.ALL,fetch=FetchType.EAGER)
         @JoinColumn(name="user_id")//user_id 字段作为外键
        private User user;
    ......}
```

## 5.3.4 @OrderBy关联查询时排序

一般和@OneToMany一起使用。

（1）源码语法如下：

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface OrderBy {
   /**
    * 要排序的字段，格式如下：
    *    orderby_list::= orderby_item [,orderby_item]*
    *    orderby_item::= [property_or_field_name] [ASC | DESC]
    * 字段可以是实体属性，也可以是数据字段，默认 ASC。
    */
    String value() default "";
}
```

（2）用法示例：

```
@Entity
@Table(name="user")
public class User implements Serializable{
@OneToMany(cascade=CascadeType.ALL,fetch=FetchType.LAZY,mappedBy="user")
@OrderBy("role_name DESC")
   private Set<role> setRole;
......}
```

## 5.3.5 @JoinTable关联关系表

如果对象与对象之间有一个关联关系表的时候，就会用到@JoinTable，一般和@ManyToMany一起使用。

（1）源码语法如下：

```
public @interface JoinTable {
    //中间关联关系表名
    String name() default "";
    //表的 catalog
    String catalog() default "";
    //表的 schema
    String schema() default "";
    //主链接表的字段
    JoinColumn[] joinColumns() default {};
    //被联机的表外键字段
    JoinColumn[] inverseJoinColumns() default {};
......
}
```

（2）假设Blog和Tag是多对多的关系，有一个关联关系表

blog_tag_relation，表中有两个属性blog_id和tag_id，那么Blog实体里面的写法如下：

```
   @Entity
   public class Blog{
      @ManyToMany
      @JoinTable(
         name="blog_tag_relation",
         joinColumns=@JoinColumn(name="blog_id",referencedColumnName="id"),

inverseJoinColumn=@JoinColumn(name="tag_id",referencedColumnName="id")
         private List<Tag> tags = new ArrayList<Tag>();
      )
   }
```

## 5.3.6 @ManyToMany关联关系

（1）源码语法如下：

```
   public @interface ManyToMany {
      Class targetEntity() default void.class;
      CascadeType[] cascade() default {};
      FetchType fetch() default LAZY;
      String mappedBy() default "";
   }
```

@ManyToMany表示多对多，和@OneToOne、@ManyToOne一样也有单向、双向之分。单向双向和注解没有关系，只看实体类之间是否相互引用。

（2）示例：一个博客可以拥有多个标签，一个标签也可以使用在多个博客上，Blog和Tag就是多对多关系。

```
    //单向多对多
    @Entity
    public class Blog{
        @Id
        @Column(name = "id")
        private Integer id;
        @ManyToMany(cascade=CascadeType.ALL)
        @JoinTable(
            name="blog_tag_relation",
            joinColumns=@JoinColumn(name="blog_id",referencedColumnName="id"),

inverseJoinColumn=@JoinColumn(name="tag_id",referencedColumnName="id")
        private List<Tag> tags = new ArrayList<Tag>();
    ......
    }
    @Entity
    public class BlogTagRelation{
       @Column(name = "blog_id")
        private Integer blogId;
       @Column(name = "tag_id")
        private Integer tagId;
    ......
    }
    @Entity
    public class Tag{
        @Id
        @Column(name = "id")
        private Integer id;
    ......}
    //双向多对多
    @Entity
    public class Blog{
        @ManyToMany(cascade=CascadeType.ALL)
        @JoinTable(
            name="blog_tag_relation",
            joinColumns=@JoinColumn(name="blog_id",referencedColumnName="id"),

inverseJoinColumn=@JoinColumn(name="tag_id",referencedColumnName="id")
        private List<Tag> tags = new ArrayList<Tag>();
    }
    @Entity
    public class Tag{
        @ManyToMany(mappedBy="BlogTagRelation")
        private List<Blog> blogs = new ArrayList<Blog>();
    }
```

## 提示

BlogTagRelation为中间关联关系表blog_tag_relation对应的实体。

## 5.4 Left join、Inner join与@EntityGraph

### 5.4.1 Left join与Inner join

当使用@ManyToMany、@ManyToOne、@OneToMany、@OneToOne关联关系的时候，FetchType怎么配置LAZY或者EAGER。SQL真正执行的时候是由一条主表查询和N条子表查询组成的。这种查询效率一般比较低下，比如子对象有N个就会执行N+1条SQL。

有时候我们需要用到Left Join或者Inner Join来提高效率，只能通过@Query的JQPL语法实现，后面我们将讲到的Criteria API也可以做到。Spring Data JPA为了简单地提高查询率，引入了EntityGraph的概念，可以解决N+1条SQL的问题。

### 5.4.2 @EntityGraph

JPA 2.1推出来的@EntityGraph、@NamedEntityGraph用来提高查询效率，很好地解决了N+1条SQL的问题。两者需要配合起来使用，缺一不可。@NamedEntityGraph配置在@Entity上面，而@EntityGraph配置在Repository的查询方法上面。我们来看一下实例。

（1）先在Entity里面定义@NamedEntityGraph，其他都不变。其中，@NamedAttributeNode可以有多个，也可以有一个。

```
    @NamedEntityGraph(name = "UserInfoEntity.addressEntityList", attributeNodes =
{
        @NamedAttributeNode("addressEntityList"),
        @NamedAttributeNode("userBlogEntityList")})
    @Entity(name = "UserInfoEntity")
    @Table(name = "user_info", schema = "test")
    public class UserInfoEntity  implements Serializable {
       @Id
       @Column(name = "id", nullable = false)
       private Integer id;
       @OneToOne(optional = false)
       @JoinColumn(referencedColumnName = "id",name = "address_id",nullable = false)
       private UserReceivingAddressEntity addressEntityList;
       @OneToMany
       @JoinColumn(name = "create_user_id",referencedColumnName = "id")
       private List<UserBlogEntity> userBlogEntityList;
    ......
    }
```

（2）只需要在查询方法上加@EntityGraph注解即可，其中value就是@NamedEntityGraph中的Name。实例配置如下：

```
    public interface UserRepository extends JpaRepository<UserInfoEntity, Integer>{
       @Override
       @EntityGraph(value = "UserInfoEntity.addressEntityList")
       List<UserInfoEntity> findAll();
    }
```

## 5.5 关于关系查询的一些坑

（1）所有的注解要么全配置在字段上，要么全配置在get方法上，不能混用，混用就会启动不起来，但是语法配置没有问题。

（2）所有的关联都是支持单向关联和双向关联的，视具体业务场景而定。JSON序列化的时候使用双向注解会产生死循环，需要人为手动转化一次，或者使用@JsonIgnore。

（3）在所有的关联查询中，表一般是不需要建立外键索引的。@mappedBy的使用需要注意。

（4）级联删除比较危险，建议考虑清楚，或者完全掌握。

（5）不同的关联关系的配置，@JoinClumn里面的name、referencedColumnName代表的意思是不一样的，很容易弄混，可以根据打印出来的SQL做调整。

（6）当配置这些关联关系的时候建议大家直接在表上面，把外键建好，然后通过后面我们介绍的开发工具直接生成，这样可以减少自己调试的时间。

# 第二部分
# 晋级之高级部分

通过第4章和第5章的内容，我们会发现Spring Data JPA是Hibernate的JPA的一次包装和升级。有一种新瓶装旧酒，换汤不换药的感觉，老内容换了新包装，Hibernate过来的开发小伙伴们可以平滑过渡。所以相似的内容，希望小伙伴都能精通一门，这样新鲜玩意来了，也可以很快精通，这也正是技术触类旁通的特性。基本功掌握牢靠，任何用到的技术都值得深入研究、掌握全面一些。后面章节我们将要介绍一些Spring Data JPA表现优秀的地方，也正是我们如果使用其他ORM框架可能需要手动去实现的部分。

# 第6章
# JpaRepository扩展详解

哪里有天才，我是把别人喝咖啡的工夫都用在了工作上了。

—— 鲁迅

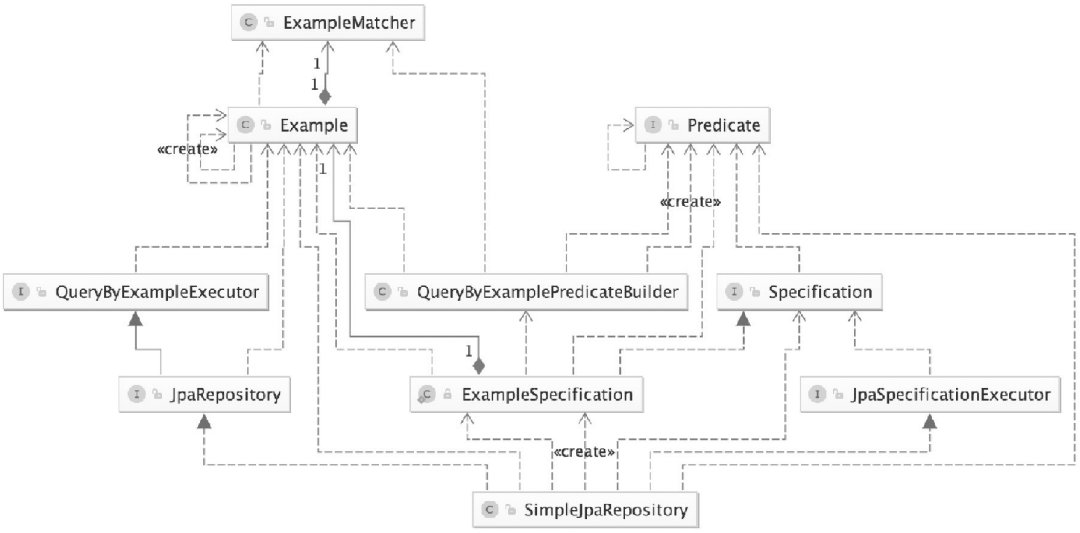



图6-1

本章我们主要介绍JpaRepostiory扩展的方法：简单的应用场景和实际工作中稍微复杂的应用场景。

## 6.1 JpaRepository介绍

从JpaRepository开始的子类，都是Spring Data项目对JPA实现的封装与扩展。JpaRepository本身继承PagingAndSortingRepository接口，是针对JPA技术的接口，提供flush()、saveAndFlush()、deleteInBatch()、deleteAllInBatch()等方法。它的实现类和子类，又提供了一些非常优雅的方法，结合第2章的UML图看其关联关系。本章从三方面介绍一下：

（1）QueryByExampleExecutor

（2）JpaSpecificationExecutor

（3）自定义Response

## 6.2 QueryByExampleExecutor的使用

按示例查询（QBE）是一种用户友好的查询技术，具有简单的接口。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。从UML图中，可以看出继承JpaRepository接口后，自动拥有了按“实例”进行查询的诸多方法。可见Spring Data的团队已经认为了QBE是Spring JPA的基本功能了，继承QueryByExampleExecutor和继承JpaRepository都会有这些基本方法。

### 6.2.1 QueryByExampleExecutor详细配置

QueryByExampleExecutor的源码：

```
public interface QueryByExampleExecutor<T> {
 //根据“实例”查找一个对象。
<S extends T> S findOne(Example<S> example);
//根据“实例”查找一批对象
<S extends T> Iterable<S> findAll(Example<S> example);
//根据“实例”查找一批对象，且排序
<S extends T> Iterable<S> findAll(Example<S> example, Sort sort);
 //根据“实例”查找一批对象，且排序和分页
<S extends T> Page<S> findAll(Example<S> example, Pageable pageable);
//根据“实例”查找，返回符合条件的对象个数
<S extends T> long count(Example<S> example);
//根据“实例”判断是否有符合条件的对象
<S extends T> boolean exists(Example<S> example);
}
```

所以我们看Example基本上就可以掌握的它的用法和API了。

```
public class Example<T> {
    @NonNull
    private final T probe;
    @NonNull
    private final ExampleMatcher matcher;
public static <T> Example<T> of(T probe) {
    return new Example(probe, ExampleMatcher.matching());
}
public static <T> Example<T> of(T probe, ExampleMatcher matcher) {
    return new Example(probe, matcher);
}
......
}
```

我们从源码中可以看出Example主要包含三部分内容：

- Probe：这是具有填充字段的域对象的实际实体类，即查询条的封装类。必填。
- ExampleMatcher：ExampleMatcher有关于如何匹配特定字段的匹配规则，它可以重复使用在多个示例。必填。如果不填，用默认的。
- Example：Example由探针和ExampleMatcher组成。它用于创建查询。

## 6.2.2 QueryByExampleExecutor的使用示例

```
    //创建查询条件数据对象
    Customer customer = new Customer();
    customer.setName("Jack");
    customer.setAddress("上海");

    //创建匹配器，即如何使用查询条件
    ExampleMatcher matcher = ExampleMatcher.matching() //构建对象
            .withMatcher("name", GenericPropertyMatchers.startsWith()) //姓名采用
"开始匹配"的方式查询
            .withIgnorePaths("focus");  //忽略属性：是否关注。因为是基本类型，需要忽略掉
    //创建实例
    Example<Customer> ex = Example.of(customer, matcher);

    //查询
    List<Customer> ls = dao.findAll(ex);

    //输出结果
    for (Customer bo:ls)
    {
        System.out.println(bo.getName());
    }
```

上面例子中，是这样创建“实例”的：Example<Customer> ex = Example.of(customer, matcher);我们看到，Example对象，由customer和matcher共同创建，为讲解方便，我们先来明确一些定义。

（1）Probe：实体对象，在持久化框架中与Table对应的域对象，一个对象代表数据库表中的一条记录，如上例中Customer对象。在构建查询条件时，一个实体对象代表的是查询条件中的“数值”部分。如：要查询姓“Jack”的客户，实体对象只能存储条件值“Jack”。

（2）ExampleMatcher：匹配器，它是匹配“实体对象”的，表示了如何使用“实体对象”中的“值”进行查询，它代表的是“查询方式”，解释了如何去查的问题。如，要查询姓“刘”的客户，即姓名以“刘”开头的客户，该对象就表示了“以某某开头的”这个查询方式，如上例中：withMatcher("name", GenericPropertyMatchers.startsWith())。

（3）Example：实例对象，代表的是完整的查询条件。由实体对

象（查询条件值）和匹配器（查询方式）共同创建。

再来理解“实例查询”，顾名思义，就是通过一个例子来查询。要查询的是Customer对象，查询条件也是一个Customer对象，通过一个现有的客户对象作为例子，查询和这个例子相匹配的对象。

### 6.2.3 QueryByExampleExecutor的特点及约束

（1）支持动态查询。即支持查询条件个数不固定的情况，如：客户列表中有多个过滤条件，用户使用时在“地址”查询框中输入了值，就需要按地址进行过滤，如果没有输入值，就忽略这个过滤条件。对应的实现是，在构建查询条件Customer对象时，将address属性值设置为具体的条件值或设置为null。

（2）不支持过滤条件分组。即不支持过滤条件用or（或）来连接，所有的过滤查件，都是简单一层的用and（并且）连接。如firstname = ?0 or (firstname = ?1 and lastname = ?2)。

（3）仅支持字符串的开始/包含/结束/正则表达式匹配和其他属性类型的精确匹配。查询时，对一个要进行匹配的属性（如：姓名name），只能传入一个过滤条件值，如以Customer为例，要查询姓“刘”的客户，“刘”这个条件值就存储在表示条件对象的Customer对象的name属性中，针对于“姓名”的过滤也只有这么一个存储过滤值的位置，没办法同时传入两个过滤值。正是由于这个限制，有些查询是没办法支持的，例如要查询某个时间段内添加的客户，对应的属性是addTime，需要传入“开始时间”和“结束时间”两个条件值，而这种查询方式没有存两个值的位置，所以就没办法完成这样的查询。

### 6.2.4 ExampleMatcher详解

#### 1．源码解读

```
   public class ExampleMatcher {
   NullHandler nullHandler;
   StringMatcher defaultStringMatcher; //默认
   boolean defaultIgnoreCase; //默认大小写忽略方式
   PropertySpecifiers propertySpecifiers; //各属性特定查询方式
   Set<String> ignoredPaths; //忽略属性列表
   //Null 值处理方式，通过构造方法，我们发现默认忽略
      NullHandler nullHandler;
       //字符串匹配方式,通过构造方法可以看出默认是 DEFAULT（默认，效果同 EXACT）,EXACT（相等）
      StringMatcher defaultStringMatcher;
       //各属性特定查询方式，默认无特殊指定的
      PropertySpecifiers propertySpecifiers;
       //忽略属性列表，默认无
      Set<String> ignoredPaths;
       //大小写忽略方式,默认不忽略
      boolean defaultIgnoreCase;
      @Wither(AccessLevel.PRIVATE) MatchMode mode;
   //通用、内部、默认构造方法
      private ExampleMatcher() {
         this(NullHandler.IGNORE, StringMatcher.DEFAULT, new PropertySpecifiers(),
Collections.<String>emptySet(), false,
               MatchMode.ALL);
      }
       //Example 的默认匹配方式
      public static ExampleMatcher matching() {
         return matchingAll();
      }
   public static ExampleMatcher matchingAll() {
      return new ExampleMatcher().withMode(MatchMode.ALL);
   }
   ......
   }
```

## 2．关键属性分析

（1）nullHandler：Null值处理方式，枚举类型，有2个可选值：

- INCLUDE（包括）
- IGNORE（忽略）

标识作为条件的实体对象中，一个属性值（条件值）为Null时，表示是否参与过滤。当该选项值是INCLUDE时，表示仍参与过滤，会

匹配数据库表中该字段值是Null的记录；若为IGNORE值，表示不参与过滤。

（2）defaultStringMatcher：默认字符串匹配方式，枚举类型，有6个可选值：

- DEFAULT（默认，效果同EXACT）
- EXACT（相等）
- STARTING（开始匹配）
- ENDING（结束匹配）
- CONTAINING（包含，模糊匹配）
- REGEX（正则表达式）

该配置对所有字符串属性过滤有效，除非该属性在propertySpecifiers中单独定义自己的匹配方式。

（3）defaultIgnoreCase：默认大小写忽略方式，布尔型，当值为false时，即不忽略，大小不相等。该配置对所有字符串属性过滤有效，除非该属性在propertySpecifiers中单独定义自己的忽略大小写方式。

（4）propertySpecifiers：各属性特定查询方式，描述了各个属性单独定义的查询方式，每个查询方式中包含4个元素：属性名、字符串匹配方式、大小写忽略方式、属性转换器。如果属性未单独定义查询方式，或单独查询方式中，某个元素未定义（如：字符串匹配方式），则采用ExampleMatcher中定义的默认值，即上面介绍的defaultStringMatcher和defaultIgnoreCase的值。

（5）ignoredPaths：忽略属性列表，忽略的属性不参与查询过滤。

## 3．字符串匹配举例

字符串匹配举例如表6-1所示。

表6-1　字符串匹配举例

| 字符串匹配方式 | 对应 JPQL 的写法 |
| --- | --- |
| Default&不忽略大小写 | firstname=?1 |
| Exact&忽略大小写 | LOWER(firstname) = LOWER(?1) |
| Staring&忽略大小写 | LOWER(firstname) like LOWER(?0)+'%' |
| Ending&不忽略大小写 | firstname like '%'+?1 |
| Containing 不忽略大小写 | firstname like '%'+?1+'%' |

### 6.2.5　QueryByExampleExecutor使用场景&实际的使用

#### 1．使用场景

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、简单的查询的使用场景，有时候还是挺方便的。

#### 2．实际使用中我们需要考虑的因素

查询条件的表示有两部分：一是条件值，二是查询方式。条件值用实体对象（如Customer对象）来存储，相对简单，当页面传入过滤条件值时，存入相对应的属性中，没入传入时，属性保持默认值。查询方式是用匹配器ExampleMatcher来表示，情况相对复杂些，需要考虑的因素有：

（1）Null值的处理。当某个条件值为Null时，是应当忽略这个过滤条件呢，还是应当去匹配数据库表中该字段值是Null的记录？

Null值处理方式：默认值是IGNORE（忽略），即当条件值为null时，则忽略此过滤条件，一般业务也是采用这种方式就可满足。当需

要查询数据库表中属性为null的记录时，可将值设为INCLUDE，这时，对于不需要参与查询的属性，都必须添加到忽略列表（ignoredPaths）中，否则会出现查不到数据的情况。

（2）基本类型的处理。如客户Customer对象中的年龄age是int型的，当页面不传入条件值时，它默认是0，是有值的，那是否参与查询呢？

关于基本数据类型处理方式：实体对象中，避免使用基本数据类型，采用包装器类型。如果已经采用了基本类型，而这个属性查询时不需要进行过滤，则把它添加到忽略列表（ignoredPaths）中。

（3）忽略某些属性值。一个实体对象，有许多个属性，是否每个属性都参与过滤？是否可以忽略某些属性？

ignoredPaths：虽然某些字段里面有值或者设置了其他匹配规则，只要放在ignoredPaths中，就会忽略此字段的，不作为过滤条件。

（4）不同的过滤方式。同样是作为String值，可能“姓名”希望精确匹配，“地址”希望模糊匹配，如何做到？

默认配置和特殊配置混合使用：默认创建匹配器时，字符串采用的是精确匹配、不忽略大小写，可以通过操作方法改变这种默认匹配，以满足大多数查询条件的需要，如将“字符串匹配方式”改为CONTAINING（包含，模糊匹配），这是比较常用的情况。对于个别属性需要特定的查询方式，可以通过配置“属性特定查询方式”来满足要求，设置propertySpecifiers的值即可。

（5）大小写匹配。字符串匹配时，有时可能希望忽略大小写，有时则不忽略，如何做到？

defaultIgnoreCase：忽略大小写的生效与否，是依赖于数据库的。例如MySQL数据库中，默认创建表结构时，字段是已经忽略大小

写的，所以这个配置与否，都是忽略的。如果业务需要严格区分大小写，可以改变数据库表结构属性来实现。

## 3．实际使用示例说明

（1）无匹配器的情况

要求：查询地址是“河南省郑州市”，且重点关注的客户。

说明：使用默认匹配器就可以满足查询条件，则不需要创建匹配器。

```
//创建查询条件数据对象
        Customer customer = new Customer();
          customer.setAddress("河南省郑州市");
        customer.setFocus(true);
          //创建实例
        Example<Customer> ex = Example.of(customer);
          //查询
        List<Customer> ls = dao.findAll(ex);
```

（2）多种条件组合

要求：根据姓名、地址、备注进行模糊查询，忽略大小写，地址要求开始匹配。

说明：这是通用情况，主要演示改变默认字符串匹配方式、改变默认大小写忽略方式、属性特定查询方式配置、忽略属性列表配置。

```
    //创建查询条件数据对象
    Customer customer = new Customer();
    customer.setName("zhang");
     customer.setAddress("河南省");
    customer.setRemark("BB");
customer.setFocus(true);//虽然有值，但是不参与过滤条件

     //创建匹配器，即如何使用查询条件
     ExampleMatcher matcher = ExampleMatcher.matching() //构建对象
                    .withStringMatcher(StringMatcher.CONTAINING) //改变默认字符串匹配
方式：模糊查询
                    .withIgnoreCase(true) //改变默认大小写忽略方式：忽略大小写
                    .withMatcher("address", GenericPropertyMatchers.startsWith())
//地址采用"开始匹配"的方式查询
                    .withIgnorePaths("focus");  //忽略属性：是否关注。因为是基本类型，需要
忽略掉

     //创建实例
    Example<Customer> ex = Example.of(customer, matcher);

    //查询
    List<Customer> ls = dao.findAll(ex);
```

（3）多级查询

要求：查询所有潜在客户。

说明：主要演示多层级属性查询。

```
    //创建查询条件数据对象
    CustomerType type = new CustomerType();
    type.setCode("01"); //编号01代表潜在客户
    Customer customer = new Customer();
    customer.setCustomerType(type);
    //创建匹配器，即如何使用查询条件
    ExampleMatcher matcher = ExampleMatcher.matching() //构建对象
                        .withIgnorePaths("focus");  //忽略属性：是否关注。因为是基本类型，
需要忽略掉
    //创建实例
    Example<Customer> ex = Example.of(customer, matcher);
    //查询
    List<Customer> ls = dao.findAll(ex);
```

（4）查询Null值

要求：地址是null的客户。

说明：主要演示改变“Null值处理方式”。

```
    //创建查询条件数据对象
    Customer customer = new Customer();
    //创建匹配器，即如何使用查询条件
    ExampleMatcher matcher = ExampleMatcher.matching() //构建对象
          .withIncludeNullValues() //改变"Null 值处理方式": 包括
          .withIgnorePaths("id", "name", "sex", "age", "focus", "addTime",
"remark", "customerType");  //忽略其他属性
    //创建实例
    Example<Customer> ex = Example.of(customer, matcher);
    //查询
    List<Customer> ls = dao.findAll(ex);
```

（5）虽然我们工作中用得最多的还是“简单查询”（因为简单，所以...）和基于JPA Criteria的动态查询（可以满足所有需求，没有局限性）。但是QueryByExampleExecutor还是个非常不错的两种中间场景的查询处理手段，别人没有用，感觉是对其不熟悉，还是希望我们学习过QueryByExampleExecutor的开发者用起来的，会增加我们的开发效率。

## 6.2.6 QueryByExampleExecutor的原理

（1）我们通过开发工具，把关键的类添加到Diagram上面进行分析，如图6-2所示。

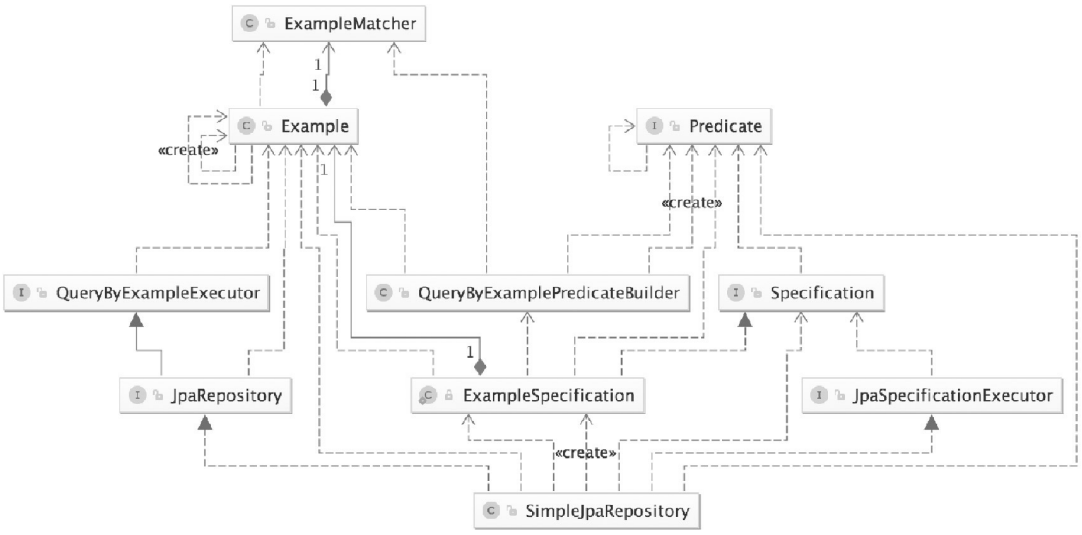



图6-2

（2）我们发现JpaSpecificationExecutor的实现类是SimpleJpaRepository。而SimpleJpaRepository也实现了JpaSpecificationExecutor，于是就利用Specification的特性，创建了内部类ExampleSpecification，通过Exmaple实现了一套工具类和对Predicate的构建，进而实现了整个ExampleQuery的逻辑。如果我们自己去看源码，QueryByExampleExecutor给我们提供了两种思路：通过JpaSpecificationExecutor自定义Response的思路和对JpaSpecificationExecutor的扩展思路。

（3）SimpleJpaRepository实现类中的关键源码：

```
   public class SimpleJpaRepository<T, ID extends Serializable>
         implements JpaRepository<T, ID>, JpaSpecificationExecutor<T> {
       private final EntityManager em;
   public <S extends T> S findOne(Example<S> example) {
      try {
        return getQuery(new ExampleSpecification<S>(example),
example.getProbeType(), (Sort) null).getSingleResult();
      } catch (NoResultException e) {
        return null;
      }
   }
   protected <S extends T> TypedQuery<S> getQuery(Specification<S> spec, Class<S>
domainClass, Sort sort) {
      CriteriaBuilder builder = em.getCriteriaBuilder();
      CriteriaQuery<S> query = builder.createQuery(domainClass);
      Root<S> root = applySpecificationToCriteria(spec, domainClass, query);
      query.select(root);
      if (sort != null) {
        query.orderBy(toOrders(sort, root, builder));
      }
      return applyRepositoryMethodMetadata(em.createQuery(query));
   }
   ......
   }
```

## 6.3 JpaSpecificationExecutor的详细使用

JpaSpecificationExecutor是JPA 2.0提供的Criteria API，可以用于动态生成query。Spring Data JPA支持Criteria查询，可以很方便地使用，足以应付工作中的所有复杂查询的情况了，可以对JPA

实现最大限度的扩展。

## 6.3.1 JpaSpecificationExecutor的使用方法

```
JpaSpecificationExecutor 源码和 API
public interface JpaSpecificationExecutor<T> {
   //根据 Specification 条件查询单个对象
   T findOne(Specification<T> spec);
   //根据 Specification 条件查询 List 结果
   List<T> findAll(Specification<T> spec);
   //根据 Specification 条件，分页查询
   Page<T> findAll(Specification<T> spec, Pageable pageable);
   //根据 Specification 条件，带排序地查询结果
   List<T> findAll(Specification<T> spec, Sort sort);
   //根据 Specification 条件，查询数量
   long count(Specification<T> spec);
}
```

这个接口基本是围绕着Specification接口来定义的，Specification接口中只定义了如下一个方法：

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

所以可看出，JpaSpecificationExecutor是针对Criteria API进行了predicate标准封装，帮我们封装了通过EntityManager的查询和使用细节，操作Criteria更加便利了一些。

## 6.3.2 Criteria概念的简单介绍

（1）Root<T>root：代表了可以查询和操作的实体对象的根。如果将实体对象比喻成表名，那root里面就是这张表里面的字段。这不过是JPQL的实体字段而已。通过里面的Path<Y>get(String attributeName)来获得我们操作的字段。

（2）CriteriaQuery<?>query：代表一个specific的顶层查询对象，它包含着查询的各个部分，比如：select、from、where、group

by、order by等。CriteriaQuery对象只对实体类型或嵌入式类型的Criteria查询起作用，简单理解，它提供了查询ROOT的方法。常用的方法有：

```
CriteriaQuery<T> where(Predicate... restrictions);
CriteriaQuery<T> select(Selection<? extends T> selection);
CriteriaQuery<T> having(Predicate... restrictions);
```

（3）CriteriaBuilder cb：用来构建CritiaQuery的构建器对象，其实就相当于条件或者是条件组合，以谓语即Predicate的形式返回。构建简单的Predicate示例：

```
Predicate p1=cb.like(root.get(“name”).as(String.class),
“%”+uqm.getName()+“%”);
Predicate p2=cb.equal(root.get("uuid").as(Integer.class), uqm.getUuid());
Predicate p3=cb.gt(root.get("age").as(Integer.class), uqm.getAge());
```

构建组合的Predicate示例：

```
Predicate p = cb.and(p3,cb.or(p1,p2));
```

（4）实际经验：到此我们发现其实JpaSpecificationExecutor帮我们提供了一个高级的入口和结构，通过这个入口，可以使用底层JPA的Criteria的所有方法，其实就可以满足了所有业务场景。但实际工作中，需要注意的是，如果一旦我们写的实现逻辑太复杂，第二个人一般看不懂的时候，那一定是有问题的，我们要寻找更简单的，更易懂的，更优雅的方式。比如：

- 分页和排序我们就没有自己再去实现一遍逻辑，直接用其开放的Pageable和Sort即可。
- 当过多地使用group或者having、sum、count等内置的SQL函数的时候，我们想想就是我们通过Specification实现了逻辑，这种效率真的高吗？是不是数据的其他算法更好？
- 当我们过多地操作left join和inner Join链表查询的时候，我们想想，是不是通过数据库的视图（view）更优雅一点？

### 6.3.3 JpaSpecificationExecutor示例

（1）新建两个实体。

```
   @Entity(name = "UserInfoEntity")
   @Table(name = "user_info", schema = "test")
   public class UserInfoEntity  implements Serializable {
      @Id
      @Column(name = "id", nullable = false)
      private Integer id;
      @Column(name = "first_name", nullable = true, length = 100)
      private String firstName;
      @Column(name = "last_name", nullable = true, length = 100)
      private String lastName;
      @Column(name = "telephone", nullable = true, length = 100)
      private String telephone;
      @Column(name = "create_time", nullable = true)
      private Date createTime;
      @Column(name = "version", nullable = true)
      private String version;
      @OneToOne(optional = false,fetch = FetchType.EAGER)
      @JoinColumn(referencedColumnName = "id",name = "address_id",nullable = false)
      @Fetch(FetchMode.JOIN)
      private UserReceivingAddressEntity addressEntity;
   ......
   }
   @Entity
   @Table(name = "user_receiving_address", schema = "test")
   public class UserReceivingAddressEntity  implements Serializable {
      @Id
      @Column(name = "id", nullable = false)
      private Integer id;
      @Column(name = "user_id", nullable = false)
      private Integer userId;
      @Column(name = "address_city", nullable = true, length = 500)
      private String addressCity;
   ......
   }
```

（2）UserRepository需要继承JpaSpecificationExecutor。

```
   public interface UserRepository extends
JpaSpecificationExecutor<UserInfoEntity> {
   }
```

（3）调用者UserInfoManager的写法。

```
    @Component
    public class UserInfoManager {
       @Autowired
       private UserRepository userRepository;
       public Page<UserInfoEntity> findByCondition(UserInfoRequest
userParam,Pageable pageable){
          return userRepository.findAll((root, query, cb) -> {
             List<Predicate> predicates = new ArrayList<Predicate>();
             if (StringUtils.isNoneBlank(userParam.getFirstName())){
                  //liked 的查询条件
    predicates.add(cb.like(root.get("firstName"),"%"+userParam.getFirstName()+"
%"));
             }
             if (StringUtils.isNoneBlank(userParam.getTelephone())){
                  //equal 查询条件
    predicates.add(cb.equal(root.get("telephone"),userParam.getTelephone()));
             }
             if (StringUtils.isNoneBlank(userParam.getVersion())){
                  //greaterThan 大于等于查询条件
    predicates.add(cb.greaterThan(root.get("version"),userParam.getVersion()));
             }
             if
(userParam.getBeginCreateTime()!=null&&userParam.getEndCreateTime()!=null){
                  //根据时间区间去查询
predicates.add(cb.between(root.get("createTime"),userParam.getBeginCreateTime(
),userParam.getEndCreateTime()));
             }
             if (StringUtils.isNotBlank(userParam.getAddressCity())) {
                  //联表查询，利用 root 的 join 方法，根据关联关系表里面的字段进行查询。
    predicates.add(cb.equal(root.join("addressEntityList").get("addressCity"),
userParam.getAddressCity()));
             }
             return query.where(predicates.toArray(new
Predicate[predicates.size()])).getRestriction();
          }, pageable);
       }
    }
```

可以仔细体会一下示例，实际工作中应该大部分都是这种写法，就算扩展也是百变不离其宗。

### 6.3.4 Specification工作中的一些扩展

我们在实际工作中会发现，如果按上面的逻辑，简单重复，总感觉是不是可以抽出一些公用方法呢，此时我们引入一种工厂模式，帮我们做一些事情。基于JpaSpecificationExecutor的思路，我们创建

一个SpecificationFactory.Java，内容如下：

```
   public final class SpecificationFactory {
      /**
       * 模糊查询，匹配对应字段
       */
      public static Specification containsLike(String attribute, String value) {
         return (root, query, cb)-> cb.like(root.get(attribute), "%" + value + "%");
      }
      /**
       * 某字段的值等于 value 的查询条件
       */
      public static Specification equal(String attribute, Object value) {
         return (root, query, cb) -> cb.equal(root.get(attribute),value);
      }
      /**
       * 获取对应属性的值所在区间
       */
      public static Specification isBetween(String attribute, int min, int max) {
         return (root, query, cb) -> cb.between(root.get(attribute), min, max);
      }
      public static Specification isBetween(String attribute, double min, double
max) {
         return (root, query, cb) -> cb.between(root.get(attribute), min, max);
      }
      public static Specification isBetween(String attribute, Date min, Date max) {
         return (root, query, cb) -> cb.between(root.get(attribute), min, max);
      }
      /**
       * 通过属性名和集合实现 in 查询
       */
      public static Specification in(String attribute, Collection c) {
         return (root, query, cb) ->root.get(attribute).in(c);
      }
      /**
       * 通过属性名构建大于等于 Value 的查询条件
       */
      public static Specification greaterThan(String attribute, BigDecimal value)
{
         return (root, query, cb) ->cb.greaterThan(root.get(attribute),value);
      }
      public static Specification greaterThan(String attribute, Long value) {
         return (root, query, cb) ->cb.greaterThan(root.get(attribute),value);
      }
   ......
   }
```

可以根据实际工作需要和场景进行不断扩充。

调用示例1：

```
   userRepository.findAll(
         SpecificationFactory.containsLike("firstName",
userParam.getLastName()),
         pageable);
```

配合Specifications使用，调用示例2：

```
   userRepository.findAll(Specifications.where(
        SpecificationFactory.containsLike("firstName",
userParam.getLastName()))
             .and(SpecificationFactory.greaterThan("version",userParam.getVers
ion())), pageable);
```

Specifications是Spring Data JPA对Specification的聚合操作工具类，里面有以下4个方法：

```
   public class Specifications<T> implements Specification<T>, Serializable {
      private final Specification<T> spec;
       //构造方法私有化，只能通过 where/not 创建 Specifications 对象。
      private Specifications(Specification<T> spec) {
         this.spec = spec;
      }
       //创建 where 后面的 Predicate 集合
      public static <T> Specifications<T> where(Specification<T> spec) {
         return new Specifications<T>(spec);
      }
       //创建 not 集合的 Predicate
      public static <T> Specifications<T> not(Specification<T> spec) {
         return new Specifications<T>(new NegatedSpecification<T>(spec));
      }
       //Specification 的 and 关系集合
      public Specifications<T> and(Specification<T> other) {
         return new Specifications<T>(new ComposedSpecification<T>(spec, other,
AND));
      }
       //Specification 的 or 关系集合
      public Specifications<T> or(Specification<T> other) {
         return new Specifications<T>(new ComposedSpecification<T>(spec, other,
OR));
      }
    ......
    }
```

## 6.3.5 JpaSpecificationExecutor实现原理

（1）我们还是先通过开发工具，把关键的类添加到Diagram上面进行分析，如图6-3所示。

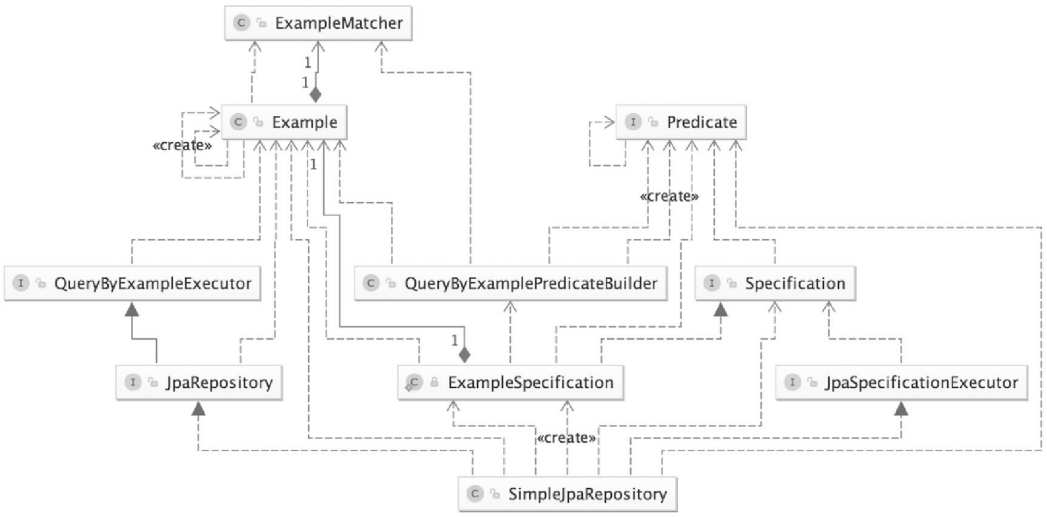



图6-3

（2）SimpleJpaRepository实现类中的关键源码：

```
    /**
    *以 findOne 为例
    */
    public T findOne(Specification<T> spec) {
       try {
          return getQuery(spec, (Sort) null).getSingleResult();
       } catch (NoResultException e) {
          return null;
       }
    }
    /*
    * 解析 Specification，利用 EntityManager 直接实现调用逻辑。
    */
    protected <S extends T> TypedQuery<S> getQuery(Specification<S> spec, Class<S>
domainClass, Sort sort) {
       CriteriaBuilder builder = em.getCriteriaBuilder();
       CriteriaQuery<S> query = builder.createQuery(domainClass);
       Root<S> root = applySpecificationToCriteria(spec, domainClass, query);
       query.select(root);
       if (sort != null) {
          query.orderBy(toOrders(sort, root, builder));
       }
       return applyRepositoryMethodMetadata(em.createQuery(query));
    }
```

