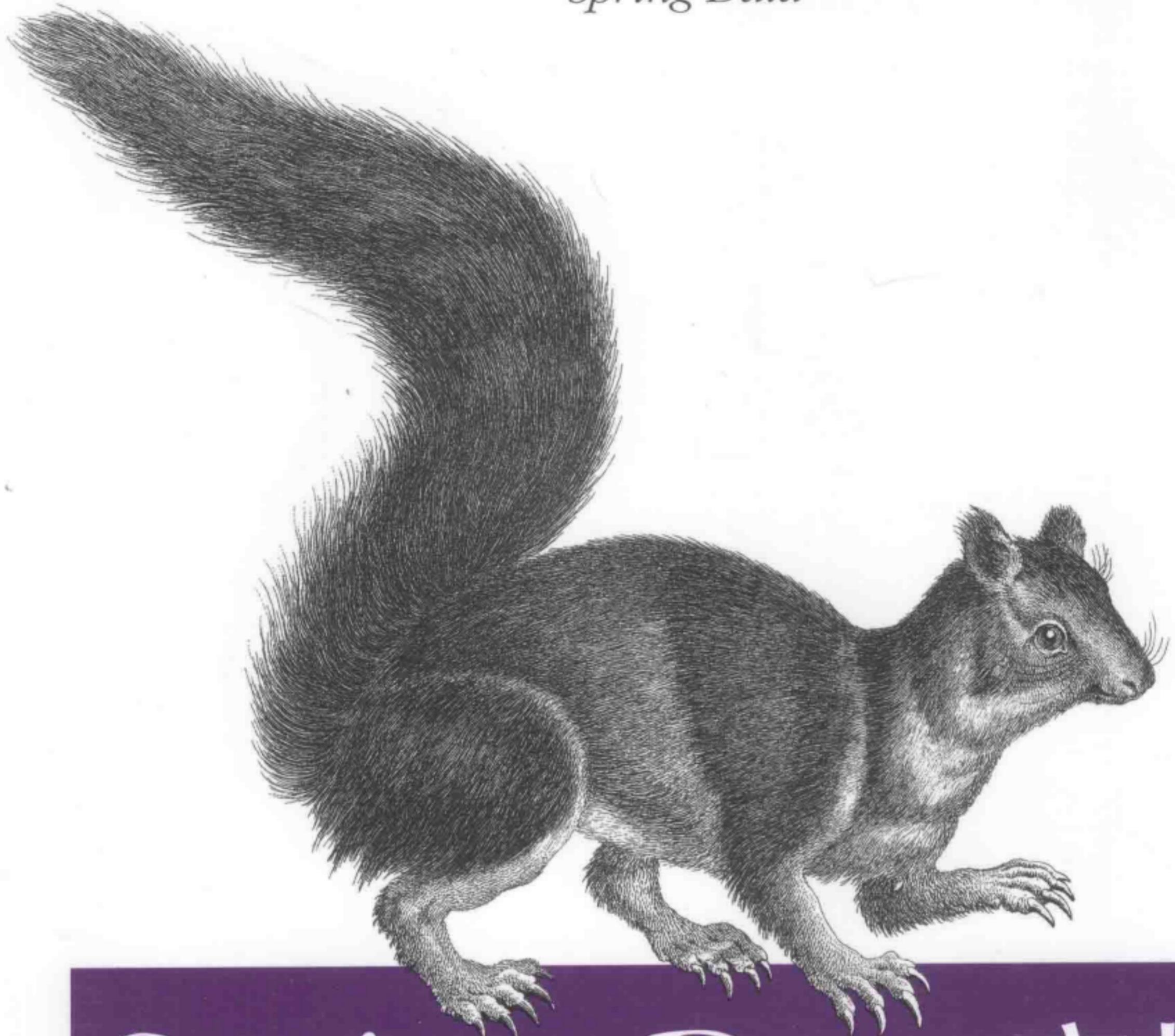


*Spring Data*



# Spring Data 实战

[美] *Mark Pollack Oliver Gierke  
Thomas Risberg Jon Brisbin  
Michael Hunger* 著  
张卫滨 文建国 译

O'REILLY®

人民邮电出版社  
POSTS & TELECOM PRESS

# Spring Data实战

基于关系型数据库构建企业级Java应用时，会有多个数据访问框架供你选择。但是该如何应对大数据呢？本书以实际经验介绍了借助Spring Data如何便利地构建应用，这些应用会用到多种新的数据访问技术，如NoSQL和Hadoop。

通过多个样例项目，你将学到Spring Data所提供的一致编程模型，这种模型保留了每种NoSQL数据库专有的特性以及功能，并且能够帮助你开发Hadoop应用来应对广泛的用例场景，如数据分析、事件流处理以及工作流。你还能学到，为了编写基于RDBMS的数据访问层，Spring Data为Spring已有的JPA和JDBC添加了哪些功能。

- 学习使用Spring的模板帮助类来简化数据库的特定功能；
- 探索Spring Data的Repository抽象以及高级查询功能；
- 借助Spring Data来使用Redis（键/值存储）、HBase（列族）、MongoDB（文档数据库）以及Neo4j（图数据库）；
- 探索GemFire分布式数据网格解决方案；
- 将Spring Data JPA管理的实体以RESTful Web服务的形式导出到Web中；
- 使用轻量级的对象映射框架，简化HBase应用的开发；
- 使用Spring Batch和Spring Integration构建大数据管道。

“你将会深刻理解现代的数据访问为何更加专门化和碎片化，NoSQL数据存储的主要分类有哪些，Spring Data如何帮助Java开发人员在新的环境下高效工作。”

——Rod Johnson  
Spring框架创始人

“通过切换到Spring Batch和Spring Data，我们彻底简化并降低了与Hadoop交互所带来的复杂性，同时增加了可靠性。”

——David Gevorkyan  
软件工程师，eHarmony

封面设计：Karen Montgomery，张健

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行

This Authorized Edition for sale only in the territory of People's Republic of China  
(excluding Hong Kong, Macao and Taiwan)

· 分类建议：计算机/程序设计/Java

人民邮电出版社网址：[www.ptpress.com.cn](http://www.ptpress.com.cn)



O'REILLY®  
[oreilly.com.cn](http://oreilly.com.cn)

ISBN 978-7-115-34370-3



ISBN 978-7-115-34370-3

定价：59.00 元

# Spring Data 实战

【美】Mark Pollack, Oliver Gierke, Thomas Risberg,  
Jon Brisbin, Michael Hunger 著

张卫滨 文建国 译

人民邮电出版社

## 图书在版编目 (C I P) 数据

Spring Data实战 / (美) 波拉克 (Pollack, M.) 等著 ; 张卫滨, 文建国译. -- 北京 : 人民邮电出版社, 2014. 4

ISBN 978-7-115-34370-3

I. ①S… II. ①波… ②张… ③文… III. ①JAVA语言—程序设计 IV. ①TP312

中国版本图书馆CIP数据核字(2014)第000746号

## 版权声明

Copyright © 2013 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2014. Authorized translation of the English edition, 2013 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

本书中文简体字版由 **O'Reilly Media, Inc.** 授权人民邮电出版社出版。未经出版者书面许可，对本书的任何部分不得以任何方式复制或抄袭。

版权所有，侵权必究。

---

◆ 著 [美] Mark Pollack Oliver Gierke Thomas Risberg

Jon Brisbin Michael Hunger

译 张卫滨 文建国

责任编辑 杜 洁

责任印制 程彦红 焦志炜

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号

邮编 100164 电子邮件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

三河市海波印务有限公司印刷

◆ 开本：787×1000 1/16

印张：17

字数：338 千字

2014 年 4 月第 1 版

印数：1—3 000 册

2014 年 4 月河北第 1 次印刷

著作权合同登记号 图字：01-2013-3679 号

---

定价：59.00 元

读者服务热线：(010) 81055410 印装质量热线：(010) 81055316

反盗版热线：(010) 81055315

广告经营许可证：京崇工商广字第 0021 号

---

# 内容提要

数据访问领域正在发生一场变革，关系型数据库无法解决的问题需要新的数据访问技术来解决。Spring Data 项目就是一种简化 Java 应用构建的数据访问技术，它可以帮助开发人员高效地使用最新的数据处理和管理工具，同时还能够以最新的方式使用传统的数据库。

本书从 Spring Data 背景知识、关系型数据库、NoSQL、快速应用开发、大数据、数据网格 6 个方面深度解析了数据访问技术，介绍的内容都是最流行和前沿的，其中文档数据库、图数据库、键 / 值存储、Hadoop 以及 GemFire 数据结构等是最重要的内容。本书介绍了 Spring Data、Repository、Querydsl 的基础理念，然后阐述了借助 Spring Data 如何简化 NoSQL 和大数据的访问，并且涵盖了使用 Spring Roo 和 Spring Data REST 导出功能如何实现应用的快速开发，除此之外，书中还涉及与其他 Spring 子项目的协同工作，如 Spring Integration 和 Spring Batch。

本书面向实战、结构清晰，示例丰富，适用于各类 Java 开发人员和数据库开发人员，也可以作为各大、中专院校相关专业师生的参考用书和相关培训机构的培训教材。

感谢我的妻子 Daniela、我的儿子 Gabriel 和 Alexandre，因为他们的宽容才使得我能够挤出时间来撰写这本书。

—Mark Pollack

我要感谢我的家人、朋友、音乐家以及迄今为止与我共同奋斗的每位同事；还有带来这段精彩旅程的整个 Spring Data 和 SpringSource 团队；最后，事实上首先要感谢的是 Sabine，她给了我无尽的关爱和支持。

—Oliver Gierke

致我的妻子 Carol 和我的儿子 Alex，感谢你们让我的人生多姿多彩，并给予我无限的支持和鼓励。

—Thomas Risberg

致我的妻子 Tisha，我的儿子 Jack、Ben 和 Daniel，我的女儿 Morgan 和 Hannah，感谢你们对我的关爱、支持和容忍。如果没有你们，所有的这一切都没有意义。

—Jon Brisbin

特别感谢 Rod 和 Emil 启动了 Spring Data 项目，特别感谢 Oliver，他使得这个项目变得意义非凡。感谢我的家人，你们总是非常支持我疯狂工作；有如此善解人意的女人在我身边，我心怀感激。

—Michael Hunger

我要感谢我的妻子 Nanette 和我的孩子们，他们给予我无限的支持、宽容和理解。同样要感谢 Rod 和 Spring Data 团队的同事们，他们使得这一切成为可能。

—David Turanski

---

# 序

我们生活在一个很有意思的时代，新的业务流程驱动着新的需求。我们以前认为理所应当的事情正面临着被颠覆的威胁——其中有一条就是关系型数据库是实现持久化的默认可选方案。尽管这个理念还被广泛认可，但是对于如何更高效地适应新世界，还远远没有明晰。

众多可选的数据存储方案导致了碎片化。Java 开发人员以前会将应用中的数据存储到关系型数据库之中，相对于此，新的存储形式需要开发者付出更多的努力才行。

本书将会帮助你解决这个现实问题。它能够帮助你全面了解当前的硬件条件下存储领域的现状，并阐述对于现代的业务问题，NoSQL 为什么那么重要。

由于保守的企业级市场对语言的认同度较低（也可能是因为 Java 对象-关系映射（ORM）解决方案的复杂性），Java 开发人员传统上往往不会涉及到 NoSQL 领域。还好，现在状况发生了变化，进而产生了这本重要和及时的图书。Spring Data 是一个重要的项目，它能够帮助开发人员征服新的挑战。

Spring 带来了众多价值，因此其成为企业级 Java 开发人员首选的平台，在持久化方案碎片化的现状下它能够带来特殊的收益。Spring 所提供的一部分价值就在于能够以一致的方式集成不同的技术（并不会将所有人拉低到一个最低的通用标准）。独特的“Spring 方式（Spring way）”有助于降低开发人员的学习曲线并简化代码的维护。如果你已经熟悉 Spring，就会发现 Spring Data 能够帮助你探索和采用并不是非常熟悉的数据存储形式。如果你对 Spring 还不熟悉的话，这是一个很好的机会去了解 Spring 如何简化代码并使其保持更加一致。

本书作者是最有资格阐述 Spring Data 的人，因为他们就是项目的领导者。他们对 Spring 有深刻的理解，同时将多种现代数据存储紧密结合了起来。他们很好地阐述了 Spring Data 背后的驱动力以及如何延续 Spring 一贯的数据访问方式。另外，本书涵盖了 Spring Data 如何与其他 Spring 组成部分协同工作这部分内容，如 Spring Integration 和 Spring Batch。书中还提供了除 Spring 之外其他很有意义的内容，如 Repository 的理念、类型安全查询的优点以及为什么 Java 持久化 API（Java Persistence API，JPA）并不适合作为通用的数据访问解决方案。

尽管本书主要讨论的是数据访问而不是 NoSQL 的使用，但是你也会发现很有价值的 NoSQL 资料，因为它所介绍的概念和代码可能是你并不熟悉的。所有的内容都是最新的，重要的内容包括文档数据库、图数据库、键 / 值存储、Hadoop 以及

Gemfire 数据结构。

我们这些编程人员通常追求实用，并且亲手实践时能够学得更好。因此，本书充分考虑到了实用性。在开始的时候，作者首先介绍了如何在业界领先的集成开发环境（Integrated Development Environment, IDE）中运行示例代码，甚至包含了方便学习的屏幕截图，还阐述了数据库驱动方面的需求以及数据库的基本安装过程。他们将代码托管在了 GitHub 上，这一点我很赞赏，这样每个人都可以对其进行访问和浏览。本书涵盖了众多的主题，所以要设计良好的样例，以便将所有内容尽量关联起来。

本书有一章介绍了 Spring Roo，这也能够证明他们是非常重视实际开发的，Spring Roo 是 Spring 提供的快速应用开发（Rapid Application Development, RAD）解决方案。大多数 Roo 的用户已经熟悉如何将其用于传统的 JPA 架构之中，作者展示了 Roo 所带来的生产率提升不仅限于关系型数据库。

读完本书之后，你会深刻理解现代的数据访问为何更加专门化和碎片化、NoSQL 数据存储的主要分类、Spring Data 如何帮助 Java 开发人员在新的环境下高效工作以及对于感兴趣的话题要到什么地方去寻找深入学习的资料。但最为重要的是，这是探索代码的绝佳起点。

Rod Johnson

Spring 框架创始人

# 前言

## 数据访问领域现状概述

数据访问领域在过去的 7 年间发生了重要的变化。过去 30 年间一直占据企业级数据存储和处理核心位置的关系型数据库已经不能再独领风骚了。在过去的 7 年间诞生了很多可选的数据存储形式，当然也有的面临着消亡，它们被使用到了带有关键任务的企业级应用程序之中。这些新的数据存储形式是为了解决特定的数据访问问题而设计的，使用关系型数据库通常无法高效地解决这些问题。

将关系型数据库推到拐点的一个问题就是扩展性（scale）。试问，我们如何将几百甚至几千 TB (terabyte) 的数据存储到关系型数据库中？这个问题让我想到了一个笑话，病人说：“大夫，我一这样动就疼”而医生则说：“那就别这样动呗！”暂且把笑话放在一边，存储如此巨量数据的推动力是什么呢？在 2001 年，IDC 报告说“人们创造和复制的数据将会超过 1.8ZB (zettabytes)，而且每隔两年就会翻番<sup>1</sup>”。新的数据涵盖各种类型，如媒体文件、日志文件、传感器数据 (RFID、GPD、遥测设备……)、Twitter 上的消息以及 Facebook 上的帖子。尽管对于企业来讲存储于关系型数据库中的数据依然非常重要，但是这些新型的数据并没有存储在关系型数据库之中。

一般用户所关注的需求是存储大量的媒体文件，而企业却发现存储和分析这些新型数据的重要性。在美国，各行各业的公司所存储的数据超过了 100TB，有的公司的数据甚至超过了 1PB (petabyte)<sup>2</sup>。大家的共识是持续地分析这些数据会为商业上带来明显的收益。例如，如果产品能够自动汇报其状况，那么公司就可以更容易地掌握产品的行为。为了更好地理解客户，公司在决策制定的过程中可以吸收社交媒体的数据。这甚至引起了主流媒体的报道，例如，Orbitz 发现 Mac 用户偏好较为昂贵的酒店 (<http://on.wsj.com/UhSlNi>)，而 Target 会预测其客户家生孩子的时间 (<http://www.nytimes.com/2012/02/19/magazine/shopping-habits.html>)，从而能够在公开的出生记录发布之前给客户邮寄优惠券。

**大数据 (Big Data)** 通常指的是这样一种流程：存储大量的数据、保持其原始格式、持续地进行分析并且会与其他的数据源结合起来提供某个领域更深入的理解，这种

<sup>1</sup> IDC; *Extracting Value from Chaos* (<http://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf>) 2011

<sup>2</sup> IDC; US Bureau of Labor Statistics

领域可能是商业上的也可能是自然科学上的。

很多的公司和科研实验室在大数据这个词流行起来之前就开始这样做了。当前的过程与以往的不同在于，智能的数据分析所带来的价值要高于硬件的成本。现在执行这种类型的分析不再需要购买4万美元一颗的CPU了；商用的硬件集群中每颗CPU的价格是1000美元。对于大型的数据集，存储区域网络（Storage Area Network, SAN）以及网络附属存储（Network Attached Storage, NAS）的价格较为昂贵：每GB（gigabyte）是1~10美元，如果副本构建到数据库中而不是硬件之中，那本地磁盘的成本每GB只有0.05美元。对于商用的硬件集群，使用本地磁盘的数据传输率也要比基于SAN或NAS的系统更高——对于相同价格的系统，前者能快500倍。在软件方面，新的数据访问技术大多数都是开源的。尽管开源并不意味着零成本，但是这显然会降低使用门槛，相对于传统的商业软件，它们能够降低采购的整体成本。

另一个能够区分新型数据存储与关系型数据库的问题域就是关系型的数据模型。如果你想分析上百万人的社交图谱，图形数据库更接近这个领域的模型，使用这种数据库难道不是很自然的事情吗？如果需求持续地要求你修改关系型数据库管理系统（RDBMS）的模式（schema）以及对象关系映射（ORM）层，那该怎么办呢？可能“无模式（schema-less）”的文档数据库能够减少对象映射的复杂度，相对于僵化的关系型模型，它所提供的系统更易于演化。尽管每种不同的数据库各有其独特之处，但是可以基于其数据模型进行大致分类。基本的分类情况如下：

## 键/值

我们所熟悉的数据模型，类似于哈希表（hashtable）。

## 列族

扩展的键/值数据模型，值的数据类型也可以是键/值对的序列。

## 文档

半结构化数据的集合，如XML或JSON。

## 图

基于图论，数据模型中包括节点（node）和边（edge），它们都可以包含属性（property）。

这些新的数据库都可以归类在“**NoSQL数据库**”之下。回顾历史，这个名字尽管朗朗上口，但是并不精确，因为它容易让人觉得这些数据库不能进行查询，但事实并不是这样。它的基本含义是摆脱关系型数据模型以及关系型数据库的ACID特性（原子性、一致性、隔离性以及持久性）。

要摆脱ACID特性的主要驱动力在于，很多的应用程序提高了可扩展写（*scaling write*）的优先级，并且希望即便系统的某一部分失效，其他部分依然可以继续运作。尽管在关系型数据库中，可以通过在数据库之前使用内存缓存来实现可扩展读，但是进行可扩展写要困难得多。为了标识这类应用程序，通常将其命名为“BASE”系统，在这里缩写代表着基本可用（*basically available*）、可扩展性（*scalable*）、最终一致性（*eventually consistent*）。具有键/值数据模型的分布式数据网格并没有归类到这种新的NoSQL数据库之中。然而，它们提供了与NoSQL数据库类似的特性，如数据的可扩展性以及组合计算能力和数据的分布式计算功能。

从上面简短的介绍中，你能够了解到数据访问的现状，目前正在发生的是—场革命，关注数据的人会非常兴奋。关系型数据库并没有消亡，在很多企业的运作中它依然是核心，并且会持续很长的时间。但是，趋势很明显：新的数据访问技术解决了关系型数据库所无法解决的问题，因此作为开发人员，我们必须要扩充自己的技能，要能够处理这两种技术。

Spring 框架长期以来都致力于简化 Java 应用程序的开发，尤其是使用 Java 数据库连接（Java Database Connectivity， JDBC）或对象关系映射器编写基于 RDBMS 的数据访问层方面。在本书中，我们力图帮助开发人员使用这些新技术高效地编写 Java 应用程序。Spring Data 项目直接处理这些新的技术，因此你能够将已有的 Spring 知识延伸到它们之中，或者通过使用 Spring Data，也能够更深入地学习 Spring。不过，我们也没有抛弃关系型数据库。Spring Data 为了 Spring 能支持 RDBMS 扩展了新功能。

## 如何阅读本书

本书希望为你提供实用的Spring Data项目的介绍，该项目的目的在于帮助Java开发人员使用最新的数据处理和管理工具，同时能够以最新的方式使用传统的数据库。我们首先会为你介绍这个项目，阐述SpringSource以及该团队背后的驱动力，并描述了示例工程的领域模型，这个例子将会在后面的章节中使用，此外还介绍了如何获取和搭建示例代码（第1章）。

接着将会讨论Spring Data Repository的基本理念，因为它们是后续各种数据存储中都会涉及的通用主题（第2章）。对于Querydsl也是如此，我们将会在第3章中讨论这个话题。这两章为探讨特定存储与Repository的集成以及高级查询功能奠定了坚实的基础。

为了从Java开发人员熟悉的领域开始，我们会用一些时间来介绍传统的持久化技术，如JPA（第4章）和JDBC（第5章）。这两章阐述了在Spring已有的对JPA和JDBC的支持之上，Spring Data的模块提供了什么新特性。

在完成这些之后，将会引入Spring Data项目所支持的NoSQL存储：以MongoDB为例介绍文档数据库（第6章），以Neo4j为例讲解图数据库（第7章），以Redis为例讲解键/值存储（第8章）。列族数据库HBase将会在稍后的章节中介绍（第12章）。这

些章节中涵盖的内容包括了将领域类映射到特定存储的数据结构、通过所提供的应用程序编程接口（Application Programming Interface, API）与存储进行便捷地交互以及对Repository抽象的使用。

我们将会介绍Spring Data REST导出器（第10章）以及Spring Roo集成（第9章）。这两章的内容都基于Repository抽象并且能够很容易地将Spring Data所管理的实体导出到Web之中，可以作为表达性状态转移（Representational State Transfer, REST），也可以作为基于Spring Roo所构建的Web应用程序。

接下来本书将会进入大数据的世界——具体来说也就是Hadoop和Spring for Apache Hadoop，我们将会介绍如何使用Hadoop技术实现样例以及Spring Data模块如何明显简化Hadoop的使用（第11章）。基于这些内容，将会使用Spring Batch和Spring Integration项目来构建复杂的大数据管道——这些项目都非常适合于大数据的处理（第12章和第13章）。

最后一章将会讨论 Spring Data 对 GemFire 的支持，这是一个分布式的数据网格解决方案（第 14 章）。

## 本书中的约定

本书中使用了如下的排版约定：

**斜体字 (*Italic*)**

表示新的术语、URL、电子邮箱、文件名以及文件扩展名。

**等宽字体 (Constant width)**

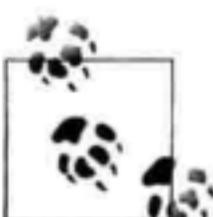
用于程序清单，以及在段落中对程序中元素的引用，如变量或函数名、数据库、数据类型、环境变量、语句和关键字。

**加粗的等宽字体 (Constant width bold)**

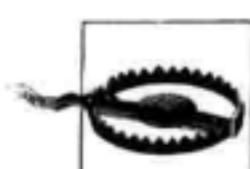
表示需要用户输入的命令或其他文本。

**斜体的等宽字体 (Constant width italic)**

表示这些文本需要根据用户提供的值或上下文确定的值进行替换。



这个图标代表提示、建议或一般说明。



这个图标代表警告或提醒。

## 代码示例

这本书的目的在于帮助你做好工作。一般来说，可以在程序或文档中使用本书的代码，除非复制了本书中大部分的代码，否则不需要获得我们的许可。例如，如果你写了一个程序，使用了本书中的几段代码是不需要许可的。销售和发行刻有O'Reilly图书中示例的光盘（CD-ROM）则需要得到许可。如果引用本书和利用书中的示例代码来回答问题是不需要许可的。但是，如果在你的产品文档中大量使用本书的示例代码，那么就需要许可了。

我们赞赏大家在使用代码时注明信息来源，但这并不是强求的。信息来源通常要包括标题、作者、出版社以及ISBN。比如“Spring Data by Mark Pollack, Oliver Gierke, Thomas Risberg, Jon Brisbin, and Michael Hunger (O'Reilly). Copyright 2013 Mark Pollack, Oliver Gierke, Thomas Risberg, Jonathan L. Brisbin, and Michael Hunger, 978-1-449-32395-0”。

如果你觉得对示例代码的使用超出了合理引用或上面给出的许可范围，可以通过permissions@oreilly.com联系我们。

本书代码示例已发布在GitHub上 (<https://github.com/SpringSource/spring-data-book>)。

## Safari®在线书店

 Safari在线书店 ([www.safaribooksonline.com](http://www.safaribooksonline.com)) 是应需而变的数字图书馆，它能够以书籍或视频的形式提供专家级的内容 (<http://www.safaribooksonline.com/content>)，这些内容来自于世界范围内领先的技术和商业领域的作者。

技术专家、软件开发人员、Web设计师以及商业和创意人士都选择Safari在线书店作为研究、解决问题、学习以及认证培训的主要信息来源。

Safari在线书店针对组织 (<http://www.safaribooksonline.com/organizations-teams>)、政府部门 (<http://www.safaribooksonline.com/government>) 以及个人 (<http://www.safaribooksonline.com/individuals>) 提供了多种产品 (<http://www.safaribooksonline.com/subscriptions>) 以及定价策略。订阅者可以通过一个完全可检索的数据库访问上千种图书、培训视频以及尚未出版的书稿，这些内容来自像O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology这样的几十家出版社 (<http://www.safaribooksonline.com/publishers>)。想了解Safari在线书店的更多信息，请访问在线站点 (<http://www.safaribooksonline.com/>)。

## 如何联系我们

请把对本书的评论和问题发给出版社。

美国：

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）

奥莱利技术咨询（北京）有限公司

O'Reilly的每一本书都有专属网页，可以在那找到关于本书的相关信息，包括勘误、样例以及其他信息。可以通过以下地址访问该页面：

<http://oreil.ly/spring-data-1e>。

对于本书的评论和技术性的问题，请发送电子邮件到：

[bookquestions@oreilly.com](mailto:bookquestions@oreilly.com)

关于本书的更多信息、会议、资料中心和网站，请访问以下网站：

<http://www.oreilly.com>

想了解我们的图书、培训课程、会议以及新闻等信息，可以访问我们的主页：

<http://www.oreilly.com>

<http://www.oreilly.com.cn>

## 致谢

我们要感谢Rod Johnson和Emil Eifrem，正是他们开启了Spring Data项目。

非常感谢David Turanski的贡献，他帮助我们编写了GemFire这一章。感谢Richard McDougall对大数据的分析，我们在本书的前言中用到了它，感谢Costin Leau帮忙编写Hadoop示例应用程序。

我们还要感谢O'Reilly Media，尤其是带领这个项目的Meghan Blanchette，制作编辑Kristen Borg以及文字编辑Rachel Monaghan。感谢Greg Turnquist、Joris Kuipers、Johannes Hiemer、Joachim Arrasz、Stephan Hochdörfer、Mark Spritzler、Jim Webber、Lasse Westh-Nielsen以及所有为本书提供技术审阅的人。感谢这个项目的社区，你们提交的反馈和缺陷促使我们不断进步。最后，还要特别感谢朋友和家人的宽容、理解和支持。

---

# 关于作者

**Mark Pollack**博士曾在布鲁克黑文国家实验室研究高能物理学方面的大数据解决方案，随后转移到金融服务领域担任前端交易系统的技术领导和架构师。他长期关注软件开发流程的最佳实践和改善，2003年就参与了核心Spring（Java）的开发，并在2004年成立了Microsoft对应的项目也就是Spring.NET。Mark现在领导着Spring Data项目，在使用大数据和NoSQL数据库这些新技术时，这个项目能够简化应用的开发。

**Oliver Gierke**是SpringSource的工程师，这是VMware<sup>1</sup>的一个子部门，目前他担任Spring Data JPA、MongoDB以及核心模块的领导者。他参与企业级应用和开源项目的开发已经超过了6年，其工作的关注点在软件架构、Spring以及持久化技术方面。他经常在德国以及一些国际会议上进行演讲，写过很多技术文章。

**Thomas Risberg**目前是Spring Data团队的成员，关注于MongoDB和JDBC扩展项目。他也是Spring框架项目的提交者，主要的贡献在于对JDBC框架的增强。Thomas在VMware的Cloud Foundry团队，为Cloud Foundry所支持的各种框架和语言开发集成方案。他是《Professional Java Development with the Spring Framework》<sup>2</sup>一书的合著者，这本书出版于2005年，作者还包括Rod Johnson、Juergen Hoeller、Alef Arendsen以及Colin Sampaleanu。

**Jon Brisbin**是SpringSource Spring Data团队的成员，致力于为开发人员提供有用的库，从而简化下一代数据形式的管理。他曾帮忙将Grails GORM的对象映射器转移到基于Java的应用程序之中，并为集成Riak数据存储与RabbitMQ消息代理提供组件。除此之外，他还针对事件应用模型撰写博客并发表演讲，他勤奋工作的领域还包括为前沿的非阻塞模式和传统的基于JVM的应用搭建桥梁。

**Michael Hunger**长期热衷于软件开发。他尤其关注开发软件的人、软件技艺、编程语言以及代码效率提升。最近两年间，他与Neo Technology协作开发Neo4j图数据库。作为Spring Data Neo4j的领导者，他为对象-图映射开发出了便利且完整的解决方案。他还参与Neo4j云托管。作为一名开发人员，Michael喜欢使用各种编程语言、每天学习新的东西、参与有趣且有前景的开源项目，并且参与编写了多本与编程相关的图书。Michael还是InfoQ活跃的编辑和采访者。

---

<sup>1</sup> 译者注：目前 SpringSource 和 Cloud Foundry 均已经被转移到了 Pivotal 之中。

<sup>2</sup> 译者注：本书中文名为《Java 框架高级编程》。

---

# 译者简介

张卫滨，软件工程师，InfoQ 社区编辑，熟悉 Java 语言，对 Java 开源框架有一定研究，如 Spring、Hibernate 以及 Eclipse 等，熟悉 Web 前端开发，了解相关技术以及 jQuery、Dojo、ExtJS 等框架，目前主要从事企业级软件的开发。译有《Spring 实战（第 3 版）》以及《Java 应用架构设计：模块化模式与 OSGi》。

文建国，系统架构师，精通 Spring 等优秀开源技术在企业中的应用，主要研究方向为云计算、大数据、业务基础平台、分布式等技术。曾参与过中国电信 ITSP 3.0 技术架构规范编写，拥有多个大型云计算项目的架构和管理经验。

---

# 目录

## 第一部分 背景知识

<b>第 1 章 Spring Data 项目 .....</b>	<b>3</b>
1.1 为 Spring 开发人员提供的 NoSQL 数据访问功能.....	3
1.2 主题概述 .....	4
1.3 领域 .....	5
1.4 示例代码 .....	6
1.4.1 将源码导入到 IDE .....	6
<b>第 2 章 Repository：便利的数据访问层 .....</b>	<b>11</b>
2.1 快速入门 .....	11
2.2 定义查询方法 .....	14
2.2.1 查找查询的策略 .....	14
2.2.2 衍生查询 .....	14
2.2.3 分页和排序 .....	15
2.3 定义 Repository .....	16
2.3.1 调整 Repository 接口 .....	17
2.3.2 手动实现 Repository 方法 .....	18
2.4 IDE 集成 .....	20
2.4.1 IntelliJ IDEA .....	21
<b>第 3 章 使用 Querydsl 实现类型安全的查询 .....</b>	<b>23</b>
3.1 Querydsl 简介 .....	23
3.2 生成查询元模型 .....	26
3.2.1 构建系统集成 .....	26
3.2.2 所支持的注解处理器 .....	27
3.2.3 使用 Querydsl 对存储进行查询 .....	28
3.3 集成 Spring Data Repository .....	28
3.3.1 执行断言 .....	29
3.3.2 手动实现 Repository .....	29

## 第二部分 关系型数据库

第 4 章 JPA Repository .....	33
4.1 示例工程 .....	33
4.2 传统方式 .....	38
4.3 启动示例代码 .....	39
4.4 使用 Spring Data Repository.....	42
4.4.1 事务性 .....	45
4.4.2 Repository 与 Querydsl 集成.....	46
第 5 章 借助 Querydsl SQL 实现类型安全的 JDBC 编程 .....	48
5.1 示例工程与搭建过程 .....	48
5.1.1 HyperSQL 数据库.....	49
5.1.2 Querydsl 的 SQL 模块 .....	50
5.1.3 构建系统集成 .....	53
5.1.4 数据库模式 .....	54
5.1.5 示例工程的领域实现 .....	54
5.2 QueryDslJdbcTemplate.....	57
5.3 执行查询 .....	58
5.3.1 Repository 实现起步 .....	59
5.3.2 查询单个对象 .....	60
5.3.3 OneToManyResultSetExtractor 抽象类 .....	61
5.3.4 CustomerListExtractor 实现.....	63
5.3.5 RowMapper 的实现类 .....	64
5.3.6 查询对象列表 .....	65
5.4 插入、更新和删除操作 .....	65
5.4.1 使用 SQLInsertClause 进行插入操作.....	65
5.4.2 使用 SQLUpdateClause 进行更新操作 .....	66
5.4.3 使用 SQLDeleteClause 进行删除行操作 .....	67

## 第三部分 NoSQL

第 6 章 MongoDB: 文档存储 .....	71
6.1 MongoDB 简介 .....	71
6.1.1 设置 MongoDB .....	72
6.1.2 使用 MongoDB Shell .....	73

6.1.3	MongoDB Java 驱动 .....	74
6.2	使用 Spring 命名空间搭建基础设施 .....	75
6.3	映射模块 .....	76
6.3.1	领域模型 .....	76
6.3.2	搭建映射的基础设施 .....	81
6.3.3	索引 .....	83
6.3.4	自定义转换 .....	84
6.4	MongoTemplate .....	86
6.5	Mongo Repository .....	88
6.5.1	搭建基础设施 .....	88
6.5.2	Repository 详解 .....	88
6.5.3	Mongo Querydsl 集成 .....	90
<b>第 7 章</b>	<b>Neo4j：图数据库 .....</b>	<b>92</b>
7.1	图数据库 .....	92
7.2	Neo4j .....	93
7.3	Spring Data Neo4j 概览 .....	95
7.4	将领域建模为图 .....	96
7.5	使用 Spring Data Neo4j 持久化领域对象 .....	101
7.5.1	Neo4jTemplate .....	103
7.6	组合发挥图和 Repository 的威力 .....	104
7.6.1	基本的图 Repository 操作 .....	106
7.6.2	衍生和基于注解的查找方法 .....	106
7.7	示例领域模型中的高级图用例 .....	109
7.7.1	单个节点的多重角色 .....	109
7.7.2	以产品分类和标签为例讲解图中的索引 .....	110
7.7.3	利用类似的兴趣（协同过滤） .....	111
7.7.4	推荐 .....	111
7.8	事务、实体生命周期以及抓取策略 .....	112
7.9	高级映射模型 .....	113
7.10	使用 Neo4j 服务器 .....	114
7.11	从这里继续学习 .....	115
<b>第 8 章</b>	<b>Redis：键/值存储 .....</b>	<b>116</b>
8.1	Redis 概述 .....	116
8.1.1	搭建 Redis .....	116
8.1.2	使用 Redis Shell .....	117
8.2	连接到 Redis .....	118

8.3 对象转换 .....	119
8.4 对象映射 .....	121
8.5 原子级计数器 .....	123
8.6 发布/订阅功能 .....	123
8.6.1 对信息进行监听和响应 .....	124
8.6.2 在 Redis 中使用 Spring 的缓存抽象 .....	125
<b>第四部分 快速应用开发</b>	
<b>第 9 章 使用 Spring Roo 实现持久层.....</b>	<b>129</b>
9.1 Roo 简介 .....	129
9.2 Roo 的持久层.....	131
9.3 快速起步 .....	132
9.3.1 借助命令行使用 Roo.....	132
9.3.2 借助 Spring Tool Suite 使用 Roo .....	133
9.4 Spring Roo JPA Repository 示例 .....	135
9.4.1 创建工程 .....	135
9.4.2 搭建 JPA 持久化.....	135
9.4.3 创建实体 .....	135
9.4.4 定义 Repository.....	137
9.4.5 创建 Web 层 .....	138
9.4.6 运行示例 .....	139
9.5 Spring MongoDB JPA Repository 的例子.....	140
9.5.1 创建工程 .....	140
9.5.2 搭建 MongoDB 持久化.....	140
9.5.3 创建实体 .....	140
9.5.4 定义 Repository.....	141
9.5.5 创建 Web 层 .....	141
9.5.6 运行示例 .....	141
<b>第 10 章 REST Repository 导出器 .....</b>	<b>143</b>
10.1 示例工程 .....	144
10.1.1 与 Rest 导出器进行交互 .....	146
10.1.2 访问 Product.....	148
10.1.3 访问 Customer.....	151
10.1.4 访问 Order .....	154

## 第五部分 大数据

第 11 章 Spring for Apache Hadoop .....	159
11.1 Hadoop 开发面临的挑战 .....	159
11.2 Hello World.....	161
11.3 揭秘 Hello World .....	163
11.4 使用 Spring for Apache Hadoop 的 Hello World.....	166
11.5 在 JVM 中编写 HDFS 脚本.....	170
11.6 结合 HDFS 脚本与 Job 提交 .....	172
11.7 Job 调度.....	173
11.7.1 使用 TaskScheduler 调度 MapReduce Job .....	173
11.7.2 使用 Quartz 调度 MapReduce Job .....	175
第 12 章 使用 Hadoop 分析数据.....	176
12.1 使用 Hive .....	176
12.1.1 Hello World .....	177
12.1.2 运行 Hive 服务器 .....	178
12.1.3 使用 Hive Thrift 客户端 .....	179
12.1.4 使用 Hive JDBC 客户端 .....	181
12.1.5 使用 Hive 分析 Apache 日志文件 .....	183
12.2 使用 Pig.....	184
12.2.1 Hello World .....	185
12.2.2 运行 PigServer .....	187
12.2.3 控制运行期脚本的执行.....	189
12.2.4 在 Spring Integration 数据管道中调用 Pig 脚本.....	191
12.2.5 使用 Pig 分析 Apache 日志文件 .....	192
12.3 使用 HBase .....	193
12.3.1 Hello World .....	193
12.3.2 使用 HBase Java 客户端 .....	194
第 13 章 使用 Spring Batch 和 Spring Integration 创建大数据管道.....	197
13.1 收集并将数据加载到 HDFS.....	197
13.1.1 Spring Integration 介绍.....	198
13.1.2 复制日志文件 .....	200
13.1.3 事件流 .....	203
13.1.4 事件转发 .....	206
13.1.5 管理 .....	207

13.1.6	Spring Batch 简介 .....	208
13.1.7	从数据库中加载并处理数据 .....	211
13.2	Hadoop 工作流.....	214
13.2.1	Spring Batch 对 Hadoop 的支持 .....	214
13.2.2	将 wordcount 样例改造为 Spring Batch 应用 .....	216
13.2.3	Hive 和 Pig 的步骤 .....	218
13.3	从 HDFS 导出数据 .....	219
13.3.1	从 HDFS 到 JDBC .....	219
13.3.2	从 HDFS 到 MongoDB.....	224
13.4	收集并加载数据到 Splunk.....	225

## 第六部分 数据网格

第 14 章	分布式数据网格: GemFire .....	231
14.1	GemFire 简介 .....	231
14.2	缓存与域 .....	232
14.3	如何获取 GemFire .....	233
14.4	通过 Spring XML 命名空间配置 GemFire .....	234
14.4.1	缓存配置 .....	234
14.4.2	域配置 .....	238
14.4.3	缓存客户端配置 .....	240
14.4.4	缓存服务端配置 .....	241
14.4.5	WAN 配置 .....	242
14.4.6	磁盘存储配置 .....	243
14.5	使用 GemfireTemplate 进行数据访问 .....	244
14.6	使用 Repository .....	245
14.6.1	POJO 映射 .....	245
14.6.2	创建 Repository .....	246
14.6.3	PDX 序列化 .....	246
14.7	支持持续查询 .....	247

**第一部分**

---

**背景知识**



# Spring Data 项目

Spring Data 项目是在“Spring One 2010 开发者大会”上创建的，该项目起源于当年早些时候 Rod Johnson (SpringSource) 和 Emil Eifrem (Neo Technologies) 共同参与的一场黑客会议。他们试图把 Neo4j 图形数据库整合到 Spring 框架中，并评估了各种不同的方式。这次会议最终为初始版本的 Spring Data Neo4j 模块奠定了基础，这个新的 SpringSource 项目旨在迎合大众对于 NoSQL 数据存储日益增长的兴趣，而这种趋势一直持续到了今天。

从创立之初，Spring 就为传统的数据访问技术提供了完善的支持。不管是使用 JDBC、Hibernate、JDO、TopLink 还是 iBatis 作为持久化技术，Spring 都大大简化了数据访问层的实现。这种支持主要包括简化基础配置、资源管理并将异常转换成 Spring 的 **DataAccessExceptions**。这种支持多年以来已经逐渐成熟，最新版本的 Spring 也对这一层提供了很好的支持。

过去涉及数据持久化时，关系型数据库是可供选择的主要工具，所以 Spring 对传统数据访问的支持只把关系数据库作为唯一的目标。但随着 NoSQL 的问世并成为工具箱中可行的替代方案，从支持开发人员的角度来看就有了新的领域需要补充。另一方面，对于传统关系型存储的支持也还有许多需要改善的地方。这两个方面是 Spring Data 项目的主要驱动力。Spring Data 包含 NoSQL 存储的专有模块以及为关系型数据库提供更好支持的 JPA 和 JDBC 模块。

## 1.1 为 Spring 开发人员提供的 NoSQL 数据访问功能

尽管用 NoSQL 这个术语统称一系列的新型数据存储，但所有的这些存储都有不同的特性和使用场景。具有讽刺意味的是，正是这种缺失特性的特点（缺乏对运行 SQL 查询的支持）命名了这一系列数据库。由于这些存储的特征非常不同，所以

它们的 Java 驱动要使用完全不同的 API 才能充分发挥其特性和功能。如果试图对这些差异进行抽象的话，就会失去每种 NoSQL 数据存储能带来的收益。图形数据库应该用来存储高度关联的数据；文件数据库应该存储树状以及聚合状的数据结构；如果需要类似缓存的功能和存取模式，那应该选择键/值（key/value）存储。

Java EE（企业版）领域通过 JPA 提供了持久化 API，这个 API 或许可以当作 NoSQL 数据库前端实现的候选方案。但是令人遗憾的是，规范的前两句话已经预示了这一点似乎不可能实现：

本文档是关于在 Java EE 和 Java SE 中管理持久化和对象/关系映射的 Java API 规范。这项成果的技术目标是为 Java 应用开发人员提供一个对象/关系映射机制，借助它可以使用域模型来管理关系型数据库。

这一主题在规范的后面有清晰的体现，它定义了与关系型持久化领域紧密关联的概念和 API。**@Table** 注解对 NoSQL 数据库而言没有太大的意义，**@Column** 或 **@JoinColumn** 也是如此。在 MongoDB 这样的存储中如何实现事务 API 呢？要知道在这些环境中并没有提供跨多文档操作的事务语义。因此在 NoSQL 存储之上实现 JPA 层，最好采用一个基于配置文件的 API。

另一方面，所有 NoSQL 存储提供的特殊功能（地理空间功能、map-reduce 操作、图形遍历）都需要以专有的方式去实现，因为 JPA 并没有为它们提供抽象。因此我们可能会以“两边不讨好”（worst-of-both-world）的局面收场：有的部分可以通过 JPA 来实现，另外还需要使用专有的特性来重新启用与特定存储相关的功能。

上文排除了采用 JPA 作为这些存储的抽象 API 的可能性。Spring 生态系统中的各种项目为开发人员带来了高效率以及一致的编程模型，我们依然希望将其用于简化对 NoSQL 存储的使用。为此，Spring Data 团队发布了如下使命宣言：

针对 NoSQL 和关系型存储，Spring Data 提供了基于 Spring 的熟知且一致的编程模型，同时保留特定存储的特性和功能。

因此，我们决定采取略有不同的方法。不再试图以单一的 API 将所有存储抽象化。相反，Spring Data 项目对不同的存储实现都提供一致的编程模型，并使用了在 Spring 框架中大家已熟悉的模式和抽象。这样，在使用不同的存储时，会有一致的体验。

## 1.2 主题概述

Spring Data 的核心目标是：支持对所有的存储进行资源配置，从而实现对该存储的访问。这种支持主要是通过 XML 命名空间和 Spring JavaConfig 的支持类实现的，这可以使我们轻松地对 Mongo 数据库、嵌入式 Neo4j 实例等建立访问。除此之外，

它也集成了 Spring 的核心功能，如 JMX。这意味着某些存储可以通过原生 API 暴露统计数据，这些数据将会由 Spring Data 暴露给 JMX。

大部分的 NoSQL Java API 并未支持将领域对象映射到存储的数据抽象（MongoDB 中的文件，Neo4j 中的节点与关系）。因此，当使用原生的 Java 驱动程序进行读取和写入操作时，通常需要编写大量的代码来将数据映射到应用程序的领域对象。所以 Spring Data 模块最核心的部分是一个映射和转换的 API，用来获取要持久化的领域类中的元数据，使得任意领域对象都可以转换成存储用的数据类型。

在此基础上，就如同著名的 **Spring JdbcTemplate**、**JmsTemplate** 等，我们也会看到以模板模式实现的 API，其中包括 **RedisTemplate**、**MongoTemplate** 等。或许你已经知道，这些模板提供了让我们可以执行常用操作的辅助方法。例如：在一条语句中持久化一个对象的时候，能够自动进行资源管理和异常处理。此外，还提供了回调接口的 API，允许在资源管理和异常处理过程中使用存储原生的 API，以提高灵活性。

这些功能给我们提供了一个工具箱，使得我们可以像使用传统的数据库那样来实现数据访问层。后面的章节将会详细介绍这些功能。为了让程序变得更简单，Spring Data 还在模板实现的基础上提供了一个存储（repository）抽象，这将减少数据访问对象在实现一个普通接口时去定义通用场景的代价，如标准的 CRUD（创建、读取、更新、删除）操作以及执行存储支持的查询语句。事实上，这种抽象位于最顶层，而且它会尽可能在合理范围内将不同存储 API 融合在一起。因此，存储的操作将拥有许多共同点。这也是后面会有专门的章节（第 2 章）来介绍基本编程模型的原因。

接着，我们来看一下用来展示这些特定存储模块功能所用的示例代码和领域模型。

## 1.3 领域

为了说明各种 Spring Data 模块的用法，我们会使用电子商务部门的示例领域（如图 1-1 所示）。由于各种 NoSQL 数据存储通常具有特定的功能和适用场景，在个别章节会对领域的实现方式做出一些调整，甚至只有它的部分实现，这种做法不代表必须以一种特定的方法来实现领域，而是强调某些存储应该更适用于特定的应用场景。

在模型的核心，有客户（customer），包含客户的基本资料，如姓、名、电子邮箱地址、地址（一组包含街道、城市和国家的集合），还有由产品名称、描述、价格和其他属性构成的产品（product）。这些抽象是组成 CRM（客户关系管理系统）和库存系统的基础。最重要的是客户可以订购订单（Order），订单信息包含订购的客户、邮寄和付款地址、订购时间、订单状态和一组商品明细。而这些商品明细又包含一个特定的产品、订购的数量和产品的价格。

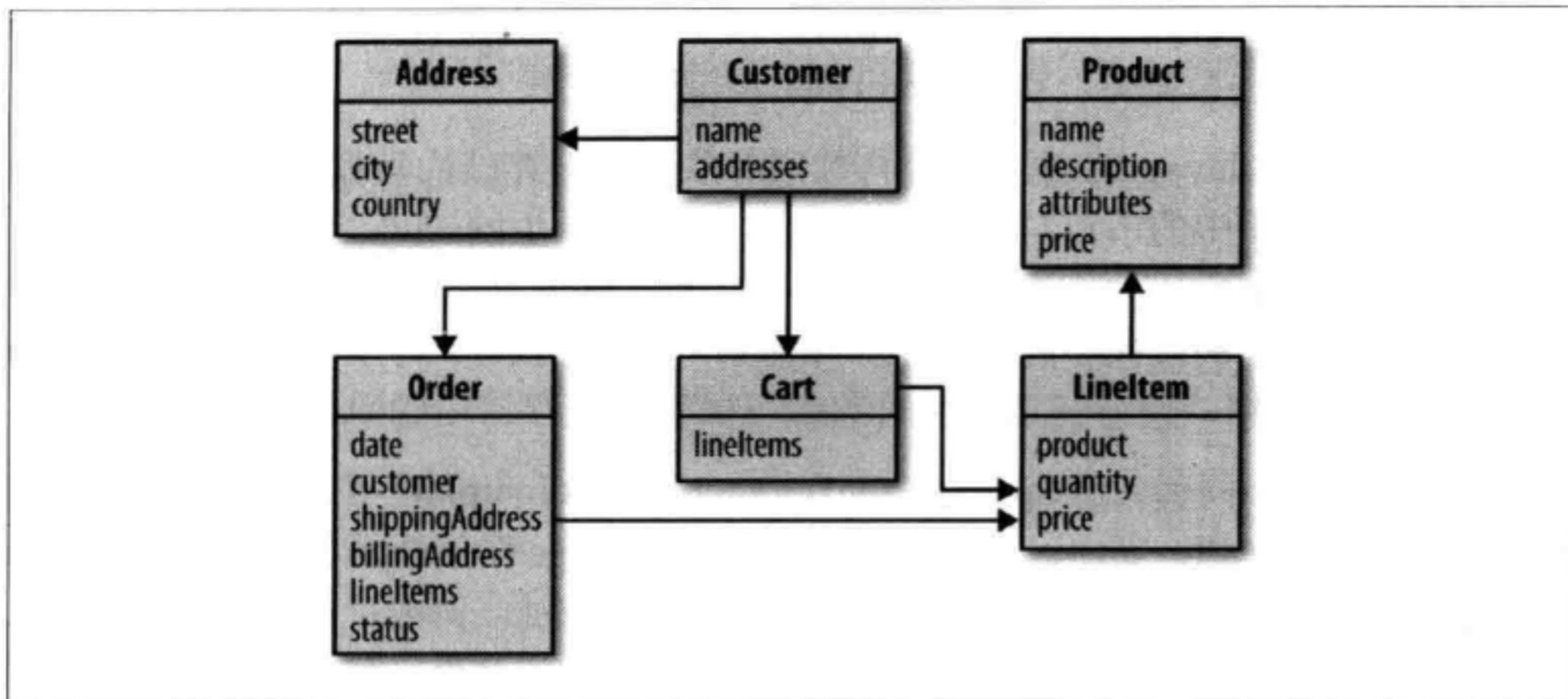


图 1-1 领域模型

## 1.4 示例代码

本书的示例代码可从 GitHub (<https://github.com/SpringSource/spring-data-book>) 上获取。它是一个 Maven 项目，包含每一章的模块。另外，还需要在电脑中安装 Maven 3 或者一个能导入 Maven 项目的 IDE，比如 Spring Tool Suite (STS)。从下面的操作中可以看到，取得示例代码就如同复制版本库一样简单：

```

$ cd ~/dev
$ git clone https://github.com/SpringSource/spring-data-book.git
Cloning into 'spring-data-book'...
remote: Counting objects: 253, done.
remote: Compressing objects: 100% (137/137), done.
Receiving objects: 100% (253/253), 139.99 KiB | 199 KiB/s, done.
remote: Total 253 (delta 91), reused 219 (delta 57)
Resolving deltas: 100% (91/91), done.
$ cd spring-data-book

```

现在可以在命令行中执行 Maven 来构建代码：

```
$ mvn clean package
```

这样 Maven 会解析依赖、编译和测试代码，执行测试，最终打包模块。

### 1.4.1 将源码导入到 IDE

#### STS/Eclipse

由于 STS 已经配备了 m2eclipse 插件，所以可以在 IDE 中轻松使用 Maven 项目。如果已经下载并安装（详情请见第 3 章）了 STS，即可从 File 菜单选择 Import 选项，并在弹出的对话框中选择 Existing Maven Projects，如图 1-2 所示。

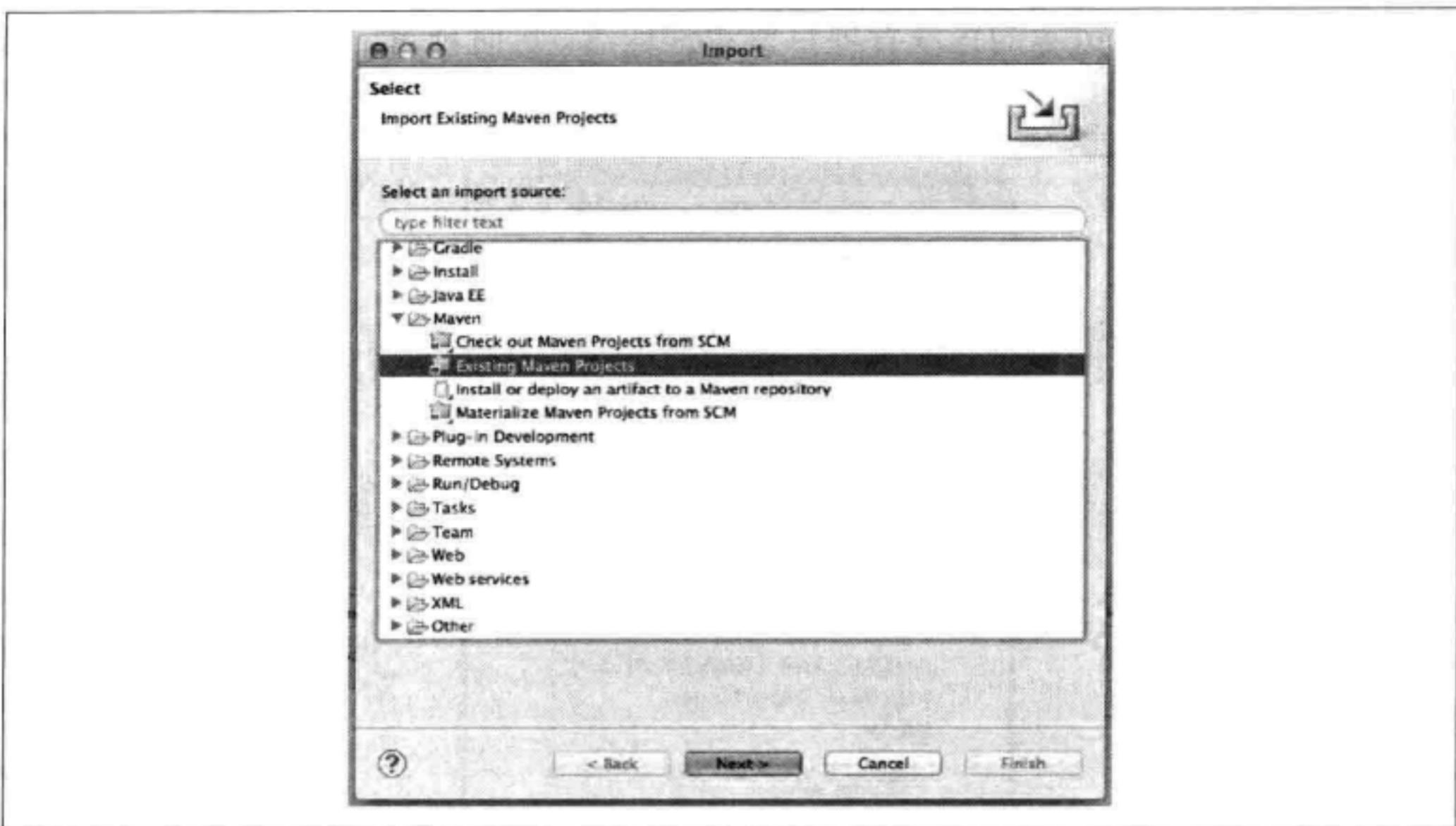


图 1-2 导入 Maven 项目到 Eclipse (步骤 1/2)

在下一个窗口中，单击 Browse 按钮来选择刚刚签出的示例项目的文件夹。之后，在正下方的窗格中会列出并选中各个 Maven 模块（如图 1-3 所示）。单击 Finish 按钮进行下一步，STS 会将选中的 Maven 模块导入到工作区。它将依照模块根目录下的 pom.xml 文件来解析所需的依赖和源文件夹。

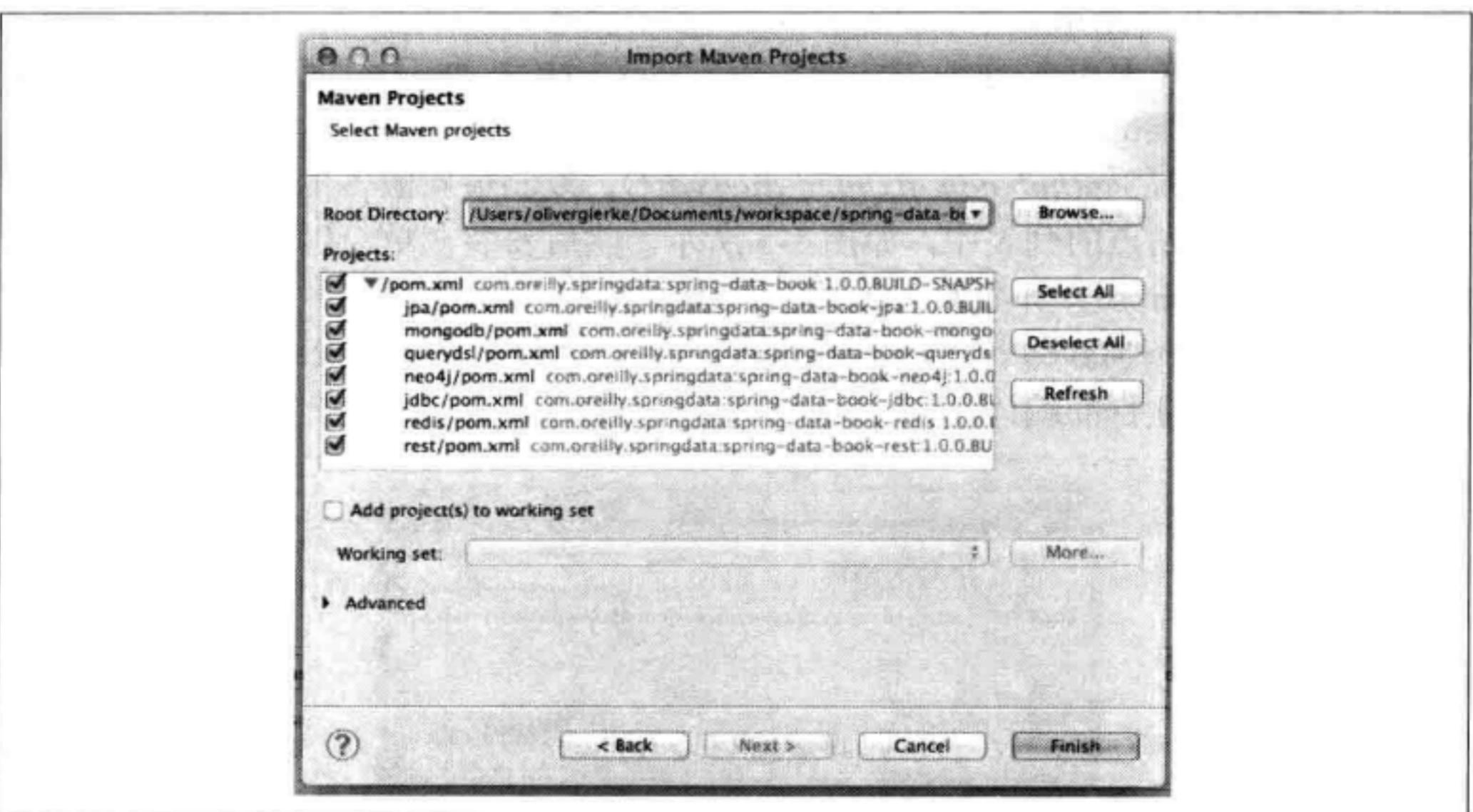


图 1-3 将 Maven 项目导入到 Eclipse (步骤 2/2)

最终会看到如图 1-4 所示的包或者项目资源管理器。这时项目应能成功编译并且不包含红色错误标记。

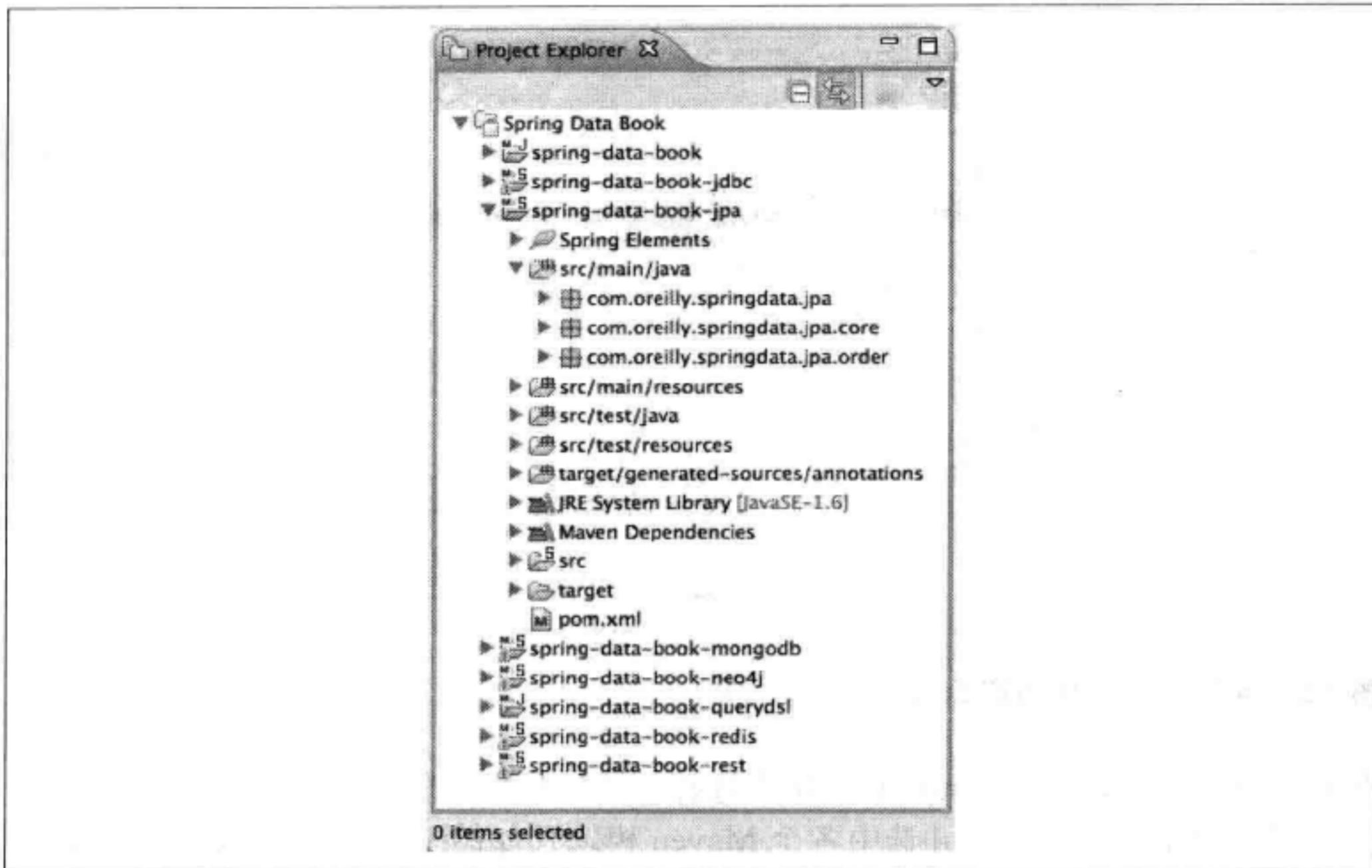


图 1-4 完成导入的 Eclipse Project Explorer

使用了 Querydsl(详见第 5 章)的项目可能会引发红色的错误标记。原因是 m2eclipse 插件需要知道：在 IDE 构建的生命周期中，哪个阶段执行 Querydsl 关联的 Maven 插件。可以从 m2e-querydsl 扩展更新站点来安装这个插件，也可以在项目主页上找最新的版本 (<https://github.com/ilx/m2e-querydsl>)，复制最新版本的链接，并将它添加到可用的更新站点的列表中，如图 1-5 所示。然后安装在更新网站上发布了的功能，重新启动 Eclipse，并更新 Maven 项目配置（在项目中单击鼠标右键，从弹出的快捷菜单中选择 Maven→Update Project），这样就能去除 Eclipse 中的错误标记，并且成功地完成项目的编译。

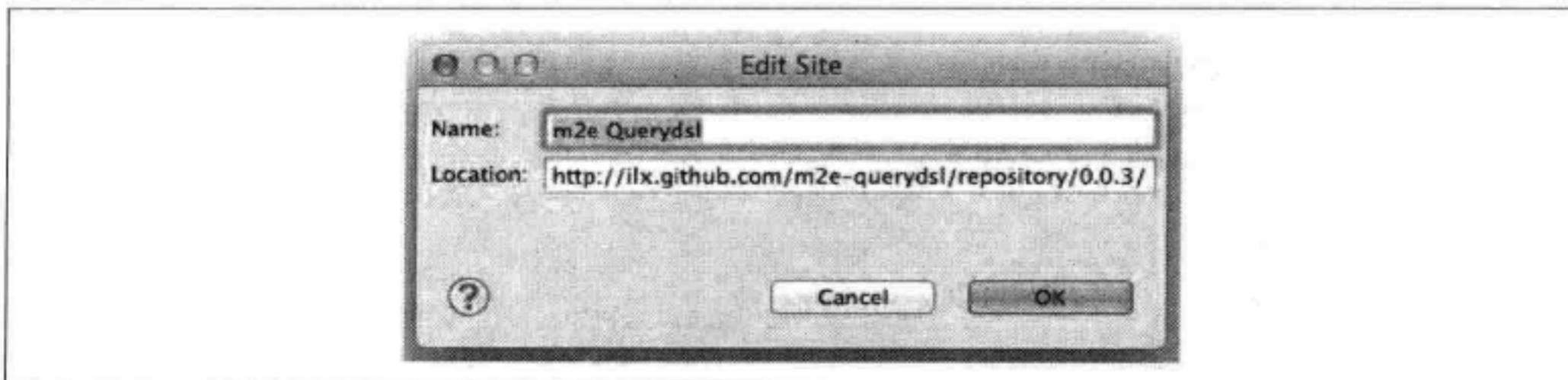


图 1-5 增加 m2e-querydsl 更新网站

## IntelliJ IDEA

IDEA 可以直接打开 Maven 项目而不需要其他额外的设置。选择菜单中的 Open Project 选项之后会弹出对话框（如图 1-6 所示）。

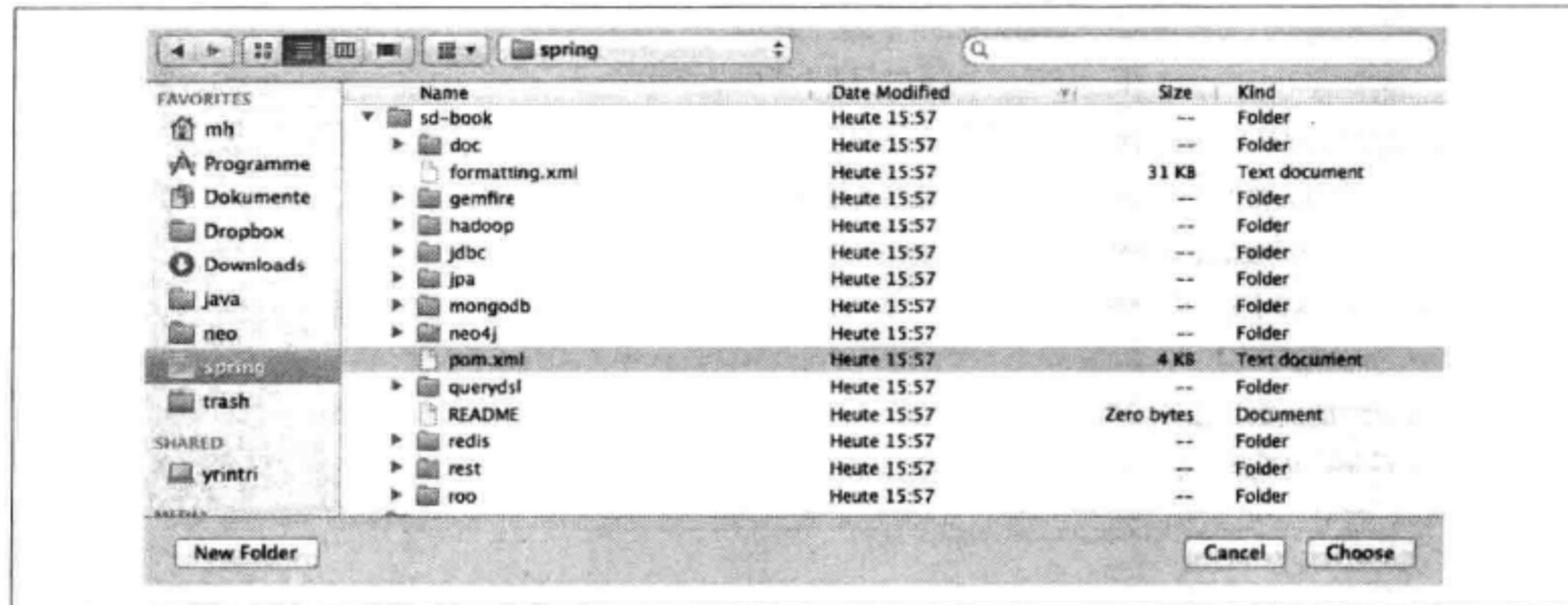


图 1-6 将 Maven 项目导入到 IDEA (步骤 1/2)

IDE 会打开项目并获取所需的依赖。在下一个步骤（如图 1-7 所示），它会探测已使用的框架（如 Spring 框架、JPA、WebApp 等）；可以使用弹出窗口的配置链接或者在事件日志中配置这些框架。

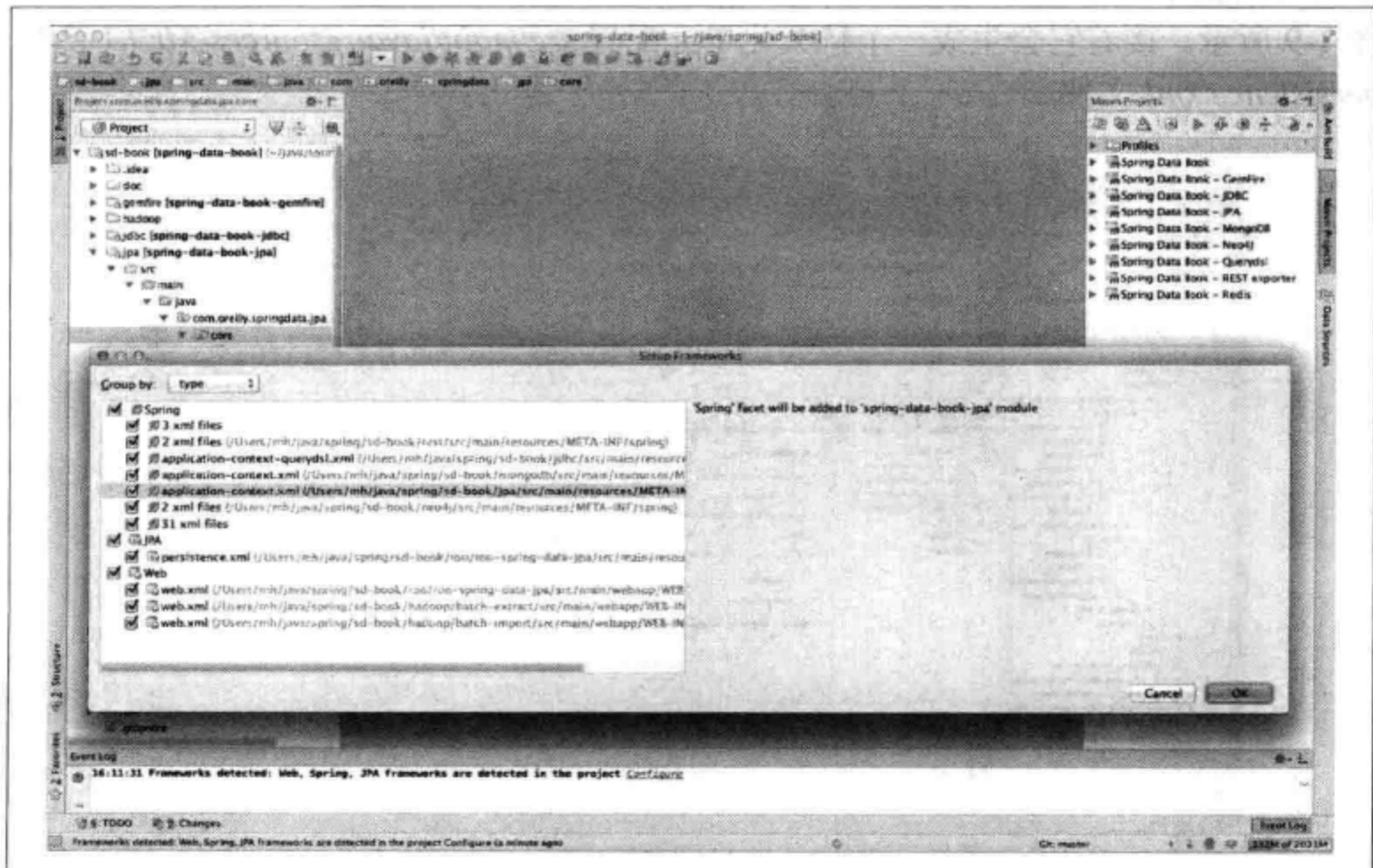


图 1-7 将 Maven 项目导入到 IDEA (步骤 2/2)

这样项目就可以使用了。此时可以看到“Project”视图和“Maven Project”视图，如图 1-8 所示。然后便可以像往常一样编译项目了。

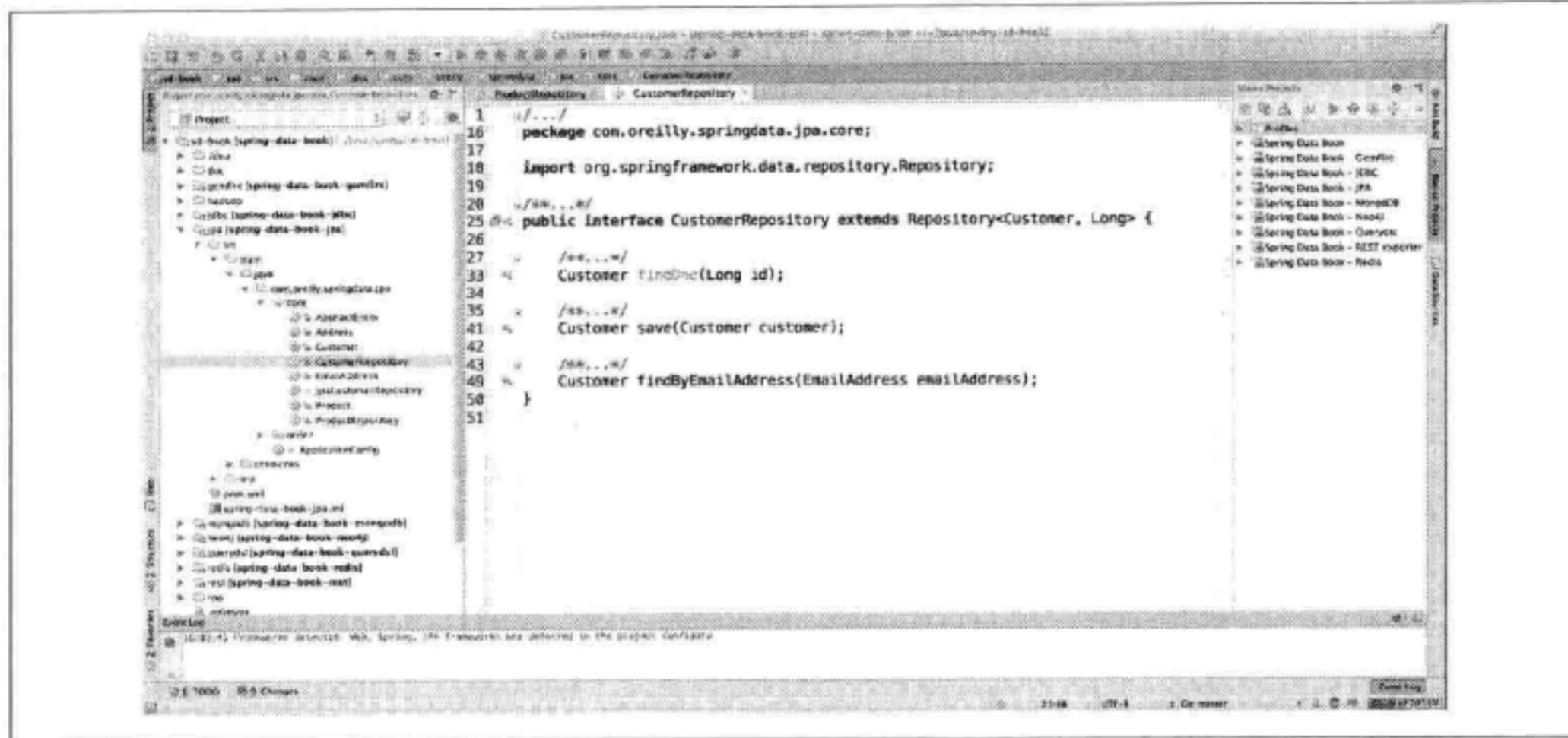


图 1-8 打开 Spring Data Book 项目的 IDEA

接下来，必须加入 Spring Data JPA 模块的 JPA 支持以启用 finder 方法以及版本库的错误检查功能。只需要右键单击该模块并选择“Add Framework Support”项，在弹出的对话框中勾选 JavaEE 持久化的支持并且选择 Hibernate 提供的持久化支持，如图 1-9 所示。接着它会生成一个持久化单元配置 *src/main/java/resources/META-INF/persistence.xml* 文件。

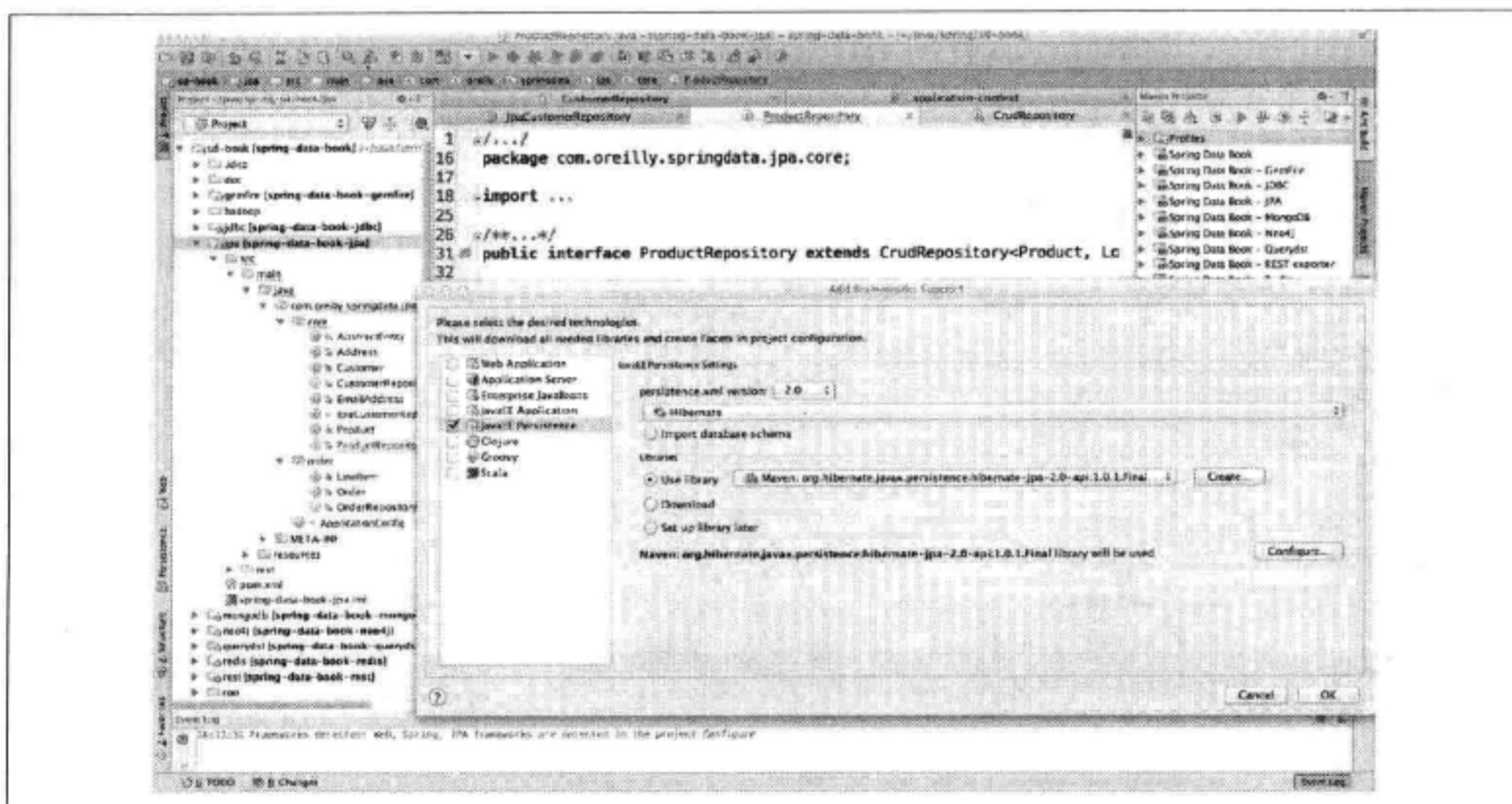


图 1-9 在 Spring Data JPA 模块启用 JPA 支持

# Repository：便利的数据访问层

长期以来，实现应用程序的数据访问层一直是件繁琐的工作，因为我们经常需要编写大量的样板式代码，而且贫血（anemic）的领域类并没有按照真正面向对象或领域驱动方式进行设计。因此 Spring Data Repository 抽象的目标就是大幅简化各种持久化存储持久层的实现。我们将会使用 Spring Data JPA 模块作为例子来讨论 Repository 抽象的基本理念。对于其他类型的存储，可以参考对应的例子。

## 2.1 快速入门

我们选取领域模型中的 Customer 领域类，它会被持久化到任意的存储之中。这个类应该如示例 2-1 所示。

示例 2-1 Customer 领域类

```
public class Customer {  
  
    private Long id;  
    private String firstname;  
    private String lastname;  
    private EmailAddress emailAddress;  
    private Address address;  
  
    ...  
}
```

传统的实现数据访问层的方式至少需要实现一个存储类（repository class），这个类会包含基本的 CRUD（Create、Read、Update 与 Delete）方法以及通过限制条件来访问实体子集的查询方法。Spring Data Repository 的方式能够避免大多数的代码，只需为这个实体存储声明简单的接口定义即可，如示例 2-2 所示。

## 示例 2-2 CustomerRepository 接口定义

```
public interface CustomerRepository extends Repository<Customer, Long> {  
    ...  
}
```

正如你所见，我们扩展了 Spring Data 的 Repository 接口，它是通用的标识接口。它的主要职责是让 Spring Data 的基础设施识别出所有用户定义的 Spring Data Repository。除此之外，它还会捕获托管的领域类以及实体的 ID 类型，稍后这些功能会提供很大的便利性。为了能够自动发现所声明的接口，可以使用存储特定的 XML 命名空间中的`<repositories />`元素（如示例 2-3 所示），或是在使用 JavaConfig 时借助相关的`@Enable...Repositories`注解（如示例 2-4 所示）。在示例中会使用 JPA。我们只需将 XML 元素的`base-package`属性配置为我们的根包（root package），Spring Data 会扫描它来查找 Repository 接口。如果没有给出更进一步的配置，那么它只会简单地检查包中带有注解的类。

## 示例 2-3 使用 XML 激活 Spring Data Repository

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

## 示例 2-4 使用 Java Config 激活 Spring Data Repository

```
@Configuration  
@EnableJpaRepositories  
class ApplicationConfig {  
  
}
```

XML 和 JavaConfig 配置都需要添加存储专用的 Bean 声明来进行完善，如 JPA 的 EntityManagerFactory 以及 DataSource 等。对于其他形式的存储，我们只需使用对应的命名空间元素或注解即可。例如，示例 2-5 所示的配置片段，将会找到 Spring Data Repository 并创建 Spring Bean，这些 Bean 实际上是由一组实现了所发现接口的代理所组成的。因此，现在可以继续编写客户端，通过 Spring 的自动装配就能访问这个 Bean 了。

CustomerRepository 接口建立之后，我们就可以继续深入学习并添加一些易于声明的查询方法。常见的需求是通过电子邮件地址来获取 Customer。为了做到这一点，我们添加合适的查询方法，如示例 2-6 所示。

## 示例 2-5 客户端使用 Spring Data Repository

```
@Component
public class MyRepositoryClient {

    private final CustomerRepository repository;

    @Autowired
    public MyRepositoryClient(CustomerRepository repository) {
        Assert.notNull(repository);
        this.repository = repository;
    }

    ...
}
```

## 示例 2-6 声明查询方法

```
public interface CustomerRepository extends Repository<Customer, Long> {

    Customer findByEmailAddress(Address email);
}
```

命名空间元素将会在容器启动的时候扫描到这个接口并触发 Spring Data 的基础设施为其创建 Spring Bean。基础设施会探查接口中声明的方法并确定方法调用时要执行的查询。如果只是这样简单地定义方法的话，那么 Spring Data 将会根据其名字衍生出一个查询。在定义查询方面还有其他的途径可选，可以阅读 2.2 小节“[定义查询方法](#)”来了解更多信息。

在示例 2-6 中，由于我们遵循了领域对象属性的命名约定，因而查询可以衍生得到。查询方法名中的 `EmailAddress` 部分其实就对应了 `Customer` 类的 `emailAddress` 属性，因此，在使用 JPA 模块时，Spring Data 会自动为声明的方法衍生出 `select C from Customer c where c.emailAddress = ?1`。它还会检查方法声明中属性引用的合法性，如果发现任何错误则会在容器启动时，出现启动失败。现在，客户端可以很容易地执行这个方法，给定的方法参数会绑定到根据方法名衍生出来的查询之中并且执行该查询，如示例 2-7 所示。

## 示例 2-7 执行查询方法

```
@Component
public class MyRepositoryClient {

    private final CustomerRepository repository;

    ...

    public void someBusinessMethod(Address email) {
        Customer customer = repository.findByEmailAddress(email);
    }
}
```

## 2.2 定义查询方法

### 2.2.1 查找查询的策略

刚才看到的接口只声明了一个简单的查询方法。声明的方法会被基础设施探测到并进行解析，最终衍生出与存储相关的查询。但是，随着查询变得更加复杂，方法名会变得很冗长，显得很笨拙。对于更复杂的查询，依靠方法解析器所支持的关键字就不够了。因此，每种存储模块都提供了@Query注解，如示例 2-8 所示，它会接受存储相关的查询语言所支持的查询字符串，从而允许查询执行时进一步地定制化。

#### 示例 2-8 使用@Query注解手动定义查询

```
public interface CustomerRepository extends Repository<Customer, Long> {  
    @Query("select c from Customer c where c.emailAddress = ?1")  
    Customer findByEmailAddress(Address email);  
}
```

在这里，我们使用 JPA 作为例子并手动定义了一个查询，当然这个查询原本也是可以通过衍生得到的。

查询甚至可以外部化配置到属性文件中（它位于 META-INF 目录下的\$store-named-queries.properties 文件中），在这里\$store 是用于替换 jpa、 mongo 以及 neo4j 等的占位符。key 值必须要遵循\$domainType.\$methodName 这样的约定。因此，为了将我们已有的方法替换成外部配置的命名查询，key 将会是 Customer.findByEmailAddress。如果是使用已命名查询的话，那就不需要使用@Query注解了。

### 2.2.2 衍生查询

如示例 2-9 所示，查询衍生机制内置于 Spring Data Repository 的基础设施之中，对基于 Repository 的实体来构建限制性的查询很有用处。我们会从方法中截取 findBy、readBy 以及 getBy 前缀并解析剩余的部分。一个基础的用法是，基于实体的属性来定义条件并使用 And 和 Or 将它们连接起来。

#### 示例 2-9 由方法名衍生查询

```
public interface CustomerRepository extends Repository<Customer, Long> {  
    List<Customer> findByEmailAndLastname(Address email, String lastname);  
}
```

解析的实际结果依赖于我们所使用的数据存储。这里也有一些需要注意的通用事项。表达式通常会是属性的遍历以及操作符，它们可以连接起来。如示例 2-9 所示的那样，可以通过 And 以及 Or 来连接属性表达式。除此之外，对于属性表达式来

说，还可以支持各种操作符，如 Between、LessThan、GreaterThan 以及 Like。因为不同的数据存储之间所支持的操作符有所区别，所以要看查阅每种存储对应的章节。

## 属性表达式

属性表达式可以直接引用所管理实体的属性（如示例 2-9 中所示）。在查询创建的时候，我们已经确保所解析的属性就是领域类的属性。但是，依然可以遍历嵌套的属性来定义限制条件。在前面我们可以看到，Customer 的 Address 中具有 ZipCode 属性。在这种情况下，如下这种方法名的查询将会创建 x.address.zipCode 这样的属性遍历。

```
List<Customer> findByAddressZipCode(ZipCode zipCode);
```

这个方案的算法首先会将整体（AddressZipCode）作为一个属性进行解析并检查领域类中是否具有该名称的属性（第一个字母小写）。如果它存在的话，就会使用该属性。如果不存在，它会从右边开始将源信息按照“驼峰”命名的规则将其拆分为头部和尾部，然后尝试查找对应的属性（如 AddressZip 和 Code）。如果按照这个头部信息找到了属性，那么我们将会使用尾部的信息继续往下构建树形的信息。因为示例中，第一次的分割并不匹配，我们将分割点继续左移（从“AddressZip、Code”移到“Address、ZipCode”）。

尽管这在大多数的场景下都是可行的，但是在一定情况下算法可能会选择错误的属性。假设 Customer 同时还有一个 addressZip 属性。那么我们的算法将会在第一次分割的时候就完成了匹配，这实际上选择了错误的属性，并且会导致最终的失败（因为 addressZip 类型可能并没有 code 属性）。为了解决这种模棱两可的问题，可以在方法名中使用下划线（\_）来手动定义遍历点。所以，我们的方法名最终看起来是这样的：

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

### 2.2.3 分页和排序

如果查询所返回的结果数量增长很明显，那么分块访问数据就很有意义了。为了做到这一点，Spring Data 提供了可与 Repository 一起使用的分页 API。要读取哪一块数据的定义隐藏在 Pageable 接口及其实现 PageRequest 之中。得到的分页数据存放在 Page 中，它不仅包含了数据本身，还包含了元信息，这些信息包括它是不是第一页或最后一页以及一共有多少页等。为了计算这个元数据，除了初始的查询外，我们需要触发第二次查询。

借助于 Repository，我们要使用分页功能时只需添加一个 Pageable 方法参数即可。不像其他的参数那样，这个参数是不与查询绑定的，用来限制返回的结果集。一种可选的方案就是返回 Page 类型，它会对结果集进行限制，但是需要另外一次查询

来获取元信息（如可用元素的总数）。另一种可选的方案是使用 `List`，它会避免额外的查询，但是不会提供元数据。如果不需要分页功能，只是想要排序，那么可以给方法签名上添加 `Sort` 参数，如示例 2-10 所示。

#### 示例 2-10 使用 Pageable 和 Sort 的查询方法

```
Page<Customer> findByLastname(String lastname, Pageable pageable);  
List<Customer> findByLastname(String lastname, Sort sort);  
List<Customer> findByLastname(String lastname, Pageable pageable);
```

第一个方法允许传递 `Pageable` 实例到查询方法中，从而为静态定义的查询动态地增加分页功能。排序的功能可以通过 `Sort` 参数显式地传递给方法，也可以内嵌到 `PageRequest` 值对象中，如示例 2-11 所示。

#### 示例 2-11 使用 Pageable 和 Sort

```
Pageable pageable = new PageRequest(2, 10, Direction.ASC, "lastname", "firstname");  
Page<Customer> result = findByLastname("Matthews", pageable);  
  
Sort sort = new Sort(Direction.DESC, "Matthews");  
List<Customer> result = findByLastname("Matthews", sort);
```

## 2.3 定义 Repository

到目前为止，我们看到了带有查询方法的 `Repository` 接口，这些查询有的是从方法名中衍生出来的，有的是手动声明的，这取决于 Spring Data 为实际存储类型所提供的使用方式。为了衍生出这些查询，我们必须扩展 Spring Data 的特定标识接口：`Repository`。除了查询以外，在你的 `Repository` 中还需要一些其他的功能：存储对象，删除对象，根据 ID 进行查找，返回所有存储的实体或按页对它们进行访问。通过 `Repository` 接口来暴露这些功能的最简单方式就是使用一个 Spring Data 所提供的更为高级的 `Repository` 接口。

### *Repository*

一个简单的标识接口，允许 Spring Data 的基础设施获取用户定义的 `Repository`。

### *CrudRepository*

扩展自 `Repository` 并添加了基本的持久化方法如对实体的保存、查找以及删除。

### *PagingAndSortingRepositories*

扩展自 `CrudRepository` 并添加了按页访问实体以及根据给定的条件 (`criteria`) 进行排序的方法。

假设我们想让 `CustomerRepository` 暴露基本的 CRUD 方法，所需要做就是修改其声

明，如示例 2-12 所示。

### 示例 2-12 暴露 CRUD 方法的 CustomerRepository

```
public interface CustomerRepository extends CrudRepository<Customer, Long> {  
    List<Customer> findByEmailAndLastname(Address email, String lastname);  
}
```

CrudRepository 接口如示例 2-13 所示。它包括了保存单个实体以及多个 Iterable 实体的方法、获取单个实体或所有实体的方法以及不同形式的 delete(...)方法。

### 示例 2-13 CrudRepository

```
public interface CrudRepository<T, ID extends Serializable> extends Repository<T, ID> {  
  
    <S extends T> S save(S entity);  
    <S extends T> Iterable<S> save(Iterable<S> entities);  
  
    T findOne(ID id);  
    Iterable<T> findAll();  
  
    void delete(ID id);  
    void delete(T entity);  
    void deleteAll();  
}
```

支持 Repository 方式的每个 Spring Data 模块都提供了这个接口的实现。因此，我们声明的命名空间元素会触发基础设施，这些设施不仅会启动那些用于执行查询方法的合适代码，同时还会使用一个通用 Repository 实现类的实例来在背后执行 CrudRepository 中所声明的方法，最终会将 save(...)、findAll() 等方法的调用委托给该实例。PagingAndSortingRepository（如示例 2-14 所示）扩展了 CrudRepository 并为通用的 findAll(...) 添加了处理 Pageable 和 Sort 实例的方法，从而能够实现逐页访问实体。

### 示例 2-14 PagingAndSortingRepository

```
public interface PagingAndSortingRepository<T, ID extends Serializable>  
    extends CrudRepository<T, ID> {  
  
    Iterable<T> findAll(Sort sort);  
  
    Page<T> findAll(Pageable pageable);  
}
```

要将这些功能引入到 CustomerRepository 中，只需简单地扩展 PagingAndSorting Repository 来取代 CrudRepository 即可。

## 2.3.1 调整 Repository 接口

正如我们在前面所见，通过扩展合适的 Spring Data 接口，可以很容易地引入大量预先定义的功能。这种级别的粒度实际上是一种权衡，那就是如果为所有的查找方

法、所有的保存方法等都定义单独的接口，我们会暴露接口的数量（以及因此导致的复杂性）以及开发人员使用的便利性之间的权衡。

但是，可能会有这样的场景，那就是只想暴露读方法（CRUD 中的 R）或者只想在 Repository 接口中将删除方法屏蔽掉。如今，Spring Data 允许定义个性化的基础 Repository，只需按照以下的步骤操作即可。

1. 创建一个接口，这个接口要么扩展自 Repository，要么添加@RepositoryDefinition 注解。
2. 添加想要暴露的方法并确保它们与 Spring Data 基础 Repository 接口所提供的方法签名相同。
3. 对于实体所对应的接口声明，要使用这个接口作为基础接口。

为了阐述这一点，假设我们只想暴露接收 Pageable 的 findAll(...) 方法以及 save 方法。这个基础接口看起来可能如示例 2-15 所示。

#### 示例 2-15 自定义基础 Repository 接口

```
@NoRepositoryBean
public interface BaseRepository<T, ID extends Serializable> extends Repository<T, ID> {
    Iterable<T> findAll(Pageable sort);
    <S extends T> S save(S entity);
    <S extends T> S save(Iterable<S> entities);
}
```

需要注意的一点是我们为这个接口添加了一个额外的注解 @NoRepositoryBean，从而确保 Spring Data Repository 的基础设施不会试图为其创建 Bean 的实例。让 CustomerRepository 扩展这个接口就能精确做到只暴露你所定义的 API。

接下来可以定义出各种基本的接口（如 ReadOnlyRepository 或 SaveOnlyRepository）甚至组成它们的继承体系，这取决于项目的需要。通常建议本地定义的 CRUD 方法在开始的时候直接位于每个实体的具体 Repository 中，必要的话，再将它们要么转移到 Spring Data 提供的基础 Repository 中，要么转移到特制的 Repository 中。按照这种方式，可以保证随着项目复杂性的增长，构件（artifact）的数量能够自然地增长。

### 2.3.2 手动实现 Repository 方法

到目前为止，看到了两种类型的 Repository 方法：CRUD 方法和查询方法。每种类型都是由 Spring Data 的基础设施实现的，要么通过背后的实现类，要么通过查询执行引擎。当构建应用程序的时候，这两种场景可能会覆盖你所面临的很大范围的数据访问操作。但是，有些场景需要手动实现代码。现在，让我们看一下如何做到

这一点。

我们开始只实现那些需要手动实现的功能并在实现类中遵循一些命名的约定，如示例 2-16 所示。

### 示例 2-16 为 Repository 实现自定义功能

```
interface CustomerRepositoryCustom {  
    Customer myCustomMethod(...);  
}  
  
class CustomerRepositoryImpl implements CustomerRepositoryCustom {  
    // Potentially wire dependencies  
  
    public Customer myCustomMethod(...) {  
        // custom implementation code goes here  
    }  
}
```

接口和实现类均不需要了解 Spring Data 的任何事情。它与使用 Spring 手动实现代码非常类似。按照 Spring Data 来看，这个代码片段最有意思的地方在于实现类的名字遵循了命名的约定，也就是在核心 Repository 接口（在我们的场景中就是 CustomerRepository）的名字上加 Impl 后缀。同时需要注意，我们将接口和实现类都设为包内私有（package private），从而阻止从包外访问它们。

最后一步是修改初始 Repository 接口的声明，使其扩展刚刚引入的接口，如示例 2-17 所示。

### 示例 2-17 在 CustomerRepository 中包含自定义功能

```
public interface CustomerRepository extends CrudRepository<Customer, Long>,  
    CustomerRepositoryCustom { ... }
```

现在，我们已经将 CustomerRepositoryCustom 暴露的 API 引入到 CustomerRepository 之中了，这会使其成为 Customer 数据访问 API 的中心点。客户端代码现在就可以调用 CustomerRepository.myCustomMethod(...) 了。但是，这个实现类会如何被发现并置于最终执行的代理之中的呢？实际上，Repository 的启动过程看起来是这样的。

1. 发现 repository 接口（如 CustomerRepository）。
2. 尝试寻找一个 Bean 定义，这个 Bean 的名字为接口的小写形式并添加 Impl 后缀（如 customerRepositoryImpl）。如果能够找到，就使用它。
3. 如果没有找到，我们会扫描寻找一个类，这个类的名字为核心 Repository 接口的名字并添加 Impl 后缀（例如，在这个例子中 CustomerRepositoryImpl 会被找到）。如果找到了这样的类，那么将其注册为 Spring Bean 并使用它。
4. 找到的自定义实现类将会装配到被发现接口的代理配置之中并且在方法调用时

会作为潜在的目标类。

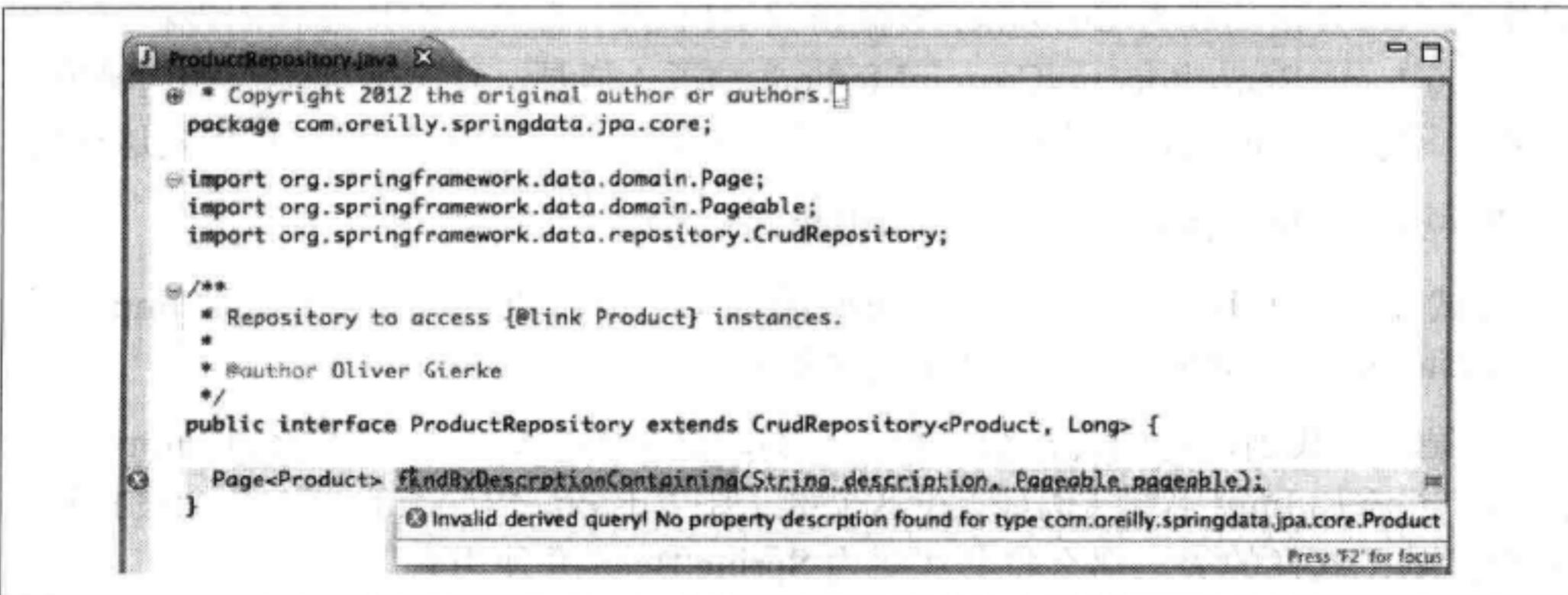
这种机制可以很容易地为特定 Repository 实现自定义代码。用于进行实现查找的后缀可以在 XML 命名空间中或启用 Repository 的注解属性中（查看各种存储相关的章节来了解更多）进行个性化设置。参考文档 (<https://bit.ly/VzYToo>) 中也包含了一些关于如何将自定义的行为应用于多个 Repository 的学习材料。

## 2.4 IDE 集成

在 3.0 版本中，Spring 工具套件 (Spring Tool Suite, STS) 提供了与 Spring Data Repository 抽象进行集成的功能。STS 为 Spring Data 所提供的核心支持是查找方法的查询衍生机制。它所能做到的第一件事就是在 IDE 中校验衍生查询方法的正确性，这样，不需要启动 ApplicationContext 就能立刻探测出方法名中引入的拼写错误。

STS 是一个特殊的 Eclipse 发布版本，它内置了一些插件从而尽可能地便于进行 Spring 应用的构建。这个工具可以在项目的站点上 (<http://www.springsource.org/sts>) 下载或者使用一般的 Eclipse 发布版本并通过 STS 更新站点进行更新（基于 Eclipse 3.8 (<http://dist.springsource.org/release/TOOLS/update/e3.8>) 或 Eclipse 4.2 (<http://dist.springsource.org/release/TOOLS/update/e4.2>))。

如图 2.1 所示，IDE 检测到 Description 是非法的，因为 Product 类中并没有这样的属性。为了发现这些拼写错误，它会分析 Product 领域类（这些事情在启动 Spring Data Repository 时也会做）来获取属性并将方法名解析为属性的遍历树。为了尽早避免这种类型的拼写错误，STS 的 Spring Data 辅助功能为属性名、条件关键字 (criteria keyword) 以及像 And 和 Or 这样的连接符提供了代码补全功能，如图 2-2 所示。



The screenshot shows the STS IDE with a Java code editor open for the file `ProductRepository.java`. The code defines a repository interface extending `CrudRepository<Product, Long>`. A method is being typed:

```
Page<Product> findByDescriptionContaining(String description, Pageable pageable);
```

A tooltip appears over the word `description`, indicating it is an invalid derived query because no property `description` was found for the type `com.oreilly.springdata.jpa.core.Product`.

图 2-1 Spring Data STS 对衍生查询方法名进行校验

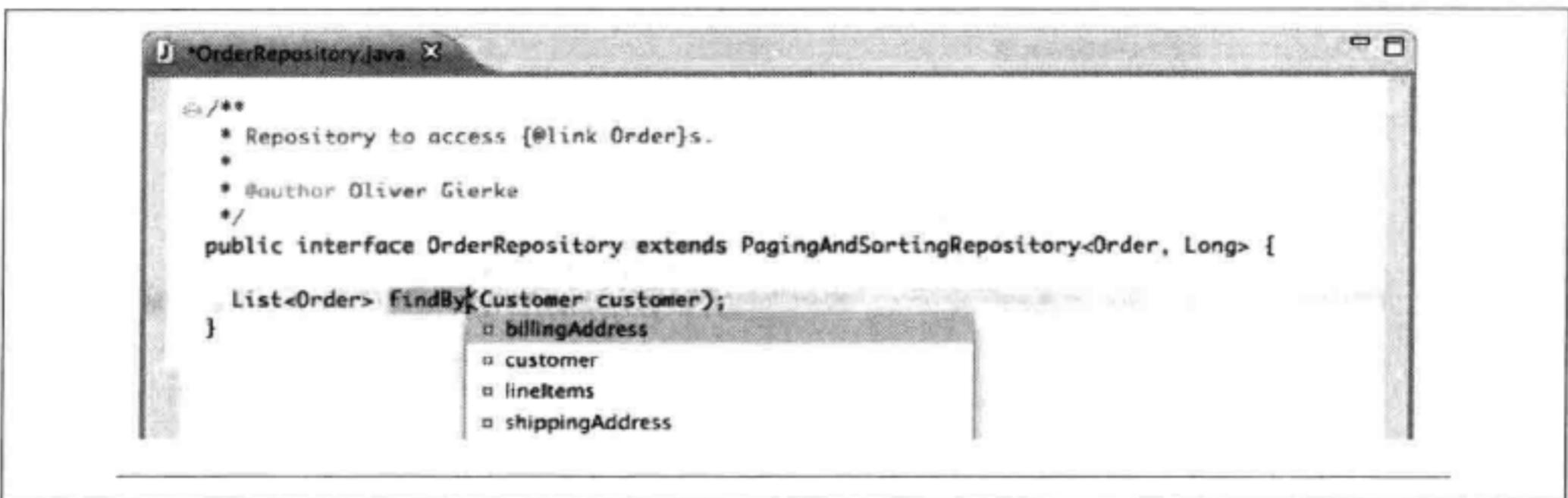


图 2-2 对衍生查询方法的属性代码补全提示

Order 类中有一些你可能想要引用的属性。假设我们要遍历 billingAddress 属性，Cmd+Space（或者在 Windows 中使用 Ctrl+Space 组合键）将会触发嵌套属性的遍历，这样将会提示出嵌套的属性并根据此时所遍历的属性类型提示所匹配的关键字（如图 2-3 所示）。因此，String 类型的属性将会多一个 Like 的提示。

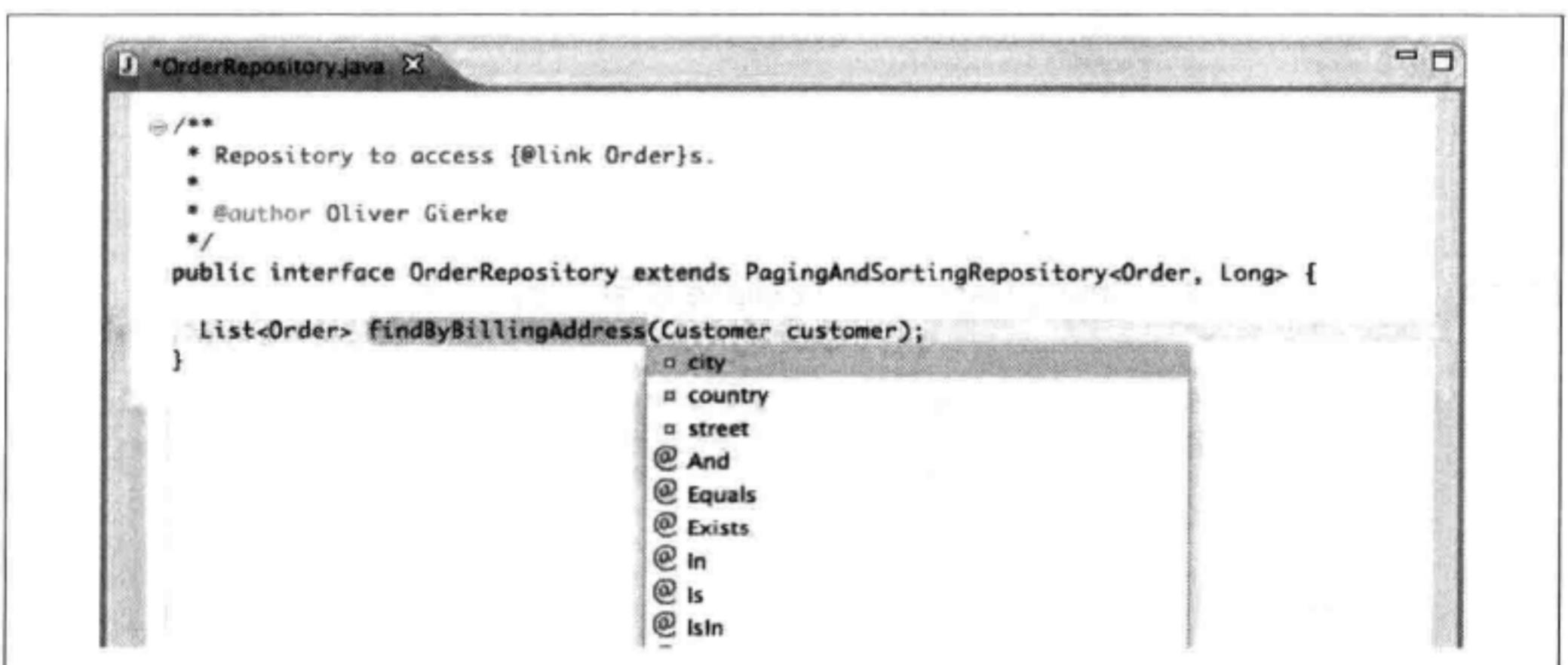


图 2-3 嵌套属性和关键字提示

为了提供一些锦上添花的特性，Spring Data STS 会将 Repository 作为 IDE 导航中的一等公民，使其带有众所周知的 Spring Bean 标识。除此之外，导航中的 Spring 元素（Spring Elements）节点将会包含一个专有的 Spring Data Repositories 节点，用来放置应用程序中所配置的所有 Repository，如图 2-4 所示。

可以看到，你能够快速找到 Repository 接口并跟踪它实际上来源于哪一个配置元素。

## 2.4.1 IntelliJ IDEA

最后，启用 JPA 支持后，IDEA 提供了 Repository 查找方法的补全功能，这种补全涵盖了衍生的属性名以及可用的关键字，如图 2-5 所示。

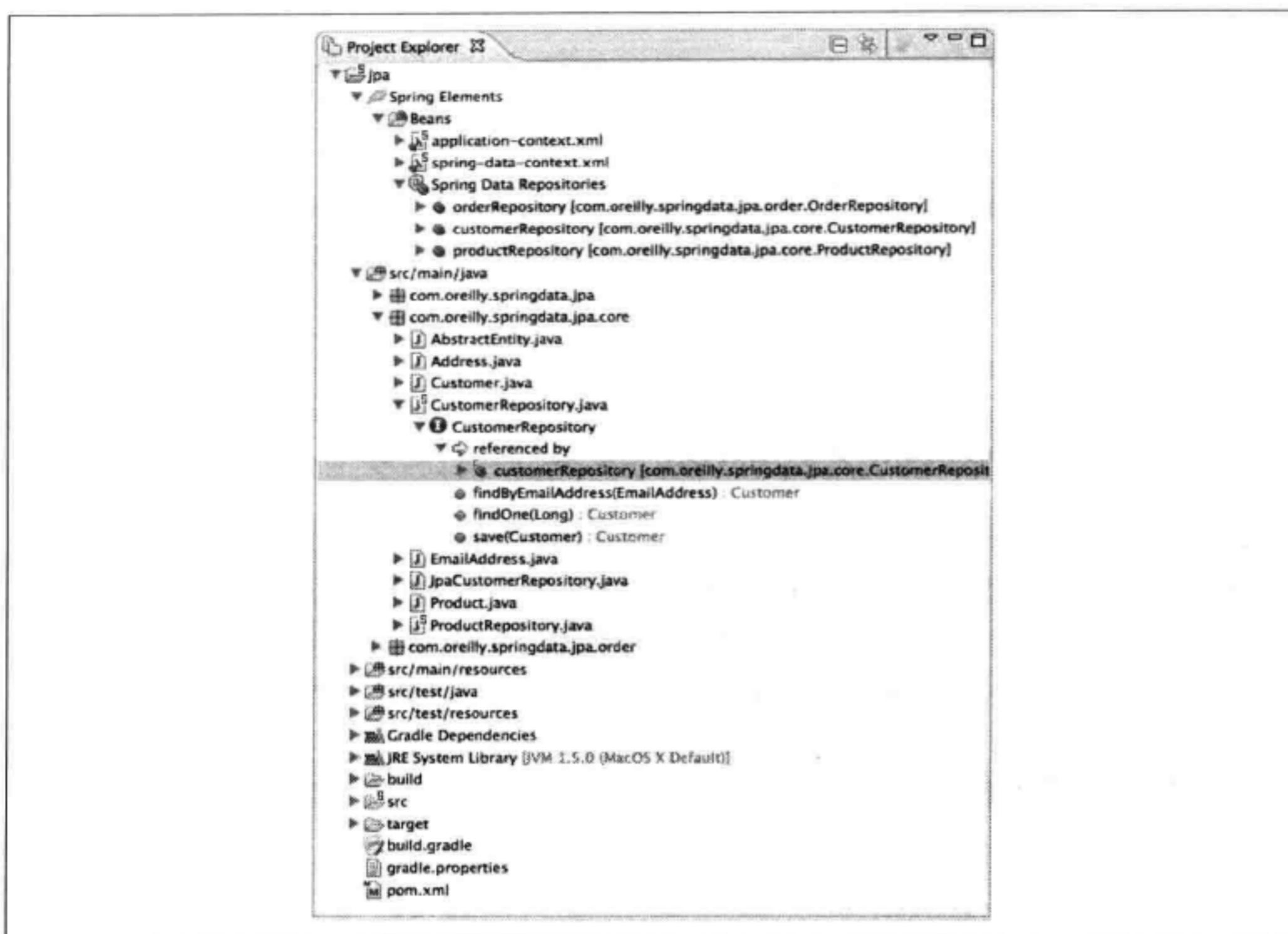


图 2-4 在 STS 中，具备 Spring Data 支持的 Eclipse 项目资源管理器

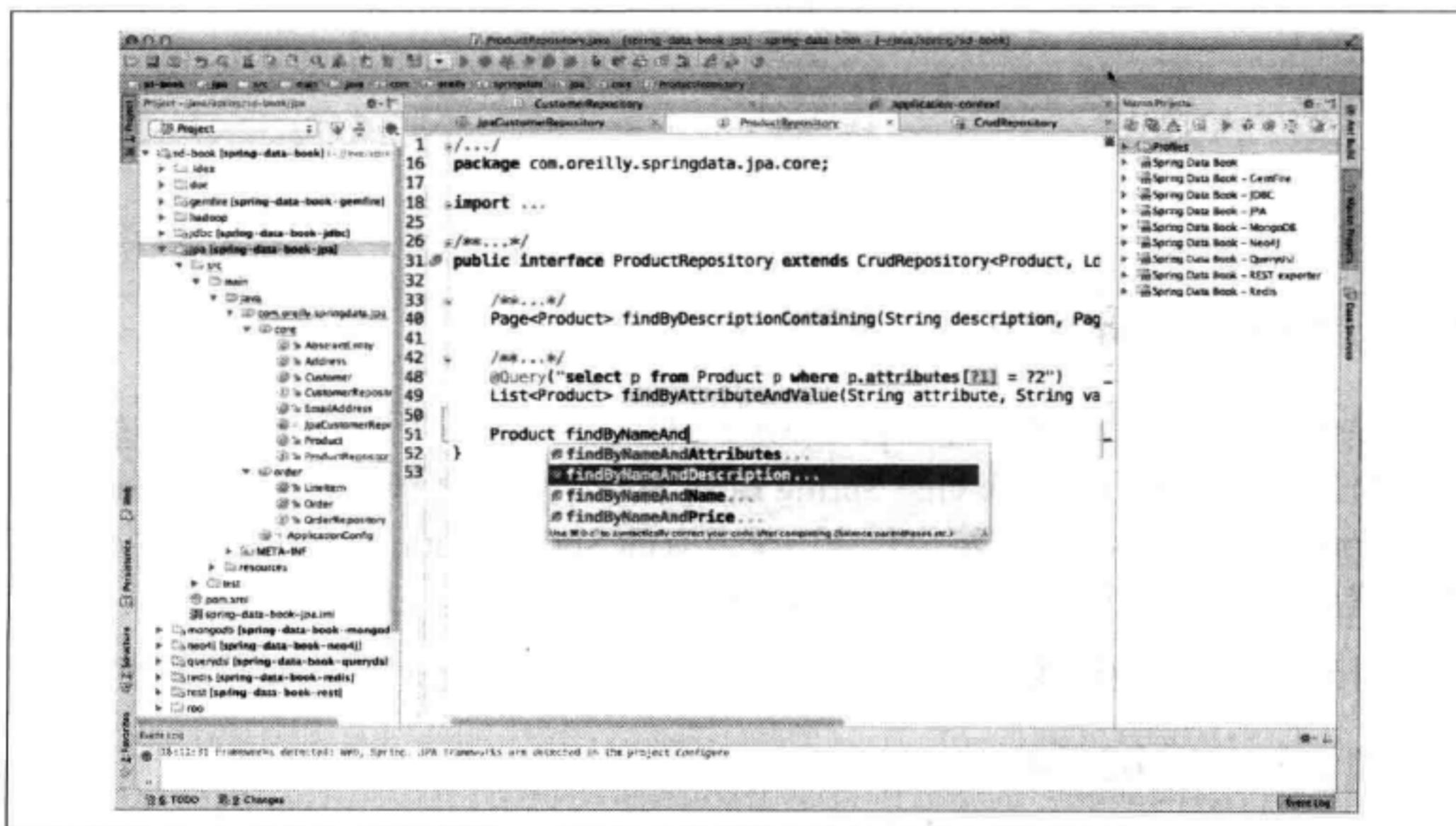


图 2-5 在 IDEA 编辑器中，查询方法的补全功能

# 使用 Querydsl 实现类型安全的查询

编写访问数据的查询通常会使用 Java 的字符串（String）来完成。对于 JDBC 来说，可供选择的查询语言就是 SQL，而对于 Hibernate/JPA 来说就是 HQL/JPQL。使用简单的字符串来定义查询的功能是很强大的，但也易于出错，因为很容易引入拼写错误。除此之外，它与实际的查询源或底层存储很少有关联，所以列引用（在 JDBC 的场景下）或属性引用（在 HQL/JPQL 上下文中）在维护方面会成为一种负担，这是因为表或对象模型的变化并不能很容易地映射到查询之中。

Querydsl 项目 (<http://www.querydsl.com>) 试图解决这样的问题，它提供了一个非常流畅的 API 来定义查询。这个 API 衍生于实际的表或对象模型，但同时又是与存储和模型高度无关的，所以它允许为各种存储类型创建和使用查询 API。它目前支持 JPA、Hibernate、JDO、原生的 JDBC、Lucene、Hibernate Search 以及 MongoDB。功能的多样性是 Spring Data 集成 Querydsl 的主要原因，因为 Spring Data 也集成了多种类型的存储。下面将会介绍 Querydsl 项目以及它的基本理念。在本书后面各种存储类型相关的章节中，我们还会介绍它对这些存储的支持。

## 3.1 Querydsl 简介

当使用 Querydsl 的时候，通常开始会从领域类中衍生出元模型。尽管这个类库也可以使用简单的字符串文本，但创建元模型会释放出 Querydsl 的全部能量，尤其是在属性的类型安全以及关键字引用方面。衍生机制基于 Java 6 的注解处理工具（Annotation Processing Tool, APT），它能够挂接到编译器中并对源码甚至编译后的类进行处理。想了解更多信息的话，可参阅 3.2 小节“生成查询模型”。开始之前，需要定义一个领域类，如示例 3-1 所示。我们使用几个原始类型和非原始类型的属

性来为 Customer 建模。

### 示例 3-1 Customer 领域类

```
@QueryEntity
public class Customer extends AbstractEntity {

    private String firstname, lastname;
    private EmailAddress emailAddress;
    private Set<Address> addresses = new HashSet<Address>();

    ...
}
```

注意我们为这个类使用了@QueryEntity 注解。这是默认的注解，Querydsl 注解处理器将会使用它来生成相关的查询对象。在与特定的存储集成使用的时候，APT 处理器能够识别出特定存储的实体（如对于 JPA 所使用的就是@Entity）并使用它们来衍生查询类。在这个简介部分，我们不会与存储一起工作，所以无法使用特定存储的映射注解，只是使用@QueryEntity 而已。生成的 Querydsl 查询类如示例 3-2 所示。

### 示例 3-2 Querydsl 生成的查询类

```
@Generated("com.mysema.query.codegen.EntitySerializer")
public class QCustomer extends EntityPathBase<Customer> {

    public static final QCustomer customer = new QCustomer("customer");
    public final QAbstractEntity _super = new QAbstractEntity(this);

    public final NumberPath<Long> id = _super.id;
    public final StringPath firstname = createString("firstname");
    public final StringPath lastname = createString("lastname");
    public final QEmailAddress emailAddress;

    public final SetPath<Address, QAddress> addresses =
        this.<Address, QAddress>createSet("addresses", Address.class, QAddress.class);

    ...
}
```

可以在该模块示例工程的 *target/generated-sources/queries* 目录下找到这些类。该类暴露了公开的 Path 属性以及对其他查询类的引用（如 QEmailAddress）。在定义断言（predicate）的时候，IDE 就能在代码补全时列出所有可用的 Path。可以使用这些 Path 表达式来定义可重用的断言，如示例 3-3 所示。

### 示例 3-3 使用查询类来定义断言

```
QCustomer customer = QCustomer.customer;

BooleanExpression idIsNull = customer.id.isNull();
BooleanExpression lastnameContainsFragment = customer.lastname.contains("thews");
BooleanExpression firstnameLikeCart = customer.firstname.like("Cart");
EmailAddress reference = new EmailAddress("dave@dmbar.com");
BooleanExpression isDavesEmail = customer.emailAddress.eq(reference);
```