# 6.4 自定义Repository

由于业务场景的千差万别，有可能需要定义自己的Repository类，其实通过上面的章节，我们也大概能想到了，Spring Data JPA可以轻松地允许你提供自定义Repository，并且还很容易与现有的抽

象和查询方法方法集成。作者将其概括为两种方式：

（1）继承JpaSpecificationExecutor通过CriteriaQuery几乎可以实现任何逻辑了。

（2）自己写独立接口继承CrudRepository，独立实现类利用EntityManager操作Criteria Query API可以实现任何逻辑。

### 6.4.1 EntityManager介绍

对EntityManager我们没必要研究那么深入，知道它主要给我们提供以下几个方法即可：

```
public interface EntityManager {
  /**
*根据主键查询实体对象
*/
  public <T> T find(Class<T> entityClass, Object primaryKey);
    /**
      *  支持 JQPL 的语法
     * @param qlString a Java Persistence query string
     */
    public Query createQuery(String qlString);
    /**
      * 利用 CriteriaQuery 来创建查询
     * @param criteriaQuery  a criteria query object
     */
    public <T> TypedQuery<T> createQuery(CriteriaQuery<T> criteriaQuery);
  /**
    * 利用 CriteriaUpdate 创建更新查询
   * @param updateQuery a criteria update query object
   */
  public Query createQuery(CriteriaUpdate updateQuery);

  /**
    * 利用 CriteriaDelete 创建删除查询
   * @param deleteQuery a criteria delete query object
   */
  public Query createQuery(CriteriaDelete deleteQuery);
    /**
      * 利用原生的 SQL 语句创建查询，可以是查询、更新、删除等
     * @param sqlString a native SQL query string
     */
    public Query createNativeQuery(String sqlString);
    /**
      * 利用原生 SQL 查询，指定返回结果类型
     * @param sqlString a native SQL query string
     * @param resultClass the class of the resulting instance(s)
     */
    public Query createNativeQuery(String sqlString, Class resultClass);
......
}
```

【示例6.1】针对复杂的原生SQL的查询。

```
    //创建 SQL 语句
    StringBuilder querySQL = new StringBuilder("SELECT spu_id AS spuId ,spu_name
AS spuName,")
            .append("SUM(system_price_count) AS systemPriceCount,")
            .append("SUM(wechat_applet_view_count) AS wechatAppletViewCount")
            .append(" FROM report_spu_summary ");
    //利用 entityManager 实现查询
    Query query = entityManager.createNativeQuery(querySQL.toString() +
whereSQL.toString() + groupBy + orderBy.toString());
    //分页
    query.setFirstResult(custom.offset()).setMaxResults(custom.getPageSize());
    //结果转换
    query.unwrap(SQLQuery.class).setResultTransformer(Transformers.aliasToBean(
ReportSpuSummarySumBo.class));
    //得到最终的返回结果
    List<ReportSpuSummarySumBo> results = query.getResultList();
    //此示例仅仅为了说明 entityManager.createNativeQuery 的查询方法,但是不推荐用这种用法,
开发思路可转换一下。
```

【示例6.2】find方法

```
    entityManager.find(UserInfoEntity.class,1);
```

【示例6.3】JPQL的用法。

```
    Query query = entityManager.createQuery("SELECT c FROM Customer c");
    List<Customer> result = query.getResultList();
```

## 6.4.2 自定义实现Repository

我们自定义实现Repository，主要的应用场景有两种：

- 单个的接口特殊化的实现方式，私有的。
- 公用的通用的场景的自定义方式。

而其中离不开EntityManager，我们下面讲解它的两种获得的方式。

- 通过@PersistenceContext注解。通过将@PersistenceContext注解标注在EntityManager类型的字段上，这样得到的

EntityManager就是容器管理的EntityManager。由于是容器管理的，所以我们不需要也不应该显式关闭注入的EntityManager实例。

- 显现@Repository的子类的任何接口，通过构造方法获得。

【示例6.4】单个私有的Repository接口的实现类中@PersistenceContext的用法。

（1）自定义存储库功能的接口。

```
interface UserRepositoryCustom {
  public User someCustomMethodFindById(Integer id);
}
```

（2）自定义存储库功能的实现。

```
public class UserRepositoryImpl implements UserRepositoryCustom {
@PersistenceContext  //获得 entityManager 的上下文
EntityManager entityManager;
  public User someCustomMethod(Integer id) {
    //你的自定义 Repository 的实现方法，根据 id 查询 User
return entityManager.find(User.class,id);
  }
}
```

（3）当然也不排除，自己通过最底层的JdbcTemplate来实现逻辑。

（4）这个接口是为User单独写的，但是同时也可以继承和@Repository的任何子类。

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

（5）调用的地方，直接调用UserRepository即可。

【示例6.5】定义一个公用的Repository接口的实现类，通过构

造方法获得EntityManager，需要用到Java的泛化技术。当你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，你首先添加一个中间接口来声明共享行为。

（1）声明定制共享行为的接口：@NoRepositoryBean。

```
@NoRepositoryBean
public interface MyRepository<T, ID extends Serializable>
extends PagingAndSortingRepository<T, ID> {
  void sharedCustomMethod(ID id);
}
```

（2）现在，你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。

```
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) {
    // 和上面一样，在此方法里面，可以通过 entityManager 实现自己的额外方法的实现逻辑。
……
  }
}
```

## 提示

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）。

（3）使用JavaConfig配置自定义MyRepositoryImpl作为其他接口的动态代理的实现基类。具有全局的性质，即使没有继承它所有的动态代理类也会变成它。

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

## 6.4.3 实际工作的应用场景

在实际工作中，有哪些场景会用到自定义Repository呢，给大家列出来几种作者在实际工作中的应用案例，扩展一下我们的思维。

### 1．逻辑删除场景

可以用到上面的两种实现方式，如果有框架级别的全局自定义Respository，那就在全局实现里面覆盖默认remove方法，这样就会统一全部只能使用逻辑删除。但是一般是自定义一个特殊的删除Respository，让大家去根据不同的domain业务逻辑去选择使用此接口即可。

### 2．当有业务场景要覆盖SimpleJpaRepository默认实现的时候

这种一般是具体情况具体分析，实现特殊化的自定义Respository即可。

### 3．UUID与ID的情况

经常在实际生产中会有这样的场景，对外暴露的是UUID查询方法，而对内暴露的是Long类型的ID，这时候我们就可以自定义一个FindByIDOrUUID的底层实现方法，在自定义的Respository接口里

面。

## 4．使用Querydsl

Spring Data JPA里面还帮我们做了QuerydslJpaRepository，用来支持Querydsl的查询方法。当我们引入Querydsl的时候，Spring就会自动帮我们把SimpleJpaRepository的实现切换到QuerydslJpaRepository的实现。

## 5．动态查询条件

由于Spring Data JPA里面的query method或者@query注解不支持动态查询条件，我们正常情况下将动态条件写在manager或者service里面。这个时候如果是针对资源的操作，并且和业务无关的查询，可以放在自定义Repository里面（有个缺点就是不能使用SimpleJpaRepository里面的很多优秀的默认的实现方法，实际工作中还是放在service和manager中多一些，这里只是给大家举个例子，让大家知道有这么回事就行）。实例如下：

```
    //我们假设要根据条件动态查询订单
    public interface OrderRepositoryCustom {
        Page<Order> findAllByCriteria(OrderCriteria criteria); // 定义一个订单的定制化 Repository 查询方法，当然实际生产过程中，这里面可能不止一个方法。
    }
    public class OrderRepositoryImpl implements OrderRepositoryCustom {
        @PersistenceContext
        EntityManager entityManager;
        /**
        * 一个动态条件的查询方法
        */
        public List<Order> findAllByCriteria(OrderCriteria criteria) {
            // 查询条件列表
            final List<String> andConditions = new ArrayList<String>();
            final Map<String, Object> bindParameters = new HashMap<String, Object>();
            // 动态绑定参数和要查询的条件
            if (criteria.getId() != null) {
                andConditions.add("o.id = :id");
                bindParameters.put("id", criteria.getId());
            }
            if (!CollectionUtils.isEmpty(criteria.getStatusCodes())) {
                andConditions.add("o.status.code IN :statusCodes");
                bindParameters.put("statusCodes", criteria.getStatusCodes());
            }
            if (andConditions.isEmpty()) {
                return Collections.emptyList();
            }
            // 动态创建 query
```

```
        final StringBuilder queryString = new StringBuilder();
        queryString.append("SELECT o FROM Order o");
        // 动态拼装条件
        Iterator<String> andConditionsIt = andConditions.iterator();
        if (andConditionsIt.hasNext()) {
            queryString.append(" WHERE ").append(andConditionsIt.next());
        }
        while (andConditionsIt.hasNext()) {
            queryString.append(" AND ").append(andConditionsIt.next());
        }
        // 添加排序
        queryString.append(" ORDER BY o.id");
        // 创建 typed query.
        final TypedQuery<Order> findQuery = entityManager.createQuery(
                queryString.toString(), Order.class);
        // 绑定参数
        for (Map.Entry<String, Object> bindParameter : bindParameters
                .entrySet()) {
            findQuery.setParameter(bindParameter.getKey(), bindParameter
                    .getValue());
        }
        //返回查询，结果。
        return findQuery.getResultList();
    }
}
//实际中此种就比较少用了，大家知道有这么回事，真是遇到特殊场景必须要用了，可以用此方法实现。
```

# 6．扩展JpaSpecificationExecutor使其更加优雅

当我们动态查询的时候经常会出现下面的代码逻辑，写起来老是感觉有点不是特别优雅有点重复的感觉。

```
   PageRequest pr = new PageRequest(page - 1, rows, Direction.DESC, "id");
       Page pageData = memberDao.findAll(new Specification() {
          @Override
          public Predicate toPredicate(Root root, CriteriaQuery query,
CriteriaBuilder cb) {
             List<Predicate> predicates = new ArrayList<>();
             if (isNotEmpty(userName)) {
                predicates.add(cb.like(root.get("userName"), "%" + userName +
"%"));
             }
             if (isNotEmpty(realName)) {
                predicates.add(cb.like(root.get("realName"), "%" + realName +
"%"));
             }
             if (isNotEmpty(telephone)) {
                predicates.add(cb.equal(root.get("userName"), telephone));
             }
             query.where(predicates.toArray(new Predicate[0]));
             return null;
          }
       }, pr);
```

使用了自定义的复杂查询，我们可以做到如下效果：

```
   Page pageData = userDao.findAll(new MySpecification<User>().and(
          Cnd.like("userName", userName),
          Cnd.like("realName", realName),
          Cnd.eq("telephone", telephone)
   ).asc("id"), pr);
```

如果对Spring Mvc比较熟悉的话，可以更进一步把其查询提交和规则直接封装到HandlerMethodArgumentResolver里面，把参数自动和规则匹配起来。我们可以对如下代码进行参考，感觉实现的还不错，此段代码可以作为参考，只是实现的还有点不完整，如下所示：

```
/**
 * 扩展 Specification
 * @param <T>
 */
public class MySpecification<T> implements Specification<T> {
    /**
     * 属性分隔符
     */
    private static final String PROPERTY_SEPARATOR = ".";
    /**
     * and 条件组
     */
    List<Cnd> andConditions = new ArrayList<>();
    /**
     * or 条件组
     */
    List<Cnd> orConditions = new ArrayList<>();
    /**
     * 排序条件组
     */
    List<Order> orders = new ArrayList<>();
```

```
      @Override
      public Predicate toPredicate(Root<T> root, CriteriaQuery<?> cq,
CriteriaBuilder cb) {
         Predicate restrictions = cb.and(getAndPredicates(root, cb));
         restrictions = cb.and(restrictions, getOrPredicates(root, cb));
         cq.orderBy(getOrders(root, cb));
         return restrictions;
      }
      public MySpecification and(Cnd... conditions) {
         for (Cnd condition : conditions) {
            andConditions.add(condition);
         }
         return this;
      }
      public MySpecification or(Collection<Cnd> conditions) {
         orConditions.addAll(conditions);
         return this;
      }
      public MySpecification desc(String property) {
         this.orders.add(Order.desc(property));
         return this;
      }
      public MySpecification asc(String property) {
         this.orders.add(Order.asc(property));
         return this;
      }
      private Predicate getAndPredicates(Root<T> root, CriteriaBuilder cb) {
         Predicate restrictions = cb.conjunction();
         for (Cnd condition : andConditions) {
            if (condition == null) {
               continue;
            }
            Path<?> path = this.getPath(root, condition.property);
            if (path == null) {
               continue;
            }
            switch (condition.operator) {
               case eq:
                  if (condition.value != null) {
                     if (String.class.isAssignableFrom(path.getJavaType()) &&
condition.value instanceof String) {
                        if (!((String) condition.value).isEmpty()) {
                           restrictions = cb.and(restrictions, cb.equal(path,
condition.value));
```

```
                        }
                    } else {
                        restrictions = cb.and(restrictions, cb.equal(path,
condition.value));
                    }
                }
                break;
            case ge:
                if (Number.class.isAssignableFrom(path.getJavaType()) &&
condition.value instanceof Number) {
                    restrictions = cb.and(restrictions, cb.ge((Path<Number>)
path, (Number) condition.value));
                }
                break;
            case gt:
                if (Number.class.isAssignableFrom(path.getJavaType()) &&
condition.value instanceof Number) {
                    restrictions = cb.and(restrictions, cb.gt((Path<Number>)
path, (Number) condition.value));
                }
                break;
            case lt:
                if (Number.class.isAssignableFrom(path.getJavaType()) &&
condition.value instanceof Number) {
                    restrictions = cb.and(restrictions, cb.lt((Path<Number>)
path, (Number) condition.value));
                }
                break;
            case ne:
                if (condition.value != null) {
                    if (String.class.isAssignableFrom(path.getJavaType()) &&
condition.value instanceof String && !((String) condition.value).isEmpty()) {
                        restrictions = cb.and(restrictions, cb.notEqual(path,
condition.value));
                    } else {
                        restrictions = cb.and(restrictions, cb.notEqual(path,
condition.value));
                    }
                }
                break;
            case isNotNull:
                restrictions = cb.and(restrictions, path.isNotNull());
                break;
        }
```

```
            }
            return restrictions;
        }
        private Predicate getOrPredicates(Root<T> root, CriteriaBuilder cb) {
            // 相同的逻辑 Need TODO
            return null;
        }
        private List<javax.persistence.criteria.Order> getOrders(Root<T> root,
CriteriaBuilder cb) {
            List<javax.persistence.criteria.Order> orderList = new ArrayList<>();
            if (root == null || CollectionUtils.isEmpty(orders)) {
                return orderList;
            }
            for (Order order : orders) {
                if (order == null) {
                    continue;
                }
                String property = order.getProperty();
                Sort.Direction direction = order.getDirection();
                Path<?> path = this.getPath(root, property);
                if (path == null || direction == null) {
                    continue;
                }
                switch (direction) {
                    case ASC:
                        orderList.add(cb.asc(path));
                        break;
                    case DESC:
                        orderList.add(cb.desc(path));
                        break;
                }
            }
            return orderList;
        }
        /**
         * 获取 Path
         *
         * @param path        Path
         * @param propertyPath 属性路径
         * @return Path
         */
        private <X> Path<X> getPath(Path<?> path, String propertyPath) {
            if (path == null || StringUtils.isEmpty(propertyPath)) {
                return (Path<X>) path;
```

```
            }
            String property = StringUtils.substringBefore(propertyPath,
PROPERTY_SEPARATOR);
            return getPath(path.get(property),
StringUtils.substringAfter(propertyPath, PROPERTY_SEPARATOR));
        }
        /**
         * 条件
         */
        public static class Cnd {
            Operator operator;
            String property;
            Object value;
            public Cnd(String property, Operator operator, Object value) {
                this.operator = operator;
                this.property = property;
                this.value = value;
            }
            /**
             * 相等
             *
             * @param property
             * @param value
             * @return
             */
            public static Cnd eq(String property, Object value) {
                return new Cnd(property, Operator.eq, value);
            }
            /**
             * 不相等
             *
             * @param property
             * @param value
             * @return
             */
            public static Cnd ne(String property, Object value) {
                return new Cnd(property, Operator.ne, value);
            }
        }
        /**
         * 排序
         */
        @Getter
        @Setter
```

```
    public static class Order {
        private String property;
        private Sort.Direction direction = Sort.Direction.ASC;
        /**
         * 构造方法
         *
         * @param property  属性
         * @param direction 方向
         */
        public Order(String property, Sort.Direction direction) {
            this.property = property;
            this.direction = direction;
        }
        /**
         * 返回递增排序
         *
         * @param property 属性
         * @return 递增排序
         */
        public static Order asc(String property) {
            return new Order(property, Sort.Direction.ASC);
        }
        /**
         * 返回递减排序
         *
         * @param property 属性
         * @return 递减排序
         */
        public static Order desc(String property) {
            return new Order(property, Sort.Direction.DESC);
        }
    }
    /**
     * 运算符
     */
    @Getter
    @Setter
    public enum Operator {
        /**
         * 等于
         */
        eq(" = "),
        /**
         * 不等于
```

```
         */
        ne(" != "),
        /**
         * 大于
         */
        gt(" > "),
        /**
         * 小于
         */
        lt(" < "),
        /**
         * 大于等于
         */
        ge(" >= "),
        /**
         * 不为Null
         */
        isNotNull(" is not NULL ");
        Operator(String operator) {
           this.operator = operator;
        }
        private String operator;
    }
}
```

# 7．类似的RSQL解决方案

与之类似的解决方案还有RSQL解决方案，可以参考Github上的此开源项目。RSQL（RESTful Service Query Language）是Feed Item Query Language（FIQL）的超集，是一种RESTful服务的查询语言。这里我们使用rsql-jpa来实践，它依赖rsql-parser来解析RSQL语法，然后将解析后的RSQL转义到JPA的Specification。maven的地址如下：

```
<dependency>
   <groupId>com.github.tennaito</groupId>
   <artifactId>rsql-jpa</artifactId>
   <version>2.0.2</version>
</dependency>
```

Github文档地址:https://github.com/tennaito/rsql-jpa。如果要立志做一个优秀的架构师，Spring Data JPA的实现还是非常好

的，包括开源的生态等也非常好。

在实际运用中，极少有机会用到自定义扩展的动态代理基类。类图如图6-4所示。

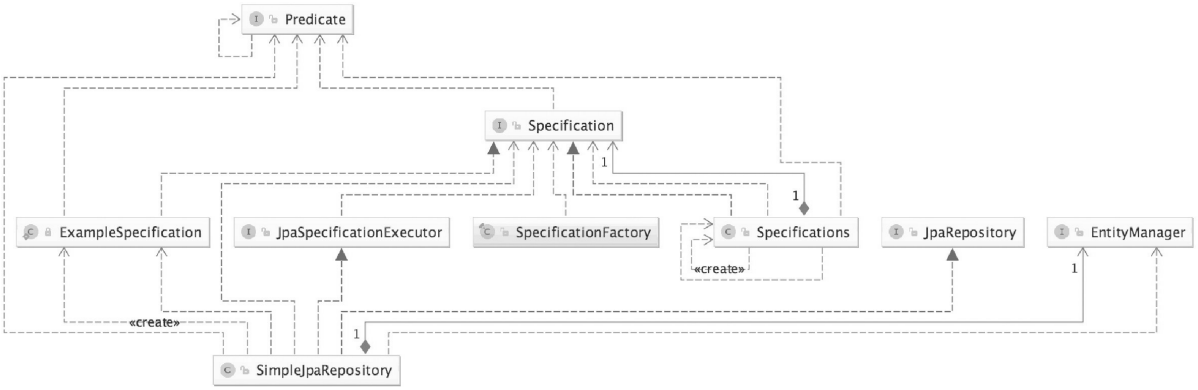



图6-4

# 第7章
# Spring Data JPA的扩展

静胜躁，寒胜热，清静为天下正！

——老子

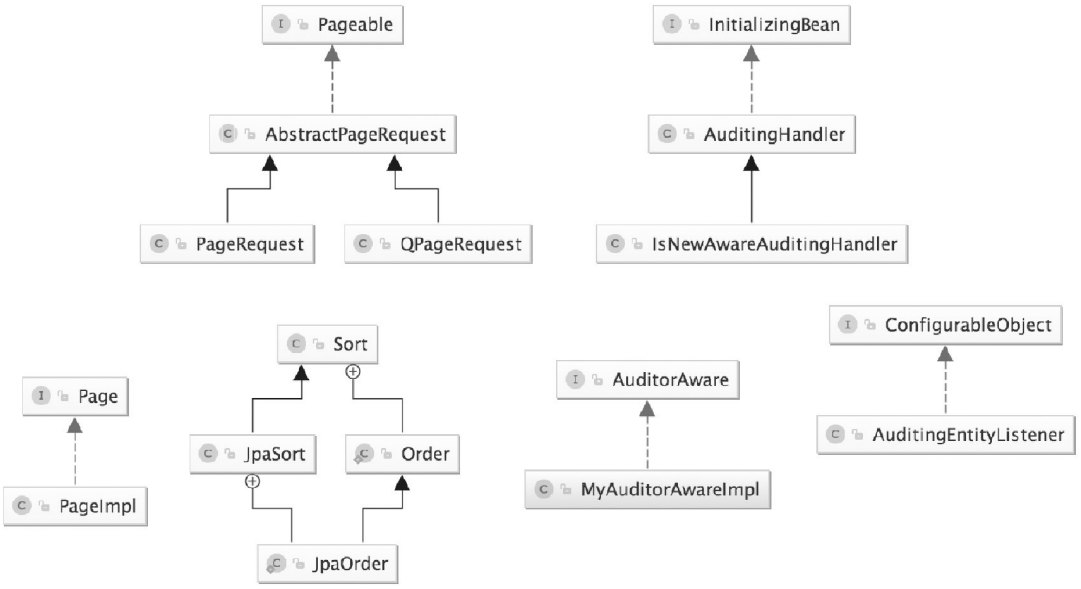



图7-1

本章主要介绍Spring Data JPA的一些扩展部分，也非常重要，有利于提高工作效率和代码的优雅度。

## 7.1 Auditing及其事件详解

Auditing翻译过来是审计和审核。Spring的优秀之处在于帮我们想到了很多我们平时烦琐事情的解决方案，我们在实际的业务系统中，针对一张表的操作大部分是需要记录谁什么时间创建的，谁什么时间修改的，并且能让我们方便地记录操作日志。Spring Data JPA为我们提供了审计功能的架构实现，提供了4个注解专门解决这件事

情：

- @CreatedBy：哪个用户创建的。
- @CreatedDate：创建的时间。
- @LastModifiedBy：修改实体的用户。
- @LastModifiedDate：最后一次修改时间。

### 7.1.1 Auditing如何配置

#### 1．UserCustomerEntity里面的写法

```
@Entity
@Table(name = "user_customer", schema = "test", catalog = "")
@EntityListeners(AuditingEntityListener.class)
public class UserCustomerEntity {
   @Id
   @Column(name = "id", nullable = false)
    @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Integer id;
   @CreatedDate
   @Column(name = "create_time", nullable = true)
   private Date createTime;
   @CreatedBy
   @Column(name = "create_user_id", nullable = true)
   private Integer createUserId;
   @LastModifiedBy
   @Column(name = "last_modified_user_id", nullable = true)
   private Integer lastModifiedUserId;
   @LastModifiedDate
   @Column(name = "last_modified_time", nullable = true)
   private Date lastModifiedTime;
   @Column(name = "customer_name", nullable = true, length = 50)
   private String customerName;
   @Column(name = "customer_email", nullable = true, length = 50)
   private String customerEmail;
......
}
```

@Entity实体中我们需要做两点：

（1）相应的字段添加。

```
   @CreatedBy,
   @CreatedDate,
   @LastModifiedBy,
   @LastModifiedDate 注解。
```

（2）增加@EntityListeners(AuditingEntityListener.class)。

## 2．实现AuditorAware接口告诉JPA当前的用户是谁

```
   public class MyAuditorAware implements AuditorAware<Integer> {
      /**
       * Returns the current auditor of the application.
       * @return the current auditor
       */
      @Override
      public Integer getCurrentAuditor() {
   //     第一种方式：如果我们集成了 spring 的 Security，我们直接通过如下方法即可获得当前请求的用户 ID.
   //    Authentication authentication =
       SecurityContextHolder.getContext().getAuthentication();
   //    if (authentication == null || !authentication.isAuthenticated()) {
   //       return null;
   //    }
   //    return ((LoginUserInfo) authentication.getPrincipal()).getUser().getId();

          //第二种方式通过 request 里面取或者 session 里面取
         ServletRequestAttributes servletRequestAttributes =
       (ServletRequestAttributes)RequestContextHolder.getRequestAttributes();
         return (Integer) servletRequestAttributes.getRequest().getSession().getAttribute("userId");
      }
   }
```

而AuditorAware的源码如下：

```
   public interface AuditorAware<T> {
      T getCurrentAuditor();
   }
```

通过实现AuditorAware接口的getCurrentAuditor()方法告诉JPA当前的用户是谁。里面实现方法千差万别，作者列举了两种最常见的

方法：

- 通过Security取。
- 通过Request取。

### 3．通过@EnableJpaAuditing注解开启JPA的Auditing功能

通过@EnableJpaAuditing注解开启JPA的Auditing功能，并且告诉应用AuditorAware的实现类是谁。

具体配置方式如下：

```
@SpringBootApplication
@EnableJpaAuditing
public class QuickStartApplication {
   public static void main(String[] args) {
      SpringApplication.run(QuickStartApplication.class, args);
   }
   @Bean
   public AuditorAware<Integer> auditorProvider() {
      return new MyAuditorAwareImpl();
   }
}
```

## 4．通过以上的三步，我们已经完成了auting的配置

通过执行下面语句：

```
userCustomerRepository.save(new UserCustomerEntity("1","Jack"));
```

数据库里面的4个字段已经填上去了。

## 7.1.2 @MappedSuperclass

实际工作中我们还会对上面的实体部分进行改进，引入@MappedSuperclass注解，我们将@Id、@CreatedBy、@CreatedDate、

@LastModifiedBy与@LastModifiedDate抽象到一个公用的基类里面，方便公用和形成每个表的字段约束。

（1）改进后我们新增一个AbstractAuditable的抽象类：

```
@MappedSuperclass
@EntityListeners(AuditingEntityListener.class)
public abstract class AbstractAuditable {
   @Id
   @Column(name = "id", nullable = false)
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Integer id;
   @CreatedDate
   @Column(name = "create_time", nullable = true)
   private Date createTime;
   @CreatedBy
   @Column(name = "create_user_id", nullable = true)
   private Integer createUserId;
   @LastModifiedBy
   @Column(name = "last_modified_user_id", nullable = true)
   private Integer lastModifiedUserId;
   @LastModifiedDate
   @Column(name = "last_modified_time", nullable = true)
   private Date lastModifiedTime;
......
}
```

（2）我们每个需要Auditing的实体只需要继承AbstractAuditable即可，内容如下：

```
@Entity
@Table(name = "user_customer", schema = "test", catalog = "")
public class UserCustomerEntity extends AbstractAuditable {
   @Column(name = "customer_name", nullable = true, length = 50)
   private String customerName;
   @Column(name = "customer_email", nullable = true, length = 50)
   private String customerEmail;
......}
```

## 7.1.3 Auditing原理解析

（1）我们先看一下关键的几个源码的关系图，如图7-2所示。

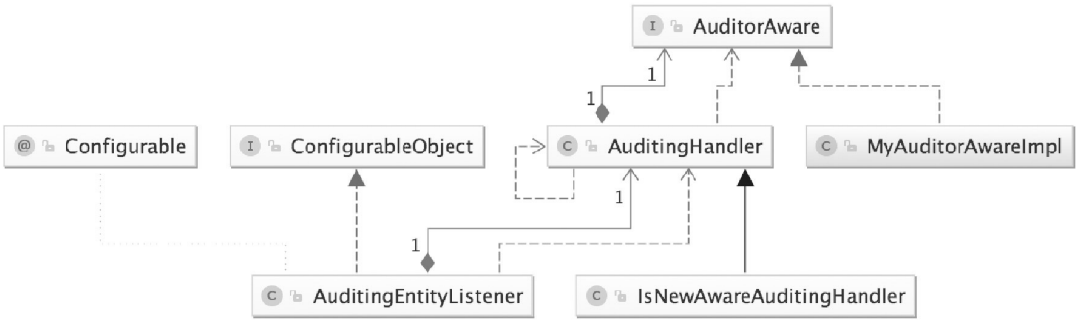



图7-2

（2）AuditingEntityListener的源码如下：

```
   @Configurable
   public class AuditingEntityListener {
      private ObjectFactory<AuditingHandler> handler;
      public void setAuditingHandler(ObjectFactory<AuditingHandler>
auditingHandler) {
         Assert.notNull(auditingHandler, "AuditingHandler must not be null!");
         this.handler = auditingHandler;
      }
       //在新增之前通过 handler 来往我们的@Entity 里面的 auditor 的那些字段塞值
      @PrePersist
      public void touchForCreate(Object target) {
         if (handler != null) {
            handler.getObject().markCreated(target);
         }
      }
      //在更新之前通过 handler 来往我们的@Entity 里面的 auditor 的那些字段塞值
      @PreUpdate
      public void touchForUpdate(Object target) {
         if (handler != null) {
            handler.getObject().markModified(target);
         }
      }
   }
```

（3）通过调用关系图和AuditingEntityListener我们其实可以发现以下两点情况：

- AuditingEntityListener通过委托设计模式，委托AuditingHandler进行处理，而我们看AuditingHandler的源码会发现，里面就是根据ID和Version（我们后面讲）来判断我们的对象是新增还是更新，从而来更改时间字段和User字段。而User字段是通过AuditorAware的实现类来取的，并且AuditorAware没有默认实现类，只有我们自己的实现类，也就是AuditorAware的实现类必须我们自己来定义，否则启动会报

错。

- AuditingEntityListener的代码如此简单，我们能不能自定义的呢？答案是肯定的，我们通过@PrePersist、@PreUpdate查看源码得出。

Java Persistence API底层又帮我们提供的Callbacks注解方式如表7-1所示。

表7-1 Callbacks注解方式

| Type | Description | 一句话描述 |
| --- | --- | --- |
| @PrePersist | Executed before the entity manager persist operation is actually executed or cascaded.<br>This call is synchronous with the persist operation | 新增之前 |
| @PreRemove | Executed before the entity manager remove operation is actually executed or cascaded.<br>This call is synchronous with the remove operation | 删除之前 |
| @PostPersist | Executed after the entity manager persist operation is actually executed or cascaded.<br>This call is invoked after the database INSERT is executed | 新增之后 |
| @PostRemove | Executed after the entity manager remove operation is actually executed or cascaded.<br>This call is synchronous with the remove operation | 删除之后 |
| @PreUpdate | Executed before the database UPDATE operation | 更新之前 |
| @PostUpdate | Executed after the database UPDATE operation | 更新之后 |
| @PostLoad | Executed after an entity has been loaded into the current persistence context or an entity has been refreshed | 加载之后 |

## 提示

这个方法都是同步机制，一旦报错将会影响所有底层代码执行。实际工作中实现这些方法的时候，方法体里面开启异步线程或者消息队列来异步处理日志，或者更繁重的工作。

### 7.1.4 Listener事件的扩展

#### 1．自定义EntityListener

随着DDD的设计模式逐渐被大家认可和热捧。JPA通过Listener这种机制可以很好地实现事件分离，状体分离。假如，订单的状态变化可能对我们来说比较重要，我们需要定一个类去监听订单状态变更，通知相应的逻辑代码各自去干各自的活。

第一步：新增一个OrderStatusAuditListener类，在相应的操作上添加Callbacks注解。

```
public class OrderStatusAuditListener {
   @PostPersist
   private void postPersist(OrderEntiy entity) {
       //当更新的时候做一些逻辑判断，及其事件通知。
   }
   @PostRemove
   private void PostRemove(OrderEntiy entity) {
       //当删除的时候做一些逻辑判断。
   }
   @PostUpdate
   private void PostUpdate(OrderEntiy entity) {
       //当更新的时候
       // entity.getOrderStatus()做一些逻辑判断
   }
}
```

第二步：我们的订单实体变化如下：

```
@Entity
@Table("orders")
@EntityListeners({AuditingEntityListener.class,
OrderStatusAuditListener.class})
public class OrderEntity  extends AbstractAuditable{
   @Enumerated(EnumType.STRING)
   @Column("order_status")
   private OrderStatusEnum orderStatus;
  ......
}
```

即可完成自定义EntityListener。

## 2．实际工作记录操作日志的实例

```
   public class ActionsLogsAuditListener {
      private static final Logger logger =
LoggerFactory.getLogger(ActionsLogsAuditListener.class);
      @PostLoad
      private void postLoad(Object entity) {
         this.notice(entity, OperateType.load);
      }
      @PostPersist
      private void postPersist(Object entity) {
         this.notice(entity, OperateType.create);
      }
      @PostRemove
      private void PostRemove(Object entity) {
         this.notice(entity, OperateType.remove);
      }
      @PostUpdate
      private void PostUpdate(Object entity) {
         this.notice(entity, OperateType.update);
      }
      private void notice(Object entity, OperateType type) {
          logger.info("{} 执行了 {} 操作", entity, type.getDescription());
          //我们通过 active mq 异步发出消息处理事件
         ActiveMqEventManager.notice(new ActiveMqEvent(type, entity));
      }
      enum OperateType {
          create("创建"), remove("删除"),update("修改"),load("查询");
         private final String description;
         OperateType(String description) {
            this.description=description;
         }
         public String getDescription() {
            return description;
         }
      }
   }
```

我们通过自定义的ActionsLogsAuditListener来监听我们要处理日志的实体，然后将事件变更，通过消息队列进行异步处理，这样就可以完全解耦了。当然了，这里我们解耦的方式也可以通过Spring的事件机制进行解决。我们通过工作中的此示例，来帮助大家更好地理解Audit的机制。顺便说一下处理操作的日志的正确思路，记录当前

真实发生的数据和状态及其时间即可，具体变化了什么那是在业务展示层面上要做的事情，这里没有必要做比对的事情，记住这一点之后就会让你的日志处理实现机制豁然明朗，变得容易许多。

## 7.2 @Version处理乐观锁的问题

### 1．@Version乐观锁

我们在研究Auditing的时候，发现了一个有趣的注解@Version，源码如下：

```
    /**
     * Demarcates a property to be used as version field to implement optimistic
locking on entities.
     */
    @Retention(RUNTIME)
    @Target(value = { FIELD, METHOD, ANNOTATION_TYPE })
    public @interface Version {}
```

发现它帮我们处理了乐观锁的问题，什么是乐观锁，还有线程的安全性，在另外一本书《Java并发编程从入门到精通》里面，作者做了深入的探讨。对于数据来说，简单理解：在数据库并发操作时，为了保证数据的正确性，我们会做一些并发处理，主要就是加锁。在加锁的选择上，常见有两种方式：悲观锁和乐观锁。

悲观锁：简单的理解就是把需要的数据全部加锁，在事务提交之前，这些数据全部不可读取和修改。

乐观锁：使用对单条数据进行版本校验和比较，来保证本次的更新是最新的，否则就失败，效率要高很多。实际工作中，乐观锁不止在数据库层面，其实我们在做分布式系统的时候，为了实现分布式系统的数据一致性，分布式事物的一种做法就是乐观锁。

### 2．数据库操作举例说明

悲观锁的做法：

```
select * from user where id=1 for update;
update user  set name='jack'  where id=1;
```

通过使用for update给这条语句加锁，如果事务没有提交，其他任何读取和修改，都得排队等待。在代码中，我们加事务的java方法就会自然地形成了一个锁。

乐观锁的做法：

```
select uid,name,version from user where id=1;
update user set name='jack', version=version+1 where id=1 and version=1
```

假设本次查询version=1，在更新操作时，带上这次查出来的Version，这样只有和我们上次版本一样的时候才会更新，就不会出现互相覆盖的问题，保证了数据的原子性。

## 3．@Version用法

在没有@Version之前，我们都是自己手动维护这个version的，这样很有可能忘掉。或者是我们自己底层做框架，用AOP的思路做拦截底层维护这个Version的值。而Spring Data JPA的@Version就是通过AOP机制，帮我们动态维护这个Version，从而更优雅地实现乐观锁。

实体上的version字段加上@Version注解即可。我们对上面的实体UserCustomerEntity改进如下：

```
@Entity
@Table(name = "user_customer", schema = "test", catalog = "")
public class UserCustomerEntity extends AbstractAuditable {
   //新增控制乐观锁的字段。并且加上@Version 注解
   @Version
   @Column(name = "version", nullable = true)
   private Long version;
......
}
```

## 4．实际调用

```
userCustomerRepository.save(new UserCustomerEntity("1","Jack"));
UserCustomerEntity uc= userCustomerRepository.findOne(1);
uc.setCustomerName("Jack.Zhang");
userCustomerRepository.save(uc);
```

我们会发现insert和update的SQL语句都会带上version的操作。当乐观锁更新失败的时候，会抛出异常org.springframework.orm.ObjectOptimisticLockingFailureExcept

## 5．实现原理关键代码

```
public <S extends T> S save(S entity) {
   if (entityInformation.isNew(entity)) {
      em.persist(entity);
      return entity;
   } else {
      return em.merge(entity);
   }
}
public boolean isNew(T entity) {
   if (versionAttribute == null || versionAttribute.getJavaType().isPrimitive())
{
      return super.isNew(entity);
   }
   BeanWrapper wrapper = new DirectFieldAccessFallbackBeanWrapper(entity);
   Object versionValue = wrapper.getPropertyValue(versionAttribute.getName());
   return versionValue == null;
}
```

**提示**

可以看出当更新的时候一定要带上version，如果没有version，只有ID，系统认为是新增。

# 7.3 对MvcWeb的支持

## 7.3.1 @EnableSpringDataWebSupport

Spring Data附带各种Web支持，如果模块支持库的编程模型。一般来说，通过使用启用集成支持@EnableSpringDataWebSupport注解在JavaConfig类配置。

（1）开启支持Spring Data Web的支持

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

（2）当我们配置了@EnableSpringDataWebSupport的注解之后，Spring容器将会帮我们配置注册几个基本组成部分：

- DomainClassConverter：让Spring MVC解决的实例库管理域类来自请求参数或路径变量。
- HandlerMethodArgumentResolver：实现让Spring MVC解决可分页和排序实例来自请求参数。

## 7.3.2 DomainClassConverter组件

DomainClassConverter允许你使用域类型在你的Spring MVC控制器直接方法签名，这句话怎么理解呢？看下面的实例：

```
@Controller
@RequestMapping("/user")
public class UserController {
  @RequestMapping("/{id}")
  public UserInfoEntity getUserInfo(@PathVariable("id") UserInfoEntity
userInfoEntity) {
  return user;
  }
}
```

我们看到我们的Controller里面没有引用任何userRepository，

但是，我们测试这个请求的时候，user里面是有我们实体的数据库里面的值的。@EnableSpringDataWebSupport这个注解注入的DomainClassConverter组件，帮我们解决通过了让Spring MVC path变量转换成的id类型域类，最终通过调用访问实例，达到了userRepository.findOne(id)的效果。

### 7.3.3 HandlerMethodArgumentResolvers可分页和排序

实际项目中离不开分页和排序，我们一般的做法是自己写一个page对象里面实现分页逻辑，Spring Data也帮我们很好地考虑了这个问题，提供了一种优雅的解决方案。通过@EnableSpringDataWebSupport注解也帮我们注册PageableHandlerMethodArgumentResolver的实例和SortHandlerMethodArgumentResolver的实例。注册使得Pageable和Sort成为有效的控制器方法参数。

```
@Controller
@RequestMapping(path = "/demo")
public class UserInfoController {
   @Autowired
   private UserRepository userRepository;
/**
 * 示例1：使用分页和排序的 Pageable 对象返回 Page 对象。
 * @param pageable
 * @return
 */
@RequestMapping(path = "/user/page")
@ResponseBody
public Page<UserInfoEntity> findAllByPage(Pageable pageable) {
   return userRepository.findAll(pageable);
}

/**
 * 示例2：单独使用排序，返回 HttpEntity 结果
 * @param sort
 * @return
 */
@RequestMapping(path = "/user/sort")
@ResponseBody
public HttpEntity<List<UserInfoEntity>> findAllBySort(Sort sort) {
   return new HttpEntity(userRepository.findAll(sort));
}
}
```

这种方法签名会导致Spring MVC尝试可分页实例来自请求参数，使用默认配置如下:

- Page: 你想要查找的第几页，如果你不传，默认是0。
- size: 分页大小，默认是20。
- sort: 格式为property,property(ASC|DESC)。默认升序排序（ASC）。使用多个sort参数，如果你想切换方向，例如?sort=firstname&sort=lastname,asc。