将静态的 `QCustomer.customer` 实例赋值给 `customer` 变量，这样就能非常简洁地引用它的 `Path` 属性。可以看到，断言的定义非常简单整洁，而且——最重要的就是——类型安全。变更领域类将会重新生成查询模型。变更所导致的属性引用失效将会产生编译错误，从而提示我们要进行修改的地方。对于每种 `Path` 类型所能使用的方法也会考虑到 `Path` 的类型（如 `like(...)` 方法就只能对 `String` 属性有意义，因此只会提供给这种类型使用）。

因为断言的定义非常简洁，所以它们能够很容易地用在方法声明的内部。另一方面，我们可以很容易地以可重用的方式来定义断言，定义原子性的断言并通过像 `And` 或 `Or` 这样的操作符将其连接起来以形成复杂的断言（如示例 3-4 所示）。

#### 示例 3-4 连接原子性的断言

```
QCustomer customer = QCustomer.customer;

BooleanExpression idIsNull = customer.id.isNull();

EmailAddress reference = new EmailAddress("dave@dmbyband.com");
BooleanExpression isDavesEmail = customer.emailAddress.eq(reference);

BooleanExpression idIsNullOrIsDavesEmail = idIsNull.or(isDavesEmail);
```

我们可以使用刚刚编写的断言来为特定的存储或者简单的集合来编写查询。鉴于对特定存储的查询执行主要会通过 `Spring Data Repository` 抽象来实现（见 3.3 小节“与 `Spring Data Repository` 集成”），所以作为例子，为了简单起见，我们会使用这个特性来对集合进行查询。首先，构建各种类型的 `Product`，这样我们就有东西可以过滤了，如示例 3-5 所示。

#### 示例 3-5 构建 Product

```
Product macBook = new Product("MacBook Pro", "Apple laptop");
Product iPad = new Product("iPad", "Apple tablet");
Product iPod = new Product("iPod", "Apple MP3 player");
Product turntable = new Product("Turntable", "Vinyl player");

List<Product> products = Arrays.asList(macBook, iPad, iPod, turntable);
```

接下来，我们可以使用 `Querydsl` 来对这个集合建立查询，也就是对其建立某种类型的过滤器（如示例 3-6 所示）。

#### 示例 3-6 使用 Querydsl 断言过滤 Product

```
QProduct $ = QProduct.product;
List<Product> result = from($, products).where($.description.contains("Apple")).list($);
assertThat(result, hasSize(3));
assertThat(result, hasItems(macBook, iPad, iPod));
```

我们使用 `from(...)` 方法来建立 `Querydsl` 查询，它是 `querydsl-collections` 模块之中 `MiniAPI` 类的一个静态方法。我们将 `Product` 的查询类实例以及作为源的集

合传递给它。现在，我们就可以使用 `where(...)` 方法将断言应用于源列表之上并使用某一个 `list(...)` 方法（如示例 3-7 所示）执行查询。在我们的场景下，只是简单地返回那些匹配预先定义断言的 `Product`。将 `$.description` 传递给 `list(...)` 方法后，我们就能将结果投射（`project`）到 `Product` 的名称上，因此返回一个 `String` 的集合。

### 示例 3-7 使用 Querydsl 断言过滤 `Product` ( 投射 )

```
QProduct $ = QProduct.product;
BooleanExpression descriptionContainsApple = $.description.contains("Apple");
List<String> result = from($, products).where(descriptionContainsApple).list($.name);

assertThat(result, hasSize(3));
assertThat(result, hasItems("MacBook Pro", "iPad", "iPod"));
```

借助于 `Querydsl`，我们能够以一种简洁和便利的方式来定义实体断言。它们可以基于各种存储以及 Java 类的映射信息而产生。`Querydsl` 的 API 以及它所支持的存储方式使得我们可以产生断言，并以此来定义查询。相同的 API 也可以用于对 Java 集合进行过滤。

## 3.2 生成查询元模型

正如在前文中所见，`Querydsl` 的核心构件（artifact）就是查询元模型类。这些类是通过注解处理工具集（APT，Annotation Processing Toolkit，见 <http://www.jcp.org/en/jsr/detail?id=175>）生成的，它是 Java 6 中 `javac` Java 编译器的一部分。APT 有一种机制，能够以编码的方式探查已有的 Java 源代码以查找特定的注解，然后再回过头来调用函数以生成 Java 代码。`Querydsl` 使用了这种机制来提供特定的 APT 处理器实现类，这个类可以用于探查注解。示例 3-1 使用了 `Querydsl` 特定的注解，如 `@QueryEntity` 以及 `@QueryEmbeddable`。如果我们已经拥有了某种存储类型的映射领域类，而这种存储类型是 `Querydsl` 所能够支持的，那么生成元模型类并不需要额外的过程。在这里的核心集成点在于需要将注解处理器传递给 `Querydsl` APT。通常处理器会在构建过程中执行。

### 3.2.1 构建系统集成

为了与 Maven 集成，`Querydsl` 提供了 `maven-apt-plugin` 插件，借助它就能够配置实际使用的处理器类了。在示例 3-8 中，将 `process` 的目标设置为 `generate-source` 阶段，这样所配置的处理器类就能探查 `src/main/java` 目录下的类。如果想要为测试源码包（`src/test/java`）中的类生成元模型类的话，需要在 `generate-test-sources` 阶段执行 `test-process` 目标。

### 示例 3-8 设置 Maven APT 插件

```
<project ...>
  <build>
    <plugins>
      <plugin>
        <groupId>com.mysema.maven</groupId>
        <artifactId>maven-apt-plugin</artifactId>
        <version>1.0.2</version>
        <executions>
          <execution>
            <goals>
              <goal>process</goal>
            </goals>
            <phase>generate-sources</phase>
            <configuration>
              <outputDirectory>target/generated-sources/java</outputDirectory>
              <processor><!-- fully-qualified processor class name --></processor>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

## 3.2.2 所支持的注解处理器

Querydsl 自带了各种 APT 处理器，以用于探查不同的注解并生成对应的元模型类。

### QuerydslAnnotationProcessor

非常核心的注解处理器，会探查 Querydsl 的特定注解，如**@QueryEntity** 和 **@QueryEmbeddable**。

### JPAAnnotationProcessor

用于探查 javax.persistence 注解，如**@Entity** 以及**@Embeddable**。

### HibernateAnnotationProcessor

类似于 JPA 处理器，但是增加了对 Hibernate 注解的支持。

### JDOAnnotationProcessor

探查 JDO 注解，如**@PersistenceCapable** 和**@EmbeddedOnly**。

### MongoAnnotationProcessor

Spring Data 的一个专用的处理器，会探查**@Document** 注解。阅读 6.3 小节“映射模块”以了解更多信息。

### 3.2.3 使用 Querydsl 对存储进行查询

现在查询类已经就绪了，让我们看一下如何实际使用它们构建与特定存储相关的查询。正如前面所提到的，Querydsl 为各种存储都提供了集成模块，这些模块以优雅且一致的 API 来创建查询对象、通过所生成的元模型类进行断言定义并执行最终的查询。

例如，JPA 模块提供了一个 JPAQuery 实现类，它会接受 EntityManager 并提供了 API 能够实现在执行之前应用断言，如示例 3-9 所示。

#### 示例 3-9 使用 Querydsl JPA 模块来查询关系型存储

```
EntityManager entityManager = ... // obtain EntityManager
JPAQuery query = new JPAQuery(entityManager);

QProduct $ = QProduct.product;
List<Product> result = query.from($).where($.description.contains("Apple")).list($);

assertThat(result, hasSize(3));
assertThat(result, hasItems(macBook, iPad, iPod));
```

如果还记得示例 3-6 的话，就会发现这两个代码片段看上去区别并不大。实际上，唯一的区别在于我们采用 JPAQuery 作为基础，而前面的示例使用的是集合包装类（Collection Wrapper）。为 MongoDB 存储实现相同的场景同样也没有太大的变化，对于这一点你应该不会感到太惊讶（如示例 3-10 所示）。

#### 示例 3-10 联合使用 Querydsl MongoDB 模块以及 Spring Data MongoDB

```
MongoOperations operations = ... // obtain MongoOperations
MongodbQuery query = new SpringDataMongodbQuery(operations, Product.class);

QProduct $ = QProduct.product;
List<Product> result = query.where($.description.contains("Apple")).list();

assertThat(result, hasSize(3));
assertThat(result, hasItems(macBook, iPad, iPod));
```

## 3.3 集成 Spring Data Repository

如你所见，使用 Querydsl 所执行的查询一般来说包括 3 个主要的步骤。

1. 构建存储相关的查询实例。
2. 在查询上使用一些过滤断言。
3. 执行查询实例，可能会对其使用投射。

其中的两个步骤可以视为样板式的，因为它们通常会编写类似的代码。另一方面，Spring Data Repository 会尽可能帮助用户减少不必要的代码的数量，因此将 Repository

抽象与 Querydsl 集成在一起是很有意义的。

### 3.3.1 执行断言

集成的核心在于 `QueryDslPredicateExecutor` 接口，它指定了用户可以执行 Querydsl 断言的 API，类似于 `CrudRepository` 所提供的 CRUD 方法，如示例 3-11 所示。

#### 示例 3-11 `QueryDslPredicateExecutor` 接口

```
public interface QueryDslPredicateExecutor<T> {  
    T findOne(Predicate predicate);  
  
    Iterable<T> findAll(Predicate predicate);  
    Iterable<T> findAll(Predicate predicate, OrderSpecifier<?>... orders);  
  
    Page<T> findAll(Predicate predicate, Pageable pageable);  
    long count(Predicate predicate);  
}
```

目前，Spring Data JPA 和 MongoDB 模块通过提供实现类来实现示例 3-11 中的 `QueryDslPredicateExecutor` 接口以支持这个 API。为了通过 `Repository` 接口暴露这个 API，需要让其扩展 `QueryDslPredicateExecutor` 以及 `Repository`（或其他可用的基础接口），参见示例 3-12。

#### 示例 3-12 `CustomerRepository` 接口扩展了 `QueryDslPredicateExecutor`

```
public interface CustomerRepository extends Repository<Customer, Long>,  
                                         QueryDslPredicateExecutor<Customer> {  
    ...  
}
```

扩展这个接口会有两个重要的结果：首先（可能也是最重要的）就是它将 API 引入了进来并且将其暴露给 `CustomerRepository` 的客户端。其次，Spring Data Repository 基础设施将会探查每个 `Repository` 接口并判断它是否扩展了 `QueryDslPredicateExecutor`。如果答案是肯定的并且 `Querydsl` 位于类路径之中，那么 Spring Data 将会选择一个特定的基类支撑 `Repository` 代理，它一般会通过创建存储相关的查询实例、绑定给定的断言、使用分页并最终执行查询来实现这些 API 方法。

### 3.3.2 手动实现 `Repository`

我们刚刚所看到的方式解决了为 `Repository` 所管理的领域类进行通用查询的问题。但是，无法通过这种机制执行更新或删除操作，或者管理特定存储的查询实例。这种场景可以很好地参与 `Repository` 抽象的特性，可以有选择地实现那些需要手动代码的方法（参见 2.3.2 小节“手动实现 `Repository` 方法”了解这个话题的更多细节）。为了便于实现自定义的 `Repository` 扩展，我们提供了特定存储的基础类。要了解更多细节，参见 4.4.2 小节“`Repository` 与 `Querydsl` 集成”以及 6.5.3 小节“`Mongo Querydsl` 集成”。



第二部分

---

## 关系型数据库



# JPA Repository

Java 持久化 API (JPA, Java Persistence API) 是将 Java 对象持久化到关系型数据库的标准方式。JPA 包含了两个部分：用于将类匹配到关系型表的映射模块以及访问对象、定义和执行查询等功能的 EntityManager API。JPA 抽象了多种实现，如 Hibernate (<http://www.hibernate.org>)、EclipseLink (<http://www.eclipse.org/eclipselink>)、OpenJpa (<http://openjpa.apache.org>) 等。Spring 框架一直以来就为简化 JPA 的存储实现提供了良好的支持。它所提供的支持包括构建 EntityManager 的辅助类、集成 Spring 的事务抽象并将 JPA 的特定异常转换为 Spring 的 DataAccessException 异常体系。

Spring Data JPA 模块实现了 Spring Data 通用的 Repository 抽象从而进一步简化了存储的实现，这样在大多数场景无需手动实现存储类。要了解 Repository 抽象的简介信息，可以参考第 2 章。本章将会带你了解这个模块的通用构建过程和特性。

## 4.1 示例工程

本章的示例工程包含了 3 个包：`com.oreilly.springdata.jpa` 这个基础包以及 `core` 和 `order` 两个子包。基础包里面包含了一个 Spring 的 `JavaConfig` 类，它可以通过简单的 Java 类而不是 XML 来配置 Spring 容器。其他的两个包中包含了我们的领域类以及 Repository 接口。正如其名字所示，`core` 包中包含了对领域模型的基础抽象：像 `AbstractEntity` 这种技术化的帮助类，同时也包含了像 `EmailAddress`、`Address`、`Customer` 以及 `Product` 这样的领域概念。然后，还有 `order` 包，基于这些基础的概念实现了订单的概念。所以在这里可以看到 `Order` 以及 `OrderItem`。在后面的段落中，我们会仔细地查看每个类，介绍其目的以及使用 JPA 映射注解将它们匹配到数据库的方式。

我们领域模型中所有实体的基础类就是 `AbstractEntity`（如示例 4-1 所示）。它使用了`@MappedSuperclass` 来表示它本身并不是受管理的实体类，而是会由其他的实体类进行扩展。我们在这里声明了一个 `Long` 类型的 `id` 并且告知持久化提供商自动选择最合适策略来生成主键。除此之外，我们通过检查 `id` 属性实现了 `equals(...)` 和 `hashCode(...)` 方法，这样一来，具有相同 `id` 的同种类型的实体会被视为相等。在这个类中，包含了持久化实体的主要技术化信息，因此在具体的实体类中就可以关注实际的领域属性。

### 示例 4-1 `AbstractEntity` 类

```
@MappedSuperclass
public class AbstractEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Override
    public boolean equals(Object obj) { ... }

    @Override
    public int hashCode() { ... }
}
```

让我们继续看一下非常简单的 `Address` 领域类。如示例 4-2 所示，它是一个带有 `@Entity` 注解的普通类，并且包含了 3 个 `String` 属性。因为它们都是基础属性，因此不需要添加额外的注解，持久化提供商会自动将它们匹配到表的列上。如果需要自定义属性要持久化的列名，可以使用`@Column` 注解。

### 示例 4-2 `Address` 领域类

```
@Entity
public class Address extends AbstractEntity {

    private String street, city, country;
}
```

`Address` 会被 `Customer` 实体所引用。`Customer` 会包含一些其他的属性（如原始类型的 `firstname` 和 `lastname`）。它们在映射上与我们看到的 `Address` 类似。每个 `Customer` 还会有一个电子邮件地址，这会通过 `EmailAddress` 类体现（如示例 4-3 所示）。

### 示例 4-3 `EmailAddress` 领域类

```
@Embeddable
public class EmailAddress {
    private static final String EMAIL_REGEX = "...";
    private static final Pattern PATTERN = Pattern.compile(EMAIL_REGEX);

    @Column(name = "email")
    private String emailAddress;

    public EmailAddress(String emailAddress) {
        Assert.isTrue(isValid(emailAddress), "Invalid email address!");
    }
}
```

```

    this.emailAddress = emailAddress;
}

protected EmailAddress() { }

public boolean isValid(String candidate) {
    return PATTERN.matcher(candidate).matches();
}
}

```

这个类是一个值对象 (value object, <http://domaindrivendesign.org/node/135>)，这个概念是 Eric Evans 在《模型驱动设计》[Evans03]一书中定义的。值对象通常用来表述领域的概念，可以简单地将其实现为原始类型（在本例中就是字符串），但是值对象还允许在其内部实现领域约束。电子邮件地址必须要遵循一定的格式，否则，就会出现不合法的邮件地址。所以，我们实际上会通过一些正则表达式实现格式检查，这样就能阻止实例化非法的 EmailAddress。

这就意味着，在处理这种类型的实例时，能够确保有合法的电子邮件地址，所以我们不再需要专门的组件来校验它。在持久化映射方面，EmailAddress 带有 @Embeddable 注解，这会导致持久化供应商会将其属性放到包含它的类所对应的表中。在我们的场景下，这是一个列，我们将其定义为个性化的名字：email。

可以看到，我们需要为 JPA 持久化厂商提供一个空的构造方法，这样它才能够通过反射实例化 EmailAddress 对象（见示例 4-3）。这是一个很明显的缺点，因为没有办法让 emailAddress 是 final 的或者断言它是非空的。用于 NoSQL 实现的 Spring Data 映射子系统并没有将这一需求暴露给开发人员。作为示例，可以查看 6.3 小节“映射模块”来了解在 MongoDB 中如何对更为严格的值对象实现进行建模。

#### 示例 4-4 Customer 领域类

```

@Entity
public class Customer extends AbstractEntity{

    private String firstname, lastname;

    @Column(unique = true)
    private EmailAddress emailAddress;
    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    @JoinColumn(name = "customer_id")
    private Set<Address> addresses;
}

```

我们在电子邮件地址上使用了 @Column 注解，以保证某一个电子邮件地址不能被多个客户所使用，这样就能根据电子邮件来唯一地查询客户了。最后，我们声明 Customer 有一个 Address 的集合。这个属性需要更多关注一下，因为我们在这里定义了很多事情。

首先，也是最基础的，我们使用 @OneToMany 注解来指明一个 Customer 可以拥有多个 Address。在这个注解中，我们将级联类型 (cascade) 设置成了 CascadeType.ALL，

并为 Address 启用了子对象移除 (orphan removal)。这会产生多种影响。例如，当我们持久化、更新或者删除 Customer 时，Address 也会被持久化、更新或删除。因此，我们不再需要在前端持久化 Address 实例了并且当删除 Customer 时，也不用再关心删除所有的 Address 了。持久化厂商会做到这一点。需要注意的一点是，这并不是数据库级别的级联，而是你的 JPA 持久化厂商所管理的级联。除此之外，将子对象移除设置为 true，那么如果 Address 在集合中被移除了，它们也会在数据库中被删除。

以上所有的结果将会导致 Address 的生命周期被 Customer 所控制，这种关系是一种经典的组合，在领域驱动设计 (domain-driven design) 术语中，Customer 会被成为聚合根 (aggregate root, [http://en.wikipedia.org/wiki/Domain-driven\\_design#Building\\_blocks\\_of\\_DDD](http://en.wikipedia.org/wiki/Domain-driven_design#Building_blocks_of_DDD))，因为它不仅为自己也为其他实体控制持久化操作以及约束。最后，我们在 addresses 属性上使用 @JoinColumn，这会导致持久化厂商为 Address 对象后端所对应的表添加另外一列。这一列会用来引用 Customer，从而实现表关联。如果我们遗漏了这个注解的话，持久化厂商将会创建一个专门的连接表。

在包 core 中，最后一部分内容是 Product，如示例 4-5 所示。就像前面讨论的其他类一样，它包含了多种基本属性，所以无需添加注解就能使持久化厂商对它们产生映射。我们只是在定义 name 和 price 属性时，使用 @Column 注解，其目的是将其定义为强制性的属性。除此之外，还添加了一个 Map，它会用来存储额外的属性，不同的产品之间这些属性可能是不同的。

#### 示例 4-5 Product 领域类

```
@Entity
public class Product extends AbstractEntity {

    @Column(nullable = false)
    private String name;
    private String description;

    @Column(nullable = false)
    private BigDecimal price;
    @ElementCollection
    private Map<String, String> attributes = new HashMap<String, String>();
}
```

对于构建基本的客户关系管理 (Customer Relation Management, CRM) 或库存系统而言，我们已经万事俱备了。接下来，为了实现系统中 Product 的订单，我们需要进行一些抽象。首先，我们引入了 LineItem，它会持有对 Product 的引用以及 Product 的数量和购买的价格。匹配 Product 属性时，我们使用了 @ManyToOne 注解，它实际上会转化成 LineItem 对应表中的 product\_id 列，这一列会用来指向 Product，如示例 4-6 所示。

### 示例 4-6 LineItem 领域类

```
@Entity
public class LineItem extends AbstractEntity {

    @ManyToOne
    private Product product;

    @Column(nullable = false)
    private BigDecimal price;
    private int amount;
}
```

完成这个“拼图游戏”的最后一块就是 Order 实体了，它基本上就是一个指针，指向了 Customer、投递地址、账单地址以及表示实际购买的多个 LineItem（如示例 4-7 所示）。LineItem 的映射与前面看到的 Customer 和 Address 之间映射类似。Order 会自动对 LineItem 实例进行级联持久化操作。因此，没有必要单独管理 LineItem 的持久化生命周期。其他的属性都是已经讨论过的多对一关系。要注意的是，我们为 Order 定义了一个个性化的表名，因为在大多数的数据库中，Order 本身就是一个保留字，因此，所生成的建表 SQL 以及为查询和数据操作生成的 SQL 在执行时都会导致异常产生。

### 示例 4-7 Order 领域类

```
@Entity
@Table(name = "Orders")
public class Order extends AbstractEntity {

    @ManyToOne(optional = false)
    private Customer customer;
    @ManyToOne
    private Address billingAddress;

    @ManyToOne(optional = false, cascade = CascadeType.ALL)
    private Address shippingAddress;
    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    @JoinColumn(name = "order_id")
    private Set<LineItem>;
    ...

    public Order(Customer customer, Address shippingAddress,
                Address billingAddress) {
        Assert.notNull(customer);
        Assert.notNull(shippingAddress);

        this.customer = customer;
        this.shippingAddress = shippingAddress.getCopy();
        this.billingAddress = billingAddress == null ? null :
            billingAddress.getCopy();
    }
}
```

最后一个值得注意的方面是 Order 类的构造器使用了投递地址和账单地址的副本。这能够保证方法传递进来的 Address 实例如果发生变化的话，不会关联影响到已经存在的订单。如果我们不创建这个副本的话，如果顾客稍后修改他的地址信息，那使用这个地址创建的订单中对应的地址也会随之发生变化。

## 4.2 传统方式

在开始之前，首先看一下 Spring Data 如何为领域模型实现数据访问层，并讨论如何按照传统的方式实现数据访问层。在示例项目中会找到示例的实现和客户端都使用了一些额外的注解如 @Profile（对于实现）和 @ActiveProfile（对于测试用例）。因为 Spring Data Repository 方式会创建一个 CustomerRepository 实例，并且我们还有一个为手动实现所创建的实例。因此，这里使用了 Spring 的 Profile 机制，只对单一的测试用例启动传统的实现方式。我们并没有在这里的实例代码中展示这些注解，因为如果按照传统的方式来实现整个数据访问层的话，它们实际上并不会被用到。

为了使用普通的 JPA 来持久化前述的实体，现在要创建一个接口以及针对我们存储的实现，如示例 4-8 所示。

### 示例 4-8 为 Customer 定义的存储接口

```
public interface CustomerRepository {  
    Customer save(Customer account);  
    Customer findByEmailAddress(EmailAddress emailAddress);  
}
```

这样，我们定义了一个 save(...) 方法来存储账号以及根据电子邮件地址查找所有账号的查询方法。现在看一下如果基于普通 JPA 实现的话，存储的实现类会是什么样子，如示例 4-9 所示。

### 示例 4-9 对于 Customer 的传统存储实现

```
@Repository  
@Transactional(readOnly = true)  
class JpaCustomerRepository implements CustomerRepository {  
  
    @PersistenceContext  
    private EntityManager em;  
  
    @Override  
    @Transactional  
    public Customer save(Customer customer) {  
  
        if (customer.getId() == null) {  
            em.persist(customer);  
            return customer;  
        } else {  
            return em.merge(customer);  
        }  
    }  
}
```

```
    }

    @Override
    public Customer findByEmailAddress(EmailAddress emailAddress) {

        TypedQuery<Customer> query = em.createQuery(
            "select c from Customer c where c.emailAddress = :emailAddress", Customer.class);
        query.setParameter("emailAddress", emailAddress);

        return query.getSingleResult();
    }
}
```

实现类中使用了 JPA 的 EntityManager，因为设置了 JPA 的@PersistenceContext 注解，因此它将会通过 Spring 容器注入进来。这个类设置了@Repository 注解，因此会将 JPA 的异常转换为 Spring 的 DataAccessException 异常体系。除此之外，我们使用了@Transactional 注解，从而保证 save(...) 操作运行在事务之中，并且还允许为 findByEmailAddress(...) 设置 readOnly 标志（在类级别）。这有助于在持久化厂商内部以及数据库级别对性能进行优化。

因为我们不想让客户端来决定该调用 EntityManager 的 merge(...) 方法还是 persist(...) 方法，所以我们使用 Customer 的 id 域来判断某一个 Customer 对象是不是新建的。当然，这个逻辑可以抽取出一个通用的存储超类之中，因为我们可能不希望对每一个特定的领域对象存储实现都重复这段代码。查询方法也很简单直接：我们创建 Query、绑定参数并执行查询获取结果。它的实现非常简单，你甚至可以将实现代码视为样板。稍微想象一下，我们可以根据方法签名衍生出实现：想要获得单个 Customer、查询与方法的名字很接近，并且只需将方法参数绑定给它。所以，正如你所看到的，这里有提升的空间。

## 4.3 启动示例代码

应用程序的组件现在已经就绪了，所以我们现在将其启动起来并运行在 Spring 容器之中。为了做到这一点，我们需要做两件事：首先，需要配置通用的 JPA 基础设施（如连接数据库的 DataSource 以及 JPA EntityManagerFactory）。对于前者，我们会使用 HSQL，它是支持运行在内存之中的数据库。对于后一个问题，我们选择使用 Hibernate 作为持久化厂商。在示例工程的 pom.xml 文件中建立了对它的依赖。其次，需要构建 Spring 容器，这个容器会找到存储实现并为其创建一个 Bean 实例。在示例 4-10 中，会有一个 Spring 的 JavaConfig 配置类，它会完成上面提到的各个步骤。

## 示例 4-10 Spring 的 JavaConfig 配置

```
@Configuration  
@ComponentScan  
@EnableTransactionManagement  
class ApplicationConfig {  
  
    @Bean  
    public DataSource dataSource() {  
        EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();  
        return builder.setType(EmbeddedDatabaseType.HSQL).build();  
    }  
  
    @Bean  
    public LocalContainerEntityManagerFactoryBean entityManagerFactory() {  
  
        HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();  
        vendorAdapter.setDatabase(Database.HSQL);  
        vendorAdapter.setGenerateDdl(true);  
  
        LocalContainerEntityManagerFactoryBean factory =  
            new LocalContainerEntityManagerFactoryBean();  
        factory.setJpaVendorAdapter(vendorAdapter);  
        factory.setPackagesToScan(getClass().getPackage().getName());  
        factory.setDataSource(dataSource());  
  
        return factory;  
    }  
  
    @Bean  
    public PlatformTransactionManager transactionManager() {  
        JpaTransactionManager txManager = new JpaTransactionManager();  
        txManager.setEntityManagerFactory(entityManagerFactory());  
        return txManager;  
    }  
}
```

@Configuration 注解声明了这个类为 Spring 的 JavaConfig 配置类。@ComponentScan 会命令 Spring 扫描 ApplicationConfig 所在的包及其子包来查找 Spring 组件（带有 @Service、@Repository 等注解的类）。@EnableTransactionManagement 为带有 @Transactional 注解的方法启用 Spring 管理的事务。

带有@Bean 注解的方法现在声明了如下的基础设施组件：dataSource()将会借助 Spring 对内嵌数据库的支持创建一个内嵌数据源。这样可以很容易地搭建各种内存数据库用于测试，这基本上不用什么配置。我们在这里选择的是 HSQL（其他的可选方案包括 H2 或 Derby）。在此之上，我们配置了一个 EntityManagerFactory。

在这里使用了 Spring 3.1 的新特性，它可以让我们将完全不用创建 persistence.xml 文件来声明实体类，而是通过 LocalContainerEntityManagerFactoryBean 类的 packagesToScan 属性实现对 Spring 的类路径的扫描。这会触发 Spring 扫描那些带有@Entity 和@MappedSuperclass 注解的类，并且会自动将其加入到 JPA PersistenceUnit 之中。

相同的配置功能如果用 XML 的话，如示例 4-11 所示。

### 示例 4-11 基于 XML 的 Spring 配置

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

    <context:component-scan base-package="com.oreilly.springdata.jpa" />

    <tx:annotation-driven />

    <bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory" />
    </bean>

    <bean id="entityManagerFactory"
        class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="packagesToScan" value="com.oreilly.springdata.jpa" />
        <property name="jpaVendorAdapter">
            <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
                <property name="database" value="HSQL" />
                <property name="generateDdl" value="true" />
            </bean>
        </property>
    </bean>

    <jdbc:embedded-database id="dataSource" type="HSQL" />
</beans>
```

在这个示例代码的底部，`<jdbc:embedded-database />` 创建了使用 HSQL 的内存数据源。LocalContainerEntityManagerFactoryBean 与我们刚刚在 JavaConfig 场景（见示例 4-10）中所看到的代码类似。在此之上，我们声明了 JpaTransactionManager 并激活了基于注解的事务配置以及扫描组件的基础包。需要注意的一点是，示例 4-11 中的 XML 配置与示例工程中 META-INF/spring/application-context.xml 文件稍微有些不同。因为示例代码的目标是基于 Spring Data JPA 的数据访问层实现，它会废弃掉一些刚刚看到的配置。

示例的应用类会创建 AnnotationConfigApplicationContext，它会使用 Spring 的 JavaConfig 配置类来启动应用组件（如示例 4-12 所示）。我们在 ApplicationConfig 配置类中所声明的基础设施组件以及添加注解的存储实现将会被找到并实例化。因

此，可以使用 CustomerRepository 类型的 Spring Bean 来创建 Customer、对其进行存储并按照它的电子邮件地址进行查找。

#### 示例 4-12 启用示例代码

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = ApplicationConfig.class)
class CustomerRepositoryIntegrationTests {

    @Autowired
    CustomerRepository customerRepository;

    @Test
    public void savesAndFindsCustomerByEmailAddress {

        Customer dave = new Customer("Dave", "Matthews");
        dave.setEmailAddress("dave@dmbar.com");

        Customer result = repository.save(dave);
        Assert.assertThat(result.getId(), is(notNullValue()));

        result = repository.findByEmailAddress("dave@dmbar.com");
        Assert.assertThat(result, is(dave));
    }
}
```

## 4.4 使用 Spring Data Repository

为了启用 Spring Data Repository 的功能，必须让 Spring Data Repository 的基础设施找到 Repository 接口。为了做到这一点，需要让 CustomerRepository 扩展自 Spring Data Repository 的标识接口。除此之外，还要保留了以前所声明的持久化方法，如示例 4-13 所示。

#### 示例 4-13 Spring Data CustomerRepository 接口

```
public interface CustomerRepository extends Repository<Customer, Long> {

    Customer save(Account account);

    Customer findByEmailAddress(String emailAddress);
}
```

save(...)方法将会依靠通用的 SimpleJpaRepository，它实现了所有 CRUD 方法。我们所声明的查询方法将会依赖于查询衍生机制，如在 2.2.2 小节“衍生查询”中所提到的那样。我们还需要一种方式来启用 Spring Data Repository 的基础设施，关于这一点可以使用 XML，也可以使用 JavaConfig。对于 JavaConfig 方式来说，所要做的只是在配置类上添加@EnableJpaRepositories 注解即可。在示例中移除了@ComponentScan，因为不再需要查找手工实现了。同样地情况也适用于@EnableTransactionManagement 注解。Spring Data Repository 的基础设施将自动处理对 Repository 的方法调用，会将其置于事务之中。关于事务设置的更多细节，可

以参考 4.4.1 小节“事务性”。本来也可以保留这些注解以创建更为完整的应用，但是移除了它们，是因为我们不想造成这样一种印象，那就是构建简单的数据访问功能，这些注解也都是必需的。最终，`ApplicationConfig` 类的头部信息如示例 4-14 所示。

#### 示例 4-14 使用 JavaConfig 启用 Spring Data Repository

```
@Configuration  
@EnableJpaRepositories  
class ApplicationConfig {  
  
    // ... as seen before  
}
```

如果使用 XML 配置的话，则需要添加 JPA 命名空间中的 `repositories` XML 命名空间元素，如示例 4-15 所示。

#### 示例 4-15 通过 XML 命名空间启用 JPARepository

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

为了看到它的运行效果，请查看 `CustomerRepositoryIntegrationTest`。它使用了 `AbstractIntegrationTest` 中所构建的 Spring 配置，将 `CustomerRepository` 装配到测试用例之中并运行与 `JpaCustomerRepositoryIntegrationTest` 相同的测试，但是我们不需要为 `Repository` 接口提供任何的实现。让我们看一下 `Repository` 中定义的每个方法并简要介绍一下 Spring Data JPA 都为每个方法做了些什么，如示例 4-16 所示。

#### 示例 4-16 为 Customer 所定义的 Repository 接口

```
public interface CustomerRepository extends Repository<Customer, Long> {  
  
    Customer findOne(Long);  
  
    Customer save(Customer account);  
  
    Customer findByEmailAddress(Address emailAddress);  
}
```

`findOne(...)` 和 `save(...)` 方法依靠 `SimpleJpaRepository` 来实现，实际上就是这个类的实例支撑了 Spring Data 基础设施所创建的代理。所以，仅仅通过匹配方法签名，对这两个方法的调用就会被分发到该实现类上。如果想要暴露更完整的 CRUD 方法，只需要扩展 `CrudRepository` 接口来代替 `Repository` 就可以了，因为它已经包含了这些方法。要注意的是，如何避免暴露 `delete(...)` 方法以阻止删除 `Customer` 实例，如果扩展了 `CrudRepository`，这个方法是会暴露出来的。关于调节选项的更多细节，可参考 2.3.1 小节“调整 Repository 接口”。

最后要讨论的方法是 `findByEmailAddress(...)`，它显然不是一个 CURD 方法，但是它的意图是要作为查询来执行的。因为我们没有做任何的手动声明，所以 Spring Data JPA 启动时就需要探查这个方法并试图根据它来衍生出查询。衍生机制（细节

可以参考 2.2.2 小节“衍生查询”) 将会发现 `EmailAddress` 是 `Customer` 所引用的合法属性, 并且最终会创建一个 JPA Criteria API 查询, 其等同的 JPQL 是“`select c from Customer c where c.emailAddress = ?1`”。因为方法会返回单个 `Customer`, 所以查询执行的预期结果最多返回一个实体。如果没有找到 `Customer`, 会得到 `null`; 如果找到的结果不止一个, 那么将会出现 `IncorrectResultSizeDataAccessException` 异常。

让我们继续看 `ProductRepository` 接口 (如示例 4-17 所示)。可能会首先发现与 `CustomerRepository` 不同, 在这里我们扩展 `CrudRepository` 接口, 因为我们想要完整的 CRUD 方法集合。`findByDescriptionContaining(...)` 方法很明显是一个查询方法。这里有几点需要注意。首先, 我们不仅引用了 `Product` 的 `description` 属性, 而且用 `Containing` 关键字来限制断言。最终会给指定的 `description` 参数前后添加%字符, 结果所形成的 String 将会通过 Like 操作符来进行限制。因此, 查询如下: `select p from Product p where p.description like ?1`, 如果给定的 `description` 是 `Apple` 的话, 限制词就会是 `%Apple%`。第二件有意思的事情就是我们使用了分页抽象 (pagination abstraction), 这样就能获取匹配条件的 `Product` 的一个子集。`ProductRepositoryIntegrationTest` 中的 `lookupProductsByDescription()` 测试用例展现了这些方法是如何被使用的 (如示例 4-18 所示)。

#### 示例 4-17 为 `Product` 所定义的 Repository 接口

```
public interface ProductRepository extends CrudRepository<Product, Long> {  
    Page<Product> findByDescriptionContaining(String description, Pageable pageable);  
  
    @Query("select p from Product p where p.attributes[?1] = ?2")  
    List<Product> findByAttributeAndValue(String attribute, String value);  
}
```

#### 示例 4-18 对 `ProductRepository` `findByDescriptionContaining(...)` 方法的测试用例

```
@Test  
public void lookupProductsByDescription() {  
  
    Pageable pageable = new PageRequest(0, 1, Direction.DESC, "name");  
    Page<Product> page = repository.findByDescriptionContaining("Apple", pageable);  
  
    assertThat(page.getContent(), hasSize(1));  
    assertThat(page, Matchers.<Product> hasItems(named("iPad")));  
    assertThat(page.getTotalElements(), is(2L));  
    assertThat(page.isFirstPage(), is(true));  
    assertThat(page.isLastPage(), is(false));  
    assertThat(page.hasNextPage(), is(true));  
}
```

我们创建了一个新的 `PageRequest` 实例, 用它来获取第一页的数据并声明每页的大小是一条, 要按照 `Product` 的 `name` 进行降序排列。然后, 我们将 `Pageable` 传递到方法之中, 并确保返回的是 `iPad`、目前是第一页并且还有其他页可访问。可以看到, 对分页方法的执行返回了必要的元数据信息, 可以用来知道如果不使用分页的话会

得到多少条记录。如果不使用 Spring Data 的话，要读取这些元数据需要手动编写额外的查询执行代码，它会基于实际的查询做一个求记录数的投射。关于使用 Repository 方法进行分页的更多细节，参见 2.2.3 小节“分页与排序”。

ProductRepository 的第二个方法是 `findByAttributeAndValue(...)`。我们希望获取到自定义属性为某个给定值的所有 Product。因为 `attributes` 被映射为 `@ElementCollection`（见示例 4-5），所以很遗憾无法使用查询衍生机制来为我们创建查询。为了手动定义要执行的查询，在这里使用了 `@Query` 注解。一般来讲，如果查询变得越来越复杂的话，这种方式也会带来便利。这是因为就算它们可以衍生出查询，但是方法的名字也会变得非常冗长。

最后，看一下 OrderRepository（如示例 4-19 所示），它看起来应该已经很熟悉了。查询方法 `findByCustomer(...)` 会触发查询衍生机制（如前面讲解所示）并且最终形成的查询为“`select o from Order o where o.customer = ? 1`”。与其他的 Repository 不同的是，我们扩展自 `PagingAndSortingRepository` 接口，而这个接口又是扩展自 `CrudRepository` 的。`PagingAndSortingRepository` 在 `CrudRepository` 已提供的 `findAll(...)` 之上又提供了分别接受 `Sort` 和 `Pageable` 参数的方法。这里的主要场景就是我们想逐页访问 Order，而不是一次加载所有的数据。

#### 示例 4-19 为 Order 定义的 Repository 接口

```
public interface OrderRepository extends PagingAndSortingRepository<Order, Long> {  
    List<Order> findByCustomer(Customer customer);  
}
```

### 4.4.1 事务性

基于 JPA `EntityManager` 的一些 CRUD 操作需要激活事务。为了让 Spring Data Repository 为 JPA 所提供的功能尽可能便利，`CrudRepository` 和 `PagingAndSortingRepository` 背后的实现类已经添加了默认配置的 `@Transactional` 注解，这样的话 Spring 事务将会自动参与进来，甚至如果当前没有激活事务的话，它将会触发新的事务。要了解 Spring 事务的基本信息的话，可以查阅 Spring 参考文档 (<http://static.springsource.org/spring/docs/current/spring-framework-reference/html/transaction.html>)。

如果 Repository 实现类触发了事务的话，那么它会为 `save(...)` 和 `delete(...)` 操作创建默认的事务（存储默认的隔离级别、没有配置超时、只对运行时异常回滚），并对所有的查询方法包括分页创建只读的事务。为读取方法创建只读的事务有以下的好处：首先，这个标识将会传递给底层的 JDBC 驱动，这个驱动（取决于数据库供应商）将会进行优化，甚至阻止你意外执行修改式的查询。除此之外，Spring 事务的基础设施与 `EntityManager` 的生命周期进行了集成并且允许将它的 `FlushMode` 设置

为 MANUAL，从而能够避免为持久化上下文中的每个实体检查状态变化（也就是所谓的脏检查），尤其是大量对象加载到持久化上下文中的情况下，这可以带来很明显的性能提升。

如果想为一些 CRUD 方法的事务配置做一些微调的话（如配置指定的超时时间），那么可以重新声明所需的 CRUD 方法并为其添加@Transactional 注解，在这个注解中可以为方法声明进行自己的设置，它的优先级会高于 SimpleJpaRepository 中声明的默认配置，如示例 4-20 所示。

#### 示例 4-20 在 CustomerRepository 接口中重新配置事务

```
public interface CustomerRepository extends Repository<Customer, Long> {  
  
    @Transactional(timeout = 60)  
    Customer save(Customer account);  
}
```

使用自定义的 Repository 基础接口，当然也是可行的，可参见 2.3.1 小节“调整 Repository 接口”。

### 4.4.2 Repository 与 Querydsl 集成

既然已经看到了如何为 Repository 接口添加查询方法，那现在让我们来看一下如何使用 Querydsl 来为实体动态地创建断言并通过 Repository 抽象来执行它们。第 3 章曾总体介绍过 Querydsl 是什么以及它是如何工作的。

为了生成元模型类，要在 pom.xml 文件中配置 Querydsl Maven 插件，如示例 4-21 所示。

#### 示例 4-21 为 JPA 构建 Querydsl APT 处理器

```
<plugin>  
    <groupId>com.mysema.maven</groupId>  
    <artifactId>maven-apt-plugin</artifactId>  
    <version>1.0.4</version>  
    <configuration>  
        <processor>com.mysema.query.apt.jpa.JPAAnnotationProcessor</processor>  
    </configuration>  
    <executions>  
        <execution>  
            <id>sources</id>  
            <phase>generate-sources</phase>  
            <goals>  
                <goal>process</goal>  
            </goals>  
            <configuration>  
                <outputDirectory>target/generated-sources</outputDirectory>  
            </configuration>  
        </execution>  
    </executions>  
</plugin>
```

在这里，唯一特定于 JPA 的事情就是使用了 `JPAAnnotationProcessor`。它会让插件按照 JPA 的映射注解来查找实体、与其他实体的关系以及可嵌入式的内容 (`embeddable`)，等等。生成会在正常的构建过程中进行，生成的类将会位于 `target` 下的一个文件夹下。因此，每次清理构建的时候，它们会被清除掉，不要将它们签入到源码控制系统之中。

如果正在使用 Eclipse 并将这个插件添加到了工程之中，则需要触发 Maven 的工程更新（在工程上点击右键并选择“`Maven→Update Project...`”）。这会把配置的输出目录设置为源码文件夹，此时，使用生成类的代码就能正常编译了。

一旦这些准备就绪，就会看到生成的查询类，如 `QCustomer`、`QProduct` 等。让我们在 `ProductRepository` 上下文中探索一下这些生成类的功能。为了能够在 `Repository` 中执行 `QueryDSL` 断言，我们将 `QueryDSLPredicateExecutor` 添加到继承类型的列表中，如示例 4-22 所示。

#### 示例 4-22 扩展了 `QueryDSLPredicateExecutor` 的 `ProductRepository` 接口

```
public interface ProductRepository extends CrudRepository<Product, Long>,  
    QueryDSLPredicateExecutor<Product> { ... }
```

这会将 `findAll(Predicate predicate)` 和 `findOne(Predicate predicate)` 这样的方法加入到 API 之中。现在一切已经就绪，可以开始使用生成的类了。让我们看一下 `QueryDSLProductRepositoryIntegrationTest`，如示例 4-23 所示。

#### 示例 4-23 使用 `QueryDSL` 断言来查找 `Product`

```
QProduct product = QProduct.product;  
  
Product iPad = repository.findOne(product.name.eq("iPad"));  
Predicate tablets = product.description.contains("tablet");  
  
Iterable<Product> result = repository.findAll(tablets);  
assertThat(result, is(Matchers.<Product> iterableWithSize(1)));  
assertThat(result, hasItem(iPad));
```

首先，获得了 `QProduct` 元模型类的一个引用并将其放到 `product` 属性中。现在，我们可以使用它来导航生成的路径表达式（path expression），从而就能创建断言了。我们使用 `product.name.eq("iPad")` 来查找名为 `iPad` 的 `Product` 并将其保存为引用。我们所构建的第二个断言指明了要查找描述中包含 `tablet` 的 `Product`。然后，基于 `Repository` 执行这个 `Predicate` 实例，并且验证找到的 `iPad` 与前面的引用是相同的。

可以看到定义的断言非常易读和简洁。已构建的断言可以重新组合起来形成更高级别的断言，因此，能在实现灵活查询的同时又不增加复杂性。

# 借助 Querydsl SQL 实现类型安全的 JDBC 编程

对于关系型数据库来说，使用 JDBC 是很流行的选择。Spring 对 JDBC 的支持大部分都包含在 Spring 框架的 spring-jdbc 模块之中。关于对 JDBC 支持的话题，Madhusudhan Konda[Konda12]所编写的《Just Spring Data Access》一书是一个很好的指导。Spring Data 项目的 Spring Data JDBC 扩展子项目也提供了一些很有用处的特性，这是本章要介绍的内容。我们将会看到围绕使用 Querydsl 实现类型安全的查询方面最新的开发进展。

除了对 Querydsl 的支持，Spring Data JDBC 扩展子项目也包含了一些对特定数据库的支持，如对 Oracle 数据库所提供的连接故障转移（connection failover）、消息队列以及增强的存储过程支持等。这些特性只针对于 Oracle 数据库，并没有通用性，因此在本书中不会涵盖这些内容。Spring Data JDBC 扩展子项目有一个非常详细的参考指导介绍了这些特性，如果感兴趣的话可以查阅。

## 5.1 示例工程与搭建过程

长期以来，在 Java 程序中，我们都使用字符串来定义数据库查询，正如前文所述这种方式易于出错。列或表的名字可能会发生变化，我们可能会添加一列或更改已有列的类型。在 Java IDE 中，我们已经习惯于为 Java 类做类似的重构，而 IDE 会指导我们找到所有需要修改的引用，包括在注释或配置文件中的引用。对于包含复杂 SQL 查询的表达式字符串来说，并没有这样的功能支持。为了避免这种问题，我们提供了类型安全查询的替代方案，也就是 Querydsl。很多的数据访问技术都能很好地集成 Querydsl，第 3 章中已提供了

一些背景知识。在这一部分，我们将会关注 Querydsl SQL 模块以及它如何与 Spring 的 JdbcTemplate 进行集成，每个 Spring 开发人员对于 JdbcTemplate 应该都很熟悉。

不过，在看新的 JDBC 支持功能之前，需要讨论一些通用的关注点，如数据库的配置以及工程构建系统的搭建。

### 5.1.1 HyperSQL 数据库

在本章的 Querydsl 示例中，使用 HyperSQL 数据库的 2.2.8 版本 (<http://hsqldb.org/>)。HyperSQL 有一个很好的特性，就是能够以服务器的模式以及内存的模式运行数据库。对于集成测试来说，基于内存的方式是很好的选择，因为数据库的启动和停止可以通过应用程序的配置来进行控制，这是通过 Spring 的 Embedded DatabaseBuilder 来实现的，如果使用 spring-jdbc XML 命名空间的话，那就是 <jdbc:embedded-database> 标签。构建脚本会自动地下载依赖并启动内存数据库。要使用独立的服务器模式的话，需要下载分发包并将其解压到系统的目录之中。这些完成之后，我们可以切换到已解压的分发包的 hsqldb 目录下，并使用以下命令来启动数据库：

```
java -classpath lib/hsqldb.jar org.hsqldb.server.Server \
--database.0 file:data/test --dbname.0 test
```

运行这个命令会启动服务器，它会产生一些输出日志以及服务器已经启动的提示信息，也会告知我们使用 Ctrl-C 会停止服务器。现在，可以打开另一个命令窗口并切换到相同的 hsqldb 目录下，我们可以启动数据库客户端，这样的话就可以与数据库交互了（如创建表以及运行查询等）。对于 Windows 来说，我们只需要执行 bin 目录下的 runManagerSwing.bat 批处理文件；对于 OS X 或 Linux，我们需要运行以下的命令：

```
java -classpath lib/hsqldb.jar org.hsqldb.util.DatabaseManagerSwing
```

这会产生如图 5-1 所示的登录对话框。我们将 Type 修改为“HSQL Database Engine Server”，并将“test”作为数据库的名字添加到 URL 上，这样 URL 就会是“jdbc:hsqldb:hsq://localhost/test”。默认的用户是“sa”，密码为空。一旦连接成功，就会看到一个激活的 GUI 数据库客户端。

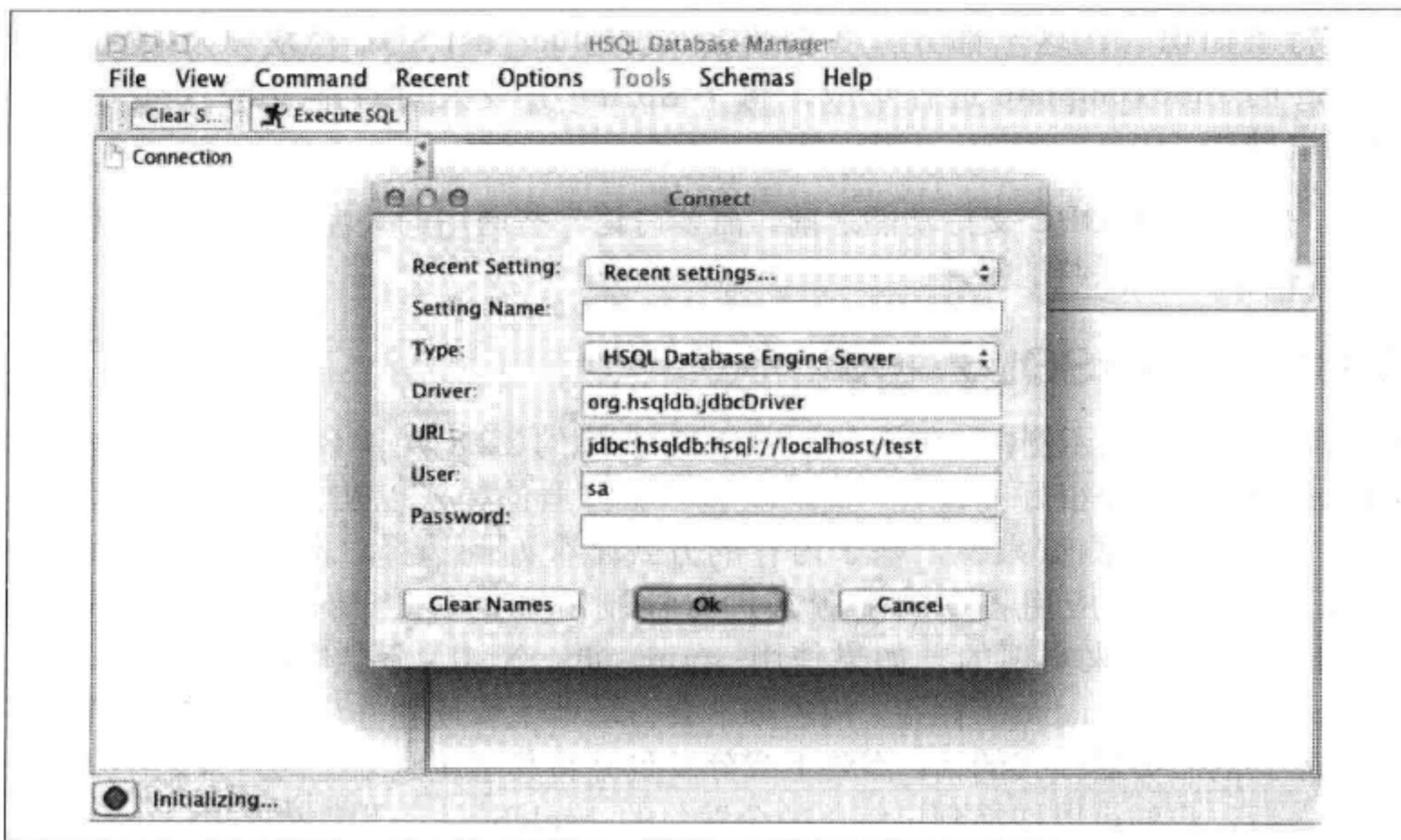


图 5-1 HSQLDB 客户端登录对话框

### 5.1.2 Querydsl 的 SQL 模块

Querydsl 的 SQL 模块为 Java 开发人员提供了一种类型安全的可选方案来与关系型数据库交互。Querydsl 会基于数据库表的元数据生成查询类型，而不是在 Java 程序中编写 SQL 查询并将其嵌入到字符串中。我们可以使用这些生成的类型来编写查询并对数据库执行 CURD 操作，从而避免以字符串的形式提供列名或表名。

相对于其他的 Querydsl 模块，SQL 模块生成查询类型的方式稍微有所不同。它不是依赖于注解，SQL 模块依赖于实际的数据库表和可用的 JDBC 元数据来生成查询类型。这意味着在生成查询类之前，要把表创建好并且需要访问一个可用的数据库。有鉴于此，我们建议将其作为一个单独的构建步骤来运行并将生成的类作为工程的一部分存储到源码控制系统之中。只有当修改了表结构并且尚没有签入代码的时候，才需要重新运行这个步骤。我们期望持续集成系统也运行这个步骤，因为 Java 类型和数据表的任何不匹配都能在构建的时候发现。

稍后会看一下我们需要生成的查询类型，但首先需要理解它们包含了什么以及如何使用它们。它们包含了 Querydsl 用来生成查询的信息，也包含了构造查询、执行更新、插入和删除以及将数据映射到领域对象的信息。让我们快速看一个表的例子，这个表包含了地址信息。表 address 有 3 个 VARCHAR 类型的列：street、city 以及 country。示例 5-1 展示了创建这个表的 SQL 语句。

## 示例 5-1 创建 address 表

```
CREATE TABLE address (
    id BIGINT IDENTITY PRIMARY KEY,
    customer_id BIGINT CONSTRAINT address_customer_ref
        FOREIGN KEY REFERENCES customer (id),
    street VARCHAR(255),
    city VARCHAR(255),
    country VARCHAR(255));
```

示例 5-2 展示了基于 address 表所生成的查询类型。它有多个构造函数、针对每列的 Querydsl 路径表达式、创建主键和外键类型的方法并提供了 QAddress 类型实例的静态域。

## 示例 5-2 生成的查询类型——QAddress

```
package com.oreilly.springdata.jdbc.domain;

import static com.mysema.query.types.PathMetadataFactory.*;
import com.mysema.query.types.*;
import com.mysema.query.types.path.*;

import javax.annotation.Generated;

/**
 * QAddress is a Querydsl query type for QAddress
 */
@Generated("com.mysema.query.sql.codegen.MetaDataSerializer")
public class QAddress extends com.mysema.query.sql.RelationalPathBase<QAddress> {

    private static final long serialVersionUID = 207732776;

    public static final QAddress address = new QAddress("ADDRESS");

    public final StringPath city = createString("CITY");
    public final StringPath country = createString("COUNTRY");
    public final NumberPath<Long> customerId = createNumber("CUSTOMER_ID", Long.class);
    public final NumberPath<Long> id = createNumber("ID", Long.class);
    public final StringPath street = createString("STREET");
    public final com.mysema.query.sql.PrimaryKey<QAddress> sysPk10055 = createPrimaryKey(id);
    public final com.mysema.query.sql.ForeignKey<QCustomer> addressCustomerRef =
        createForeignKey(customerId, "ID");

    public QAddress(String variable) {
        super(QAddress.class, forVariable(variable), "PUBLIC", "ADDRESS");
    }

    public QAddress(Path<? extends QAddress> entity) {
        super(entity.getType(), entity.getMetadata(), "PUBLIC", "ADDRESS");
    }

    public QAddress(PathMetadata<?> metadata) {
        super(QAddress.class, metadata, "PUBLIC", "ADDRESS");
    }
}
```

在 Java 代码中，可以通过以下的方式创建引用：

```
QAddress qAddress = QAddress.address;
```

这样就可以更容易地使用 `qAddress` 来引用表和列，而不用再借助于字符串常量了。在示例 5-3 中，查找所有 `city` 为“London”的 `address`，并获取其 `street`、`city` 以及 `country` 字段。

### 示例 5-3 使用生成的查询类

```
QAddress qAddress = QAddress.address;
SQLTemplates dialect = new HSQLDBTemplates();
SQLQuery query = new SQLQueryImpl(connection, dialect)
    .from(qAddress)
    .where(qAddress.city.eq("London"));
List<Address> results = query.list(
    new QBean<Address>(Address.class, qAddress.street,
    qAddress.city, qAddress.country));
```

首先，创建对查询类型的引用以及对应于所使用数据库的 `SQLTemplates` 实例，在我们的场景下也就是 `HSQLDBTemplates`。`SQLTemplates` 封装了不同数据库之间的差异，类似于 Hibernate 中的 `Dialect`。接下来，使用 JDBC `javax.sql.Connection` 和 `SQLTemplates` 作为参数构造了一个 `SQLQuery`。使用 `from` 方法来指明要查询的表，这需要传递进来查询的类型。然后，通过 `where` 方法来提供 `where` 子句或断言，在这里我们使用了 `qAddress` 引用来指明 `city` 等于“London”。

执行这个 `SQLQuery`，使用了 `list` 方法，它会返回结果的一个 `List`。我们同时使用 `QBean` 提供了映射的实现，它以领域类型以及包含了 `street`、`city` 以及 `country` 列的投射作为参数。

我们得到的结果是 `Address` 的 `List`，其中 `Address` 已经通过 `QBean` 进行了填充。`QBean` 类似于 Spring 的 `BeanPropertyRowMapper`，它需要领域类型遵循 JavaBean 的规则。作为替代方案，也可以使用 `MappingProjection`，它类似于 Spring 中的 `RowMapper`，对于结果如何映射到领域对象上你会有更多的控制权。

基于这个简单的例子，我们总结一下 SQL 查询所用到的 `Querydsl` 组件。

- `SQLQueryImpl` 类，它持有目标表，可能还会有断言或 `where` 子句相关的其他表，如果我们查询多表的话，可能还会有连接表达式。
- 断言（`Predicate`），通常会是 `BooleanExpression` 的形式，允许我们对结果声明过滤器。
- 映射或结果提取器（`results extractor`），通常是 `QBean` 或 `MappingProjection` 的形式，其参数是一个或多个表达式所形成的投射。

到此为止，我们还没有与任何的 Spring 特性进行集成，不过本章后面的内容将会涉及这种集成。第一个例子的目的就在于介绍 `Querydsl` SQL 模块的基础

知识。

### 5.1.3 构建系统集成

本章中 Querydsl 部分的代码位于 GitHub 示例工程 (<https://github.com/SpringSource/spring-data-book>) 的 `jdbc` 模块之中。

在工程中真正使用 Querydsl 之前，需要配置构建系统，这样才能生成查询类型。Querydsl 提供了 Maven 和 Ant 的集成，其文档位于 Querydsl 参考文档 (<http://www.querydsl.com/static/querydsl/latest/reference/html/>) 的“Querying SQL”一章。

在 Maven 的 `pom.xml` 文件中添加了插件配置，如示例 5-4 所示。

#### 示例 5-4 搭建用于代码生成的 Maven 插件

```
<plugin>
  <groupId>com.mysema.querydsl</groupId>
  <artifactId>querydsl-maven-plugin</artifactId>
  <version>${querydsl.version}</version>
  <configuration>
    <jdbcDriver>org.hsqldb.jdbc.JDBCDriver</jdbcDriver>
    <jdbcUrl>jdbc:hsqldb:hsq://localhost:9001/test</jdbcUrl>
    <jdbcUser>sa</jdbcUser>
    <schemaPattern>PUBLIC</schemaPattern>
    <packageName>com.oreilly.springdata.jdbc.domain</packageName>
    <targetFolder>${project.basedir}/src/generated/java</targetFolder>
  </configuration>
  <dependencies>
    <dependency>
      <groupId>org.hsqldb</groupId>
      <artifactId>hsqldb</artifactId>
      <version>2.2.8</version>
    </dependency>
    <dependency>
      <groupId>ch.qos.logback</groupId>
      <artifactId>logback-classic</artifactId>
      <version>${logback.version}</version>
    </dependency>
  </dependencies>
</plugin>
```

需要使用如下的 Maven 命令明确执行这个插件：

```
mvn com.mysema.querydsl:querydsl-maven-plugin:export
```

通过指明执行目标，可以将这个插件设置为 `generate-sources` 生命周期的一部分来执行。在示例工程中，就是这样做的，同时也使用了预先定义的 HSQL 数据库，这样就能避免在构建示例工程时强制启动数据库。不过，在真实的工作场景中，需要有一个可修改的模式的数据库并且要重新运行 Querydsl 的代码生成过程。

## 5.1.4 数据库模式

既然构建已经配置完成，那就可以生成查询类了，但是先看看这一部分所使用的数据库模式。我们已经看到了 address 表，现在添加了 customer 表，这个表与 address 表有一对多的关系。为 HSQLDB 数据库所定义的模式如示例 5-5 所示。

示例 5-5 schema.sql

```
CREATE TABLE customer (
    id BIGINT IDENTITY PRIMARY KEY,
    first_name VARCHAR(255),
    last_name VARCHAR(255),
    email_address VARCHAR(255));

CREATE UNIQUE INDEX ix_customer_email ON CUSTOMER (email_address ASC);