所以请求的方式如下：

（1）$ curl http://127.0.0.1:8080/demo/user/page

```
{
  "content": [
    //UserInfoEntity 的20条数据
  ],
  "last": false,
  "totalPages": 3,
  "totalElements": 41,
  "size": 20,
  "number": 0,
  "sort": null,
  "first": true,
  "numberOfElements": 20
}
```

我们看到返回结果有两部分组成：

- 一是content，即返回的内容结果。
- 二是page本身的一些信息。

（2）$ curl http://127.0.0.1:8080/demo/user/page?page=2&size=5

```
{
  "content": [
//第二页的 UserInfoEntity 的5条数据
  ],
  "last": false,
  "totalPages": 9,
  "totalElements": 41,
  "size": 5,
  "number": 2,
  "sort": null,
  "first": false,
  "numberOfElements": 5
}
```

我们看到返回结果分页的页数变了，这种结构使得我们的API接口相当的灵活。可以仔细体会一下。

（3）$ curl http://127.0.0.1:8080/demo/user/page?page=2&size=5&sort=firstName

```
  {
    "content": [
  //第二页的 UserInfoEntity 的5条数据
    ],
    "last": false,
    "totalPages": 9,
    "totalElements": 41,
    "size": 5,
    "number": 2,
    "sort":
[{"direction":"ASC","property":"firstName","ignoreCase":false,"nullHandling":"
NATIVE","ascending":true,"descending":false}],
    "first": false,
    "numberOfElements": 5
  }
```

（4）$ curl http://127.0.0.1:8080/demo/user/sort?sort=firstName,desc

按照名称倒序显示结果。

## 7.3.4 @PageableDefault改变默认的page和size

@PageableDefault改变默认的page和size。我们假设默认显示第三页的内容，默认一个的大小是10条：

```
  @RequestMapping(path = "/user/page")
  @ResponseBody
  public Page<UserInfoEntity> findAllByPage(@PageableDefault(page = 3,size = 10)
Pageable pageable) {
     return userRepository.findAll(pageable);
  }
```

请求结果如下：

```
$ curl http://127.0.0.1:8080/demo/user/page
{
  "content": [
//默认显示第3页的 UserInfoEntity 的10条数据
  ],
  "last": false,
  "totalPages": 5,
  "totalElements": 41,
  "size": 10,
  "number": 3,
  "first": false,
  "numberOfElements": 10
}
```

### 7.3.5 Page原理解析

我们通过源码发现，Spring通过动态代理机制绑定了Pageable的实现类PageRequest对象，用来存储请求中关于分写的相关参数。我们通过debug来发现spring jpa返回我们的Page的实现类是PageImpl。我们看一下类的UML图，如图7-3所示。

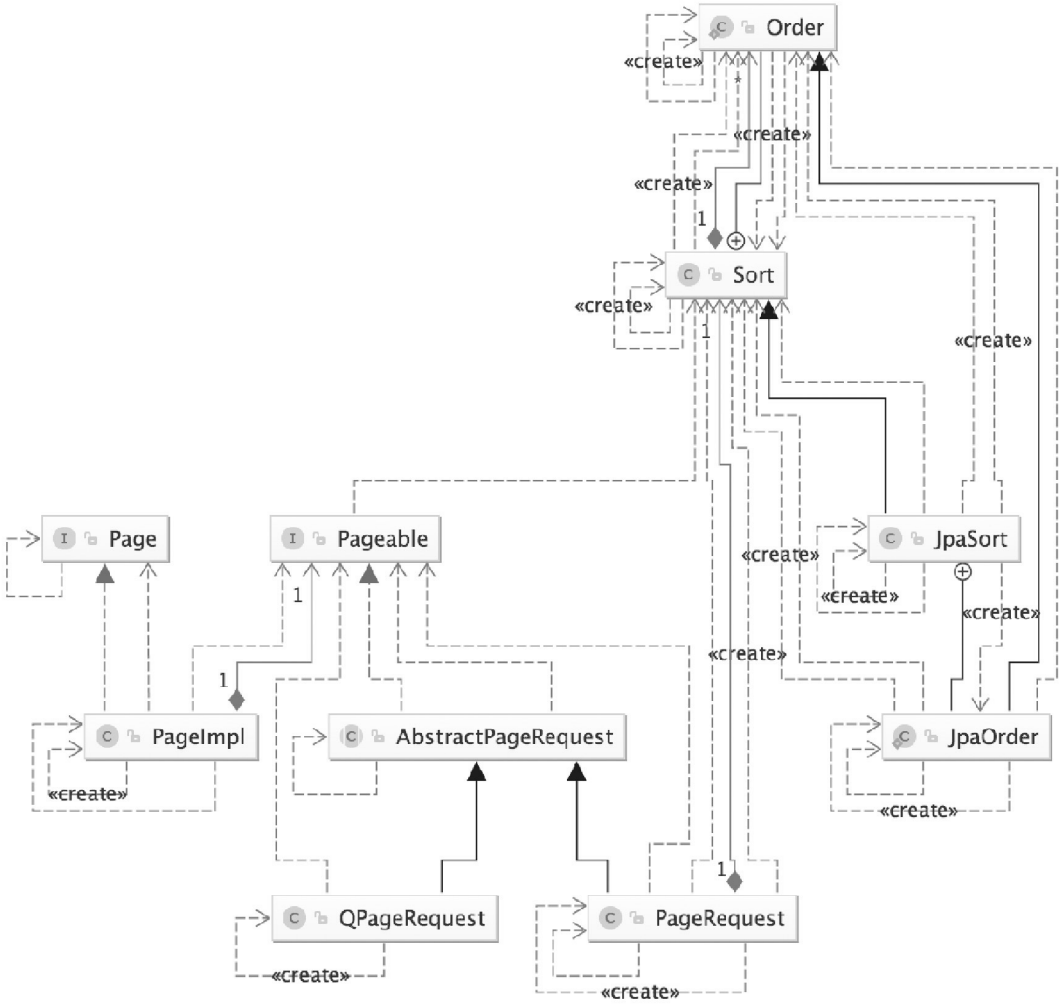



图7-3

在实际工作中，由于微服务的整个环境，我们可能通过RPC协议，如Dubbo等对外提供service的服务，service的接口的jar要尽量少引用和接口本身无关的jar，所以我们发现，其实上面说的这些对MVC的page的支持，都是在spring data common的jar里面，所以只要对外多引用这一个包即可。

## 7.4 @EnableJpaRepositories详解

### 7.4.1 Spring Data JPA加载Repositories配置简介

我们要集成Spring Data JPA，在没有引用Spring Boot的时候，

通常的做法有下面3种。

（1）XML配置。

```
<?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>
```

Spring Data通过repositories标签，允许你简单地定义一个基本方案。Spring是扫描com.acme.repositories和所有的子包接口，扩展Repository或其sub-interfaces之一。对于发现的每个接口，注册持久性特定于技术的基础设施FactoryBean创建合适的代理，处理调用查询的方法。每个bean注册的bean名称来源于接口名称，所以一个接口UserRepository将注册下userRepository的base-package属性允许通配符，你可以定义一个模式扫描包。

使用过滤器。你可能希望更细粒度的控制接口获得创建bean实例。要做到这一点，你使用< include-filter / >和<exclude-filter />元素内的<repositories />。元素的语义是完全等价的Spring的上下文名称空间。

要从实例库中排除某些接口，你可以使用以下配置:

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

（2）基于JavaConfig，基于注解的存储库配置示例。

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

（3）当我们引用Spring Boot之后，我们只需要添加@SpringBootApplication注解，什么都不需要做。我们查看其源码，找到spring.factories默认加载了org.springframework.boot.autoconfigure.data.jpa.JpaRepositor查看JpaRepositoriesAutoConfiguration源码发现其里面帮我们加载了@EnableJpaRepositories。当然了我们也可以在我们的javaconfig里面直接配置@EnableJpaRepositories，覆盖默认配置。

## 7.4.2 @EnableJpaRepositories详解

```
@Import(JpaRepositoriesRegistrar.class)
public @interface EnableJpaRepositories {
   String[] value() default {};
   String[] basePackages() default {};
   Class<?>[] basePackageClasses() default {};
   Filter[] includeFilters() default {};
   Filter[] excludeFilters() default {};
   String repositoryImplementationPostfix() default "Impl";
   String namedQueriesLocation() default "";
   Key queryLookupStrategy() default Key.CREATE_IF_NOT_FOUND;
   Class<?> repositoryFactoryBeanClass() default
JpaRepositoryFactoryBean.class;
   Class<?> repositoryBaseClass() default DefaultRepositoryBaseClass.class;
   String entityManagerFactoryRef() default "entityManagerFactory";
   String transactionManagerRef() default "transactionManager";
   boolean considerNestedRepositories() default false;
   boolean enableDefaultTransactions() default true;
}
```

（1）value等于basePackage。

用于配置扫描Repositories所在的package及子package。简单配置中的配置则等同于此项配置值，basePackages可以配置为单个字符串，也可以配置为字符串数组形式。

```
@EnableJpaRepositories(basePackages = "com.jack")
@EnableJpaRepositories(
basePackages = {"com.jack.sample.repository",
"com.jack.tower.repository"})
```

（2）basePackageClasses指定Repository所在包的类。

```
   @EnableJpaRepositories(basePackageClasses = BookRepository.class)
   @EnableJpaRepositories(basePackageClasses = {ShopRepository.class,
OrganizationRepository.class})
```

备注：测试的时候发现，配置包类的一个Repositories类，该包内其他Repositores也会被加。

（3）includeFilters（包含过滤器）。

该过滤区采用ComponentScan的过滤器类，哪些包含在内。

```
   @EnableJpaRepositories( includeFilters={@ComponentScan.Filter(type=FilterTy
pe.ANNOTATION, value=Repository.class)})
```

（4）excludeFilters

不包含过滤器，该过滤区采用ComponentScan的过滤器类，哪些不包含在内。

```
   @EnableJpaRepositories( excludeFilters={ @ComponentScan.Filter(type=FilterT
ype.ANNOTATION, value=Service.class),
@ComponentScan.Filter(type=FilterType.ANNOTATION, value=Controller.class)})
```

（5）repositoryImplementationPostfix实现类的默认尾部结束字符。

```
   @EnableJpaRepositories(value="com.jackzhang.example.quickstart.repository",
repositoryImplementationPostfix="Impl")默认"Impl"结尾。比如：ShopRepository，对应的为 ShopRepositoryImpl
```

（6）namedQueriesLocation

```
   named SQL 存放的位置，默认为 META-INF/jpa-named-queries.properties
```

内容如下：

```
   Todo.findBySearchTermNamedFile=SELECT t FROM Table t WHERE LOWER(t.description)
LIKE LOWER(CONCAT('%', :searchTerm, '%')) ORDER BY t.title ASC
```

（7）queryLookupStrategy，构建条件查询的策略，包含三种方

式CREATE，USE_DECLARED_QUERY，CREATE_IF_NOT_FOUND。

- CREATE：按照接口名称自动构建查询。
- USE_DECLARED_QUERY：用户声明查询。
- CREATE_IF_NOT_FOUND：先搜索用户声明，不存在则自动构建，此策略针对通过接口名称自动生成查询的场景。

（8）repositoryFactoryBeanClass，指定Repository的工厂类。

（9）entityManagerFactoryRef，实体管理工厂引用名称，对应到@Bean注解对应的方法。

```
    @Bean
     public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
            LocalContainerEntityManagerFactoryBean entityManagerFactoryBean = new
LocalContainerEntityManagerFactoryBean();
        ...
            return entityManagerFactoryBean;
    }
```

（10）transactionManagerRef，事务管理工厂引用名称，对应到@Bean注解对应的方法。

```
    @Bean
    public JpaTransactionManager transactionManager() {
    JpaTransactionManager transactionManager = new JpaTransactionManager();
    transactionManager.setEntityManagerFactory(entityManagerFactory().getObject
());
    return transactionManager;
    }
```

### 7.4.3 JpaRepositoriesAutoConfiguration源码解析

```
   @Configuration
   @ConditionalOnBean(DataSource.class)
   @ConditionalOnClass(JpaRepository.class)
   @ConditionalOnMissingBean({ JpaRepositoryFactoryBean.class,
        JpaRepositoryConfigExtension.class })
   @ConditionalOnProperty(prefix = "spring.data.jpa.repositories", name =
"enabled", havingValue = "true", matchIfMissing = true)
   @Import(JpaRepositoriesAutoConfigureRegistrar.class)
   @AutoConfigureAfter(HibernateJpaAutoConfiguration.class)
   public class JpaRepositoriesAutoConfiguration {

   }
```

通过源码发现此类加载了HibernateJpaAutoConfiguration，开启了JPA的默认配置文件源码如下：

```
   @Configuration
   @ConditionalOnClass({ LocalContainerEntityManagerFactoryBean.class,
EntityManager.class })
   @Conditional(HibernateEntityManagerCondition.class)
   @AutoConfigureAfter({ DataSourceAutoConfiguration.class })
   public class HibernateJpaAutoConfiguration extends JpaBaseConfiguration {
   ......}
```

还有类JpaRepositoriesAutoConfigureRegistrar开启了@EnableJpaRepositories源码如下：

```
   class JpaRepositoriesAutoConfigureRegistrar
        extends AbstractRepositoryConfigurationSourceSupport {
      @Override
      protected Class<? extends Annotation> getAnnotation() {
         return EnableJpaRepositories.class;
      }
      @EnableJpaRepositories
      private static class EnableJpaRepositoriesConfiguration {
      }
   ......
   }
```

类关系图如图7-4所示。

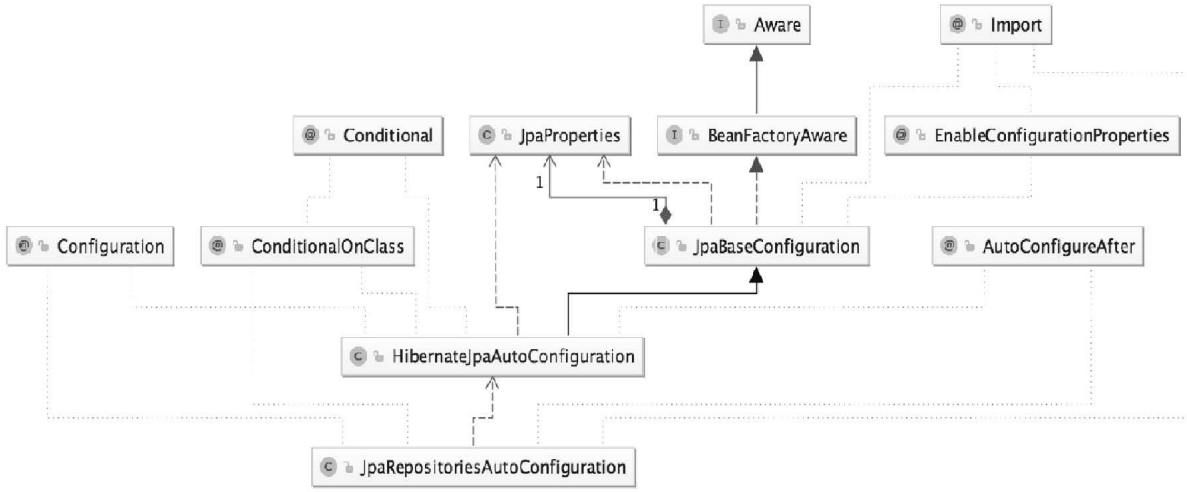



图7-4

后面的DataSource部分我们还会详细讲解。

## 7.5 默认日志简单介绍

Spring Boot在所有内部日志中使用Commons Logging，但是默认配置也提供了对常用日志的支持，如：Java Util Logging、Log4J、Log4J2和Logback。每种Logger都可以通过配置使用控制台或者文件输出日志内容。默认日志是Logback。

SLF4J——Simple Logging Facade For Java，它是一个针对于各类Java日志框架的统一Facade抽象。Java日志框架众多——常用的有java.util.logging、log4j、logback、commons-logging。Spring框架使用的是Jakarta Commons Logging API （JCL）。而SLF4J定义了统一的日志抽象接口，而真正的日志实现则是在运行时决定的——它提供了各类日志框架的binding。

Logback是Log4j框架的作者开发的新一代日志框架，它效率更高、能够适应诸多的运行环境，同时天然支持SLF4J。

默认情况下，Spring Boot会用Logback来记录日志，并用INFO级别输出到控制台。在运行应用程序和其他例子时，你应该已经看到很多INFO级别的日志了，如图7-5所示。


图7-5

从上图可以看到，日志输出内容元素具体如下：

- 时间日期：精确到毫秒。
- 日志级别：ERROR、WARN、INFO、DEBUG与TRACE。
- 进程ID：如上图57801就是进程ID。
- 分隔符：如上图---标识实际日志的开始。
- 线程名：方括号括起来（可能会截断控制台输出）。
- Logger名：通常使用源代码的类名。
- 日志内容：实际的日志内容详情。

Spring Boot为我们提供了很多默认的日志配置，所以，只要将spring-boot-starter-logging作为依赖加入到当前应用的classpath，即可“开箱即用”。

下面介绍几种在application.properties就可以配置的日志相关属性。

## 1．控制台输出

日志级别从低到高分为TRACE < DEBUG < INFO < WARN < ERROR < FATAL，如果设置为WARN，则低于WARN的信息都不会输出。

Spring Boot中默认配置ERROR、WARN和INFO级别的日志输出到控制台。你还可以通过启动你的应用程序--debug标志来启用“调试”模式（开发的时候推荐开启），以下两种方式皆可：

- 在运行命令后加入--debug标志，如：$ java -jar springTest.jar --debug。
- 在application.properties中配置debug=true，该属性置为true的时候，核心Logger（包含嵌入式容器、hibernate、Spring）会输出更多内容，但是你自己应用的日志并不会输出为DEBUG级别。

## 2．文件输出

默认情况下，Spring Boot将日志输出到控制台，不会写到日志文件。如果要编写除控制台输出之外的日志文件，则需在application.properties中设置logging.file或logging.path属性。

- logging.file：设置文件，可以是绝对路径，也可以是相对路径。如：logging.file=my.log。
- logging.path：设置目录，会在该目录下创建spring.log文件，并写入日志内容，如：logging.path=/var/log。

如果只配置logging.file，会在项目的当前路径下生成一个xxx.log日志文件。

如果只配置logging.path，在/var/log文件夹生成一个日志文件为spring.log。

**提示**

二者不能同时使用，若同时使用，则只有logging.file生效。

默认情况下，日志文件的大小达到10MB时会切分一次，产生新的日志文件，默认级别为：ERROR、WARN、INFO。

## 3．级别控制

所有支持的日志记录系统都可以在Spring环境中设置记录级别（例如在application.properties中）。

格式为：'logging.level.* = LEVEL'

- logging.level：日志级别控制前缀，*为包名或Logger名。
- LEVEL：选项TRACE、DEBUG、INFO、WARN、ERROR、FATAL、OFF。

举例：

- logging.level.com.dudu=DEBUG：com.dudu包下所有class以DEBUG级别输出。
- logging.level.root=WARN：root日志以WARN级别输出。

## 4．自定义日志配置

由于日志服务一般都在ApplicationContext创建前就初始化了，它并不是必须通过Spring的配置文件控制。因此通过系统属性和传统的Spring Boot外部配置文件依然可以很好地支持日志控制和管理。

根据不同的日志系统，你可以按如下规则组织配置文件名，就能被正确加载：

- Logback：logback-spring.xml，logback-spring.groovy，logback.xml，logback.groovy
- Log4j：log4j-spring.properties，log4j-spring.xml，log4j.properties，log4j.xml
- Log4j2：log4j2-spring.xml，log4j2.xml
- JDK（Java Util Logging）：logging.properties

Spring Boot官方推荐优先使用带有-spring的文件名作为你的日志配置（如使用logback-spring.xml，而不是logback.xml），命名为logback-spring.xml的日志配置文件，Spring Boot可以为它添加一些Spring Boot特有的配置项（下面会提到）。

上面是默认的命名规则，并且放在src/main/resources下面即可。

如果你想完全掌控日志配置，但又不想用logback.xml作为Logback配置的名字，可以通过logging.config属性指定自定义的名字：

```
logging.config=classpath:logging-config.xml
```

虽然一般并不需要改变配置文件的名字，但是如果你想针对在不同运行时Profile使用不同的日志配置，这个功能会很有用。

Debug的时候我们关系的一些日志配置：

```
//显示出 SQL 并显示 SQL 的参数，并且显示 SQL 的执行状况
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.type=trace
spring.jpa.properties.hibernate.use_sql_comments=true
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
```

## 7.6 Spring Boot JPA的版本问题

（1）由于咱们是基于Spring Boot开始配置的，本书出版的时候是根据最新的稳定的1.5.8.RELEASE版本。

```
<groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>1.5.8.RELEASE</version>
```

Spring Boot默认引用的Spring Data JPA的1.11.8.RELEASE版

本：

```
<dependency>
       <groupId>org.springframework.data</groupId>
       <artifactId>spring-data-jpa</artifactId>
       <version>1.11.8.RELEASE</version>
   </dependency>
```

而这时候官方的Spring Data JPA的最新版本是2.0.1.RELEASE：

```
<dependency>
       <groupId>org.springframework.data</groupId>
       <artifactId>spring-data-jpa</artifactId>
       <version>2.0.1.RELEASE</version>
</dependency>
```

所以这时候需要注意一下，可能有些Spring Data JPA的一些新特性、一些优化和bug的解决已经在新版本里面了。需要手动升级一下Spring Data JPA的版本。

（2）而Spring Boot的2.0版本呼之欲出，源码和代码层次结构优化了很多，但是还处于SNAPSHOT版本，本书上市的时候，希望读者注意一下新老版本的问题。

```
<parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>2.0.0.BUILD-SNAPSHOT</version>
</parent>
```

Spring Boot的2.0对应的Spring Data JPA的2.0.1.RELEASE最新版本如下：

```
<groupId>org.springframework.data</groupId>
<artifactId>spring-data-jpa</artifactId>
<version>2.0.1.RELEASE</version>
```

而Spring Data JPA的2.0.1的升级没有做向下兼容，有些方法做了重构，如果findOne变成findById()，很多单条的默认返回结构都采用了java.util.Optional的封装，其实这个默认挺好的，优雅地解

决null的返回结果问题。

# 第8章
# DataSource的配置

才能是在寂静中造就的。

——歌德

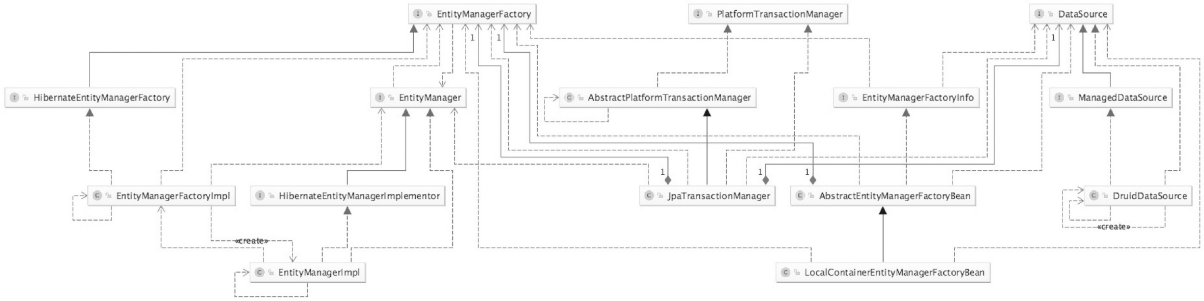



图8-1

本章我们讲解一下数据源配置的一些事情。

## 8.1 默认数据源的讲解

回顾第1章我们的Demo，我们发现只需要两步，application就可以正常运行连上数据库，但是实际工作不可能这么简单，因为在实际工作中，我们会使用其他数据源，而不会使用默认数据源。本节我们先来一步一步了解一下，一起来开启默认数据源的探索之旅吧。

### 8.1.1 通过三种方法查看默认的DataSource

第一种：日志法，我们在application.properties增加如下配置：

```
logging.level.org.springframework=DEBUG
```

然后在启动成功之后，通过开发工具IntelliJ IDEA的Debug的Console控制台，搜索“DataSource”，我们可以找到了如下日志，发现它默认是jdbc的pool的DataSource。

```
spring.datasource.type=org.apache.tomcat.jdbc.pool.DataSource
```

第二种：Debug方法，我们在manager里面的如下代码处“userRepository.findByLastName (names);” 设置一个断点，然后请求一个URL让断点进来，然后通过开发工具IntelliJ IDEA的Debug的Memory View视图，在里面搜索，如图8-2所示。

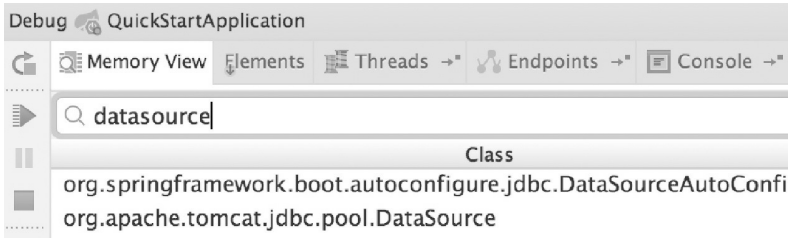



图8-2

也能发现DataSource，然后双击就能看到我们想看的内容。

第三种：是最原始的方法，最常用的原理分析方法：

（1）回到QuickStartApplication，点击@SpringBootApplication查看其源码，关键部分如下：

```
@SpringBootConfiguration
   @EnableAutoConfiguration
   public @interface SpringBootApplication {......}
```

（2）打开@EnableAutoConfiguration所在Jar包：

我们打开spring-boot-autoconfigure-1.5.8.RELEASE.jar/META-INF/spring.factories文件，会发现如下内容：

```
    # Auto Configure
    org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
    ......
    org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfigur
ation
```

（3）打开JpaRepositoriesAutoConfiguration类，内容如下：

```
    @Configuration
    @ConditionalOnBean(DataSource.class)
    @ConditionalOnClass(JpaRepository.class)
    @ConditionalOnMissingBean({ JpaRepositoryFactoryBean.class,
         JpaRepositoryConfigExtension.class })
    @ConditionalOnProperty(prefix = "spring.data.jpa.repositories", name =
"enabled", havingValue = "true", matchIfMissing = true)
    @Import(JpaRepositoriesAutoConfigureRegistrar.class)
    @AutoConfigureAfter(HibernateJpaAutoConfiguration.class)
    public class JpaRepositoriesAutoConfiguration {}
```

这时候可以发现，如果使用了Spring Boot的注解方式和传统的XML配置方式是有优先级的。如果我们配置了XML中的JpaRepositoryFactoryBean，那么我们就会沿用XML配置的一整套，而通过@ConditionalOnMissingBean这个注解来判断，就不会加载Spring Boot的JpaRepositoriesAutoConfiguration类的配置。还有就是前提条件DataSource和JpaRepository必须有相关的Jar存在。

（4）打开HibernateJpaAutoConfiguration类：

```
    @Configuration
    @ConditionalOnClass({ LocalContainerEntityManagerFactoryBean.class,
EntityManager.class })
    @Conditional(HibernateEntityManagerCondition.class)
    @AutoConfigureAfter({ DataSourceAutoConfiguration.class })
    public class HibernateJpaAutoConfiguration extends JpaBaseConfiguration
{......}
```

我们这个时候发现了DataSourceAutoConfiguration的配置类即DataSource的配置内容。

（5）打开DataSourceAutoConfiguration，这个时候我们发现最

关键的类出现了。

```
    @Configuration
    @ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })
    @EnableConfigurationProperties(DataSourceProperties.class)
    @Import({ Registrar.class,
DataSourcePoolMetadataProvidersConfiguration.class })
    public class DataSourceAutoConfiguration {......}
```

（6）先看一下DataSourcePoolMetadataProvidersConfiguration类，内容如下：

```
   @Configuration
   public class DataSourcePoolMetadataProvidersConfiguration {
      @Configuration
      @ConditionalOnClass(org.apache.tomcat.jdbc.pool.DataSource.class)
      static class TomcatDataSourcePoolMetadataProviderConfiguration {
         @Bean
         public DataSourcePoolMetadataProvider
tomcatPoolDataSourceMetadataProvider() {
            return new DataSourcePoolMetadataProvider() {
               @Override
               public DataSourcePoolMetadata getDataSourcePoolMetadata(
                     DataSource dataSource) {
                  if (dataSource instanceof org.apache.tomcat.jdbc.pool.DataSource)
{
                     return new TomcatDataSourcePoolMetadata(
                           (org.apache.tomcat.jdbc.pool.DataSource) dataSource);
                  }
                  return null;
               }
            };
         }
      }
   @Configuration
   @ConditionalOnClass(org.apache.commons.dbcp.BasicDataSource.class)
   @Deprecated
   static class CommonsDbcpPoolDataSourceMetadataProviderConfiguration {
      @Bean
      public DataSourcePoolMetadataProvider
commonsDbcpPoolDataSourceMetadataProvider() {
         return new DataSourcePoolMetadataProvider() {
            @Override
            public DataSourcePoolMetadata getDataSourcePoolMetadata(
                  DataSource dataSource) {
               if (dataSource instanceof org.apache.commons.dbcp.BasicDataSource)
{
                  return new CommonsDbcpDataSourcePoolMetadata(
                        (org.apache.commons.dbcp.BasicDataSource) dataSource);
               }
               return null;
            }
         };
      }
   }
   ......//其他我们不常用的 DataSource 省略，可以自行查看源码
   }
```

通过查看它的代码发现，Spring Boot为我们的DataSource提供了最常见的两种默认配置Tomcat的JDBC和Apache的dbcp，就看你引用哪个datasoure的jar包了。因为我们开篇的示例使用了Spring Boot

的默认配置，而它默认引用Tomcat的容器，所以默认我们用的是Tomcat 的JDBC的DataSource及其连接池。当我们引用了Jetty或者Netty等容器，连接池和DataSource的实现方式也会跟着变化。

### 8.1.2 DataSource和JPA的配置属性

我们来看下我们的DataSource和JPA都有哪些配置属性。我们接着上面的类DataSourceAutoConfiguration，通过@EnableConfigurationProperties(DataSourceProperties.class)我们确认了DataSource该如何配置，打开DataSourceProperties源码：

```
    @ConfigurationProperties(prefix = "spring.datasource")
    public class DataSourceProperties
         implements BeanClassLoaderAware, EnvironmentAware, InitializingBean {
    /**
     * Name of the datasource.
     */
    private String name = "testdb";
    /**
     * Generate a random datasource name.
     */
    private boolean generateUniqueName;
    /**
     * Fully qualified name of the JDBC driver. Auto-detected based on the URL by
default.
     */
    private String driverClassName;
    /**
     * JDBC url of the database.
     */
    private String url;
    /**
     * Login user of the database.
     */
    private String username;
    /**
     * Login password of the database.
     */
    private String password;
    /**
     * JNDI location of the datasource. Class, url, username & password are ignored
when
     * set.
     */
    private String jndiName;
    ......//如果还有一些特殊的配置直接看这个类的源码即可。
    }
```

我们看到了配置数据的关键的几个属性的配置，及其一共有哪些属性值可以去配置。

@ConfigurationProperties(prefix = "spring.datasource")告诉我们application.properties里面的DataSource相关的配置必须由spring.datasource开头，这样当启动的时候，DataSourceProperties就会自动加载进来DataSource的一切配置。正如我们前面配置的一样：

```
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/test
spring.datasource.username=jack
spring.datasource.password=jack123
```

### 8.1.3 JpaBaseConfiguration

回过头来再来看HibernateJpaAutoConfiguration的父类JpaBaseConfiguration，打开关键内容如下：

```
@EnableConfigurationProperties(JpaProperties.class)
@Import(DataSourceInitializedPublisher.Registrar.class)
public abstract class JpaBaseConfiguration implements BeanFactoryAware {
   private final DataSource dataSource;
   private final JpaProperties properties;
......}
```

这个时候我们发现了JpaProperties类：

```
@ConfigurationProperties(prefix = "spring.jpa")
public class JpaProperties {
  //jpa 原生的一些特殊属性
   private Map<String, String> properties = new HashMap<String, String>();
    //databasePlatform 名字，默认和 Database 一样。
   private String databasePlatform;
    //数据库平台 MYSQL、DB2......
   private Database database;
    //是否根据实体创建 Ddl
   private boolean generateDdl = false;
    //是否显示 sql，默认不显示
   private boolean showSql = false;
   private Hibernate hibernate = new Hibernate();
......
}
```

这个时候我们再打开Hibernate类：

```
    public static class Hibernate {
    private String ddlAuto;
    /**
     * Use Hibernate's newer IdentifierGenerator for AUTO, TABLE and SEQUENCE. This
is
     * actually a shortcut for the "hibernate.id.new_generator_mappings" property.
     * When not specified will default to "false" with Hibernate 5 for back
     * compatibility.
     */
    private Boolean useNewIdGeneratorMappings;
    @NestedConfigurationProperty
    private final Naming naming = new Naming();
    ......//我们看到 Hibernate 类就这三个属性。
    }
```

我们再打开Naming源码看一下命名规范：

```
    public static class Naming {
      private static final String DEFAULT_HIBERNATE4_STRATEGY =
"org.springframework.boot.orm.jpa.hibernate.SpringNamingStrategy";
      private static final String DEFAULT_PHYSICAL_STRATEGY =
"org.springframework.boot.orm.jpa.hibernate.SpringPhysicalNamingStrategy";
      private static final String DEFAULT_IMPLICIT_STRATEGY =
"org.springframework.boot.orm.jpa.hibernate.SpringImplicitNamingStrategy";
      /**
       * Hibernate 5 implicit naming strategy fully qualified name.
       */
      private String implicitStrategy;
      /**
       * Hibernate 5 physical naming strategy fully qualified name.
       */
      private String physicalStrategy;
      /**
       * Hibernate 4 naming strategy fully qualified name. Not supported with Hibernate
       * 5.
       */
      private String strategy;
    ......}
```

可以看到，这里面Naming命名策略，兼容了Hibernate4和Hibernate5并且给出了默认的策略。后面章节我们做详细解释。

所以我们看得到的配置文件中关于JPA的配置基本上就这些。

```
spring.jpa.database-platform=mysql
spring.jpa.generate-ddl=false
spring.jpa.hibernate.ddl-auto=none
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.type=trace
spring.jpa.properties.hibernate.use_sql_comments=true
spring.jpa.properties.hibernate.jdbc.batch_size=50
```

### 8.1.4 Configuration思路

其实我们实际工作中，遇到问题，经常看到开发人员用百度狂搜，看看别人是怎么配置的，然后试了半天发现怎么配置都没效果。其实这里给大家提供了一个思路，我们在找配置项的时候，看看源码都支持哪些key，这些key分别代表什么意思，再到百度搜索，这样我们才能对症下药，正确完美地完成配置文件的配置。

## 8.2 AliDruidDataSource的配置

（1）在实际工作中，我们其实很少直接用Tomcat的JDBC连接的，一般都会用其他形式的DataSource。这里列举一个使用频次最高的AliDruid，看看如何配置。

```
<!--druid-->
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid</artifactId>
        <version>1.1.5</version>
    </dependency>
```

（2）一样的思路，我们打开DruidDataSourceAutoConfigure配置类。

```
    @Configuration
    @ConditionalOnClass(com.alibaba.druid.pool.DruidDataSource.class)
    @AutoConfigureBefore(DataSourceAutoConfiguration.class)
    @EnableConfigurationProperties({DruidStatProperties.class,
DataSourceProperties.class})
    @Import({DruidSpringAopConfiguration.class,
           DruidStatViewServletConfiguration.class,
           DruidWebStatFilterConfiguration.class,
           DruidFilterConfiguration.class})
    public class DruidDataSourceAutoConfigure {
        @Bean
        @ConditionalOnMissingBean
        public DataSource dataSource() {
            return new DruidDataSourceWrapper();
        }
        /**
         * Register the {@link DataSourcePoolMetadataProvider} instances to support
DataSource metrics.
         *
         * @see DataSourcePoolMetadataProvidersConfiguration
         */
        @Bean
        public DataSourcePoolMetadataProvider
druidDataSourcePoolMetadataProvider() {
            return new DataSourcePoolMetadataProvider() {
                @Override
                public DataSourcePoolMetadata getDataSourcePoolMetadata(DataSource
dataSource) {
                    if (dataSource instanceof DruidDataSource) {
                        return new DruidDataSourcePoolMetadata((DruidDataSource)
dataSource);
                    }
                    return null;
                }
            };
        }
    }
```

可以发现druid继承了DataSourceProperties的配置。

（3）我们打开DruidDataSourceWrapper：

```
    @ConfigurationProperties("spring.datasource.druid")
    class DruidDataSourceWrapper extends DruidDataSource implements
InitializingBean {
        @Autowired
        private DataSourceProperties basicProperties;
        @Override
        public void afterPropertiesSet() throws Exception {
            //if not found prefix 'spring.datasource.druid' jdbc
properties ,'spring.datasource' prefix jdbc properties will be used.
            if (super.getUsername() == null) {
                super.setUsername(basicProperties.determineUsername());
            }
            if (super.getPassword() == null) {
                super.setPassword(basicProperties.determinePassword());
            }
            if (super.getUrl() == null) {
                super.setUrl(basicProperties.determineUrl());
            }
            if (super.getDriverClassName() == null) {

super.setDriverClassName(basicProperties.determineDriverClassName());
            }
        }
    ......}
```

我们发现了DataSource的配置方法：

```
    spring.datasource.druid.url=jdbc:mysql://127.0.0.1:3306/test # 或
spring.datasource.url=
    spring.datasource.druid.username=jack # 或 spring.datasource.username=
    spring.datasource.druid.password=jack123 # 或 spring.datasource.password=
    spring.datasource.druid.driver-class-name=com.mysql.jdbc.Driver #或
spring.datasource.driver-class-name=
```

（4）如果再打开DruidDataSource类，就会发现了连接池的配置方法：

```
spring.datasource.druid.initial-size=
spring.datasource.druid.max-active=
spring.datasource.druid.min-idle=
spring.datasource.druid.max-wait=
spring.datasource.druid.pool-prepared-statements=
spring.datasource.druid.max-pool-prepared-statement-per-connection-size=
spring.datasource.druid.max-open-prepared-statements= #和上面的等价
spring.datasource.druid.validation-query=
spring.datasource.druid.validation-query-timeout=
spring.datasource.druid.test-on-borrow=
spring.datasource.druid.test-on-return=
spring.datasource.druid.test-while-idle=
spring.datasource.druid.time-between-eviction-runs-millis=
spring.datasource.druid.min-evictable-idle-time-millis=
spring.datasource.druid.max-evictable-idle-time-millis=
spring.datasource.druid.filters= #配置多个英文逗号分隔
....//more
```

如果我们再继续往上面看DruidAbstractDataSource就会发现了很多默认值。

（5）如果依次打开以下类：

```
@Import({DruidSpringAopConfiguration.class,
DruidStatViewServletConfiguration.class,
DruidWebStatFilterConfiguration.class,
DruidFilterConfiguration.class})
```

就会发现druid的更多配置：

```
# WebStatFilter 配置，说明请参考 Druid Wiki，配置 WebStatFilter
spring.datasource.druid.web-stat-filter.enabled= #是否启用 StatFilter 默认值
true
spring.datasource.druid.web-stat-filter.url-pattern=
# StatViewServlet 配置，说明请参考 Druid Wiki，配置_StatViewServlet
spring.datasource.druid.stat-view-servlet.enabled= #是否启用 StatViewServlet
默认值 true
spring.datasource.druid.stat-view-servlet.login-username=
spring.datasource.druid.stat-view-servlet.login-password=
```

Druid的更多配置请读者参看官方文档吧，这里只是给大家举例如何一步一步地查看这些配置，从而得到如何配置的方法。

## 8.3 事务的处理及其讲解

### 8.3.1 默认@Transactional注解式事务

#### 1．@EnableTransactionManagement

正常情况下，需要在我们的ApplicationConfig类加上@EnableTransactionManagement注解才能开启事务管理。通过DataSource的研究步骤spring.factories里面默认加载TransactionAutoConfiguration类，查看源码可以确认里面已经加了此注解，默认采用AdviceMode.PROXY，所以默认情况的事务管理机制是代理方式的，通过添加@Transactional注解式配置方法。我们通过查看可以知道，SimpleJpaRepository每个方法都是有事务的。

#### 2．查看@Transactional源码

```
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface Transactional {
   @AliasFor("transactionManager")
   String value() default "";
   @AliasFor("value")
   String transactionManager() default "";
   Propagation propagation() default Propagation.REQUIRED;
   Isolation isolation() default Isolation.DEFAULT;
   int timeout() default TransactionDefinition.TIMEOUT_DEFAULT;
   boolean readOnly() default false;
   Class<? extends Throwable>[] rollbackFor() default {};
   String[] rollbackForClassName() default {};
   Class<? extends Throwable>[] noRollbackFor() default {};
   String[] noRollbackForClassName() default {};
}
```