CREATE TABLE address (
    id BIGINT IDENTITY PRIMARY KEY,
    customer_id BIGINT CONSTRAINT address_customer_ref FOREIGN KEY REFERENCES customer (id),
    street VARCHAR(255),
    city VARCHAR(255),
    country VARCHAR(255));
```

customer 和 address 这两个表通过由 address 到 customer 的外键连接了起来。同时，我们为 address 表的 email\_address 列上定义了唯一索引。

这样，我们的领域模型实现就如图 5-2 所示。

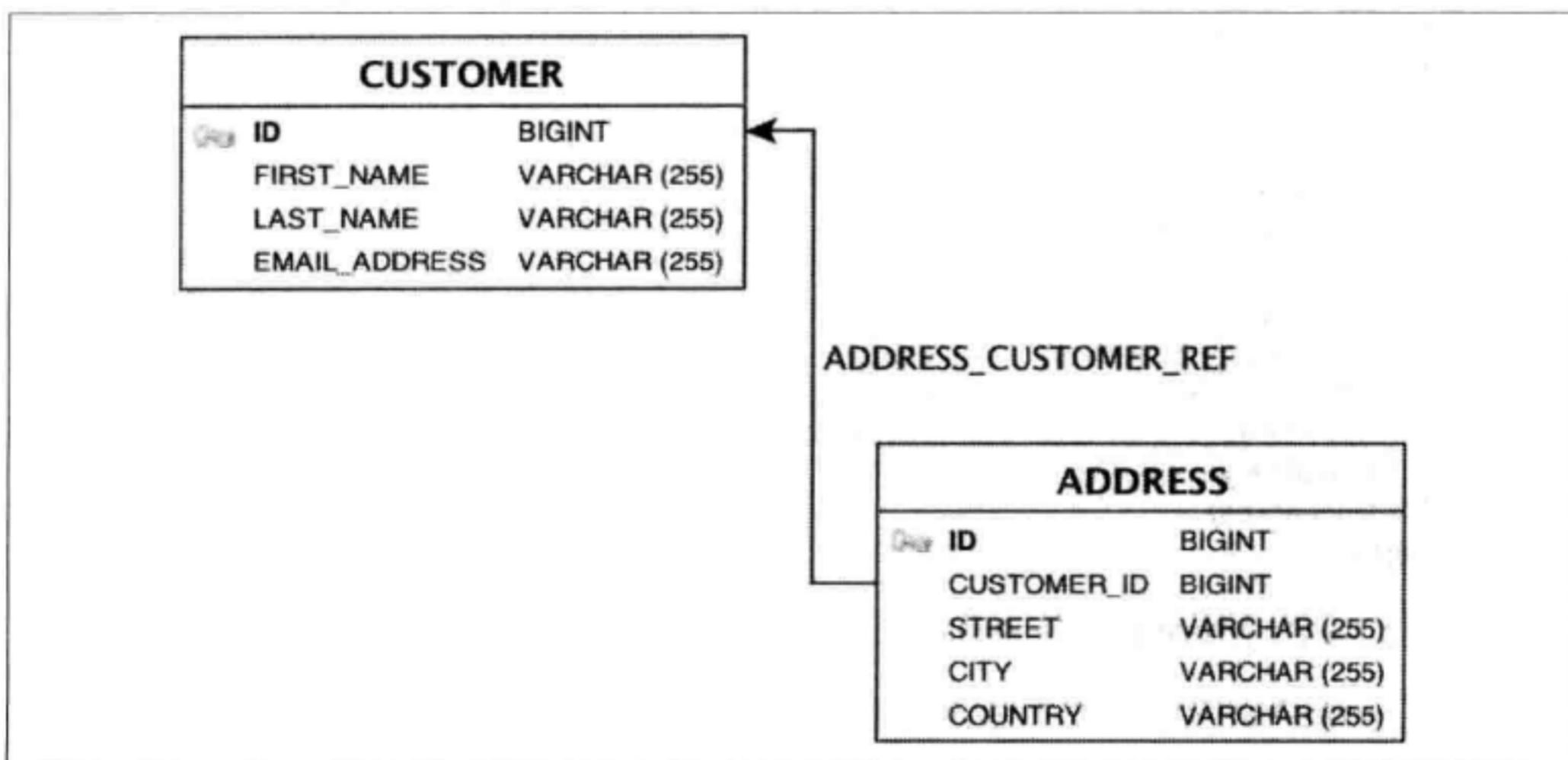


图 5-2 使用 QueryDSL SQL 的领域模型实现

## 5.1.5 示例工程的领域实现

我们已经看到了数据库的模式，接下来将会看到对应的 Java 领域类，这些类会在示例中用到。我们需要 Customer 类和 Address 类来持有数据库表中的数据。这两

个类都扩展自 `AbstractEntity`，这个类除了有 `equals(...)` 和 `hashCode()` 方法以外，还有 `id` 的 `setter` 和 `getter` 方法，在这里 `id` 字段是 `Long` 类型的：

```
public class AbstractEntity {  
  
    private Long id;  
  
    public Long getId() {  
        return id;  
    }  
  
    public void setId(Long id) {  
        this.id = id;  
    }  
  
    @Override  
    public boolean equals(Object obj) { ... }  
  
    @Override  
    public int hashCode() { ... }  
}
```

`Customer` 类有名字和电子邮件信息以及地址的集合。这个实现是一个传统的 JavaBean，包含了所有属性的 `setter` 和 `getter` 方法：

```
public class Customer extends AbstractEntity {  
  
    private String firstName;  
    private String lastName;  
    private EmailAddress emailAddress;  
    private Set<Address> addresses = new HashSet<Address>();  
  
    public String getFirstName() {  
        return firstName;  
    }  
  
    public void setFirstName(String firstName) {  
        this.firstName = firstName;  
    }  
  
    public String getLastName() {  
        return lastName;  
    }  
  
    public void setLastName(String lastName) {  
        this.lastName = lastName;  
    }  
  
    public EmailAddress getEmailAddress() {  
        return emailAddress;  
    }  
  
    public void setEmailAddress(EmailAddress emailAddress) {  
        this.emailAddress = emailAddress;  
    }  
}
```

```
public Set<Address> getAddresses() {
    return Collections.unmodifiableSet(addresses);
}

public void addAddress(Address address) {
    this.addresses.add(address);
}

public void clearAddresses() {
    this.addresses.clear();
}
}
```

电子邮件地址在数据库中存储为 VARCHAR 列，但是在 Java 类中我们使用了 EmailAddress 值对象类型，它会使用正则表达式对电子邮件地址进行校验。这个类与我们在其他章节看到的是相同的：

```
public class EmailAddress {

    private static final String EMAIL_REGEX = "...";
    private static final Pattern PATTERN = Pattern.compile(EMAIL_REGEX);

    private String value;

    protected EmailAddress() {
    }

    public EmailAddress(String emailAddress) {
        Assert.isTrue(isValid(emailAddress), "Invalid email address!");
        this.value = emailAddress;
    }

    public static boolean isValid(String source) {
        return PATTERN.matcher(source).matches();
    }
}
```

最后一个领域类是 Address 类，同样也是一个传统的 JavaBean，包含了地址属性的 setter 和 getter 方法。除了无参的构造器，我们还有一个接收所有地址属性的构造器：

```
public class Address extends AbstractEntity {
    private String street, city, country;

    public Address() {
    }
    public Address(String street, String city, String country) {
        this.street = street;
        this.city = city;
        this.country = country;
    }

    public String getCountry() {
        return country;
    }

    public void setCountry(String country) {
```

```

    this.country = country;
}

public String getStreet() {
    return street;
}

public void setStreet(String street) {
    this.street = street;
}

public String getCity() {
    return city;
}

public void setCity(String city) {
    this.city = city;
}
}

```

前面介绍的 3 个类构成了我们的领域模型，它们位于 JDBC 示例工程的 com.oreilly.springdata.jdbc.domain 包中。现在，我们看一下 CustomerRepository 的接口定义：

```

public interface CustomerRepository {

    Customer findById(Long id);

    List<Customer> findAll();

    void save(Customer customer);

    void delete(Customer customer);

    Customer findByEmailAddress(EmailAddress emailAddress);
}

```

我们有几个查找方法以及用于保存和删除的方法。这其中并没有任何的存储方法来保存和删除 Address 对象，因为它们始终属于 Customer 实例。当 Customer 实例保存的时候，我们必须要持久化它所提供的 address 信息。

## 5.2 QueryDslJdbcTemplate

Spring Data 与 QueryDSL 集成的核心类是 QueryDslJdbcTemplate。它对 Spring 标准的 Jdbc-Template 进行了包装，这个类具有管理 SQLQuery 实例和执行查询的方法，另外还有一些方法，可以按照命令特定的回调执行插入、更新以及删除操作。我们将会在这一部分涵盖上述的所有内容，但首先需要创建 QueryDslJdbcTemplate。

要配置 QueryDslJdbcTemplate，只需要在构造器中传入 DataSource：

```
QueryDslJdbcTemplate qdslTemplate = new QueryDslJdbcTemplate(dataSource);
```

或者是一个配置好的 JdbcTemplate：

```
jdbcTemplate = new JdbcTemplate(dataSource);
QueryDslJdbcTemplate qdslTemplate = new QueryDslJdbcTemplate(jdbcTemplate);
```

现在已经具有了配置好的 `QueryDslJdbcTemplate` 供使用。在前面我们看到，当创建 `SQLQuery` 对象的时候，通常需要提供 `Connection` 和匹配数据库的 `SQLTemplates`。不过，当使用 `QueryDslJdbcTemplate` 的时候，没有必要这样做。按照通常的 Spring 编程风格，JDBC 层会管理所有的数据库资源，如连接和结果集。它还会根据所管理的连接，提供基于数据库元数据的 `SQLTemplates` 实例。为了获得所管理的 `SQLQuery` 实例，需要使用 `QueryDslJdbcTemplate` 的静态工厂方法 `newSqlQuery`:

```
SQLQuery sqlQuery = qdslTemplate.newSqlQuery();
```

所获得的 `SQLQuery` 实例还没有活跃的连接，所以需要使用 `QueryDslJdbcTemplate` 的 `query` 方法以便连接管理生效:

```
SQLQuery addressQuery = qdslTemplate.newSqlQuery()
    .from(qAddress)
    .where(qAddress.city.eq("London"));

List<Address> results = qdslTemplate.query(
    addressQuery,
    BeanPropertyRowMapper.newInstance(Address.class),
    qAddress.street, qAddress.city, qAddress.country);
```

这里有两个查询方法: `query` 会返回一个 `List` 而 `queryForObject` 只会返回一个结果。每个方法又有 3 种重载的版本，分别接收如下的参数。

- 通过 `newSqlQuery` 工厂方法获得的 `SQLQuery` 对象。
- 如下映射器和投射实现组合的某一种:
  - `RowMapper` 以及用一个或多个 `Expression` 构成的投射;
  - `ResultSetExtractor` 以及用一个或多个 `Expression` 构成的投射;
  - `ExpressionBase`, 通常表现为 `QBean` 或 `MappingProjection`。

前两个映射器 `RowMapper` 和 `ResultSetExtractor` 是标准的 Spring 接口，在常规的 `JdbcTemplate` 中也经常用到。它们负责抽取查询所返回的结果数据。`ResultSetExtractor` 会抽取返回的所有行的数据，而 `RowMapper` 只会处理一行并且对每行数据会重复调用。`QBean` 和 `MappingProjection` 是 `Querydsl` 的类，同样也是映射一行数据。选择哪一个完全取决于你；它们都能完成相同的任务。对于我们的大多数例子将会使用 Spring 类型——毕竟我们这本书叫做《Spring Data 实战》。

## 5.3 执行查询

现在，通过实现 `CustomerRepository` 中的查询方法，来看一下如何使用 `QueryDslJdbc`

Template 执行查询。

### 5.3.1 Repository 实现起步

实现类将会自动装配 DataSource；在这个设置中，将创建 QueryDslJdbcTemplate 以及针对表中各列的投射，当获取 Customer 实例所需的数据时，这个投射会被所有的查询用到，如示例 5-6 所示。

#### 示例 5-6 构建 QueryDslCustomerRepository 实例

```
@Repository  
@Transactional  
public class QueryDslCustomerRepository implements CustomerRepository {  
  
    private final QCustomer qCustomer = QCustomer.customer;  
    private final QAddress qAddress = QAddress.address;  
  
    private final QueryDslJdbcTemplate template;  
    private final Path<?>[] customerAddressProjection;  
  
    @Autowired  
    public QueryDslCustomerRepository(DataSource dataSource) {  
  
        Assert.notNull(dataSource);  
  
        this.template = new QueryDslJdbcTemplate(dataSource);  
        this.customerAddressProjection = new Path<?>[] { qCustomer.id, qCustomer.firstName,  
            qCustomer.lastName, qCustomer.emailAddress, qAddress.id, qAddress.customerId,  
            qAddress.street, qAddress.city, qAddress.country };  
    }  
  
    @Override  
    @Transactional(readOnly = true)  
    public Customer findById(Long id) { ... }  
  
    @Override  
    @Transactional(readOnly = true)  
    public Customer findByEmailAddress(EmailAddress emailAddress) { ... }  
    @Override  
    public void save(final Customer customer) { ... }  
  
    @Override  
    public void delete(final Customer customer) { ... }  
}
```

我们正在编写的是存储类，所以开始添加了@Repository 注解。这是一个标准的 Spring 通用注解，这样，组件在类路径扫描的时候能够被发现。除此之外，如果使用 ORM 风格的数据访问技术来实现实存存储类，那么它还会为存储类进行异常的转换，也就是从特定 ORM 的异常转换为 Spring 的 DataAccessException 异常体系。在我们的场景中，所使用的是基于模板的方式，模板本身会提供这种异常转换功能。

接下来是@Transactional 注解。这也是一个标准的 Spring 注解，它表明对这个类中

所有方法的调用都会包装在数据库事务之中。只要在配置中包含了事务管理器实现，那我们就不用关心在存储类的代码中启动和完成事务了。

我们还定义了对两个所生成的查询类型的引用，也就是 QCustomer 和 QAddress。在数组 customerAddressProjection 中为查询保存了 Querydsl Path 条目，每个 Path 对应于要查询的一列。

构造器添加了@Autowired 注解，这意味着当 Repository 实现在进行配置的时候，容器将会把应用上下文中所定义的 DataSource 注入进来。这个类剩余的部分由 CustomerRepository 中所定义的方法组成，我们需要为其提供实现，那现在就开始吧。

### 5.3.2 查询单个对象

首先，实现 findById 方法，如示例 5-7 所示。要用来进行查找的 ID 是作为唯一的参数传递进来的。因为这是只读的方法，添加@Transactional(readOnly = true)注解以给出一个提示，一些 JDBC 驱动就可以利用它来提升事务处理的能力。即便有的 JDBC 驱动不会使用它，对只读的方法添加这样一个可选的属性也不会有什么负面影响。

#### 示例 5-7 查询单个对象

```
@Transactional(readOnly = true)
public Customer findById(Long id) {

    SQLQuery findByIdQuery = template.newSqlQuery()
        .from(qCustomer)
        .leftJoin(qCustomer._addressCustomerRef, qAddress)
        .where(qCustomer.id.eq(id));
    return template.queryForObject(
        findByIdQuery,
        new CustomerExtractor(),
        customerAddressProjection);
}
```

首先，创建一个 SQLQuery 实例。正如前文所述，当使用 QueryDslJdbcTemplate 的时候，需要让模板来管理 SQLQuery 实例，这就是为什么需要用静态工厂方法 newSqlQuery() 来获得实例。SQLQuery 类提供了流畅的接口，每个方法又会返回这个 SQLQuery 实例。这样就能将多个方法连接在一起，代码会更易阅读。使用 from 方法指明要查询的主表。然后，使用 leftJoin(...) 方法添加对 address 表的左连接。表 address 和 customer 之间所有匹配外键的 address 行会被包含进来。如果没有匹配的，那么在返回的结果集中 address 列为 null。如果不止有一个 address，那么对于每个 customer 将会得到多行的 address。在稍后将其映射为 Java 对象时，这是必须要进行处理的。SQLQuery 的最后一部分使用 where 方法来声明断言并且所提供的条件是 id 列必须等于 id 参数。

SQLQuery 创建完成之后，通过 QueryDslJdbcTemplate 的 queryForObject 方法执行查询，在调用的时候要将 SQLQuery 和映射器及投射一起传递进去，在我们的例子中，也就是之前所创建的 ResultSetExtractor 和 customerAddressProjection。前面曾经提到过，因为我们的查询包含了 leftJoin，所以需要处理每个 Customer 拥有多行的潜在可能性。

示例 5-8 是 CustomerExtractor 的实现。

#### 示例 5-8 用于单个对象的 CustomerExtractor

```
private static class CustomerExtractor implements ResultSetExtractor<Customer> {

    CustomerListExtractor customerListExtractor =
        new CustomerListExtractor(OneToManyResultSetExtractor.ExpectedResults.ONE_OR_NONE);

    @Override
    public Customer extractData(ResultSet rs) throws SQLException, DataAccessException {
        List<Customer> list = customerListExtractor.extractData(rs);
        return list.size() > 0 ? list.get(0) : null;
    }
}
```

可以看到，CustomerListExtractor 用来抽取 Customer 对象的 List，如果 List 中存在对象的话会返回里面第一个对象，如果 List 为空的话，将会返回 null。在 CustomerListExtractor 的构造器中，因为将 expectedResults 设置为 OneToManyResultSetExtractor.ExpectedResults.ONE\_OR\_NONE，所以我们能够知道结果不会超过一个。

### 5.3.3 OneToManyResultSetExtractor 抽象类

在查看 CustomerListExtractor 之前，先看一下它的基类，这是由 Spring Data JDBC 扩展项目所提供的特殊实现 OneToManyResultSetExtractor。示例 5-9 给出了 OneToManyResultSetExtractor 的概览。

#### 示例 5-9 用于抽取对象 List 的 OneToManyResultSetExtractor 的概览

```
public abstract class OneToManyResultSetExtractor<R, C, K>
    implements ResultSetExtractor<List<R>> {

    public enum ExpectedResults {
        ANY,
        ONE_AND_ONLY_ONE,
        ONE_OR_NONE,
        AT_LEAST_ONE
    }

    protected final ExpectedResults expectedResults;
    protected final RowMapper<R> rootMapper;
    protected final RowMapper<C> childMapper;

    protected List<R> results;
```

```

public OneToManyResultSetExtractor(RowMapper<R> rootMapper, RowMapper<C> childMapper) {
    this(rootMapper, childMapper, null);
}

public OneToManyResultSetExtractor(RowMapper<R> rootMapper, RowMapper<C> childMapper,
    ExpectedResults expectedResults) {

    Assert.notNull(rootMapper);
    Assert.notNull(childMapper);

    this.rootMapper = rootMapper;
    this.childMapper = childMapper;
    this.expectedResults = expectedResults == null ? ExpectedResults.ANY : expectedResults;
}

public List<R> extractData(ResultSet rs) throws SQLException, DataAccessException { .. }

/**
 * Map the primary key value to the required type.
 * This method must be implemented by subclasses.
 * This method should not call {@link ResultSet#next()}.
 * It is only supposed to map values of the current row.
 *
 * @param rs the ResultSet
 * @return the primary key value
 * @throws SQLException
 */
protected abstract K mapPrimaryKey(ResultSet rs) throws SQLException;
/**
 * Map the foreign key value to the required type.
 * This method must be implemented by subclasses.
 * This method should not call {@link ResultSet#next()}.
 * It is only supposed to map values of the current row.
 *
 * @param rs the ResultSet
 * @return the foreign key value
 * @throws SQLException
 */
protected abstract K mapForeignKey(ResultSet rs) throws SQLException;

/**
 * Add the child object to the root object
 * This method must be implemented by subclasses.
 *
 * @param root the Root object
 * @param child the Child object
 */
protected abstract void addChild(R root, C child);
}

```

OneToManyResultSetExtractor 扩展自 ResultSetExtractor 并以 List<T> 作为返回类型。方法 extractData 负责迭代结果集并抽取行数据。OneToManyResultSetExtractor 有 3 个抽象的方法必须由子类来实现，也就是 mapPrimaryKey、mapForeign Key 和 addChild。当迭代结果集时，这些方法用来识别主键和外键，这样就能识别何时会有新的根对象并且会帮助我们将映射的子对象添加到根对象之中。

`OneToManyResultSetExtractor` 类也需要 `RowMapper` 实现，以便为根对象和子对象提供所需要的映射。

### 5.3.4 CustomerListExtractor 实现

现在，继续看一下实际的 `CustomerListExtractor` 实现，它负责抽取 `customer` 和 `address` 的结果，如示例 5-10 所示。

#### 示例 5-10 用于抽取对象 List 的 CustomerListExtractor 实现

```
private static class CustomerListExtractor
    extends OneToManyResultSetExtractor<Customer, Address, Integer> {

    private static final QCustomer qCustomer = QCustomer.customer;

    private final QAddress qAddress = QAddress.address;

    public CustomerListExtractor() {
        super(new CustomerMapper(), new AddressMapper());
    }
    public CustomerListExtractor(ExpectedResults expectedResults) {
        super(new CustomerMapper(), new AddressMapper(), expectedResults);
    }

    @Override
    protected Integer mapPrimaryKey(ResultSet rs) throws SQLException {
        return rs.getInt(qCustomer.id.toString());
    }

    @Override
    protected Integer mapForeignKey(ResultSet rs) throws SQLException {
        String columnName = qAddress.addressCustomerRef.getLocalColumns().get(0).toString();
        if (rs.getObject(columnName) != null) {
            return rs.getInt(columnName);
        } else {
            return null;
        }
    }

    @Override
    protected void addChild(Customer root, Address child) {
        root.addAddress(child);
    }
}
```

`CustomerListExtractor` 扩展自 `OneToManyResultSetExtractor`，它调用了超类的构造方法并将所需的映射器传递进来，也就是为 `Customer` 类使用的 `CustomerMapper`（它是一对多关系的根元素）和为 `Address` 类使用的 `AddressMapper`（它是同一个一对多关系的子元素）。

除了这两个映射器，我们还需要为抽象的 `OneToManyResultSetExtractor` 类提供 `mapPrimaryKey`、`mapForeignKey` 和 `addChild` 方法的实现。

接下来，看一下所使用的 RowMapper 实现。

### 5.3.5 RowMapper 的实现类

这里所使用的 RowMapper 与常规 JdbcTemplate 中所使用的是一样的。它们实现了一个名为 mapRow 的方法，这个方法以 ResultSet 和行号作为参数。使用 QueryDslJdbcTemplate 所带来的唯一区别在于不需要使用字符串常量访问列了它使用查询类型来引用列的标签（label）。在 CustomerRepository 中有一个静态的方法，这个方法通过 Path 的 toString 方法来获取标签：

```
private static String columnLabel(Path<?> path) {  
    return path.toString();  
}
```

因为将 RowMapper 实现为静态内部类，所以它们可以访问这个私有的静态方法。

首先来看一下 Customer 对象的映射器，如示例 5-11 所示，通过 qCustomer 来指名列，这个对象引用的是 QCustomer 查询类型。

#### 示例 5-11 为 Customer 所提供的根 RowMapper 实现

```
private static class CustomerMapper implements RowMapper<Customer> {  
  
    private static final QCustomer qCustomer = QCustomer.customer;  
  
    @Override  
    public Customer mapRow(ResultSet rs, int rowNum) throws SQLException {  
  
        Customer c = new Customer();  
  
        c.setId(rs.getLong(columnLabel(qCustomer.id)));  
        c.setFirstName(rs.getString(columnLabel(qCustomer.firstName)));  
        c.setLastName(rs.getString(columnLabel(qCustomer.lastName)));  
  
        if (rs.getString(columnLabel(qCustomer.emailAddress)) != null) {  
            c.setEmailAddress(  
                new EmailAddress(rs.getString(columnLabel(qCustomer.emailAddress))));  
        }  
  
        return c;  
    }  
}
```

接下来看一下 Address 对象的映射器，它使用 qAddress 引用 QAddress 查询类型，如示例 5-12 所示。

#### 示例 5-12 为 Address 所提供的子 RowMapper 实现

```
private static class AddressMapper implements RowMapper<Address> {  
  
    private final QAddress qAddress = QAddress.address;  
  
    @Override  
    public Address mapRow(ResultSet rs, int rowNum) throws SQLException {
```

```

        String street = rs.getString(columnLabel(qAddress.street));
        String city = rs.getString(columnLabel(qAddress.city));
        String country = rs.getString(columnLabel(qAddress.country));

        Address a = new Address(street, city, country);
        a.setId(rs.getLong(columnLabel(qAddress.id)));

        return a;
    }
}

```

因为 Address 类具备所有属性的设置方法，所以我们原本可以使用 Spring 的 BeanPropertyRowMapper 而不必提供自定义的实现。

### 5.3.6 查询对象列表

当查询对象的列表时，与查询单个对象的过程是几乎是一样的，唯一的区别在于可以直接使用 CustomerListExtractor 了，而不必对其进行包装，也不必再返回 List 中的第一个对象，如示例 5-13 所示。

#### 示例 5-13 查询对象列表

```

@Transactional(readOnly = true)
public List<Customer> findAll() {

    SQLQuery allCustomersQuery = template.newSqlQuery()
        .from(qCustomer)
        .leftJoin(qCustomer._addressCustomerRef, qAddress);

    return template.query(
        allCustomersQuery,
        new CustomerListExtractor(),
        customerAddressProjection);
}

```

我们创建了一个 SQLQuery 并使用了 from(...) 和 leftJoin(...) 方法，但是这一次没有提供断言因为我们要返回所有的 customer。当执行这个查询时，直接使用了 CustomerListExtractor 以及前面所用到的 customerAddressProjection。

## 5.4 插入、更新和删除操作

除了刚刚讨论的查询特性，我们最后还要为 CustomerRepository 的实现类添加插入、更新和删除功能。借助于 Querydsl，数据是通过操作特定的子句（clause）来进行管理的，如 SQLInsertClause、SQLUpdateClause 和 SQLDeleteClause。在这一部分我们将会介绍借助 QueryDslJdbcTemplate 时如何使用它们。

### 5.4.1 使用 SQLInsertClause 进行插入操作

想往数据库中插入数据的时候，可以使用 Querydsl 提供的 SQLInsertClause 类。根

据表是自动生成主键还是明确提供主键，会有两个不同的 `execute(...)` 方法。对于自动生成主键的情况，要使用 `executeWithKey(...)` 方法。这个方法会返回生成的主键，你就可以将其设置到领域对象上。当由你来提供主键的时候，要使用 `execute` 方法，它会返回受影响的行数。`QueryDslJdbcTemplate` 有两个对应的方法：`insertWithKey(...)` 和 `insert(...)`。

我们所使用的场景是自动生成主键，因此对于数据的插入要使用 `insertWithKey(...)`，如示例 5-14 所示。`insertWithKey(...)` 方法的参数是查询类型的引用以及 `SqlInsertWithKeyCallback` 类型的回调，这个回调以生成主键的类型进行参数化。`SqlInsertWithKeyCallback` 回调接口只有一个名为 `doInSqlInsertWithKeyClause(...)` 的方法。这个方法以 `SQLInsertClause` 作为其参数。我们需要使用 `SQLInsertClause` 来设置值并调用 `executeWithKey(...)`。调用所返回的主键就是 `doInSqlInsertWithKeyClause` 的返回值。

#### 示例 5-14 插入对象

```
Long generatedKey = qdslTemplate.insertWithKey(qCustomer,
    new SqlInsertWithKeyCallback<Long>() {

    @Override
    public Long doInSqlInsertWithKeyClause(SQLInsertClause insert) throws SQLException {
        EmailAddress emailAddress = customer.getEmailAddress();
        String emailAddressString = emailAddress == null ? null : emailAddress.toString();

        return insert.columns(
            qCustomer.firstName, qCustomer.lastName, qCustomer.emailAddress)
            .values(customer.getFirstName(), customer.getLastName(), emailAddress);
            .executeWithKey(qCustomer.id);
    }
});

customer.setId(generatedKey);
```

#### 5.4.2 使用 `SQLUpdateClause` 进行更新操作

执行更新操作与插入操作很类似，不过此时不用担心生成主键。`QueryDslJdbcTemplate` 中的方法称为 `update`，它接收的参数是查询类型的引用以及 `SqlUpdateCallback` 类型的回调。`SqlUpdateCallback` 只有一个名为 `doInSqlUpdateClause(...)` 的方法，其唯一的参数为 `SQLUpdateClause`。在设置完更新的值并指明 `where` 子句后，就可以调用 `SQLUpdateClause` 的 `update` 方法，它会返回更新的行数，这个更新的行数也是我们需要从回调中返回的值，如示例 5-15 所示。

#### 示例 5-15 更新对象

```
qdslTemplate.update(qCustomer, new SqlUpdateCallback() {

    @Override
    public long doInSqlUpdateClause(SQLUpdateClause update) {
```

```
    EmailAddress emailAddress = customer.getEmailAddress();
    String emailAddressString = emailAddress == null ? null : emailAddress.toString();

    return update.where(qCustomer.id.eq(customer.getId()))
        .set(qCustomer.firstName, customer.getFirstName())
        .set(qCustomer.lastName, customer.getLastName())
        .set(qCustomer.emailAddress, emailAddressString)
        .execute();
    });
});
```

### 5.4.3 使用 SQLDeleteClause 进行删除行操作

删除比更新更为简单，使用 `QueryDslJdbcTemplate` 的 `delete` 方法即可，它接收的参数是查询类型的引用以及 `SqlDeleteCallback` 类型的回调。`SqlDeleteCallback` 只有一个名为 `doInSqlDeleteClause` 的方法，其唯一的参数为 `SQLDeleteClause`。在这里没有必要设置任何的值——只需提供 `where` 子句并执行即可。具体如示例 5-16 所示。

#### 示例 5-16 删除对象

```
qdslTemplate.delete(qCustomer, new SqlDeleteCallback() {

    @Override
    public long doInSqlDeleteClause(SQLDeleteClause delete) {
        return delete.where(qCustomer.id.eq(customer.getId())).execute();
    }
});
```



第三部分

---

# NoSQL



# MongoDB：文档存储

本章介绍 Spring Data MongoDB 项目。首先快速浏览用于文档存储的 MongoDB，并介绍为了使用我们的示例项目应该如何对它进行配置，最后以概述 MongoDB 的基本概念与它的原生 Java 驱动 API 来结束简介部分。在此之后，将会讨论 Spring Data MongoDB 模块的特性、Spring 命名空间、如何建立领域对象并将它映射到存储中，以及如何使用核心存储交互 API 即 MongoTemplate 来读写数据。本章结束时，将使用 Spring Data Repository 抽象来实现领域类的数据访问层。

## 6.1 MongoDB 简介

MongoDB 是一个文档存储数据库。文档是结构化的数据（基本上就是 Map），可包含基本类型值、集合值，甚至可将内嵌文档作为给定键的值。MongoDB 以 BSON 格式来存储这些文档，它是衍生于 JSON 的二进制格式。示例的文档如示例 6-1 所示。

### 示例 6-1 MongoDB 的文档样例

```
{ firstname : "Dave",
  lastname : "Matthews",
  addresses : [ { city : "New York", street : "Broadway" } ] }
```

firstname 和 lastname 都是基本类型 String 值，address 字段是一个数组值，包含一个嵌套的地址文档。文档（Documents）通过集合（Collection）组织在一起，而集合是一组文档的容器。通常会在单个集合中保存相同类型的文档，在这里，类型（type）的本质含义是“类似的结构”。从 Java 的观点来看，通常意味着每种类型（Customer 为一个，Product 也为一个）或每种类型层级（有一个集合保存 Contact，它可以是 Person 也可以是 Company）会有一个集合。

## 6.1.1 设置 MongoDB

开始使用 MongoDB 之前，可以先从项目网站 (<http://www.mongodb.org/downloads>) 将其下载下来。它提供了 Windows、OS X、Linux 和 Solaris 的二进制版本以及源码。最简单的方式是下载二进制包并将它们解压缩到适当的硬盘目录之中，如示例 6-2 所示。

### 示例 6-2 下载并解压 MongoDB 发布包

```
$ cd ~/dev
$ curl http://fastdl.mongodb.org/osx/mongodb-osx-x86_64-2.0.6.tgz > mongo.tgz

% Total    % Received % Xferd  Average Speed   Time     Time     Time  Current
                                         Dload  Upload   Total   Spent   Left  Speed
100 41.1M  100 41.1M    0      0  704k      0  0:00:59  0:00:59  ---:---  667k

$ tar -zxvf mongo.tgz

x mongodb-osx-x86_64-2.0.6/
x mongodb-osx-x86_64-2.0.6/bin/
x mongodb-osx-x86_64-2.0.6/bin/bsondump
x mongodb-osx-x86_64-2.0.6/bin/mongo
x mongodb-osx-x86_64-2.0.6/bin/mongod
x mongodb-osx-x86_64-2.0.6/bin/mongodump
x mongodb-osx-x86_64-2.0.6/bin/mongoexport
x mongodb-osx-x86_64-2.0.6/bin/mongofiles
x mongodb-osx-x86_64-2.0.6/bin/mongoimport
x mongodb-osx-x86_64-2.0.6/bin/mongorestore
x mongodb-osx-x86_64-2.0.6/bin/mongos
x mongodb-osx-x86_64-2.0.6/bin/mongosniff
x mongodb-osx-x86_64-2.0.6/bin/mongostat
x mongodb-osx-x86_64-2.0.6/bin/mongotop
x mongodb-osx-x86_64-2.0.6/GNU-AGPL-3.0
x mongodb-osx-x86_64-2.0.6/README
x mongodb-osx-x86_64-2.0.6/THIRD-PARTY-NOTICES
```

要启动 MongoDB，必须先建一个目录来存储数据然后使用 mongod 命令启动，并将它指向刚才建立的目录（见示例 6-3）。

### 示例 6-3 准备以及启动 MongoDB

```
$ cd mongodb-osx-x86_64-2.0.6
$ mkdir data
$ ./bin/mongod --dbpath=data

Mon Jun 18 12:35:00 [initandlisten] MongoDB starting : pid=15216 port=27017 dbpath=data
64-bit ...
Mon Jun 18 12:35:00 [initandlisten] db version v2.0.6, pdfile version 4.5
Mon Jun 18 12:35:00 [initandlisten] git version: e1c0cbc25863f6356aa4e31375add7bb49fb05bc
Mon Jun 18 12:35:00 [initandlisten] build info: Darwin erh2.10gen.cc 9.8.0 Darwin Kernel
Version 9.8.0: ...
Mon Jun 18 12:35:00 [initandlisten] options: { dbpath: "data" }
Mon Jun 18 12:35:00 [initandlisten] journal dir=data/journal
Mon Jun 18 12:35:00 [initandlisten] recover : no journal files present, no recovery needed
Mon Jun 18 12:35:00 [websvr] admin web console waiting for connections on port 28017
Mon Jun 18 12:35:00 [initandlisten] waiting for connections on port 27017
```

如你所见，MongoDB 已启动并使用指定的路径来存储数据，现在正在等待连接。

## 6.1.2 使用 MongoDB Shell

我们接着通过 MongoDB Shell 来了解最基本的操作。切换到刚刚解压 MongoDB 的目录并且使用 mongo 命令来运行 Shell，如示例 6-4 所示。

### 示例 6-4 启动 MongoDB Shell

```
$ cd ~/dev/mongodb-osx-x86_64-2.0.6  
$ ./bin/mongo
```

```
MongoDB shell version: 2.0.6  
connecting to: test  
>
```

Shell 将连接本地运行的 MongoDB 实例。可以使用 show dbs 命令来查看目前这个实例中所有可用的数据库。在示例 6-5 中，选择“local”数据库并执行 show collections 命令，目前为止我们的数据库还是空的，因此它没有显示任何数据。

### 示例 6-5 选择数据库并且查看集合

```
> show dbs  
local (empty)  
> use local  
switched to db local  
> show collections  
>
```

现在在数据库中增加一些数据，通过使用 save (...) 命令来存储我们所选择的集合，并且把相关的数据组成 JSON 格式输入到方法中。在示例 6-6 中增加了两个客户，也就是 Dave 和 Carter。

### 示例 6-6 将数据插入到 MongoDB

```
> db.customers.save({ firstname : 'Dave', lastname : 'Matthews',  
    emailAddress : 'dave@dmdband.com' })  
> db.customers.save({ firstname : 'Carter', lastname : 'Beauford' })  
> db.customers.find()  
{ "_id" : ObjectId("4fdf07c29c62ca91dcfd71c"), "firstname" : "Dave",  
  "lastname" : "Matthews", "emailAddress" : "dave@dmdband.com" }  
{ "_id" : ObjectId("4fdf07da9c62ca91dcfd71d"), "firstname" : "Carter",  
  "lastname" : "Beauford" }
```

命令中 customers 部分指明了数据所要存储到的集合，如果集合尚未存在的话，会立即创建。注意增加 Carter 时并没有包含 E-mail 地址，这就意味着文档中可以包含不同的属性组合。在默认情况下，MongoDB 不会强制使用模式（schema）。事实上，find(...) 命令可接受 JSON 文档作为输入来创建查询。如果要查找 E-mail 地址为 dave@dmdband.com 的客户，Shell 的交互过程如示例 6-7 所示。

### 示例 6-7 在 MongoDB 中查询数据

```
> db.customers.find({ emailAddress : 'dave@dmdband.com' })  
{ "_id" : ObjectId("4fdf07c29c62ca91dcfd71c"), "firstname" : "Dave",  
  "lastname" : "Matthews", "emailAddress" : "dave@dmdband.com" }
```

可以在 MongoDB 的首页 (<http://www.mongodb.org/display/DOCS/Tutorial>) 找到更多使用 MongoDB Shell 的知识。除此之外, [ChoDir10]也是很好的资源, 它更进一步探讨了存储的内部原理以及通常如何使用它来进行工作。

### 6.1.3 MongoDB Java 驱动

若要从 Java 程序中访问 MongoDB, 可以选择名为 10gen 的公司所提供并维护的 Java 驱动程序, 也是支撑 MongoDB 的公司。它与存储实例交互的核心抽象是 `Mongo`、`Database` 和 `DBCollection`。`Mongo` 类抽象了到 MongoDB 实例的连接, 它的默认构造器会为后续的操作连接一个本地运行的实例。正如你所见, 通用的 API 非常简洁易懂, 如示例 6-8 所示。

#### 示例 6-8 通过 Java 驱动访问 MongoDB 实例

```
Mongo mongo = new Mongo();
DB database = mongo.getDb("database");
DBCollection customers = db.getCollection("customers");
```

如同访问关系型数据库时所使用 `DataSource` 抽象一样, 它看起来很像典型的基础设施代码, 你可能会希望 Spring 能够在一定程度上对其进行管理。除此之外, 实例化 `Mongo` 对象或随后使用 `DBCollection` 均有可能抛出异常, 但是这些异常属于 MongoDB 特有的, 不应该侵入到客户端代码中。Spring Data MongoDB 会通过一些基础设施的抽象以及 Spring 命名空间来与 Spring 进行基本的集成从而进一步简化配置。关于这个问题请查阅 6.2 小节“使用 Spring 命名空间搭建基础设施”。

驱动的核心数据抽象是 `DBObject` 接口及其 `BasicDBObject` 实现类。它的用法基本上与简单的 Java Map 一样, 如示例 6-9 所示。

#### 示例 6-9 使用 Java 驱动创建 MongoDB 文档

```
DBObject address = new BasicDBObject("city", "New York");
address.put("street", "Broadway");
DBObject addresses = new BasicDBList();
addresses.add(address);

DBObject customer = new BasicDBObject("firstname", "Dave");
customer.put("lastname", "Matthews");
customer.put("addresses", addresses);
```

首先, 我们创建最终会成为嵌套文档的 `address` 数据, 并且把它封装到列表中。然后构造出基本的 `customer` 文档, 最后设置复杂的 `addresses` 属性。你可能已经意识到了, 与存储数据的交互非常底层, 如果要持久化 Java 领域对象, 必须手动将它们与 `BasicDBObject`s 进行映射。稍后我们将会看到如何使用 Spring Data MongoDB 来改善这个状况。刚才创建的文档可以通过 `DBCollection` 对象进行存储, 如示例 6-10 所示。

### 示例 6-10 使用 MongoDB Java 驱动持久化文档

```
DBCollection customers = db.getCollection("customers");
customers.insert(customer);
```

## 6.2 使用 Spring 命名空间搭建基础设施

首先，Spring Data MongoDB 会帮我们设置必要的基础设施来与 MongoDB 实例进行交互，这些基础设施就是一些 Spring Bean。使用 `JavaConfig`，可以简单地继承 `AbstractMongoConfiguration` 类，它包含许多基本配置，但是也可以通过重写方法来进行按需调整。我们的配置类如示例 6-11 所示。

### 示例 6-11 使用 JavaConfig 构建 MongoDB 基础设施

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

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

    @Override
    public Mongo mongo() throws Exception {
        Mongo mongo = new Mongo();
        mongo.setWriteConcern(WriteConcern.SAFE);
        return
    }
}
```

必须实现两个方法来设置 MongoDB 的基础设施，要提供数据库的名称以及一个封装了如何连接到数据库的 `Mongo` 实例。我们使用默认的构造函数，它会假设已经有一个运行在本地机器上的 MongoDB 实例并监听默认的端口 27017。稍后，将 `WriteConcern` 设置为 `SAFE` 模式。`WriteConcern` 定义了在写操作时驱动等待服务器的时间。默认设置是不等待，也不会对网络异常或者尝试写入非法数据做出响应（译者注：`WriteConcern` 默认设置为 `NORMAL`，会在网络错误时抛出异常，但是服务器错误不会抛异常。原文描述的应该是 `WriteConcern` 的 `NONE` 模式，而不是默认模式）。设置为 `SAFE` 将在网络错误时抛出异常，并让驱动一直等待，直到服务器可以正常写入数据为止。它也将在违反索引约束时做出响应，稍后会加以介绍。

这两个配置项将会在 `SimpleMongoDbFactory` Bean 定义里面结合起来（见 `AbstractMongoConfiguration` 的 `mongoDbFactory()` 方法）。`MongoDbFactory` 会被 `MongoTemplate` 实例使用，它也会通过基类来进行配置。`MongoTemplate` 是与 MongoDB 实例交互的主要 API，并可以持久化或取回对象。注意，在示例项目中所看到的配置类已经包含了扩展配置，稍后将会加以说明。

上述配置的 XML 版本如示例 6-12 所示。

### 示例 6-12 使用 XML 搭建 MongoDB 环境

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

    <mongo:db-factory id="mongoDbFactory" dbname="e-store" />

    <bean id="mongoTemplate" class="org.springframework.data.mongodb.core.MongoTemplate">
        <constructor-arg ref="mongoDbFactory" />
        <property name="writeConcern" value="SAFE" />
    </bean>

</beans>
```

<db-factory>元素使用类似于 JavaConfig 示例的方式来配置 SimpleMongoDbFactory，唯一的不同是它也可以预置要使用的 Mongo 实例，这在 JavaConfig 中必须手动设置。我们可以手动定义<mongo:mongo />元素以按需定制它的属性值，但是为了避免使用这种方式，我们直接在 MongoTemplate 中配置了 WriteConcern，这样只能调用模板配置好的 concern 方式来执行所有的写操作。

## 6.3 映射模块

为了便于持久化对象，Spring Data MongoDB 提供了一个映射模块，它能够检测到领域类的持久化元数据，并自动将这些对象转换成 MongoDB 的 DBObjects。我们接下来介绍领域建模的方法，以及为了满足对象-文档映射的需求都需要哪些元数据。

### 6.3.1 领域模型

首先，我们为顶级文档引入一个基类，如示例 6-13 所示。它只有一个 id 属性，这样就不用在所有要成为文档的类中重复定义这个属性。`@Id` 注解是可选的，默认情况下，我们以 `id` 或者 `_id` 的属性标识文档的 ID 域。因此，想使用不同属性名称或者想表达特殊的含义时，这个注解就可以派上用场。

#### 示例 6-13 AbstractDocument 类

```
public class AbstractDocument {
    @Id
    private BigInteger id;
}
```

`id` 属性的类型为 `BigInteger`。通常可以对 `id` 使用任何的类型，但某些类型对文档有着特殊的意义。在持久化文档中最被推荐的 `id` 类型一般是 `ObjectID`。`ObjectID` 是一个可支持集群环境的自增值对象。此外，MongoDB 也可以自动生成它们。同样在使用 Java 驱动时也推荐使用 `ObjectID` 作为 `id` 的类型，遗憾的是这将让领域对象对 Java 驱动形成依赖，而这可能是要避免的事情。因为 `ObjectID` 是 12 位的二进制值，它们可以很容易地转换成 `String` 或者 `BigInteger` 值。因此，如果使用 `String`、`BigInteger` 或者 `ObjectID` 作为 `id` 的类型，就可以利用 MongoDB 的 `id` 自增长功能，它会在保存之前自动将 `id` 值转换成 `ObjectID`，并且在读取时转换回来。如果手动把 `String` 值赋给不能转换成 `ObjectID` 的 `id` 字段的话，它将会按照原有的类型进行存储。所有用于 `id` 的其他类型也都会以这种方式存储。

## 地址和 Email 地址

如示例 6-14 所示，`Address` 是一个再简单不过的领域类，它简单地封装了 3 个定义为 `final` 的原生 `String` 值。映射模块会将这种类型的对象转换成 `DBObject`，在这个过程中会使用属性的名字作为字段的键并会恰当地设置它们的值，如示例 6-15 所示。

### 示例 6-14 Address 领域类

```
public class Address {  
  
    private final String street, city, country;  
  
    public Address(String street, String city, String country) {  
  
        Assert.hasText(street, "Street must not be null or empty!");  
        Assert.hasText(city, "City must not be null or empty!");  
        Assert.hasText(country, "Country must not be null or empty!");  
  
        this.street = street;  
        this.city = city;  
        this.country = country;  
    }  
  
    // ... additional getters  
}
```

### 示例 6-15 Address 对象的 JSON 表达方式

```
{ street : "Broadway",  
  city : "New York",  
  country : "United States" }
```

你也许注意到了，`Address` 类使用了一个复杂的构造函数以避免它的值被设置为非法状态。它与 `final` 字段结合，形成了一个经典案例：对象值的不可变性（immutable）。`Address` 之所以永远不会发生变化，原因在于要改变属性的值就会强制要求创建一个新的 `Address` 实例。这个类不提供无参构造函数，这就带来了一个问题，当 `DBObject` 从数据库读取后必须转换成 `Address` 实例时，对象该如何初始化？Spring Data 有一个持久化构

造函数（persistence constructor）的概念，这个构造函数用于初始化持久化的对象。类所提供的无参构造函数（隐式或显示）是最便利的方案，映射模块将会使用反射机制通过它来实例化实体对象。如果使用带参的构造函数，它会尝试根据属性名称来匹配参数名，并从存储表现类中获取值，在 MongoDB 的场景下指的也就是 DBObject。

另一个通过值对象将领域概念具体化的示例是 EmailAddress（见示例 6-16）。要将业务规则封装成代码，值对象是一种非常强大的方式，它可以让代码变得更有表现力、易读、易测试并且易于维护。更多深入的研究，请参考 Dan Bergh-Johnsson 针对这个主题的讨论，它位于 InfoQ (<http://www.infoq.com/presentations/Value-Objects-Dan-Bergh-Johnsson>) 站点上。如果是用普通的 String 类型来定义 Email 地址，就不能确保它是否进行过校验，或者是否为有效的 Email 地址。因此，这个简单的包装类会使用正则表达式来验证输入值，如果不符合表达式就不能通过验证。通过这种方法，如果客户端得到一个 EmailAddress 实例，就可以确保它们所处理的是一个合法的 Email 地址。

### 示例 6-16 EmailAddress 领域类

```
public class EmailAddress {  
  
    private static final String EMAIL_REGEX = ...;  
    private static final Pattern PATTERN = Pattern.compile(EMAIL_REGEX);  
  
    @Field("email")  
    private final String value;  
  
    public EmailAddress(String emailAddress) {  
        Assert.isTrue(isValid(emailAddress), "Invalid email address!");  
        this.value = emailAddress;  
    }  
  
    public static boolean isValid(String source) {  
        return PATTERN.matcher(source).matches();  
    }  
}
```

属性 value 使用了@Field 注解，它允许以一种自定义的方式将属性映射到 DBObject 中的字段。在我们的场景中，将通用的 value 映射为更为具体的 email。虽然可以将属性简单地命名为“email”，但目前的做法对于以下两种场景都能提供便利：假如要将类映射到已经存在的文档上，而该文档已经选择了你不想泄漏给领域对象的字段键（field key）时，@Field 通常可以将字段键和属性名解耦。其次，与关系模型不同，每一份文档中字段键都是重复的，因此它们会在文件数据中占据很大一部分空间，特别是在存储的值非常小的情况下。因此可以定义很简短的字段键名称来减少所需要的空间，不过这会稍微降低 JSON 表达式的可读性。

现在我们已经借助于基本的领域概念的实现完成了准备工作，接下来看看实际组成文档的类。

## Customer

如示例 6-17 所示，在 Customer 领域类中首先会注意到的是@Document 这个注解。在某种程度上，它其实是个可选的注解。如果没有使用这个注解话，映射模块仍然可以将类转换成 DBObject。那么为什么我们要在这里使用它呢？首先，我们可以配置映射基础设施来扫描要持久化的领域类，它只会选择以@Document 注解的类。当映射模块无法得知它所要处理的对象类型时，映射模块会马上自动检测这个类以获取映射信息，这会稍微降低第一次转换的效率。使用@Document 的第二个原因是定制领域对象存储时的 MongoDB 集合名称。如果没有使用注解或是没有为 Document 设置集合属性，集合的名称会是将第一个字母改为小写的类名，举个例子，Customer 的集合名称将会是“customer”。



示例项目中的代码稍微有些不同，因为之后在改进映射的时候会对模型进行微调。我们希望在现阶段尽量保持简单以简化介绍的过程，让你先专注于通用的映射部分。

### 示例 6-17 Customer 领域类

```
@Document
public class Customer extends AbstractDocument {

    private String firstname, lastname;

    @Field("email")
    private EmailAddress emailAddress;
    private Set<Address> addresses = new HashSet<Address>();

    public Customer(String firstname, String lastname) {
        Assert.hasText(firstname);
        Assert.hasText(lastname);

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

    // additional methods and accessors
}
```

Customer 类包含两个基本属性，这两个属性用来获取 firstname 和 lastname，除此之外还有 EmailAddress 领域类的属性以及一个 Address 的 Set 集合。emailAddress 属性使用了@Field 注解，如前所述，它可以让我们在 MongoDB 文档中定制键名。

注意，实际上不需要任何注解来配置 Customer 与 EmailAddress 和 Address 之间的关系，这么做的原因是 MongoDB 文档可能包含更复杂的值（如内嵌文档），这会严重影响类的设计和对象的持久化。从设计的角度来看，Customer 在领域驱动设计术语中叫做聚合根（aggregate root），Address 与 EmailAddresses 必须通过 Customer 实例进行存取。基本上我们在这里建立了一个树状结构以便更好地对应 MongoDB 文档模

型。因此相对于对象-关系的场景，对象-文件的映射更为简单。从持久化的观点看，存储整个 Customer 与它的 Address 以及 EmailAddress，就会是一次性的原子操作。在关系型数据库的世界里持久化一个这样的对象，需要逐个插入 Address，再加上一次 Customer 本身的插入操作（假设已经把 EmailAddress 内嵌到 Customer 表的列中）。由于表中每一行数据都是松耦合的关系，因此我们必须使用事务来保证插入的一致性。除此之外，为了满足外键关系还必须按照正确的顺序执行插入操作。

然而，文档模型不仅影响到持久化操作的写入端，还影响到了读取端，这通常会导致更多的数据库访问操作。由于集合中的文档是自包含的实体，存取它并不需要触及到其他的集合、文档等。按照关系型数据库的术语，文档实际上是一组预关联的数据集。特别是如果应用程序以特定的粒度来存取数据（在一定程度，这通常是驱动类设计的因素）的话，会在写入时将数据分开并且在每一次读取时又将它重新组合，这种做法显然不合理。完整的 Customer 文档如示例 6-18 所示。

### 示例 6-18 Customer 文档

```
{ "firstname": "Dave",
  "lastname": "Matthews",
  "email": { "email": "dave@dmdband.com" },
  "addresses": [ { "street": "Broadway",
    "city": "New York",
    "country": "United States" } ] }
```

需要注意，建模 Email 地址作为值对象时，需要将它序列化成内嵌对象，这会产生重复的键，从而让文档增加没必要的复杂度。我们暂且搁置这个话题，在 6.3.4 小节“自定义转换”中将会讨论改善的方法。

## Product

Product 领域类（见示例 6-19）不会带来太多的意外了。最有趣的部分或许是原生支持 Map 类型的存储。这也是由于文档本身的特点所致，`attributes` 会成为嵌套文档，Map 的条目会转换成文档的字段。注意，目前只能使用 String 类型作为 Map 的键。

### 示例 6-19 Product 领域类

```
@Document
public class Product extends AbstractDocument {

    private String name, description;
    private BigDecimal price;
    private Map<String, String> attributes = new HashMap<String, String>();
    //... additional methods and accessors
}
//... 其他方法和存取器
```

## Order 与 LineItem

接着我们介绍应用程序的订单模块，首先来看一下 LineItem 类，如示例 6-20 所示。

### 示例 6-20 LineItem 领域类

```
public class LineItem extends AbstractDocument {  
  
    @DBRef  
    private Product product;  
    private BigDecimal price;  
    private int amount;  
  
    // ... additional methods and accessors  
}
```

首先可以看到两个基本的属性，也就是 `price` 和 `amount`，因为它们会依照原生类型转换到文档字段中，所以在它们的定义中没有使用映射注解。而 `product` 属性使用了 `@DBRef` 注解，会使得 `LineItem` 中的 `Product` 对象不会被嵌入进来，而是存储一个指针，这个指针指向了 `Product` 集合中的某一个文档。这种方式很像关系型数据库中的外键。

注意，存储 `LineItem` 时，所引用的 `Product` 实例必须已经被存储，因此不会有级联操作。从存储中读取 `LineItem` 时，`Product` 的引用也会被解析，使得被引用的文档会被读取出来，并转换到 `Product` 实例中。

在结束该章节之前，我们来看一下 `Order` 领域类（见示例 6-21）。

### 示例 6-21 Order 领域类

```
@Document  
public class Order extends AbstractDocument {  
  
    @DBRef  
    private Customer customer;  
    private Address billingAddress;  
    private Address shippingAddress;  
    private Set<LineItem> lineItems = new HashSet<LineItem>();  
  
    // - additional methods and parameters  
}
```

在这里可以看到它组合使用了我们之前看到过的映射方式。这个类使用了 `@Document` 注解，因此在应用程序上下文启动时，它可以被映射模块发现并检测到对应的信息。对 `Customer` 的引用使用了 `@DBRef`，因为相对于内嵌到文档中，引用是一种更合适的方式。`Address` 属性以及 `LineItem` 使用自身的类型嵌入起来。

## 6.3.2 搭建映射的基础设施

前面已经介绍了领域类持久化的方法，现在来介绍如何搭建映射的基础设施，使之为我们工作。在大多数情况下这非常简单，一些使用基础设施的组件（稍后介绍）将使用适当的默认值来启用映射模块。也可以自行设置参数，稍微调整一下配置。在此发挥作用的两个核心抽象体是：`MongoMappingContext` 和 `MappingMongoConverter`。前者实际上负责建立领域类的元模型以避免反射查询（例如在每次持久化操作之前检测 `id` 属性或者确定字段键）。后者使用 `MappingContext` 所提供映射信

息来执行转换。可以简单地使用这两个抽象形式以编程的方式触发对象到 DBObject 的相互转换（见示例 6-22）。

### 示例 6-22 以编程的方式使用映射模块

```
MongoMappingContext context = new MongoMappingContext();
MongoDbFactory dbFactory = new SimpleMongoDbFactory(new Mongo(), "database");
MappingMongoConverter converter = new MappingMongoContext(dbFactory, context);

Customer customer = new Customer("Dave", "Matthews");
customer.setEmailAddress(new EmailAddress("dave@dmdband.com"));
customer.add(new Address("Broadway", "New York", "United States"));

DBObject sink = new BasicDBObject();
converter.write(customer, sink);

System.out.println(sink.toString());
{
    firstname : "Dave",
    lastname : "Matthews",
    email : { email : "dave@dmdband.com" },
    addresses : [ { street : "Broadway",
                    city : "New York",
                    country : "United States" } ]
}
```

首先创建了 MongoMappingContext 以及 SimpleMongoDbFactory 的对象。如果要立即加载带有@DBRef 注解的文档，那 SimpleMongoDbFactory 实例就是必要的。接着创建了 Customer 和 BasicDBObject 实例，并且通过调用 converter 实例完成转换。正如预期的一样，DBObject 会被数据填充。

### 使用 Spring 命名空间

Spring Data MongoDB 提供的 Spring 命名空间中包含<mongo:mapping-converter />元素，用来配置之前介绍过的 MappingMongoConverter 实例。它会在内部创建 MongoMappingContext 实例，并会试图从 ApplicationContext 中获取一个名为 mongoDbFactory 的 Spring Bean，当定义的 MongoDbFactory 实例名称不是 mongoDbFactory 时，可以使用命名空间元素的 db-factory-ref 属性来调整，如示例 6-23 所示。

### 示例 6-23 在 XML 中设置 MappingMongoConverter

```
<mongo:mapping-converter id="mongoConverter"
                           base-package="com.oreilly.springdata.mongodb" />
```

这个配置片段中，在 Spring 应用上下文中设置了 MappingMongoConverter 并将其 id 设为 mongoConverter。将 base-package 属性指向工程的基础包，以获取领域类并且在应用上下文启动时构建持久化元数据。

### 在 Spring JavaConfig 中

当使用 Spring JavaConfig 类时，为了简化配置，Spring Data MongoDB 包含一个配

置类，它在默认设置中声明了必要的基础组件，并提供回调的方法以便在必要时可对它们进行调整。为了模拟刚才介绍的设置，我们的配置类如示例 6-24 所示。

#### 示例 6-24 使用 JavaConfig 进行 MongoDB 的基本设置

```
@Configuration
class ApplicationConfig extends AbstractMongoConfiguration {

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

    @Override
    public Mongo mongo() throws Exception {

        Mongo mongo = new Mongo();
        mongo.setWriteConcern(WriteConcern.SAFE);
        return mongo;
    }

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

前两个方法是超类限制必须要实现的，因为它创建了用来访问 MongoDB 的 SimpleMongo-DbFactory 类。除了必须实现的方法，我们还重写了 getMappingBasePackage() 方法用来指定映射模块扫描的包及其子包，以查找带有@Document 注解的类。但这并不是绝对必须的，因为在默认情况下，映射模块会扫描配置类所在的包，我们在此列出只是为了说明如何进行重新配置。

### 6.3.3 索引

MongoDB 和关系型数据库一样支持索引。它支持以编码的方式或映射注解的方式设置索引。我们通常会通过 Email 地址来检索 Customer，因此希望对它们建立索引。所以，在 Customer 类的 emailAddre 属性上添加@Index 注解，如示例 6-25 所示。

#### 示例 6-25 为 Customer 的 emailAddress 配置索引

```
@Document
public class Customer extends AbstractDocument {

    @Index(unique = true)
    private EmailAddress emailAddress;

    ...
}
```

为了避免在系统中出现重复的 Email 地址，我们将 unique 标志设置为 true。这样 MongoDB 会避免在新建或者更新 Customer 时，使用与其他 Customer 相同的 Email 地址。我们在领域类中使用@CompundIndex 注解来定义含多个属性的索引。



当类被 `MappingContext` 识别时，索引元数据也会被发现。由于信息跟集合存储在一起，保存在这个类所对应的持久化集合中，如果移除了集合，索引信息也会丢失。若要避免这件事发生，请在集合中移除所有文档，而不是移除集合。

可以在样例应用程序的 `CustomerRepositoryIntegrationTests` 类中找到领域对象中被拒绝持久化的用例。请注意，我们预期将会抛出 `DuplicateKeyException` 异常，因为所持久化的 `customer` 中，它持有的 Email 地址已经被其他的 `customer` 占用了。

### 6.3.4 自定义转换

映射模块提供了一个通用的方法，实现 Java 对象和 MongoDB DBObjects 的互相转换。然而，有时候会想手动实现指定类型的转换，例如之前曾介绍过值对象获取 Email 地址时会产生内嵌文档，我们想避免这样以保持文档结构的简洁，尤其是想把 `EmailAddress` 值直接内联到 `customer` 对象中时。为了回顾这个场景，我们从示例 6-26 开始。

#### 示例 6-26 Customer 类和它的 DBObject 表现形式

```
@Document
public class Customer extends AbstractDocument {

    private String firstname, lastname;

    @Field("email")
    private EmailAddress emailAddress;

    ...
}

{
    firstname : "Dave",
    lastname : "Matthews",
    email : { email : "dave@dmdband.com" }, ...
}
```

我们最终想要获得更简单的文档形式，如示例 6-27 所示。

#### 示例 6-27 想要得到的 Customer 文档结构

```
{ firstname : "Dave",
  lastname : "Matthews",
  email : "dave@dmdband.com", ... }
```

### 实现自定义的转换器

映射子系统允许手动实现对象到文档的相互转换，这是借助 Spring 转换服务的 `Converter` 抽象做到的。因为我们想要将复杂的对象变成普通的 `String`，本质上需要实现一个写入的 `Converter<EmailAddress, String>`，还需要以 `String` 类型构建 `EmailAddress`（即 `Converter<String, EmailAddress>`），如示例 6-28 所示。

### 示例 6-28 自定义 EmailAddress 转换器

```
@Component
class EmailAddressToStringConverter implements Converter<EmailAddress, String> {

    public String convert(EmailAddress source) {
        return source == null ? null : source.value;
    }
}

@Component
class StringToEmailAddressConverter implements Converter<String, EmailAddress> {

    public EmailAddress convert(String source) {
        return StringUtils.hasText(source) ? new EmailAddress(source) : null;
    }
}
```

### 注册自定义转换器

刚刚实现的转换器必须注册到映射模块。不管使用 Spring XML 命名空间还是 Spring JavaConfig 所提供的配置基类，注册过程都非常简单。在 XML 中，只要在 <mongo:mapping-converter> 内定义一个子元素，然后配置它的 base-package 属性来启动组件扫描即可，如示例 6-29 所示。

### 示例 6-29 使用 XML 命名空间注册自定义转换器

```
<mongo:mapping-converter id="mongoConverter" base-package="com.oreilly.springdata.mongodb">
    <mongo:custom-converters base-package="com.oreilly.springdata.mongodb" />
</mongo:mapping-converter>
```

在 JavaConfig 中，配置基类提供了回调方法来返回 CustomConversions 实例。这个类会封装已实现的 Converter 实例，稍后会查找到它，并用它来适当配置 MappingContext 和 MongoConverter，最终由 ConversionService 来执行转换。在示例 6-30 中，启用组件扫描来访问 Converter 实例并将它们自动装配到配置类中，最终将它们封装到 CustomConversions 实例中。

### 示例 6-30 使用 Spring JavaConfig 注册自定义转换器

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

    @Autowired
    private List<Converter<?, ?>> converters;

    @Override
    public CustomConversions customConversions() {
        return new CustomConversions(converters);
    }
}
```

如果如示例 6-22 所示，则从应用上下文中获取 MappingMongoConverter 并且触发转换的过程，所输出的结果如示例 6-31 所示。

### 示例 6-31 应用 EmailAddress 自定义转换器所形成的文档结构

```
{ firstname : "Dave",
  lastname : "Matthews",
  email : "dave@dmdband.com", ... }
```

## 6.4 MongoTemplate

现在基础设施已经就绪，我们也掌握了对象映射与配置的方法，接下来继续探讨与存储进行交互的 API。正如其他 Spring Data 模块一样，核心 API 是 MongoOperations 接口，这个接口的实现是 MongoTemplate。在 Spring 中，模板实现有两个主要的目的：资源管理和异常转换。这就意味着 MongoTemplate 会通过已配置的 MongoDbFactory 取得连接，并且在与存储交互完成或者发生异常时正确地完成清理工作。由 MongoDB 抛出的异常会被显式地转换成 Spring DataAccessException 异常等级中的类，这样就能够避免用户端必须要了解所使用的持久化技术。

我们继续以 CustomerRepository 接口的存储实现来说明 API 的使用方法（见示例 6-32）。它名为 MongoDbCustomerRepository，位于 com.oreilly.springdata.mongodb.core 包中。

### 示例 6-32 MongoDbCustomerRepository 实现

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

...
@Repository
@Profile("mongodb")
class MongoDbCustomerRepository implements CustomerRepository {

    private final MongoOperations operations;

    @Autowired
    public MongoDbCustomerRepository(MongoOperations operations) {
        Assert.notNull(operations);
        this.operations = operations;
    }

    @Override
    public Customer findOne(Long id) {

        Query query = query(where("id").is(id));
        return operations.findOne(query, Customer.class);
    }

    @Override
    public Customer save(Customer customer) {

        operations.save(customer);
        return customer;
    }
}
```

```
    @Override  
    public Customer findByEmailAddress(EmailAddress emailAddress) {  
        Query query = query(where("emailAddress").is(emailAddress));  
        return operations.findOne(query, Customer.class);  
    }  
}
```

这里有一个使用@Repository 注解的标准 Spring 组件，这样在扫描类路径时，这个类就能够被发现。增加@Profile 注解以确保只有在所配置的 Spring Profile 启动时，它才会被激活。这可以避免该类侵入到默认的 Bean 设置中，稍后介绍 MongoDB 的 Spring Data Repository 时将会用到。

这个类唯一的依赖关系是 MongoOperations，它是 MongoTemplate 的接口，在应用上下文中已经将它设置好了（见 application-context.xml 或 ApplicationConfig 类[示例 6-12]）。MongoTemplate 提供两类可用方法。

- 通用、高层次的方法只用一行语句就能执行常见的操作。它包含基本的方法如 findOne(...), findAll(...), save(...) 和 delete(...); 还包括 MongoDB 特有的方法，如 updateFirst(...), updateMulti(...) 和 upsert(...), 除此之外，还有 map-reduce 和地理空间相关的操作，如 mapReduce(...) 和 geoNear(...). 以上所有方法会自动应用 6.3 小节“映射模块”所介绍的对象-存储映射。
- 低层次、回调驱动的方法允许在高层次操作的功能无法满足需求时与 MongoDB 驱动 API 进行交互。这些方法以 execute 开头并且使用 Collection Callback(提供对 MongoDB DbCollection 的访问)、DbCallback(提供对 MongoDB DB 的访问) 或者 DocumentCallbackHandler(直接处理DBObject)。

使用高层次 API 的最简单例子就是 MongoDbCustomerRepository 的 save(...) 方法。只需使用 MongoOperations 接口的 save(...) 方法并将 Customer 传给它处理，它会将领域对象转换成 MongoDBDBObject 并使用 MongoDB 驱动 API 对其进行保存。

实现中的其他两个方法 findOne(...) 和 findByEmailAddress(...), 会使用 Spring Data MongoDB 提供的查询 API, 从而简化访问 MongoDB 文档时所构建的查询。query(...) 方法实际上是 Query 类中所定义的静态工厂方法，我们在这个类声明的顶部导入了 Query 类。where(...) 方法也是一样，但它源自 Criteria。正如你所见，定义一个查询非常简单。同样的，这里有一些需要注意的事情。

在 findByEmailAddress(...) 中，引用了 Customer 类的 emailAddress 属性。由于它已经通过@Filed 注解映射到 email 键，属性引用将会自动地转换到正确的字段引用。并且我们将普通的 EmailAddress 对象传递到条件中以建立相等的断言，在真正查询之前，映射模块会将它进行转换。其中也会包含为特定类型所注册的自定义转换器。因此，以DBObject 表示的查询如示例 6-33 所示。

### 示例 6-33 findByEmailAddress(...)经过转换后的查询对象

```
{ "email" : "dave@dmband.com" }
```

在 6.3.4 小节“实现自定义转换器”中，针对 `EmailAddress` 类我们使用了自定义的转换器，所以字段键被正确地转换成了 `Email`，而值对象也是正确的内联格式。

## 6.5 Mongo Repository

如刚才所介绍，`MongoOperations` 接口提供了一份相当完整的 API 用来手动实现存储类。然而，我们可以使用第 2 章介绍的 `Spring Data Repository` 抽象进一步简化这个过程。接下来我们将讨论示例工程的 `Repository` 接口定义，并看看如何调用 `Repository` 方法。

### 6.5.1 搭建基础设施

使用 `JavaConfig` 注解（示例 6-34）或者 XML 命名空间元素来启用 `Repository` 机制。

#### 示例 6-34 在 JavaConfig 中启用 Spring Data MongoDB Repository

```
@Configuration
@ComponentScan(basePackageClasses = AppConfig.class)
@EnableMongoRepositories
public class AppConfig extends AbstractMongoConfiguration {

    ...
}
```

在这个配置示例中，`@EnableMongoRepositories` 是最关键的部分。它将搭建 `Repository` 的基础设施，默认情况下会在配置类所在的包中扫描 `Repository` 接口。可以设置注解的 `basePackage` 或者 `basePackageClasses` 属性来对它进行调整。等效的 XML 配置也非常相似，但必须手动配置基础包（见示例 6-35）。

#### 示例 6-35 在 XML 中启用 Spring Data MongoDB repository

```
<mongo:repositories base-package="com.oreilly.springdata.mongodb" />
```

### 6.5.2 Repository 详解

针对示例应用程序中的每一个 `Repository`，都有一个测试类，这些测试类可以基于本地 `MongoDB` 实例运行。它们会与 `Repository` 交互并调用暴露的方法。将日志的级别设置为 `DEBUG`，就可以看到实际的发现 `Repository`、执行查询等过程。

我们从最基本的 `CustomerRepository` 开始，如示例 6-36 所示。

#### 示例 6-36 CustomerRepository 接口

```
public interface CustomerRepository extends Repository<Customer, Long> {
    Customer findOne(Long id);
```

```
Customer save(Customer customer);

Customer findByEmailAddress(EmailAddress emailAddress);
}
```

前面两个方法是 CURD 方法，并且会被路由到通用类 SimpleMongoRepository 上，这个类提供了所声明的方法。第 2 章已说明过它的通用机制，所以在这里真正有意思的方法是 findByEmailAddress(...). 由于没有定义手工查询，所以将会引入查询的衍生机制，它将解析方法并据此推导出查询。因为引用了 emailAddress 属性，衍生出的查询逻辑就会是 emailAddress = ?0。因此，基础设施会使用 Spring Data MongoDB 的查询 API 建立 Query 实例。它会依次把属性引用转换成对应的字段映射，因此我们最终所使用的查询就是 { email : ?0 }。在方法调用的时候，给定的参数将会和查询绑定在一起并最终执行。

下一个 Repository 是 ProductRepository，如示例 6-37 所示。

### 示例 6-37 ProductRepository 接口

```
public interface ProductRepository extends CrudRepository<Product, Long> {

    Page<Product> findByDescriptionContaining(String description, Pageable pageable);

    @Query("{ ?0 : ?1 }")
    List<Product> findByAttributes(String key, String value);
}
```

首先请注意 ProductRepository 继承了 CrudRepository 而不是普通的 Repository 接口。这样 CRUD 方法会引入到 Repository 定义中。因此，不需要手动的定义 findOne(...) 和 save(...) 方法。和 CustomerRepository.findByEmailAddress(...) 类似， findByDescription Containing(...) 再次使用了衍生查询机制。与前一个方法不同的地方在于它使用 Containing 关键字来限制断言，这会将传入的描述参数放到一个正则表达式之中，从而限制描述信息能够匹配给定的 String。

第二个要注意的是分页 API 的使用（见 2.2.3 小节“分页与排序”）。客户端可将 Pageable 传入到方法中，从而将返回结果限制为特定的某一页，Pageable 中包含了页码和每页大小的信息，如示例 6-38 所示。返回 Page 对象中包括了结果集以及一些元信息，如全部页数等。可以在 ProductRepositoryIntegration 的 lookupProductsBy Description() 方法中看到这个方法的使用示例。

### 示例 6-38 使用 findByDescriptionContaining(...) 方法

```
Pageable pageable = new PageRequest(0, 1, Direction.DESC, "name");
Page<Product> page = repository.findByDescriptionContaining("Apple", pageable);

assertThat(page.getContent(), hasSize(1));
assertThat(page, Matchers.<Product> hasItems(named("iPad")));
assertThat(page.isFirstPage(), is(true));
assertThat(page.isLastPage(), is(false));
assertThat(page.hasNextPage(), is(true));
```

首先，我们将 `PageRequest` 设为要请求第一页，分页大小为 1，结果集需要按照 `name` 降序排列。可以看到它返回的 `Page` 对象不仅包含了结果集，还包括所返回的页在全部页中的位置信息。

第二个方法（回到示例 6-37）在定义时使用`@Query` 注解来手动定义 MongoDB 查询。当查询衍生机制没有提供我们需要的功能或查询方法的名称非常冗长时，它是非常方便的。我们构建了一个通用的查询`{ ?0 : ?1 }`，将方法的第一个参数当作键而第二个参数作为值。现在客户端可以使用这个方法来查询含有特定属性的 `Product`（例如，基座连接器插头），如示例 6-39 所示。

### 示例 6-39 查询有基座连接器插头的 `Product`

```
List<Product> products = repository.findByAttributes("attributes.connector", "plug");  
assertThat(products, Matchers.<Product> hasItems(named("Dock")));
```

正如所预期的，这个方法调用返回了 `iPod` 基座。通过这样的方法，业务逻辑基于匹配的属性（连接器插头与插座）很容易就能实现产品的推荐功能。

最后但同样重要的是 `OrderRepository`（见示例 6-40）。我们已经讨论两个 `Repository` 接口了，最后一个并没有太大的差别。

### 示例 6-40 `OrderRepository` 接口

```
public interface OrderRepository extends PagingAndSortingRepository<Order, Long> {  
    List<Order> findByCustomer(Customer customer);  
}
```

这里所定义的方法只是简单地使用了查询衍生机制。与前面的 `Repository` 接口相比，不同的地方在于所扩展的接口，它继承了 `PagingAndSortingRepository`，这样不仅暴露了 CRUD 方法，还有许多其他的方法，例如 `findAll(Pageable pageable)`，借助这个方法能够对所有的 `Order` 进行分页操作。更多关于分页的通用 API，可以参见 2.2.3 小节“分页与排序”。

## 6.5.3 Mongo Querydsl 集成

我们已经知道如何将查询方法添加到 `Repository` 接口中了，接下来看看如何使用 `Querydsl` 为实体动态地创建断言，并通过 `Repository` 抽象来执行。第 3 章中提供了 `Querydsl` 是什么以及它如何工作的概要介绍。如果已经阅读过 4.4.2 小节“`Repository Querydsl 集成`”，就会看到即使我们查询完全不同的存储形式，API 的设置以及使用都是非常相似的。

为了生成元模型类，在 `pom.xml` 文件中配置了 `Querydsl` 的 Maven 插件，如示例 6-41 所示。

### 示例 6-41 为 MongoDB 设置 Querydsl APT 处理器

```
<plugin>
  <groupId>com.mysema.maven</groupId>
  <artifactId>maven-apt-plugin</artifactId>
  <version>1.0.5</version>
  <executions>
    <execution>
      <phase>generate-sources</phase>
      <goals>
        <goal>process</goal>
      </goals>
      <configuration>
        <outputDirectory>target/generated-sources</outputDirectory>
        <processor>...data.mongodb.repository.support.MongoAnnotationProcessor</processor>
      </configuration>
    </execution>
  </executions>
</plugin>
```

与 JPA 方式唯一不同的是，我们配置了 MongoAnnotationProcessor。它会设置 APT 处理器，它会检查 Spring Data MongoDB 映射模块提供的注解，以正确地产生元模型。除此之外，还提供了一个集成方式，让 Querydsl 在创建 MongoDB 查询时，能够识别映射以及可能已注册的自定义转换器。

有的断言是基于生成的元模型类所构建的，为了执行这些断言需要将相关的 API 包含进来，因此 ProductRepository 还继承了 QueryDslPredicateExecutor（见示例 6-42）。

### 示例 6-42 ProductRepository 接口继承了 QueryDslPredicateExecutor

```
public interface ProductRepository extends CrudRepository<Product, Long>,
                                         QueryDslPredicateExecutor<Product> { ... }
```

QuerydslProductRepositoryIntegrationTest 展现了如何使用断言。同样的，这段代码几乎百分之百复制了 JPA 中的代码。通过执行 product.name.eq("iPad") 的断言得到了一个 iPad 的引用，并根据产品的描述进行查找以验证断言执行所得到的结果，如示例 6-43 所示。

### 示例 6-43 使用 Querydsl 断言查询 Product

```
QProduct product = QProduct.product;

Product iPad = repository.findOne(product.name.eq("iPad"));
Predicate tablets = product.description.contains("tablet");

Iterable<Product> result = repository.findAll(tablets);
assertThat(result, is(Matchers.<Product> iterableWithSize(1)));
assertThat(result, hasItem(iPad));
```

# Neo4j：图数据库

## 7.1 图数据库

本章会介绍一种很有意思的 NoSQL 存储：图数据库 ([http://en.wikipedia.org/wiki/Graph\\_database](http://en.wikipedia.org/wiki/Graph_database))。很明显，图数据库是一种后关系型 (post-relational) 的数据存储，因为它们进一步演进了数据库的概念并保持了一些原有的特性。它们提供了一种有效的方式来存储半结构化 (semistructured) 但高度连通的数据，并且允许我们以很快的速度查询和遍历链接数据 (Linked Data)。

图数据库由节点 (node) 组成，节点通过带有标向和标记的关系 (即边) 进行连接，如图 7-1 所示。在属性图中，节点和关系都可以存有任何的键值 (key/value) 对。图由这些元素形成了复杂的网络，它鼓励我们按照真实世界原始数据结构进行领域建模。关系型的数据库依赖于固定的模式 (schema)，而图数据是无模式的 (schema-free) 并且数据结构上没有限制。关系可以很容易添加和修改，因为它们不是模式的一部分，而是实际数据的一部分。

可以将图数据库的高性能归功于将关联实体 (join) 的成本转移到了插入阶段——这是通过将关系作为数据结构的一等公民来实现的——从而允许我们以相同的时间从一个实体 (节点) 遍历到另一个节点。所以，不管数据集的规模如何，在图中进行指定遍历所消耗的时间是由遍历时跳跃的次数决定，而不是图中一共有多少节点和关系决定。在其他的数据库模型中，查找两个 (或更多) 实体之间关联的成本花费在每个查询之上。

借助于这一点，一个图就可以存储很多不同的领域实体，并在所有的实体之间建立有意思的关联。二次存取 (secondary access) 和索引结构也可以集成到图中，从而为一些节点或子图提供特殊的分组和访问路径。

鉴于图形数据库的特性，它们不依赖于聚集边界来管理原子性的操作，而是构建在 ACID (原子性、一致性、隔离性、持久性) 的数据存储之上，从而具备了行之有

效的事务保证。

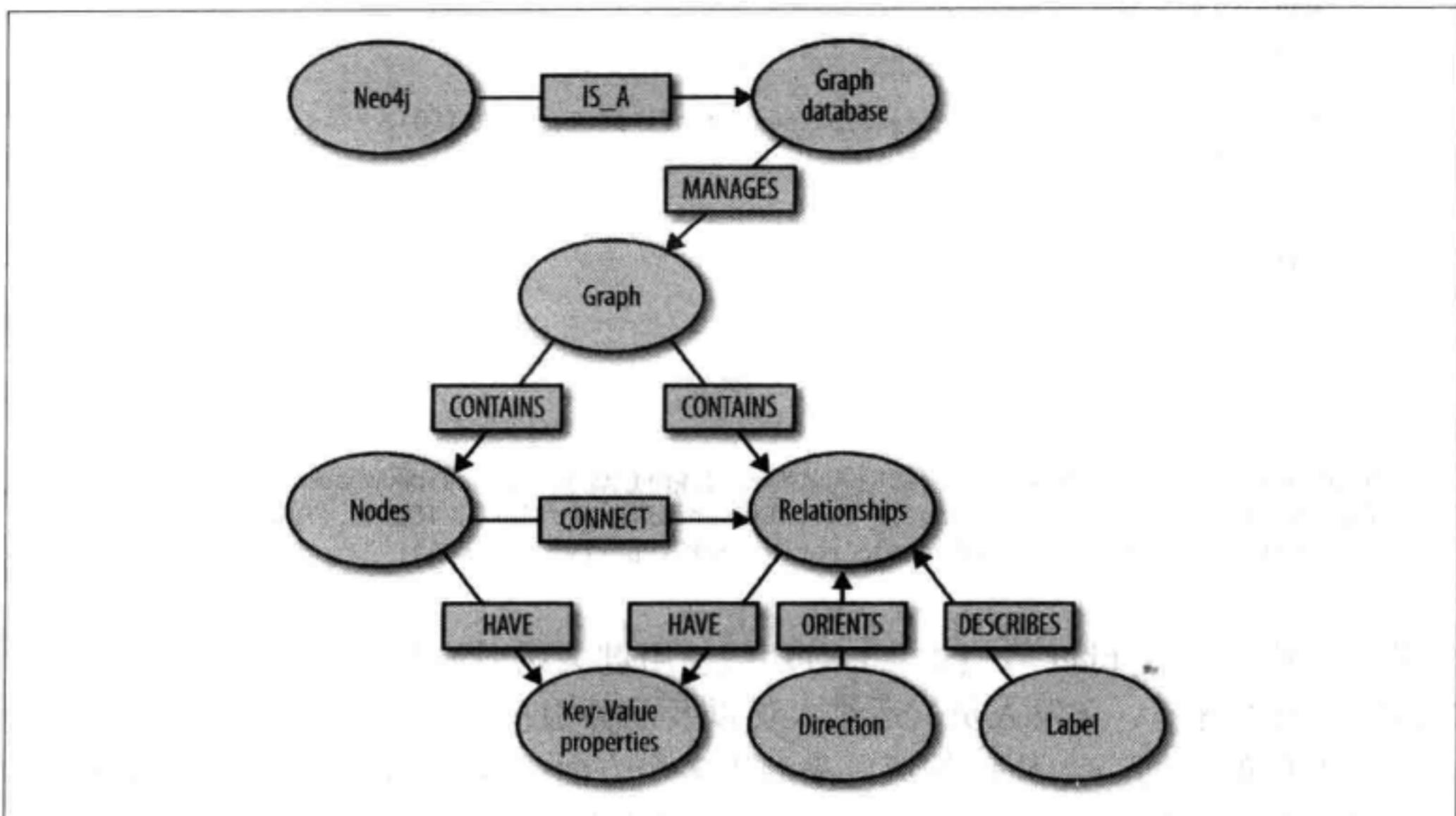


图 7-1 图数据库概览

## 7.2 Neo4j

Neo4j (<http://neo4j.org>) 是领先的属性数据库实现。它主要基于 Java 编写，使用了自定义的存储格式以及 Java 事务 API (Java Transaction API, JTA) 来提供 XA 事务。Java API 能够以面向对象的方式使用图中的节点和关系（如示例 7-1 所示），可以使用流畅的 API 来进行遍历。因为是图数据库，所以 Neo4j 内置提供了很多图算法，如最短路径、Dijkstra 以及 A\*。

Neo4j 集成了一个事务性的、可插拔的索引子系统，它默认使用了 Lucene (<http://lucene.apache.org>)。这个索引主要用来为遍历定位起始节点。它的第二个作用就是支持唯一实体的创建。要使用 Neo4j 嵌入式的 Java 数据库，需要在构建脚本中添加 org.neo4j:neo4j:<version> 依赖，这样就准备就绪了。示例 7-1 列出了在事务范围内创建带有属性的节点和关系的代码。随后展现了如何对其进行访问和读取。

### 示例 7-1 Neo4j 核心 API 演示

```
GraphDatabaseService gdb = new EmbeddedGraphDatabase("path/to/database");

Transaction tx=gdb.beginTx();
try {
    Node dave = gdb.createNode();
    dave.setProperty("email","dave@dmbar.com");
    gdb.index().forNodes("Customer").add
```

```

        (dave,"email",dave.getProperty("email"));

        Node iPad = gdb.createNode();
        iPad.setProperty("name","Apple iPad");

        Relationship rel=dave.createRelationshipTo(iPad,Types.RATED);
        rel.setProperty("stars",5);

        tx.success();
    } finally {
        tx.finish();
    }

    // to access the data

    Node dave = gdb.index().forNodes("Customer").get("email","david@dbband.com").getSingle();
    for (Relationship rating : dave.getRelationships(Direction.OUTGOING, Types.RATED)) {
        aggregate(rating.getEndNode(), rating.getProperty("stars"));
    }
}

```

借助于声明式的 Cypher 查询语言，对那些使用过关系型数据库 SQL 的人来说使用 Neo4j 会更加容易。开发人员、运维人员以及业务用户都能根据各种使用场景在图上运行专门的查询。Cypher 的灵感来源于多个方面：SQL、SparQL、ASCIIArt 以及函数式编程。核心理念在于用户描述图中要匹配的模式并提供起始节点。然后，数据库引擎在图中高效地匹配给定的模式，这样用户就可以定义复杂的查询了，如“为我找到一些顾客，他们的朋友最近都购买了类似的产品”。像其他的查询语言一样，它也支持过滤、分组以及分页。Cypher 允许进行简单的创建、删除、更新以及图构造。

示例 7-2 所示的 Cypher 的表达式展现了一个典型的用户场景。它首先根据索引查找顾客，然后通过其订单关系找到所订购的产品。查询语句会过滤掉较老的订单，随后会根据产品计算出他最大的 20 笔购买量。

## 示例 7-2 Cypher 示例表达式

```

START      customer=node:Customer(email = "dave@dbband.com")
MATCH      customer-[:ORDERED]->order-[item:LINEITEM]->product
WHERE      order.date > 20120101
RETURN     product.name, sum(item.amount) AS product
ORDER BY   products DESC
LIMIT      20

```

因为它基于 Java 编写，因此 Neo4j 可以很容易地嵌入任意的 Java 应用程序之中，此时它会引用一个单例的部署环境。但是很多的 Neo4j 部署会使用独立的 Neo4j 服务器，服务器提供了便利的 HTTP API 来进行交互以及全面的 Web 界面可用于管理、探查、可视化以及监控等目的。下载 Neo4j 服务器 (<http://docs.neo4j.org/chunked/milestone/server-installation.html>) 很方便，然后可以解压缩并直接启动。

可以在嵌入式的数据库之上 (<http://docs.neo4j.org/chunked/milestone/server-embedded.html>) 运行 Neo4j 服务器，这样可以便利地通过 Web 界面来进行检查和监控，如图 7-2

所示。

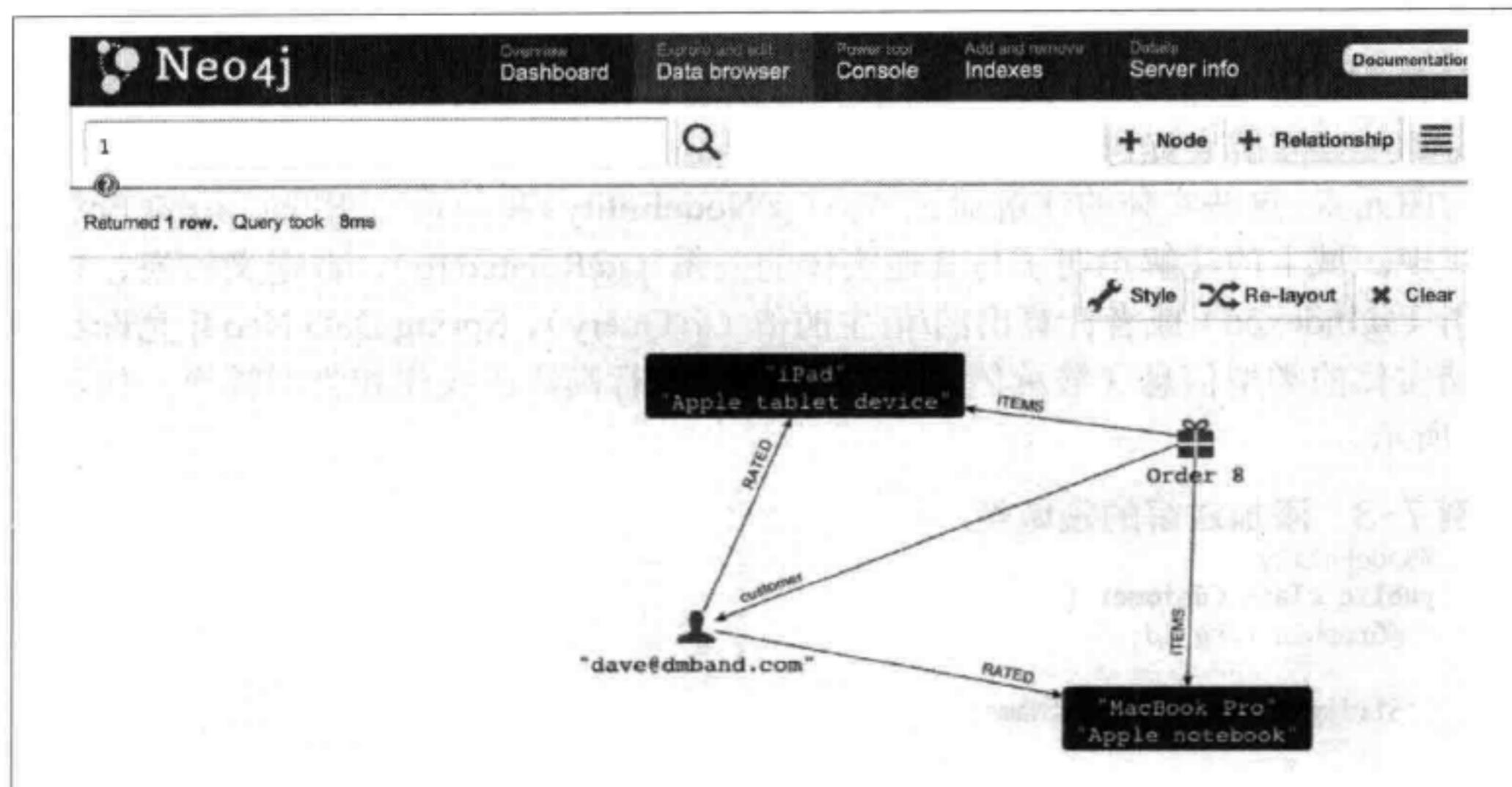


图 7-2 Neo4j 服务器的 Web 界面

在 Web 界面上，可以看到数据库的统计信息。在 Data browser 中，可以根据 ID 来查找节点、可以使用索引查找以及 Cypher 查询（单击蓝色的小问号可以得到语法帮助），并且能够使用右边按钮切换到图形可视化界面来探查图的结构，如图 7-2 所示。在 Console 中可以直接输入 Cypher 表达式甚至处理 HTTP 请求。Server Info 中列出了 JMX bean，尤其是在企业版中它会提供更多的信息。

作为开源的产品，Neo4j 拥有活跃的生态系统，包括贡献者、社区成员以及用户。Neo Technology 是 Neo4j 研发的发起者，他们对社区版本采用的是开源许可证（Open Source Licensing, GPL）并且会对企业版提供专业的支持，从而保证产品的持续研发。

要访问 Neo4j，会有多种可用的驱动，它们大多数是由社区维护的。对于嵌入式和服务器部署模式都有多种语言的类库。它们中有一些由 Neo4j 进行维护，Spring Data Neo4j 就是其中之一。

### 7.3 Spring Data Neo4j 概览

Spring Data Neo4j 是由 Rod Johnson 和 Emil Eifrem 创建的，它是整个 Spring Data 的发端。它由 VMware 和 Neo Technology 密切合作开发，为 Spring 开发人员提供了熟悉的方式与 Neo4j 进行交互。它的意图在于使用熟知的基于注解的编程模型实现与 Spring 框架生态系统的紧密集成。作为 Spring Data 项目的一部分，Spring Data

Neo4j 集成了 Spring Data Commons Repository（可参阅第 2 章）以及其他通用基础设施。

如同在 JPA 中一样，在 POJO（简单 Java 对象，Plain Old Java Object）实体及其域上添加一些注解会提供必要的元信息，这样 Spring Data Neo4j 就能将 Java 对象映射为图元素。这些实体的注解通过节点(@NodeEntity)和关系(@RelationshipEntity)来实现。域上的注解声明了与其他实体的关系(@RelatedTo)、自定义转换、自动标引(@Indexed)或者计算出的/衍生的值(@Query)。Spring Data Neo4j 允许我们存储实体的类型信息（继承体系），从而能够执行高级的操作和类型转换，如示例 7-3 所示。

### 示例 7-3 添加注解的领域类

```
@NodeEntity
public class Customer {
    @GraphId Long id;

    String firstName, lastName;

    @Indexed(unique = true)
    String emailAddress;

    @RelatedTo(type = "ADDRESS")
    Set<Address> addresses = new HashSet<Address>();
}
```

Spring Data Neo4j 的核心基础设施是 Neo4jTemplate，它提供了（类似于其他的模板实现）各种低层级的功能，这些功能对 Neo4j API 进行了封装以支持所匹配的领域对象。Spring Data Neo4j 的基础设施和 Repository 实现会使用 Neo4jTemplate 来完成其操作。类似于其他的 Spring Data 项目，Spring Data Neo4j 通过两个 XML 命名空间元素来进行配置——用来进行通用的搭建和 Repository 的配置。

为了让 Neo4j 适应个性化的使用场景，Spring Data Neo4j 支持 Neo4j 的嵌入式模式以及服务器部署模式，后者要通过 Neo4j 的 Java-REST 绑定来进行访问。两种不同的映射模式能够支持开发人员的个性化需求。在简单映射模式下，图中的数据会复制到领域对象中，然后会与图进行分离。在更为高级的映射模式下，会借助 AspectJ 为所绑定的领域对象提供实时（live）的、保持连接的图元素展现形式。

## 7.4 将领域建模为图

对于 Neo4j 这样的图数据库，在第 1 章中所描述的领域模型就很合适，如图 7-3 所示。为了进行一些更为高级的图操作，我们对其进行了进一步的规范化，并添加了额外的一些关系来对模型进行丰富。

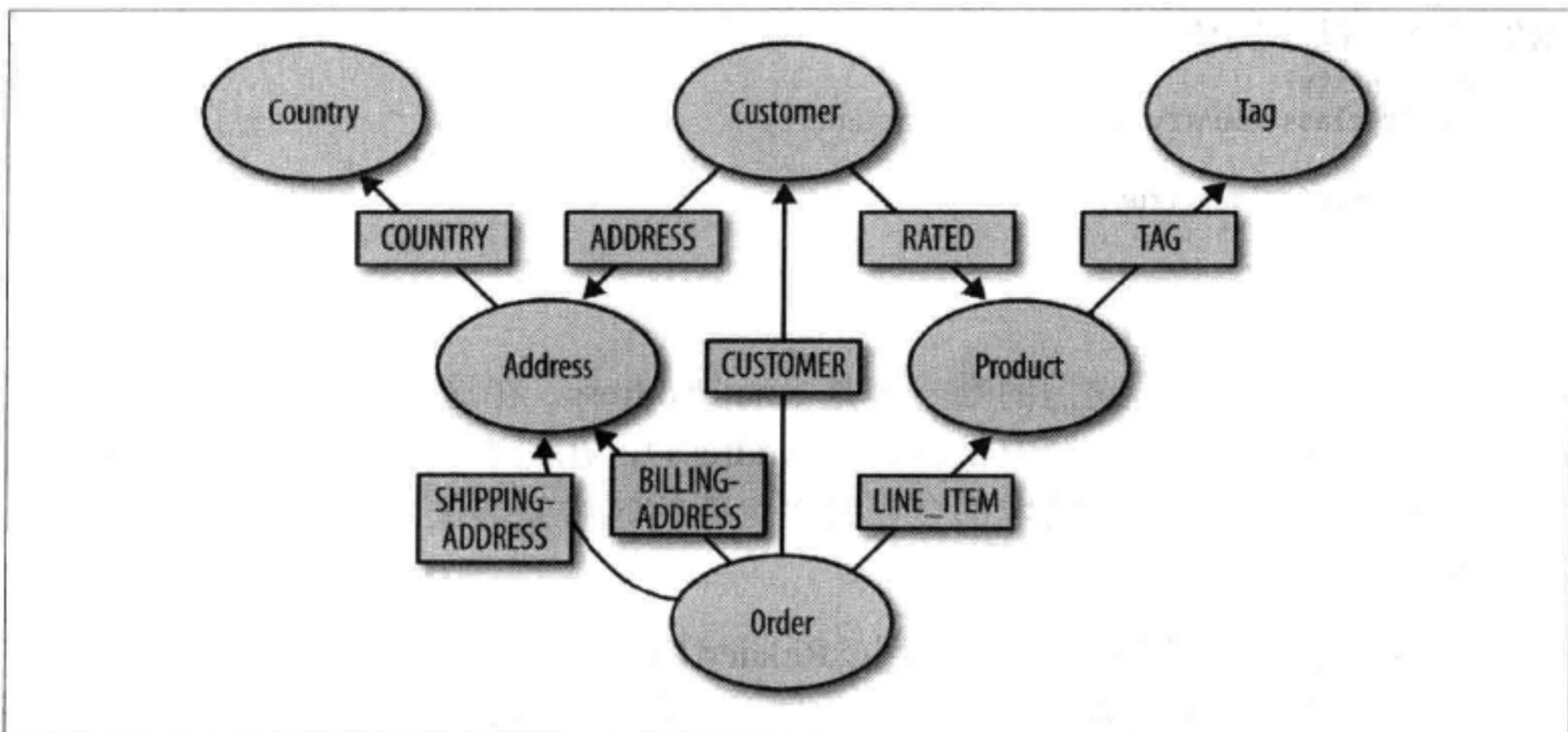


图 7-3 作为图的领域模型

这里所列出的示例代码并不完整，但包含了理解映射理念所必需的信息。参见示例源码库中的 Neo4j 工程可以了解全貌。

在示例 7-4 中，`AbstractEntity` 会作为超类，它包含了一个相同的 `id` 域（如前文所述，这个类具有`@GraphId` 注解，同时类中包含了 `equals(...)` 和 `hashCode()` 方法）。在简单的映射模式下需要有 `id` 的注解，因为这是保持节点或关系的 `id` 能够存储到实体的唯一方式。实体可以根据其 `id` 进行加载，这是通过 `Neo4jTemplate.findOne()` 方法实现的，在 `GraphRepository` 中有类似的方法。

#### 示例 7-4 基础领域类

```
public abstract class AbstractEntity {  
  
    @GraphId  
    private Long id;  
}
```

最简单的映射类只需使用`@NodeEntity`，以便 Spring Data Neo4j 的映射基础设施能够找到它。它可以包含任意数量的原始域，它们将被视为节点属性。原始类型会直接进行映射。借助所提供的 Spring 转换器（converter），能够将 Neo4j 所不支持的类型转换为等价的原始类型展现形式。对 `Enum` 和 `Date` 类型的转换器是在类库中内置的。

在 `Country` 中，有两个简单字符串类型的域，如示例 7-5 所示。域 `code` 代表一个唯一的“业务”键，并用`@Indexed(unique=true)` 进行标示，这样就会启用内置的唯一索引设施；这是通过 `Neo4jTemplate.getOrCreateNode()` 进行暴露的。在 `Neo4jTemplate` 中有多种方法来访问 Neo4j 的索引，我们可以使用 `Neo4jTemplate.lookup()` 根据索引键来查找实体。

### 示例 7-5 简单实体 Country

```
@NodeEntity
public class Country extends AbstractEntity {

    @Indexed(unique=true)
    String code;
    String name;
}
```

Customer 存储为节点；它们的唯一键是 emailAddress。在这里，我们第一次看到了对其他对象的引用（在本例中，也就是 Address），在图中代表着一个关系。所以，单个引用或集合形式引用的域都会导致在更新的时候创建关系，或者是在访问时进行导航使用。

如示例 7-6 所示，引用的域可以使用@RelatedTo，这个注解会说明它们是引用域，我们同时可以设置像关系类型（在本例中，就是“ADDRESS”）这样的属性。如果没有提供类型的话，它默认就是域的名字。默认情况下，关系指向被引用的对象（Direction.OUTGOING），可以通过注解来指明相反的方向；对于双向引用来讲，这尤为重要，这种情况下它应该被映射为单个关系。

### 示例 7-6 Customer 与其 Address 产生了关联

```
@NodeEntity
public class Customer extends AbstractEntity {

    private String firstName, lastName;

    @Indexed(unique = true)
    private String emailAddress;

    @RelatedTo(type = "ADDRESS")
    private Set<Address> addresses = new HashSet<Address>();
}
```

Address 同样也很简单。示例 7-7 展现了 country 引用域并不需要添加注解——对于输出型的关系，它会使用域的名字作为关系类型。Customer 与 Address 的关系在这个映射中没有表现出来，因为对于我们的用例来说没有这样的必要。

### 示例 7-7 Address 与 Country 之间的关联

```
@NodeEntity
public class Address extends AbstractEntity {

    private String street, city;
    private Country country;
}
```

Product 有一个唯一的名字并且展现了非原始类型域的使用方法；price 会使用 Spring 的转换器将其转化为原始类型的表现形式。对于自定义的类型（如值对象），可以在应用上下文中注册自己的转换器。

域 description 添加了索引，从而允许全文搜索。必须明确地给索引命名，因为它

使用了与默认精确索引不同的配置。现在可以调用诸如 `neo4jTemplate.lookup("search","description: Mac")` 这样的方式来查找产品，它会接收 Lucene 的查询字符串。

为了使用更为有趣的图操作，我们添加了 Tag 实体，并且在 Product 中与其进行关联。这些标签可以用来查找类似的产品、提供推荐或分析购买行为。

要处理实体中的动态属性（任意键值对形成的 map），在 Spring Data Neo4j 中有一个专门的类。我们决定不直接处理 map，因为它们自带的一些额外语义并不适合我们的上下文。目前，`DynamicProperties` 转化为节点的属性并使用加前缀的名字进行分割，如示例 7-8 所示。

### 示例 7-8 具有标签的产品并且包含动态属性

```
@NodeEntity
public class Product extends AbstractEntity {

    @Indexed(unique = true)
    private String name;
    @Indexed(indexType = IndexType.FULLTEXT, indexName = "search")
    private String description;
    private BigDecimal price;

    @RelatedTo
    private Set<Tag> tags = new HashSet<Tag> ();
    private DynamicProperties attributes = new PrefixedDynamicProperties("attributes");
}
```

Tag 的唯一不同之处在于其 `Object value` 属性。这个属性将会根据运行时的值转换为 Neo4j 可以存储的原始值。如示例 7-9 所示，`@GraphProperty` 注解允许进行一些个性化的存储（比如，所使用的属性名或指明图中的目标原始类型）。

### 示例 7-9 很简单的 Tag

```
@NodeEntity
public class Tag extends AbstractEntity {

    @Indexed(unique = true)
    String name;

    @GraphProperty
    Object value;
}
```

我们所遇到的第一个`@RelationshipEntity` 是之前领域模型中所不存在的新概念，但是在网站上它却是广为大家所熟知的。为了进行一些更为有趣的图操作，在 Customer 和 Product 之间添加了 Rating 关系。这个实体使用了`@RelationshipEntity` 注解来标识这一点。除了两个基本的域来持有 Rating 的 stars 和 comment 以外，还可以看到它包含了描述关系开始和结束的域，它们分别添加了对应的注解，如示例 7-10 所示。

### 示例 7-10 Customer 与 Product 之间的 Rating

```
@RelationshipEntity(type = "RATED")
public class Rating extends AbstractEntity {
    @StartNode Customer customer;
    @EndNode Product product;
    int stars;
    String comment;
}
```

关系实体可以作为普通的 POJO 类来进行创建，这个类需要提供其开始和结束端点并通过 Neo4jTemplate.save() 进行保存。在示例 7-11 中，通过 Order 展现了这些实体如何作为映射的一部分进行检索。在关于图操作更深入的讨论中（参见 7.7.3 小节“利用类似的兴趣（协同过滤）”，借助 Neo4jTemplate.query 的 Cypher 查询以及 Repository 的查找方法，我们将会看到如何使用这些关系。

Order 是目前为止连接最多的实体，它位于领域的中央。如示例 7-11 所示，在对 Customer 的关系中，对于双向关联且共用相同关系的情况下，我们展示了反向的 Direction.INCOMING。

对不同类型的地址（派送和账单）进行建模的最简单方式就是使用不同的关系类型——在本例中，只是依赖不同的域名即可。注意，一个地址对象/节点可以用在多个地方，如某位顾客的派送和账单地址，某个地址甚至可能被多个顾客共享（如一个家庭）。在实践之中，图会比关系型数据库更为标准化，这种对副本的移除会带来一些好处，这包括了存储的方面也包括了运行趣味查询的能力。

### 示例 7-11 Order，领域的核心

```
@NodeEntity
public class Order extends AbstractEntity {

    @RelatedTo(type = "ORDERED", direction = Direction.INCOMING)
    private Customer customer;

    @RelatedTo
    private Address billingAddress;

    @RelatedTo
    private Address shippingAddress;

    @Fetch
    @RelatedToVia
    private Set<LineItem> lineItems = new HashSet<LineItem>();
}
```

LineItem 并没有建模为节点，而是作为 Order 和 Product 之间的关系。LineItem 并没有自己的标识，只有当它的两个端点存在的时候它才能存在，在这里通过 order 和 product 域来进行引用。在这个模型中，LineItem 只包含 quantity 属性，但是在其他的用户场景中，它还可以包含其他的属性。

Order 和 LineItem 中，有意思的地方在于 @RelatedToVia 和 @Fetch 注解，我们对其稍作介绍。域 lineItems 上的注解类似于 @RelatedTo，它只是用在关系实体的引用上。我们可以指明自定义的的类型和方向。所指定的类型将会覆盖 @RelationshipEntity 所提供的类型，如示例 7-12 所示。

### 示例 7-12 LineItem 是一个关系

```
@RelationshipEntity(type = "ITEMS")
public class LineItem extends AbstractEntity {

    @StartNode private Order order;

    @Fetch
    @EndNode
    private Product product;
    private int amount;
}
```

现在我们该看一下图映射很重要的一个方面：抓取声明（fetch declaration）。从 JPA 可以知道，这是很棘手的问题。如今在 Spring Data Neo4j 中，通过默认不抓取关联的实体会让事情变得很简单。

因为简单映射模式下需要将图中的数据复制到对象之中，所以对于抓取的深度要特别小心；否则的话，稍有不慎就会将整个图加载在内存之中，因为图结构经常是环形的。这就是为何默认的情况下会以很浅层级的方式加载关联的实体。@Fetch 注解用来声明对应的域马上进行完整的加载。其实可以事后通过 template.fetch(entity.field) 加载它们。这可以用于单关系（一对一）和多关系（一对多）的域。

在 Order 之中，LineItems 默认会被抓取出来，因为当加载 Order 时，它们在大多数场景下都是很重要的。对于 LineItem 来说，Product 也是立即加载的，这样它就直接可用了。根据用例，建模的方式可能会稍有不同。现在我们已经创建了领域类，该把它们的数据存储到图中了。

## 7.5 使用 Spring Data Neo4j 持久化领域对象

在将领域对象存储到图中之前，首先需要搭建工程。除了那些常用的 Spring 依赖，还需要添加 org.springframework.data:spring-data-neo4j:2.1.0.RELEASE（对于简单映射）或 org.springframework.data:spring-data-neo4j-aspects:2.1.0.RELEASE（对于基于 AspectJ 的高级映射，参见 7.9 小节）的依赖。Neo4j 会自动引入进来（简单起见，假设是嵌入式的 Neo4j 部署）。

最少的 Spring 配置只是简单的命名空间设置，它同时会建立图数据库，如示例 7-13 所示。

### 示例 7-13 Spring 的搭建配置

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:neo4j="http://www.springframework.org/schema/data/neo4j"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/data/neo4j
        http://www.springframework.org/schema/data/neo4j/spring-neo4j.xsd">

    <neo4j:config storeDirectory="target/graph.db" />
    <neo4j:repositories base-package="com.oreilly.springdata.neo4j" />

</beans>
```

如示例 7-14 所示，也可以传递 graphDatabaseService 实例到 neo4j:config 中，这样的话就可以配置图数据库的缓存、内存使用以及升级策略了。在这里甚至可以使用基于内存的 ImpermanentGraphDatabase 来进行测试。

### 示例 7-14 传递 graphDatabaseService 到配置之中

```
<neo4j:config graphDatabaseService="graphDatabaseService" />

<bean id="graphDatabaseService" class="org.neo4j.test.ImpermanentGraphDatabase" />

<!-- or -->
<bean id="graphDatabaseService" class="org.neo4j.kernel.EmbeddedGraphDatabase"
    destroy-method="shutdown">
    <constructor-arg value="target/graph.db" />
    <constructor-arg> <!-- passing configuration properties -->
        <map>
            <entry key="allow_store_upgrade" value="true" />
        </map>
    </constructor-arg>
</bean>
```

在定义了领域对象并将环境搭建起来之后，可以很容易地生成用于阐述用例的示例数据集（如示例 7-15 和图 7-4 所示）。用于阐述这个用例的领域类、数据集生成以及集成测试都能在本书的 GitHub 仓库中找到（更多信息可以参考 1.4 小节“示例代码”了解）。为了导入数据，可以填充领域类并使用 template.save(entity) 方法，这个方法会与图中已有的元素进行合并或者创建新的元素。这取决于映射 ID 以及可能存在的唯一域声明，它们可以用来识别图中我们要进行合并的已有实体。

### 示例 7-15 使用示例数据集填充图

```
Customer dave = template.save(new Customer("Dave", "Matthews", "dave@dmdband.com"));
template.save(new Customer("Carter", "Beauford", "carter@dmdband.com"));
template.save(new Customer("Boyd", "Tinsley", "boyd@dmdband.com"));

Country usa = template.save(new Country("US", "United States"));
template.save(new Address("27 Broadway", "New York", usa));
```

```

Product iPad = template.save(new Product("iPad", "Apple tablet device").withPrice(499));
Product mbp = template.save(new Product("MacBook Pro", "Apple notebook").withPrice(1299));

template.save(new Order(dave).withItem(iPad,2).withItem(mbp,1));

```

这里显示的实体使用了一些便利的方法进行构造，从而提供了可读性更好的搭建过程，如图 7-4 所示。

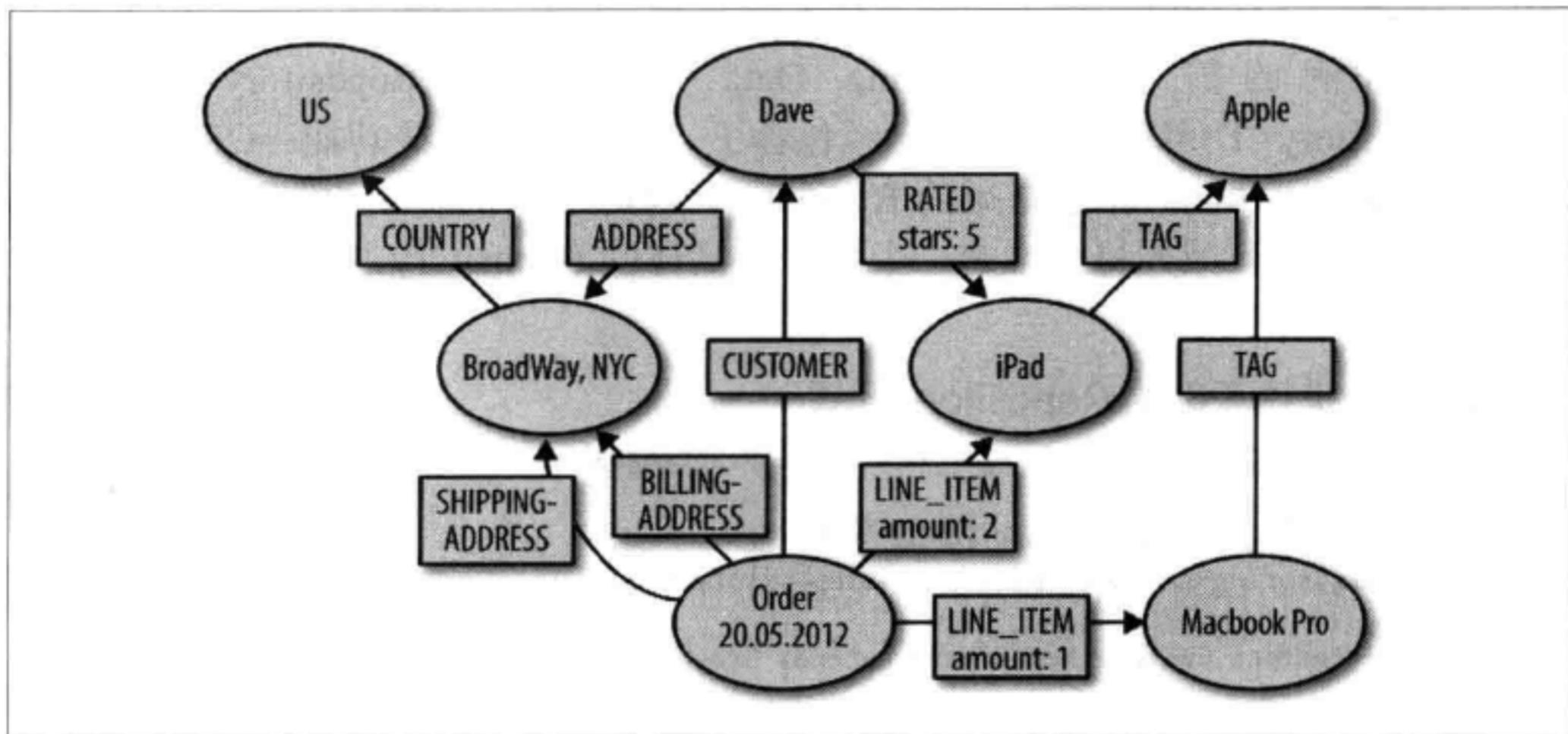


图 7-4 导入领域数据的图

### 7.5.1 Neo4jTemplate

Neo4jTemplate 类似于其他的 Spring 模板：在低层级的 API 之上构建了便利化 API，在这里低层级的 API 指的就是 Neo4j API。它添加了一些常见的便捷功能，如事务处理和异常转换，但更为重要的是对领域实体的自动化映射。Neo4jTemplate 也用于 Spring Data Neo4j 的其他基础设施之中。对其进行搭建时，需要在应用上下文中添加<neo4j:config>或者创建实例，它要传递 Neo4j 的 GraphDatabaseService 进来（这个类可以作为 Spring Bean 来进行获取，如果想直接访问 Neo4j API，可以把它注入到代码之中）。

创建实体、节点和关系并且按照 id 对其进行查找和移除构成了基本的操作（`save()`、`getOrCreateNode()`、`findOne()`、`getNode()`、`getRelationshipsBetween()`等）。其他的一些大多数机制都是以更为高级的方式在图中进行有趣的查找——使用 `lookup()`方法进行索引查找、使用 `query()`执行 Cypher 语句以及通过 `traverse()`运行遍历。Neo4jTemplate 提供了 `load()`方法将节点转换为实体或者将实体通过 `projectTo()`方法转换为不同的类型（参见 7.7.1 小节的“单个节点的多重角色”）。懒加载的实体可以通过 `fetch()`方法进行完整的加载。

通过 Neo4jTemplate 这个类就可以实现大多数你所希望的 Spring Data Neo4j 功能，但是对 Repository 的支持能够让你以更为便利的方式执行很多操作。

## 7.6 组合发挥图和 Repository 的威力

一切准备就绪之后，就可以看看 Repository 如何与 Spring Data Neo4j 进行集成，以及怎样以“图”的方式使用它们。

Spring Data Commons Repository（参见第 2 章）能够很容易让持久化访问相关的代码（或者说没有代码）位于同一个地方并且允许我们尽可能少地编写代码来满足特定用例的需求。在 Spring Data Neo4j 中，Repository 来源于 GraphRepository<T> 基础接口，它已经包含了一些常用的所需功能：CRUD 操作以及索引和遍历功能。搭建基础的 Repository 需要另一行命名空间配置，如示例 7-16 所示。每个领域类将会绑定一个独立和具体的 Repository 接口，如示例 7-17 所示。

### 示例 7-16 搭建基础的 Repository 配置

```
<neo4j:repositories base-package="com.oreilly.springdata.neo4j" />
```

### 示例 7-17 基础的 Repository 接口声明

```
import org.springframework.data.neo4j.repository.GraphRepository;  
  
public interface CustomerRepository extends GraphRepository<Customer> {  
    Customer findByEmailAddress(String emailAddress);  
}
```

Spring Data Neo4j Repository 提供了对 @Query 注解和衍生查找方法的支持，这是通过 Cypher 表达式实现的。为了理解这种映射是如何工作的，需要了解 Cypher 的表示语法，这在下面的边栏“Cypher 查询语言”中进行了讲解。

#### Cypher 查询语言

Neo4j 自带了简洁和面向对象的 Java API，并且可以使用众多的 JVM（Java 虚拟机）语言以及大量针对 Neo4j 服务器的驱动。但是通常来讲，更好的数据操作声明表达方式是以“是什么（what）”来进行，而不是指明“怎样做（how）”。

这就是开发 Cypher 查询语言 (<http://neo4j.org/resources/cypher>) 的原因。它基于图中的模式匹配，模式绑定到特定的节点或关系并允许对结果进行进一步的过滤和分页。Cypher 有数据操作的特性，这样就可以修改图。Cypher 的查询部分支持链式（piped）连接，从而支持更为高级和强大的图操作。

每个 Cypher 查询可以包含以下几个部分。

**START**

定义标示符（identifier）、绑定节点和关系，这可以通过索引或 ID 进行查找。

```
START user=node:customers(name="dave@...")
```

## MATCH

使用 ASCII-ART 描述在图中要查找的模式。模式要绑定到标示符上并且会定义新的标示符。在查询执行时所找到的子图会产生结果。

```
MATCH user-[rating:RATED]->product
```

## WHERE

使用布尔表达式对结果进行过滤，使用点号来访问属性、函数、集合断言和函数（collection predicates and functions）以及算术运算符等。

```
WHERE user.name = "Dave" AND ANY(color in product.colors : color = 'red')  
OR rating.stars > 3
```

## SKIP LIMIT

使用偏移量和每页的数量对结果进行分页。

```
SKIP 20 LIMIT 10
```

## RETURN

声明查询要返回什么。如果使用了聚集函数，那么所有非聚集的值将会用作分组值。

```
return user.name, AVG(rating.stars) AS WEIGHT, product
```

## ORDER BY

根据属性或其他的表达式进行排序。

```
ORDER BY user.name ASC, count(*) DESC
```

## UPDATES

Cypher 还有更多的东西。借助于 CREATE [UNIQUE]、SET 以及 DELETE，图可以在运行期进行修改。WITH 和 FOREACH 允许使用更为高级的查询结构。

## PARAMETERS

Cypher 可以传递进来参数的 Map，它可以通过键（或位置）进行引用。

```
start n=node({nodeId}) where n.name=~{0} return n
```

Cypher 返回的结果在内部是列表式的结构，这一点上就像是 JDBC ResultSet。列名会作为行值的键。

存在针对 Cypher 的 Java DSL，这样就不用借助缺乏语义的字符串来进行查询了，而是可以使用类型安全的 API 来构建 Cypher 查询。可以基于所生成的领域对象，使用 Querydsl（参见第 3 章）构建用于过滤和索引查询的表达式。借助于已有的 JDBC 驱动（<https://github.com/rickardoberg/neo4j-jdbc>），Cypher 查询能够很容易地集成到现存的 Java（Spring）应用和其他工具之中。

## 7.6.1 基本的图 Repository 操作

Repository 所提供的基本操作模仿了 Neo4jTemplate 的操作，只不过要绑定到声明 Repository 的领域类。所以，`findOne(...)`、`save(...)`、`delete(...)`和 `findAll(...)`等方法接收和返回领域类的实例。

Spring Data Neo4j 在图中存储了映射实体的类型（层级）信息。为了实现这一点，它会采用多个策略中的某一个，默认情况是基于索引的存储。这种类型的信息可以用于所有的 Repository 和模板方法，它们会操作某种类型的所有实例，并且会校验请求类型与实际存储的类型。

Repository 的更新方法默认就是事务性的，所以没有必要在它们上面声明事务。但是对于领域用例，比较明智的做法还是要这样做的，因为通常在一个业务事务中会封装不止一个数据库操作（这要使用 Neo4j 所提供的对 JtaTransactionManager 的支持）。

对于索引操作，IndexRepository 中有一些特定的方法，如 `findAllByPropertyValue()`、`findAllByQuery()`和 `findAllByRange()`，它们会直接映射到 Neo4j 的底层索引基础设施，但是要考虑 Repository 领域类和既有的索引相关的注解。TraversalRepository 中暴露了类似的方法，它的 `findAllByTraversal()`方法允许直接访问 Neo4j 强大的图遍历机制。其他的 Repository 接口提供了空间查询（spatial query）和 Cypher-DSL 集成的方法。

## 7.6.2 衍生和基于注解的查找方法

除了之前所讨论的基本操作之外，Spring Data Neo4j Repository 也支持借助 Cypher 查询语言自定义查找方法。在基于注解和衍生的查找方法中，额外的 `Pageable` 和 `Sort` 方法参数会在查询执行的时候自动考虑进来。它们会被转换成合适的 `ORDER BY`、`SKIP` 和 `LIMIT` 声明。

### 基于注解的查找方法

查找方法可以直接使用 Cypher，只需添加`@Query`注解并包含查询字符串就可以了，如示例 7-18 所示。方法的参数也会作为参数传递给 Cypher 查询，这可以通过参数的位置也可以通过`@Parameter`注解所声明的名称来完成，所以可以在查询字符串中使用`{index}`或`{name}`。

#### 示例 7-18 在 Repository 查询方法上添加 Cypher 查询注解

```
public interface OrderRepository extends GraphRepository<Order> {  
  
    @Query(" START c=node({0}) " +  
           " MATCH c-[:ORDERED]->order-[item:LINE_ITEM]->product " +  
           " WITH order, SUM (product.price * item.amount) AS value " +  
           " WHERE value > {orderValue} " +  
           " RETURN order")  
}
```

```
    Collection<Order> findOrdersWithMinimumValue(Customer customer,
                                                @Parameter("orderValue") int value);
}
```

## 结果处理

查找方法的返回类型可以是 `Iterable<T>`，在这种情况下查询的求值是延迟进行的，也可以是其他的接口：`Collection<T>`、`List<T>`、`Set<T>` 以及 `Page<T>`。`T` 是查询的结果类型，它可以是映射的领域实体（当返回节点或关系时）也可以是原始类型。关于查询结果的映射，这里提供了很简单的基于接口的支持。要映射结果，首先要创建带有 `@MapResult` 注解的接口。在这个接口中声明检索每列值的方法。对这些方法分别添加上 `@ResultColumn("columnName")` 注解，如示例 7-19 所示。

### 示例 7-19 定义 MapResult 并将其用于接口方法中

```
@MapResult
interface RatedProduct {
    @ResultColumn("product")
    Product getProduct();

    @ResultColumn("rating")
    Float getRating();

    @ResultColumn("count")
    int getCount();
}

public interface ProductRepository extends GraphRepository<Product> {

    @Query(" START tag=node({0}) " +
           " MATCH tag-[ :TAG ] -> product <- [ rating:RATED ] - () " +
           " RETURN product, avg(rating.stars) AS rating, count(*) as count " +
           " ORDER BY rating DESC")
    Page<RatedProduct> getTopRatedProductsForTag(Tag tag, Pageable page);
}
```

为了避免针对不同的粒度、结果类型和容器类产生过多的查询方法，Spring Data Neo4j 为结果处理提供了 API。这些 API 涵盖了自动化和编码形式的值转换。结果处理 API 的核心在于将可迭代 (iterable) 的结果转换为不同的类型，在这里会使用到所配置的或给定的 `ResultConverter`，这要取决于结果大小的粒度，可能还要与目标容器的类型有关，如示例 7-20 所示。

### 示例 7-20 结果处理 API

```
public interface ProductRepository extends GraphRepository<Product> {

    Result<Map<String, Object>> findByName(String name);

    Result<Map<String, Object>> result = repository.findByName("mac");

    // return a single node (or null if nothing found)
    Node n = result.to(Node.class).singleOrNull();
    Page<Product> page = result.to(Product.class).as(Page.class);
```

```
Iterable<String> names = result.to(String.class,
    new ResultConverter<Map<String, Object>, String>>() {
    public String convert(Map<String, Object> row) {
        return (String) ((Node) row.get("name")).getProperty("name");
    }
});
```

## 衍生的查找方法

如第 2 章中所述，衍生查找方法（参见 2.2.2 小节“衍生查询”）是真正与众不同的方式。它们会使用目标领域实体的既有映射信息并且会对查找方法的名字进行智能的解析，从而生成能够获取所需信息的查询。

衍生查找方法——如 `ProductRepository.findByNameAndColorAndTagName (name, color, tagName)`——以 `find(By)` 或 `get(By)` 开头，接下来会包含一系列的属性表达式。每个属性表达式指向当前类型的属性名，或者指向其他关联领域实体类型及其一个属性。这些属性必须存在于实体之中。如果不是这样的话，在 `ApplicationContext` 启动时，`Repository` 会出现创建错误。

对于所有合法的查找方法，`Repository` 会通过领域实体的映射信息构建适当的查询。在查询构建的时候，有很多方面会考虑起来，如图类型的展现、索引信息、域类型、关系类型以及方向。这也是进行适当转义的地方。

因此，示例 7-20 将会被转换成的查询如示例 7-21 所示。

### 示例 7-21 生成的衍生查询

```
@NodeEntity
class Product {

    @Indexed
    String name;
    int price;