（1）@Transactional注解中常用参数说明如表8-1所示。

表8-1 @Transactional注解中常用参数说明

| 参数名称 | 功能描述 |
| --- | --- |
| readOnly | 该属性用于设置当前事务是否为只读事务，设置为 true 表示只读，false 则表示可读写，默认值为 false。例如：@Transactional(readOnly=true) |
| rollbackFor | 该属性用于设置需要进行回滚的异常类数组，当方法中抛出指定异常数组中的异常时，则进行事务回滚。例如：<br>指定单一异常类：@Transactional(rollbackFor=RuntimeException.class)<br>指定多个异常类：@Transactional(rollbackFor={RuntimeException.class, Exception.class}) |
| rollbackForClassName | 该属性用于设置需要进行回滚的异常类名称数组，当方法中抛出指定异常名称数组中的异常时，则进行事务回滚。例如：<br>指定单一异常类名称：@Transactional(rollbackForClassName="RuntimeException")<br>指定多个异常类名称：@Transactional(rollbackForClassName={"RuntimeException", "Exception"}) |
| noRollbackFor | 该属性用于设置不需要进行回滚的异常类数组，当方法中抛出指定异常数组中的异常时，不进行事务回滚。例如：<br>指定单一异常类：@Transactional(noRollbackFor=RuntimeException.class)<br>指定多个异常类：@Transactional(noRollbackFor={RuntimeException.class, Exception.class}) |
| noRollbackForClassName | 该属性用于设置不需要进行回滚的异常类名称数组，当方法中抛出指定异常名称数组中的异常时，不进行事务回滚。例如：<br>指定单一异常类名称：@Transactional(noRollbackForClassName="RuntimeException")<br>指定多个异常类名称：<br>@Transactional(noRollbackForClassName={"RuntimeException","Exception"}) |
| propagation | 该属性用于设置事务的传播行为<br>例如：@Transactional(propagation=Propagation.NOT_SUPPORTED,readOnly=true) |
| isolation | 该属性用于设置底层数据库的事务隔离级别，事务隔离级别用于处理多事务并发的情况，通常使用数据库的默认隔离级别即可，基本不需要进行设置 |
| timeout | 该属性用于设置事务的超时秒数，默认值为-1，表示永不超时 |
| transactionManager/value | 指定 transactionManager，当有多个 datasource 的时候 |

（2） propagation：传播行为。

所谓事务的传播行为是指：如果在开始当前事务之前，一个事务上下文已经存在，此时有若干选项可以指定一个事务性方法的执行行为。

我们可以看
org.springframework.transaction.annotation.Propagation枚举类

中定义的7个表示传播行为的枚举值：

```
public enum Propagation {
REQUIRED(0),
SUPPORTS(1),
MANDATORY(2),
REQUIRES_NEW(3),
NOT_SUPPORTED(4),
NEVER(5),
NESTED(6);
}
```

- REQUIRED：如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务。
- SUPPORTS：如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。
- MANDATORY：如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。
- REQUIRES_NEW：创建一个新的事务，如果当前存在事务，则把当前事务挂起。
- NOT_SUPPORTED：以非事务方式运行，如果当前存在事务，则把当前事务挂起。
- NEVER：以非事务方式运行，如果当前存在事务，则抛出异常。
- NESTED：如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于REQUIRED。

指定方法：通过使用propagation属性设置，例如：

```
@Transactional(propagation = Propagation.REQUIRED)
```

（3）Isolation：隔离级别。

隔离级别是指若干个并发的事务之间的隔离程度，与我们开发时候主要相关的场景包括：脏读取、重复读、幻读。

我们可以看org.springframework.transaction.annotation.Isolation枚举类中定义的4个表示隔离级别的值：

```
public enum Isolation {
DEFAULT(-1),
READ_UNCOMMITTED(1),
READ_COMMITTED(2),
REPEATABLE_READ(4),
SERIALIZABLE(8);
}
```

- DEFAULT：这是默认值，表示使用底层数据库的默认隔离级别。对大部分数据库而言，通常这值就是：READ_COMMITTED。
- READ_UNCOMMITTED：该隔离级别表示一个事务可以读取另一个事务修改但还没有提交的数据。该级别不能防止脏读和不可重复读，因此很少使用该隔离级别。
- READ_COMMITTED：该隔离级别表示一个事务只能读取另一个事务已经提交的数据。该级别可以防止脏读，这也是大多数情况下的推荐值。
- REPEATABLE_READ：该隔离级别表示一个事务在整个过程中可以多次重复执行某个查询，并且每次返回的记录都相同。即使在多次查询之间有新增的数据满足该查询，这些新增的记录也会被忽略。该级别可以防止脏读和不可重复读。
- SERIALIZABLE：所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。但是这将严重影响程序的性能。通常情况下也不会用到该级别。

指定方法：通过使用isolation属性设置，例如：

```
@Transactional(isolation = Isolation.DEFAULT)
```

（4）Spring Boot的这种默认机制，只需要在我们用事务时，在

方法上或者此方法的类上加上@Transactional注解即可。而实际工作中，我们一般都要在Service层的某些方法上加事务，以保证整个方法的事务。示例如下：

```
    @Transactional(rollbackOn = Exception.class)
    public void saveUserInfo() throws Exception {
       userCustomerRepository.save(new
UserCustomerEntity("jackzhang@mail.com","jackzhang"));
       userRepository.save(new UserInfoEntity("jack_test","name"));
        throw new Exception("test");......//此方法体有多个 repository 的调用，模拟异常，
事务会回滚的
    }
```

## 3．注意的几点

（1）@Transactional只能被应用到public方法上，对于其他非public的方法，如果标记了@Transactional也不会报错，但方法没有事务功能。

（2）用spring事务管理器，由Spring来负责数据库的打开、提交、回滚。默认遇到运行期例外（throw new RuntimeException("注释");）会回滚，即遇到不受检查（unchecked）的例外时回滚；而遇到需要捕获的例外（throw new Exception("注释");）不会回滚，即遇到受检查的例外（就是非运行时抛出的异常，编译器会检查到的异常叫受检查例外或说受检查异常）时，需我们指定方式来让事务回滚；要想所有异常都回滚，要加上@Transactional(rollbackFor={Exception.class,其他异常})。如果让unchecked例外不回滚，如@Transactional(notRollbackFor=RunTimeException.class)。

（3）@Transactional注解应该只被应用到public可见度的方法上。如果你在protected、private或者package-visible的方法上使用@Transactional注解，它也不会报错，但是这个被注解的方法将不会展示已配置的事务设置。

（4）@Transactional注解可以被应用于接口定义和接口方法、类定义和类的public方法上。然而，请注意仅仅@Transactional注解

的出现不足以开启事务行为，它仅仅是一种元数据，能够被识别@Transactional注解和上述的配置适当的具有事务行为的beans所使用。上面的例子中，其实正是元素的出现开启了事务行为。

（5）Spring团队的建议是你在具体的类（或类的方法）上使用@Transactional注解，而不要使用在类所要实现的任何接口上。你当然可以在接口上使用@Transactional注解，但是这将只能当你设置了基于接口的代理时它才生效。因为注解是不能继承的，这就意味着如果你正在使用基于类的代理时，那么事务的设置将不能被基于类的代理所识别，而且对象也将不会被事务代理所包装（将被确认为严重的）。因此，请接受Spring团队的建议并且在具体的类上使用@Transactional注解。

（6）事务有两种配置方法，一种是我们现在说的显式的注解式事务，当我们在注解式事务下，不加注解service方法上是没有任何事务的。还有一种是隐式事务，ASPECTJ的思路配置方法，所以不是没有加@Transactional注解就一定没有事务。

### 8.3.2 声明式事务

声明式事务，又叫隐式事务，或者叫ASPECTJ事务。

实际工作中，每个方法都让我们加上@Transactional注解，可能工作量有点大，也有时候会忘，所以我们经常看到有开发团队配置拦截式事务，虽然Spring官方不太推荐。

只需要在我们的项目中新增一个子类AspectjTransactionConfig即可，如下：

```
    @Configuration
    @EnableTransactionManagement
    public class AspectjTransactionConfig {
       public static final String transactionExecution = "execution (*
com.jackzhang.example..service.*.*(..))";
       @Autowired
       private PlatformTransactionManager transactionManager;
       @Bean
       public DefaultPointcutAdvisor defaultPointcutAdvisor() {
           //指定一般要拦截哪些类
          AspectJExpressionPointcut pointcut = new AspectJExpressionPointcut();
          pointcut.setExpression(transactionExecution);

          //配置 advisor
          DefaultPointcutAdvisor advisor = new DefaultPointcutAdvisor();
          advisor.setPointcut(pointcut);

           //指定不同的方法用不通的策略
          Properties attributes = new Properties();
          attributes.setProperty("get*", "PROPAGATION_REQUIRED,-Exception");
          attributes.setProperty("add*", "PROPAGATION_REQUIRED,-Exception");
          attributes.setProperty("save*", "PROPAGATION_REQUIRED,-Exception");
          attributes.setProperty("update*", "PROPAGATION_REQUIRED,-Exception");
          attributes.setProperty("delete*", "PROPAGATION_REQUIRED,-Exception");

           //创建 Interceptor
          TransactionInterceptor txAdvice = new
TransactionInterceptor(transactionManager, attributes);
          advisor.setAdvice(txAdvice);
          return advisor;
       }
    }
```

这样我们的Service就会自动拥有了事务，可以加@Transactional来覆盖全局的配置。

## 8.4 如何配置多数据源

### 8.4.1 在application.properties中定义两个DataSource

定义两个DataSource用来读取application.properties中的不同配置。如下例子中，主数据源配置为spring.datasource.one开头的配置，第二数据源配置为spring.datasource.two开头的配置。

```
//这是默认配置，我们做一下对比
spring.datasource.url=db1
spring.datasource.username=db1_username
spring.datasource.password=db1_password
//# Druid 数据源配置，继承 spring.datasource.* 配置，相同则覆盖
...
spring.datasource.druid.initial-size=5
spring.datasource.druid.max-active=5
...
//#Druid 数据源 1 配置，继承 spring.datasource.druid.* 配置，相同则覆盖
//#db1的配置会上面的配置
...
spring.datasource.druid.one.url=db1
spring.datasource.druid.one.username=db1_username
spring.datasource.druid.one.password=db1_password
spring.datasource.druid.one.max-active=10
spring.datasource.druid.one.max-wait=10000
...
//# Druid 数据源 2 配置，继承 spring.datasource.druid.* 配置，相同则覆盖
...
spring.datasource.druid.two.url=db2
spring.datasource.druid.two.username=db2_username
spring.datasource.druid.two.password=db2_password
spring.datasource.druid.two.max-active=20
spring.datasource.druid.two.max-wait=20000
...
```

### 8.4.2 定义两个DataSourceConfigJava类

两个DataSourceConfig类内容如下：

```
   @Configuration
   @EnableTransactionManagement
   @EnableJpaRepositories(
        entityManagerFactoryRef="entityManagerFactoryOne",
        transactionManagerRef="transactionManagerOne",
         basePackages= { "com.jackzhang.example.one" }) //设置 Repository 所在位置
   @EnableConfigurationProperties(JpaProperties.class)
   public class DataSourceOneConfig {
      /**
       * 配置数据源1
       */
      @Primary
      @Bean(name = "dataSourceOne")
      @ConfigurationProperties("spring.datasource.druid.one")
      public DataSource dataSourceOne(){
         return DruidDataSourceBuilder.create().build();
      }

      @Autowired
      @Qualifier("dataSourceOne")
      private DataSource oneDataSource;
      @Primary
      @Bean(name = "entityManagerOne")
      public EntityManager entityManager(EntityManagerFactoryBuilder builder) {
         return
entityManagerFactoryOne(builder).getObject().createEntityManager();
      }
      @Primary
      @Bean(name = "entityManagerFactoryOne")
      public LocalContainerEntityManagerFactoryBean entityManagerFactoryOne
(EntityManagerFactoryBuilder builder) {
         return builder
              .dataSource(oneDataSource)
              .properties(getVendorProperties(oneDataSource))
                .packages("com.jackzhang.example.one") //设置实体类所在位置
              .persistenceUnit("onePersistenceUnit")
```

```
                .build();
      }
      @Autowired
      private JpaProperties jpaProperties;
      private Map<String, String> getVendorProperties(DataSource dataSource) {
         return jpaProperties.getHibernateProperties(dataSource);
      }
      @Primary
      @Bean(name = "transactionManagerOne")
      public PlatformTransactionManager
transactionManagerOne(EntityManagerFactoryBuilder builder) {
         return new
JpaTransactionManager(entityManagerFactoryOne(builder).getObject());
      }
   }
   @Configuration
   @EnableTransactionManagement
   @EnableJpaRepositories(
        entityManagerFactoryRef="entityManagerFactoryTwo",
        transactionManagerRef="transactionManagerTwo",
         basePackages= { "com.jackzhang.example.two" }) //设置 Repository 所在位置
   @EnableConfigurationProperties(JpaProperties.class)
   public class DataSourceTwoConfig {
      /**
        * 配置数据源2
       */
      @Primary
      @Bean(name = "dataSourceTwo")
      @ConfigurationProperties("spring.datasource.druid.two")
      public DataSource dataSourceTwo(){
         return DruidDataSourceBuilder.create().build();
      }

      @Autowired
      @Qualifier("dataSourceTwo")
      private DataSource twoDataSource;
      @Primary
      @Bean(name = "entityManagerTwo")
      public EntityManager entityManager(EntityManagerFactoryBuilder builder) {
         return
entityManagerFactoryTwo(builder).getObject().createEntityManager();
      }
      @Primary
      @Bean(name = "entityManagerFactoryTwo")
      public LocalContainerEntityManagerFactoryBean entityManagerFactoryTwo
(EntityManagerFactoryBuilder builder) {
         return builder
               .dataSource(twoDataSource)
               .properties(getVendorProperties(twoDataSource))
                 .packages("com.jackzhang.example.two") //设置实体类所在位置
               .persistenceUnit("twoPersistenceUnit")
               .build();
      }
      @Autowired
```

```
      private JpaProperties jpaProperties;
      private Map<String, String> getVendorProperties(DataSource dataSource) {
         return jpaProperties.getHibernateProperties(dataSource);
      }
      @Primary
      @Bean(name = "transactionManagerTwo")
      public PlatformTransactionManager
transactionManagerTwo(EntityManagerFactoryBuilder builder) {
         return new
JpaTransactionManager(entityManagerFactoryTwo(builder).getObject());
      }
   }
```

我们发现DataSourceTwoConfig、DataSourceOneConfig内容基本一样，思路就是管理两套DataSource，从而带来了两套transactionManager。分别在这两个package下创建各自的实体和数据访问接口即可。当然了也可以通过@Transactional(rollbackFor = Exception.class, transactionManager= "transactionManagerOne")来手动选择哪个数据源。

随着微服务的推行，其实很少有多数据源的场景的，作者不建议出现多数据源，当出现的时候就要想想，模块划分的是否合理，是否可以通过服务去解决，但不排除Job等。

## 8.5 Naming命名策略详解及其实践

用JPA离不开@Entity实体，我都知道实体里面有字段映射，而字段映射的方法有两种：

- 显式命名：在映射配置时，设置的数据库表名、列名等，就是进行显式命名，即通过@Column注解配置。
- 隐式命名：显式命名一般不是必要的，所以可以选择不设置名称，这时就交由hibernate进行隐式命名。另外，隐式命名还包括那些不能进行显式命名的数据库标识符，即不加@Column注解时的默认映射规则。

### 8.5.1 Naming命名策略详解

Naming的源码发现Hibernate 4的时候隐式命名策略是org.springframework.boot.orm.jpa.hibernate.SpringNamingStrat而我们发现Hibernate 5将隐式命名策略拆分成了两步：implicitStrategy和physicalStrategy。从官方的文档中得知这样做的目的是提高灵活性，减少构建命名策略过程中用到的重复的信息。

- 第一个阶段是从对象模型中提取一个合适的逻辑名称，这个逻辑名称可以由用户指定，通过@Column和@Table等注解完成，也可以通过被Hibernate的ImplicitNamingStrategy指定。
- 第二个阶段是将上述的逻辑名称解析成物理名称，物理名称是由Hibernate中的PhysicalNamingStrategy决定。两个阶段是有先后顺序的。

## 1．ImplicitNamingStrategy

当一个实体对象没有显式地指明它要映射的数据库表或者列的名称时，在Hibernate内部就要为我们隐式处理，比如一个实体没有在@Table中指明表名，那么表名隐式地被认为是实体名，或者@Entity中提供的名称。比如一个实体没有在@Column中的指明列名，那么列名隐式地被认为是该实体对应的字段名。而我们看到源码默认的隐式策略采用的类是org.springframework.boot.orm.jpa.hibernate.SpringImplicitNam

Hibernate中定义了多个ImplicitNamingStrategy的实现，可以开箱即用。而之前的逻辑名称就是物理名称，这种策略只是其中一种，其他还包括：

- ImplicitNamingStrategyJpaCompliantImpl：默认的命名策略，兼容JPA 2.0的规范。
- ImplicitNamingStrategyLegacyHbmImpl：兼容Hibernate老版本中的命名规范。
- ImplicitNamingStrategyLegacyJpaImpl：兼容JPA 1.0规范中

的命名规范。

- ImplicitNamingStrategyComponentPathImpl：大部分与ImplicitNamingStrategyJpaCompliantImpl效果相同，但是对于@Embedded等注解标志的组件处理是通过使用attributePath完成的，因此如果我们在使用@Embedded注解的时候，如果要指定命名规范，可以直接继承这个类来实现。

看一下UML类图，如图8-3所示。

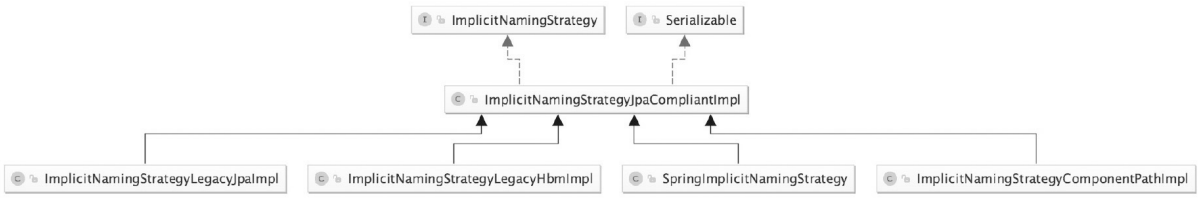



图8-3

SpringImplicitNamingStrategy继承ImplicitNamingStrategyJpaCompliantImpl，我们看源码可以看到，对外键、链表查询、索引如果未定义，都有下划线的处理策略，而table和column名字都默认与字段一样。

## 2．PhysicalNamingStrategy

在这个阶段中，是根据业务需要制定自己的命名规范，通过使用PhysicalNamingStrategy可以实现这些规则，而不需要将表名和列名通过@Table和@Column等注解显式指定。无论对象模型中是否显式地指定列名或者已经被隐式决定，PhysicalNamingStrategy都会被调用。但是对于ImplicitNamingStrategy，仅仅只有当没有显式地提供名称时才会使用，也就是说当对象模型中已经指定了@Table或者@Entity等name时，设置的ImplicitNamingStrategy并不会起作用。

所以可以看应用场景，我们可以根据实际需求来定义自己的策略是继承ImplicitNamingStrategy还是继承PhysicalNamingStrategy，

比如加上前缀t_等，或者使用分隔符“-”等。需要注意的是：PhysicalNamingStrategy永远是在ImplicitNamingStrategy之后执行的，并且永远会被执行到。我们看下PhysicalNamingStrategyStandardImpl的源码：

```
   public class PhysicalNamingStrategyStandardImpl implements
PhysicalNamingStrategy, Serializable {
      public static final PhysicalNamingStrategyStandardImpl INSTANCE = new
PhysicalNamingStrategyStandardImpl();
      @Override
      public Identifier toPhysicalCatalogName(Identifier name, JdbcEnvironment
context) {
         return name;
      }
      @Override
      public Identifier toPhysicalSchemaName(Identifier name, JdbcEnvironment
context) {
         return name;
      }
      @Override
      public Identifier toPhysicalTableName(Identifier name, JdbcEnvironment
context) {
         return name;
      }
      @Override
      public Identifier toPhysicalSequenceName(Identifier name, JdbcEnvironment
context) {
         return name;
      }
      @Override
      public Identifier toPhysicalColumnName(Identifier name, JdbcEnvironment
context) {
         return name;
      }
   }
```

默认情况下，使用的是这种策略，将ImplicitNamingStrategy传过来的逻辑名直接作为数据库中的物理名称，什么都没干直接返回，如图8-4所示。


图8-4

## 8.5.2 实际工作中的一些扩展

实际工作中，我们有这样的应用场景：当我们使用MySQL数据库的时候，一般架构师会给我们定义规范和要求，字段一般都是小写加“_”下划线组成。而我们的Java类当中都是驼峰式的字段命名方式。如果一个新的微服务，每个表都非常标准和规范，那么我们就可以让我的实体当中省去@column指定名称的过程。

（1）定义自己的PhysicalNamingStrategy，继承PhysicalNamingStrategyStandardImpl类即可，内容如下：

```
   package com.jack.model.naming;
   public class MyPhysicalNamingStrategy extends
PhysicalNamingStrategyStandardImpl {
   //重载 PhysicalColumnName 方法，修改字段的物理名称。
       @Override
       public Identifier toPhysicalColumnName(Identifier name, JdbcEnvironment
context) {
           String text = warp(name.getText());
           if(Objects.equals(text.charAt(0) , '_')){
              text = text.replaceFirst("_","");
           }
           return super.toPhysicalColumnName(new Identifier(text,name.isQuoted()),
context);
       }
   //将驼峰式命名转化成下划线分割的形式
       public static String warp(String text){
           text = text.replaceAll("([A-Z])","_$1").toLowerCase();
           if(Objects.equals(text.charAt(0) , '_')){
              text = text.replaceFirst("_","");
           }
           return text;
       }
   }
```

（2）修改application.properties，添加如下内容即可：

```
   spring.jpa.hibernate.naming.physical-strategy=com.jack.model.naming.MyPhysi
calNamingStrategy
```

（3）总结：实际工作中还是建议大家用显式@Table、@Column等注解比较好，这样可以在我们扩展的MyPhysicalNamingStrategy做规则验证工作，避免有些程序员没有按照我们的数据库规范命名。

## 8.6 完整的传统XML的配置方法

由于Spring Boot是采用约定大于配置的思路，很多东西本来需要我们手动去指定的，而现在里面很多默认机制，帮我们程序员减少了很多工作量。但是，实际工作中，我们要了解的反倒比原来多很多，因为实际场景可能有各种突发事件和Bug，都需要究其原因去分析。所以作者提供了一种传统的XML配置方法作为参考和对比。

```
<?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:tx="http://www.springframework.org/schema/tx"
         xmlns:jpa="http://www.springframework.org/schema/data/jpa"
         xmlns:aop="http://www.springframework.org/schema/aop"
         xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
     http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/tx
     http://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/data/jpa
     http://www.springframework.org/schema/data/jpa/spring-jpa-1.3.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd
      ">
       <context:property-placeholder
              ignore-unresolvable="true" ignore-resource-not-found="true"
              location="classpath*:/*.properties"/>
        <!-- 数据源配置，使用 Tomcat JDBC 连接池 -->
       <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"
init-method="init" destroy-method="close">
          <!-- Connection Info -->
          <property name="driverClassName" value="${jdbc.driver}"/>
          <property name="url" value="${jdbc.url}"/>
          <property name="username" value="${jdbc.username}"/>
          <property name="password" value="${jdbc.password}"/>
           <!-- 开启监控 -->
          <property name="filters" value="stat"/>
          <property name="maxActive" value="${jdbc.pool.maxActive}"/>
          <property name="initialSize" value="${jdbc.pool.initialSize}"/>
          <property name="maxWait" value="${jdbc.pool.maxWait}"/>
          <property name="minIdle" value="${jdbc.pool.minIdle}"/>
          <property name="poolPreparedStatements" value="true"/>
          <property name="maxOpenPreparedStatements" value="200"/>
       </bean>

       <!-- JPA Entity Manager 配置 -->
       <bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
          <property name="dataSource" ref="dataSource"/>
          <property name="packagesToScan" value="com.jackzhang.dao"/>
          <property name="jpaVendorAdapter">
             <bean id="hibernateJpaVendorAdapter"

class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"/>
          </property>
          <property name="jpaProperties">
             <props>
                <prop
key="hibernate.hbm2ddl.auto">${hibernate.hbm2ddl.auto}</prop>
                <prop key="hibernate.show_sql">${hibernate.show_sql}</prop>
```

```
                <prop
key="hibernate.format_sql">${hibernate.format_sql}</prop>
                <prop
key="hibernate.use_sql_comments">${hibernate.show_sql}</prop>
                <prop key="hibernate.dialect">${hibernate.dialect}</prop>
                <prop
key="hibernate.jdbc.batch_size">${hibernate.batch_size}</prop>
            </props>
         </property>
      </bean>

      <!-- JPA 事务配置 -->
      <bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
         <property name="entityManagerFactory" ref="entityManagerFactory"/>
      </bean>
       <!-- 注解方式配置事物：使用 annotation 定义事务 -->
      <!--<tx:annotation-driven transaction-manager="transactionManager"
proxy-target-class="true"/>-->

       <!-- 拦截器方式配置事物 -->
      <tx:advice id="transactionAdvice"
transaction-manager="transactionManager">
         <tx:attributes>
            <tx:method name="save*" propagation="REQUIRED"/>
            <tx:method name="update*" propagation="REQUIRED"/>
            <tx:method name="find*" propagation="SUPPORTS"/>
            <tx:method name="select*" propagation="REQUIRED" read-only="true"/>
            <tx:method name="*" propagation="SUPPORTS"/>
         </tx:attributes>
      </tx:advice>
      <aop:config>
         <aop:pointcut id="transactionPointcut" expression="execution(*
com.jackzhang.service.*.*(..))"/>
         <aop:advisor pointcut-ref="transactionPointcut"
advice-ref="transactionAdvice"/>
      </aop:config>
   </beans>
```

# 第三部分
# 延展部分

通过前面两大部分的讲解，Spring Data JPA本身算是介绍完毕了，实际工作中可能环境、版本等诸多因素会导致有些细节要在实际配置中调试，但是本质的概念和模块是不会变的。第三部分我们将介绍一下JPA生态的周边工具。

# 第9章
# IntelliJ IDEA与Spring JPA

把每件一般的事做好显现不一般；把每件简单的事做好显现不简单；把每件平凡的事做好显现不平凡。

——经典语录

## 9.1 IntelliJ IDEA概述

IntelliJ IDEA支持Java，也支持其他语言，如表9-1所示。

表9-1 IntelliJ IDEA支持的语言

| 安装插件后支持 | SQL 类 | 基本 JVM |
|---|---|---|
| PHP | PostgreSQL | Java |
| Python | MySQL | Groovy |
| Ruby | Oracle | |
| Scala | SQL Server | |
| Kotlin | | |
| Clojure | | |

IntelliJ IDEA支持的语言框架如表9-2所示。

表9-2 IntelliJ IDEA支持的语言框架

| 支持的框架 | 额外支持的语言代码提示 | 支持的容器 |
|---|---|---|
| Spring MVC | HTML5 | Tomcat |
| Spring boot | CSS3 | TomEE |
| Spring Data | SASS | WebLogic |
| Spring Cloud | LESS | JBoss |
| Spring 全家桶基本上都能找到相应插件 | JavaScript | Jetty |
| Web Services | CoffeeScript WebSphere | |
| JSF | Node.js | |
| Struts | ActionScript | |
| Hibernate | | |
| Flex | | |

本章我们重点介绍Spring Data JPA在IDEA中用到的插件。

## 9.2 DataBase插件

IntelliJ IDEA支持MySQL、PostgreSQL、Microsoft SQL Server、Microsoft Azure、Oracle、Amazon Redshift、Sybase、DB2、SQLite、HyperSQL、Apache Derby and H2，以及各种内存数据库。只要你有驱动，那就能连接上去，并且跨平台。只是如果混用Window、Mac、Linux，得找各种数据库的客户端，甚是不方便。

在IntelliJ的最右边的Database视图中，我们可以添加各种DataSource的连接，如图9-1所示。

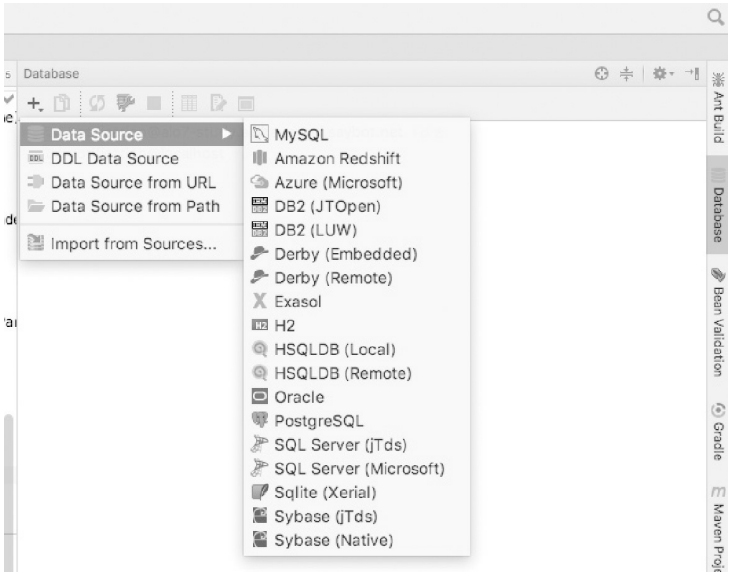



图9-1

方便直观的各种数据库本身的操作界面，如图9-2所示。

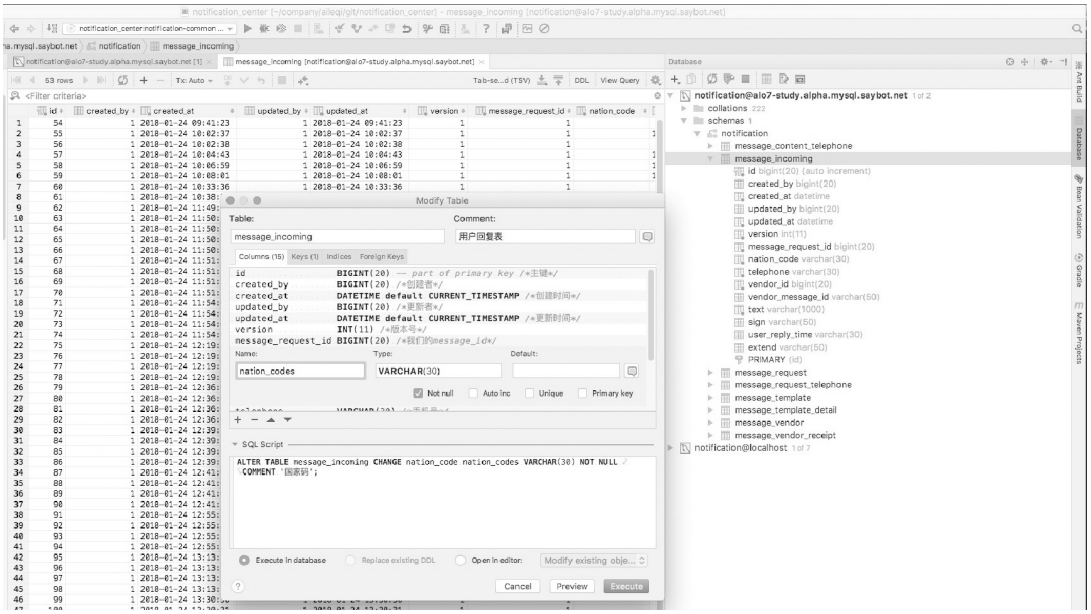



图9-2

数据库表、字段、数据等直接编辑和修改，并且还会自动帮你生成相应的SQL语句。在右边还可以直接进行搜索表、字段等，如图9-3所示。

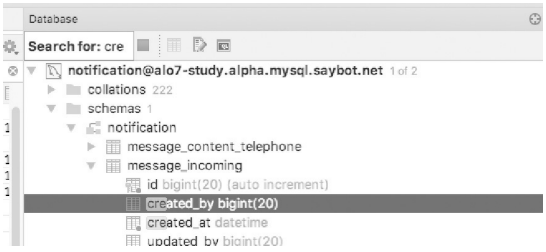



图9-3

超强和快捷的Query console。SQL控制台完美的提示，如图9-4所示。


图9-4

查询历史、SQL查询数据结果的呈现，如图9-5所示。

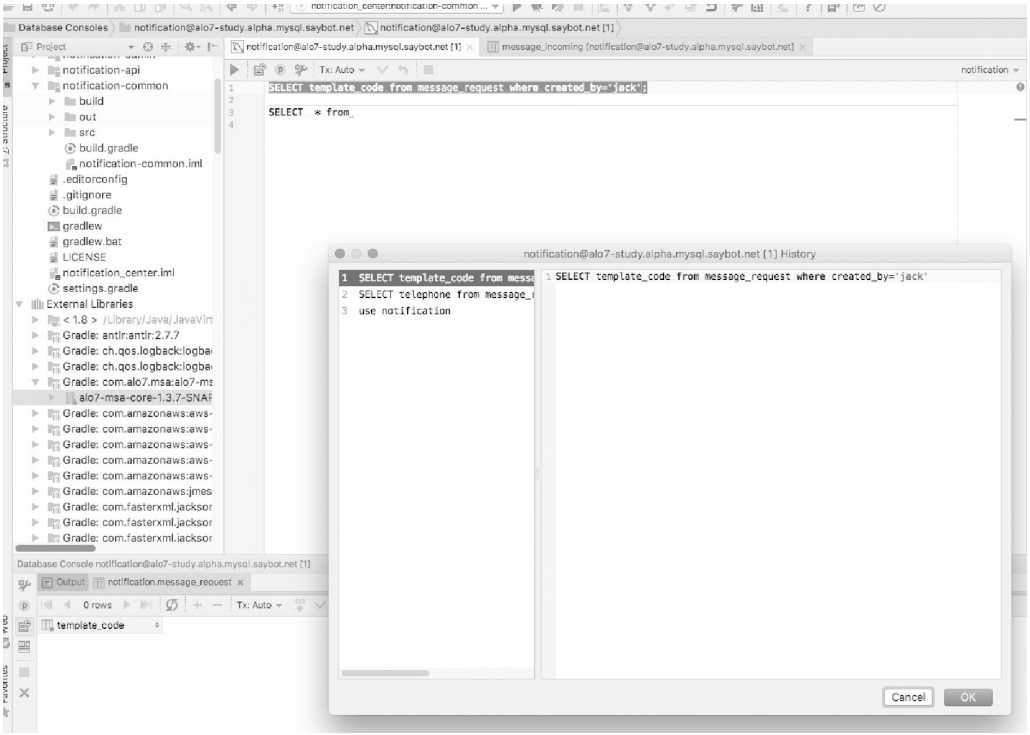



图9-5

超强的数据导出工具和操作，如图9-6所示。

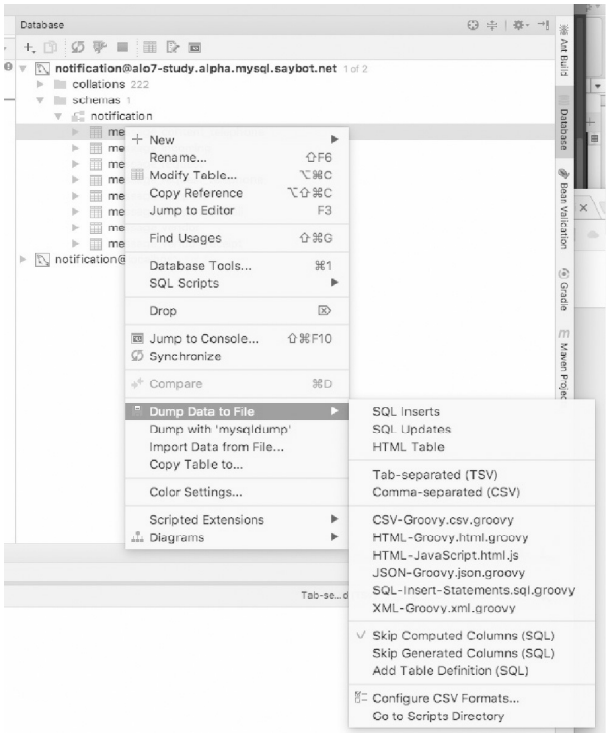



图9-6

- Find Usages：可以查出代码里面哪里用到这张表。
- SQL Script：里面支持各种DDL的操作。
- Dump Data to File：导出功能相当的丰富，各种支持（导出CSV、导出text、导出SQL语句）
- Import Data from File：也是比较常用且方便的工具，用于数据导入。

Diagrams直接生成数据模型图，唯一的缺陷是没有注释，要不就完美了，如图9-7所示。

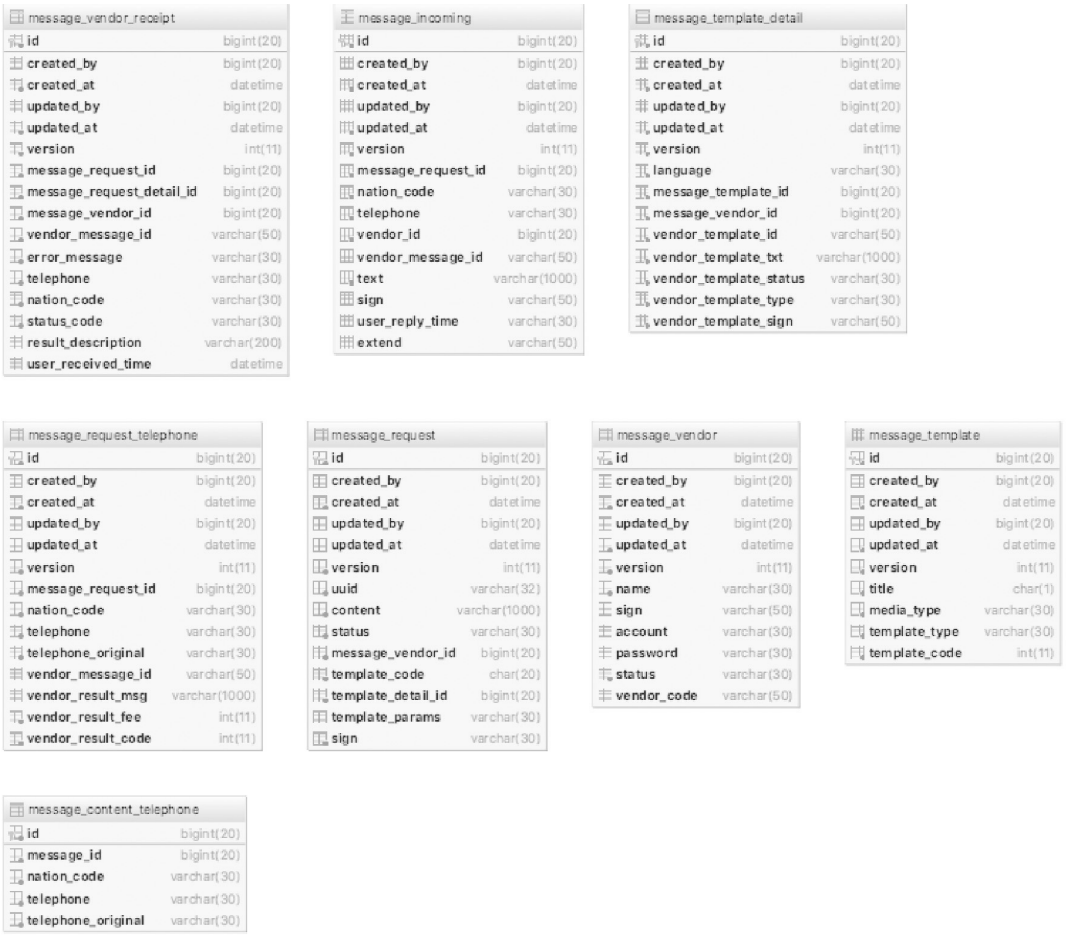



图9-7

支持查看执行计划和SQL的参数，如图9-8所示。

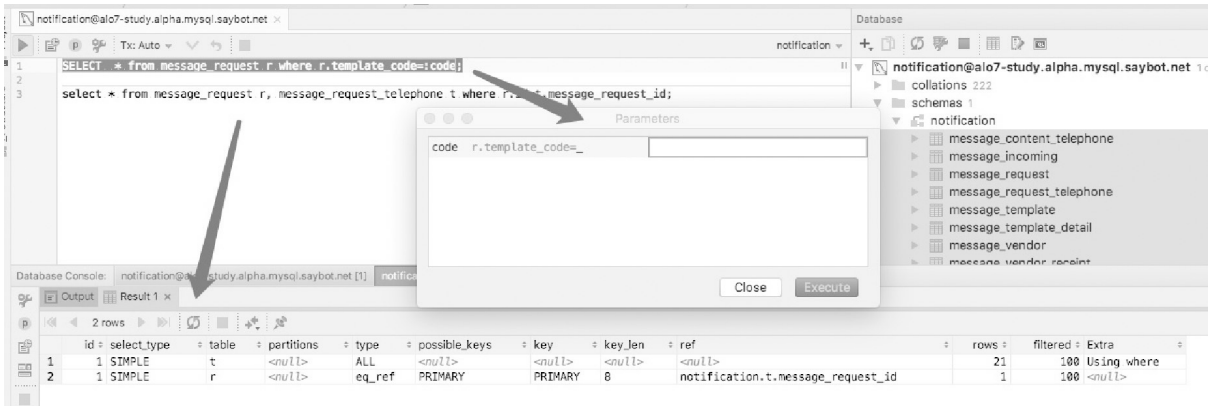



图9-8

至此Database的基本插件使用介绍完毕，剩下的大家慢慢体会它的便利性吧，个人觉得基本上你的电脑上不需要装任何其他数据库的client了。

## 9.3 Persistence及JPA相关的插件介绍

IntelliJ IDEA目前还没有针对Spring Data JPA的独立的完整插件，但是由于Spring Data JPA也是满足Java的Persistence API标准的，所有工具里面的Persistence也支持得很好的。