    @RelatedTo(type = "TAG")
    Set<Tag> tags;
}

@NoArgsConstructor
class Tag {

    @Indexed
    String name;
}

public interface ProductRepository extends GraphRepository<Product> {

    List<Product> findByNameAndPriceGreaterThanOrTagsName(String name, int price,
        String tagName);
}

// Generated query
```

```
START product = node:Product(name = {0}), productTags = node:Tag(name = {3})
MATCH product-[:TAG]->productTags
WHERE product.price > {1}
RETURN product
```

这个示例展现了对已添加索引的属性如何基于索引进行查找以及简单的属性比较。如果方法名要引用其他关联的实体，那么查询构造器会在生成的查询中检查这些实体包含了什么。构造器还将关系的方向和关系类型添加到实体中。如果路径中有更多的属性，要重复相同的行为。

属性对比所支持的关键字包括：

- 算数比如 GreaterThan、Equals 或 NotEquals；
- 用于检查空值（或不存在）的 IsNull 和 IsNotNull；
- 用于字符串对比的 Contains、StartsWith、EndsWith 和 Like；
- 用于对表达式求反的 Not 前缀；
- 用于匹配正则表达式的 Regexp。

对于很多典型的查询用例，在 Repository 接口中编写并使用衍生查找声明是很容易的。仅对那些关联较多的查询才需要基于注解的查询、遍历描述（Traversal Description）或按需手动遍历。

## 7.7 示例领域模型中的高级图用例

除了能够方便地将现实世界中互相关联的数据映射到图中，图形化数据模型还允许你以很有意思的方式使用数据。通过关注于领域中关系的值，可以找到新的视角和答案，它们都隐藏在关系之中，等着我们去发掘。

### 7.7.1 单个节点的多重角色

借助 Neo4j 无模式的特性，单一的节点或关系并不限于只能映射到一个领域类中。有时候，对于有限的范围/上下文，一种很好的方式就是将领域类构造为更小的概念/角色。

例如，Order 在生命周期不同阶段使用的方式会有所区别。根据当前的状态，它可能会是购物车、客户订单、派发单或回单。每个状态都会关联不同的属性、约束或操作。通常来讲，这会建模为多个实体并分别存储到各自的表中，或者只有一个 Order 类，然后作为很大很稀疏的一个行存储到表中。借助于图数据库无模式的特性，Order 可以存储到一个节点中，但是只包含当前状态（以及过去状态中包含的并且以后会继续需要的属性）所需要的属性（和关系）。通常，会在生命周期之中获取属性和关系，并且在完成时被简化和锁定。

Spring Data Neo4j 允许使用不同的类来对这种实体进行建模，每个类涵盖生命周期的某一个阶段。这些实体共享一些属性；每个又有一些唯一的属性。所有的实体映射到同一个节点之上，根据在使用 `template.findOne(id,type)` 进行加载时所提供的类型，或者在运行时使用 `template.projectTo(object, type)`，这样就能够在不同的上下文中使用不同的实体了。当所投影的实体要进行存储时，只有它目前的属性（和关系）会被更新；其他已有的会保持现状。

### 7.7.2 以产品分类和标签为例讲解图中的索引

为了处理更大的产品目录并简化信息的探查，将产品进行分类是很重要的。一个原始的方式就是为每个产品使用一个分类属性，这在长期可用性方面是有所不足的。在图中，每个实体有多个指向分类节点的关联关系是很自然的事情。添加一个分类树，这样每个树都能与其子树保持关系并且每个产品与其所在的分类也能保持关系，这是很简单的。典型的应用场景为：

- 基于分类树导航；
- 列出某个分类子树下的所有产品；
- 列出相同分类下的类似产品；
- 查找产品分类之间隐式/不明显的关系（如儿童护理产品与年轻父母的生活产品）。

对于标签也是一样，它比分类的限制性更小，通常会形成一个更自然的图，所有实体都与标签关联，而不是像分类这样的等级树。在图数据库中，多个分类和标签都会形成隐形的二级索引结构，它们能够以很多不同的方式对存储的实体进行导航式访问，还有其他的二级索引（如地理信息、时间相关的指标或其他有意思的维度），如示例 7-22 所示。

#### 示例 7-22 产品的分类和标签

```
@NodeEntity
public class Category extends AbstractEntity {
    @Indexed(unique = true) String name;
    @Fetch // loads all children eagerly (cascading!)
    @RelatedTo(type="SUB_CAT")
    Set<Category> children = new HashSet<Category>();

    public void addChild(Category cat) {
        this.children.add(cat);
    }
}

@NoArgsConstructor
public class Product extends AbstractEntity {
    @RelatedTo(type="CATEGORY")
    Set<Category> categories = new HashSet<Category>();
```

```

public void addCategory(Category cat) {
    this.categories.add(cat);
}

public interface ProductRepository extends GraphRepository<Product> {
    @Query("START cat=node:Category(name={0}) "+
        "MATCH cat-[SUB_CAT*0..5]-leaf<-[:CATEGORY]-product "+
        "RETURN distinct product")
    List<Product> findByCategory(String category);
}

```

Category 形成了一个父子关系的嵌套组合结构。每个分类有一个唯一的名字和一组子类。Category 对象用来创建结构以及产品到分类之间的关系。为了使用产品之间的互联性，一个自定义（使用注解）的查询将会从起始（或根）分类开始，通过 0 到 5 个关系找到连接到这个子树上的产品。所有附着在关系上的产品将会作为列表返回。

### 7.7.3 利用类似的兴趣（协同过滤）

协同过滤（Collaborative filtering），如示例 7-23 所示，它基于这样的假设，那就是我们能找到其他的“人”，这些人与当前的用户相比具有类似的/可对比的兴趣和行为。至于用于描述相似的实际条件，如搜索/购买历史、评论等，都是与领域相关的。算法能够得到的信息越多，结果会越好。

在下一步中，将会考虑类似的人所购买或喜欢的产品（根据他们提到的次数以及/或他们的评分），除此之外，还可能会排除用户已经购买、拥有或不感兴趣的条目。

#### 示例 7-23 协同过滤

```

public interface ProductRepository extends GraphRepository<Product> {
    @Query("START cust=node({0}) " +
        " MATCH cust-[r1:RATED]->product<-[r2:RATED]-people " +
        " -[:ORDERED]->order-[:ITEMS]->suggestion " +
        " WHERE abs(r1.stars - r2.stars) <= 2 " +
        " RETURN suggestion, count(*) as score" +
        " ORDER BY score DESC")
    List<Suggestion> recommendItems(Customer customer);

    @MapResult
    interface Suggestion {
        @ResultColumn("suggestion") Product getProduct();
        @ResultColumn("score") Integer getScore();
    }
}

```

### 7.7.4 推荐

几乎在所有的领域，尤其是电子商务领域之中，为用户推荐感兴趣的产品主要会使用所收集到的信息，这些信息来源于所评论的产品和购买行为。很显然，我们可以

根据用户直接的评论来生成推荐，尤其是当缺乏实际的购买历史或没有关联的用户账号时。对于初始级别的建议来说，简单地按照数量和评论等级（或更为高级的评分机制）依序列出就足够了。

对于更高级的推荐来说，可以使用一些考虑到多个输入数据维度（如等级、购买历史、人口信息、广告曝光情况以及地理信息等）的算法。

示例 7-24 中的查询会查找顾客的所有评级信息并返回评级最高（取决于平均）的一页产品。

#### 示例 7-24 简单的推荐

```
public interface ProductRepository extends GraphRepository<Product> {  
    @Query("START product=node:product_search({0}) "+  
        "MATCH product<-[r:RATED]-customer "+  
        "RETURN product ORDER BY avg(r.stars) DESC")  
    Page<Product> listProductsRanked(String description, Pageable page);  
}
```

## 7.8 事务、实体生命周期以及抓取策略

Neo4j 是完全事务化的数据库，Spring Data Neo4j 集成了（声明式）Spring 事务管理，这构建在 Neo4j 所提供的事务管理器之上，兼容于 Spring JtaTransactionManager。事务管理器的 bean 命名为 neo4jTransactionManager（别名为 transactionManager）是在<neo4j:config />元素中创建的。因为事务管理器是默认设置的，@Transactional 注解只需要定义事务作用域。对图数据库的所有写操作都需要事务，但是读不需要事务。可以进行事务嵌套，但是嵌套的事务只会参与正在运行的父事务（类似于 REQUIRED）。

Spring Data Neo4j 以及 Neo4j 本身都能集成外部的 XA 事务管理器；Neo4j 手册 (<http://www.springsource.org/spring-data/neo4j>) 对其进行了详细描述。

对于简单的映射模式来说，生命周期很简单直接：新的实体在存储到图中之前只是一个 POJO 实例，此时它会将这个元素（节点或关系）的内部 id 保存到具有@GraphId 注解的域中，以用于随后的再次连接或合并。如果 id 没有设置的话，它会被作为新的实体来进行处理，当保存的时候会创建新的图元素。

当在简单映射模式下从图中抓取实体时，它们会自动分离开。数据会从图中复制出来并存储到领域对象实例中。使用简单映射模式中有一个很重要的方面就是抓取深度（fetch depth）。作为一种预防措施，在加载数据的时候，事务只会抓取实体的直接属性，默认并不会按照关系进行加载。

为了更为深入地对图进行抓取，需要在希望立即抓取的域上提供@Fetch 注解。对于已经加载的实体和域，template.fetch(...)方法将会从图中加载数据并对其进行更新。

## 7.9 高级映射模型

Spring Data Neo4j 也提供了更为高级的映射模式。它与简单映射模式的主要区别在于它提供了图的一个活跃 (live) 视图，并将其投影到领域对象。所以，每个域的访问会被拦截并被路由到对应的属性或关系（对于@RelatedTo[Via]的域）上。这个拦截在内部使用了 AspectJ 以发挥其魔力。

我们可以通过添加 org.springframework.data:spring-data-neo4j-aspects 依赖，然后配置 AspectJ 构建插件或激活加载时织入 (load-time-weaving) 来启用高级映射模式，如示例 7-25 所示。

### 示例 7-25 搭建 Spring Data Neo4j 高级映射

```
<properties>
    <aspectj.version>1.6.12</aspectj.version>
</properties>

<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-neo4j-aspects</artifactId>
    <version>${spring-data-neo4j.version}</version>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>${aspectj.version}</version>
</dependency>

.....
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>aspectj-maven-plugin</artifactId>
    <version>1.2</version>
    <dependencies>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjrt</artifactId>
            <version>${aspectj.version}</version>
        </dependency>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjtools</artifactId>
            <version>${aspectj.version}</version>
        </dependency>
    </dependencies>
    <executions>
        <execution>
            <goals>
                <goal>compile</goal>
                <goal>test-compile</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

```
<outxml>true</outxml>
<aspectLibraries>
  <aspectLibrary>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aspects</artifactId>
  </aspectLibrary>
  <aspectLibrary>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-neo4j-aspects</artifactId>
  </aspectLibrary>
</aspectLibraries>
<source>1.6</source>
<target>1.6</target>
</configuration>
</plugin>
```

任何时候域都会自动从图中进行读取，但是对于操作中的即时写入必须要包含在一个事务之中。因为对象可以在事务之外进行修改，所以会建立对象的关联/分离（attached/detached）生命周期。从图中加载的对象或者刚刚在一个事务中保存的对象是关联的；如果一个对象在事务外进行了修改或者是新创建的，那它就是分离的。对分离的对象进行修改只会保存在这个对象之中，而且只会在进行下次保存操作的时候才能反映到图中，这会使得实体再次变成关联的。

图数据库的活跃视图允许更为快速地进行操作以及对图“直接”修改。修改会立即体现到其他的图操作中，如遍历、Cypher 查询或 Neo4j 核心 API。因为读取都会基于当前的图，其他事务所提交的变化会立即可见。因为都是直接读取图数据库，所以在高级映射模式下没有必要进行抓取处理和设置@Fetch 注解。

## 7.10 使用 Neo4j 服务器

我们曾经提到过 Neo4j 有两种类型。可以很容易地基于任何 JVM 语言使用高性能嵌入式的 Java 数据库，最好要选择对应语言常用的 API/驱动 (<http://docs.neo4j.org/chunked/milestone/languages.html>)。集成嵌入式的数据库只需将 Neo4j 库添加到依赖之中即可。

另外的部署可选方案就是 Neo4j 服务器。Neo4j 服务器模块 (<http://www.neo4j.org/download>) 是一个简单的下载文件或操作系统包，它可以作为独立的服务运行。针对服务器的访问，提供了用于监控、操作和可视化的 Web 界面（见图 7-2）。通用的 REST API 可以用于进行编码式地访问数据库。REST API 暴露了 Cypher 端点（endpoint）。通过使用 Neo4j-Java-Rest-Binding (<https://github.com/neo4j/java-rest-binding>)（对 REST 调用的 Neo4j Java API 进行了封装）与服务器进行透明地交互，Spring Data Neo4j 可以很容易地使用服务器。

通过依赖于 org.springframework.data:spring-data-neo4j-rest 并在搭建时指向远程服

务器的 URL，就可以基于安装版的服务器使用 Spring Data Neo4j 了（参见示例 7-26）。需要注意的是使用当前的实现，并不是所有的调用在网络 API 上的传输都是最优的，所以每个操作的服务端交互都会受到网络传输时间和带宽的影响。建议尽可能地使用远程执行的查询以降低这种影响。

### 示例 7-26 搭建服务器连接配置

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:neo4j="http://www.springframework.org/schema/data/neo4j"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/neo4j
    http://www.springframework.org/schema/data/neo4j/spring-neo4j.xsd">

    <neo4j:config graphDatabaseService="graphDatabaseService" />
    <bean id="graphDatabaseService"
        class="org.springframework.data.neo4j.rest.SpringRestGraphDatabase">
        <constructor-arg index="0" value="http://localhost:7474/db/data" />
    </bean>
</beans>
```

SpringRestGraphDatabase 通过 RestAPI 实例进行连接，可以用它更为高效地执行单个或批量的 REST 操作。例如，借助于 RestAPI，不管是对于传统的实体还是特定的实体，进行创建后立即设置属性都是更为高效的。

## 7.11 从这里继续学习

本章展现了图数据库，尤其是 Neo4j，为我们所带来的可改变的东西以及 Spring Data Neo4j 如何实现对它们的便利访问，同时还能保持原始低层级图处理方式是可用的。

接下来应该做的事情就是考虑所使用（或者想要使用）的数据并了解这些实体是如何联系在一起的。近距离的观察——你会发现它们的联系程度比乍看上去要紧密得多。选取一个领域对象，并将它首先画在白板上然后再存储到图数据库中，这是实现这些理念背后强大功能的第一步。要编写使用这些相互联系数据的应用，借助 Spring Data Neo4j 是一个很好的开始。它能够让你很容易地创建图数据并将图查询的结果暴露为你所熟知的 POJO，这会简化与其他库或（UI）框架的集成。

要了解如何创建完整的 Web 应用，可以参见参考资料中的[Hunger12]，它也是参考文档和 GitHub 仓库的一部分。这个教程很详细地介绍了用于电影交流的社交类型的数据库 cineasts.net，并且阐述了数据建模、与外部服务集成以及 Web 层。

如果有问题的话，可以随时去 Springsource 社区 (<http://spring.neo4j.org/discussions>)、Stackoverflow 或 Neo4j 的 Google 群组 (<http://neo4j.org/forums>) 来寻找答案。希望你能喜欢它！

## Redis：键/值存储

本章将会介绍 Spring Data 对键/值存储 Redis (<http://redis.io>) 所提供的支持。我们将简单看一下 Redis 如何管理数据、如何安装和配置服务器并了解如何通过命令行与之进行交互。然后，我们会看一下如何从 Java 连接到服务器以及 RedisTemplate 如何组织众多的操作，以便我们操作 Redis 中所存储的数据。我们会了解使用 JSON 存储 POJO 的方式并且还会简要讨论如何使用快速高效的发布/订阅 (Publish/Subscribe，发布/订阅) 功能以实现基本的基于事件编程。

### 8.1 Redis 概述

Redis (<http://redis.io>) 是非常高性能和轻量级的数据存储。它提供了键/值的数据访问方式来持久化字节数组、列表、集合以及哈希数据结构。它支持原子级的计数器 (counter) 并具有一个高效的基于主题的发布/订阅消息功能。Redis 易于安装和运行，但最重要的一点在于其非常非常快的数据访问能力。它所缺乏的是复杂的查询功能（如在 Riak ([wiki.basho.com/Riak.html](http://wiki.basho.com/Riak.html)) 或 MongoDB (<http://www.mongodb.org/>) 中所提供的），但是会以速度和效率来弥补。Redis 服务器也可以进行集群，从而提供非常灵活的部署方式。通过使用安装时自带的 redis-cli 二进制文件，可以很容易地通过命令行与 Redis 进行交互。

#### 8.1.1 搭建 Redis

为了使用 Redis，可能希望在本地进行安装。根据平台不同，安装过程可能简单到只需输入一个文字命令。最简单的安装过程，如示例 8-1 所示，是在 Mac OS X 上借助 Homebrew (<http://mxcl.github.com/homebrew>) 来安装。在其他的 UNIX 系统中，如果你基于源码构建服务器的话，这是原生支持的（构建指令在 Redis 的站点上可以找到，它与我们所构建的其他\*NIX 包并无二致——也就是将其解压、

通过 cd 命令转移到指定目录中，然后输入 make）。Redis 的下载页面 (<http://redis.io/download>) 也列出了许多非官方的版本，可将 Redis 安装到 Win32/64 平台，尽管它们没有太多考虑到产品的质量。在本章中使用\*NIX 版本，这也是 Redis 最擅长的地方。

### 示例 8-1 借助 Homebrew 在 Mac OS X 上安装 Redis

```
$ brew install redis
==> Downloading http://redis.googlecode.com/files/redis-2.4.15.tar.gz
#####
100.0%
==> make -C /private/tmp/homebrew-redis-2.4.15-WbS5/redis-2.4.15/src CC=/usr/bin/clang
==> Caveats
If this is your first install, automatically load on login with:
  mkdir -p ~/Library/LaunchAgents
  cp /usr/local/Cellar/redis/2.4.15/homebrew.mxcl.redis.plist ~/Library/LaunchAgents/
  launchctl load -w ~/Library/LaunchAgents/homebrew.mxcl.redis.plist

If this is an upgrade and you already have the homebrew.mxcl.redis.plist loaded:
  launchctl unload -w ~/Library/LaunchAgents/homebrew.mxcl.redis.plist
  cp /usr/local/Cellar/redis/2.4.15/homebrew.mxcl.redis.plist ~/Library/LaunchAgents/
  launchctl load -w ~/Library/LaunchAgents/homebrew.mxcl.redis.plist

To start redis manually:
  redis-server /usr/local/etc/redis.conf

To access the server:
  redis-cli
==> Summary
/usr/local/Cellar/redis/2.4.15: 9 files, 556K, built in 12 seconds
```

这样很快就可以让服务器运行起来，并且看到一些结果，我们在前台使用一个终端来运行服务器。这对于调试来说是很好的，因为它会直接将日志打印在控制台上，这样就能够知道服务器在内部是如何运行的。当然，要让服务器在机器重启后都能自动运行的启动脚本是因平台而异的。这个搭建过程作为练习留给读者。

我们使用服务器的默认设置，所以启动它非常简单，只需执行 redis-server，如示例 8-2 所示。

### 示例 8-2 启动服务器

```
$ redis-server
[91688] 25 Jul 09:37:36 # Warning: no config file specified, using the default config.
  In order to specify a config file use 'redis-server /path/to/redis.conf'
[91688] 25 Jul 09:37:36 * Server started, Redis version 2.4.15
[91688] 25 Jul 09:37:36 * The server is now ready to accept connections on port 6379
[91688] 25 Jul 09:37:36 - 0 clients connected (0 slaves), 922304 bytes in use
```

## 8.1.2 使用 Redis Shell

Redis 自带了非常有用的命令行 Shell，这样就能够以交互的方式或者通过批处理任务的方式来进行使用。我们只会用到 Shell 的交互式部分，以便查看服务器内部的状态、查找数据并与之进行交互。命令行 Shell 有一个扩展了的帮助系统，如示例

8-3 所示，所以当使用它的时候，连续敲击 Tab 键就可以得到 Shell 的帮助提示。

### 示例 8-3 与 Redis 服务器进行交互

```
$ redis-cli
redis 127.0.0.1:6379> help
redis-cli 2.4.15
Type: "help @<group>" to get a list of commands in <group>
      "help <command>" for help on <command>
      "help <tab>" to get a list of possible help topics
      "quit" to exit
redis 127.0.0.1:6379> |
```

Redis 文档 (<http://redis.io/commands>) 很有用，因为它对所有可用的命令都进行了整体的介绍并展示了一些使用示例。请保持这一页很容易翻阅到，因为在后面你会经常参考它。

花点时间来熟悉基本的 SET 和 GET 命令是有好处的。让我们花一点时间来看看如何插入和检索数据，如示例 8-4 所示。

### 示例 8-4 在 Redis 中 SET 和 GET 数据

```
$ redis-cli
redis 127.0.0.1:6379> keys *
(empty list or set)
redis 127.0.0.1:6379> set spring-data-book:redis:test-value 1
OK
redis 127.0.0.1:6379> keys *
1) "spring-data-book:redis:test-value"
redis 127.0.0.1:6379> get spring-data-book:redis:test-value
"1"
redis 127.0.0.1:6379> |
```

需要注意的一点是，当 SET 值的时候，我们并没有在值 1 上加引号。Redis 并不会像其他的数据存储那样具有数据类型，所以它将所有的值都视为一串字节。在命令行中，可以看到它们会作为字符串进行打印。当通过 GET 得到返回的数据时，在命令行 Shell 看到的会是“1”。可以看到，使用了引号，那么它就是字符串。

## 8.2 连接到 Redis

Spring Data Redis 支持通过 Jedis (<https://github.com/xetorthio/jedis>)、JRedis (<https://github.com/alphazero/jredis>)、RJC (<https://github.com/e-mzungu/rjc>) 或 SRP (<https://github.com/spullara/redis-protocol>) 驱动库连接到 Redis 上。不论选择哪一个驱动，在使用 Spring Data Redis 库时都不会有任何差异。这些驱动之间的差异已经被抽象到了一组通用的 API 和模板风格的帮助类中。简单起见，示例中使用了 Jedis 驱动。

为了使用 Jedis 来连接 Redis，我们需要创建一个 `rg.springframework.data.redis.connection.`

`jedis.JedisConnectionFactory` 实例。其他的驱动库也有对应的 `ConnectionFactory` 子类。使用 `JavaConfig` 的配置可能会如示例 8-5 所示。

#### 示例 8-5 通过 Jedis 连接到 Redis

```
@Configuration
public class ApplicationConfig {

    @Bean
    public JedisConnectionFactory connectionFactory() {
        JedisConnectionFactory connectionFactory = new JedisConnectionFactory();
        connectionFactory.setHostName("localhost");
        connectionFactory.setPort(6379);
        return connectionFactory;
    }
}
```

当通过 Spring Data Redis 访问 Redis 时，可能会用到的核心抽象是 `org.springframework.data.redis.core.RedisTemplate`。因为 Redis 的特性集合过于庞大，以至于无法有效地将其封装到一个类之中，各种关于数据的操作被拆分到了单独的 `Operations` 类中，如下所示（名字说明了自身的功能）：

- `ValueOperations`
- `ListOperations`
- `SetOperations`
- `ZSetOperations`
- `HashOperations`
- `BoundValueOperations`
- `BoundListOperations`
- `BoundSetOperations`
- `BoundZSetOperations`
- `BoundHashOperations`

### 8.3 对象转换

因为 Redis 直接处理字节数组并没有原生支持对象到 `byte[]` 的转换，Spring Data Redis 项目提供了一些帮助类，以便于更加简单地从 Java 代码中读取和写入数据。默认情况下，所有的键和值都存储为序列化的 Java 对象。不过，如果要大量地处理 `String`，也会有一个模板类——`StringRedisTemplate`，如示例 8-6 所示——这样就安装了 `String` 的序列化器（`serializer`），所带来的好处就是 Redis 命令行界面的键和

值更加易于人工阅读。

### 示例 8-6 使用 StringRedisTemplate

```
@Configuration
public class ApplicationConfig {
    @Bean
    public JedisConnectionFactory connectionFactory() { ... }

    @Bean
    public StringRedisTemplate redisTemplate() {
        StringRedisTemplate redisTemplate = new StringRedisTemplate();
        redisTemplate.setConnectionFactory(connectionFactory());
        return redisTemplate;
    }
}
```

为了影响键和值进行序列化和反序列化的方式，Spring Data Redis 提供了 RedisSerializer 抽象，它负责对 Redis 中所存储的字节进行实际上的读取和写入。我们需要为模板的 keySerializer 或 valueSerializer 属性设置一个 org.springframework.data.redis.serializer.RedisSerializer。String 已经有一个内置的 RedisSerializer，所以对于 String 类型作为键并且 Long 类型作为值的情况下，需要为 Long 类型创建一个序列化器，如示例 8-7 所示。

### 示例 8-7 创建可重用的序列化器

```
public enum LongSerializer implements RedisSerializer<Long> {
    INSTANCE;

    @Override
    public byte[] serialize(Long aLong) throws SerializationException {
        if (null != aLong) {
            return aLong.toString().getBytes();
        } else {
            return new byte[0];
        }
    }

    @Override
    public Long deserialize(byte[] bytes) throws SerializationException {
        if (bytes.length > 0) {
            return Long.parseLong(new String(bytes));
        } else {
            return null;
        }
    }
}
```

为了在使用 Redis 时借助这些序列化器来简化类型转换，需要设置模板类的 keySerializer 和 valueSerializer 属性，如示例 8-8 所示的 JavaConfig 代码片段。

### 示例 8-8 在模板实例中使用序列化器

```
@Bean
public RedisTemplate<String, Long> longTemplate() {
```

```

private static final StringRedisSerializer STRING_SERIALIZER =
    new StringRedisSerializer();

RedisTemplate<String, Long> tmpl = new RedisTemplate<String, Long>();
tmpl.setConnectionFactory(connFac);
tmpl.setKeySerializer(STRING_SERIALIZER);
tmpl.setValueSerializer(LongSerializer.INSTANCE);

return tmpl;
}

```

现在，可以存储数字到 Redis 中，而不用担心 byte[] 到 Long 之间的类型转换了。因为 Redis 提供了如此之多的操作（帮助类中会有很多的方法），所以 set 和 get 值的方法定义在各个 RedisOperation 接口之中 (<http://static.springsource.org/spring-data/data-redis/docs/current/api/org/springframework/data/redis/core/package-summary.html>)。通过在 RedisTemplate 中调用对应的 opsForX 方法，可以访问到这些接口。因为在示例中我们只会存储离散的值，所以将会用到的是 ValueOperations 模板，如示例 8-9 所示。

### 示例 8-9 当 set 和 get 值时，自动进行类型转换

```

public class ProductCountTracker {

    @Autowired
    RedisTemplate<String, Long> redis;

    public void updateTotalProductCount(Product p) {
        // Use a namespaced Redis key
        String productCountKey = "product-counts:" + p.getId();

        // Get the helper for getting and setting values
        ValueOperations<String, Long> values = redis.opsForValue();

        // Initialize the count if not present
        values.setIfAbsent(productCountKey, 0L);

        // Increment the value by 1
        Long totalOfProductInAllCarts = values.increment(productCountKey, 1);
    }
}

```

在应用中调用这个方法并将 id 值为 1 的 Product 传递进来的时候，可以在 redis-cli 中查看这个值，通过在命令行中使用 get product-counts:1，可以看到字符串“1”。

## 8.4 对象映射

能够将像计数器和字符串这样的简单值存储到 Redis 中是很棒的，但是通常也会需要存储更为丰富的互相关联的信息。在有些场景下，它们可能会是对象的属性。在另外的一些场景下，可能会是哈希的键和值。

通过使用 RedisSerializer，可以将对象存储为 Redis 中的值。但是，这样做的话，会很难单独地探查或检索对象的属性。在这种场景下，可能希望使用 Redis 哈希。将属性存储到一个哈希中，这样就可以将所有的属性作为 Map<String, String>取出，然后进行访问，也可以只引用哈希中的单个属性而不去涉及其他的属性。

在 Redis 中所有的事务都是 byte[]，对于这个哈希的例子我们会使用简单的 String 作为键和值。对哈希的操作，与对其他值、集合等是一样的，要通过 RedisTemplate 的 opsForHash()方法，如示例 8-10 所示。

### 示例 8-10 使用 HashOperations 接口

```
private static final RedisSerializer<String> STRING_SERIALIZER =
    new StringRedisSerializer();

public void updateTotalProductCount(Product p) {

    RedisTemplate<String, String> tmpl = new RedisTemplate();
    tmpl.setConnectionFactory(connectionFactory);
    // Use the standard String serializer for all keys and values
    tmpl.setKeySerializer(STRING_SERIALIZER);
    tmpl.setHashKeySerializer(STRING_SERIALIZER);
    tmpl.setHashValueSerializer(STRING_SERIALIZER);

    HashOperations<String, String, String> hashOps = tmpl.opsForHash();

    // Access the attributes for the Product
    String productAttrsKey = "products:attrs:" + p.getId();

    Map<String, String> attrs = new HashMap<String, String>();

    // Fill attributes
    attrs.put("name", "iPad");
    attrs.put("deviceType", "tablet");
    attrs.put("color", "black");
    attrs.put("price", "499.00");

    hashOps.putAll(productAttrsKey, attrs);

}
```

假设 Product 的 id 为 1，通过在 redis-cli 中使用 HKEYS 命令，可以列出哈希中所有的键，如示例 8-11 所示。

### 示例 8-11 列出哈希的键

```
redis 127.0.0.1:6379> hkeys products:attrs:1
1) "price"
2) "color"
3) "deviceType"
4) "name"
redis 127.0.0.1:6379> hget products:attrs:1 name
"iPad"
```

尽管在本例中只是使用 String 作为哈希的值，但是可以使用任意的 RedisSerializer 作为模板的 hashValueSerializer。例如，如果想存储复杂的对象而不是 String，那么可

能需要将模板中的 `hashValueSerializer` 替换为 `org.springframework.data.redis.serializer.JacksonJsonRedisSerializer` (<http://static.springsource.org/spring-data/data-redis/docs/current/api/org/springframework/data/redis/serializer/JacksonJsonRedisSerializer.html>) 或者 `org.springframework.data.redis.serializer.OxmSerializer` (<http://static.springsource.org/spring-data/data-redis/docs/current/api/org/springframework/data/redis/serializer/OxmSerializer.html>)，前者会将对象序列化 JSON，而后者会使用 Spring OXM 对对象进行编组和解组（marshalling 和 unmarshalling）。

## 8.5 原子级计数器

很多人选择使用 Redis，是因为它所支持的原子级计数器。如果多个应用都指向同一个 Redis 实例，那么这些分布式的应用可以对某个计数器进行一致且原子的增长，从而保证其唯一性。Java 提供了 `AtomicInteger` 和 `AtomicLong` 类，可以跨线程实现原子级的计算器值增加，但是如果这些计数器位于其他的 JVM 处理过程或其他 `ClassLoader` 之中的时候，它就没有任何用处了。Spring Data Redis 实现了几个类似于 `AtomicInteger` 和 `AtomicLong` 的帮助类，它们都由 Redis 实例作为支持。在应用中访问分布式的计数器，只需要创建这些帮助类的实例并将它们都指向相同的 Redis 服务器即可，如示例 8-12 所示。

### 示例 8-12 使用 RedisAtomicLong

```
public class CountTracker {  
  
    @Autowired  
    RedisConnectionFactory connectionFactory;  
  
    public void updateProductCount(Product p) {  
        // Use a namespaced Redis key  
        String productCountKey = "product-counts:" + p.getId();  
  
        // Create a distributed counter.  
        // Initialize it to zero if it doesn't yet exist  
        RedisAtomicLong productCount =  
            new RedisAtomicLong(productCountKey, connectionFactory, 0);  
  
        // Increment the count  
        Long newVal = productCount.incrementAndGet();  
    }  
}
```

## 8.6 发布/订阅功能

使用 Redis 的另一个重要好处在于其简单快速的发布/订阅功能 (<http://redis.io/topics/pubsub>)。尽管它并没有完整消息代理（message broker）等高级特性，但是 Redis

的发布/订阅功能可以用来创建轻量级和灵活的事件总线。Spring Data Redis 提供了一些帮助类，让这个功能使用起来特别容易。

### 8.6.1 对信息进行监听和响应

遵循 JMS MessageListenerAdapter 的模式，Spring Data Redis 有一个 MessageListener Adapter 的抽象，它们的工作方式基本上是相同的，如示例 8-13 所示。按照 JMS，如果你不想与特定的接口绑定的话，那么在所能接受的监听器种类方面，MessageListenerAdapter 是很灵活的。可以传递进来一个带有 handleMessage 方法的 POJO，这个方法的第一个参数可以是 org.springframework.data.redis.connection.Message、String、byte[]，或者如果使用了对应的 RedisSerializer，它也可以是任意可转换的类型。可以定义可选的第二个参数，它是触发这次调用的通道（channel）或模式。如果想避免基于反射进行调用（当传递进来 POJO 的时候，会这样做），那么还可以实现 MessageListener 接口，从而为你的 Bean 实现固定的协议。

#### 示例 8-13 使用 JavaConfig 添加简单的 MessageListener

```
@Bean public MessageListener dumpToConsoleListener() {  
    return new MessageListener() {  
        @Override  
        public void onMessage(Message message, byte[] pattern) {  
            System.out.println("FROM MESSAGE: " + new String(message.getBody()));  
        }  
    };  
}
```

Spring Data Redis 允许你在 MessageListenerAdapter 上使用 POJO，容器将会使用你所提供的转换器将传入的信息转换为你所自定义的类型，如示例 8-14 所示。

#### 示例 8-14 搭建 MessageListenerAdapter 以及使用 POJO 的简单信息监听器

```
@Bean MessageListenerAdapter beanMessageListener() {  
    MessageListenerAdapter listener = new MessageListenerAdapter( new BeanMessageListener()  
    );  
    listener.setSerializer( new BeanMessageSerializer() );  
    return listener;  
}
```

BeanMessageListener 如示例 8-15 所示，是一个简单的 POJO，它有一个名为 handleMessage 的方法定义，其第一个参数类型为 BeanMessage（为这个示例任意创建的一个类）。它只有一个名为 message 的属性。我们的 RedisSerializer 会把这个 String 的内容存储为字节。

#### 示例 8-15 使用 JavaConfig 添加 POJO 监听器

```
public class BeanMessageListener {  
    public void handleMessage( BeanMessage msg ) {  
        System.out.println( "msg: " + msg.message );  
    }  
}
```

当事件触发时，真正负责调用监听器的组件是 `org.springframework.data.redis.listener.RedisMessageListenerContainer`。如示例 8-16 所示，它的配置需要 `RedisConnectionFactory` 和一组监听器。如果在 `ApplicationContext` 创建它的话，这个容器本身会有生命周期方法，这些方法会被 Spring 容器调用。如果以编码的形式创建这个容器的话，需要手动调用 `afterPropertiesSet()` 和 `start()` 方法。但是需要记住的一点是，要在调用 `start()` 方法之前设置监听器，否则的话，处理方法不能被调用到，这是因为装配已经在 `start()` 中完成了。

#### 示例 8-16 配置 RedisMessageListenerContainer

```
@Bean RedisMessageListenerContainer container() {
    RedisMessageListenerContainer container = new RedisMessageListenerContainer();
    container.setConnectionFactory(redisConnectionFactory());
    // Assign our BeanMessageListener to a specific channel
    container.addMessageListener(beanMessageListener(),
        new ChannelTopic("spring-data-book:pubsub-test:dump"));
    return container;
}
```

### 8.6.2 在 Redis 中使用 Spring 的缓存抽象

Spring 3.1 (<http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/cache.html>) 引入了通用且可重用的缓存抽象。这样，可以很容易地将方法调用的结果缓存到 POJO 中，而无需明确地管理某些处理过程，如检查缓存条目是否存在、加载新的数据以及过期旧的缓存条目等。在完成这些功能方面，Spring 3.1 为使用各种不同的后端缓存技术提供了帮助。

Spring Data Redis 通过 `org.springframework.data.redis.cache.RedisCacheManager` 支持了这种通用的缓存抽象。为了将 Redis 指定为缓存后端并在 Spring 中使用缓存的注解，需要在 `ApplicationContext` 中定义 `RedisCacheManager` Bean，然后就可以像以往那样对 POJO 添加注解，在想缓存的方法上使用 `@Cacheable` 注解。

`RedisCacheManager` 的构造器中需要一个配置好的 `RedisTemplate`。在本例中，我们让缓存抽象层生成唯一的整数作为缓存的键值。关于缓存管理器如何存储结果方面，有很多的可选项。可以在 `@Cacheable` 的方法上添加适当的注解 (<http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/cache.html#cache-annotations>) 来配置这些行为。在示例 8-17 中，我们对键使用了整数序列化器并对值使用了内置的 `JdkSerializationRedisSerializer`，原因是并不知道所要存储的是什么。使用 JDK 的序列化，就可以缓存任意实现了 `Serializable` 接口的 Java 对象。

在使用 `JavaConfig` 时，为了在 `ApplicationContext` 中启用缓存拦截器，只需在 `@Configuration` 注解之上再添加 `@EnableCaching` 就可以了。

### 示例 8-17 使用 RedisCacheManager 配置缓存

```
@Configuration  
@EnableCaching  
public class CachingConfig extends ApplicationConfig {  
  
    @SuppressWarnings({"unchecked"})  
    @Bean public RedisCacheManager redisCacheManager() {  
        RedisTemplate tmpl = new RedisTemplate();  
        tmpl.setConnectionFactory( redisConnectionFactory() );  
        tmpl.setKeySerializer( IntSerializer.INSTANCE );  
        tmpl.setValueSerializer( new JdkSerializationRedisSerializer() );  
        RedisCacheManager cacheMgr = new RedisCacheManager( tmpl );  
        return cacheMgr;  
    }  
  
    @Bean public CacheableTest cacheableTest() {  
        return new CacheableTest();  
    }  
}
```

## 第四部分

---

# 快速应用开发



# 使用 Spring Roo 实现持久层

Spring Roo 是针对 Java 开发人员的快速应用开发工具。借助于 Roo，在几分钟内就能很容易地构建出完整的 Java 应用程序。

本章不会涵盖 Roo 开发的各个方面，而是关注于它通过 Spring Data 为 JPA 和 MongoDB 提供的新的 Repository 支持。如果想阅读关于 Roo 的更多资料，可以访问 Spring Roo 项目的主页 (<http://www.springsource.org/spring-roo/>)，在这里可以找到参考手册的链接。在项目的主页上，可以看到 Josh Long 和 Steve Mayzak 编写的免费的 O'Reilly 电子书《Getting Started with Roo》[LongMay11]。这本电子书介绍了较旧的 Roo1.1 版本，这个版本并不支持 Repository 层，但总的来说这是一个很好的 Roo 入门文档。使用 Spring Roo 的最新指导是 Ken Rimple 和 Srini Penchikala 编写的《Spring Roo in Action》[RimPen12]。

## 9.1 Roo 简介

Roo 的魔力体现在结合 AspectJ 并通过代码生成的方式为领域和 Web 类注入行为。使用 Roo 工程的时候，工程文件被 Roo 所监控并生成额外的构件（artifact）。你还会有关于可编辑的常规 Java 类，但是它们会自动提供一些额外的特性。如果使用 Roo 创建类并为这个类添加一个或多个提供额外功能的注解，那么 Roo 会生成对应的 AspectJ 文件，这个文件中包含了一个或多个的 AspectJ 类型间声明（Inter Type Declarations, ITD）。例如，@RooJavaBean 注解会触发 AspectJ 方面声明的生成，这个声明提供了在 Java 类中引入 getter 和 setter 的 ITD。这样就不再需要自己编码了。现在快速地看一个例子，了解这是如何使用的，示例 9-1 展示了一个简单的 Bean 类：

### 示例 9-1 简单的 Java Bean 的类：Address.java

```
@RooJavaBean  
public class Address {
```

```
    private String street;  
  
    private String city;  
    private String country;  
}
```

可以看到，我们并没有编写 getter 和 setter 方法。它们会由提供支持的 AspectJ 方面文件引入进来，因为在这里我们使用了@RooJavaBean 注解。生成的 AspectJ 方面文件如示例 9-2 所示。

### 示例 9-2 所生成的 AspectJ 方面定义：Address\_Roo\_JavaBean.aj

```
// WARNING: DO NOT EDIT THIS FILE. THIS FILE IS MANAGED BY SPRING ROO.  
// You may push code into the target .java compilation unit if you wish to edit any member(s).  
  
package com.oreilly.springdata.roo.domain;  
  
import com.oreilly.springdata.roo.domain.Address;  
  
privileged aspect Address_Roo_JavaBean {  
  
    public String Address.getStreet() {  
        return this.street;  
    }  
  
    public void Address.setStreet(String street) {  
        this.street = street;  
    }  
  
    public String Address.getCity() {  
        return this.city;  
    }  
  
    public void Address.setCity(String city) {  
        this.city = city;  
    }  
  
    public String Address.getCountry() {  
        return this.country;  
    }  
  
    public void Address.setCountry(String country) {  
        this.country = country;  
    }  
}
```

可以看到，这里定义为一个有权限的切面，也就是可以访问目标类中所声明的私有变量。定义 ITD 的方式就是在方法名之前添加上目标类的名字，中间用点号分割。所以 public String Address.getStreet() 将会为 Address 引入一个方法签名名为 public String getStreet() 的新方法。

Roo 遵循了一个特定的命名模式，这样很容易识别它生成了什么文件。要使用 Roo，可以使用命令行 Shell 也可以直接编辑源文件。Roo 会同步和维护源文件，并在必要的时候进行文件生成。

当使用 Roo 创建工程的时候，它会生成一个 pom.xml 文件，当使用 Maven 构建工

程的时候会用到这个文件。在这个 pom.xml 文件中，有 Maven 的编译器并定义了一个 AspectJ 的插件。这意味着所有的 AspectJ 方面是在编译期织入进来的。实际上，在构建生成的 Java 类文件中并没有保留任何 Roo 相关的东西。因此没有运行时的依赖。另外，Roo 的注解只会保留在源码级别，所以它们不会成为类文件的一部分。实际上，如果愿意的话，可以很容易地去除掉 Roo。可以将 AspectJ 文件中所定义的所有代码放到对应的源文件中并将这些 AspectJ 文件移除。这称为推进式重构 (push-in refactoring)，这样的话，得到的是一个纯 Java 的解决方案，就像所有的事情都是从头开始自己编写的。应用依然会保留所有的功能。

## 9.2 Roo 的持久层

Spring Roo 在开始的时候只支持 JPA 作为唯一的持久化可选方案。就数据访问层来说，这未免有些太武断了。Roo 规定了一种称为活动记录 (Active Record) 的数据访问风格，在这里每个实体要提供自己的查找、保存和删除方法。

从 Roo 1.2 版本开始，对于持久层有了更多的可选方案，如图 9-1 所示。Roo 现在允许采用默认的活动记录风格也能采用基于 Repository 的持久层，这可以选择。如果选择 Repository 方式，则选择 JPA 或 MongoDB 作为持久化提供者。Roo 实际上所支持的 Repository 就是由 Spring Data 所提供的，这个在第 2 章已经看到过了。

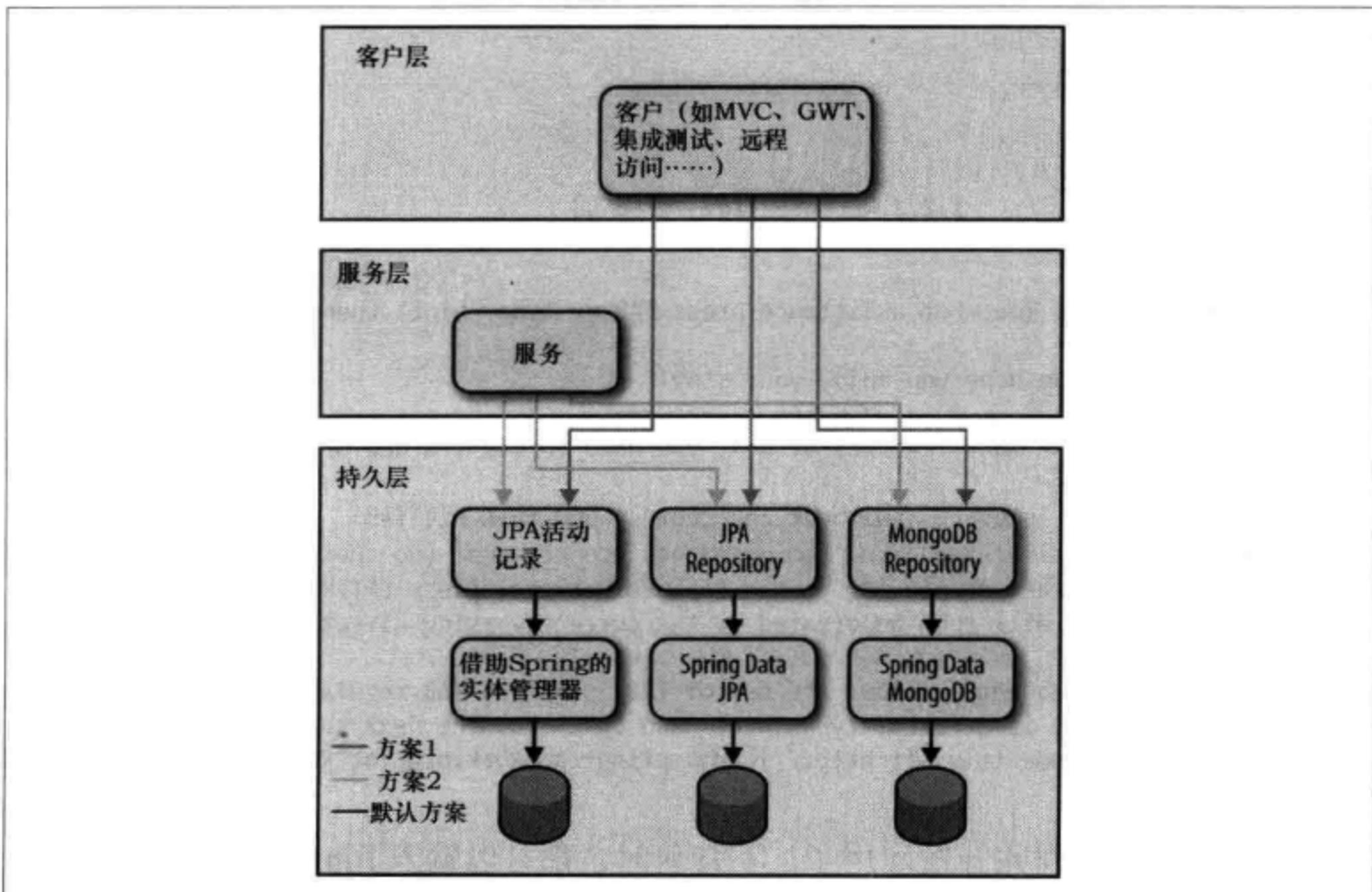


图 9-1 Spring Roo 1.2 的分层架构

除了可选的 Repository 层，Roo 现在还允许定义服务层，这一层可以定义在活动记录编程风格之上也可以定义在 Repository 持久层之上。

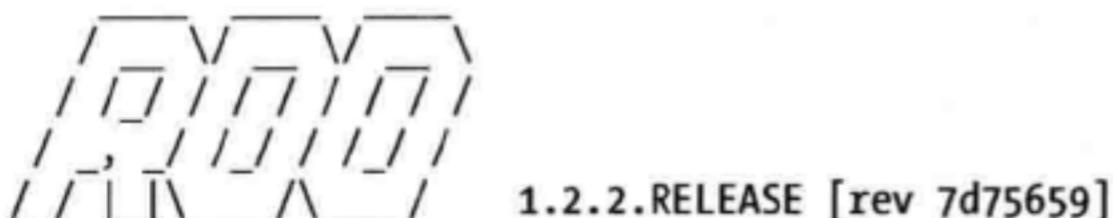
## 9.3 快速起步

可以在命令行工具中使用 Roo，也可以在 IDE 中使用，免费的 Spring Tool Suite 就内置了对 Roo 的支持。另外一个支持 Roo 的 IDE 是 IntelliJ IDEA，但是在这里不会进行介绍。

### 9.3.1 借助命令行使用 Roo

首先，在下载页面(<http://www.springsource.com/download/community?project=Spring%20Roo>) 中下载最新的 Spring Roo 发布版本。文件下载完成后，将其解压到系统中的某个位置。在 *bin* 目录下，对于 UNIX 风格的系统来说，有一个名为 *roo.sh* 的 Shell 脚本，对于 Window 来说也会有一个 *roo.bat* 批处理文件。如果想创建 Roo 工程，只需创建工程目录，然后使用 Shell 脚本或批处理文件启动 Roo。如果已将 *bin* 目录添加到系统的 path 之中，那么可以使用命令的名字来启动 Roo，否则的话，需要提供完整的限定路径。

Roo 启动后，可以看到如下的欢迎界面（在提示符中输入了 *hint* 来获取更多的信息）：



Welcome to Spring Roo. For assistance press TAB or type "hint" then hit ENTER.

roo> hint

Welcome to Roo! We hope you enjoy your stay!

Before you can use many features of Roo, you need to start a new project.

To do this, type 'project' (without the quotes) and then hit TAB.

Enter a --topLevelPackage like 'com.mycompany.projectname' (no quotes).

When you've finished completing your --topLevelPackage, press ENTER.

Your new project will then be created in the current working directory.

Note that Roo frequently allows the use of TAB, so press TAB regularly.

Once your project is created, type 'hint' and ENTER for the next suggestion.

You're also welcome to visit <http://forum.springframework.org> for Roo help.

roo>

现在，可以创建工程并开发应用了。不管何时，都可以输入 **hint**，Roo 将会基于应用开发目前的状态，给出下一步要做什么的指导。为了减少输入量，单击 Tab 键的

时候，Roo 会尝试帮助你完成正在输入的命令。

### 9.3.2 借助 Spring Tool Suite 使用 Roo

Spring Tool Suite 内置了对 Roo 的支持，并且它还打包了 Maven 以及开发版本的 VMware vFabric tc Server。这意味着备齐了使用 Roo 开发应用程序的所有工具。使用以下的菜单项来创建第一个 Roo 应用程序：File→New→Spring Roo Project。可以在图 9-2 中看到这个过程。

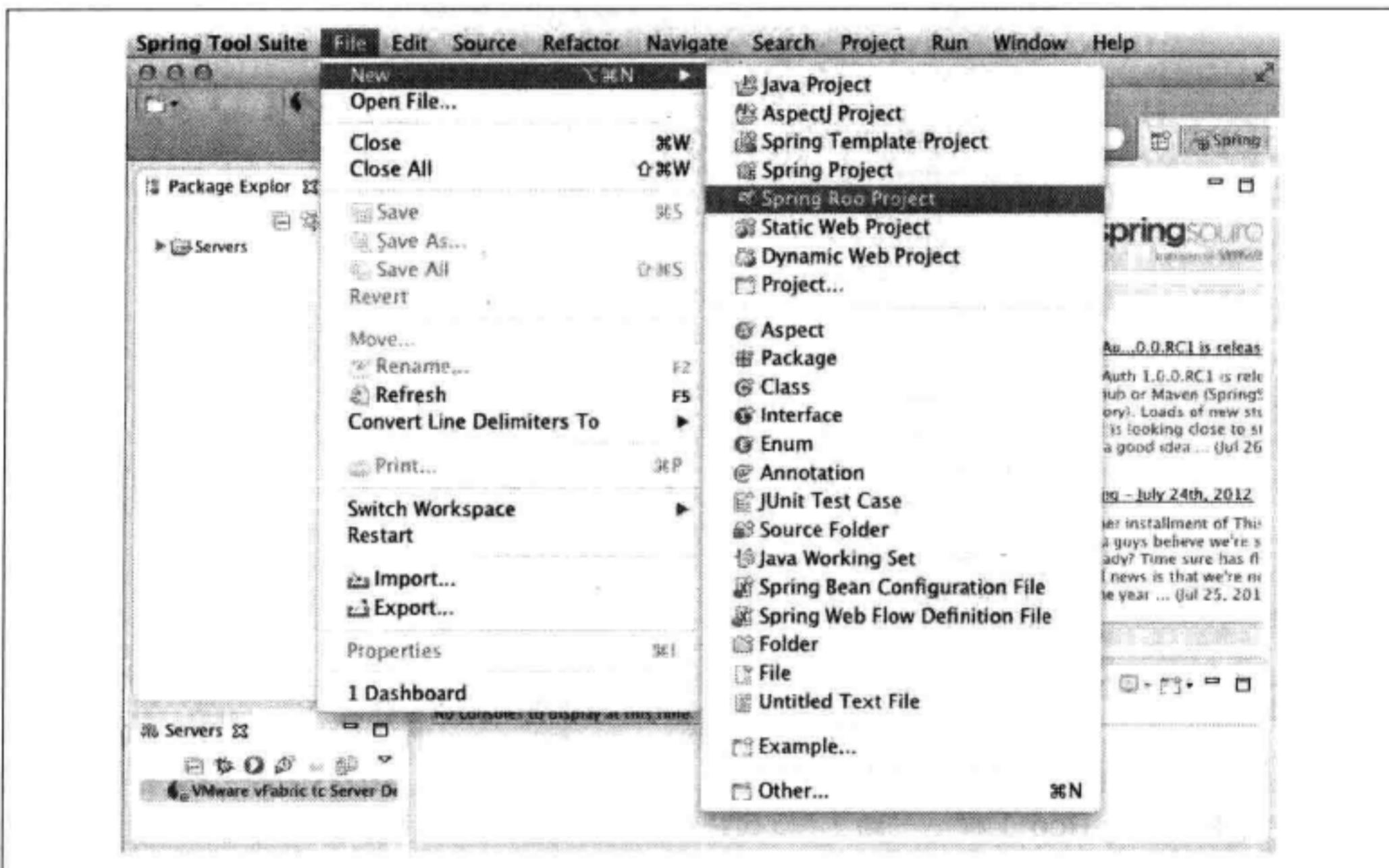


图 9-2 创建 Spring Roo 工程——菜单项

这会打开“Create a new Roo Project”对话框，其界面如图 9-3 所示。

输入“Project name”以及“Top level package name”并在“Packaging”中选择 WAR 包的方式。单击“Next”按钮，在下一个界面中单击“Finish”按钮。现在，工程应该已经创建完成了并且应该也能看到 Roo 的 Shell 窗口，如图 9-4 所示。

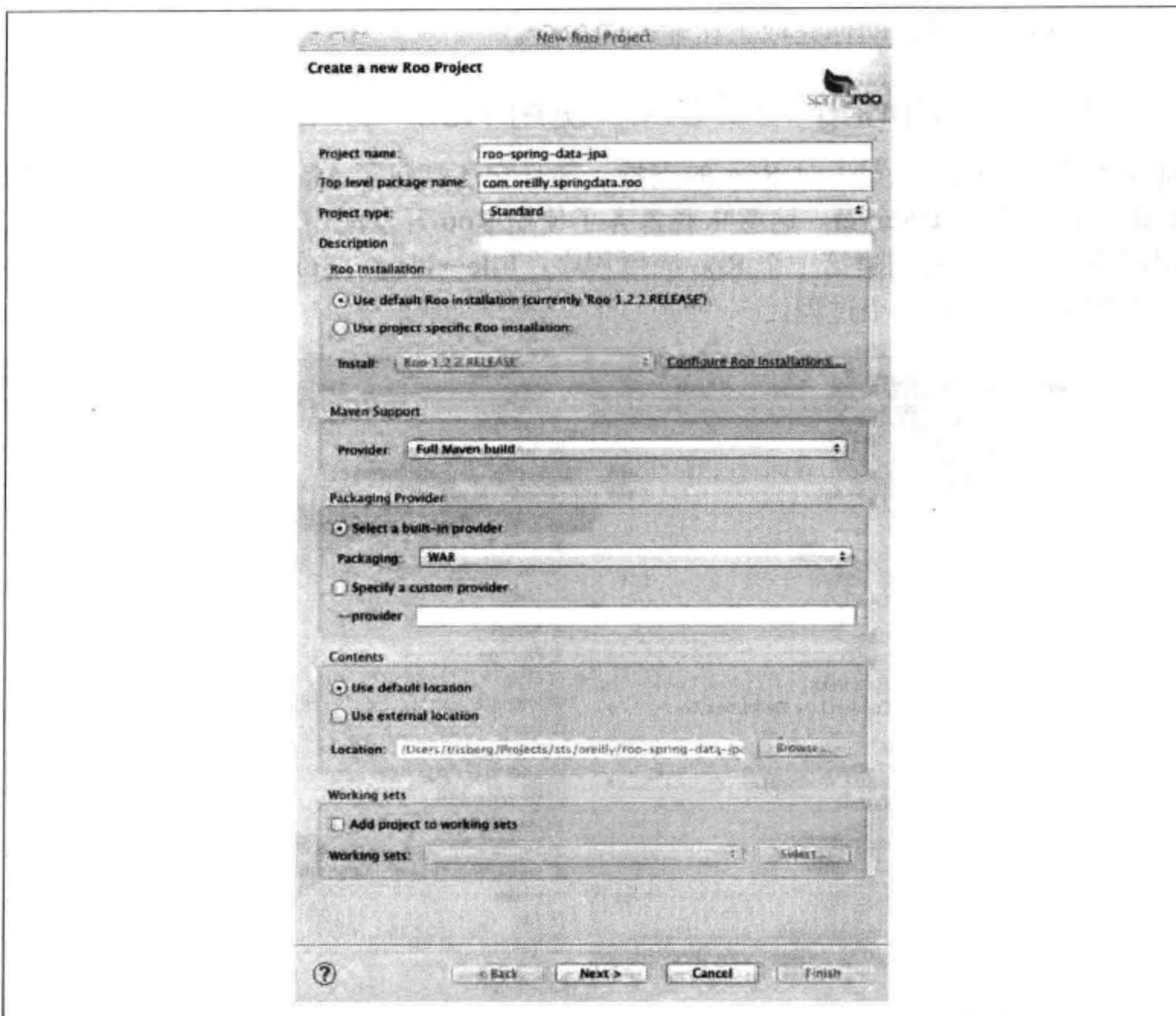


图 9-3 创建 Spring Roo 工程——新工程对话框

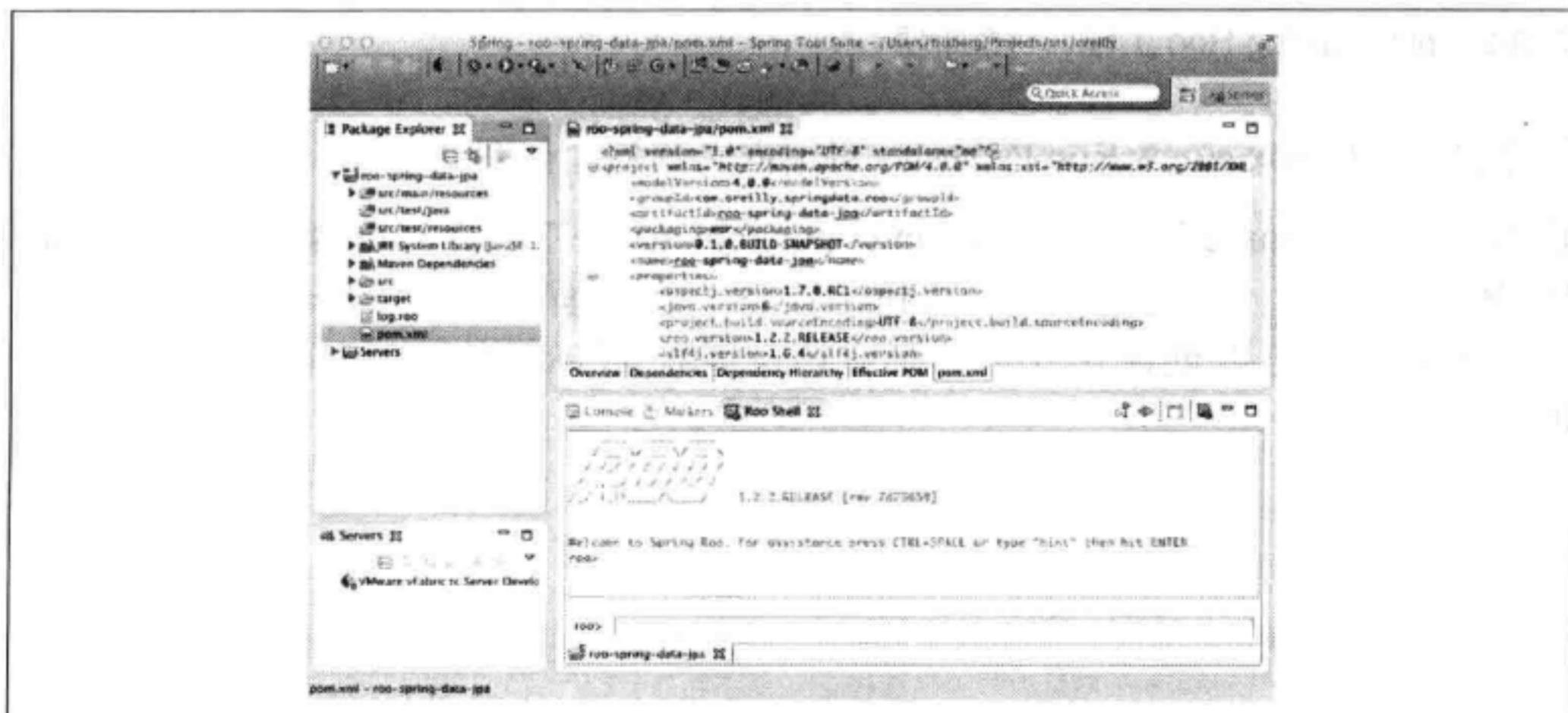


图 9-4 创建 Spring Roo 工程——带有 Roo Shell 的新工程

## 9.4 Spring Roo JPA Repository 示例

现在，已经可以构建第一个 Roo 工程了。我们将基于前面章节所看到的相同的领域模型构建一个客户服务应用。我们会创建 Customer 类和关联的 Address 类，将其关联起来并创建 Repository 以及非常基础的数据访问界面。因为 Roo 的 Repository 支持 JPA 和 MongoDB，在这里它借助了 Spring Data Repository 的支持，所以我们会分别为其创建应用。你会看到，它们基本是相同的，但是也有些许的不同，我们会进行重点介绍。让我们先从 JPA 应用开始吧。

### 9.4.1 创建工程

如果使用 Spring Tool Suite 的话，那么按照之前的介绍创建新的 Spring Roo 工程。在“Create a new Roo Project”对话框界面中，进行以下设置。

- Project name: roo-spring-data-jpa
- Top level package name: com.oreilly.springdata.roo
- Packaging: WAR

如果使用命令行 Roo Shell 的话，需要创建一个名为 *roo-spring-data-jpa* 的目录，切换到这个新的目录后，可以使用刚刚介绍过的方法启动 Roo shell。在 roo>提示符中，输入如下的命令：