### 1．开启persistence界面

persistence界面如图9-9所示。

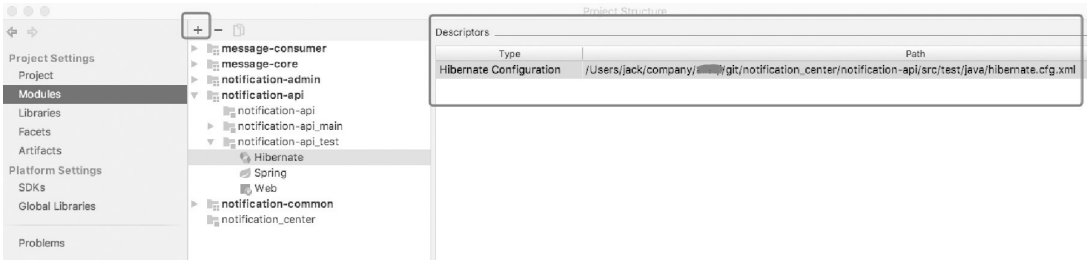



图9-9

需要打开Project Setting的设置界面，在Modules里面添加一个Hibernate模块，并且创建一个Hibernate.cfg.xml文件（建议放在

test目录，这样不至于影响你的生产环境）。直接通过图上的“+”号就可以一路默认搞定，如图9-10所示。

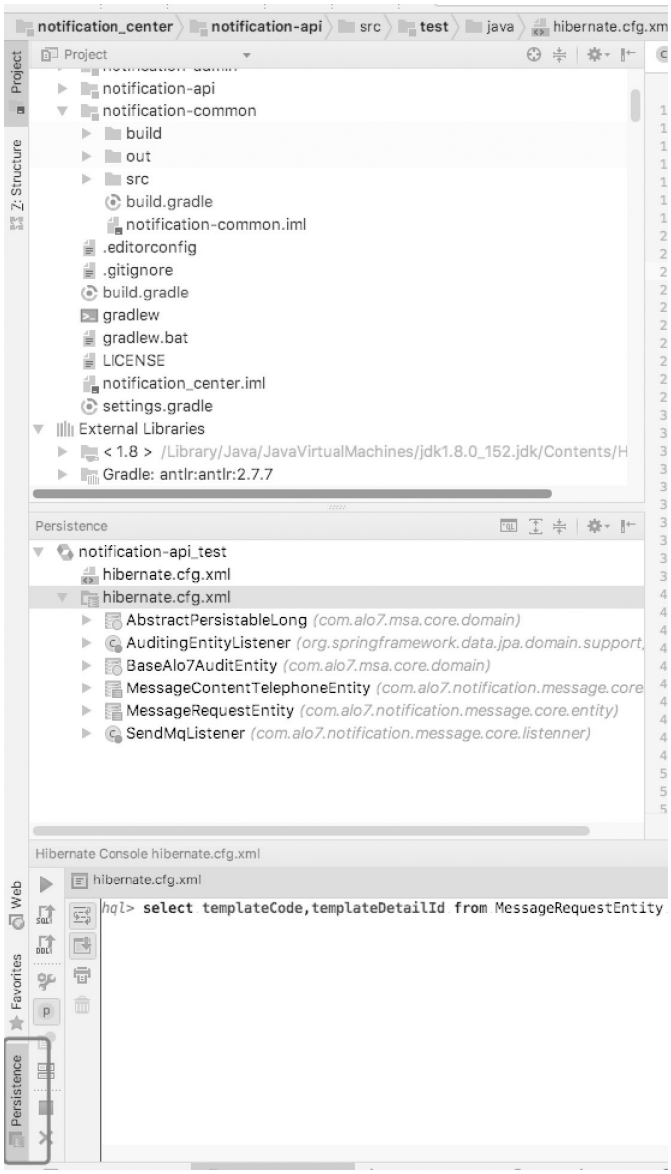



图9-10

这时候打开IntelliJ　IDEA的右下角，就会多一个插件视图，打开就会如上图所示。

## 2．利用Persistence生成Entity

（1）选中图9-10中的Persistence里面的Hibernate.xml，右击，打开生成Entity的界面，如图9-11所示。

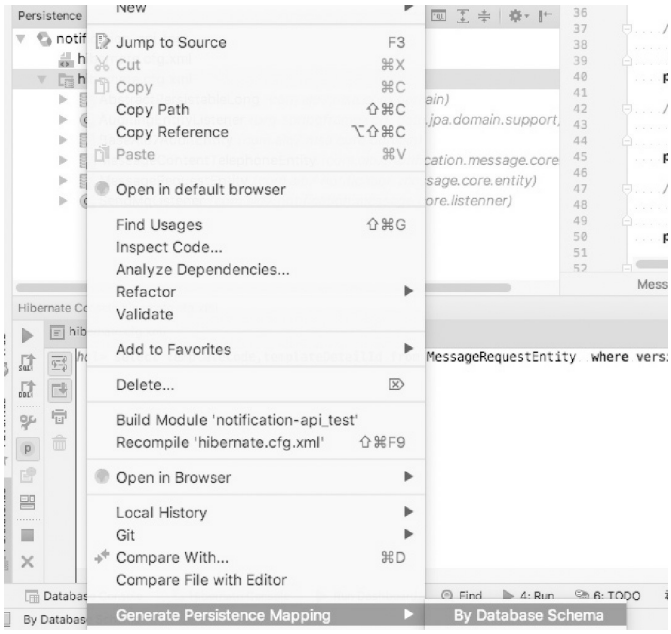



图9-11

（2）打开的界面选择要生成的实体的表和字段，非常详细，并且后面的Map As都可以编辑，太人性化了，包路径、实体后缀都可以改，如图9-12所示。

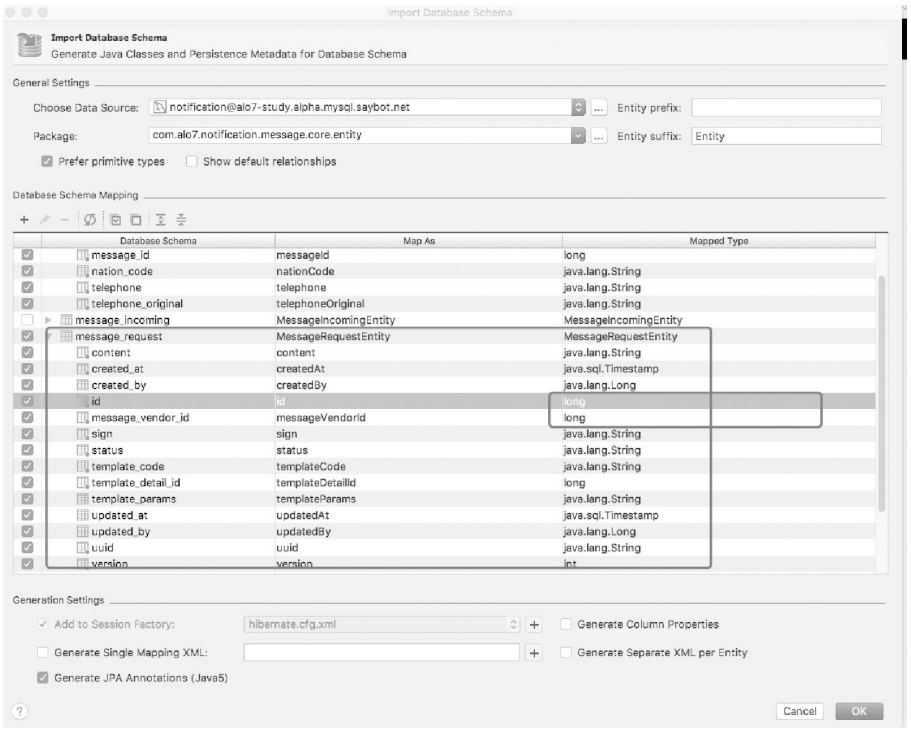



图9-12

（3）Persistence的ERP图和HQL控制台，相当帅气了，如图9-13

所示。

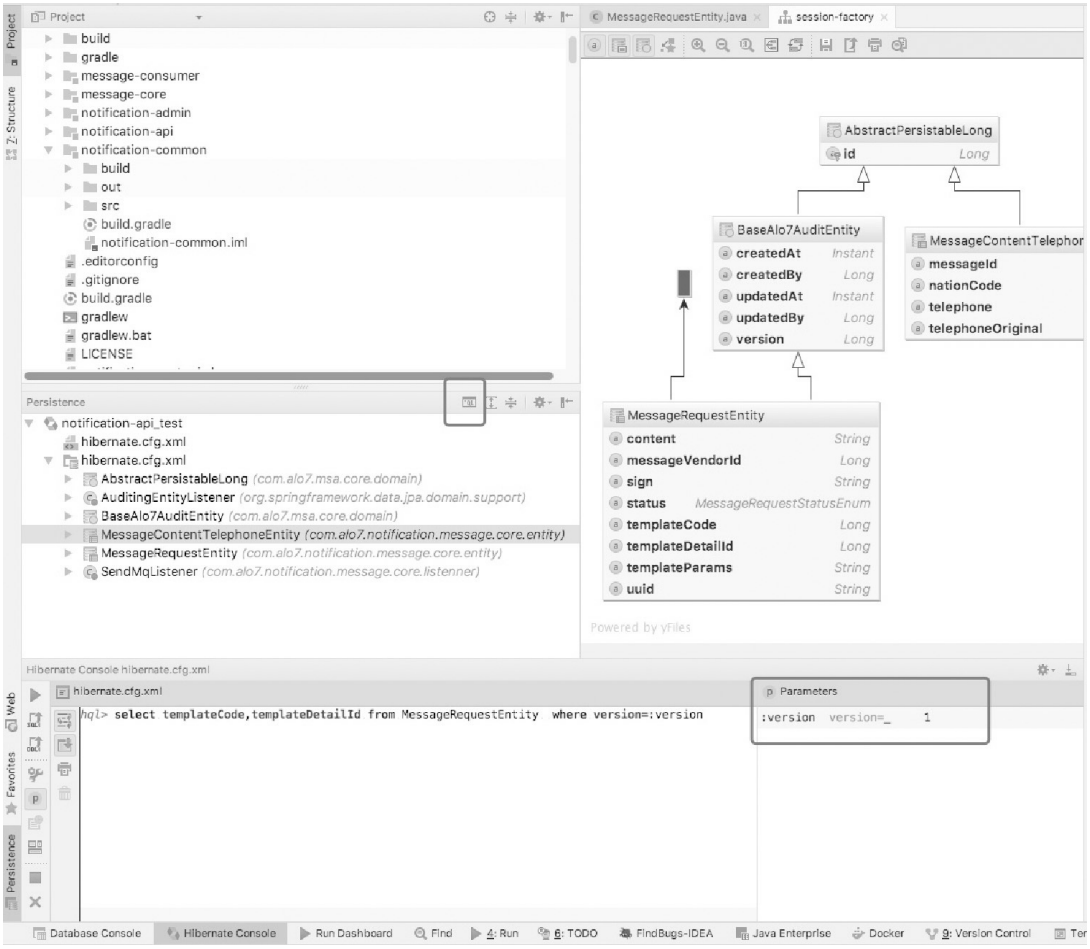



图9-13

通过这个视图我们可以充分验证我们的JPQL及其里面的参数。

（4）整体项目Spring Data的预览情况。

查看项目里面的所有的Repository，如图9-14所示。

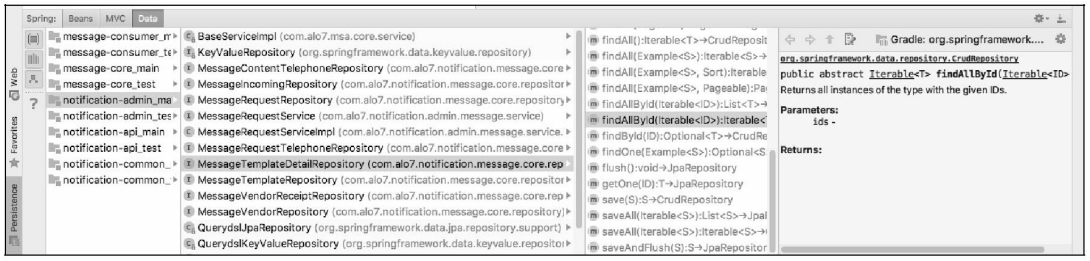



图9-14

查看项目里面的所有Entity，如图9-15所示。

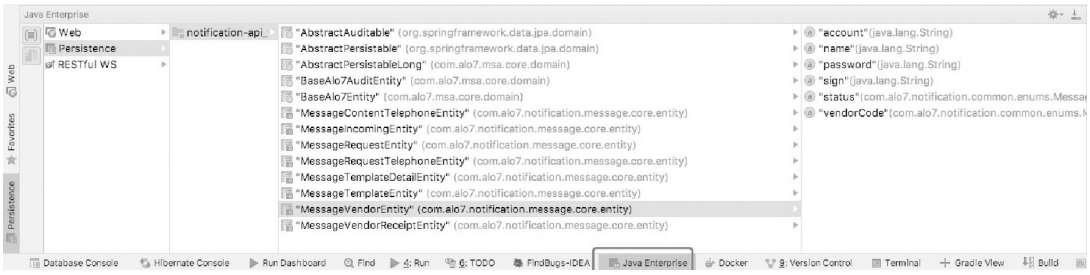



图9-15

（5）Spring Data JPA使用的时候自动提示。

当字段和表里面的字段不一致的时候的提示，如图9-16所示。

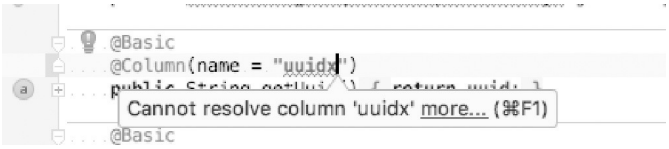



图9-16

其实，当我们写@ManyToOne、@ManyToMany字段填写不对的时候提示也挺好的。

当我们在书写Repository的时候也会根据我们的Entity和Database有很好的提示，如图9-17所示。


图9-17

Naming Strategy的提示，如图9-18所示。

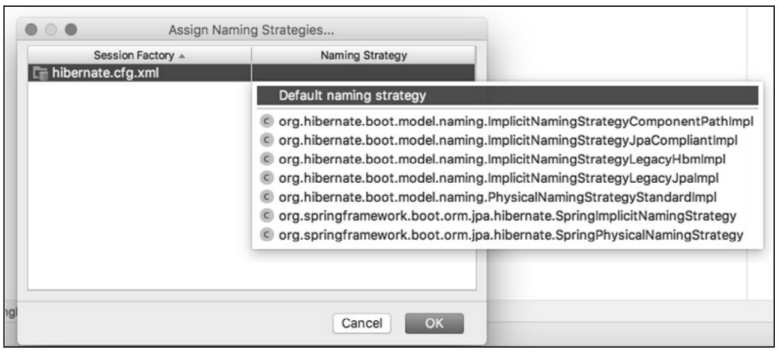



图9-18

IntelliJ IDEA如果使用慢调优方法，如图9-19所示。

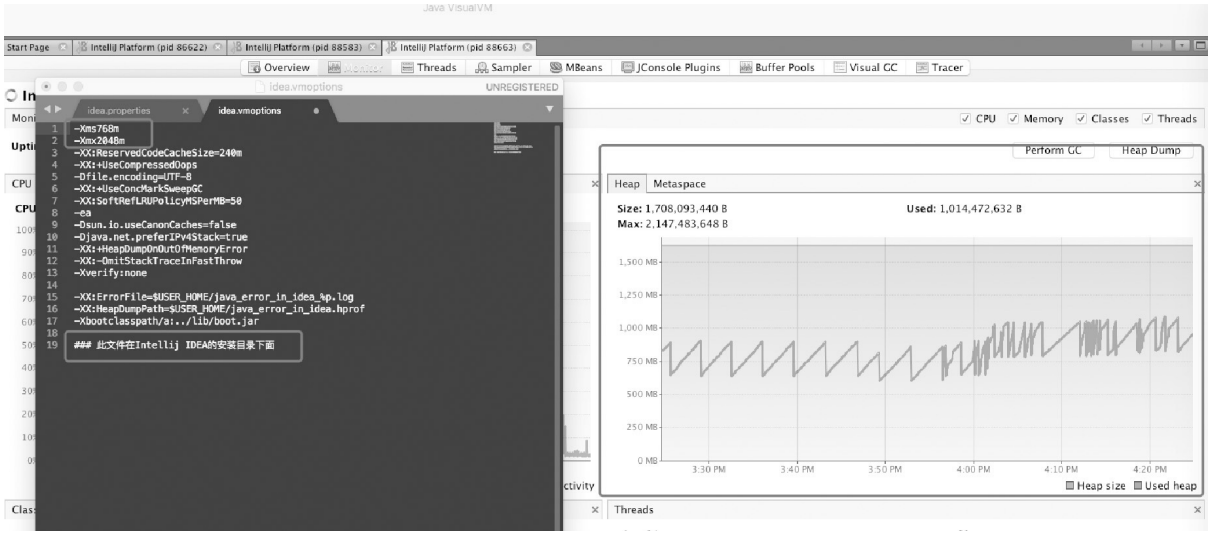



图9-19

我们利用JVM调优的方法，调一下它的堆的大小，让我们的工具飞起来。有的时候我会同时打开7～10个项目。

## 9.4 IntelliJ IDEA分析源码用到的视图

（1）第一个：Hierarchy视图，可以看到类的父亲和儿子有哪些。每个人的工具的快捷键都不一样，我的是F4，如图9-20所示。

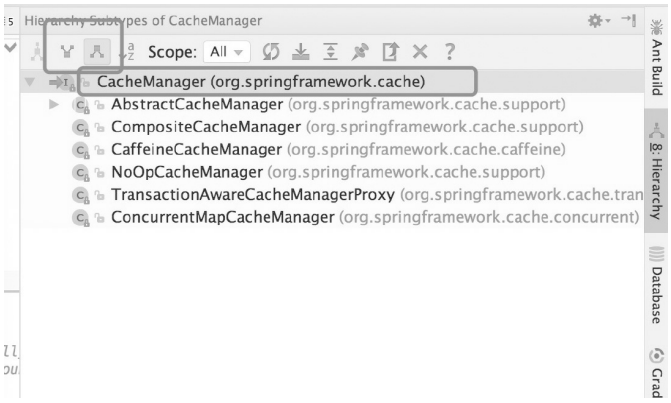



图9-20

（2）第二个：Structure视图，通过这几个按钮可以很清楚地看出本类里面有些什么东西，如图9-21所示。

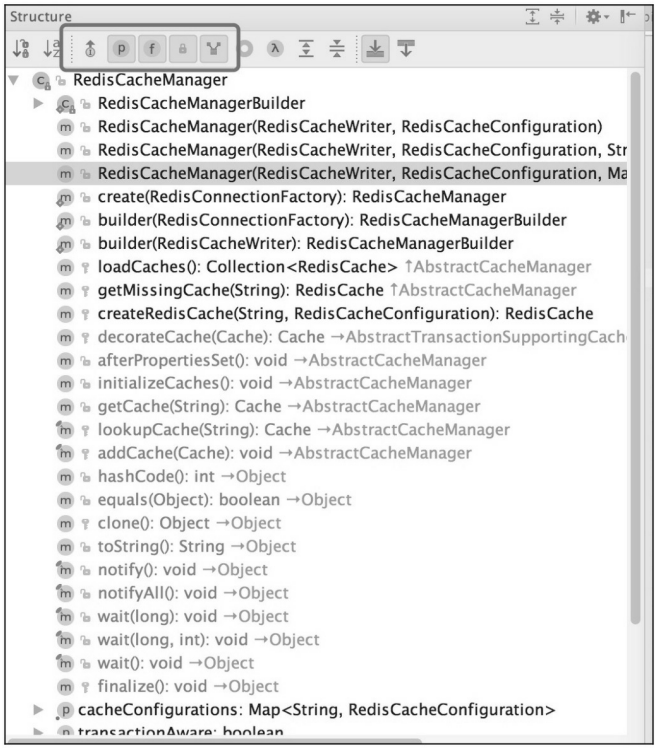



图9-21

（3）第三个：diagrams视图，可以图形化分析类之间的关系，及其调用关系，就是所谓的ER图，如图9-22所示。


图9-22

（4）第四个：debug视图，可以针对每个断点，右键可以设置很多参数，如图9-23～图9-26所示。

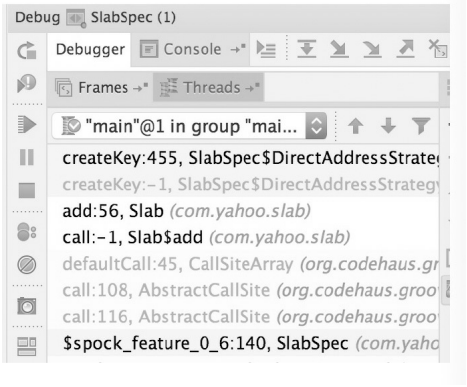



图9-23

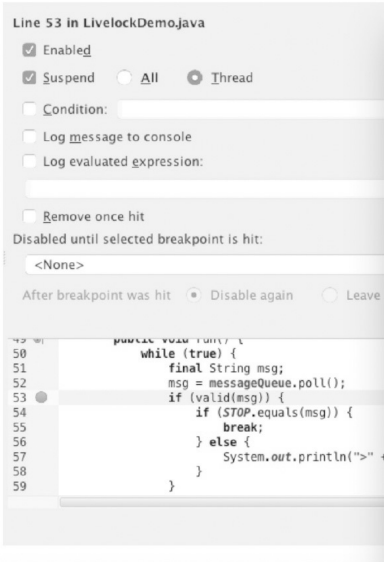


图9-24

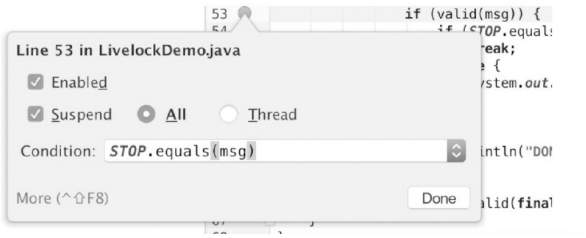



图9-25

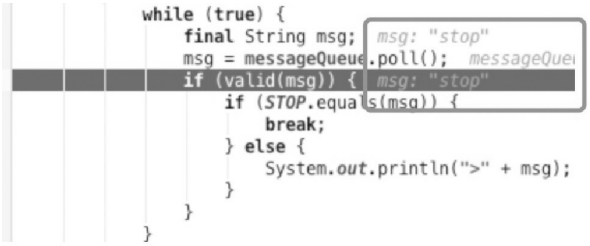



图9-26

这个工具非常好的一点就是debug的值都在后面，如图9-27所示。

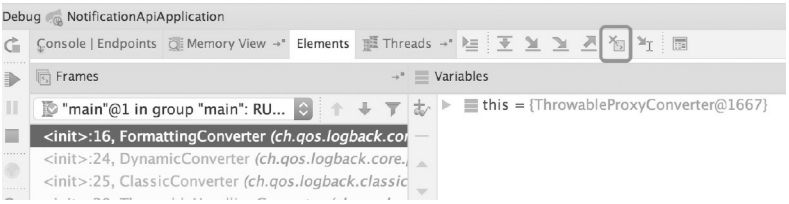



图9-27

# 第10章
# Spring Data Redis详解

## 10.1 Redis之Jedis的使用

Jedis是最受欢迎的Redis的Java版本的Client的实现端。这种使用方式属于裸用，就是不加任何修饰，直接通过Jedis操作Redis的N多特性。

主要有这么几种方式：

- 基本使用。
- 连接池的使用。
- 高可用连接（master/salve）。
- 客户端分片。

通过本节来体验一下Jedis传统模式下是如何使用的。

条件加入Jedis的jar依赖。

利用Maven添加Jedis的依赖jar：

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <!--这个注意，建议一般都选最新的-->
    <version>2.9.0</version>
    <type>jar</type>
    <scope>compile</scope>
</dependency>
```

### 1．基本使用

（1）单线程环境下使用：

```
/**
 * Created By jack on 16/12/2017
 * 单线程环境下使用，简单 Util
 **/
public class JedisClientUtil {
    public static void main(String[] args) {
        Jedis jedis = new Jedis("localhost",6379);
        jedis.set("foot", "bar");
        String value = jedis.get("foot");
        //通过这种方式就可以直接使用 redis 里面的很多命令了
    }
}
```

（2）单线程环境的正确使用姿势如下，但是在实际环境中，我们（1）里面的写法可能过于简单，真正在生产模式下，写法如下：

```
package com.example.redis.utils;

import org.springframework.beans.factory.annotation.Value;
import redis.clients.jedis.Jedis;

/**
 * Created By jack on 16/12/2017
 * 单线程环境下使用，简单 Util
 * 正常正式开发中，会把 Jedis 包装在一个单例模式中，避免每次都去重新连接，把 localhost 和 port 放到 properties 的配置文件中
 **/
public class JedisClientUtil {
    @Value("{spring.redis.host}")
    private  String host;
    @Value("{spring.redis.port}")
    private  Integer port;

    private final byte[] temp_lock = new byte[1];
    private  Jedis jedis;

    private JedisClientUtil(){}

    public Jedis getRedisClient() {
        if (jedis == null) {
            synchronized (temp_lock) {
                if (jedis == null) {
                    jedis = new Jedis(host,port);
                }
            }
        }
        return jedis;
    }
    public static void main(String[] args) {
```

```
//       @Autowired
//       JedisClientUtil jedisClientUtil;
//       如果在其他地方使用，直接 Autowired 即可。
        JedisClientUtil jedisClientUtil = new JedisClientUtil();
        Jedis jedis = jedisClientUtil.getRedisClient();
        try {
            jedis.set("foot", "bar");
            String value = jedis.get("foot");
            System.out.println(value);
        } finally {
            //注意关闭
            jedis.close();
        }
    }
}
```

## 2．连接池的使用

（1）多线程环境的正确使用姿势。

一般正常工作中很少有单线程模式，在Web环境下都是多线程进行的，这个时候引入连接池的概念来帮我们管理各个连接。简单概括一下，引入连接池是为了管理连接对象，也就是Jedis对象可能要从一个池里面取，所以Jedis提供了JedisPool的类。

PS：连接池、线程池、线程概念不清楚的，请看我的另外一篇Chat哦。

```
public class JedisClientPoolUtil {
    public static void main(String[] args) {
        JedisPool pool = new JedisPool(new JedisPoolConfig(), "127.0.0.1",6379);
        Jedis jedis = pool.getResource();
        try {
    jedis.set("foot", "bar");
    String value = jedis.get("foot");
    System.out.println(value);
} finally {
    //注意关闭
    jedis.close();
}
    }
}
```

（2）工作中一般会做如下改进，来保证可用性。

```
package com.example.redis.utils;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;
import redis.clients.jedis.Jedis;
```

```
import redis.clients.jedis.JedisPool;
import redis.clients.jedis.JedisPoolConfig;

/**
 * Created By jack on 16/12/2017
 *
 * 多线程环境下，线程池的正确使用方法，单例的连接池，单例的配置。
 * 此处给大家提供一个种思路，如果用 Spring Boot 的话，可以基于@Configuration 和@Bean
的配置方法，此处仅仅是举例说明。
 **/
@Component
public class JedisClientPoolUtil {
    @Value("${spring.redis.host}")
    private String host;
    @Value("${spring.redis.port}")
    private Integer port;

    private final static byte[] temp_lock = new byte[1];
    private JedisPool jedisPool;

    /**
     * 把连接池做成单例的，这点需要注意
     *
     * @return
     */
    private JedisPool getJedisPool() {
        if (jedisPool == null) {
            synchronized (temp_lock) {
                if (jedisPool == null) {
                    jedisPool = new JedisPool(jedisPoolConfig(),host,port);
                }
            }
        }
        return jedisPool;
    }

    /**
     * 设置一些连接池的配置，来管理每一个连接。
     *
     * @return
     */
    private JedisPoolConfig jedisPoolConfig(){
        JedisPoolConfig jedisPoolConfig = new JedisPoolConfig();
        jedisPoolConfig.setMaxTotal(20);
        jedisPoolConfig.setMaxIdle(10);
```

```
        jedisPoolConfig.setMaxWaitMillis(1000);
        return jedisPoolConfig;
    }

    /**
     * 对外只暴露这一个方法即可
     *
     * @return
     */
    public Jedis getJedis(){
        return getJedisPool().getResource();
    }
    public static void main(String[] args) {
//        @Autowired
//        JedisClientPoolUtil jedisClientPoolUtil;
//        如果在其他地方使用，直接 Autowired 即可。
        JedisClientPoolUtil jedisClientPoolUtil = new JedisClientPoolUtil();
        Jedis jedis = jedisClientPoolUtil.getJedis();
        try {
            jedis.set("foot", "bar");
            String value = jedis.get("foot");
            System.out.println(value);
        } finally {
            //注意关闭

            jedis.close();
        }
    }
}
```

# 3．高可用连接（master/salve）

（1）高可用场景JedisSentinel。

Jedis提供哨兵模式的使用，我们都知道Redis支持master和salve模式，当发生故障的时候如何选择。新版的Redis和Jedis已经做了很好的支持，来保证我们的Reids高可用，服务器端的配置这里忽略一下，我们看看Jedis的客户端下怎么写的。

```
    /**
     * Created By jack on 16/12/2017
     * 通过哨兵获得一个Master连接，DEMO
     **/
    public class JedisSentinelPoolUtil {
        public static void main(String[] args) {
            //添加N个哨兵，当添加的时候，如果去看源码就会发现，顺带通过哨兵帮我们初始化了一个
master链接地址
            JedisSentinelPool pool = new
JedisSentinelPool("redis_master_name",Sets.newHashSet("127.0.0.1:63791","127.0
.0.1:63792"));
            //通过哨兵获得Master节点，如果有问题会重新通过哨兵获得一个Master节点
            Jedis jedis = pool.getResource();
            try {
                jedis.set("foot", "bar");
                String value = jedis.get("foot");
            } finally {
                //注意关闭
                jedis.close();
            }
        }
    }
```

（2）生产正确姿势。

和上面连接池的用法一样，也需要建立一个单例模式来获得Pool，然后根据Pool对调用者提供Jedis的使用，此处不再重复叙述。

## 4．客户端分片

```
    /**
     * 简单测试切片的写法
     */
    public class ShardedJedisPoolUtil {
       public static void main(String[] args) {
          List<JedisShardInfo> shards = Lists.newArrayList();
          shards.add(new JedisShardInfo("127.0.0.1",6379));
          shards.add(new JedisShardInfo("127.0.0.1",6378));
          //通过 list 可以创建 N 个切片
          ShardedJedisPool shardedJedisPool = new ShardedJedisPool(new
GenericObjectPoolConfig(),shards);
          ShardedJedis shardedJedis =shardedJedisPool.getResource();
          shardedJedis.set("key1","abc");
          System.out.println(shardedJedis.get("key1"));
       }
    }
```

Cluster和Sentinel的应用场景和使用方法基本上同理，目前切片个人觉得Jedis实现的还不是特别成熟，这里就不多说了，感兴趣的读者可以私下交流。

## 5．Jedis需要关心的类图

其实Jedis的客户端相对来说比较简单，主要的类如图10-1所示，底层原理就是基于Socket创建连接，然后通过redisClient发送Redis的命令到服务器端。

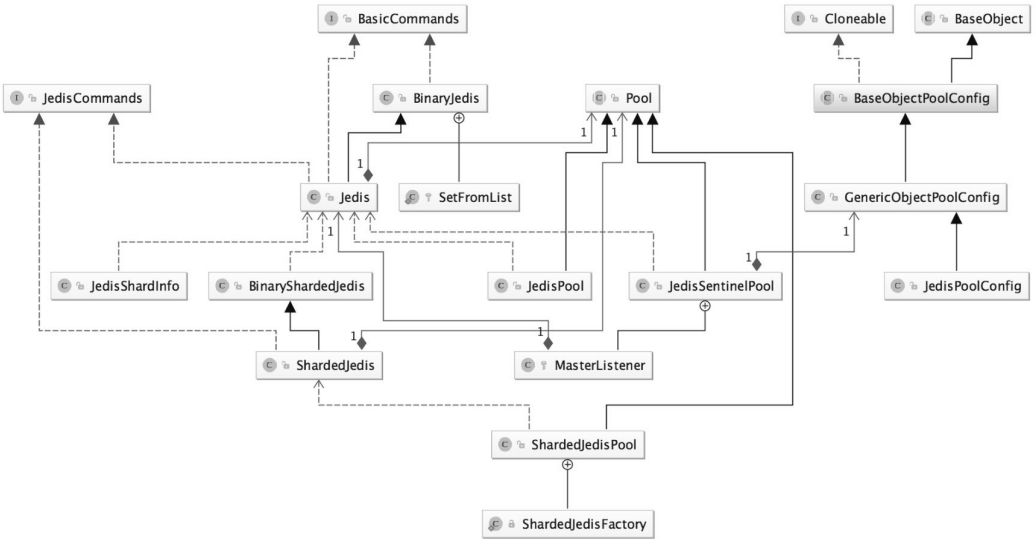



图10-1

上面介绍了基于Jedis的比较常见的配置方法，后面带读者领略一下Spring体系下面怎么玩。

### 6．Jedis实际工作中的正确使用场景

最常见的场景就是对Service这层的数据加缓存。

- 第一种做法

通常初级程序员的做法：在Service方法中，在得到数据之前，先判断缓存里面有没有通过显式地调用JedisUtil类。如果没有就从DB层去捞取，然后再丢到Redis里面。在更新的时候显式地调用RedisUtils去更新缓存，其实这时候会发现大部分代码是重复的，很不优雅。

- 第二种做法

稍微资深一点程序员的做法：可能会考虑自定义一个注解，放在每个方法上，有更新注解、有添加缓存注解，利用@Aspect拦截器机制，调用JedisUtils在拦截器里面处理上下文，其实这时候已经处理好了，只是还有很多优化的地方。

接下来我们来看看Spring Boot和Spring Data模式下怎么玩？

## 10.2 Spring Boot+Spring Data Redis配置

本节我们以Spring Boot为基础看一下Redis在里面是怎么兼容和配置的。

以Spring Boot为例分别介绍一下这四种配置方法：

- 基本使用。
- 连接池的使用。

- 高可用连接（master/salve）。
- 客户端分片。

添加Spring Data Redis依赖

如果是Spring Boot项目直接添加spring-boot-starter-data-redis即可。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

## 10.2.1 第1步：分析一下源码

一旦当我们使用Spring Boot，其实任何一个starter都会引入spring-boot-autoconfigure的jar包，然后autoconfigure就会做很多事情。

我们用Spring Boot都知道starter的原理（spring-boot-autoconfigure.jar包里面的spring.factories定义了Spring Boot默认加载的AutoConfiguration），因此，打开spring.factories文件可以找到Spring自动加载了两个Configuration类。

```
   org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration,
   org.springframework.boot.autoconfigure.data.redis.RedisRepositoriesAutoConf
iguration,
```

我们先打开RedisAutoConfiguration的源码，来一起看一下里面的关键代码片段。

（1）代码片段一：自动加载 JedisConnectionFactory。

```
@Bean
@ConditionalOnMissingBean(RedisConnectionFactory.class)
public JedisConnectionFactory redisConnectionFactory()
       throws UnknownHostException {
   return applyProperties(createJedisConnectionFactory());
}
```

通过这一段代码可以看到，JedisConnectionFactory可以自己配置，也可以直接用Spring Boot给我们提供的默认配置。

（2）代码片段二：查看createJedisConnectionFactory()的具体方法。

```
    private JedisConnectionFactory createJedisConnectionFactory() {
        //这里会取我们配置文件里面的配置，如果没有配置，new 一个默认连接池
        JedisPoolConfig poolConfig = this.properties.getPool() != null
                ? jedisPoolConfig() : new JedisPoolConfig();

        //如果配置了 Sentinel 就取哨兵的配置直接返回
        if (getSentinelConfig() != null) {
            return new JedisConnectionFactory(getSentinelConfig(), poolConfig);
        }
        //如果没有配置中 Sentinel，而配置了 Cluster 切片的配置方法，它就取 Cluster 的配置方法
        if (getClusterConfiguration() != null) {
            return new JedisConnectionFactory(getClusterConfiguration(),
poolConfig);
        }
        //默认取连接 pool 的配置方法
        return new JedisConnectionFactory(poolConfig);
    }
    .......
    //取配置文件里面的 Pool 的配置
    private JedisPoolConfig jedisPoolConfig() {
        JedisPoolConfig config = new JedisPoolConfig();
        RedisProperties.Pool props = this.properties.getPool();
        config.setMaxTotal(props.getMaxActive());
        config.setMaxIdle(props.getMaxIdle());
        config.setMinIdle(props.getMinIdle());
        config.setMaxWaitMillis(props.getMaxWait());
        return config;
    }
    .......
    //JedisPoolConfig 的类默认构造函数
    public class JedisPoolConfig extends GenericObjectPoolConfig {
        public JedisPoolConfig() {
            this.setTestWhileIdle(true);
            this.setMinEvictableIdleTimeMillis(60000L);
            this.setTimeBetweenEvictionRunsMillis(30000L);
            this.setNumTestsPerEvictionRun(-1);
        }
    }
```

通过这段代码可以看出来，前面讲到的4种Jedis的配置方式，这里默认只支持了3种。而上一节我们讲的第一种单例的基本模式Spring不推荐使用，而Spring将Pool作为默认的配置方法。

可以看出：

其一，3种配置中Pool（连接池）、Sentinel（哨兵，master/slave）和Cluster（切片）只能选择一种来配置。

其二，只需要配置我们的配置文件就可以了，剩下的就交给Spring的JedisConnectionFactory。

（3）代码片段三：查看RedisAutoConfiguration的关键源码。

```
   @Configuration
   @ConditionalOnClass({ JedisConnection.class, RedisOperations.class,
Jedis.class })
   @EnableConfigurationProperties(RedisProperties.class)
   public class RedisAutoConfiguration {
   ......
   }
   @ConfigurationProperties(prefix = "spring.redis")
   public class RedisProperties {
      /**
       * Database index used by the connection factory.
       */
      private int database = 0;
      /**
       * Redis url, which will overrule host, port and password if set.
       */
      private String url;
      /**
       * Redis server host.
       */
      private String host = "localhost";
      /**
       * Login password of the redis server.
       */
      private String password;
      /**
       * Redis server port.
       */
      private int port = 6379;
      /**
       * Enable SSL.
       */
      private boolean ssl;
      /**
       * Connection timeout in milliseconds.
       */
      private int timeout;

      private Pool pool;

      private Sentinel sentinel;

      private Cluster cluster;
      ......
   }
```

这里省略了一些中间的代码，有兴趣的读者可以看一下源码，到这一步，其实已经发现了配置文件应该怎么配置了（PS：字段名字+spring.redis前缀就是application.yml里面的key了）。如果使用

Intellij IDEA，当在application.properties输入spring.redis开头的key值的时候会给我们提示这类里面的属性值。

**提示**

（4）代码片段四：RedisConfiguration关键源码。

```
	/**
	 * Standard Redis configuration.
	 */
	@Configuration
	protected static class RedisConfiguration {

		@Bean
		@ConditionalOnMissingBean(name = "redisTemplate")
		public RedisTemplate<Object, Object> redisTemplate(
				RedisConnectionFactory redisConnectionFactory)
						throws UnknownHostException {
			RedisTemplate<Object, Object> template = new RedisTemplate<Object,
Object>();
			template.setConnectionFactory(redisConnectionFactory);
			return template;
		}

		@Bean
		@ConditionalOnMissingBean(StringRedisTemplate.class)
		public StringRedisTemplate stringRedisTemplate(
				RedisConnectionFactory redisConnectionFactory)
						throws UnknownHostException {
			StringRedisTemplate template = new StringRedisTemplate();
			template.setConnectionFactory(redisConnectionFactory);
			return template;
		}

	}
```

由以上代码可知，我们可以使用类RedisTemplate或者StringRedisTemplate来直接操作Redis的client的相关API（PS：后面有介绍）。

## 10.2.2 第2步：配置方法

通过上一小节源码分析得出来了和本节相对应的配置方法，具体介绍如下。

### 1．基本使用配置方法&连接池的配置方法

只需要在我们的application.properties里面增加如下配置即可。

```
spring.redis.host=127.0.0.1 #redis 服务器地址
spring.redis.port=6379 #端口
spring.redis.timeout=6000 #连接超时时间 毫秒
spring.redis.pool.max-active=8 # 连接池的配置，最大连接激活数
spring.redis.pool.max-idle=8 # 连接池配置，最大空闲数
spring.redis.pool.max-wait=-1 #  连接池配置，最大等待时间
spring.redis.pool.min-idle=0 # 连接池配置，最小空闲活动连接数
```

调用的地方可以直接引用redisTemplate进行使用了。这时候启动pool的很多设置，如果不配置pool的一些相关参数，我们看源码的话，也会发现启动JedisPoolConfig里面的默认配置（源码分析里面的代码片段二里面的内容）。

### 2．实例

DEMO1：

```
//例如某个 Service 里面只需要引用 RedisTemplate 类即可：
@Autowired
private static RedisTemplate redisTemplate;
//某个 service 方法中，直接调用 redisTemplate 操作 redis 的 set 集合，储存 key 和 value
public Object cacheAround(String key,String value) throws Throwable {
    ....
    //直接调用 redisTemplate 操作 redis 的 set 集合，储存 key 和 value
    redisTemplate.opsForSet().add(key,value);
    //这里不需要，关心 redisTemplate 里面配置的是连接池，还是哨兵，还是 cluster。
    .....
}
```

## 3．sentinel哨兵的高可用配置方法

我们可以看RedisProperties的Sentinel类得出如下配置方式：只需要在application.properties里面配置如下内容即可。

```
   spring.redis.sentinel.master= redis_master_name #master 名字
   spring.redis.sentinel.nodes= 127.0.0.1:63791,127.0.0.1:63792  #我们配置多个哨兵，用","分割即可
```

使用的地方保持不变，这就体现了Spring的大牛们超强的封装思想，当改变Redis的Server使用方式的时候，让redisTemplate使用的地方不受任何影响，这里体现了Java的封装思想（因为我们只需要改变配置文件即可，调用的地方不需要发生任何改变，如上面的实例DEMO1）。

## 4．Cluster分布式切片的配置方法

RedisProperties的Cluster类得出如下配置方式：只需要在application.properties里面配置如下内容即可。

```
   spring.redis.cluster.max-redirects=3 # Maximum number of redirects to follow when executing commands across the cluster.
   spring.redis.cluster.nodes= 127.0.0.1:6379,127.0.0.1:6376,127.0.0.1:6378# Comma-separated list of "host:port" pairs to bootstrap from.
```

# 10.2.3　第3步：调用的地方

DEMO2：通过RedisTemplate直接操作key/value。

```
   //声明
   @Resource(name = "redisTemplate")
   private RedisTemplate<String, String> template;

   //调用方法
   template.opsForValue().set("key","value");
```

DEMO3：通过注入 resdisTemplate 操作 ValueOperations。

```
    //RedisTemplate 还提供了对应的*OperationsEditor,用来通过 RedisTemplate 直接注入对应
的 Operation。
    //声明
    @Resource(name = "redisTemplate")
    private ValueOperations<String, Object> vOps;

    //调用方法
    vOps.set("key","value");
```

DEMO4：通过注解HashOperations操作Hash数据结构。

```
    //注入 HashOperations 对象
    @Resource(name = "redisTemplate")
    private HashOperations<String,String,Object> hashOps;

    //具体调用
    Map<String,String> map = new HashMap<String, String>();
    map.put("value","code");
    map.put("key","keyValue");
    hashOps.putAll("hashOps",map);
```

DEM5：通过template操作Hash数据结构。

```
    //注入 RedisTemplate 对象
    @Resource(name = "redisTemplate")
    private RedisTemplate<String, String> template;

    //具体调用
    Map<String,String> map = new HashMap<String, String>();
    map.put("value","code");
    map.put("key","keyValue");
    template.opsForHash().putAll("hashOps",map);
```

## 10.2.4 第4步：总结

我们总结一下Spring Boot的配置方法：

- 引入spring-boot-starter-data-redis.jar包的依赖。
- 修改application.properties文件的三种配置方法即可。
- 直接调用RedisTemplate进行Redis的相关操作。

## 10.2.5 主要的几个类&简单用法介绍

看一下我们要关心的几个重要的类图，如图10-2、图10-3所示。

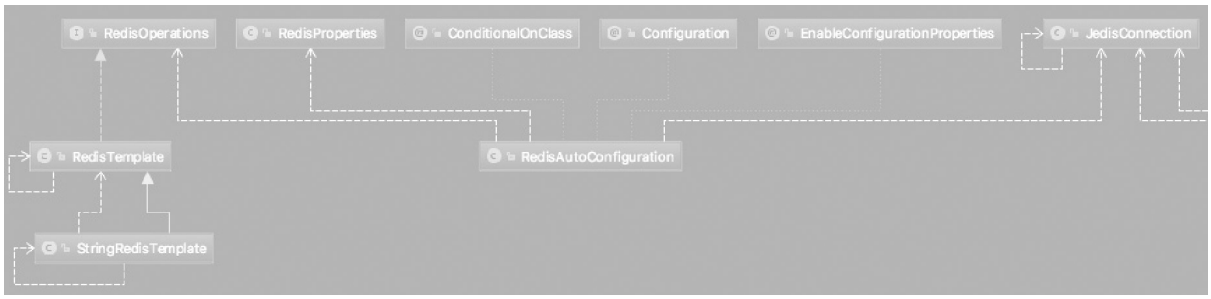



图10-2

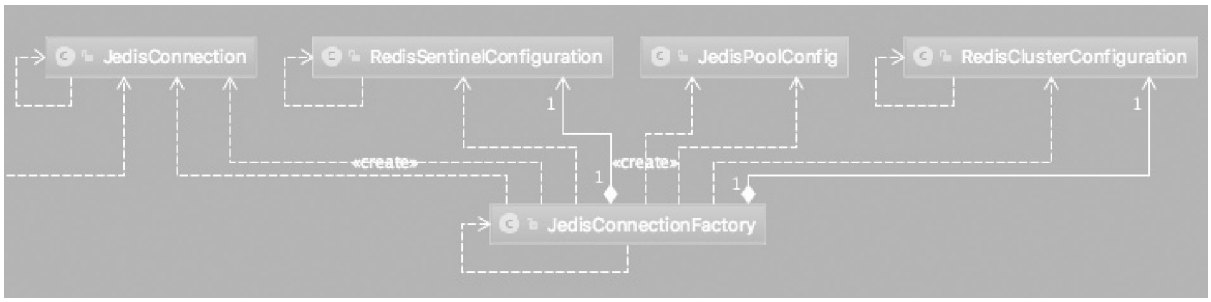



图10-3

（1）JedisConnectionFactory里面依赖了JedicConnection、JedisPoolConfig、RedisSentine Configuration、RedisCLusterConfiguration 4种配置方法。

（2）RedisAutoConfiguration自动加载了RedisProperties的配置文件。

（3）RedisTemplate抽象保证了Redis相关的Operations方法。

（4）StringRedisTemplate继承和扩展了RedisTemplate，为我们提供了一种扩展思路。

我们主要关心的类RedisTemplate的API简单介绍：

```
redisTemplate.opsForValue();//操作字符串
redisTemplate.opsForHash();//操作 hash
redisTemplate.opsForList();//操作 list
redisTemplate.opsForSet();//操作 set
redisTemplate.opsForZSet();//操作有序 set
```

StringRedisTemplate与RedisTemplate封装的Reids操作要比我

们第2节讲的自己调用Jedis的API的方式更优雅了一步。

而StringRedisTemplate与RedisTemplate对应API请仔细查看此文档Spring Data Redis官方操作手册，这里不是本节的重点介绍对象了，不再赘述。

## 10.3 Spring Data Redis结合Spring Cache配置方法

### 10.3.1 Spring Cache介绍

#### 1．简单介绍

我们都知道有经验的程序员，如果配置缓存，当用到Jedis的时候会自己配置一些注解，并且利用@Aspect。而Spring的Cache就帮我们做了一些这样的事情。

Spring 3.1 之后引入了基于注释（annotation）的缓存（cache）技术，它本质上不是一个具体的缓存实现方案（如EHCache或者Redis），而是一个对缓存使用的抽象，通过在既有代码中添加少量它定义的各种annotation，即能够达到缓存方法的返回对象的效果。

Spring的缓存技术还具备相当的灵活性，不仅能够使用SpEL（Spring Expression Language）来定义缓存的key和各种condition，还提供开箱即用的缓存临时存储方案，也支持和主流的专业缓存，例如Redis、EHCache集成。

Spring 4.1之后又改进了很多，Spring Cache属于Spring framework的一部分，在如图10-4所示的包里面。

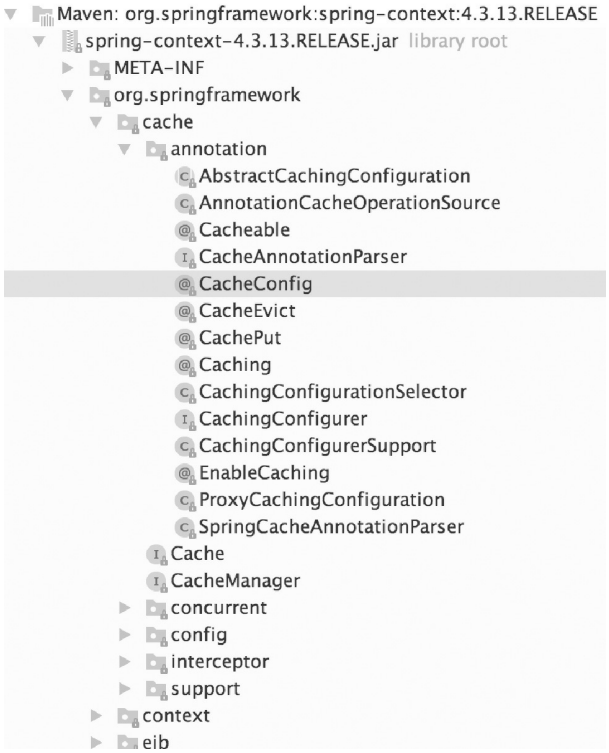



图10-4

## 2．Spring Cache里面的主要的类

Spring定义了 org.springframework.cache.CacheManager和org.springframework.cache。Cache接口用来统一不同的缓存技术。我们打开CachingConfigurer也会发现主要的几个东西：

```
public interface CachingConfigurer {
   CacheManager cacheManager();
   CacheResolver cacheResolver();
   KeyGenerator keyGenerator();
   CacheErrorHandler errorHandler();
}
```

Cache接口包含缓存的各种操作（增加、删除、获得缓存）。Spring的Framework又做了好多，Cache的默认的缓存的实现如图10-5所示。

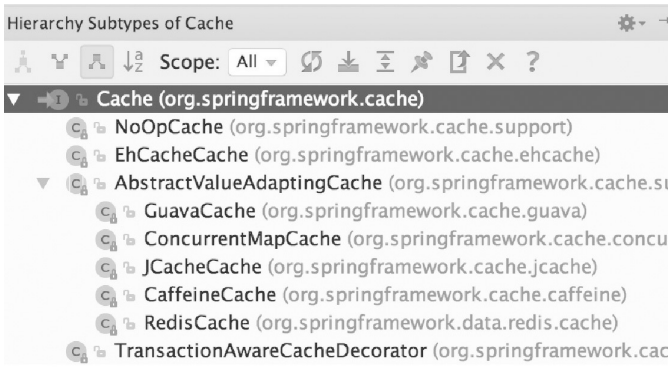



图10-5

其中包括：Ehcache、GuavaCache等很多流行的Cache的实现，而RedisCache的实现我们后面再讲。我们打开CacheManager看一下：

```
public interface CacheManager {
   Cache getCache(String name);
   Collection<String> getCacheNames();
}
```

所以 CacheManager是Spring提供的各种缓存技术抽象接口，通过它管理Spring Framework里面默认实现的CacheManager，内容如图10-6所示。

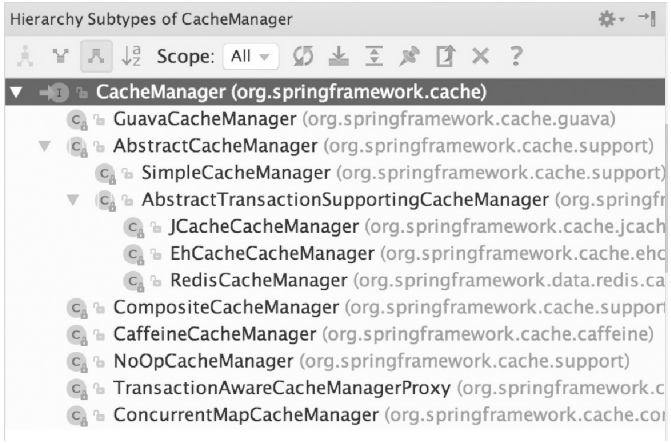



图10-6

CacheResolver解析器，用于根据实际情况来动态解析使用哪个Cache，默认实现的如图10-7所示。

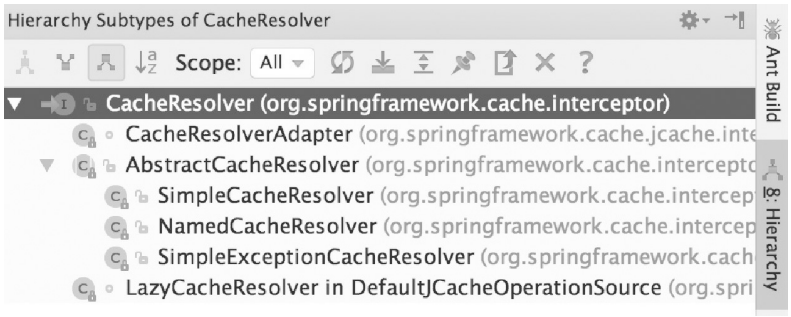



图10-7

KeyGenerator，当我们使用Cache注解的时候，默认key的生成规则如图10-8所示。

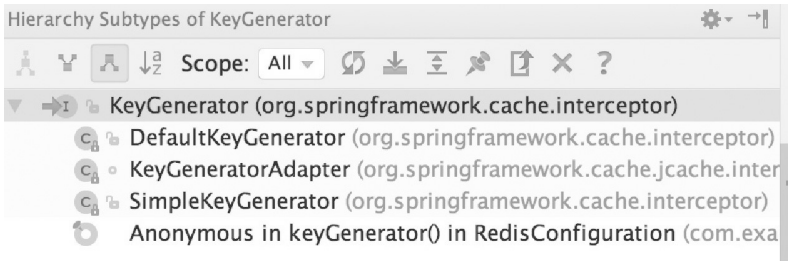



图10-8

## 3．Spring Cache里面的主要的注解

@Cacheable　应用到读取数据的方法上，即可缓存的方法，如查找方法：先从缓存中读取，如果没有再调用方法获取数据，然后把数据添加到缓存中。

```
   public @interface Cacheable {
      @AliasFor("cacheNames")
      String[] value() default {};
   //cache 的名字。可以根据名字设置不同 cache 处理类。Redis 里面可以根据 cache 名字设置不同的
失效时间。
      @AliasFor("value")
      String[] cacheNames() default {};
   //缓存的 key 的名字，支持 spel
      String key() default "";
   //key 的生成策略，不指定可以用全局的默认的。
      String keyGenerator() default "";
      //客户选择不同的 CacheManager
      String cacheManager() default "";
      //配置不同的 cache resolver
      String cacheResolver() default "";
      //满足什么样的条件才能被缓存，支持 SpEL，用 SpEL 可以取到方法名和方法的参数
      String condition() default "";
   //排除哪些返回结果不加入到缓存里面去，支持 SpEL，实际工作中常见的是 result ==null 等
      String unless() default "";
      //是否 同步读取缓存，更新缓存
      boolean sync() default false;
   }
```

例子：

```
   @Cacheable(cacheNames="book", condition="#name.length() < 32",
unless="#result.hardback")
      public Book findBook(String name)

   @CachePut
```

调用方法时会自动把相应的数据放入缓存，它与@Cacheable不同的是所有注解的方法每次都会执行，一般配置在Update和Insert方法上。看了一下源码里面的字段和用法与@Cacheable 相同，只是使用场景不一样。

@CacheEvict：删除缓存，一般配置在删除方法上面。

```
   public @interface CacheEvict {
   //与@Cacheable 相同的部分就不重复叙述了。
   ......
   //是否删除所有的实体对象
      boolean allEntries() default false;
      //是否方法执行之前执行。默认在方法调用成功之后删除
      boolean beforeInvocation() default false;
   }
```

@Caching：所有Cache注解的组合配置方法，源码如下。

```
public @interface Caching {
   Cacheable[] cacheable() default {};
   CachePut[] put() default {};
   CacheEvict[] evict() default {};
}
```

- @CacheConfig：全局Cache配置。
- @EnableCaching：开启SpringCache的默认配置。

## 10.3.2 Spring Boot快速开始Demo

我们通过一个快速实例来体会一下Spring Cache是什么东西，步骤如下。

（1）第一步：pom.xml添加Spring Boot的jar依赖。

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
```

（2）第二步：添加@EnableCaching注解开启Caching，实例如下。

```
@SpringBootApplication
@EnableJpaRepositories
@EnableCaching
public class RedisApplication {
   public static void main(String[] args) {
      SpringApplication.run(RedisApplication.class, args);
   }
}
```

（3）第三步：使用的地方直接用@Cacheable、@CachePut等注解即可，实例如下。

```
@Controller
@RequestMapping("hello")
public class UserInfoController {
    @Autowired
    private UserInfoService userInfoService;
    @RequestMapping("users")
    @ResponseBody
    @Cacheable(value = "user")
    public List<UserInfoEntity> findAll(){
        System.out.println("user........");
        return userInfoService.findAll();
    }
}
```

结果：当我们请求第二次的时候就不会进Controller的这个方法里面了。此处作者只是举个例子，实际工作中，配置在Service层的场景比较多。

### 10.3.3 Spring Boot Cache实现过程解析

#### 1．spring.factories

（1）我们都知道当引入 SpringBoot 的时候就会多一个spring-boot-autoconfigure 的 jar，而此 Jar 里面 auto config和加载很多相关的类。可以通过打开其包下面的 spring.factories文件，可以看到SpringBoot会默认加载org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration配置文件。

（2）spring-boot-starter-cache这个jar还会帮我们加载Spring Cache所需要的其他jar包，如spring-context和spring-context-support，是Spring Cache的核心jar包。

#### 2．CacheAutoConfiguration 关键源码解读

```
   @Configuration
   @ConditionalOnClass(CacheManager.class)
   @ConditionalOnBean(CacheAspectSupport.class)
   @ConditionalOnMissingBean(value = CacheManager.class, name = "cacheResolver")
   @EnableConfigurationProperties(CacheProperties.class)
   @AutoConfigureBefore(HibernateJpaAutoConfiguration.class)
   @AutoConfigureAfter({ CouchbaseAutoConfiguration.class,
HazelcastAutoConfiguration.class,
         RedisAutoConfiguration.class })
   @Import(CacheConfigurationImportSelector.class)
   public class CacheAutoConfiguration {
      static final String VALIDATOR_BEAN_NAME = "cacheAutoConfigurationValidator";
      @Bean
      @ConditionalOnMissingBean
      public CacheManagerCustomizers cacheManagerCustomizers(
            ObjectProvider<List<CacheManagerCustomizer<?>>> customizers) {
         return new CacheManagerCustomizers(customizers.getIfAvailable());
      }
   ....}
```

- 第一件事是通过@Conditional来判断是否满足条件进而加载Cache的配置文件。
- 第二件事情是留下了很多定义和扩展的口子，如Reids，后面章节讲。
- 第三件事情是寻找默认的@cache 的处理方法。

## 3．CacheConfigurationImportSelector

CacheAutoConfiguration里面还有一个关键类就是CacheConfigurationImportSelector。

```
   static class CacheConfigurationImportSelector implements ImportSelector {
      @Override
      public String[] selectImports(AnnotationMetadata importingClassMetadata) {
         CacheType[] types = CacheType.values();
         String[] imports = new String[types.length];
         for (int i = 0; i < types.length; i++) {
            imports[i] = CacheConfigurations.getConfigurationClass(types[i]);
         }
         return imports;
      }
   }
```

CacheType的内容如下：

```
package org.springframework.boot.autoconfigure.cache;
public enum CacheType {
    GENERIC,
    JCACHE,
    EHCACHE,
    HAZELCAST,
    INFINISPAN,
    COUCHBASE,
    REDIS,
    CAFFEINE,
    /** @deprecated */
    @Deprecated
    GUAVA,
    SIMPLE,
    NONE;

    private CacheType() {
    }
}
```

所以，通过这两段代码，当我们没有显式手动地指定CacheManager或者CacheResolver的时候，Spring Boot Cache会按照以下顺序查找Cache的默认实现者，并会自动导入各大提供商的config。

- Generic
- JCache (JSR-107) (EhCache 3, Hazelcast, Infinispan, etc)
- EhCache 2.x
- Hazelcast
- Infinispan
- Couchbase
- Redis
- Caffeine
- Guava (deprecated)
- Simple

所以这里要注意下，默认情况下Spring的context-support里面至少是有了GUAVA和SIMPLE相关的自动Cache条件，看了一下源码都仍在Java自己的JVM中，用ConcurrentHashMap的类进行储存的。

所以你会发现我们什么都没有配置，直接开启Cache和引入相关的jar就可以直接实现Cache的行为。

在实际场景中有些小项目，如果只是临时的方案，做此Aplication的临时缓存，这种方式其实是可以考虑一下的，不一定要用很重的Redis分布式缓存。

### 10.3.4 Cache和Spring Data Redis结合快速开始

基于Spring Boot的配置为例来让Spring Cache和Spring Data Redis结合使用。不同的Spring Boot版本，可能源码或者细节有点区别，但是基本思路和思想是不变的。

（1）pom.xml里面的配置，引入Spring Boot，以1.5.9为例：

```
    <parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>1.5.9.RELEASE</version>
   <relativePath/> <!-- lookup parent from repository -->
</parent>
```

（2）添加 spring-boot-starter-cache 和 spring-boot-starter-data-redis：

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
```

（3）在application.properties里面做Redis相关配置，如下：

```
spring.redis.host=127.0.0.1
spring.redis.port=6379
spring.redis.timeout=6000
spring.redis.pool.max-active=8
spring.redis.pool.max-idle=8
spring.redis.pool.max-wait=-1
spring.redis.pool.min-idle=0
```

通过第三节可以知道，我们这里默认用Redis的pool的配置方式。

（4）添加@EnableCaching注解，开启cache。

```
package com.example.redis;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@SpringBootApplication
@EnableJpaRepositories
@EnableCaching
public class RedisApplication {
    public static void main(String[] args) {
      SpringApplication.run(RedisApplication.class, args);
   }
}
```

（5）直接使用Spring Data Cache的注解即可。

如：我们在Controller里面调用JPA的方法上添加@Cacheable即可使用。

```
package com.example.redis.controller;

import com.example.redis.dao.UserInfoEntity;
import com.example.redis.service.UserInfoService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

import java.util.List;

@Controller
@RequestMapping("hello")
public class UserInfoController {
    @Autowired
    private UserInfoService userInfoService;

    @RequestMapping("users")
    @ResponseBody
    @Cacheable(value = "user",cacheManager = "redisCacheManagerString")
    public List<UserInfoEntity> findAll(){
        System.out.println("user........");
        return userInfoService.findAll();
    }
}
```

（6）这时候已经配置成功，启动我们的Application，我们验证一下结果。

- curl http://127.0.0.1:8080/hello/users 就会发现，第一次会进入到Controller的method里面，第二次就不进去了。
- 我们打开redis-client可以看到，Reids的Server端已经有我们的缓存了。

（7）配置的核心和关键点。

- Spring Cache 和 Spring Data Redis 两个jar的引入。
- 在application.properties配置我们前面介绍Spring Data Redis的正确配置方法即可。
- 在用到的地方直接用我们前面讲到的Cacheable相关的配置即

可。

- 也可以在application.properties指定spring.cache.type=redis。改变Cache的默认行为，让其用Redis来实现。

最后，会发现代码比起原始的配置变得优雅很多。

## 10.3.5 Spring Boot实现过程

当我们引入spring-boot-starter-data-redis的时候，前面讲过，会自动把Spring Data Redis 的jar也加入进来，同时会激活RedisCacheConfiguration，把RedisCacheManager默认加载进来。

当我们引入spring-boot-starter-cache的时候，前面讲过，会自动加载CacheAutoConfiguration，并且里面CacheType有顺序，这时候会把RedisCacheManager激活。

```
/**
 * Bean used to validate that a CacheManager exists and provide a more meaningful
 * exception.
 */
static class CacheManagerValidator {

    @Autowired
    private CacheProperties cacheProperties;

    @Autowired(required = false)
    private CacheManager cacheManager;

    @PostConstruct
    public void checkHasCacheManager() {
        Assert.notNull(this.cacheManager,
        "No cache manager could "
              + "be auto-configured, check your configuration (caching "
              + "type is '" + this.cacheProperties.getType() + "')");
    }

}
```

当我们打断点在这个类上的时候，会发现cacheManager会变成

RedisCacheManager，而不是默认的Cache Manager。

## 1．实际工作的正确姿势

而实际生产环境可能不像Demo这么随意，接下来说一下都有哪些自定义场景，如何自定义。

先来分析一下Spring Boot的源码。

- 我们知道关键类CacheConfigurationImportSelector，找到CacheConfigurations关键代码如下：

```
   final class CacheConfigurations {

      private CacheConfigurations() {
      }

      static {
         Map<CacheType, Class<?>> mappings = new HashMap();
         mappings.put(CacheType.GENERIC, GenericCacheConfiguration.class);
         mappings.put(CacheType.EHCACHE, EhCacheCacheConfiguration.class);
         mappings.put(CacheType.HAZELCAST, HazelcastCacheConfiguration.class);
         mappings.put(CacheType.INFINISPAN,
InfinispanCacheConfiguration.class);
         mappings.put(CacheType.JCACHE, JCacheCacheConfiguration.class);
         mappings.put(CacheType.COUCHBASE, CouchbaseCacheConfiguration.class);
         mappings.put(CacheType.REDIS, RedisCacheConfiguration.class);
         mappings.put(CacheType.CAFFEINE, CaffeineCacheConfiguration.class);
         addGuavaMapping(mappings);
         mappings.put(CacheType.SIMPLE, SimpleCacheConfiguration.class);
         mappings.put(CacheType.NONE, NoOpCacheConfiguration.class);
         MAPPINGS = Collections.unmodifiableMap(mappings);
      }
   }
```

- 通过上面代码，我们发现Redis的Type类型的Cache调用的是RedisCacheConfiguration。

```
    @Configuration
    @AutoConfigureAfter({RedisAutoConfiguration.class})
    @ConditionalOnBean({RedisTemplate.class})
    @ConditionalOnMissingBean({CacheManager.class})
    @Conditional({CacheCondition.class})
    class RedisCacheConfiguration {
        private final CacheProperties cacheProperties;
        private final CacheManagerCustomizers customizerInvoker;

        RedisCacheConfiguration(CacheProperties cacheProperties,
CacheManagerCustomizers customizerInvoker) {
            this.cacheProperties = cacheProperties;
            this.customizerInvoker = customizerInvoker;
        }

        @Bean
        public RedisCacheManager cacheManager(RedisTemplate<Object, Object>
redisTemplate) {
            RedisCacheManager cacheManager = new RedisCacheManager(redisTemplate);
            cacheManager.setUsePrefix(true);
            List<String> cacheNames = this.cacheProperties.getCacheNames();
            if (!cacheNames.isEmpty()) {
                cacheManager.setCacheNames(cacheNames);
            }
            return
(RedisCacheManager)this.customizerInvoker.customize(cacheManager);
        }
    }
```

通过RedisCacheConfiguration其实可以发现很多，如可以自定义Redis的Configuration和自定义Redis的CacheManager。

- CachingConfigurer是Spring为我们预留的自定义接口，打开它的默认实现类CachingConfigurerSupport。

```
public class CachingConfigurerSupport implements CachingConfigurer {
    @Override
    public CacheManager cacheManager() {
        return null;
    }

    @Override
    public KeyGenerator keyGenerator() {
        return null;
    }

    @Override
    public CacheResolver cacheResolver() {
        return null;
    }

    @Override
    public CacheErrorHandler errorHandler() {
        return null;
    }

}
```

通过继承此类就可以实现自定义cacheManager和KeyGenerator、CacheResolver、CacheErrorHandler。

## 2．实现自定义的配置

（1）新建RedisConfiguration，并且配置两个CacheManager。

```
/**
 * 自定义 RedisConfiguration，扩展 Redis 和 Cache 的默认行为
 *     通过@AutoConfigureAfter 使其在(RedisAutoConfiguration.class)后面加载
 *     我们也用 CacheProperties 的配置，这样不需要我们另行建一套配置文件
 * @author jack
 */
```

```
    @Configuration
    @AutoConfigureAfter(RedisAutoConfiguration.class)
    @EnableConfigurationProperties(CacheProperties.class)
    public class RedisConfiguration extends CachingConfigurerSupport {
        /**
         * 我们自定义的时候也要基于 cacheProperties
         */
        @Autowired
        private CacheProperties cacheProperties;
        /**
         * 为全局的 redisTemplate 配置一个默认的 RedisCacheManager,
         *    并根据一些 cache name 设置不同过期时间
         * @param redisTemplate
         * @return
         */
        @Bean(name = "redisCacheManager")
        @Primary
        public RedisCacheManager cacheManager(RedisTemplate<Object, Object>
redisTemplate) {
            RedisCacheManager cacheManager = new RedisCacheManager(redisTemplate);
            cacheManager.setUsePrefix(true);
            //可以设置不同的 cache 的 name 不同的过期时间
            Map<String,Long> expries= Maps.newHashMap();
            expries.put("user:",10*60L);
            cacheManager.setExpires(expries);
            List<String> cacheNames = cacheProperties.getCacheNames();
            if (!cacheNames.isEmpty()) {
                cacheManager.setCacheNames(cacheNames);
            }

            return cacheManager;
        }

        /**
         * 同样我们也可以改变 stringRedisTemplate 的默认 RedisCacheManager
         * @param stringRedisTemplate
         * @return
         */
        @Bean(name = "redisCacheManagerString")
        public RedisCacheManager cacheManagerString(StringRedisTemplate
stringRedisTemplate) {
            RedisCacheManager cacheManager = new
RedisCacheManager(stringRedisTemplate);
            cacheManager.setUsePrefix(true);
            //设置默认过期时间
```

```
        cacheManager.setDefaultExpiration(10*60);
        //cache name
        List<String> cacheNames = cacheProperties.getCacheNames();
        if (!cacheNames.isEmpty()) {
            cacheManager.setCacheNames(cacheNames);
        }

        return cacheManager;
    }
}
```

我们都知道RedisTemplae和StringRedisTemplate默认的：

```
setKeySerializer(stringSerializer);
setValueSerializer(stringSerializer);
setHashKeySerializer(stringSerializer);
setHashValueSerializer(stringSerializer);
```

都是不一样的，所以我们分别配置了两个 CacheManager，以至于我们配置
@Cacheable(value="user",cacheManager="redisCacheManagerStrin
可以选择用哪个cacheManager。这里只是列举了工作频率最高的自定义配置，通过自定义CacheManager实现如下操作：

- defaultExpiration：默认过期时间是不一样的。
- 不同的Cache的Name我们也可以定制化，实现不一样的过期时间。
- 不同的Cache我们可以选择不同的RedisTemplate。

（2）自定义KeyGenerator覆盖默认的Cache key生成规则，只需要在RedisConfiguration中增加如下配置即可。

```
/**
 * 覆盖默认的 key 的生成器
 *
 * @return
 */
@Override
@Bean
public KeyGenerator keyGenerator() {
    return new KeyGenerator() {
        @Override
        public Object generate(Object o, Method method, Object... objects) {
            // This will generate a unique key of the class name, the method name,
            // and all method parameters appended.
            StringBuilder sb = new StringBuilder("Jack-Test:");
            sb.append(o.getClass().getName());
            sb.append(method.getName());
            for (Object obj : objects) {
                sb.append(obj.toString());
            }
            return sb.toString();
        }
    };
}
```

（3）修改RedisTemplate和StringRedisTemplate的KeySerializer和ValueSerializer默认规则。

```
/**
 * 覆盖默认的 redisTemplate，修改 KeySerializer 为 String 的这样，key 值我们能看
 *
 * @param redisConnectionFactory
 * @return
 * @throws UnknownHostException
 */
@Bean
@ConditionalOnMissingBean(name = "redisTemplate")
public RedisTemplate<Object, Object> redisTemplate(
    RedisConnectionFactory redisConnectionFactory)
    throws UnknownHostException {
    RedisTemplate<Object,Object>template=new RedisTemplate<Object,Object>();
    template.setConnectionFactory(redisConnectionFactory);
    template.setKeySerializer(new StringRedisSerializer());
    return template;
}

/**
 * 覆盖默认的 StringRedisTemplate
 *      修改 KeySerializer 为 String 的这样，key 值我们能看
 *      修改 ValueSerializer 为 Jackson，这样便于我们监控和查看
 * @param redisConnectionFactory
 * @return
 * @throws UnknownHostException
 */
@Bean
@ConditionalOnMissingBean(StringRedisTemplate.class)
public StringRedisTemplate stringRedisTemplate(
    RedisConnectionFactory redisConnectionFactory)
    throws UnknownHostException {
    StringRedisTemplate template = new StringRedisTemplate();
    template.setConnectionFactory(redisConnectionFactory);
    template.setKeySerializer(new StringRedisSerializer());
    template.setValueSerializer(new GenericJackson2JsonRedisSerializer());
    return template;
}
```

（4）使用的地方，其controller中的写法，配置如下：

```
@RequestMapping("users")
@ResponseBody
@Cacheable(value = "user",cacheManager = "redisCacheManagerString")
public List<UserInfoEntity> findAll(){
    System.out.println("user........");
    return userInfoService.findAll();
}
```

（5）我们调用http://127.0.0.1:8080/hello/users  得到的结

果如图10-9所示。


图10-9

到此完美实现自定义过程。

# 第11章
# SpEL表达式讲解

博观而约取，厚积而薄发。

——苏轼

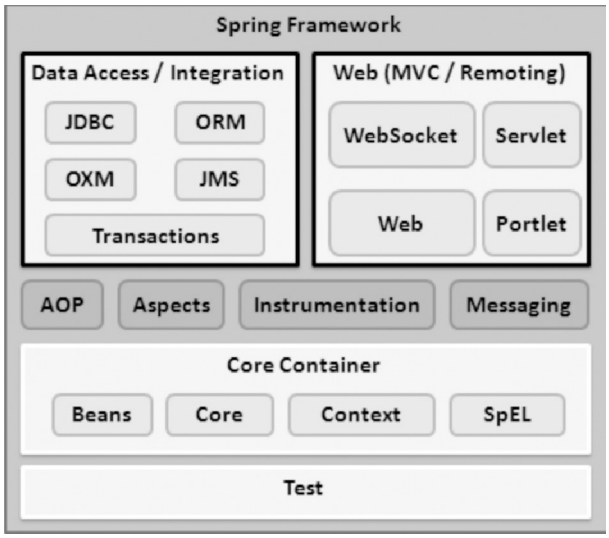



图11-1

## 11.1 SpEL介绍

SpEL是Spring Expression Language的简称。SpEL的诞生是为了给Spring社区提供一种能够与Spring生态系统所有产品无缝对接、一站式支持的表达式语言。它的语言特性由Spring生态系统的实际项目需求驱动而来。

### 11.1.1 SpEL主要特点

（1）SpEL是一种功能强大的表达式语言、用于在运行时查询和操作对象的动态语言。语法上类似于Unified EL，但提供了更多的特

性，特别是方法调用和基本字符串模板函数。

（2）有点类似目前已经有的许多其他的Java表达式语言，例如OGNL、 MVEL和JBoss EL，也有点类似Freemark表达式语言。

（3）SpEL在Spring产品中是作为表达式求值的核心基础模块，贯穿于Spring的整个项目模块，几乎每个Spring模块都会依赖到SpEL。但是其本身也可以脱离Spring独立使用，spring-expression*.jar可以独立被其他任何项目开源地引用和使用。

## 11.1.2 使用方法

SpEL的使用方法有三种：一种是XML中，一种是注解中，这两种使用SpEL的时候用${#.....}作为模板语言表上符号，后面会有详解实现原理，第三种是使用ExpressionParser.class直接操作SpEL表达式的实现类的API。

（1）XML中使用方法、类中的property或者构造方法用到#{}SpEL表达式。

```
    <!--给 NumberGuess 类中的 randomNumber 属性用 SpEL 表达式赋值一个随机数-->
    <bean id="numberGuess" class="org.spring.samples.NumberGuess">
        <property name="randomNumber" value="#{ T(java.lang.Math).random() *
100.0 }"/>
    </bean>
          <!--systemProperties 是系统预定义的，取一个变量的值赋值给 defaultLocale-->
    <bean id="taxCalculator" class="org.spring.samples.TaxCalculator">
        <property name="defaultLocale"
value="#{ systemProperties['user.region'] }"/>
    </bean>
    <!--直接引用上面的 NumberGuess 的 randomNumber 进行赋值-->
    <bean id="shapeGuess" class="org.spring.samples.ShapeGuess">
        <property name="initialShapeSeed" value="#{ numberGuess.randomNumber }"/>
    </bean>
```

（2）@注解中使用方法。

我们先以@Value注解为例，能被使用在fields或者setMethods或者method/constructor参数指定的默认值中，在运行的时候动态取

值。

```
   /**
    * 给一个字段设置默认值
    */
   public class FieldValueTestBean {
      @Value("#{ systemProperties['user.region'] }")
      private String defaultLocale;
   }

   /**
    * 给 set 的方法上设置默认值
    */
   public class PropertyValueTestBean {
      private String defaultLocale;
      @Value("#{ systemProperties['user.region'] }")
      public void setDefaultLocale(String defaultLocale) {
         this.defaultLocale = defaultLocale;
      }
   }

   /**
    * 方法的参数上赋值
    */
   public class SimpleMovieLister {
      private String defaultLocale;
      @Autowired
      public void configure(@Value("#{ systemProperties['user.region'] }") String
defaultLocale) {
         this.defaultLocale = defaultLocale;
      }
      // ...
   }
```

（3）SpEL Java API使用方法。

我们先通过SpelExpressionParser简单地对字符串进行操作，详细的用法下面详解。

```
   public static void main(String[] args) {
      ExpressionParser parser = new SpelExpressionParser();
      Expression exp = parser.parseExpression("'Hello World'.concat('!')");
      String message = (String) exp.getValue();
      System.out.println(message);
   }
```

控制台输出如下：

```
Hello World!
```

## 11.2 SpEL的基础语法

由于Spring Boot的使用场景越来越多，而且后面的编程中大部分都是注解这种形式，我们就通过注解来介绍一下SpEL的基本操作语法，如表11-1所示。

表11-1 SpEL的基本操作语法

| 类型 | 操作符 |
| --- | --- |
| 逻辑运算 | +、-、*、/、%、^、div、mod |
| 逻辑比较符号 | <、>、==、!=、<=、>=、lt、gt、eq、ne、le、ge |
| 逻辑关系 | and、or、not、&&、\|\|、! |
| 三元表达式 | ?: |
| 正则表达式 | matches |

SpEL表达式默认以 # 开始，以大括号进行包住，如：#{expression}。注意要与Spring中的properties 进行区别，properties相关的表达式是以 $ 开始的大括号进行包住的，如：${property.name}。Property placeholders不能包含SpEL表达式，但是SpEL表达式可以包含property的引用，如：

```
#{${someProperty} + 2}
```

如果someProperty=2，那么效果将是#{ 2 + 2}，最终的结果将是4。

### 11.2.1 逻辑运算操作

```
@Value("#{19 + 1}") // 20
private double add;

@Value("#{'String1 ' + 'string2'}") // "String1 string2"
private String addString;

@Value("#{20 - 1}") // 19
private double subtract;

@Value("#{10 * 2}") // 20
private double multiply;

@Value("#{36 / 2}") // 19
private double divide;

@Value("#{36 div 2}") // 18, the same as for / operator
private double divideAlphabetic;

@Value("#{37 % 10}") // 7
private double modulo;

@Value("#{37 mod 10}") // 7, the same as for % operator
private double moduloAlphabetic;

@Value("#{2 ^ 9}") // 512
private double powerOf;

@Value("#{(2 + 2) * 2 + 9}") // 17
private double brackets;
```

**提示**

== /、mod == %、“+”符号也可以用来连接字符串。

## 11.2.2 逻辑关系比较

```
@Value("#{1 == 1}") // true
private boolean equal;

@Value("#{1 eq 1}") // true
private boolean equalAlphabetic;

@Value("#{1 != 1}") // false
private boolean notEqual;

@Value("#{1 ne 1}") // false
private boolean notEqualAlphabetic;

@Value("#{1 < 1}") // false
private boolean lessThan;