```
project --topLevelPackage com.oreilly.springdata.roo ↵
-- projectName roo-spring-data-jpa --java 6 --packaging WAR
```

现在创建完一个新的工程，可以开发应用程序了。从现在开始，不管是通过命令行还是在 Spring Tool Suite 中使用 Roo Shell，所采用的步骤都是一样的。

### 9.4.2 搭建 JPA 持久化

搭建 JPA 持久化配置包括选择 JPA 供应商以及数据库。在这个例子中使用 Hibernate 以及 HSQLDB。在 roo>提示符中，输入如下的命令：

```
jpa setup --provider HIBERNATE --database HYPERSONIC_PERSISTENT
```



需要记住的是，当输入这些命令的时候，可以单击 Tab 键来自动完成并获取可选项的建议。如果使用 Spring Tool Suite 的话，对应的要输入 Ctrl+Space 进行替代。

### 9.4.3 创建实体

让我们从 Address 类开始创建实体：

```
entity jpa --class ~.domain.Address --activeRecord false
field string --fieldName street --notNull
field string --fieldName city --notNull
field string --fieldName country --notNull
```

这并不太难。注意我们声明了--activeRecord false，这意味着需要通过 Repository 来提供 CRUD 功能。生成的 Address 类如下所示：

```
package com.oreilly.springdata.roo.domain;

import javax.validation.constraints.NotNull;
import org.springframework.roo.addon.javabean.RooJavaBean;
import org.springframework.roo.addon.jpa.entity.RooJpaEntity;
import org.springframework.roo.addon.tostring.RooToString;

@RooJavaBean
@RooToString
@RooJpaEntity
public class Address {

    @NotNull
    private String street;

    @NotNull
    private String city;

    @NotNull
    private String country;
}
```

我们看到了所声明的私有域以及 3 个 Roo 注解：@RooJavaBean、@RooToString 和 @RooJpaEntity。这些注解都会有对应的 AspectJ 方面的文件，可以在 Java 类的相同目录中找到它们。

接下来，创建 EmailAddress 和 Customer 类。EmailAddress 是嵌入式的类，它具有一个 value 域。我们需要 Roo 忽略一点，那就是对于一些 SQL 数据库来说 value 是保留字。我们也提供了一个列名 email，这样对于查看数据库表的人来说它更具有描述性。通过在域声明的时候使用 embeddable，我们将其指定为嵌入式的域。

```
embeddable --class ~.domain.EmailAddress
field string --fieldName value --notNull --column email --permitReservedWords
entity jpa --class ~.domain.Customer --activeRecord false
field string --fieldName firstName --notNull
field string --fieldName lastName --notNull
field embedded --fieldName emailAddress --type ~.domain.EmailAddress
field set --fieldName addresses --type ~.domain.Address
```

最后一条命令为 address 表创建了多对多关联，这样我们就能为每个 customer 提供多个 address。以下是所形成的 Customer 类：

```
package com.oreilly.springdata.roo.domain;

import java.util.HashSet;
import java.util.Set;
```

```

import javax.persistence.CascadeType;
import javax.persistence.Embedded;
import javax.persistence.ManyToMany;
import javax.validation.constraints.NotNull;
import org.springframework.roo.addon.javabean.RooJavaBean;
import org.springframework.roo.addon.jpa.entity.RooJpaEntity;
import org.springframework.roo.addon.tostring.RooToString;

@RooJavaBean
@RooToString
@RooJpaEntity
public class Customer {

    @NotNull
    private String firstName;

    @NotNull
    private String lastName;

    @Embedded
    private EmailAddress emailAddress;
    @ManyToMany(cascade = CascadeType.ALL)
    private Set<Address> addresses = new HashSet<Address>();
}

```

为了完整起见，我们也给出了 EmailAddress 类：

```

package com.oreilly.springdata.roo.domain;

import javax.persistence.Column;
import javax.persistence.Embeddable;
import javax.validation.constraints.NotNull;
import org.springframework.roo.addon.javabean.RooJavaBean;
import org.springframework.roo.addon.tostring.RooToString;

@RooJavaBean
@RooToString
@Embeddable
public class EmailAddress {

    @NotNull
    @Column(name = "email")
    private String value;
}

```

这个类最有意思的部分在于@Embeddable注解以及定义了value属性存储在数据库中的列名为email。

#### 9.4.4 定义 Repository

实体就绪之后，就可以创建 JPA Repository 了。我们为每个实体分别创建 Repository。

```

repository jpa --interface ~.repository.CustomerRepository --entity ~.domain.Customer
repository jpa --interface ~.repository.AddressRepository --entity ~.domain.Address

```

此时，也可以创建一个服务层，但鉴于这是一个很简单的应用程序，所以这个步骤省略了。

## 9.4.5 创建 Web 层

现在，需要一些简单的 Web 页面，这样才能够输入和修改 customer 和 address 数据。我们会使用 Roo 生成界面。

```
web mvc setup
web mvc scaffold --class ~.web.CustomerController --backingType ~.domain.Customer
web mvc scaffold --class ~.web.AddressController --backingType ~.domain.Address
```

还有一件事情需要去做。对于 EmailAddress 类来说，在 Web 页面上用于展现时，使用的是 String 类型，在持久化的时候使用的是 EmailAddress 类型，Roo 并不知道如何进行匹配。我们需要为 Roo 生成的 ApplicationConversionServiceFactoryBean 添加一个转换器。示例 9-3 展现了如何实现。

示例 9-3 生成的 ApplicationConversionServiceFactoryBean，它添加了转换器

```
package com.oreilly.springdata.roo.web;

import org.springframework.core.convert.converter.Converter;
import org.springframework.format.FormatterRegistry;
import org.springframework.format.support.FormattingConversionServiceFactoryBean;
import org.springframework.roo.addon.web.mvc.controller.converter.RooConversionService;

import com.oreilly.springdata.roo.domain.EmailAddress;

/**
 * A central place to register application converters and formatters.
 */
@RooConversionService
public class ApplicationConversionServiceFactoryBean
    extends FormattingConversionServiceFactoryBean {

    @Override
    protected void installFormatters(FormatterRegistry registry) {
        super.installFormatters(registry);
        // Register application converters and formatters
        registry.addConverter(getStringToEmailAddressConverter());
        registry.addConverter(getEmailAddressConverterToString());
    }

    public Converter<String, EmailAddress> getStringToEmailAddressConverter() {
        return new Converter<String, EmailAddress>() {
            @Override
            public EmailAddress convert(String source) {
                EmailAddress emailAddress = new EmailAddress();
                emailAddress.setAddress(source);
                return emailAddress;
            }
        };
    }
}
```

```
    @Override  
    public String convert(EmailAddress source) {  
        return source.getAddress();  
    }  
};  
}  
}
```

## 9.4.6 运行示例

现在，可以构建和部署应用了。对于 Spring Tool Suite 来说，只需将应用程序拖曳到 tc server 实例上并启动服务器即可。使用命令行的话，退出 Roo Shell，然后在命令行中运行如下的 Maven 命令：

```
mvn clean package  
mvn tomcat:run
```

现在打开一个浏览器并导航到 <http://localhost:8080/roo-spring-data-jpa/>，可以看到如图 9-5 所示的界面。

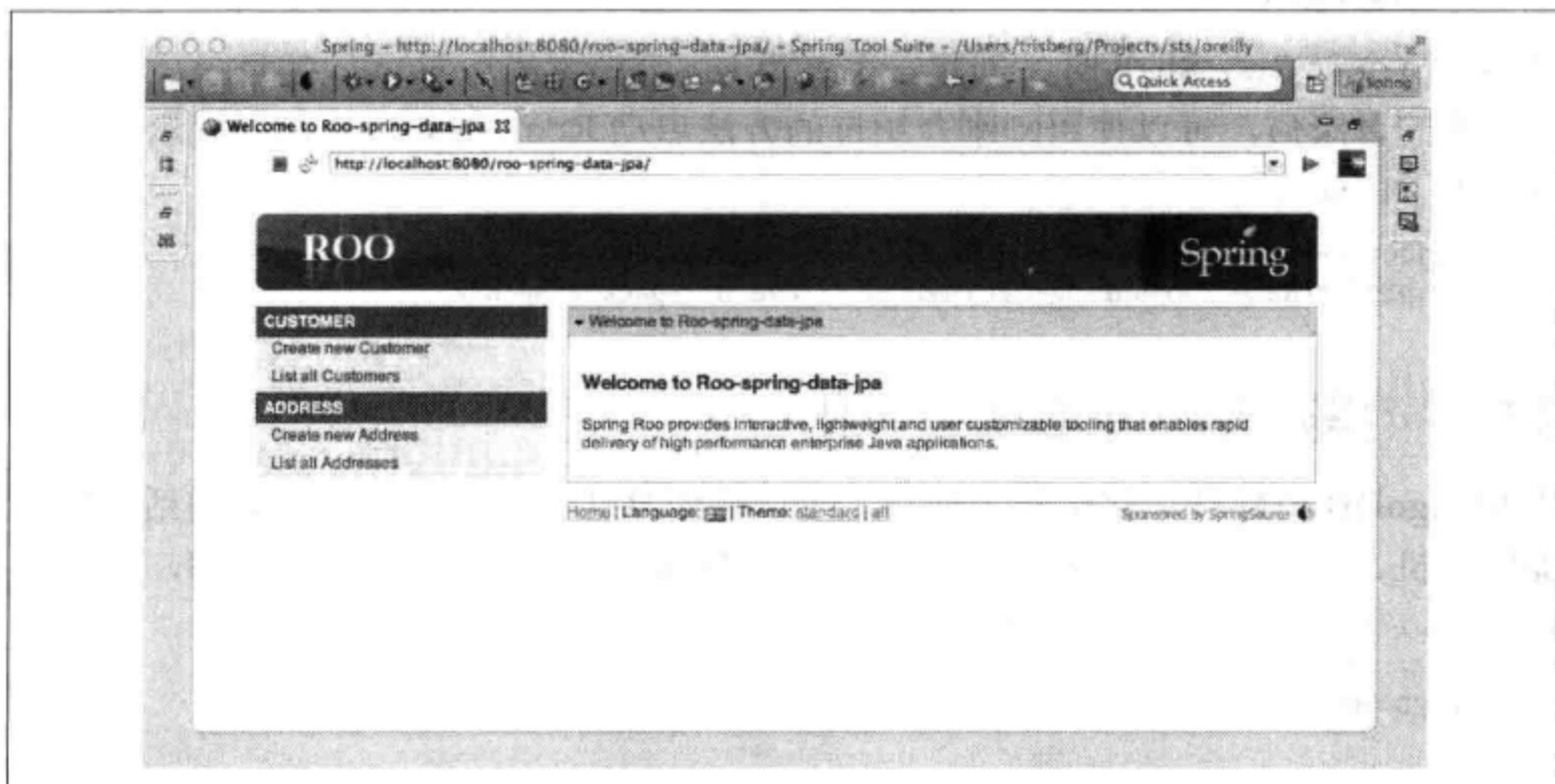


图 9-5 JPA 应用程序

我们的应用已经完成了，现在可以添加一些 Address 并且增加一两个 customer 了。



如果每次重启应用服务器都丢失数据让你感到厌烦的话，可以在 `src/main/resources/META-INF/persistence.xml` 文件中修改模式创建属性，将 `<property name = "hibernate.hbm2ddl.auto" value = "create" />` 的值修改为 “update”

## 9.5 Spring MongoDB JPA Repository 的例子

因为 Spring Data 中包含了对 MongoDB Repository 的支持，所以当使用 Roo 的时候，我们可以将 MongoDB 作为可选的持久化方案。对于 MongoDB 来说，不能采用活动记录风格的持久层，只能使用 Repository。除了这一点区别之外，使用的过程与 JPA 方案基本相同。

### 9.5.1 创建工程

如果使用 Spring Tool Suite，那么按照之前介绍的步骤来创建新的 Spring Roo 工程。在“Create a new Roo Project”对话框界面中，进行以下设置。

- Project name: roo-spring-data-mongo
- Top level package name: com.oreilly.springdata.roo
- Packaging: WAR

当使用命令行 Roo Shell 时，创建一个名为 *roo-spring-data-mongo* 的目录。切换到这个新的目录后，可以使用刚刚介绍过的方法启动 Roo Shell。在 roo>提示符中，输入如下的命令：

```
project --topLevelPackage com.oreilly.springdata.roo
-- projectName roo-spring-data-mongo --java 6 --packaging WAR
```

### 9.5.2 搭建 MongoDB 持久化

为 MongoDB 搭建持久化配置很简单。可以接受默认设置。如果你愿意的话，可以提供主机、端口、用户名和密码，但是对于默认的本地安装的 MongoDB，默认设置就可以了。所以，输入以下命令即可：

```
mongo setup
```

### 9.5.3 创建实体

当创建实体的时候，没有使用活动记录风格的可选项了，在 JPA 方案中，我们需要提供--activeRecord 参数以屏蔽掉这种方案。现在我们没有必要这样做了，对于 MongoDB 来说，Repository 是默认和唯一的持久层方案。同样，我们从 Address 类开始：

```
entity mongo --class ~.domain.Address
field string --fieldName street --notNull
field string --fieldName city --notNull
field string --fieldName country --notNull
```

这看起来与 JPA 的例子很相似。当再看 Customer 类时，你所能发现的第一个区别可能是使用 MongoDB 时，没有使用可嵌入的类，它只对 JPA 有效。在 MongoDB 中，只需创建一个简单的类并指定--rooAnnotations 为 true 以启用@RooJavaBean 的支持。为了使用这个类，需要将域设置为 other。除了这些小的差别，实体声明与 JPA 示例非常类似：

```
class --class ~.domain.EmailAddress --rooAnnotations true
field string --fieldName value --notNull --permitReservedWords
entity mongo --class ~.domain.Customer
field string --fieldName firstName --notNull
field string --fieldName lastName --notNull
field other --fieldName emailAddress --type ~.domain.EmailAddress
field set --fieldName addresses --type ~.domain.Address
```

## 9.5.4 定义 Repository

声明 MongoDB Repository 的方式与 JPA Repository 方式相同，唯一的区别在于 mongo 关键字：

```
repository mongo --interface ~.repository.CustomerRepository --
  --entity ~.domain.Customer
repository mongo --interface ~.repository.AddressRepository --entity ~.domain.Address
```

## 9.5.5 创建 Web 层

Web 层的创建与 JPA 示例完全一样：

```
web mvc setup
web mvc scaffold --class ~.web.CustomerController --backingType ~.domain.Customer
web mvc scaffold --class ~.web.AddressController --backingType ~.domain.Address
```

不要忘记为 ApplicationConversionServiceFactoryBean 添加转换器，就像在示例 9-3 中为 JPA 所做的那样。

## 9.5.6 运行示例

现在，可以构建和部署示例了。同样与 JPA 示例一样，只不过需要系统中有正在运行的 MongoDB。关于 MongoDB 的安装和运行，可以参阅本书的第 6 章。

对于 Spring Tool Suite，只需将应用拖曳到 tc server 实例上并启动服务器即可。使用命令行的话，退出 Roo Shell 并在命令行中运行如下的 Maven 命令：

```
mvn clean package
mvn tomcat:run
```

现在打开浏览器并导航到 <http://localhost:8080/roo-spring-data-mongo/>，可以看到如图 9-6 所示的界面。

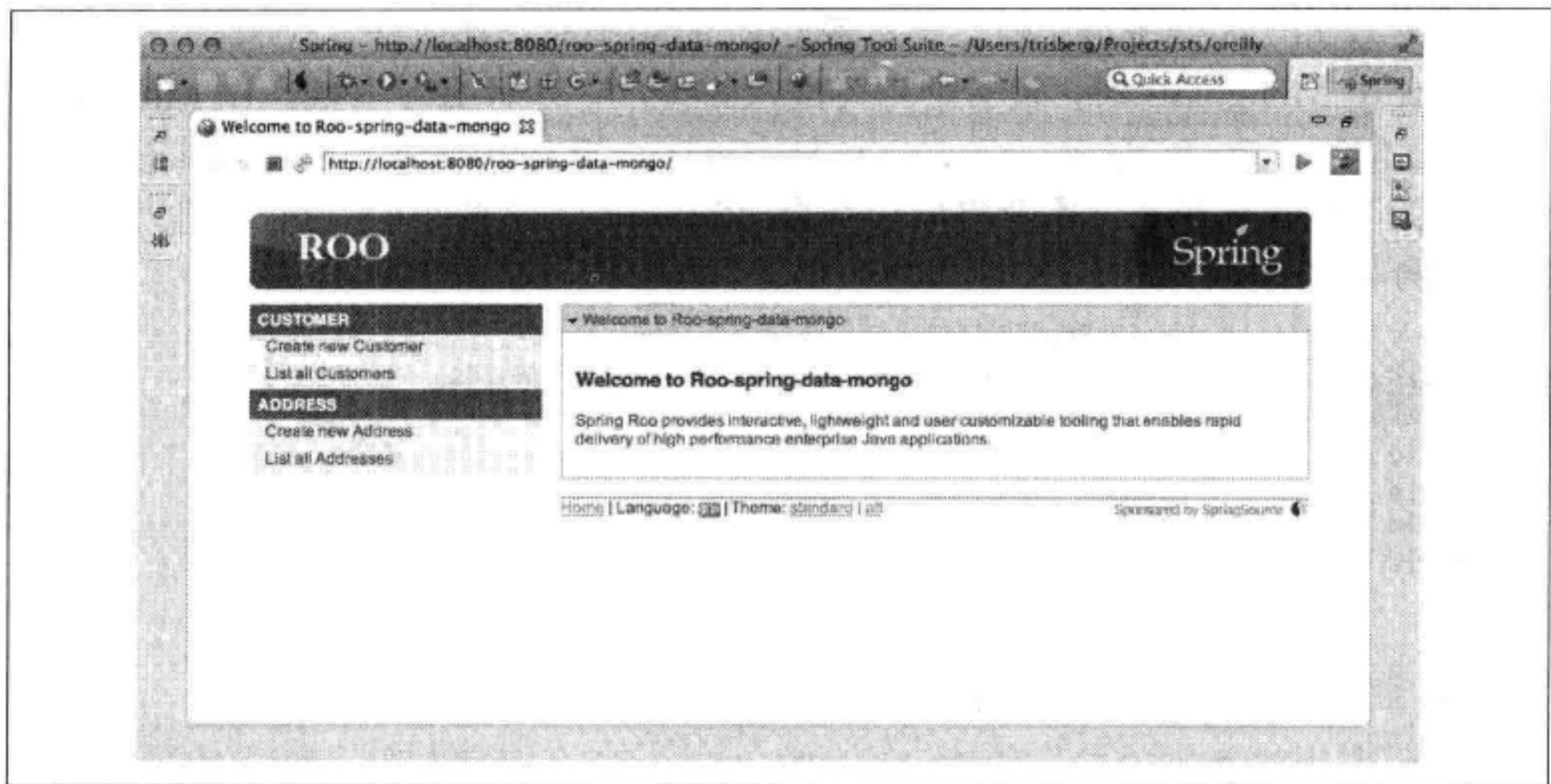


图 9-6 MongoDB 应用程序

我们的第二个示例应用已经完成了，现在可以添加一些 address 并加入一两个 customer 了。

# REST Repository 导出器

使用 Spring Data Repository 抽象（详见第 2 章）的时候，管理实体的 Repository 接口成为访问实体的核心。使用 Spring Data REST Repository 导出器（exporter）项目，现在可以通过 REST Web 服务导出（顾名思义）这些被管理的实体，从而很容易地与数据进行交互。导出器机制会暴露每个 Repository 的导出资源、匹配该资源的 CRUD 操作到 HTTP 方法并且提供了一种方式来执行 Repository 接口所暴露的查询方法。

### REST 是什么？

表述性状态转移（Representational State Transfer, REST）是一种架构风格，最早由 Roy Fielding 在他的论文《架构风格与基于网络的软件架构设计》[Fielding00] 中进行了描述。它对 HTTP 协议背后的理念进行了概括，它基于此衍生出了以下几个核心理念。

#### 资源（Resource）

系统暴露资源给其他的系统：订单、客户等。

#### 标识符（Identifier）

这些资源可以通过标识符进行寻址。在 HTTP 的领域中，这些标识符就是 URL。

#### 动词（Verb）

每个资源可以通过一组定义好的动词进行访问和操作。这些动词具有专用的语义并且必须要据此进行使用。在 HTTP 中，通用的动词是 GET、PUT、POST、DELETE、HEAD 以及 OPTIONS，还有很少用到的（甚至从不会使用的）TRACE 和 CONNECT。并不是每个资源都需要支持上面列出的所有动词，但是有一点是需要做到的，那就是不能为每个资源设置特殊的动词。

## 表述 (Representation)

客户端不会直接与资源进行交互，而是通过其表述进行交互。表述要通过媒体类型进行定义，它明确标识了表述的结构。常见的媒体类型有很通用的，如 application/xml 和 application/json，也有更为结构化的，如 application/atom+xml。

## 超媒体 (Hypermedia)

资源的表述中通常会包含指向其他资源的链接，这样就允许客户端基于资源的状态和所提供的链接对系统进行导航。这种理念被称为用作应用程序状态引擎的超媒体（Hypermedia as the Engine of Application State，HATEOAS）。

事实已经证明，基于这些理念所构建的 Web 服务具有可伸缩性、可靠性以及可进化的能力。这也是为什么在集成软件系统时，REST Web 服务成为了一种普遍采用的方式。尽管 Fielding 的论文非常易读，但我们还是推荐 Jim Webber、SavasParastatidis 和 Ian Robinson 所著的《*REST in Practice*》。关于这个话题，这本书提供了广泛且详细的介绍，这些介绍是以现实世界中的样例作为驱动的 [WePaRo10]。

通过示例工程中的 rest 模块，我们将会以导览的方式介绍这些功能。它是兼容于 Servlet 3.0 规范且基于 Spring 的 Web 应用。这个工程最重要的依赖是 *spring-data-rest-webmvc* 库，它提供了对 Spring MVC 的集成，因此可以将 Spring Data JPA Repository 导出到 Web 中。目前，它只能用于以 JPA 作为后端的 Repository，但是对其他存储形式的支持已经提到路线图之上了。这个工程的基础 Spring 设施与第 4 章非常类似，如果没有看过第 4 章的话，请阅读一下此章以了解基本的知识。

## 10.1 示例工程

运行示例应用的最简单方式是通过命令行使用 Maven 的 Jetty 插件。Jetty 是一个很小的 servlet 容器，可以运行 Servlet 3.0 的 Web 应用。Maven 插件可以从模块目录中启动应用，所使用的命令如示例 10-1 所示。

### 示例 10-1 从命令行运行示例应用

```
$ mvn jetty:run -Dspring.profiles.active=with-data

[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building Spring Data Book - REST exporter 1.0.0.BUILD-SNAPSHOT
[INFO] -----
[INFO] <<< jetty-maven-plugin:8.1.5.v20120716:run (default-cli) @ spring-data-book-rest <<<
[INFO] --- jetty-maven-plugin:8.1.5.v20120716:run (default-cli) @ spring-data-book-rest ---
```

```

[INFO] Configuring Jetty for project: Spring Data Book - REST exporter
[INFO] webAppSourceDirectory .../spring-data-book/rest/src/main/webapp does not exist. \
      Defaulting to .../spring-data-book/rest/src/main/webapp
[INFO] Reload Mechanic: automatic
[INFO] Classes = .../spring-data-book/rest/target/classes
[INFO] Context path = /
[INFO] Tmp directory = .../spring-data-book/rest/target/tmp
[INFO] Web defaults = org/eclipse/jetty/webapp/webdefault.xml
[INFO] Web overrides = none
[INFO] web.xml file = null
[INFO] Webapp directory = .../spring-data-book/rest/src/main/webapp
2012-07-31 17:58:01.709:INFO:oejs.Server:jetty-8.1.5.v20120716
2012-07-31 17:58:03.769:INFO:oejpw.PlusConfiguration:No Transaction manager found \
    - if your webapp requires one, please configure one.
2012-07-31 17:58:10.641:INFO:/:Spring WebApplicationInitializers detected on classpath: \
    [com.oreilly.springdata.rest.RestWebApplicationInitializer@34cbcc24]

...
2012-07-31 17:58:16,032  INFO est.webmvc.RepositoryRestExporterServlet: 444 - \
    FrameworkServlet 'dispatcher': initialization started

...
2012-07-31 17:58:17,159  INFO est.webmvc.RepositoryRestExporterServlet: 463 - \
    FrameworkServlet 'dispatcher': initialization completed in 1121 ms
2012-07-31 17:58:17.179:INFO:oejs.AbstractConnector:Started SelectChannelConnector@ \
    0.0.0.0:8080
[INFO] Started Jetty Server

```

在这里需要注意的第一件事就是在执行命令中传入了 JVM 参数，也就是 `spring.profiles.active`。这会为内存数据库填充一些示例产品、客户以及订单，这样我们就会有一些用于进行交互的实际数据了。Maven 将一些通用的活动信息打印到了控制台上。下一个比较有意思的行在 17:58:10 那一行，它告诉我们 Jetty 已经发现了 `WebApplicationInitializer`——具体来说也就是我们的 `RestWebApplicationInitializer`。`WebApplicationInitializer` 是等同于 `web.xml` 文件的 API，是在 Servlet API 3.0 中引入的。它可以让我们摆脱基于 XML 的方式来配置 Web 应用的基础设施，取而代之的是使用 API。我们的实现如示例 10-2 所示。

## 示例 10-2 RestWebApplicationInitializer

```

public class RestWebApplicationInitializer implements WebApplicationInitializer {

    public void onStartup(ServletContext container) throws ServletException {
        // Create the 'root' Spring application context
        AnnotationConfigWebApplicationContext rootContext =
            new AnnotationConfigWebApplicationContext();
        rootContext.register(ApplicationConfig.class);

        // Manage the life cycle of the root application context
        container.addListener(new ContextLoaderListener(rootContext));

        // Register and map the dispatcher servlet
        DispatcherServlet servlet = new RepositoryRestExporterServlet();
        ServletRegistration.Dynamic dispatcher = container.addServlet("dispatcher", servlet);
        dispatcher.setLoadOnStartup(1);
        dispatcher.addMapping("/");
    }
}

```

```
}
```

首先，搭建 AnnotationConfigWebApplicationContext 并注册了 ApplicationConfigJavaConfig 类，它稍后会作为 Spring 配置来使用。我们将包装到 ContextLoaderListener 之中的 ApplicationContext 注册到实际的 ServletContext 里面。这个上下文稍后会触发监听器，这将会启动 ApplicationContext。到目前为止，这段代码与在 *web.xml* 文件中注册一个 ContextLoaderListener 并指向一个 XML 配置文件是相同的，只不过不用处理 XML 以及基于 String 的文件地址，它是以类型安全的方式引用了配置。所配置的 ApplicationContext 现在会启动嵌入式的数据库、包含事务管理器在内的 JPA 基础设施并最终启用 Repository。这个过程已经在 4.3 小节“启动示例代码”中进行了介绍。

稍后，声明了一个 RepositoryRestExporterServlet，它负责真正有意思的部分。它会注册很多的 Spring MVC 基础设施组件并会探查到用于 Spring Data Repository 实例的根应用上下文，并会为那些实现了 CrudRepository 接口的所有 Repository 暴露 HTTP 资源。目前来说这是个限制，这个模块的后续版本会将其移除。我们将 servlet 映射到 servlet 根上，这样这个应用就可以通过 <http://localhost:8080> 进行访问了。

### 10.1.1 与 Rest 导出器进行交互

现在已经启动了应用，来看一下实际上是怎样使用它的。使用命令行工具 curl 来与系统进行交互，因为它提供了一种便利的方式来触发 HTTP 请求并且它显示响应的方式非常适合在书中进行展现。当然，可以使用其他能够触发 HTTP 请求的客户端：命令行工具（如 Windows 下的 wget）或者直接使用所选择 Web 浏览器。要注意的是，后者只允许通过 URL 地址栏触发 GET 请求。如果想使用更为高级的请求（POST、PUT、DELETE 等），建议使用浏览器插件，如针对 Google Chrome 的 Dev HTTP Client (<http://bit.ly/PZ5lCt>)。对于其他的浏览器来说，也有类似的工具可供使用。

让我们触发一些对这个应用的请求，如示例 10-3 所示。目前我们所知的就是其部署为监听 <http://localhost:8080>，因此看一下它实际提供的资源是什么。

#### 示例 10-3 使用 curl 触发初始请求

```
$ curl -v http://localhost:8080
* About to connect() to localhost port 8080 (#0)
* Trying ::1... connected
* Connected to localhost (::1) port 8080 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.21.4 (universal-apple-darwin11.0) libcurl/7.21.4 OpenSSL/0.9.8r
  zlib/1.2.5
> Host: localhost:8080
> Accept: */*
>
```

```
< HTTP/1.1 200 OK
< Content-Length: 242
< Content-Type: application/json
< Server: Jetty(8.1.5.v20120716)
<
{
  "links" : [ {
    "rel" : "product",
    "href" : "http://localhost:8080/product"
  }, {
    "rel" : "order",
    "href" : "http://localhost:8080/order"
  }, {
    "rel" : "customer",
    "href" : "http://localhost:8080/customer"
  } ]
}
* Connection #0 to host localhost left intact
* Closing connection #0
```

这里要注意的第一件事就是触发 curl 命令时使用了-v 标记。这个标记会激活详细输出，列出所有的请求和响应头以及实际的响应数据。可以看到，默认情况下，服务器所返回数据的内容类型为 application/json。在真正的响应体中包含了一个链接的集合，可以按照它来对应用进行查看。它所提供的每个链接实际上都是由 ApplicationContext 中可用的 Spring Data Repository 衍生而来。我们具有 CustomerRepository、ProductRepository 以及 OrderRepository，因此关系类型 (rel) 属性就是 customer、product 和 order (Repository 名字的前半部分，并且首字符要小写)。资源的 URL 也使用默认规则衍生而来。如果要自定义这种行为，可以在 Repository 接口上使用@RestResource 注解，它允许你明确定义 path (URI 部分) 以及 rel (关系类型)。

### 链接 (Link)

链接的表述通常衍生自 Atom RFC (<http://tools.ietf.org/html/rfc4287>) 中所定义的链接元素。基本上来讲，它包含了两个属性：关系类型 (rel) 和超文本引用 (href)。前者定义了链接的实际语义（因此需要文档化和标准化），而后者实际上对客户端是不透明的。通常客户端会探查链接的响应体以获取关系类型，并访问它所感兴趣的关系类型和链接。所以，一般来讲，客户端会知道在链接后添加 order 类型的 rel 就能获取所有的订单。这种结构会使得服务器和客户端实现解耦，因为服务端会告知客户端去哪里获取数据。如果 URL 会发生变化或者服务器希望客户端指向不同的机器以实现请求的负载均衡，这就会特别有用。

继续来查看系统中可用的产品。我们知道产品会通过关系类型 product 来进行暴露，因此我们访问带有这个 rel 的链接。

## 10.1.2 访问 Product

示例 10-4 展现了如何访问系统中所有可用的产品。

### 示例 10-4 访问产品

```
$ curl http://localhost:8080/product
```

```
{ "content" : [ {
    "price" : 499.00,
    "description" : "Apple tablet device",
    "name" : "iPad",
    "links" : [ {
        "rel" : "self",
        "href" : "http://localhost:8080/product/1"
    } ],
    "attributes" : {
        "connector" : "socket"
    }
}, ... , {
    "price" : 49.00,
    "description" : "Dock for iPhone/iPad",
    "name" : "Dock",
    "links" : [ {
        "rel" : "self",
        "href" : "http://localhost:8080/product/3"
    } ],
    "attributes" : {
        "connector" : "plug"
    }
],
"links" : [ {
    "rel" : "product.search",
    "href" : "http://localhost:8080/product/search"
} ]
}
```

触发这个访问所有产品的请求会返回 JSON 表述，它包含了两个主要的域。域 `content` 包含了所有可用的产品的集合并且直接在响应中进行展现。每个元素中包含了 `Product` 类的序列化属性以及一个非原生的 `links` 容器。这个容器包含了一个链接，其关系类型为 `self`。`self` 类型通常作为一种标识符，因为它指向了资源本身。所以，可以根据表述中具有 `self` 关系类型的链接直接访问 `iPad` 产品，如示例 10-5 所示。

### 示例 10-5 访问单个产品

```
$ curl http://localhost:8080/product/1
```

```
{ "price" : 499.00,
  "description" : "Apple tablet device",
  "name" : "iPad",
  "links" : [ {
    "rel" : "self",
    "href" : "http://localhost:8080/product/1"
  } ],
  "attributes" : {
```

```
        "connector" : "socket"
    }
}
```

要更新一个产品，只需要对这个资源发送一个 PUT 请求并提供新的内容即可，如示例 10-6 所示。

### 示例 10-6 更新一个产品

```
$ curl -v -X PUT -H "Content-Type: application/json" \
      -d '{ "price" : 469.00, \
            "name" : "Apple iPad" }' \
      http://localhost:8080/spring-data-book-rest/product/1

* About to connect() to localhost port 8080 (#0)
* Trying ::1...
* connected
* Connected to localhost (::1) port 8080 (#0)
> PUT /spring-data-book-rest/product/1 HTTP/1.1
> User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r
  zlib/1.2.5
> Host: localhost:8080
> Accept: */*
> Content-Type: application/json
> Content-Length: 82
>
* upload completely sent off: 82 out of 82 bytes
< HTTP/1.1 204 No Content
< Server: Apache-Coyote/1.1
< Date: Fri, 31 Aug 2012 10:23:58 GMT
```

通过使用-X 参数将 HTTP 方法设置为 PUT 并且提供了 Content-Type 头信息以表明我们所发送的是 JSON。通过-d 参数提交了更新后的 price 和 name 属性。服务端返回 204 No Content 表明请求已经成功了。对这个产品的 URL 再次发起一个 GET 请求会返回更新后的内容，如示例 10-7 所示。

### 示例 10-7 更新后的产品

```
$ curl http://localhost:8080/spring-data-book-rest/product/1

{
  "links" : [
    {
      "rel" : "self",
      "href" : "http://localhost:8080/spring-data-book-rest"
    }
  ],
  "price" : 469.00,
  "description" : "Apple tablet device",
  "name" : "Apple iPad",
  "attributes" : {
    "connector" : "socket"
  }
}
```

资源集合的 JSON 表达中还包含了 links 属性，它指向了一个通用的资源，据此我们可以探索 Repository 所暴露的查询方法。根据约定要使用集合资源的关系类型（在我们的例子中，也就是 product）再加上.search。让我们访问这个链接并查看一

下实际上可执行的查询是什么，如示例 10-8 所示。

### 示例 10-8 访问 Product 可用的搜索功能

```
$ curl http://localhost:8080/product/search

{
  "links": [
    {
      "rel": "product.findByDescriptionContaining",
      "href": "http://localhost:8080/product/search/findByDescriptionContaining"
    },
    {
      "rel": "product.findByAttributeAndValue",
      "href": "http://localhost:8080/product/search/findByAttributeAndValue"
    }
  ]
}
```

可以看到，Repository 导出器为 ProductRepository 接口中所声明的每一个查询方法均暴露了一个资源。同样的，这里的关系类型模式基于资源的关系类型并且要再加上查询方法的名字，但是我们可以通过在查询方法上使用@RestResource 注解来进行自定义。令人遗憾的是，JVM 并不支持从接口方法上得到参数名，所以我们必须要在查询方法的参数上使用@Param，并且在 findByAttributeAndValue(...)的方法定义上，对手动定义的查询方法要使用命名参数，如示例 10-9 所示。

### 示例 10-9 ProductRepository 接口

```
public interface ProductRepository extends CrudRepository<Product, Long> {

  Page<Product> findByDescriptionContaining(
    @Param("description") String description, Pageable pageable);

  @Query("select p from Product p where p.attributes[:attribute] = :value")
  List<Product> findByAttributeAndValue(
    @Param("attribute") String attribute, @Param("value") String value);
}
```

现在，按照 product.findByAttributeAndValue 链接，发送带有匹配参数的 GET 请求到服务器来触发第二个查询方法。搜索一下 connector 属性为 plug 的产品，如示例 10-10 所示。

### 示例 10-10 搜索 connector 属性为 plug 的产品

```
$ curl http://localhost:8080/product/search/findByAttributeAndValue?attribute=connector\
  /&value=plug
```

```
{
  "results": [
    {
      "price": 49.00,
      "description": "Dock for iPhone/iPad",
      "name": "Dock",
      "_links": [
        {
          "rel": "self",
          "href": "http://localhost:8080/product/3"
        }
      ],
      "attributes": {
        "connector": "plug"
      }
    }
  ],
}
```

```
        "links" : [ ... ]  
    }
```

### 10.1.3 访问 Customer

我们已经看到了如何对可用的产品进行导航以及如何执行 Repository 接口所暴露的查找方法，现在让我们调转方向看一下系统中的注册用户。我们初始对 `http://localhost:8080` 的请求暴露了一个 `customer` 连接，如示例 10-3 所示。在示例 10-11 中，按照这个链接看看能够得到什么样的 `customer` 结果。

#### 示例 10-11 访问顾客 (1/2)

```
$ curl -v http://localhost:8080/customer  
  
* About to connect() to localhost port 8080 (#0)  
* Trying ::1...  
* connected  
* Connected to localhost (::1) port 8080 (#0)  
> GET /customer HTTP/1.1  
> User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r  
    zlib/1.2.5  
> Host: localhost:8080  
> Accept: */*  
>  
< HTTP/1.1 500 Could not write JSON: No serializer found for class  
    com.oreilly.springdata.rest.core.EmailAddress and no properties  
    discovered to create BeanSerializer ...  
< Content-Type: text/html;charset=ISO-8859-1  
< Cache-Control: must-revalidate,no-cache,no-store  
< Content-Length: 16607  
< Server: Jetty(8.1.5.v20120716)
```

呀，看起来情况并不妙。我们得到的是 500 服务器错误（500 Server Error）的响应，表明处理这个请求的时候出现了错误。终端输出可能更为详尽，但是重要的一行在示例 10-11 中，也就是在 HTTP 状态码下面。Jackson（Spring Data Rest 所采用的 JSON 编组技术）在序列化 `EmailAddress` 值对象时似乎被阻塞住了。这是因为我们并没有暴露任何的 `getter` 和 `setter` 方法，Jackson 要使用这样的方法来寻找要渲染到响应之中的属性。

实际上，我们并不想将 `EmailAddress` 作为嵌入式的对象来进行渲染，而是将其作为简单的 `String` 值。可以通过使用 Jackson 提供的 `@JsonSerialize` 注解来自定义渲染以实现这一点。我们将其 `using` 属性配置为预先定义的 `ToStringSerializer.class`，它会调用这个对象的 `toString()` 方法来进行渲染。

那么，让我们再试一次，如示例 10-12 所示。

#### 示例 10-12 访问顾客 (2/2)

```
$ curl -v http://localhost:8080/customer  
  
* About to connect() to localhost port 8080 (#0)  
* Trying ::1...  
* connected  
* Connected to localhost (::1) port 8080 (#0)
```

```
> GET /customer HTTP/1.1
> User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r
  zlib/1.2.5
> Host: localhost:8080
> Accept: /*
>
< HTTP/1.1 500 Could not write JSON: Infinite recursion (StackOverflowError)
  (through reference chain: com.oreilly.springdata.rest.core.Address["copy"]
   ->com.oreilly.springdata.rest.core.Address["copy"]...)
< Content-Type: text/html;charset=ISO-8859-1
< Cache-Control: must-revalidate,no-cache,no-store
< Content-Length: 622972
< Server: Jetty(8.1.5.v20120716)
```

看起来并没有好到哪里去，但是我们至少又往前进一步。这一次 Jackson 的渲染器提示 Address 类暴露了 copy 属性，这会导致递归。产生这个问题的原因在于 Address 类的 getCopy()方法遵循了 Java Bean 属性的语义，而不是传统意义上的 getter 方法。实际上，它会返回 Address 对象的拷贝版本以便于我们很容易地复制 Address 实例，并将其指定到 Order 之中，从而避免了 Customer 的 Address 所发生的变化会影响到已有的订单（参见示例 4-7）。所以，在这里有两个可选方案：重命名这个方法使其不再匹配 Java Bean 的属性约定或者添加一个注解告知 Jackson 忽视掉这个属性。我们选择了后者，因为不想重构客户端的代码。因此，使用 @JsonIgnore 注解来将 copy 属性排除在渲染的属性之外，如示例 10-13 所示。

### 示例 10-13 将 Address 类的 copy 属性排除在渲染之外

```
@Entity
public class Address extends AbstractEntity {

    private String street, city, country;

    ...
    @JsonIgnore
    public Address getCopy() { ... }
}
```

做完这些改动之后，让我们重启服务器并再次发出请求，如示例 10-14 所示。

### 示例 10-14 在进行编组调整之后访问顾客

```
$ curl http://localhost:8080/customer

{ "results" : [ {
    "links" : [ {
        "rel" : "self",
        "href" : "http://localhost:8080/customer/1"
    } ],
    "lastname" : "Matthews",
    "emailAddress" : "dave@dmbar.com",
    "firstname" : "Dave",
    "addresses" : [ {
        "id" : 1,
        "street" : "27 Broadway",
```

```

        "city" : "New York",
        "country" : "United States"
    }, {
        "id" : 2,
        "street" : "27 Broadway",
        "city" : "New York",
        "country" : "United States"
    } ]
}, [
"links" : [ {
    "rel" : "customer.search",
    "href" : "http://localhost:8080/customer/search"
} ],
"page" {
    "number" : 1,
    "size" : 20,
    "totalPages" : 1,
    "totalElements" : 3
}
}

```

可以看到，实体已经可以正确地进行渲染了。也可以看到预期的 links 区域中指向了顾客类可用的查询方法。不过在这里，与前面的不同在于返回的 JSON 中设置了额外的 page 属性。它包含了目前的页数 (number)、请求的分页大小 (size，这里是默认的 20)、可用的总页数 (totalPages) 以及可用的总元素数量 (totalElements)。

出现这些属性是因为 CustomerRepository 扩展了 PagingAndSortingRepository，因此允许逐页地访问顾客。要了解关于这一点的更多信息，可以参考 2.3 小节“定义 Repository”。这意味着在发起请求的时候，可以使用 page 和 limit 参数来限制返回的顾客数量。由于我们一共有 3 位顾客需要展现，所以可以手动设置每页的数量是一位顾客，如示例 10-15 所示。

### 示例 10-15 访问第一页的顾客信息

```
$ curl http://localhost:8080/customer?limit=1
```

```

{
    "content" : [ ... ],
    "links" : [ {
        "rel" : "customer.next",
        "href" : "http://localhost:8080/customer?page=2&limit=1"
    }, {
        "rel" : "customer.search",
        "href" : "http://localhost:8080/customer/search"
    } ],
    "page" : {
        "number" : 1,
        "size" : 1,
        "totalPages" : 3,
        "totalElements" : 3
    }
}

```

注意，现在得到了元数据信息而且变成了只有一条结果。totalPages 域表明有三页的数据，因为我们所选择的每页大小是 1。更好的一点在于，服务端提示我们可以

按照 `customer.next` 链接来得到下一页的顾客信息。它已经包含了请求第二页数据的请求参数，所以客户端并不需要手动构造 URL。访问这个链接，看一下在结果集中进行导航时元数据是如何发生变化的，如示例 10-16 所示。

#### 示例 10-16 访问第二页的顾客信息

```
$ curl http://localhost:8080/customer?page=2&limit=1

{
    "content": [ ... ],
    "links": [
        {
            "rel": "customer.prev",
            "href": "http://localhost:8080/customer?page=1&limit=1"
        },
        {
            "rel": "customer.next",
            "href": "http://localhost:8080/customer?page=3&limit=1"
        },
        {
            "rel": "customer.search",
            "href": "http://localhost:8080/customer/search"
        }
    ],
    "page": {
        "number": 2,
        "size": 1,
        "totalPages": 3,
        "totalElements": 3
    }
}
```



注意，将示例 10-16 中的 URI 粘贴到控制台时，可能需要对&符进行转义。如果使用专用的 HTTP 客户端，那么就不需要进行转义了。

除了实际返回的数据，注意 `number` 属性提示我们已经到了第二页。除此之外，服务端会探测到有可用的上一页数据并且提供了 `customer.prev` 链接以便导航到那里。按照 `customer.next` 链接进行第二次访问时，所得到的表述中就不会再有 `customer.next` 链接了，因为我们已经到达了可用的最后一页。

#### 10.1.4 访问 Order

最后要探讨的根链接关系就是 `order`。顾名思义，它允许我们访问系统中可用的 Order 信息。支撑这个资源的 Repository 接口是 `OrderRepository`。现在，访问这个资源，看看服务端返回了什么，如示例 10-17 所示。

#### 示例 10-17 访问订单信息

```
$ curl http://localhost:8080/order
```

```
{
    "content": [
        {
            "billingAddress": {
                "id": 2,
                "street": "27 Broadway",
                "city": "New York",
                "country": "United States"
            }
        }
    ]
}
```

```

},
"shippingAddress" : {
  "id" : 2,
  "street" : "27 Broadway",
  "city" : "New York",
  "country" : "United States"
},
"lineItems" : [ ... ]
"links" : [ {
  "rel" : "order.Order.customer",
  "href" : "http://localhost:8080/order/1/customer"
}, {
  "rel" : "self",
  "href" : "http://localhost:8080/order/1"
} ],
"links" : [ {
  "rel" : "order.search",
  "href" : "http://localhost:8080/order/search"
} ],
"page" : {
  "number" : 1,
  "size" : 20,
  "totalPages" : 1,
  "totalElements" : 1
}
}
}

```

响应中包含了许多已经讨论过的我们所熟知的模式，包括指向 OrderRepository 所暴露的查询方法的链接以及嵌套的 content 域，在这个域中包含了序列化的 Order 对象、内联的 Address 对象以及 LineItems。也可以看到分页的元数据信息，因为 OrderRepository 实现了 PagingAndSortingRepository。

在这里要注意的是，Order 对象中所持有的 Customer 实例并没有进行内联显示，而是通过链接指向了它。这是因为 Customer 是通过 Spring Data Repository 来进行管理的。因此它们表现为 Order 的从属资源，从而允许操作它们之间的所属关系。按照这个链接来访问触发这个 Order 的 Customer，如示例 10-18 所示。

### 示例 10-18 访问下订单的顾客信息

```

$ curl http://localhost:8080/order/1/customer

{
  "links" : [ {
    "rel" : "order.Order.customer.Customer",
    "href" : "http://localhost:8080/order/1/customer"
  }, {
    "rel" : "self",
    "href" : "http://localhost:8080/customer/1"
  } ],
  "emailAddress" : "dave@dmbar.com",
  "lastname" : "Matthews",
  "firstname" : "Dave",
  "addresses" : [ {
    "street" : "27 Broadway",

```

```
        "city" : "New York",
        "country" : "United States"
    }, {
        "street" : "27 Broadway",
        "city" : "New York",
        "country" : "United States"
    }
}
```

这个调用返回了关联 Customer 的详细信息并提供了两个链接。具有 `order.Order.customer` 关系类型的链接指向 Customer 的关联资源，而 `self` 链接指向了实际的 Customer 资源。这有什么区别吗？前者展现的是 Customer 与订单之间的分配关系。我们可以通过对这个 URI 提交一个 PUT 请求来修改分配关系，也可以触发一个 DELETE 请求解除这种分配关系。在我们的场景下，DELETE 调用将会产生 405 Method not allowed 的响应，因为 JPA 映射要求在 `customer` 属性的 `@ManyToOne` 注解上要通过 `optional = false` 标识来映射 Customer。如果这个关系是可选的，那么 DELETE 请求就能正常运行了。

假设最初并不是 Dave 下的订单，而是 Carter。该怎样更新这种分配关系呢？首先，所选的 HTTP 方法应该是 PUT，我们已经知道管理这个资源的 URI。由于我们想要告诉服务器端的是“这个已有的 Customer 才是下订单的人”，因此将实际的数据发送到服务端并没有什么实际意义。因为 Customer 是通过其 URI 来进行识别的，所以将其通过 PUT 请求发送到服务器端，并将 Content-Type 请求头设置为 `text/uri-list`，这样服务器端就能知道发送的是什么，如示例 10-19 所示。

### 示例 10-19 修改下订单的顾客

```
$ curl -v -X PUT -H "Content-Type: text/uri-list" \
-d "http://localhost:8080/customer/2" http://localhost:8080/order/1/customer

* About to connect() to localhost port 8080 (#0)
*   Trying ::1...
* connected
* Connected to localhost (::1) port 8080 (#0)
> PUT /order/1/customer HTTP/1.1
> User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5
> Host: localhost:8080
> Accept: */*
> Content-Type: text/uri-list
> Content-Length: 32
>
* upload completely sent off: 32 out of 32 bytes
< HTTP/1.1 204 No Content
< Content-Length: 0
< Content-Type: application/octet-stream
< Server: Jetty(8.1.5.v20120716)
```

`text/uri-list` (<http://www.rfc-editor.org/rfc/rfc2483.txt>) 是一种标准的媒体类型，用来定义所传输的一个或多个 URI 格式。注意，我们从服务器端得到的是 204 No Content，表明它已经接受了请求并完成了关系的重新分配。

**第五部分**

---

**大数据**



# Spring for Apache Hadoop

Apache Hadoop (<http://hadoop.apache.org/>) 是一个开源项目，该项目起源于雅虎研发的一个新型网络搜索引擎的核心组件。Hadoop 的架构是在谷歌开发的闭源网络搜索引擎架构基础上构建的，可以在 <http://research.google.com/archive/gfs.html> 以及 <http://research.google.com/archive/mapreduce.html> 这两个研究刊物链接中找到相关描述。Hadoop 架构由两个主要的部分组成：一个分布式文件系统以及一个在大型的商业服务器集群上运行的分布式数据处理引擎。Hadoop 分布式文件系统(Hadoop Distributed File System, HDFS) 负责存储和复制数据，以维持跨集群的数据访问的可靠性。Hadoop MapReduce 负责提供编程模型，并且在数据存储位置附近执行计算使得运行时间得到优化。代码和数据托管在同一个物理节点是最大限度缩短处理大（高达 PB 级）数据所需时间的核心技术之一。

虽然 Apache Hadoop 原本的目的是实现网页搜索引擎，但它却是一个通用的平台，可以用于针对海量数据的各种处理任务。由于它结合了开源软件、廉价的商业服务器，以及对大量新的非结构化数据来源（例如，推特、日志文件、遥测）进行分析所带来的切实利益，这三者结合，促使 Hadoop 成为企业寻求实现大数据解决方案的事实标准。

本章从介绍 Hadoop 的“Hello world”应用程序 wordcount 开始。wordcount 应用程序使用 Hadoop MapReduce API 编写。它会读取文本文件作为输入并创建一个输出文件记录所读取的每一个单词出现的次数。首先介绍 Hadoop 应用程序的传统开发方式并使用命令行工具来执行，接着说明如何将这个应用程序开发为标准的 Java 应用程序并使用依赖注入来进行配置。我们使用 Spring for Apache Hadoop 的 HDFS 提供的脚本功能，将输入的文件复制到 HDFS 之中并从 HDFS 中输出结果文件。

### 11.1 Hadoop 开发面临的挑战

在开发 Hadoop 应用程序时会面临几项挑战。首先是 Hadoop 集群的安装。搭建并

管理 Hadoop 集群需要花费大量的时间并且需要具备专业的知识技能，这超出了本书的范围。好在目前许多公司都在这方面积极努力并提供协助，例如亚马逊的 Elastic Map Reduce，不用投入大量的前期成本便可使用 Hadoop。其次，通常开发一个 Hadoop 应用程序并非仅编写一个单独的 MapReduce、Pig 或 Hive job，需要开发一个完整的数据处理管道（pipeline）。这个数据处理管道包括以下几个步骤。

1. 从大量远程计算机或设备采集原始数据。
2. 将数据加载到 HDFS，通常它来自各种资源（例如，应用程序日志）以及事件流的持续处理过程。
3. 在数据经过系统并且被载入 HDFS 时对它执行实时分析。
4. 数据清洗（data cleansing，[http://en.wikipedia.org/wiki/Data\\_cleansing](http://en.wikipedia.org/wiki/Data_cleansing)）并转换原始数据来为分析做好准备。
5. 选择一个框架和编程模型来编写数据分析的 job。
6. 协调多个数据分析 job（例如工作流），每个 job 分别代表一个产生最终分析结果的步骤。
7. 从 HDFS 将最终的分析结果导入到结构化数据存储中，例如关系型数据库或者像 MongoDB 或 Redis 这样的 NoSQL 数据库，用于展示或提供更进一步的分析。

Spring for Apache Hadoop 还有另外两个 Spring 项目，即 Spring Integration 和 Spring Batch，为构建具有一致配置和编程模型的数据处理管道提供了完整的解决方案，这个主题将在第 13 章中介绍。在本章中我们须从基础开始：如何与 HDFS 和 MapReduce 进行交互，这个主题本身就有一些挑战性了。

目前在 Hadoop 文档和培训课程中都建议以命令行工具作为与 HDFS 交互以及执行数据分析 job 的主要方式。在逻辑上，这就相当于使用 SQL\*Plus 来操作 Oracle。使用命令行工具可以使得应用程序变成 bash、Perl、Python 或 Ruby 脚本的一个松散集合。命令行工具也要求为不同的应用环境创建特定参数，以及从一个处理步骤将信息传到另一个步骤。如果使用其他文件系统或者数据访问技术的话，也有简便的方法，通过程序来与 Hadoop 交互。

Spring for Apache Hadoop 旨在简化使用 Java 构建 Hadoop 应用程序的过程。它基于 Spring 框架来提供编写 Hadoop 应用程序的结构。它使用了熟知的基于 Spring 的配置模型，就可以充分利用强大的 Spring 容器配置功能，例如可替换的属性占位符和便利的数据访问异常体系，这样编写的 Hadoop 应用程序和编写其他基于 Spring 的应用程序就能够保持一致的风格。

## 11.2 Hello World

wordcount 示例是介绍 Hadoop MapReduce 编程的经典案例。这个应用程序用来统计文本文件中单词出现的频率。虽然使用 UNIX 命令行，如 sed、awk、或 wc ([http://en.wikipedia.org/wiki/Wc\\_%28Unix%29](http://en.wikipedia.org/wiki/Wc_%28Unix%29)) 看上去也可以轻松完成这个任务，但使用 Hadoop 来实现的原因是可以将它扩大规模以符合 Hadoop 的分布式特性。UNIX 命令行程序可以将数据扩展到 MB 甚至 GB 的级别。然而，它们运行在单一的进程中并且受到单台机器的磁盘传输速率的限制，大约只能达到 100MB/s。读取 1TB 的数据将会花费大约两个半小时的时间。使用 Hadoop，可以将数据分布到 HDFS 集群中，可把数据增长到数百 GB、TB 甚至 PB 级别。将 1TB 的数据集分布到 100 台机器可以将读取时间缩短到两分钟之内。HDFS 会将文件分成多个部分，分别存储在 Hadoop 集群的节点中。执行数据逻辑的 MapReduce 代码会被传送到数据驻留的节点之中，在接近数据的地方执行以增加 I/O 带宽并减少全部 job 的延迟时间，这个阶段就是 MapReduce 的“Map”阶段。为了汇总每一个节点所产生的结果，在集群中有一个节点被用来将各个部分的结果“Reduce”成最终的数据集。在字数统计的示例中，各机器所累加的单词计数会汇总为最终的单词频率清单。

在执行 wordcount 时最有趣的地方是选择要输入的示例文本。Gutenberg 项目提供了从公共领域书籍中下载大量文本的便捷方法，尽管这肯定不是原作者的意图。Gutenberg 项目的目的是将公共领域书籍全面数字化，目前已有超过 39 000 本书籍可供使用。可以浏览这个项目的网站并且使用 wget 下载一些经典文献。在示例 11-1 中，我们在/tmp/gutenberg/download 目录下执行该命令。

### 示例 11-1 使用 wget 下载 wordcount 所需的文本

```
wget -U firefox http://www.gutenberg.org/ebooks/4363.txt.utf-8
```

现在需要使用 HDFS Shell 命令将这些数据放入 HDFS 中。



在运行 Shell 命令前，需要安装 Hadoop ([http://hadoop.apache.org/docs/r1.0.3/file\\_system\\_shell.html](http://hadoop.apache.org/docs/r1.0.3/file_system_shell.html))，有一个很棒的 Michael Noll 在线教程能够引导你在单一机器上设置自己的 Hadoop 集群 (<http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-single-node-cluster>)。

调用位于 Hadoop 发布版 *bin* 目录下的 hadoop 命令即可执行 HDFS Shell 命令。可以使用 Hadoop 命令行参数 dfs 来使用 HDFS，接在它后面的是传统的文件命令和参数，如 cat 或者 chmod。将文件从本地文件系统复制到 HDFS 的命令是 copyFromLocal，如示例 11-2 所示。

## 示例 11-2 复制本地文件到 HDFS

```
hadoop dfs -copyFromLocal /tmp/gutenberg/download /user/gutenberg/input
```

检查文件是否已经存储在 HDFS 里，使用 ls 命令，如示例 11-3 所示。

## 示例 11-3 浏览 HDFS 文件

```
hadoop dfs -ls /user/gutenberg/input
```

我们使用 Hadoop 发布版中的示例 jar 文件来执行 wordcount 应用程序。这个应用程序的参数是要执行的应用程序的名称——在例子中也就是 wordcount——之后加上 HDFS 的输入目录和输出目录，如示例 11-4 所示。

## 示例 11-4 使用 Hadoop 命令行工具运行 wordcount

```
hadoop jar hadoop-examples-1.0.1.jar wordcount /user/gutenberg/input  
/user/gutenberg/output
```

在发送命令后，Hadoop 会执行一段时间，最后的结果放在目录 /user/gutenberg/output 下。可以使用示例 11-5 的命令在 HDFS 中查看输出结果。

## 示例 11-5 在 HDFS 查看 wordcount 输出结果

```
hadoop dfs -cat /user/gutenberg/output/part-r-00000
```

输出文件可能会多于一个，这取决于有多少输入文件。在默认情况下，输出文件名会符合示例 11-5 所示的模式，最后一组递增数字用来表示增加到输出中的每一个文件。若要将结果从 HDFS 复制到本地系统文件中，可使用示例 11-6 的命令。

## 示例 11-6 将结果从 HDFS 复制到本地系统文件中

```
hadoop dfs -getmerge /user/gutenberg/output /tmp/gutenberg/output/wordcount.txt
```

如果在 HDFS 中有多个输出文件，从 HDFS 复制数据到本地文系统文件的时候，可使用 getmerge 选项将它们合并成一个单独的文件。查看该文件的内容，会看到按字母顺序排序的单词和它们在文件中出现的次数。那些看起来有点多余的引号是 MapReduce 程序标记单词的工具。示例 11-7 为 wordcount 应用程序示例的输出。

## 示例 11-7 wordcount 输出文件的部分内容

```
> cat /tmp/gutenberg/output/wordcount.txt  
A 2  
"AWAY 1  
"Ah, 1  
"And 2  
"Another 1  
"  
"By 2  
"Catholicism" 1  
"Cease 1  
"Cheers 1  
"
```

在下一个章节会介绍 Hadoop 发布版中的示例应用程序是如何提交 job 给 Hadoop 的。这有助于了解开发和运行自己的应用程序时需要关注的内容。

### 11.3 揭秘 Hello World

如果需要自行开发和运行 MapReduce 应用程序，而不是只使用“开箱即用”的程序，那么必须先了解幕后所发生的一些重要的事情。为了了解示例应用程序是如何工作的，我们应该首先查看 `hadoop-examples.jar` 的 `META-INF/manifest.mf` 文件。这个清单文件列出 Java 运行的主体类 `org.apache.hadoop.examples.ExampleDriver`。`ExampleDriver` 负责将命令行的第一个参数 `wordcount` 与 Java 类 `org.apache.hadoop.examples.Wordcount` 关联，并且使用 `ProgramDriver` 辅助类执行 `Wordcount` 的 `main` 方法。示例 11-8 为 `ExampleDriver` 的简化版本。

#### 示例 11-8 wordcount 应用程序的主程序

```
public class ExampleDriver {  
  
    public static void main(String... args){  
  
        int exitCode = -1;  
        ProgramDriver pgd = new ProgramDriver();  
  
        try {  
            pgd.addClass("wordcount", WordCount.class,  
                         "A map/reduce program that counts the words in the input files.");  
            pgd.addClass("randomwriter", RandomWriter.class,  
                         "A map/reduce program that writes 10GB of random data per node.");  
  
            // additional invocations of addClass excluded that associate keywords  
            // with other classes  
  
            exitCode = pgd.driver(args);  
        } catch(Throwable e) {  
            e.printStackTrace();  
        }  
  
        System.exit(exitCode);  
    }  
}
```

`WordCount` 类也有 `main` 方法，在启动的时候它不会直接被 JVM 调用，而是在 `ProgramDriver` 的 `driver` 方法被调用的时候执行。`WordCount` 类如示例 11-9 所示。

#### 示例 11-9 ProgramDriver 所调用的 wordcount main 方法

```
public class WordCount {  
  
    public static void main(String... args) throws Exception {  
  
        Configuration conf = new Configuration();  
        String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();  
  
        if (otherArgs.length != 2) {  
            System.err.println("Usage: wordcount <in> <out>");  
            System.exit(2);  
        }  
    }  
}
```

```

    }

    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}

```

现在要介绍的是配置和执行 MapReduce 应用程序的核心代码。必要的步骤为：创建一个新的 Hadoop Configuration 对象、创建一个 Job 对象、设置 job 的一些属性，然后使用方法 `waitForCompletion(...)` 运行 job。当创建自己的应用程序时，Mapper 和 Reducer 类组成了你要编写的核心代码逻辑。

虽然名字显得非常通用，但 TokenizerMapper 和 IntSumReducer 是 WordCount 类的静态内部类，它们将负责计算单词数并汇总最终结果，如示例 11-10 所示。

#### 示例 11-10 内置的 wordcount 应用程序所对应的 Mapper 与 Reducer