@Value("#{1 lt 1}") // false
private boolean lessThanAlphabetic;

@Value("#{1 <= 1}") // true
private boolean lessThanOrEqual;

@Value("#{1 le 1}") // true
private boolean lessThanOrEqualAlphabetic;

@Value("#{1 > 1}") // false
private boolean greaterThan;

@Value("#{1 gt 1}") // false
private boolean greaterThanAlphabetic;

@Value("#{1 >= 1}") // true
private boolean greaterThanOrEqual;

@Value("#{1 ge 1}") // true
private boolean greaterThanOrEqualAlphabetic;
```

## 提示

所有的逻辑关系比较符都有一个缩写的别称 (<、<=、>、>=)，对应的别称分别为 lt (less than)、le (less than or equal)、gt (greater than)、ge(greater than or equal)。

### 11.2.3 逻辑关系

```
@Value("#{250 > 200 && 200 < 4000}") // true
private boolean and;

@Value("#{250 > 200 and 200 < 4000}") // true
private boolean andAlphabetic;

@Value("#{400 > 300 || 150 < 100}") // true
private boolean or;

@Value("#{400 > 300 or 150 < 100}") // true
private boolean orAlphabetic;

@Value("#{!true}") // false
private boolean not;

@Value("#{not true}") // false
private boolean notAlphabetic;
```

### 11.2.4 三元表达式& Elvis运算符

正常的三元表达式：

```
    @Value("#{2 > 1 ? 'a' : 'b'}") // "b"
    private String ternary;

    @Value("#{someBean.someProperty != null ? someBean.someProperty : 'default'}")
    private String ternary;

    Elvis 运算符是三元表达式简写
    /**
    * 如果 someProperty 为 null 则返回 default 值。
    */
    @Value("#{someBean.someProperty ?: 'default'}")
    private String elvis;

    /**
    * 如果系统属性 pop3.port 已定义会直接注入，如果未定义，则返回默认值25.
    */
    @Value("#{systemProperties['pop3.port'] ?: 25}")
    private Integer port;

    /**
    * 还可以用于安全引用运算符主要为了避免空指针，源于 Groovy 语言。很多时候你引用一个对象的方法或者属性时都需要做非空校验。为了避免此类问题、使用安全引用运算符只会返回 null 而不是抛出一个异常。
    */
    @Value("#{someBean?:.someProperty}") // 如果 someBean 不为 null 则返回 someProperty 值。
    private String someProperty;
```

## 11.2.5 正则表达式的支持

```
    @Value("#{'100' matches '\\d+' }") // true
    private boolean validNumericStringResult;

    @Value("#{'100fghdjf' matches '\\d+' }") // false
    private boolean invalidNumericStringResult;

    @Value("#{'valid alphabetic string' matches '[a-zA-Z\\s]+' }") // true
    private boolean validAlphabeticStringResult;

    @Value("#{'invalid alphabetic string #$1' matches '[a-zA-Z\\s]+' }") // false
    private boolean invalidAlphabeticStringResult;

    @Value("#{someBean.someValue matches '\d+'}") // true 如果 someValue 只有数字
    private boolean validNumericValue;
```

## 11.2.6 Bean的引用

在Spring Context的整个上下文中，当使用Spring的注解的时候，可以直接引用Bean中的字段和方法。

通过@Component加载Bean：

```
@Component("workersHolder")
public class WorkersHolder {
 private int capacity=1;
    private int horsePower=2;
}
```

注解中使用的地方直接#{beanName.filder}即可，如下：

```
@Value("#{workersHolder.capacity}") // 1
private Integer capacity;
@Value("#{workersHolder.horsePower}") // 2
private Integer horsePower;
```

## 11.2.7 List和Map的操作

我们通过@Component加载一个类，并且给其中的List和Map附上值。

```
@Component("workersHolder")
public class WorkersHolder {
    private List<String> workers = new LinkedList<>();
    private Map<String, Integer> salaryByWorkers = new HashMap<>();

    public WorkersHolder() {
        workers.add("John");
        workers.add("Susie");
        workers.add("Alex");
        workers.add("George");

        salaryByWorkers.put("John", 35000);
        salaryByWorkers.put("Susie", 47000);
        salaryByWorkers.put("Alex", 12000);
        salaryByWorkers.put("George", 14000);
    }
    //Getters and setters ...
}
```

SpEL直接读取Map和List里面的值：

```
@Value("#{workersHolder.salaryByWorkers['John']}") // 35000
private Integer johnSalary;

@Value("#{workersHolder.salaryByWorkers['George']}") // 14000
private Integer georgeSalary;

@Value("#{workersHolder.salaryByWorkers['Susie']}") // 47000
private Integer susieSalary;

@Value("#{workersHolder.workers[0]}") // John
private String firstWorker;

@Value("#{workersHolder.workers[3]}") // George
private String lastWorker;

@Value("#{workersHolder.workers.size()}") // 4
private Integer numberOfWorkers;
```

# 11.3 主要的类及其原理

## 11.3.1 ExpressionParser

下面代码介绍了使用SpEL API来解析字符串表达式“Hello World”的示例：

```
ExpressionParser expressionParser = new SpelExpressionParser();
Expression expression = expressionParser.parseExpression("'Hello World'");
String result = (String) expression.getValue();
```

最常用的SpEL类和接口都放在包org.springframework.expression及其子包和spel.support下。

接口ExpressionParser用来解析一个字符串表达式。在这个例子中字符串表达式为用单引号括起来的字符串。接口Expression用于对上面定义的字符串表达式求值。调用parser.parseExpression和exp.getValue分别可能抛出ParseException和EvaluationException。SpEL支持一系列广泛的特性，例如方法调用、访问属性、调用构造函数等。

接下去是一个访问JavaBean属性的例子，String类的Bytes属性通过以下的方法调用：

```
    // 调用'getBytes()'
    Expression exp = parser.parseExpression("'Hello World'.bytes");
    byte[] bytes = (byte[]) exp.getValue();
```

SpEL同时也支持级联属性调用，和标准的prop1.prop2.prop3方式是一样的，同样属性值设置也是类似的方式：

```
    ExpressionParser parser = new SpelExpressionParser();
    // 调用 'getBytes().length'
    Expression exp = parser.parseExpression("'Hello World'.bytes.length");
    int length = (Integer) exp.getValue();
```

除了使用字符串表达式，也可以调用String的构造函数：

```
    ExpressionParser parser = new SpelExpressionParser();
    Expression exp = parser.parseExpression("new String('hello
world').toUpperCase()");
    String message = exp.getValue(String.class);
```

针对泛型方法的使用，例如：public <T> T getValue(Class<T> desiredResultType)，使用这样的方法不需要将表达式的值转换成具体的结果类型。如果具体的值类型或者使用类型转换器都无法转成对应的类型，会抛出EvaluationException的异常。

## 11.3.2 root object

SpEL中更常见的用途是提供一个针对特定对象实例（叫做根对象）求值的表达式字符串。使用方法有两种，具体用哪一种要看每次调用表达式求值时相应的对象实例是否每次都会变化。

（1）通过EvaluationContext设置root object。

【示例11.1】我们从Inventor类的实例中解析name属性。

```
// 创建并设置一个 calendar 实例
GregorianCalendar c = new GregorianCalendar();
c.set(1856, 7, 9);
// 构造器参数有： name, birthday and nationality.
Inventor tesla = new Inventor("Nikola Tesla", c.getTime(), "Serbian");
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("name");
EvaluationContext context = new StandardEvaluationContext(tesla);
String name = (String) exp.getValue(context);
```

最后一行，字符串变量name将会被设置成“Nikola Tesla”。通过StandardEvaluationContext类你能指定哪个对象的“name”会被求值这种机制用于当根对象不会被改变的场景，在求值上下文中只会被设置一次。

（2）在每次调用getValue的时候被设置Root Object。

```
// 创建并设置一个 calendar 实例
GregorianCalendar c = new GregorianCalendar();
c.set(1856, 7, 9);

// 构造器参数有： name, birthday and nationality.
Inventor tesla = new Inventor("Nikola Tesla", c.getTime(), "Serbian");

ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("name");
String name = (String) exp.getValue(tesla);
```

在上面这个例子中inventor类实例tesla直接在getValue中设置，表达式解析器底层框架会自动创建和管理一个默认的求值上下文，不需要被显式声明。StandardEvaluationContext创建相对比较耗资源，在重复多次使用的场景下，内部会缓存部分中间状态加快后续的表达式求值效率。因此建议在使用过程中尽可能被缓存和重用，而不是每次在表达式求值时都重新创建一个对象。

（3）实际场景。

在很多使用场景下，理想的方式是事先配置好求值上下文，但是在实际调用中仍然可以给getValue设置一个不同的根对象。getValue

允许在同一个调用中同时指定两者，在这种场景下运行时传入的根对象会覆盖在求值上下文中事先指定的根对象。

Spring上下文的注解中，当使用SpEL的时候，其实root object更多的是Spring的Context。

### 11.3.3 EvaluationContext

EvaluationContext接口在求值表达式中需要解析属性、方法、字段的值以及类型。其默认实现类StandardEvaluationContext使用反射机制来操作对象。为获得更好的性能缓存了java.lang.reflect.Method，java.lang.reflect.Field和java.lang.reflect.Constructor实例。

在StandardEvaluationContext中，可以使用setRootObject()方法显式设置根对象，或通过构造器直接传入根对象，还可以通过调用setVariable()和registerFunction()方法指定在表达式中用到的变量和函数。变量和函数的使用在官方语言参考文档中的 Variables和Functions两章节有详细说明。使用StandardEvaluationContext还可以注册自定义的构造器解析器（ConstructorResolvers）、方法解析器（MethodResolvers）和属性存取器（PropertyAccessor）来扩展SpEL计算表达式，详见具体类的JavaDoc文档。

### 11.3.4 类型转换

SpEL默认使用Spring核心代码中的conversion service来做类型转换（org.springframework.core.convert.ConversionService）。这个类本身内置了很多常用的转换器，同时也可以扩展使用自定义的类型转换器。另外一个核心功能是它可以识别泛型，这意味着当在表达式中使用泛型类型时SpEL会确保任何处理的对象的类型正确性。

实际应用中这意味着什么？这里拿赋值来说，比如使用setValue来设置List属性。属性的类型实际上是List<Boolean>，SpEL可以识

别List中的元素类型并转换成Boolean类型。下面是示例代码：

```
    class Simple {
      public List<Boolean> booleanList = new ArrayList<Boolean>();
    }
    Simple simple = new Simple();
    simple.booleanList.add(true);
    StandardEvaluationContext simpleContext = new
StandardEvaluationContext(simple);
    // false is passed in here as a string. SpEL and the conversion service will
    // correctly recognize that it needs to be a Boolean and convert it
    parser.parseExpression("booleanList[0]").setValue(simpleContext, "false");
    // b will be false
    Boolean b = simple.booleanList.get(0);
```

## 11.3.5 SpelParserConfiguration编译器配置

可以通过使用一个解析器配置对象（org.springframework.expression.spel.SpelParserConfiguration）来配置SpEL表达式解析器。这个配置对象可以控制一些表达式组件的行为。例如：数组或者集合元素查找的时候，如果当前位置对应的对象是Null，可以通过事先配置来自动创建元素。这个在表达式多次属性链式引用的时候比较重要。在设置的数组或者List位置越界时，可以自动增加数组或者List长度来兼容。

```
    class Demo {
      public List<String> list;
    }

    // Turn on:
    // - auto null reference initialization
    // - auto collection growing
    SpelParserConfiguration config = new SpelParserConfiguration(true,true);

    ExpressionParser parser = new SpelExpressionParser(config);

    Expression expression = parser.parseExpression("list[3]");

    Demo demo = new Demo();

    Object o = expression.getValue(demo);
```

如果不配置config，由于Demo里面的list是null，运行的时候将会报错；如果配置了config，这时候demo.list会变成里面有4个对象，list自动添加empty object进去。

SpelParserConfiguration的更多参数如下：

```
   /**
    * Create a new {@code SpelParserConfiguration} instance.
    * @param compilerMode the compiler mode that parsers using this configuration
object should use
    * @param compilerClassLoader the ClassLoader to use as the basis for expression
compilation
    * @param autoGrowNullReferences if null references should automatically grow
    * @param autoGrowCollections if collections should automatically grow
    * @param maximumAutoGrowSize the maximum size that the collection can auto grow
    */
   public SpelParserConfiguration(@Nullable SpelCompilerMode compilerMode,
@Nullable ClassLoader compilerClassLoader,
        boolean autoGrowNullReferences, boolean autoGrowCollections, int
maximumAutoGrowSize) {
      this.compilerMode = (compilerMode != null ? compilerMode :
defaultCompilerMode);
      this.compilerClassLoader = compilerClassLoader;
      this.autoGrowNullReferences = autoGrowNullReferences;
      this.autoGrowCollections = autoGrowCollections;
      this.maximumAutoGrowSize = maximumAutoGrowSize;
   }
```

SpelCompilerMode是个枚举，所有的模式如下。

（1）OFF：编译器关闭，默认是关闭的。

（2）IMMEDIATE：即时生效模式，表达式会尽快的被编译。基本是在第一次求值后马上就会执行。如果编译表达式出错（往往都是因为上面提到的类型发生改变的情况），则表达式求值的调用点会抛出异常。

（3）MIXED：混合模式，在混合模式中表达式会自动在解释器模式和编译器模式之间切换。在发生了几次解释处理后会切换到编译模式，如果编译模式哪里出错了（像上面提到的类型发生变化），则表达式会自动切换回解释器模式。过一段时间如果运用正常又会切换回编译模式。基本上像在IMMEDIATE模式下会抛出的那些异常都会被内

部处理掉。

（4）compilerClassLoader：允许自定义classload。

（5）autoGrowNullReferences：当Spel所引用的对象数组或者集合元素查找的时候如果当前位置对应的对象是Null的时候，是否自动增加empty元素，默认false。

（6）autoGrowCollections：当Spel所引用的对象数组或者List位置越界时是否可以自动增加数组或者List长度来兼容，默认false。

### 11.3.6 表达式模板设置

表达式模板运行在一段文本中混合包含一个或多个求值表达式模块。各个求值块都通过可被自定义的前后缀字符分隔，一个通用的选择是使用#{ }作为分隔符，例如：

```
String randomPhrase = parser.parseExpression(
   "random number is #{T(java.lang.Math).random()}",
   new TemplateParserContext()).getValue(String.class);

// evaluates to "random number is 0.7038186818312008"
```

求值的字符串是通过字符文本random number is以及 #{} 分隔符中的表达式求值结果拼接起来的，在这个例子中就是调用random()的结果。方法parseExpression() 的第二个传入参数类型是ParserContext。ParserContext接口用来确定表达式该如何被解析，从而支持表达式的模板功能，其实现类TemplateParserContext的定义如下：

```
public class TemplateParserContext implements ParserContext {

  public String getExpressionPrefix() {
    return "#{";
  }
  public String getExpressionSuffix() {
    return "}";
  }

  public boolean isTemplate() {
    return true;
  }
}
```

## 11.3.7 主要类关系图

主要类关系图如图11-2所示。

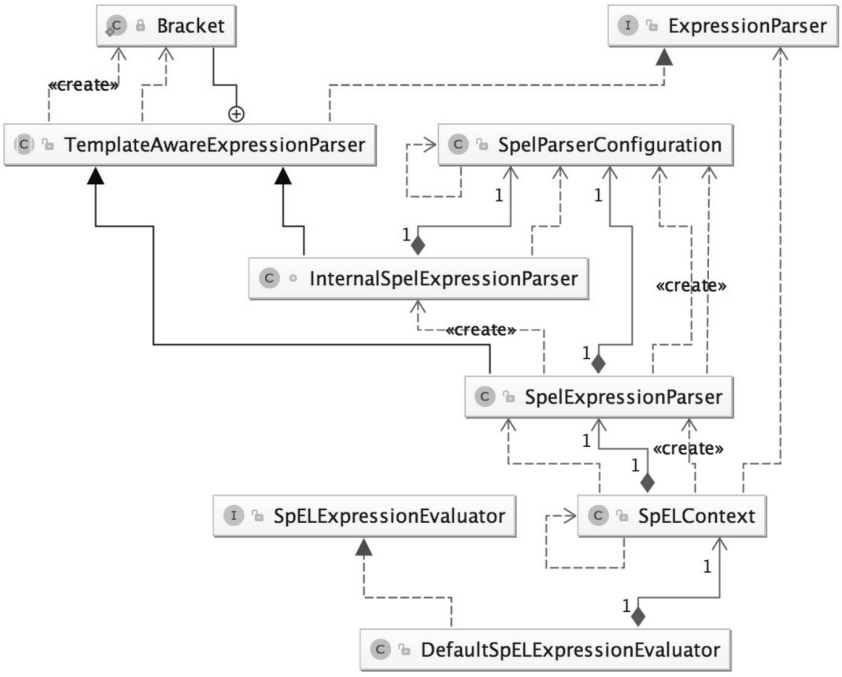



图11-2

## 11.3.8 SpEL支持的一些特性

SpEL支持以下的一些特性：

- 字符表达式
- 布尔和关系操作符
- 正则表达式
- 类表达式
- 访问properties、arrays、lists、maps等集合
- 方法调用
- 关系操作符
- 赋值
- 调用构造器
- Bean对象引用
- 创建数组
- 内联lists
- 内联maps
- 三元操作符
- 变量复值

## 11.4 Spring的主要使用场景

### 11.4.1 Spring Data JPA中SpEL支持

Spring Data JPA 1.4以后，支持在@Query中使用SpEL表达式（简介）来接收变量。

SpEL支持的变量：正如我们第4章讲的@Query查询中的#entityName保留关键字，如表11-2所示。

表11-2 entityName变量

| 变量名 | 使用方式 | 描述 |
| --- | --- | --- |
| entityName | select x from #{#entityName} x | 根据指定的 Repository 自动插入相关的 entityName。<br>有两种方式能被解析出来：<br>（1）如果定义了@Entity 注解，直接用其属性名。<br>（2）如果没定义，直接用实体的类的名称 |

```
public interface UserRepository extends JpaRepository<User,Long> {
    @Query("select u from #{#entityName} u where u.lastname = ?1")
    List<User> findByLastname(String lastname);
}
```

SpEL支持提供对查询方法参数的访问。这使你可以简单地绑定参数，或者在绑定之前执行其他操作。

```
/**
* [ 0 ]在方法中第一个声明的参数
*/
@Query("select u from User u where u.age = ?#{[0]}")
List<User> findUsersByAge(int age);
/**
*去参数中的 customer.firstname 属性
*/
@Query("select u from User u where u.firstname = :#{#customer.firstname}")
List<User> findUsersByCustomersFirstname(@Param("customer") Customer
customer);
```

## 11.4.2 Spring Cachae

（1）@Cacheable中key、condition、unless属性对SpEL的支持。

```
@Cacheable(value = "reservationsCache", key = "#restaurand.id", sync = true)
public List<Reservation> getReservationsForRestaurant( Restaurant restaurant )
{
}
@Cacheable(value = "userCache", unless = "#result != null",condition="#id>0")
public User getUserById( long id ) {
    return userRepository.getById( id );
}
```

（2）@CachePut对SpEL支持的实例：

```
   class UserRepository {
      @Caching(
         put = {
            @CachePut(value="userCache",key="'username:'+#result.username",
condition = "#result != null"),
            @CachePut(value = "userCache", key = "#result.id", condition =
"#result != null")
         }
      )
      @Transactional(readOnly = true)
      public User getById( long id ) {
         ...
      }
   }
```

而其中#result 是Spring Cache中SpEL保留关键字。

Spring Cache的详解在本书第10章也有详细介绍。

## 11.4.3 @Value

我们通过@Value取配置文件里面的key和value是最常见的场景。

```
   @Value("#{systemProperties['unknown'] ?: 'some default'}")
   private String spelSomeDefault;
```

## 11.4.4 Web验证应用场景

验证输入内容：

```
   @ValidateClassExpression(value="!(#this.login==null&&#this.email==null)",
message = "Login or email must be defined.")
   public class User {
      private String login;
      @Email
      private String email;
   }
```

## 11.4.5 总结

如果SpEL用好了，可以贯穿Spring项目的整个生命周期。当自己

写框架的时候，可以考虑支持SpEL的语法，特别是当需要自己自定义注解的时候。

# 第12章
# Spring Data REST

不是看到希望才会去坚持，而是坚持了才会看到希望。

——网络名言

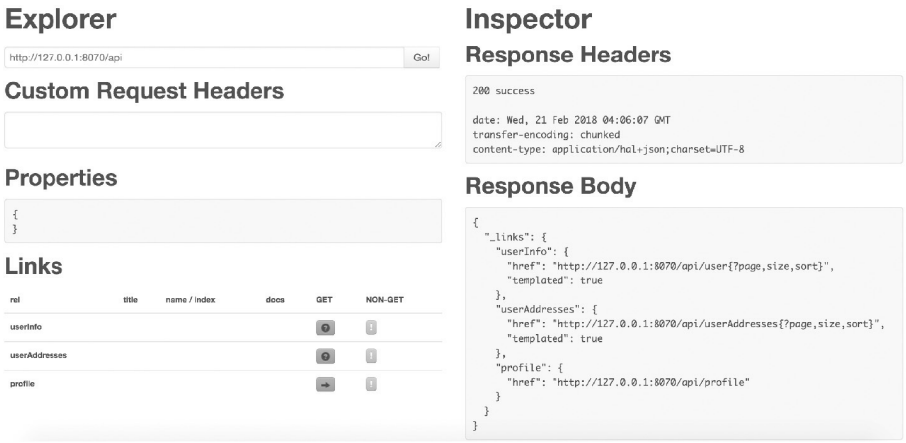



图12-1

本章主要介绍如何利用Spring Data REST快速搭建RESTFul风格的API的Server端。

## 12.1 快速入门

### 12.1.1 Spring Data REST介绍

REST风格的 Web API服务已成为在Web上应用程序集成的首选方式。市场上都在争相定义REST风格的JSON API返回格式，并且提供相应的解决方案。目前Java社区常见的对HTTP的服务接口返回的JSON解决方案有两种。

#### 1．JSON API

JSON API来自JSON的数据传输，它被隐式地定义在Ember的REST风格数据适配器中。

一般来说，Ember Data被设计用来实现这样的目的：消除哪些为不同应用程序与服务器之间通信而写的特殊代码，而是用REST风格数据适配器将它们转换成统一的方式。通过遵循共同的约定可以提高开发效率，利用更普遍的工具，可以使你更加专注于开发重点：你的程序。基于JSON API的客户端还能够充分利用缓存，以提升性能，有时甚至可以完全不需要网络请求。

下面是一个使用JSON API发送响应（response）的示例：

```
{
  "links": {
    "posts.author": {
      "href": "http://example.com/people/{posts.author}",
      "type": "people"
    },
    "posts.comments": {
      "href": "http://example.com/comments/{posts.comments}",
      "type": "comments"
    }
  },
  "posts": [{
    "id": "1",
    "title": "Rails is Omakase",
    "links": {
      "author": "9",
      "comments": [ "5", "12", "17", "20" ]
    }
  }]
}
```

JSON API严格规定了返回的JSON文档结果的格式，JSON API服务器支持通过GET方法获取资源，而且必须独立实现HTTP POST、PUT和DELETE方法的请求响应，以支持资源的创建、更新和删除。

JSON API还有N多与之协议规定相对应的客户端实现，包括Java语言的。

## 2．Spring Data REST

Spring Data REST是基于Spring Data Repositories分析实体之间的关系，为我们生成Hypermedia API（HATEOAS）风格的Http RESTFul API接口。

HATEOAS（Hypermedia as the engine of application state）是 REST架构风格中最复杂的约束，也是构建成熟REST服务的核心。它的重要性在于打破了客户端和服务器之间严格的契约，使得客户端可以更加智能和自适应，而REST服务本身的演化和更新也变得更加容易。

在介绍HATEOAS之前，先介绍一下Richardson提出的REST成熟度模型。该模型把REST服务按照成熟度划分成4个层次：

- 第一个层次（Level 0）的Web服务只是使用HTTP作为传输方式，实际上只是远程方法调用（RPC）的一种具体形式。SOAP和XML-RPC都属于此类。
- 第二个层次（Level 1）的Web服务引入了资源的概念。每个资源有对应的标识符和表达。
- 第三个层次（Level 2）的Web服务使用不同的HTTP方法来进行不同的操作，并且使用HTTP状态码来表示不同的结果。如HTTP GET方法用来获取资源，HTTP DELETE方法用来删除资源。
- 第四个层次（Level 3）的Web服务使用HATEOAS。在资源的表达中包含了链接信息。客户端可以根据链接来发现可以执行的动作。

Spring Data REST通常构建在Spring Data Repositories之上，自动将其导出为REST资源的api，减少了大量重复代码和无聊的样板代码。它利用超媒体来允许客户端查找存储库暴露的功能，并将这些资源自动集成到相关的超媒体功能中。

Spring Data REST本身就是一个Spring MVC应用程序，它的设计方式应该是尽可能少地集成到现有的Spring MVC应用程序中。现有的（或将来的）服务层可以与Spring Data REST一起运行，稍作考虑。

## 12.1.2 快速开始

我们以Gradle、Spring Boot 2.0、Spring Data JPA和Spring Data REST快速建一个REST风格的消费Server版API。

（1）为本地数据库建立两张表（user 1对多 user_address），创建脚本如下：

```
create table user (
  id int auto_increment  primary key,
  name varchar(50) null,
  email varchar(200) null
);
create table user_address(
  id int auto_increment primary key,
  user_id int null,
  city varchar(50) null,
  constraint user_address_user_id_fk foreign key (user_id) references user (id)
);
//建立外键
create index user_address_user_id_fk  on user_address (user_id);
```

（2）我们利用Gradle创建一个项目目录，如图12-2所示。

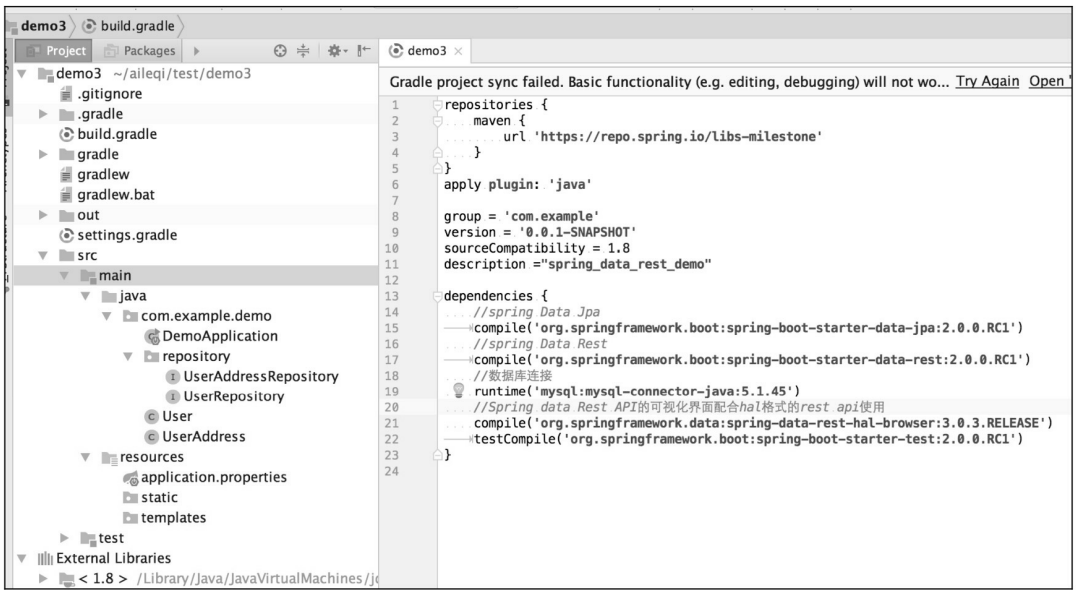



图12-2

利用gradle引入如下相关的jar包：

```
      //spring Data JPA

compile('org.springframework.boot:spring-boot-starter-data-jpa:2.0.0.RC1')
      //spring Data Rest

compile('org.springframework.boot:spring-boot-starter-data-rest:2.0.0.RC1')
      //数据库连接
      runtime('mysql:mysql-connector-java:5.1.45')
      //Spring Data REST API 的可视化界面配合 hal 格式的 rest api 使用

compile('org.springframework.data:spring-data-rest-hal-browser:3.0.3.RELEASE')
```

（3）application.properties内容如下：

```
spring.profiles.active=dev
spring.profiles=dev
server.port=8070
###Data Sources Setting
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/sys?useSSL=false
spring.datasource.username=root
spring.datasource.password=123456
spring.jpa.show-sql=true
###Spring Data Rest Setting
spring.data.rest.base-path=/api
```

（4）DemoApplication内容如下：

```
package com.example.demo;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class DemoApplication {
   public static void main(String[] args) {
      SpringApplication.run(DemoApplication.class, args);
   }
}
```

（5）两个实体的内容User：

```
package com.example.demo;
import javax.persistence.*;
import java.util.Collection;
@Entity
public class User {
   private int id;
   private String name;
   private String email;
   private Collection<UserAddress> userAddressesById;
   @Id
   @Column(name = "id", nullable = false)
   public int getId() {
      return id;
   }
   @Basic
   @Column(name = "name", nullable = true, length = 50)
   public String getName() {
      return name;
   }
   @Basic
   @Column(name = "email", nullable = true, length = 200)
   public String getEmail() {
      return email;
   }
   @OneToMany(mappedBy = "userByUserId")
   public Collection<UserAddress> getUserAddressesById() {
      return userAddressesById;
   }
}
package com.example.demo;
import javax.persistence.*;
@Entity
```

```
@Table(name = "user_address", schema = "sys", catalog = "")
public class UserAddress {
  private int id;
  private String city;
  private User userByUserId;
  @Id
  @Column(name = "id", nullable = false)
  public int getId() {
    return id;
  }
  @Basic
  @Column(name = "city", nullable = true, length = 50)
  public String getCity() {
    return city;
  }
  @ManyToOne
  @JoinColumn(name = "user_id", referencedColumnName = "id")
  public User getUserByUserId() {
    return userByUserId;
  }
}
```

（6）两个Repository内容如下：

```
package com.example.demo.repository;
import com.example.demo.UserAddress;
import org.springframework.data.jpa.repository.JpaRepository;
public interface UserAddressRepository extends
JpaRepository<UserAddress,Integer> {}

package com.example.demo.repository;
import com.example.demo.User;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.repository.query.Param;
public interface UserRepository extends JpaRepository<User,Integer> {
  Page<User> findByName(@Param("name") String name, Pageable pageable);
}
```

（7）这个时候我们不用新建Service，也不用新建Controller，我们直接启动application。控制台输出内容如图12-3所示。

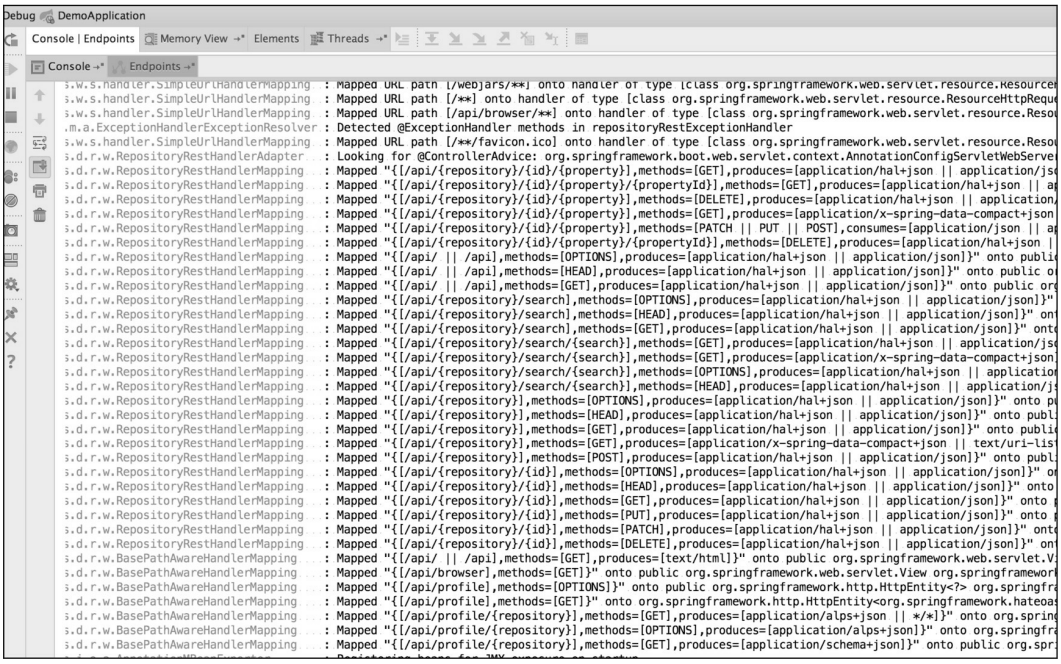



图12-3

这时候我们发现Spring Data REST通过RepositoryRestHandlerMapping帮我们自动创建了很多REST风格的API。

（8）直接调用API访问：

- {repository}默认是@Entity的name。
- {search}默认是**Repository中的自定义的方法。

```
# curl http://127.0.0.1:8070/api
Response Headers
200 success

date: Tue, 20 Feb 2018 14:34:14 GMT
transfer-encoding: chunked
content-type: application/hal+json;charset=UTF-8

Response Body
{
  "_links": {
    "userAddresses": {
      "href": "http://127.0.0.1:8070/api/userAddresses{?page,size,sort}",
      "templated": true
    },
    "users": {
      "href": "http://127.0.0.1:8070/api/users{?page,size,sort}",
      "templated": true
    },
    "profile": {
      "href": "http://127.0.0.1:8070/api/profile"
    }
  }
}
```

由于我们集成了spring-data-rest-hal-browser，所以我们可以通过控制台界面看到如图12-4所示的效果：application里面提供的api的方法及其返回结果。

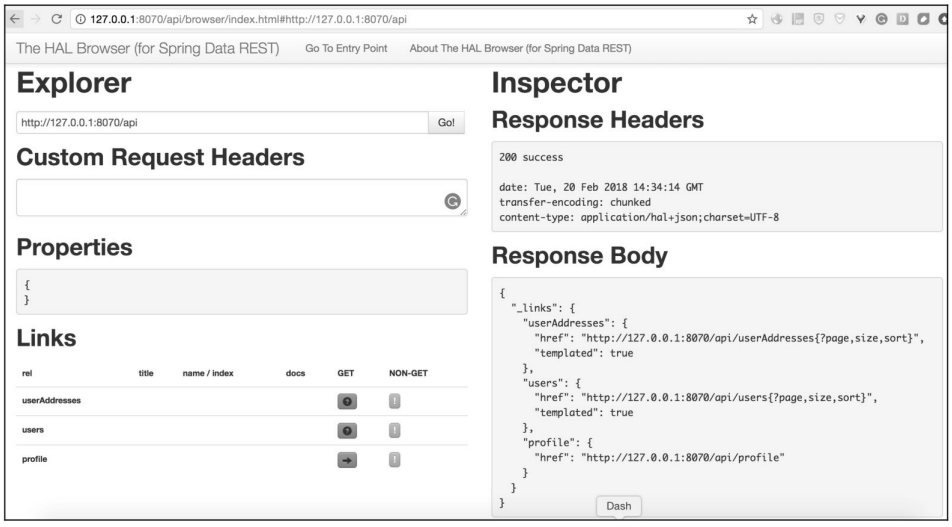



图12-4

```
#curl http://127.0.0.1:8070/api/users?page=0
Response Headers
200 success

date: Tue, 20 Feb 2018 14:38:41 GMT
transfer-encoding: chunked
content-type: application/hal+json;charset=UTF-8

Response Body
{
  "_embedded": {
    "users": [
      {
        "name": "jack",
        "email": "jack@email.com",
        "_links": {
          "self": {
            "href": "http://127.0.0.1:8070/api/users/1"
          },
          "user": {
            "href": "http://127.0.0.1:8070/api/users/1"
          },
          "userAddressesById": {
            "href": "http://127.0.0.1:8070/api/users/1/userAddressesById"
          }
        }
      }
    ]
  },
  "_links": {
    "self": {
      "href": "http://127.0.0.1:8070/api/users{?page,size,sort}",
      "templated": true
    },
    "profile": {
      "href": "http://127.0.0.1:8070/api/profile/users"
    },
    "search": {
      "href": "http://127.0.0.1:8070/api/users/search"
    }
  },
  "page": {
    "size": 20,
    "totalElements": 1,
    "totalPages": 1,
    "number": 0
  }
}
```

图例如图12-5所示。

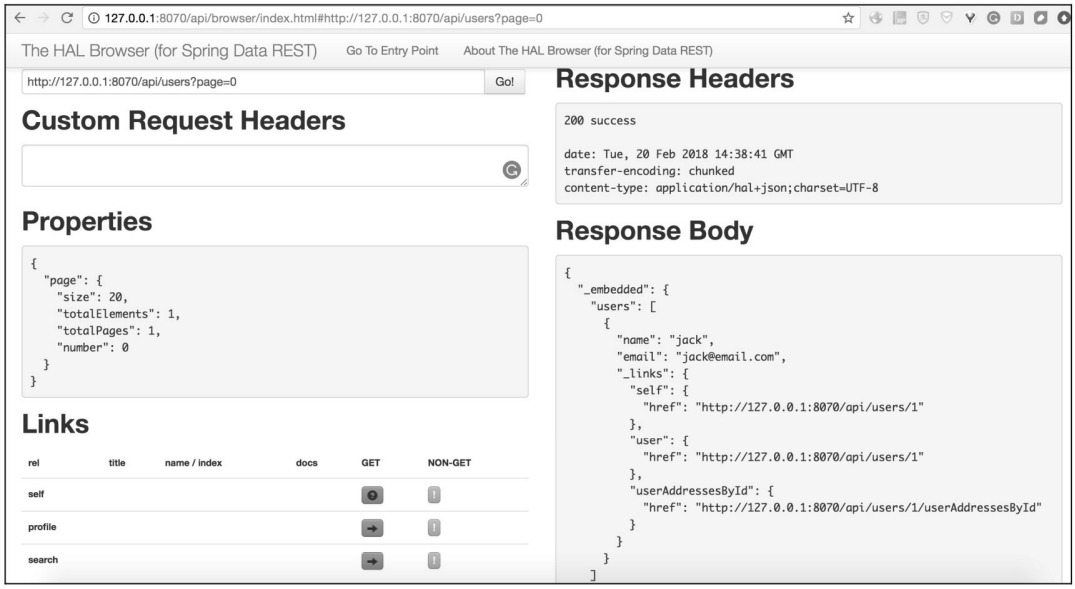



图12-5

我们会发现使用JPA和REST会如此方面和快捷，这就是约定大于配置的好处，可以使用很多开源产品。

## 12.1.3 Repository资源接口介绍

### 1．基本原理

Spring Data REST的核心功能是导出Spring Data Repositories的资源。因此，潜在调整的核心组件可以自定义导出工作的方式是存储库接口。假设以下存储库接口：

```
public interface OrderRepository extends CrudRepository<Order, Long> { }
```

对于此存储库，Spring Data REST在/orders显示集合资源。它还为URI模板/orders/{id}下的存储库管理的每个项目公开了一个项目资源。默认情况下，与这些资源交互的HTTP方法映射到CrudRepository的相应方法。

### 2．默认状态码

对于暴露的资源，我们使用一组默认状态代码：

- 200 OK：适用于纯粹的GET请求。
- 201 Created：针对创建新资源的POST和PUT请求。
- 204 No Content：对于PUT、PATCH和DELETE请求，如果配置设置为不返回资源更新的响应体（RepositoryRestConfiguration.returnBodyOnUpdate）。如果配置值设置为包含PUT的响应，则将返回200 OK进行更新，PUT将为PUT创建的资源返回201 Created。如果配置值（RepositoryRestConfiguration.returnBodyOnUpdate和Repository RestConfiguration.returnBodyCreate）显式设置为null，则会使用HTTP Accept标头的存在来确定响应代码。

## 3．支持的HTTP方法

项目资源通常支持GET、PUT、PATCH、DELETE和POST。

- GET：返回单个实体。
- PUT：更新资源。
- PATCH：与PUT类似，但部分更新资源状态。
- DELETE：删除暴露的资源。
- POST：从给定的请求正文创建一个新的实体。

## 4．分页排序

Spring Data REST会识别一些影响页面大小和起始页码的URL参数。如果你扩展PagingAndSortingRepository<T, ID>并访问所有实体的列表，你将获得前20个实体的链接。要将页面大小设置为任何其他数字，请添加size参数、Page参数，如：

```
http://127.0.0.1:8070/api/users/search/findByName{?name,page,size,sort}
```

遵循Spring Data JPA的Page参数，如：

```
curl -v"http: // localhost: 8080 / people / search / nameStartsWith? name = K
&sort = name, desc"
```

# 12.2 Spring Data REST定制化

## 12.2.1 @RepositoryRestResource改变***Repository对应的Path路径和资源名字

（1）默认情况下，导出器将使用域类的名称来显示你的CrudRepository。Spring Data REST还应用“Evo Inflector”来将资源定义成复数，所以存储库定义如下：

```
public interface UserRepository extends JpaRepository<User,Integer> {}
```

默认情况下，将会显示在URL http://localhost:8080/users/下面。

如果我们向更改users的path请添加如下注解：

```
@RepositoryRestResource(path = "user")
public interface UserRepository extends JpaRepository<User,Integer> {}
```

现在可以通过URL访问存储库：http://localhost:8080/user/。

（2）@RepositoryRestResource详解。

```
@RepositoryRestResource 使用是在***Repository 的接口上
@RepositoryRestResource(
     exported = true, //资源是否暴露，默认 true
     path = "users",//资源暴露的 path 访问路径，默认实体名字+s
     collectionResourceRel = "userInfo",//资源名字，默认实体名字
     collectionResourceDescription = @Description("用户基本信息资源"),//资源描述
     itemResourceRel = "userDetail",//取资源详情的 Item 名字
     itemResourceDescription = @Description("用户详情")
)
```

使用示例，我们改变默认的User资源的信息：

```
@RepositoryRestResource(
     exported = true,
     path = "users",
     collectionResourceRel = "userInfo",
      collectionResourceDescription = @Description("用户资源"),
     itemResourceRel = "userDetail",
      itemResourceDescription = @Description("用户详情")
)
public interface UserRepository extends JpaRepository<User,Integer> {}
```

现在可以通过URL访问存储库：http://127.0.0.1:8070/api/user，会得到如下返回结果，注意user、UserInfo、UserDetail所在的位置。

```
{
  "_embedded": {
    "userInfo": [
      {
        "email": "jack@email.com",
        "userName": "jack",
        "_links": {
          "self": {
            "href": "http://127.0.0.1:8070/api/user/1"
          },
          "userDetail": {
            "href": "http://127.0.0.1:8070/api/user/1"
          },
          "userAddress": {
            "href": "http://127.0.0.1:8070/api/user/1/userAddresses"
          }
        }
      }
    ]
  }
}
```

## 12.2.2 @RestResource 改变SearchPath

```
@RestResource(
      exported = true,//是否暴露给 Search
      path = "findName",//Search 后面的 path 路径
      rel = "names"//资源名字
)
```

其用于***Repository中的方法和@Entity的实体关系上。

【示例12.1】作用在UserRepository方法上。

```
public interface UserRepository extends JpaRepository<User,Integer> {
   @RestResource(
         exported = true,//是否暴露给 Search
         path = "findName",//Search 后面的 path 路径
         rel = "names"//资源名字
   )
   Page<User> findByName(@Param("name") String name, Pageable pageable);
}
```

访问http://127.0.0.1:8070/api/user/search会得到如下结果（注意：names和findName变化位置）：

```
{
  "_links": {
    "names": {
      "href":
"http://127.0.0.1:8070/api/user/search/findName{?name,page,size,sort}",
      "templated": true
    },
    "self": {
      "href": "http://127.0.0.1:8070/api/user/search"
    }
  }
}
```

【示例12.2】也可以填写在@Entity中的关联关系上。

```
@OneToMany(mappedBy = "userByUserId")
@RestResource(path = "userAddresses",rel = "userAddress")
public Collection<UserAddress> getUserAddressesById() {
   return userAddressesById;
}
```

可以通过http://127.0.0.1:8070/api/user/1/userAddresses访问子资源。

### 12.2.3 改变返回结果

Spring Data REST是利用Jackson来处理JSON结果的，所以Jackson的注解同样在此起作用。

Jackson的@JsonIgnore用于阻止password字段序列化为JSON。

Jackson的@JsonProperty用于改变JSON返回字段的名字。

实例如下：

```
@Entity
public class User {
   @Basic
   @Column(name = "name", nullable = true, length = 50)
   @JsonProperty("userName")
   public String getName() {
      return name;
   }
   @Basic
   @Column(name = "email", nullable = true, length = 200)
   @JsonIgnore
   public String getEmail() {
      return email;
   }
}
```

### 12.2.4 隐藏某些Repository、Repository的查询方法或@Entity关系字段

你可能不想要一个存储库、存储库上的查询方法，或者实体导出

的一个字段。这种情况，请使用@RestResource、@RepositoryRestResource注解它们，并设置exported = false。

例如，要跳过Repository：

```
@RepositoryRestResource(exported = false)
interface PersonRepository extends CrudRepository<Person, Long> {}
```

要跳过查询方法：

```
@RepositoryRestResource(path = "people", rel = "people")
interface PersonRepository extends CrudRepository<Person, Long> {
  @RestResource(exported = false)
  List<Person> findByName(String name);
}
```

跳过字段：

```
@Entity
public class Person {
  @OneToMany
  @RestResource(exported = false)
  private Map<String, Profile> profiles;
}
```

## 12.2.5 隐藏Repository的CRUD方法

如果你不想在CrudRepository上公开保存或删除方法，则可以使用@RestResource(exported = false) 设置来覆盖要关闭的方法，并将注释放在覆盖版本上。例如，为了防止HTTP用户调用CrudRepository的删除方法，请覆盖所有这些删除方法，并将注释添加到覆盖方法中。

```
@RepositoryRestResource(path = "people", rel = "people")
interface PersonRepository extends CrudRepository<Person, Long> {
  @Override
  @RestResource(exported = false)
  void delete(Long id);
  @Override
  @RestResource(exported = false)
  void delete(Person entity);
}
```

### 12.2.6 自定义JSON输出

有时在你的应用程序中，需要提供来自特定实体的其他资源的链接。例如，Customer响应可能会丰富与当前购物车的链接，或链接以管理与该实体相关的资源。Spring Data REST提供与Spring HATEOAS的集成，并为用户提供一个扩展挂钩来更改向客户端出来的资源的表示。

Spring HATEOAS定义了一个用于处理实体的ResourceProcessor<>接口。类型为ResourceProcessor<Resource<T>>的所有bean将自动由Spring Data REST导出器拾取，并在序列化类型为T的实体时触发。

例如，要为Person实体定义一个处理器，请向你的ApplicationContext添加一个@Bean，如下所示：

```
@Bean
public ResourceProcessor<Resource<Person>> personProcessor() {
   return new ResourceProcessor<Resource<Person>>() {
     @Override
     public Resource<Person> process(Resource<Person> resource) {
        //可以扩展很多
      resource.add(new Link("http://localhost:8080/people", "added-link"));
      return resource;
     }
   };
}
```

## 12.3 Spring Boot 2.0加载原理

通过前面的两节内容，我们大概知道了如何配置Spring Data REST，本节我们来解剖一下它在Spring Boot 2.0下是如何工作的。

Gradle引入compile('org.springframework.boot:spring-boot-starter-data-rest:2.0.0.RC1')，它会帮我们引入Spring boot 2.0和Spring Boot AutoConfigure 2.0。AutoConfigure所在的jar包下面，我们可以找到spring.factories文件，里面有默认加载的类，我们可以找到
org.springframework.boot.autoconfigure.data.rest.RepositoryR

打开RepositoryRestMvcAutoConfiguration有如下内容：

```
   @Configuration
   @ConditionalOnWebApplication(type = Type.SERVLET)
   @ConditionalOnMissingBean(RepositoryRestMvcConfiguration.class)
   @ConditionalOnClass(RepositoryRestMvcConfiguration.class)
   @AutoConfigureAfter({ HttpMessageConvertersAutoConfiguration.class,
        JacksonAutoConfiguration.class })
   @EnableConfigurationProperties(RepositoryRestProperties.class)
   @Import(RepositoryRestMvcConfiguration.class)
   public class RepositoryRestMvcAutoConfiguration {
      @Bean
      public SpringBootRepositoryRestConfigurer
springBootRepositoryRestConfigurer() {
         return new SpringBootRepositoryRestConfigurer();
      }
   }
```

我们会发现Spring Boot帮我们做了自动加载RepositoryRest的事情。看一下有哪些配置，打开RepositoryRestProperties。

```
   @ConfigurationProperties(prefix = "spring.data.rest")
   public class RepositoryRestProperties {
   private String basePath;
   private Integer defaultPageSize;
   private Integer maxPageSize;
   private String pageParamName;
   private String limitParamName;
   private String sortParamName;
   }
```

所以我们可以通过application.proeroties添加spring.data.rest***来配置Spring Data REST的很多默认值。

通过源码可以发现@Import(RepositoryRestMvcConfiguration.class)这个类，它是SpringRestMvc的配置类。

也就是说，如果你有一个现成的Spring MVC应用程序，希望集成Spring Data REST，其实很简单。

你的Spring MVC配置（很可能在配置MVC资源的地方）的某处会向负责配置RepositoryRestController的JavaConfig类添加一个bean引用。类名称为org.springframework.data.rest.webmvc.RepositoryRestMvcConfigu

在Java中，这样就像：

```
import org.springframework.context.annotation.Import;
import org.springframework.data.rest.webmvc.RepositoryRestMvcConfiguration;
@Configuration
@Import(RepositoryRestMvcConfiguration.class)
public class MyApplicationConfiguration {
  …
}
```

## 12.4 未来发展

由于Spring Data REST是Spring的全家桶项目之一，所以对Spring Security、Spring Boot、Spring Cloud等结合起来使用比较方便，个人感觉Spring Data REST会渐渐地进入大家的视野。

但是最近两年JSON API的协议也在逐步进入程序员的视野，JSON API也规定了最新的REST的HTTP协议的格式和交互方式。就不知道不久的将来Spring是否会考虑支持JSON API的协议规范。

# 附录1
# Repository Query Method关键字列表

世上并没有用来鼓励工作努力的赏赐，所有的赏赐都只是被用来奖励工作成果的。

——网络名言

表附-1列出了Spring Data JPA Query Method机制支持的关键字。

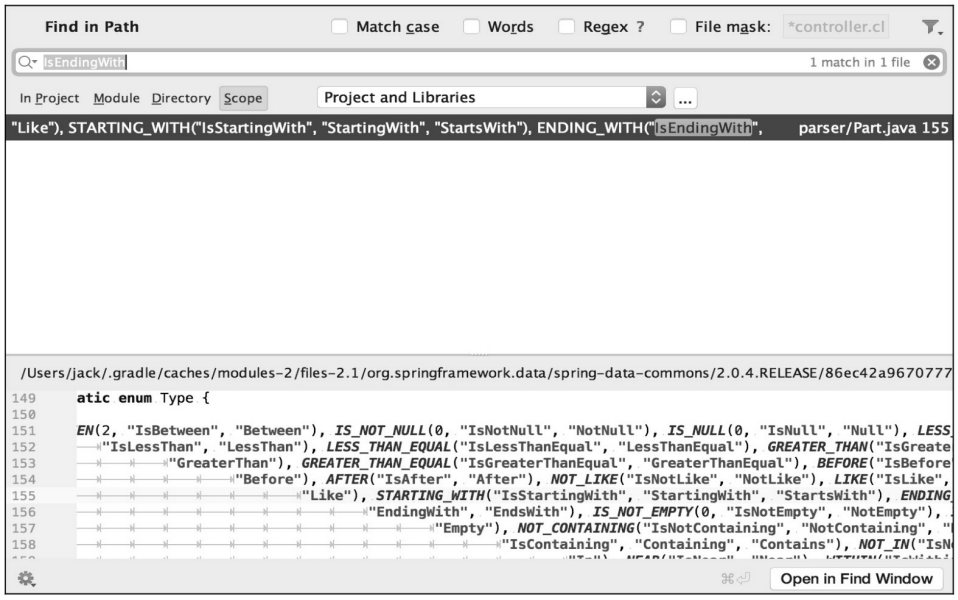
表附-1 Query Method 机制支持的关键字

| 逻辑 | Spring Data JPA 对应关键字 |
| --- | --- |
| AND | And |
| OR | Or |
| AFTER | After、IsAfter |
| BEFORE | Before、IsBefore |
| CONTAINING | Containing、IsContaining、Contains |
| BETWEEN | Between、IsBetween |
| ENDING_WITH | EndingWith、IsEndingWith、EndsWith |
| EXISTS | Exists |
| FALSE | False、IsFalse |
| GREATER_THAN | GreaterThan、IsGreaterThan |
| GREATER_THAN_EQUALS | GreaterThanEqual、IsGreaterThanEqual |
| IN | In、IsIn |
| IS | Is、Equals、(or no keyword) |
| IS_EMPTY | IsEmpty、Empty |
| IS_NOT_EMPTY | IsNotEmpty、NotEmpty |

（续表）

| 逻辑 | Spring Data JPA 对应关键字 |
| --- | --- |
| IS_NOT_NULL | NotNull、IsNotNull |
| IS_NULL | Null、IsNull |
| LESS_THAN | LessThan、IsLessThan |
| LESS_THAN_EQUAL | LessThanEqual、IsLessThanEqual |
| LIKE | Like、IsLike |
| NEAR | Near、IsNear |
| NOT | Not、IsNot |
| NOT_IN | NotIn、IsNotIn |
| NOT_LIKE | NotLike、IsNotLike |
| REGEX | Regex、MatchesRegex、Matches |
| STARTING_WITH | StartingWith、IsStartingWith、StartsWith |
| TRUE | True、IsTrue |
| WITHIN | Within、IsWithin |

大家也可以通过IntelliJ IDEA的菜单Edit→Find→Find In Path 工具查找到关键字对应的枚举在哪个类里面，如图附-1所示。




图附-1

所以在这里作者介绍一个工作中的小技巧，直接查看源码就可以知道框架支持了哪些关键字。Type枚举的关键源码如下：

```
   public static enum Type {
   BETWEEN(2, new String[]{"IsBetween", "Between"}),
   IS_NOT_NULL(0, new String[]{"IsNotNull", "NotNull"}),
   IS_NULL(0, new String[]{"IsNull", "Null"}),
   LESS_THAN(new String[]{"IsLessThan", "LessThan"}),
   LESS_THAN_EQUAL(new String[]{"IsLessThanEqual", "LessThanEqual"}),
   GREATER_THAN(new String[]{"IsGreaterThan", "GreaterThan"}),
   GREATER_THAN_EQUAL(new String[]{"IsGreaterThanEqual", "GreaterThanEqual"}),
   BEFORE(new String[]{"IsBefore", "Before"}),
   AFTER(new String[]{"IsAfter", "After"}),
   NOT_LIKE(new String[]{"IsNotLike", "NotLike"}),
   LIKE(new String[]{"IsLike", "Like"}),
   STARTING_WITH(new String[]{"IsStartingWith", "StartingWith", "StartsWith"}),
   ENDING_WITH(new String[]{"IsEndingWith", "EndingWith", "EndsWith"}),
   IS_NOT_EMPTY(0, new String[]{"IsNotEmpty", "NotEmpty"}),
   IS_EMPTY(0, new String[]{"IsEmpty", "Empty"}),
   NOT_CONTAINING(new String[]{"IsNotContaining", "NotContaining",
"NotContains"}),
   CONTAINING(new String[]{"IsContaining", "Containing", "Contains"}),
   NOT_IN(new String[]{"IsNotIn", "NotIn"}),
   IN(new String[]{"IsIn", "In"}),
   NEAR(new String[]{"IsNear", "Near"}),
   WITHIN(new String[]{"IsWithin", "Within"}),
   REGEX(new String[]{"MatchesRegex", "Matches", "Regex"}),
   EXISTS(0, new String[]{"Exists"}),
   TRUE(0, new String[]{"IsTrue", "True"}),
   FALSE(0, new String[]{"IsFalse", "False"}),
   NEGATING_SIMPLE_PROPERTY(new String[]{"IsNot", "Not"}),
   SIMPLE_PROPERTY(new String[]{"Is", "Equals"});
   ....}
```

# 附录2
# Repository Query Method返回值类型

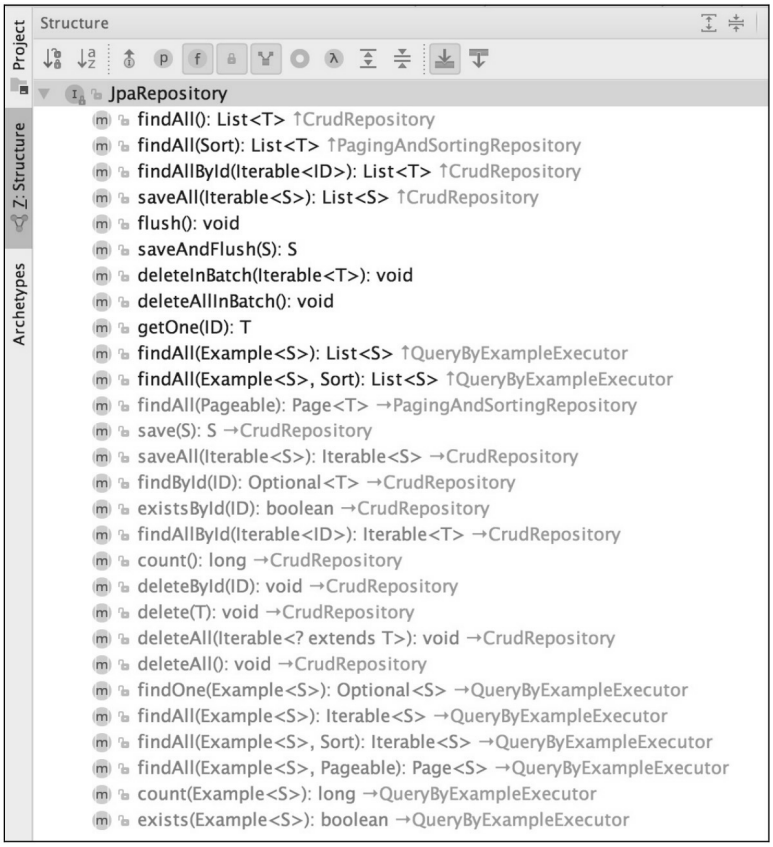
表附-2列出了Spring Data JPA Query Method机制支持的方法的返回值类型。某些特定的存储可能不支持全部的返回类型。

表附-2 Query Method机制支持的方法的返回值类型

| 返回值类型 | 描述 |
| --- | --- |
| void | 不返回结果，一般是更新操作 |
| Primitives | Java 的基本类型，一般常见的是统计操作（如：long、boolean 等） |
| Wrapper types | Java 的包装类 |
| T | 最多只返回一个实体。没有查询结果时返回 null。<br>如果超过了一个结果会抛出 IncorrectResultSizeDataAccessException 的异常 |
| Iterator<T> | 一个迭代器 |
| Collection<T> | A 集合 |
| List<T> | List 及其任何子类 |
| Optional<T> | 返回 Java 8 或 Guava 中的 Optional 类。<br>查询方法的返回结果最多只能有一个。<br>如果超过了一个结果会抛出 IncorrectResultSizeDataAccessException 的异常 |
| Option<T> | Scala 或者 javaslang 选项类型 |
| Stream<T> | Java 8 Stream |
| Future<T> | Future。<br>查询方法需要带有@Async 注解，并开启 Spring 异步执行方法的功能。<br>一般配合多线程使用。关系数据库，实际工作很少用到 |
| CompletableFuture<T> | 返回 Java8 中新引入的 CompletableFuture 类，查询方法需要带有@Async 注解，并开启 Spring 异步执行方法的功能 |
| ListenableFuture | 返回 org.springframework.util.concurrent.ListenableFuture 类，查询方法需要带有@Async 注解，并开启 Spring 异步执行方法的功能 |
| Slice | 返回指定大小的数据和是否还有可用数据的信息。需要方法带有 Pageable 类型的参数 |
| Page<T> | 在 Slice 的基础上附加返回分页总数等信息。需要方法带有 Pageable 类型的参数 |
| GeoResult<T> | 返回结果会附带诸如到相关地点距离等信息 |
| GeoResults<T> | 返回 GeoResult 的列表，并附带到相关地点平均距离等信息 |
| GeoPage<T> | 分页返回 GeoResult，并附带到相关地点平均距离等信息 |

只有支持地理空间查询的数据存储才支持GeoResult、GeoResults、GeoPage等返回类型。

而我们要看我们引用的那个Spring Data的实现子模块，我们以Spring Data JPA为例，看看JPA默认帮我们实现了哪些返回值类型，如图附-2所示。




图附-2

我们还是通过工具分析JpaRepository帮我们实现了哪些返回类型。这样不至于我们直接看官方文档的时候一头雾水。

# 附录3
# JPA注解大全

JPA实体里面的注解大全如表附-3所示，详细用法见第5章。

表附-3　JPA实体里面的注解大全

| 注解 | 描述 |
| --- | --- |
| @Entity | 声明类为实体或表 |
| @Table | 声明表名 |
| @Basic | 指定非约束明确的各个字段 |
| @Embedded | 指定类或它的值是一个可嵌入的类的实例的实体的属性 |
| @EmbeddedId | 可嵌入式联合主键 |
| @Enumerated | 这个注解很好用，直接映射 enum 枚举类型的字段 |
| @Id | 指定的类的属性，用于识别（一个表中的主键） |
| @IdClass | 联合主键 |
| @GeneratedValue | 指定如何标识属性可以被初始化，例如自动、手动，或从序列表中获得的值 |
| @Transient | 表示该属性并非一个到数据库表的字段的映射，JPA 将忽略该属性。如果一个属性并非数据库表的字段映射，就务必将其标注为@Transient，否则 JPA 默认其注解为@Basic |
| @Temporal | 在数据库中，表示时间类型的数据有 DATE、TIME 和 TIMESTAMP 三种精度（即单纯的日期、时间，或者两者兼备）。可使用@Temporal 来设置 Date 类型的属性映射到对应精度的字段。<br>@Temporal(TemporalType.DATE)映射为日期 // date （只有日期）<br>@Temporal(TemporalType.TIME)映射为日期 // time （是有时间）<br>@Temporal(TemporalType.TIMESTAMP)映射为日期 // date time（日期+时间） |
| @Column | 指定持久属性栏属性 |
| @Lob | 将属性映射成数据库支持的大对象类型 |
| @SequenceGenerator | 指定在@GeneratedValue 注解中指定的属性的值。它创建了一个序列 |

（续表）

| 注解 | 描述 |
| --- | --- |
| @TableGenerator | 指定在@GeneratedValue 批注指定属性的值发生器。它创造了的值生成的表 |
| @AccessType | 这种类型的注释用于设置访问类上。而其有两个值：<br>① @AccessType（FIELD），直接访问 Entity 的变量，可以不定义 getter 和 setter 方法，但是需要将变量定义为 public。需要在变量上定义字段的属性；<br>② @AccessType（PROPERTY），通过 getter 和 setter 方法访问 Entity 的变量，可以把变量定义为 private。需要在 getter 方法上定义字段的属性 |
| @JoinColumn | 指定一个实体组织或实体的集合。这是用在多对一和一对多关联 |
| @JoinColumns | 里面有多个@JoinColumn。一般用于多个字段关联关系 |
| @UniqueConstraint | 指定的字段和用于主要或辅助表的唯一约束 |
| @SqlResultSetMappings<br>@ColumnResult<br>@SqlResultSetMapping<br>@EntityResult | 配合@NamedNativeQuery 一起使用，映射 SQL 的查询结果字段 |
| @OrderBy | 排序 |
| @ManyToMany | 定义了连接表之间的多对多的关系 |
| @ManyToOne | 定义了连接表之间的多对一的关系 |
| @OneToMany | 定义了连接表之间存在一个一对多的关系 |
| @OneToOne | 定义了连接表之间有一个一对一的关系 |
| @NamedQueries<br>@NamedQuery | 在实体类上，声明式 JPQL 配置方法 |
| @NamedNativeQuery | 在实体类上，声明式，原始 SQL 配置 |
| @NamedEntityGraph<br>@EntityGraph | 为了提高查询效率。解决 N+1 条 SQL 的问题 |

Auth相关注解如表附-4所示，需要注意的是，需要配合@EntityListeners(AuditingEntityListener. class)一起使用。

表附-4　Auth相关注解

| 注解 | 描述 |
| --- | --- |
| @CreatedDate | 当 Insert 的时候默认添加时间进去 |
| @CreatedBy | 当 Insert 的时候默认添加当前 user 进去 |
| @LastModifiedDate | 每次 update 的时候就会添加时间进去 |
| @LastModifiedBy | 每次 Update 的时候，都会添加 user 进去 |
| @Version | |

其中Jackson相关的注解如表附-5所示，我们有时会在实体上会用到。

表附-5　Jackson相关注解

| Jackson 常用注解 | 描述 |
| --- | --- |
| @JsonProperty | 此注解是属性或方法注解。修改 Jackson 返回的 property 名字 |
| @JsonIgnoreProperties | 此注解是类注解，作用是 JSON 序列化时将 Java Bean 中的一些属性忽略掉，序列化和反序列化都受影响 |
| @JsonIgnore | 此注解用于属性或者方法上（最好是属性上），作用和上面的@JsonIgnoreProperties 一样 |
| @JsonFormat | 此注解用于属性或者方法上（最好是属性上），可以方便地把 Date 类型直接转化为我们想要的模式，比如@JsonFormat(pattern = "yyyy-MM-dd HH-mm-ss") |
| @JsonSerialize | 此注解用于属性或者 getter 方法上，用于在序列化时嵌入我们自定义的代码，比如序列化一个 double 时在其后面限制两位小数点 |
| @JsonDeserialize | 此注解用于属性或者 setter 方法上，用于在反序列化时可以嵌入我们自定义的代码，类似于上面的@JsonSerialize |
| @JsonEnumDefaultValue | 此注解用在枚举类型的属性或者枚举类型的 setter 方法上，去定义枚举类型的属性的默认值，当解码序列化的时候发现未知枚举值异常的时候 |
| @JsonSetter | 告诉 jackson 哪个方法是 setter 方法 |

Hibernate　Validator注解，内置的验证约束注解如表附-6所示。

表附-6　内置的验证约束注解

| 验证注解 | 验证的数据类型 | 说明 |
| --- | --- | --- |
| @AssertFalse | Boolean、boolean | 验证注解的元素值是 false |
| @AssertTrue | Boolean、boolean | 验证注解的元素值是 true |
| @NotNull | 任意类型 | 验证注解的元素值不是 null |
| @Null | 任意类型 | 验证注解的元素值是 null |
| @Min(value=值) | BigDecimal、BigInteger、byte、short、int、long 等任何 Number 或 CharSequence（存储的是数字）子类型 | 验证注解的元素值大于等于@Min 指定的 value 值 |
| @Max（value=值） | 和@Min 要求一样 | 验证注解的元素值小于等于@Max 指定的 value 值 |
| @DecimalMin(value=值) | 和@Min 要求一样 | 验证注解的元素值大于等于@ DecimalMin 指定的 value 值 |
| @DecimalMax(value=值) | 和@Min 要求一样 | 验证注解的元素值小于等于@ DecimalMax 指定的 value 值 |
| @Digits(integer=整数位数, fraction=小数位数) | 和@Min 要求一样 | 验证注解的元素值的整数位数和小数位数上限 |

（续表）

| 验证注解 | 验证的数据类型 | 说明 |
| --- | --- | --- |
| @Size(min=下限，max=上限) | 字符串、Collection、Map、数组等 | 验证注解的元素值在 min 和 max（包含）指定区间之内，如字符长度、集合大小 |
| @Past | java.util.Date、<br>java.util.Calendar、<br>Joda Time 类库的日期类型 | 验证注解的元素值（日期类型）比当前时间早 |
| @Future | 与@Past 要求一样 | 验证注解的元素值（日期类型）比当前时间晚 |
| @NotBlank | CharSequence 子类型 | 验证注解的元素值不为空（不为 null、去除首位空格后长度为 0），不同于@NotEmpty，@NotBlank 只应用于字符串且在比较时会去除字符串的首位空格 |
| @Length(min=下限，max=上限) | CharSequence 子类型 | 验证注解的元素值长度在 min 和 max 区间内 |
| @NotEmpty | CharSequence 子类型、Collection、Map、数组 | 验证注解的元素值不为 null 且不为空（字符串长度不为 0、集合大小不为 0） |
| @Range(min=最小值，max=最大值) | BigDecimal、BigInteger、CharSequence、byte、short、int、long 等原子类型和包装类型 | 验证注解的元素值在最小值和最大值之间 |
| @Email(regexp=正则表达式，flag=标志的模式) | CharSequence 子类型（如 String） | 验证注解的元素值是 Email，也可以通过 regexp 和 flag 指定自定义的 email 格式 |
| @Pattern(regexp=正则表达式,flag=标志的模式) | String、任何 CharSequence 的子类型 | 验证注解的元素值与指定的正则表达式匹配 |
| @Valid | 任何非原子类型 | 指定递归验证关联的对象。如用户对象中有个地址对象属性，如果想在验证用户对象时一起验证地址对象的话，在地址对象上加@Valid 注解即可级联验证 |

# 附录4
# Spring中涉及的注解

Spring MVN常用注解如表附-7所示。

表附-7 Spring MVN常用注解

| 注解 | 描述 |
| --- | --- |
| @CrossOrigin | 解决跨域问题 |
| @Controller | 用于定义控制器类，在 spring 项目中由控制器负责将用户发来的 URL 请求转发到对应的服务接口（service 层），一般这个注解在类中，通常方法需要配合注解@RequestMapping |
| @RestController | 用于标注控制层组件（如 struts 中的 action）、@ResponseBody 和@Controller 的合集 |
| @RequestMapping | 提供路由信息，负责 URL 到 Controller 中的具体函数的映射 |
| @RequestBody | 该注解用于读取 Request 请求的 body 部分数据，使用系统默认配置的 HttpMessageConverter 进行解析，然后把相应的数据绑定到要返回的对象上，再把 HttpMessageConverter 返回的对象数据绑定到 controller 中方法的参数上 |
| @ResponseBody | 该注解用于将 Controller 的方法返回的对象，通过适当的 HttpMessageConverter 转换为指定格式后，写入到 Response 对象的 body 数据区 |
| @ModelAttribute | 在方法定义上使用 @ModelAttribute 注解：Spring MVC 在调用目标处理方法前，会先逐个调用在方法级上标注了@ModelAttribute 的方法。在方法的入参前使用 @ModelAttribute 注解：可以从隐含对象中获取隐含的模型数据中获取对象，再将请求参数绑定到对象中，再传入入参将方法入参对象添加到模型中 |
| @RequestParam | 在处理方法入参处使用 @RequestParam 可以把请求参 数传递给请求方法 |
| @PathVariable | 绑定 URL 占位符到入参 |
| @ExceptionHandler | 注解到方法上，出现异常时会执行该方法 |
| @ControllerAdvice | 使一个 Contoller 成为全局的异常处理类，类中用@ExceptionHandler 注解的方法可以处理所有 Controller 发生的异常 |

Spring boot中常用注解如表附-8所示。

表附-8 Spring boot中常用注解

| 注解 | 描述 |
| --- | --- |
| @EnableAutoConfiguration | Spring Boot 自动配置（auto-configuration）：尝试根据你添加的 jar 依赖自动配置你的 Spring 应用。例如，如果你的 classpath 下存在 HSQLDB，并且你没有手动配置任何数据库连接 beans，那么我们将自动配置一个内存型（in-memory）数据库。你可以将 @EnableAutoConfiguration 或者 @SpringBootApplication 注解添加到一个 @Configuration 类上来选择自动配置。如果发现应用了你不想要的特定自动配置类，你可以使用@EnableAutoConfiguration 注解的排除属性来禁用它们 |
| @ComponentScan | 表示将该类自动发现扫描组件。个人理解相当于如果扫描到有@Component、@Controller、@Service 等这些注解的类，并注册为 Bean，可以自动收集所有的 Spring 组件，包括@Configuration 类。我们经常使用@ComponentScan 注解搜索 beans，并结合@Autowired 注解导入。可以自动收集所有的 Spring 组件，包括@Configuration 类。我们经常使用@ComponentScan 注解搜索 beans，并结合@Autowired 注解导入。如果没有配置的话，Spring Boot 会扫描启动类所在包下以及子包下的使用了@Service、@Repository 等注解的类 |
| @Configuration | 相当于传统的 XML 配置文件，如果有些第三方库需要用到 XML 文件，建议仍然通过@Configuration 类作为项目的配置主类——可以使用@ImportResource 注解加载 XML 配置文件 |
| @Import | 用来导入其他配置类 |
| @ImportResource | 用来加载 XML 配置文件 |
| @Resource(name=”name”,type=”type”) | 没有括号内内容的话，默认 byName。与@Autowired 干类似的事 |
| @Value | 注入 Spring boot application.properties 配置的属性的值。示例代码：<br>@Value(value = “#{message}”) private String message; |
| @Bean | 相当于 XML 中的<bean></bean>。放在方法的上面，而不是类，意思是产生一个 bean，并交给 spring 管理 |
| @Component | 泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注 |
| @Service | 一般用于修饰 service 层的组件 |
| @Repository | 使用@Repository 注解可以确保 DAO 或者 repositories 提供异常转译，这个注解修饰的 DAO 或者 repositories 类会被 ComponetScan 发现并配置，同时也不需要为它们提供 XML 配置项 |
| @AutoWired | 自动导入依赖的 bean。byType 方式。把配置好的 Bean 拿来用，完成属性、方法的组装，它可以对类成员变量、方法及构造函数进行标注，完成自动装配的工作。当加上（required=false）时，就算找不到 bean 也不报错 |
| @Inject | 等价于默认的@Autowired，只是没有 required 属性 |
| @Qualifier | 当有多个同一类型的 Bean 时，可以用@Qualifier(“name”)来指定，与@Autowired 配合使用。@Qualifier 限定描述符除了能根据名字进行注入，能进行更细粒度的控制如何选择候选者，具体使用方式如下：<br>@Autowired @Qualifier(value = “demoInfoService”) private DemoInfoService demoInfoService; |

加载条件注解如表附-9所示。

表附-9　加载条件注解

| 注解 | 描述 |
| --- | --- |
| @ConditionalOnBean | 当且仅当指定的 bean classes and/or bean names 不存在当前容器中，才创建标记上该注解的类的实例，有指定忽略 ignored 的参数存在，可以忽略 Class、Type 等 |
| @ConditionalOnClass | 当且仅当 ClassPath 存在指定的 Class 时，才创建标记上该注解的类的实例 |
| @ConditionalOnMissingClass | 当且仅当 ClassPath 不存在指定的 Class 时，创建标记上该注解的类的实例 |
| @ConditionalOnProperty | 当且仅当 Application.properties 存在指定的配置项时，创建标记上了该注解的类的实例 |
| @ConditionalOnJava | 指定 JDK 的版本 |
| @ConditionalOnExpression | 表达式用${..}=false 等来表示 |
| @ConditionalOnJndi | JNDI 存在该项时创建 |
| @ConditionalOnResource | 在 classpath 下存在指定的 resource 时创建 |

# 附录5 application.properties里面关于JPA的配置大全

```
    # LOGGING 日志配置 key
     logging.config=
    # Location of the logging configuration file. For instance `classpath:logback.xml`
for Logback logging.exception-conversion-word=%wEx
    # Conversion word used when logging exceptions.
    logging.file=
    # Log file name. For instance `myapp.log`
    logging.level.*=
     # Log levels severity mapping. For instance
`logging.level.org.springframework=DEBUG`
     logging.path=
    # Location of the log file. For instance `/var/log`
    logging.pattern.console=
    # Appender pattern for output to the console. Only supported with the default
logback setup.
     logging.pattern.file=
     # Appender pattern for output to the file. Only supported with the default logback
setup.
    logging.pattern.level=
    # Appender pattern for log level (default %5p). Only supported with the default
logback setup. logging.register-shutdown-hook=false
    # Register a shutdown hook for the logging system when it is initialized.
    # DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties.java) 数据源
配置项
    spring.datasource.continue-on-error=false
     # Do not stop if an error occurs while initializing the database.
    spring.datasource.data=
    # Data (DML) script resource references.
    spring.datasource.data-username=
     # User of the database to execute DML scripts (if different).
    spring.datasource.data-password=
```

```
    # Password of the database to execute DML scripts (if different).
    spring.datasource.dbcp2.*=
    # Commons DBCP2 specific settings
    spring.datasource.driver-class-name=
    # Fully qualified name of the JDBC driver. Auto-detected based on the URL by
default. spring.datasource.generate-unique-name=false
    # Generate a random datasource name.
    spring.datasource.hikari.*=
    # Hikari specific settings
    spring.datasource.initialize=true
    # Populate the database using 'data.sql'.
     spring.datasource.jmx-enabled=false
    # Enable JMX support (if provided by the underlying pool).
    spring.datasource.jndi-name=
    # JNDI location of the datasource. Class, url, username & password are ignored
when set.
    spring.datasource.name=testdb  # Name of the datasource.
    spring.datasource.password= # Login password of the database.
     spring.datasource.platform=all
    # Platform to use in the DDL or DML scripts (e.g. schema-${platform}.sql or
data-${platform}.sql).
    spring.datasource.schema=
    # Schema (DDL) script resource references.
    spring.datasource.schema-username=
    # User of the database to execute DDL scripts (if different).
    spring.datasource.schema-password=
    # Password of the database to execute DDL scripts (if different).
     spring.datasource.separator=;
     # Statement separator in SQL initialization scripts.
     spring.datasource.sql-script-encoding= # SQL scripts encoding.
     spring.datasource.tomcat.*=  # Tomcat datasource specific settings
     spring.datasource.type=
     # Fully qualified name of the connection pool implementation to use. By default,
it is auto-detected from the classpath.
     spring.datasource.url= # JDBC url of the database.
    spring.datasource.username= # Login user of the database.
    spring.datasource.xa.data-source-class-name= # XA datasource fully qualified
name.
    spring.datasource.xa.properties= # Properties to pass to the XA data source.
    # JPA (JpaBaseConfiguration, HibernateJpaAutoConfiguration)
spring.data.jpa.repositories.enabled=true # Enable JPA repositories.
    spring.jpa.database=
     # Target database to operate on, auto-detected by default. Can be alternatively
set using the "databasePlatform" property.
    spring.jpa.database-platform=
```

```
    # Name of the target database to operate on, auto-detected by default. Can be
alternatively set using the "Database" enum.
   spring.jpa.generate-ddl=false
    # Initialize the schema on startup.
    spring.jpa.hibernate.ddl-auto=
   # DDL mode. This is actually a shortcut for the "hibernate.hbm2ddl.auto" property.
Default to "create-drop" when using an embedded database, "none" otherwise.
    spring.jpa.hibernate.naming.implicit-strategy=
   # Hibernate 5 implicit naming strategy fully qualified name.
   spring.jpa.hibernate.naming.physical-strategy=
    # Hibernate 5 physical naming strategy fully qualified name.
   spring.jpa.hibernate.naming.strategy=
   # Hibernate 4 naming strategy fully qualified name. Not supported with Hibernate
5.
   spring.jpa.hibernate.use-new-id-generator-mappings=
   # Use Hibernate's newer IdentifierGenerator for AUTO, TABLE and SEQUENCE.
    spring.jpa.open-in-view=true
   # Register OpenEntityManagerInViewInterceptor. Binds a JPA EntityManager to the
thread for the entire processing of the request.
   spring.jpa.properties.*=
   # Additional native properties to set on the JPA provider.
   spring.jpa.show-sql=false
   # Enable logging of SQL statements.
   # REDIS (RedisProperties.java)配置项
   spring.redis.cluster.max-redirects=
   # Maximum number of redirects to follow when executing commands across the cluster.
   spring.redis.cluster.nodes=
   # Comma-separated list of "host:port" pairs to bootstrap from.
   spring.redis.database=0
   # Database index used by the connection factory.
   spring.redis.url=
   # Connection URL, will override host, port and password (user will be ignored),
e.g. redis://user:password@example.com:6379
   spring.redis.host=localhost
   # Redis server host.
   spring.redis.password=
   # Login password of the redis server.
   spring.redis.ssl=false
   # Enable SSL support.
    spring.redis.pool.max-active=8
   # Max number of connections that can be allocated by the pool at a given time.
Use a negative value for no limit.
   spring.redis.pool.max-idle=8
   # Max number of "idle" connections in the pool. Use a negative value to indicate
an unlimited number of idle connections.
```

```
    spring.redis.pool.max-wait=-1
   # Maximum amount of time (in milliseconds) a connection allocation should block
before throwing an exception when the pool is exhausted. Use a negative value to
block indefinitely.
   spring.redis.pool.min-idle=0
   # Target for the minimum number of idle connections to maintain in the pool.
This setting only has an effect if it is positive.
   spring.redis.port=6379 # Redis server port.
   spring.redis.sentinel.master= # Name of Redis server.
   spring.redis.sentinel.nodes= # Comma-separated list of host:port pairs.
   spring.redis.timeout=0 # Connection timeout in milliseconds.
```

以上配置属性是摘自官方的关键内容，只是让大家做到心中有数。特别是在Spring Boot的环境下，如果掌握application-properties文件和Spring全家桶里面的注解，基本上就能掌握spring的精髓。其实这些东西也不用死记硬背，这里给大家一个找这个配置文件属性key的一个方法，通过IDEA找到 ***Properties.java 就可以找到相关的property的配置key有哪些了。

官方参考地址：

```
   https://docs.spring.io/spring-boot/docs/current/reference/html/common-appli
cation-properties.html
```