```

public class WordCount {

    public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(Object key, Text value, Context context)
            throws IOException, InterruptedException {
            StringTokenizer itr = new StringTokenizer(value.toString());
            while (itr.hasMoreTokens()) {
                word.set(itr.nextToken());
                context.write(word, one);
            }
        }
    }

    public static class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
        private IntWritable result = new IntWritable();

        public void reduce(Text key, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
        }
    }
}

```

```
    ... // ...main method as shown before  
}
```

在 Hadoop 发布版中有许多内置的示例，ProgramDriver 工具类会根据第一个命令行参数来指定要运行的 Hadoop Job。也可以不使用 ProgramDriver 工具类，而使用标准的 Java 主应用程序来运行 WordCount，只需要对处理命令行参数的部分稍微做一些修改即可。修改后的 WordCount 如示例 11-1 所示。

### 示例 11-11 独立的 wordCount 主应用程序

```
public class WordCount {  
  
    // ... TokenizerMapper shown before  
    // ... IntSumReducer shown before  
  
    public static void main(String[] args) throws Exception {  
  
        Configuration conf = new Configuration();  
        if (args.length != 2) {  
            System.err.println("Usage: <in> <out>");  
            System.exit(2);  
        }  
  
        Job job = new Job(conf, "word count");  
        job.setJarByClass(WordCount.class);  
        job.setMapperClass(TokenizerMapper.class);  
        job.setCombinerClass(IntSumReducer.class);  
        job.setReducerClass(IntSumReducer.class);  
        job.setOutputKeyClass(Text.class);  
        job.setOutputValueClass(IntWritable.class);  
  
        FileInputFormat.addInputPath(job, new Path(args[0]));  
        FileOutputFormat.setOutputPath(job, new Path(args[1]));  
        System.exit(job.waitForCompletion(true) ? 0 : 1);  
    }  
}
```

本章的示例应用程序位于目录./hadoop/wordcount，同时还有一个 WordCount 应用程序的 Maven 构建文件。这可以让你将 WordCount 当成常规的 Java 应用程序来执行而不必使用 Hadoop 的命令行工具。使用 Maven 来构建应用程序并执行标准的 Java 主程序是按照常规 Java 应用方式开发和部署 Hadoop 的第一步，另外，还需要一个不依赖于 Hadoop 命令行的工具。Maven 构建能够使用 Appassembler(<http://mojo.codehaus.org/appassembler/appassembler-maven-plugin/>) 插件来产生 UNIX 和 Windows 的启动脚本，并且将所需要的依赖关系收集到本地的 lib 目录下，这个目录会作为生成脚本的类路径。

如果要使用相同的输出路径重新执行之前的 WordCount 示例，则必须首先删除已经存在的文件和目录，因为 Hadoop 不允许在已经存在的目录执行写操作。HDFS Shell 的 rmr 命令可以实现这一目标，如示例 11-12 所示。

### 示例 11-12 删除 HDFS 中的目录和内容

```
hadoop dfs -rmr /user/gutenberg/output
```

为了构建这个应用程序，需要执行 Appassembler 的 assemble target 并运行生成 wordcount shell 脚本，如示例 11-13 所示。

### 示例 11-13 构建、运行和查看独立的 wordcount 应用程序的输出

```
$ cd hadoop/wordcount  
$ mvn clean package appassembler:assemble  
$ sh ./target/appassembler/bin/wordcount hdfs://localhost:9000/user/gutenberg/inpu  
hdfs://localhost:9000/user/gutenberg/output  
  
INFO: Total input paths to process : 1  
INFO: Running job: job_local_0001  
...  
INFO:      Map output records=65692  
  
$ hadoop dfs -cat /user/gutenberg/output/part-r-00000  
"A 2  
"AWAY 1  
"Ah, 1  
"And 2  
"Another 1  
"Are 2  
"BIG 1  
..."
```

与使用 Hadoop 命令行不同的地方是需要在 HDFS 目录前加上 URL 模式的 `dfs://` 前缀以及主机名称和 namenode 的端口。需要这么做的原因是，在使用 Hadoop 命令行时，会设置 Hadoop Configuration 类可识别的环境变量，并且它会在命令行参数传入的路径前面加上相关信息。请注意还有其他可用于 Hadoop 的 URL 模式。webhdfs 模式非常实用，因为它提供了一个基于 HTTP 的通讯协议来与 HDFS 进行通信，而不要求客户端（也就是我们的应用程序）以及 HDFS 服务器使用版本完全相同（要精确到发行版本的小版本号）的 HDFS 类库。

## 11.4 使用 Spring for Apache Hadoop 的 Hello World

如果一直阅读本章到现在，你或许会感到疑惑，Spring 可以对这些应用程序做些什么呢？在本节将开始介绍 Spring for Apache Hadoop 所提供的功能来帮助你构造、配置以及运行 Hadoop 应用程序。首先要介绍的功能是使用 Spring 来配置并运行 Hadoop Job，可以将应用程序参数外部化（externalize）到不同的配置文件中。这能够让应用程序很容易地适应不同的运行环境——如开发、QA 以及生产环境——而不需要改变任何程序代码。

使用 Spring 来配置与运行 Hadoop job 时，可以借助 Spring 丰富的配置功能，就和使用 Spring 配置和运行的其他应用程序一样，例如属性占位符。对于像 wordcount 这样简单的应用程序，或许不必花功夫设置 Spring 应用程序上下文，但实际上构建如此简单的应用程序并不常见。应用程序通常会有几个 HDFS 操作和 MapReduce

job（或等效的 Pig 和 Hive 脚本）连接在一起。同样正如 11.1 小节中“Hadoop 开发面临的挑战”所提到的，在构建完整的数据管道解决方案时，必须考虑许多其他的开发活动。以 Spring for Apache Hadoop 为基础开发的 Hadoop 应用程序为我们奠定了一个基础，可通过重用组件来应对越来越复杂的应用程序。

下面将上一节所开发的 WordCount 版本移到 Spring 容器内运行。使用 Hadoop XML 命名空间来定义 namenode 的位置并使用最少的信息来定义 org.apache.hadoop.mapreduce.Job 实例，如示例 11-14 所示。

#### 示例 11-14 使用 Spring 的 Hadoop 命名空间定义 Hadoop job

```
<configuration>
    fs.default.name=hdfs://localhost:9000
</configuration>

<job id="wordcountJob"
    input-path="/user/gutenberg/input"
    output-path="/user/gutenberg/output"
    mapper="org.apache.hadoop.examples.WordCount.TokenizerMapper"
    reducer="org.apache.hadoop.examples.WordCount.IntSumReducer"/>

<job-runner id="runner" job="wordcountJob" run-at-startup="true"/>
```

这个配置将创建一个由 Spring 容器管理的单例 org.apache.hadoop.mapreduce.Job。在之前的一些示例中以编程的方法在 job 类中设置的属性可以从 Mapper 和 Reducer 类签名中推导出来。Spring 可以判断出 outputKeyClass 的类型是 org.apache.hadoop.io.Text 并且 outputValueClass 的类型是 org.apache.hadoop.io.IntWritable，因此不需要显式地设置这两个属性。还有许多其他类似于 Hadoop 命令行可选项的 job 属性可以进行设置（例如，Combiner、输入格式、输出格式以及通用的 job 键/值属性）。可以在 Eclipse 或者其他编辑器中使用 XML 模式的自动完成功能来查看各种可用的选项，也可以参阅 *Spring for Apache Hadoop* 参考文档来获取更多信息。目前 namenode 的位置、输入和输入路径都是硬编码的方式确定的，稍后会将它们提取到外部的属性文件中。

如同使用 Hadoop 命令行来运行 job，不需要在指定输入、输出路径时指明 URL 模式以及 namenode 的主机和端口。因为<configuration/>元素定义了的 URL 模式和 namenode 信息。如果想要使用 webhdfs 协议的话，可将 fs.default.name 设置成 webhdfs://localhost。也可以指定其他 Hadoop 配置项的值，如 dfs.permissions、hadoop.job.ugi,mapred.job.tracker 与 dfs.datanode.address。

要在创建 Spring 应用上下文时运行 MapReduce job，需使用工具类 JobRunner 来引用一个或者多个被 Spring 管理的 Job 对象并且将 run-atstartup 属性设置为 true。示例 11-15 是取代 org.apache.hadoop.examples.ExampleDriver 的主应用程序类。在默认情况下，这个应用程序会根据已知的目录来查找 XML 配置文件，当然，也可以提供一个引用其他配置文件位置的命令行参数来重写它。

### 示例 11-15 Spring 所管理的 wordcount 应用程序的主类

```
public class Main {  
  
    private static final String[] CONFIGS = new String[] {  
        "META-INF/spring/hadoop-context.xml" };  
  
    public static void main(String[] args) {  
        String[] res = (args != null && args.length > 0 ? args : CONFIGS);  
        AbstractApplicationContext ctx = new ClassPathXmlApplicationContext(res);  
        // shut down the context cleanly along with the VM  
        ctx.registerShutdownHook();  
    }  
}
```

应用程序的示例代码位于 `./hadoop/wordcount-spring-basic`。可以像前面章节那样构建并运行这个应用程序，如示例 11-16 所示。请确保在 HDFS 中已经删除了输出文件，因为运行上一小节的 wordcount 时已经建立了输出文件。

### 示例 11-16 构建并运行基于 Spring 的 wordcount 应用程序

```
$ hadoop dfs -rmr /user/gutenberg/output  
$ cd hadoop/wordcount-spring-basic  
$ mvn clean package appassembler:assemble  
$ sh ./target/appassembler/bin/wordcount
```

现在 Hadoop job 是由 Spring 管理的对象，它可以被注入到任意 Spring 管理的其他对象中。例如，如果要让 wordcount job 运行在 Web 应用程序中，那么可以将它注入到 Spring MVC 控制器中，如示例 11-17 所示。

### 示例 11-17 将 Hadoop job 注入到 WebMVC 控制器中

```
@Controller  
public class WordController {  
    private final Job mapReduceJob;  
  
    @Autowired  
    public WordService(Job mapReduceJob) {  
        Assert.notNull(mapReduceJob);  
        this.mapReduceJob = mapReduceJob;  
    }  
  
    @RequestMapping(value = "/runjob", method = RequestMethod.POST)  
    public void runJob() {  
        mapReduceJob.waitForCompletion(false);  
    }  
}
```

为了启动应用并将程序的配置参数外部化，可以使用 Spring 的属性占位符功能并且将键参数移至配置文件中，如示例 11-18 所示。

### 示例 11-18 使用 Spring 的 Hadoop 命名空间声明参数化的 Hadoop job

```
<context:property-placeholder location="hadoop-default.properties"/>  
  
<configuration>  
    fs.default.name=${hd.fs}
```

```
</configuration>

<job id="wordcountJob"
    input-path="${wordcount.input.path}"
    output-path="${wordcount.output.path}"
    mapper="org.apache.hadoop.examples.WordCount.TokenizerMapper"
    reducer="org.apache.hadoop.examples.WordCount.IntSumReducer"/>

<job-runner id="runner" job="wordcountJob" run-at-startup="true"/>
```

变量名称 `hd.fs`、`wordcount.input.path` 以及 `wordcount.output.path` 都在 `hadoop-default.properties` 配置文件中指定，如示例 11-19 所示。

### 示例 11-19 `hadoop-default.properties` 属性文件，为默认的开发环境设置 Hadoop 应用程序参数

```
hd.fs=hdfs://localhost:9000
wordcount.input.path=/user/gutenberg/input/
wordcount.output.path=/user/gutenberg/output/
```

这个文件位于 `src/main/resources` 目录下，因此构建脚本时可以在类路径中找到它。我们也创建了另外一个名为 `hadoop-qa.properties` 的配置文件，它定义了 QA 环境下 `namenode` 的位置。为了在同一台机器上运行样例，只改变输出路径的名称即可。在真实的 QA 环境中，HDFS 集群的位置以及 HDFS 输入输出路径可能会有所不同。

### 示例 11-20 `hadoop-qa.properties` 属性文件，为 QA 环境设置 Hadoop 应用程序参数

```
hd.fs=hdfs://localhost:9000
wordcount.input.path=/data/words/input
wordcount.output.path=/data/words/qa/output
```

为了充分利用 Spring 对环境支持的功能，能够在不同的配置文件中实现轻松地切换，我们更改属性占位符的定义，将变量  `${ENV}`  设置为要加载的属性文件名称。在默认情况下，Spring 首先会搜索 JVM 系统属性，之后再搜索环境变量来解析变量名称。使用语法  `${ENV:<Default Value>}`  对变量指定默认值。在示例 11-21 中，如果没有设置 shell 的环境变量 ENV，那么  `${ENV}`  将使用默认值并且加载 `hadoop-default.properties` 属性文件。

### 示例 11-21 根据不同的运行环境来引用不同的配置文件

```
<context:property-placeholder location="hadoop-${ENV:default}.properties"/>
```

要在 QA 环境中通过命令行运行应用程序，可以执行如示例 11-22 所示的命令。注意 shell 的环境变量（ENV）是如何被设置为 QA 的。

### 示例 11-22 在 QA 环境中构建并运行基于 Spring 的 wordcount 应用程序

```
$ hadoop dfs -copyFromLocal /tmp/gutenberg/download /user/gutenberg/input
$ hadoop dfs -rmr /user/gutenberg/qa/output
$ cd hadoop/wordcount-spring-intermediate
$ mvn clean package appassembler:assemble
```

```
$ export ENV=qa  
$ sh ./target/appassembler/bin/wordcount
```

如之前的示例一样，要重新执行 Hadoop 应用程序时，一定要将结果写到一个空目录中，否则 Hadoop job 将会失败。在开发期，使用 IDE 时，如果每次启动应用程序时都要记得做这件事会让人觉得非常乏味。有一个方法可以将输出引导到新的目录，这个目录的名称以时间戳（例如`/user/gutenberg/output/2012/6/30/14/30`）来取代静态的目录名称。现在来看一下如何使用 Spring 的 HDFS 脚本特性协助我们完成这个通用的任务。

## 11.5 在 JVM 中编写 HDFS 脚本

在开发 Hadoop 应用程序时，需要经常与 HDFS 交互。作为开始，最常见的方法是使用 HDFS 命令行 Shell。例如下面的命令可获取目录内的文件清单：

```
hadoop dfs -ls /user/gutenberg/input
```

这在初始阶段足以满足需求，但是在编写 Hadoop 应用程序时，通常需要执行一连串更复杂的文件系统操作命令。例如，测试目录是否存在，如果存在则将它删除并将一些新的文件复制进去。作为一个 Java 开发者，你可能会觉得将这个功能加到 bash 脚本中去是一种落后的办法。或许可以使用程序来做这件事，是吗？好消息是：是的，可使用 HDFS 文件系统 API 来完成，坏消息是 Hadoop 文件系统 API 并不是很易用，它会抛出已检查异常并要求我们为它的诸多方法构建 Path 实例作为参数，这会让调用结构变得冗长和笨拙。此外，Hadoop 文件系统 API 并没有提供太多在命令行中可以使用的高级方法，如 test 与 chmod。

Spring for Apache Hadoop 在这中间提供了辅助功能，它对 Hadoop 的 FileSystem 类进行了封装，从而可以接受 String 类型的参数，而不是 Path 类型的参数。更重要的是，它提供了一个 FsShell 类用以模仿命令行 Shell 的功能，这就意味着可用编程的方式来使用它。FsShell 方法所返回的对象或集合可以进行查看也能够以编码的方式来使用，而不是直接将信息输出到控制台上。FsShell 类也可以跟 JVM 脚本语言结合，这样就可以继续使用脚本风格的交互模型，但增加了 JRuby、Jython 或 Groovy 的功能以取代 bash。示例 11-23 在 Groovy 脚本中使用 FsShell。

### 示例 11-23 定义一个可执行的 Groovy 脚本

```
<configuration>  
  fs.default.name=hdfs://localhost:9000  
</configuration>  
  
<script location="org/company/basic-script.groovy"/>
```

`<script/>`元素用于创建一个 FsShell 实例，它会通过变量名 fsh 隐式地传入 Groovy 脚本之中。Groovy 脚本如示例 11-24 所示。

### 示例 11-24 在 HDFS 中使用的 Groovy 脚本

```
srcDir = "/tmp/gutenberg/download/"

// use the shell (made available under variable fsh)
dir = "/user/gutenberg/input"
if (!fsh.test(dir)) {
    fsh.mkdir(dir)
    fsh.copyFromLocal(srcDir, dir)
    fsh.chmod(700, dir)
}
```

还有其他的选项可以用来控制脚本何时执行以及求值的方式。如果不想在启动时执行脚本，需在 `script` 标签的元素中设置 `run-at-startup = "false"`。如果想要脚本在文件系统中被更改时重新执行，需在 `script` 标签的元素中设置 `evaluate = "IF_MODIFIED"`。也可以将要执行脚本参数化，传入由 Spring 的属性占位符功能所定义的参数，如示例 11-25 所示。

### 示例 11-25 配置参数化的 Groovy 脚本来使用 HDFS

```
<context:property-placeholder location="hadoop.properties"/>

<configuration>
    fs.default.name=${hd.fs}
</configuration>

<script id="setupScript" location="copy-files.groovy">
    <property name="localSourceFile" value="${localSourceFile}" />
    <property name="inputDir" value="${inputDir}" />
    <property name="outputDir" value="${outputDir}" />
</script>
```

属性文件 `hadoop.properties` 以及 `copy-files.groovy` 分别如示例 11-26 与示例 11-27 所示。

### 示例 11-26 包含 HDFS 脚本变量的属性文件

```
hd.fs=hdfs://localhost:9000
localSourceFile=data/apache.log
inputDir=/user/gutenberg/input/word/
outputDir=
    #{T(org.springframework.data.hadoop.util.PathUtils).format('/output/%1$tY/%1$tm/%1$td')}
```

Spring for Apache Hadoop 还提供了一个 `PathUtils` 类，在创建基于时间命名的目录路径时它非常有用。调用静态的 `format` 方法之后，会生成一个以时间命名的路径，它会基于目前的日期，并使用 `java.util.Formatter` 的约定来对时间进行格式化。可以在 Java 中使用这个类，同时也可以在配置属性中通过 Spring 的表达式语言即 SpEL (<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/expressions.html>) 来引用它。SpEL 的语法类似 Java，它的表达式通常只是一行要求值的代码。在这里它不是使用语法 `${...}` 来引用变量，而是以语法`# {...}` 的形式来执行表达式。在 SpEL 中，特殊的“T”运算符用来指定一个 `java.lang.Class` 实例，我们可以用“T”运算符调用静态方法。

### 示例 11-27 以参数化的 Groovy 脚本使用 HDFS

```
if (!fsh.test(hdfsInputDir)) {  
    fsh.mkdir(hdfsInputDir);  
    fsh.copyFromLocal(localSourceFile, hdfsInputDir);  
    fsh.chmod(700, hdfsInputDir)  
}  
if (fsh.test(hdfsOutputDir)) {  
    fsh.rmtree(hdfsOutputDir)  
}
```

FsShell 可以很容易地在 Java 或 JVM 脚本语言中使用 HDFS Shell 操作，与之类似，org.springframework.data.hadoop.fs.DistCp 也可以让我们很容易地使用 Hadoop 命令行的 distcp 操作。distcp 工具类的用途是在单一的 Hadoop 集群或集群之间复制大量的文件，并且利用 Hadoop 本身以分布式的方式执行复制。distcp 变量会隐式地暴露到脚本之中，如示例 11-28 所示，在这个示例中，脚本内嵌于<hdp:script/>元素之中，而不是位于独立的文件中。

### 示例 11-28 在 Groovy 脚本内使用 distcp

```
<configuration>  
    fs.default.name=hdfs://localhost:9000  
</configuration>  
  
<script language="groovy">  
    distcp.copy("${src}", "${dest}")  
</script>
```

在这个例子中，Groovy 脚本内嵌于 XML 之中，而不是引用一个外部的文件。值得注意的是，可以使用 Spring 的属性占位符功能在脚本中引用如\${src}和\${dst}这样的变量，从而实现脚本参数化。

## 11.6 结合 HDFS 脚本与 Job 提交

基本的 Hadoop 应用程序都会包含一些 HDFS 操作与一些 job 提交的任务。可以使用 JobRunner 的 pre-action 和 post-action 属性将这些任务排序，让 HDFS 脚本操作在 job 提交之前或者之后执行，示例 11-29 说明了这一点。

### 示例 11-29 在 Bean 之间使用依赖关系来控制执行顺序

```
<context:property-placeholder location="hadoop.properties"/>  
  
<configuration>  
    fs.default.name=${hd.fs}  
</configuration>  
  
<job id="wordcountJob"  
      input-path="${wordcount.input.path}"  
      output-path="${wordcount.output.path}"  
      mapper="org.apache.hadoop.examples.WordCount.TokenizerMapper"  
      reducer="org.apache.hadoop.examples.WordCount.IntSumReducer"/>
```

```

<script id="setupScript" location="copy-files.groovy">
    <property name="localSourceFile" value="${localSourceFile}" />
    <property name="inputDir" value="${wordcount.input.path}" />
    <property name="outputDir" value="${wordcount.output.path}" />
</script>

<job-runner id="runner" run-at-startup="true"
            pre-action="setupScript"
            job="wordcountJob"/>

```

`pre-action` 属性引用 `setupScript` Bean，这个 Bean 又引用 `copy-files.groovy` 脚本，该脚本会重置系统的状态，这样不需要任何的命令行交互即可运行或重新运行这个程序。示例 11-30 说明了使用命令来构建和运行应用程序的方式。

### 示例 11-30 构建并运行基于 Spring 的 wordcount 应用程序

```

$ hadoop dfs -rmr /user/gutenberg/output
$ cd hadoop/wordcount-hdfs-copy
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/wordcount

```

示例 11-31 的配置让 `JobRunner` 在执行多个 Hadoop job 之前与之后执行一些 HDFS 脚本。`JobRunner` 也实现了 Java 的 `Callable` 接口，它使用了 Java 的 `Executor` 框架让 `JobRunner` 的执行变得更为简单。这个方法在执行简单的工作流程时很方便，但是随着应用程序变得更为复杂，它并不能提供更多的帮助。针对 Hadoop 的扩展 Spring Batch 将处理 HDFS 与 job 操作链作为首要任务，这个时候就可以派上用场。我们将在 13.2 小节 “Hadoop 工作流” 中探讨这些扩展。

### 示例 11-31 配置 `JobRunner` 来执行多个 HDFS 脚本和 job

```

<job-runner id="runner"
            pre-action="setupScript1,setupScript"
            job="wordcountJob1,wordcountJob2"
            post-action="cleanupScript1,cleanupScript2"/>

```

## 11.7 Job 调度

应用程序通常需要调度和执行任务，这些任务可能是发送一封 Email 或运行一个耗时的批量处理程序。*Job Schedulers* 是一类提供这种功能的软件。你可能已经知道，这类产品有很多，包括通用的 UNIX `cron` 工具甚至更复杂的开源和商业产品，如 Quartz、Control-M 与 Autosys。Spring 提供多种 Job 调度方式，包括 `JDKTimer`、集成 Quartz 以及 Spring 自带的 `TaskScheduler`。本节介绍如何使用 Quartz 与 Spring 的 `TaskScheduler` 来调度和执行 Hadoop 应用程序。

### 11.7.1 使用 `TaskScheduler` 调度 MapReduce Job

在这个示例中，我们会在之前开发的应用程序中加上任务调度的功能，用来执行 HDFS 脚本以及 wordcount 的 MapReduce Job。Spring 的 `TaskScheduler` 和 `Trigger`

接口可对将要运行的任务进行调度。Spring 提供了一些实现，常见的选择为 ThreadPoolTaskScheduler 和 CronTrigger。我们可以使用 XML 命名空间加上一个基于注解的编程模型将任务设置为由触发器进行调度。

在示例 11-32 的配置中，我们使用 XML 命名空间，它会调用任意由 Spring 管理的对象的方法，在这个案例中也就是 JobRunner 实例的 call 方法。触发器是一个 cron 表达式，它会从第 3 秒开始，每隔 30 秒触发一次。

### 示例 11-32 定义 TaskScheduler 来执行 HDFS 脚本以及 MapReduce Job

```
<!-- job definition as before -->
<job id="wordcountJob" ... />

<!-- script definition as before -->
<script id="setupScript" ... />
<job-runner id="runner" pre-action="setupScript" job="wordcountJob"/>

<task:scheduled-tasks>
  <task:scheduled ref="runner" method="call" cron="3/30 * * * * ?"/>
</task:scheduled-tasks>
```

JobRunner 的 run-at-startup 元素的默认值是 false，因此在这个配置中，不会在应用程序启动的时候执行 HDFS 脚本与 Hadoop Job。使用这个配置能够让调度器成为系统中唯一负责执行脚本和 Job 的组件。

这个示例应用程序位于目录 hadoop/scheduling。在执行这个应用程序时，可以看到如示例 11-33 所示的（被截断的一部分）输出，其中时间戳符合 cron 表达式的定义。

### 示例 11-33 调度 wordcount 应用程序的输出

```
removing existing input and output directories in HDFS...
copying files to HDFS...
23:20:33.664 [pool-2-thread-1] WARN o.a.hadoop.util.NativeCodeLoader
- Unable to load native-hadoop library for your platform...
23:20:33.689 [pool-2-thread-1] WARN org.apache.hadoop.mapred.JobClient
- No job jar file set. User classes may not be found.
23:20:33.711 [pool-2-thread-1] INFO o.a.h.m.lib.input.FileInputFormat
- Total input paths to process : 1
23:20:34.258 [pool-2-thread-1] INFO org.apache.hadoop.mapred.JobClient
- Running job: job_local_0001 ...
23:20:43.978 [pool-2-thread-1] INFO org.apache.hadoop.mapred.JobClient
- map 100% reduce 100%
23:20:44.979 [pool-2-thread-1] INFO org.apache.hadoop.mapred.JobClient
- Job complete: job_local_0001
23:20:44.982 [pool-2-thread-1] INFO org.apache.hadoop.mapred.JobClient
- Counters: 22
removing existing input and output directories in HDFS...
copying files to HDFS...
23:21:03.396 [pool-2-thread-1] INFO org.apache.hadoop.mapred.JobClient
- Running job: job_local_0001
23:21:03.397 [pool-2-thread-1] INFO org.apache.hadoop.mapred.JobClient
- Job complete: job_local_0001
```

正如你所见，task 的命名空间非常易于使用，但是它也有许多功能（在这里我们将

不涉及这些功能) 与线程池策略有关或者委托给 CommonJ WorkManager 代理。Spring 框架的参考文档 (<http://docs.spring.io/spring/docs/3.0.x/spring-framework-reference/html/scheduling.html>) 详细说明了这些特性。

### 11.7.2 使用 Quartz 调度 MapReduce Job

Quartz (<http://quartz-scheduler.org/>) 是一个广受欢迎的开源任务调度框架，具有许多高级功能，如集群。在这个示例中，我们将之前示例所使用的 Spring TaskScheduler 替换成 Quartz。Quartz 调度要求定义 Job (即 JobDetail)、Trigger 以及 Scheduler，如示例 11-34 所示。

#### 示例 11-34 定义 Quartz 调度来执行 HDFS 脚本和 MapReduce Job

```
<!-- job definition as before -->
<hdःjob id="wordcountJob" ... />

<!-- script definition as before -->
<hdःscript id="setupScript" ... />

<!-- simple job runner as before -->
<hdःjob-runner

    <bean id="jobDetail"
        class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
        <property name="targetObject" ref="runner"/>
        <property name="targetMethod" value="run"/>
    </bean>

    <bean id="cronTrigger" class="org.springframework.scheduling.quartz.CronTriggerBean">
        <property name="jobDetail" ref="jobDetail"/>
        <property name="cronExpression" value="3/30 * * * * ?"/>
    </bean>

    <bean class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
        <property name="triggers" ref="cronTrigger"/>
    </bean>
```

Quartz 的 JobDetail 类封装了满足触发条件时要执行的程序。Spring 的辅助类 MethodInvokingJobDetailFactoryBean 会建立一个 JobDetail 对象，它的功能是调用 Spring 管理对象中的指定方法。这个应用程序位于目录 hadoop/scheduling-quartz。程序的执行结果与基于 Spring TaskScheduler 示例所得到的结果类似。

# 使用 Hadoop 分析数据

虽然 MapReduce 编程模型是 Hadoop 的核心，但是它的实现较为底层，因此开发人员无法以高效的方式编写复杂分析的 Job。为了提升开发效率，一些高级语言与 API 提供了对 MapReduce 编程模型底层的抽象。目前有许多编写数据分析 Job 的方案供选择，其中 Hive 和 Pig 项目是最流行的，它们分别提供了类 SQL 的查询语言以及面向数据流的编程语言。HBase 也是往 HDFS 中存储数据和对其进行分析的常选工具，它是一个面向列的分布式数据库。HBase 与 MapReduce 的不同之处是它提供了低延时的数据随机读写功能。MapReduce Job 可以读写 HBase 表结构的数据，但通常我们会通过 HBase 客户端 API 来完成数据处理。本章将介绍如何使用 Spring for Apache Hadoop 来编写使用这些 Hadoop 技术的 Java 应用程序。

## 12.1 使用 Hive

上一章使用了 MapReduce API 来分析存储在 HDFS 中的数据。虽然用 MapReduce API 来统计单词出现频率是个比较简单的任务，但是复杂的分析任务就不适合使用 MapReduce 模型了，这会降低开发效率。为了解决这个难题，Facebook 开发了 Hive 来与 Hadoop 以更具声明式、类 SQL 的方式交互。Hive 提供了一个名为 HiveQL 的语言用以分析存储在 HDFS 中的数据，因为它类似 SQL，所以非常容易学习。在底层，HiveQL 查询会被转换成基于 MapReduce API 的多个 Job。目前 Hive 是 Apache 上的顶级项目，但 Facebook 依然还在全力投入开发。

关于 Hive 更深入的介绍超出了本书的范围。它的基本编程模型是创建一个 Hive 表模式用来对 HDFS 中存储的数据提供结构。接着 HiveQL 查询会在 Hive 引擎中解析，并把它们转换成 MapReduce Job 以执行查询。HiveQL 语句可以通过命令行提交给 Hive 引擎，也可以通过名为 Hive 服务器的组件提交给引擎，这个组件提供了通过 JDBC、ODBC 或 Thrift 来进行访问功能。关于安装、运行以及使用 Hive 与 HiveQL

的更多详细介绍，可参考项目网站 (<http://hive.apache.org/>) 以及《*Programming Hive*》(O'Reilly 出版) 一书。

如同 MapReduce Job 一样，Spring for Apache Hadoop 的目标是简化 Hive 编程，不依赖于命令行工具来开发和运行 Hive 应用程序。取而代之，Spring for Apache Hadoop 连接 Hive 服务器（可选择内嵌方式）、创建 Hive Thrift 客户端，并通过 Hive JDBC 驱动来使用 Spring 丰富的 JDBC 支持（JdbcTemplate），进而简化 Java 应用程序的编写过程。

### 12.1.1 Hello World

为了介绍 Hive 的使用方式，这个小节将使用 Hive 命令行来对 UNIX passwd 文件执行简单的分析，目标是建立一个特定 Shell（例如 bash 或 sh）使用者数量的报表。可从 Hive (<http://hive.apache.org/>) 网站下载并安装 Hive，安装完 Hive 的发布版之后，需将它的 bin 目录加入到你的路径中。现在如示例 12-1 所示，可以启动 Hive 命令行控制台并执行一些 HiveQL 命令。

#### 示例 12-1 在 Hive 命令行界面中分析 passwd 文件

```
$ hive
hive> drop table passwords;
hive> create table passwords (user string, passwd string, uid int, gid int,
    userinfo string, home string, shell string)
    > ROW FORMAT DELIMITED FIELDS TERMINATED BY ':' LINES TERMINATED BY '\n';
hive> load data local inpath '/etc/passwd' into table passwords;
Copying data from file:/etc/passwd
Copying file: file:/etc/passwd
Loading data to table default.passwords
OK
hive> drop table grpshell;
hive> create table grpshell (shell string, count int);
hive> INSERT OVERWRITE TABLE grpshell SELECT p.shell, count(*)
    FROM passwords p GROUP BY p.shell;
Total MapReduce jobs = 1
Launching Job 1 out of 1
...
Total MapReduce CPU Time Spent: 1 seconds 980 msec
hive> select * from grpshell;
OK
/bin/bash 5
/bin/false 16
/bin/sh 18
/bin/sync 1
/usr/sbin/nologin 1
Time taken: 0.122 seconds
```

也可以将 HiveQL 命令放到文件中并在命令行执行它，如示例 12-2 所示。

#### 示例 12-2 在命令行执行 Hive

```
$ hive -f password-analysis.hql
$ hadoop dfs -cat /user/hive/warehouse/grpshell/000000_0
/bin/bash 5
```

```
/bin/false 16  
/bin/sh 18  
/bin/sync 1  
/usr/sbin/nologin 1
```

Hive 命令行会将命令直接传给 Hive 引擎。Hive 也支持变量替换，脚本中可使用符号 \${hiveconf:varName} 来替换在命令行中通过 -hiveconf varName = varValue 设置的变量。如果不使用命令行与 Hive 交互，就需要通过 Thrift 客户端或者 JDBC 连接到 Hive 服务器。下一个节将介绍如何在命令行中启动 Hive 服务器，以及如何在 Java 应用程序中启动内嵌的服务器。

## 12.1.2 运行 Hive 服务器

在生产环境中，最常见的场景是将 Hive 服务器作为独立的服务器进程来运行——可能在 HAProxy 后面有多个 Hive 服务器——从而避免处理大量并发客户端连接所带来的问题<sup>1</sup>。

如果想把示例应用程序运行在独立的服务器中，可使用下面的命令来启动 Hive：

```
hive --service hiveserver -hiveconf fs.default.name=hdfs://localhost:9000 \  
-hiveconf mapred.job.tracker=localhost:9001
```

另一个对开发期很实用而且可免于运行其他服务器的方法是在同一个运行 Hive 客户端应用程序的 Java 进程中启动 Hive 服务。通过 Hadoop 命名空间只需一行配置即可嵌入 Hive 服务器，如示例 12-3 所示。

### 示例 12-3 使用默认配置项创建 Hive 服务器

```
<hive-server/>
```

在默认配置下，主机名称为 localhost，端口为 10000，可以使用 host 与 port 属性来更改这些值。也可以通过 properties-location 引用属性文件或在<hive-server/> XML 元素中内联属性来对 Hive 服务器进行其他的配置。当应用上下文创建时，Hive 服务器会自动启动。如果不想 Hive 服务器自动启动的话，可以设置 auto-startup 元素为 false。最后，可以引用特定的 Hadoop 配置对象，创建多个连接不同 Hadoop 集群的 Hive 服务器。这些配置方式如示例 12-4 所示。

### 示例 12-4 创建与配置 Hive 服务器

```
<context:property-placeholder location="hadoop.properties,hive.properties" />  
  
<configuration id="hadoopConfiguration">  
  fs.default.name=${hd.fs}  
  mapred.job.tracker=${mapred.job.tracker}  
</configuration>  
  
<hive-server port="${hive.port}" auto-startup="false"  
            configuration-ref="hadoopConfiguration"  
            properties-location="hive-server.properties">
```

<sup>1</sup> <https://cwiki.apache.org/confluence/display/Hive/HiveServer2+Thrift+API>。

```
hive.exec.scratchdir=/tmp/hive/  
</hive-server>
```

hadoop.properties 和 hive.properties 文件是从类路径加载，它们组合起来的值如示例 12-5 所示。可以使用属性文件 hive-server.properties 来配置这个服务器，这些值和放在 hive-site.xml 里面的值一样。

#### 示例 12-5 配置简单 Hive 应用程序的属性

```
hd.fs=dfs://localhost:9000  
mapred.job.tracker=localhost:9001  
hive.host=localhost  
hive.port=10000  
hive.table=passwords
```

### 12.1.3 使用 Hive Thrift 客户端

Hadoop 命名空间可用来创建 Thrift 客户端，如示例 12-6 所示。

#### 示例 12-6 创建并配置 Hive Thrift 客户端

```
<hive-client-factory host="${hive.host}" port="${hive.port}"/>
```

这个命名空间会创建一个 HiveClientFactory 类的实例。调用 HiveClientFactory 的 getHiveClient 方法会返回一个新的 HiveClient 实例。这是 Spring 提供的一种便捷方式，因为 HiveClient 不是线程安全的类，因此在多线程共享的方法中必须要创建新的实例。通过 XML 命名空间，可以对 HiveClient 设置的参数还包括连接的超时时间以及当客户端建立连接时需要执行的一组脚本。为了使用 HiveClient，要创建 HivePasswordRepository 类来执行前一节使用的 password-analysis.hql 脚本，然后对 passwords 表进行查询。在之前的 Hive 服务器配置中加入<context:component-scan> 元素，即可扫描类路径找出带有 Spring 通用@Repository 注解的类进而自动把 HivePasswordRepository 类注册到容器中，如示例 12-7 所示。

#### 示例 12-7 在数据访问层使用 Thrift HiveClient

```
@Repository  
public class HivePasswordRepository implements PasswordRepository {  
  
    private static final Log logger = LoggerFactory.getLog(HivePasswordRepository.class);  
  
    private HiveClientFactory hiveClientFactory;  
    private String tableName;  
  
    // constructor and setters omitted  
  
    @Override  
    public Long count() {  
        HiveClient hiveClient = hiveClientFactory.getHiveClient();  
        try {  
            hiveClient.execute("select count(*) from " + tableName);  
            return Long.parseLong(hiveClient.fetchOne());  
            // checked exceptions  
        } catch (HiveServerException ex) {  
    }
```

```

        throw translateExcption(ex);
    } catch (org.apache.thrift.TException tex) {
        throw translateExcption(tex);
    } finally {
        try {
            hiveClient.shutdown();
        } catch (org.apache.thrift.TException tex) {
            logger.debug("Unexpected exception on shutting down HiveClient", tex);
        }
    }
}

@Override
public void processPasswordFile(String inputFile) {
    // Implementation not shown
}

private RuntimeException translateExcption(Exception ex) {
    return new RuntimeException(ex);
}
}

```

这个示例应用程序的代码位于 `./hadoop/hive`。参考示例目录中的 `readme` 文件可了解更多运行这个示例应用程序的信息。示例应用程序的驱动会调用 `HivePasswordRepository` 的 `processPasswordFile` 方法，接着调用它的 `count` 方法，对于我们的结果集所返回的值为 41。这个示例的错误处理方式是开发数据访问层的最佳实践，它避免了将已检查型异常抛给调用代码。

`HiveTemplate` 帮助类为简化编程式 `Hive` 开发提供了众多的好处。它会将 `HiveClient` 的检查型异常和错误转换成 Spring 便捷的 DAO 异常体系，这意味着调用代码不需要知道 `Hive` 细节。`HiveClient` 也不是线程安全的，因此同其他 Spring 模板类类似，`HiveTemplate` 会对底层资源提供线程安全的访问，因此不需要处理复杂的 `HiveClient API`，只需将精力集中在执行 HSQL 和获取结果集上。为创建 `HiveTemplate`，要使用 XML 命名空间以及一个可选的 `HiveClientFactory` 引用。示例 12-8 是新 `PasswordRepository` 实现的最小配置，这个实现使用了 `HiveTemplate`。

### 示例 12-8 配置 `HiveTemplate`

```

<context:property-placeholder location="hadoop.properties,hive.properties"/>

<configuration>
    fs.default.name=${hd.fs}
    mapred.job.tracker=${mapred.job.tracker}
</configuration>

<hive-client-factory host="${hive.host}" port="${hive.port}" />

<hive-template/>

```

在 `<hive-template/>` XML 命名空间中，可以使用 `hive-client-factory-ref` 元素通过名称显式地引用 `HiveClientFactory`。借助于 `HiveTemplate` 使得实现 `HiveTemplatePassword`

Repository 类更为容易，如示例 12-9 所示。

#### 示例 12-9 借助 HiveTemplate 实现 HiveTemplatePasswordRepository

```
@Repository
public class HiveTemplatePasswordRepository implements PasswordRepository {

    private HiveOperations hiveOperations;
    private String tableName;

    // constructor and setters omitted

    @Override
    public Long count() {
        return hiveOperations.queryForLong("select count(*) from " + tableName);
    }

    @Override
    public void processPasswordFile(String inputFile) {
        Map parameters = new HashMap();
        parameters.put("inputFile", inputFile);
        hiveOperations.query("classpath:password-analysis.hql", parameters);
    }
}
```

注意，HiveTemplate 类实现了 HiveOperations 接口。这是 Spring 模板类通用的实现方式，因为接口很容易模拟（mock）或提供存根（stub）实现，所以便于进行单元测试。借助 Spring 的 Resource 抽象，HiveTemplate 的查询方法允许将脚本地址的引用传递进来，这具有非常大的灵活性，可以从类路径、文件或 HTTP 加载 InputStream。这个查询方法的第二个参数用来替换脚本变量的值。HiveTemplate 也提供了一个很便利的回调方法，它会传递给你一个托管的 HiveClient 实例。类似于 Spring 中的其他模板类，如果这些便利的方法不能满足需求，则可以通过回调使用底层的 API 了，不过依然能够从模板的异常转换以及资源管理等特性中收益。

Spring for Apache Hadoop 还提供了一个类似于 JobRunner 的 HiveRunner 帮助类，可在运行 HiveQL 脚本之前和之后执行 HDFS 脚本操作。可以使用 XML 命名空间元素<hive-runner/>来配置这个 Runner。

### 12.1.4 使用 Hive JDBC 客户端

通过 JDBC 对 Hive 的支持，可以使用熟悉的 Spring JdbcTemplate 来与 Hive 交互。Hive 提供了 HiveDriver 类，它可以传入 Spring 的 SimpleDriverDataSource，如示例 12-10 所示。

#### 示例 12-10 创建和配置基于 Hive JDBC 的访问

```
<bean id="hiveDriver" class="org.apache.hadoop.hive.jdbc.HiveDriver" />

<bean id="dataSource" class="org.springframework.jdbc.datasource.SimpleDriverDataSource">
    <constructor-arg name="driver" ref="hiveDriver" />
    <constructor-arg name="url" value="${hive.url}" />
```

```
</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.simple.JdbcTemplate">
    <constructor-arg ref="dataSource" />
</bean>
```

SimpleDriverDataSource 提供标准 JDBC DataSource 接口的简单实现，它需要一个 java.sql.Driver 实现。每当 DataSource 的 getConnection 方法被调用时，它都会返回一个新的连接。这对大部分的 Hive JDBC 应用程序已经足够了，因为与执行 Hive 操作所耗费的时间长度相比，建立连接的开销已经很小了。如果需要连接池，也可以简单地改变配置来使用 Apache Commons DBCP 或者 c3p0 连接池。

JdbcTemplate 提供了多种 ResultSet 到 POJO 映射的功能并能将错误码转换为 Spring 便利的 DAO（数据访问对象）异常体系。自 Hive 0.10 开始，JDBC 驱动可产生有意义的错误码，可以轻松的区别 Spring 的 TransientDataAccessException 与 NonTransientDataAccessException。暂时性的（transient）异常表示可以进行重试操作而且可能会成功，而非暂时性的（nontransient）异常表示重试操作不会成功。

使用 JdbcTemplate 的 PasswordRepository 实现如示例 12-11 所示。

#### 示例 12-11 以 JdbcTemplate 实现 PasswordRepository

```
@Repository
public class JdbcPasswordRepository implements PasswordRepository {

    private JdbcOperations jdbcOperations;
    private String tableName;

    // constructor and setters omitted

    @Override
    public Long count() {
        return jdbcOperations.queryForLong("select count(*) from " + tableName);
    }

    @Override
    public void processPasswordFile(String inputFile) {
        // Implementation not shown.
    }
}
```

由于需要替换脚本中的变量，processPasswordFile 的实现显得有些冗长。请参考示例代码获取更详细的信息。注意 Spring 提供的工具类 SimpleJdbcTestUtils 是测试包中的一部分，它通常用来对关系型数据库执行 DDL 脚本，但是当需要执行没有变量替换的 HiveQL 脚本时，它就可以派上用场了。

## 12.1.5 使用 Hive 分析 Apache 日志文件

接下来，使用 Hive 对 Apache HTTPD 的日志文件进行简单分析。运行这个分析的配置与之前使用 HiveTemplate 分析 password 文件的配置类似。示例 12-12 中的 HiveQL 脚本会产生一个包含每个 URL 累计点击数的文件。它也会提取最少与最多点击数，并提供一个表格，以简单图表的形式展现点击数的分布情况。

### 示例 12-12 基本 Apache HTTPD 日志分析的 HiveQL

```
ADD JAR ${hiveconf:hiveContribJar};

DROP TABLE IF EXISTS apachelog;
CREATE TABLE apachelog(remoteHost STRING, remoteLogname STRING, user STRING, time STRING,
                      method STRING, uri STRING, proto STRING, status STRING,
                      bytes STRING, referer STRING, userAgent STRING)
ROW FORMAT SERDE 'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'
WITH SERDEPROPERTIES (
    "input.regex" = "^([^\n]* )+([^\n]* )+([^\n]* )+\\[([^\n]* )\\]\\]+\\\"([^\n]* ) ([^\n]* )\n([^\n]* )\\\" ([^\n]* ) ([^\n]* ) (?:\\\"-\\\")*\\\"(.*)\\\" (.*)$",
    "output.format.string" = "%1$s %2$s %3$s %4$s %5$s %6$s %7$s %8$s %9$s %10$s %11$s")
STORED AS TEXTFILE;

LOAD DATA LOCAL INPATH "${hiveconf:localInPath}" INTO TABLE apachelog;

-- basic filtering
-- SELECT a.uri FROM apachelog a WHERE a.method='GET' AND a.status='200';

-- determine popular URLs (for caching purposes)

INSERT OVERWRITE LOCAL DIRECTORY 'hive_uri_hits' SELECT a.uri, "\t", COUNT(*)
FROM apachelog a GROUP BY a.uri ORDER BY uri;

-- create histogram data for charting, view book sample code for details
```

这个示例使用了工具类库 `hive-contrib.jar`，其中包含可读取和解析 Apache 日志文件格式的序列化器 (serializer) / 反序列化器 (deserializer)。`hive-contrib.jar` 可由 Maven 中心库下载或直接通过源码构建。我们已经将 `hive-contrib.jar` 的位置参数化，但另外一个可选的方案就是也可以将这个 jar 复制到所有执行任务的机器中，并将其放到 Hadoop 库目录下。执行结果将会存储在本机目录内。这个程序的代码位于 `/hadoop/hive`。请参考示例目录内的 `readme` 文件来获取更详细的程序运行信息。在 `hive_uri_hits` 目录中可以看到示例的数据内容，如示例 12-13 所示。

### 示例 12-13 每个 URI 的点击统计

```
/archives.html 3
/archives/000005.html 2
/archives/000021.html 1
...
/archives/000055.html 1
/archives/000064.html 2
```

`hive_histogram` 目录的内容显示有 1 个 URI 已经被请求了 22 次，有 3 个 URL 各被单击了 4 次，有 74 个 URL 只被单击了 1 次，这告诉我们哪些 URL 缓存起来可以

带来更大的好处。示例应用程序介绍了两种执行 Hive 命令的方法，分别使用了 HiveTemplate 和 HiveRunner。HiveRunner 的配置如示例 12-14 所示。

#### 示例 12-14 使用 HiveRunner 运行 Apache 日志文件分析

```
<context:property-placeholder location="hadoop.properties,hive.properties"/>

<configuration>
  fs.default.name=${hd.fs}
  mapred.job.tracker=${mapred.job.tracker}
</configuration>
<hive-server port="${hive.port}" properties-location="hive-server.properties"/>

<hive-client-factory host="${hive.host}" port="${hive.port}"/>

<hive-runner id="hiveRunner" run-at-startup="false" >
  <script location="apache-log-simple.hql">
    <arguments>
      hiveContribJar=${hiveContribJar}
      localInPath=".//data/apache.log"
    </arguments>
  </script>
</hive-runner>
```

这个结果集非常小，可以使用 UNIX 命行工具来分析，不过 Hadoop 能够把要分析的数据扩展到非常大的集合。Hadoop 也能够让我们以很低的成本存储原始数据，因此可以重新进行分析而不会丢失任何信息，这种信息丢失通常是因为保持历史数据的摘要所导致的。

## 12.2 使用 Pig

要编写 MapReduce 应用来分析存储在 HDFS 中的数据，Pig 是另外一个可选方案。Pig 应用程序使用 Pig Latin 语言编写，Pig Latin 是一种高级数据处理语言，它更多地秉持了 sed 或 awk 的精神而不是像 Hive 那样提供类 SQL 的语言。Pig Latin 脚本描述了一系列的步骤，每一个步骤都会对集合中的数据项（item）进行转换。简单的步骤可以包括加载、过滤以及保存数据，也支持更复杂的操作，如基于相同的值对两个数据项进行连接操作。Pig 可以使用用户自定义函数（User-Defined Function，UDF）来扩展，它们封装了常用的功能，如特定的算法或读写常用的数据格式，如 Apache HTTPD 日志文件。PigServer 负责将 Pig Latin 脚本转换成基于 MapReduce API 的多个 Job 并执行它们。

通常 Pig Latin 脚本开发的入门方式是使用 Pig 附带的 Grunt 控制台进行交互。可以采用两种不同的运行模式来执行脚本。第一种是 LOCAL 模式，它会使用存储在本地文件系统的数据以及嵌入式版本的 Hadoop 在本地执行 MapReduce Job。第二种是 MAPREDUCE 模式，使用 HDFS 并且在 Hadoop 集群上运行 MapReduce Job。使

用本地文件系统，可以使用少量的数据集以迭代的方式开发脚本。对脚本功能比较满意时，可以轻松地切换到集群中并用完整的数据集运行同样的脚本。除了使用交互式控制台或者在命令行运行 Pig 外，也可以将 Pig 嵌入到应用程序中。PigServer 类封装了以编程式连接 Pig、执行脚本以及注册函数的方法。

Spring for Apache Hadoop 能够很容易地将 PigServer 嵌入到应用程序中并以编码的方式运行 Pig Latin 脚本。因为 Pig Latin 没有像条件分支（if-else）或循环这样的控制流语句，因此 Java 可以用来填充这个空白。以编程的方式使用 Pig 时，也可以使用 Spring Integration 来执行 Pig 脚本以响应事件驱动的活动，或是使用 Spring Batch 处理较大型的工作流程。

为了熟悉 Pig，我们首先将编写一个基础的应用程序，它会使用 Pig 的命令行工具来分析 UNIX password 文件。然后我们将介绍如何借助 Spring for Apache Hadoop 开发使用 Pig 的 Java 应用程序。要获取 Pig 和 Pig Latin 安装、运行以及开发的更详细信息，可以参考项目的站点 (<http://pig.apache.org/>) 以及《Programming Pig》(O'Reilly 出版) 一书。

### 12.2.1 Hello World

作为 Hello World 级别的练习，我们将对 UNIX password 文件进行一个小型的分析，分析的目标在于建立一个特定 Shell（例如 bash 或 sh）使用者数量的报表。通过熟悉的 UNIX 工具，可以很容易地看到正在使用 bash Shell 的人数，如示例 12-15 所示。

#### 示例 12-15 使用 UNIX 工具统计 bash Shell 用户数

```
$ $ more /etc/passwd | grep /bin/bash
root:x:0:0:root:/root:/bin/bash
couchdb:x:105:113:CouchDB Administrator,,,,:/var/lib/couchdb:/bin/bash
mpollack:x:1000:1000:Mark Pollack,,,,:/home/mpollack:/bin/bash
postgres:x:116:123:PostgreSQL administrator,,,,:/var/lib/postgresql:/bin/bash
testuser:x:1001:1001:testuser,,,,:/home/testuser:/bin/bash

$ more /etc/passwd | grep /bin/bash | wc -l
5
```

为了使用 Pig 进行类似的分析，我们先将/etc/password 文件加载到 HDFS 中，如示例 12-16 所示。

#### 示例 12-16 复制/etc/password 文件到 HDFS 中

```
$ hadoop dfs -copyFromLocal /etc/passwd /test/passwd
$ hadoop dfs -cat /test/passwd
```

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
...
...
```

要安装 Pig，首先要从 Pig 主页下载 (<http://pig.apache.org/>)。安装完发布版后，需要将 Pig 发布版的 bin 目录加入到路径之中，并将环境变量 PIG\_CLASSPATH 指向 Hadoop 配置目录（例如，`export PIG_CLASSPATH=$HADOOP_INSTALL/conf/`）。

现在以 LOCAL 模式启动 Pig 交互控制台 Grunt，并执行一些 Pig Latin 命令，如示例 12-17 所示。

### 示例 12-17 使用 Grunt 执行 Pig Latin 命令

```
$ pig -x local
grunt> passwd = LOAD '/test/passwd' USING PigStorage':' \
    AS (username:chararray, password:chararray, uid:int, gid:int, userinfo:chararray,
        home_dir:chararray, shell:chararray);
grunt> grouped_by_shell = GROUP passwd BY shell;
grunt> password_count = FOREACH grouped_by_shell GENERATE group, COUNT(passwd);
grunt> STORE password_count into '/tmp/passwordAnalysis';
grunt> quit
```

由于示例的数据集比较小，所有结果会被调整成一个用制表符分隔的文件，如示例 12-18 所示。

### 示例 12-18 命令行执行结果

```
$ hadoop dfs -cat /tmp/passwordAnalysis/part-r-00000
/bin/sh 18
/bin/bash 5
/bin/sync 1
/bin/false 16
/usr/sbin/nologin 1
```

在 Pig Latin 脚本中进行的数据转换流程如下，脚本的第一行把 HDFS 文件 `/test/passwd` 中的数据加载到 `passwd` 变量中。LOAD 命令会获取文件在 HDFS 中的位置以及格式，文件中的行会使用这个格式进行分解进而创建数据集（也称为 Pig 关联）。在这个示例中，我们使用 `PigStorage` 函数来加载文本文件并用冒号来分割文件中的字段。Pig 可以将模式应用到文件中的列上，这是通过定义每列的名称与数据类型实现的。将输出结果集指定给 `passwd` 变量后，可以操作数据集以将其转换为其他衍生的数据集了。我们可使用 GROUP 操作来创建数据集 `grouped_by_shell`。`grouped_by_shell` 数据集会使用 Shell 的名字作为 key，并且还会包含 `passwd` 数据集中所有具有该 Shell 值的记录所组成的集合。如果使用 DUMP 操作查看以 `/bin/bash` 为键的 `grouped_by_shell` 数据集中的内容，将会看到如示例 12-19 所示的结果。

### 示例 12-19 group-by-shell 结果集

```
(/bin/bash,{(testuser,x,1001,1001,testuser,,,/home/testuser,/bin/bash),
            (root,x,0,0,root,/root,/bin/bash),
            (couchdb,x,105,113,CouchDB Administrator,,,/var/lib/couchdb,/bin/bash),
            (mpollack,x,1000,1000,Mark Pollack,,,/home/mpollack,/bin/bash),
            (postgres,x,116,123,PostgreSQL administrator,,,/var/lib/postgresql,/bin/bash)
)})
```

Shell 名称作为键，值是具有相同键的 password 记录所组成集合或 bag。在下一行，表达式 FOREACH grouped\_by\_shell GENERATE 会操作 grouped\_by\_shell 数据集的每一行记录并且生成新的记录。所形成的结果集会基于相同的键值对所有记录进行分组并且进行计数。

我们也可以将脚本参数化来避免硬编码，例如，输入和输出的位置。在示例 12-20 中，把所有输入到控制台的命令放入名为 password-analysis.pig 的文件中，通过 inputFile 和 outputDir 变量实现参数化。

### 示例 12-20 Pig 脚本参数化

```
passwd = LOAD '$inputFile' USING PigStorage(':')
    AS (username:chararray, password:chararray, uid:int, gid:int, userinfo:chararray,
        home_dir:chararray, shell:chararray);
grouped_by_shell = GROUP passwd BY shell;
password_count = FOREACH grouped_by_shell GENERATE group, COUNT(passwd);
STORE password_count into '$outputDir';
```

使用 run 命令能够在交互式控制台运行该脚本，如示例 12-21 所示。

### 示例 12-21 在 Grunt 中运行参数化的 Pig 脚本

```
grunt> run -param inputFile=/test/passwd outputDir=/tmp/passwordAnalysis
      password-analysis.pig
grunt> exit

$ hadoop dfs -cat /tmp/passwordAnalysis/part-r-00000
/bin/sh 18
/bin/bash 5
/bin/sync 1
/bin/false 16
/usr/sbin/nologin 1
```

或者直接在命令行中运行脚本，如示例 12-22 所示。

### 示例 12-22 在命令行中运行参数化的 Pig 脚本

```
$ pig -file password-analysis.pig -param inputFile=/test/passwd
      -param outputDir=/tmp/passwordAnalysis
```

## 12.2.2 运行 PigServer

现在我们将以更结构化和程序化的方式来运行 Pig 脚本。通过 Spring for Apache Hadoop，可以很容易地在 Java 应用程序中以声明的方式配置和创建 PigServer，与 Grunt Shell 的底层实现类似。在 Spring 容器中运行 PigServer 时，能够参数化和外部化配置一些属性，这些属性会控制哪个 Hadoop 集群执行脚本、PigServer 的配置以及传入脚本的参数等。Spring 针对 Hadoop 的 XML 命名空间简化了 PigServer 的创建和配置。如示例 12-23 所示，它的配置方式与其他应用的配置方式相同。可选的 Pig 初始化脚本位置可以是任意的 Spring 资源 URL，可位于文件系统、类路径、HDFS 或 HTTP 之中。

### 示例 12-23 配置 PigServer

```
<context:property-placeholder location="hadoop.properties" />

<configuration>
    fs.default.name=${hd.fs}
    mapred.job.tracker=${mapred.job.tracker}
</configuration>

<pig-factory properties-location="pig-server.properties">
    <script location="initialization.pig">
        <arguments>
            inputFile=${initInputFile}
        </arguments>
    </script>
</pig-factory>
```

在示例 12-24 中，脚本通过类路径来定位，要替换的变量包含在属性文件 hadoop.properties 中。

### 示例 12-24 hadoop.properties

```
hd.fs=hdfs://localhost:9000
mapred.job.tracker=localhost:9001
inputFile=/test/passwd
outputDir=/tmp/passwordAnalysis
```

<pig-factory/>命名空间的其他一些属性包括： properties-location，它引用一个属性文件以配置 PigServer 的属性； job-tracker，当与 Hadoop 配置不同的时候，可使用该属性设置 Job Tracker 的位置； job-name，它可以设置 Pig 所创建的 MapReduce Job 的根名称，这样就能很容易地识别它们属于该脚本。

由于 PigServer 类不是线程安全的对象而且在每次执行之后都需要清除一个已建立的状态，<pig-factory/>命名空间会创建一个 PigServerFactory 实例，这样在需要时可以很容易地创建 PigServer 实例。类似于 JobRunner 和 HiveRunner 的用途，PigRunner 帮助类提供了一个便捷的方法，可以重复执行 Pig Job，也可在执行它们之前或之后执行 HDFS 脚本。示例 12-25 为 PigRunner 的配置。

### 示例 12-25 配置 PigRunner

```
<pig-runner id="pigRunner"
            pre-action="hdfsScript"
            run-at-startup="true" >
    <script location="password-analysis.pig">
        <arguments>
            inputDir=${inputDir}
            outputDir=${outputDir}
        </arguments>
    </script>
</pig-runner>
```

我们将 run-at-startup 元素设置为 true，让 Pig 脚本在 Spring 应用上下文启动时执行（默认为 false）。示例应用程序位于目录 hadoop/pig 之中。要运行示例 password 文件分析程序，可运行如示例 12-26 所示的命令。

### 示例 12-26 构建并运行 Pig 示例脚本的命令

```
$ cd hadoop/pig  
$ mvn clean package appassembler:assemble  
$ sh ./target/appassembler/bin/pigApp
```

因为 PigServerFactory 和 PigRunner 是由 Spring 管理的对象，它们也可以注入其他由 Spring 管理的对象中。通常会注入 PigRunner 帮助类来确保每次执行脚本时都会建立新的 PigServer 实例，并且在执行结束后将它的所有资源进行清理。例如，为了在应用的服务层异步运行 Pig Job，我们注入 PigRunner 并使用 Spring 的@Async 注解，如示例 12-27 所示。

### 示例 12-27 依赖注入 PigRunner 并异步执行 Pig Job

```
@Component  
public class AnalysisService {  
  
    private PigRunner pigRunner;  
  
    @Autowired  
    public AnalysisService(PigRunner pigRunner)  
        this.pigRunner = pigRunner;  
    }  
  
    @Async  
    public void performAnalysis() {  
        pigRunner.call();  
    }  
}
```

### 12.2.3 控制运行期脚本的执行

为了可以更多地在运行期控制要执行的 Pig 脚本与传入的参数，可以使用 PigTemplate 类。类似于 Spring 中的其他模板类，PigTemplate 会管理底层资源，一旦配置完成它就是线程安全的，并且会将 Pig 的错误和异常转换为 Spring 便利的 DAO 异常体系 (<http://static.springsource.org/spring/docs/current/spring-framework-reference/html/dao.html>)。通过 Spring 的 DAO 异常体系，可以轻松地使用不同的数据访问技术，不需要捕获特定技术的异常或者查找返回码。Spring DAO 异常体系也会帮助你分别出暂时性的异常和非暂时性的异常。如果出现暂时性的异常，重新执行失败的操作有可能会成功。这个功能是通过在数据访问层使用 Spring AOP(面向切面编程)的重试通知 (advice) 实现的。因为 Spring 的 JDBC 帮助类也会执行相同的异常转换，由 Spring JDBC 所支持的基于 Hive 的数据访问也会映射到 DAO 体系之中。在 Hive 和 Pig 之间切换不是一个轻松的任务，因为分析脚本需要重写，但至少可以按照 Hive 以及 Pig 数据访问层的差异来隔离调用的代码，也可以把调用 Hive 和 Pig 的数据访问层类混合在一起，并且使用一致的错误处理机制。

要配置 PigTemplate，可以和之前一样创建一个 PigServerFactory 定义并且加入

<pig-template>元素，如示例 12-28 所示。可以在 properties-location 元素指定的属性文件中定义 PigServer 的通用配置属性，然后在 DAO 或 Repository 类中引用模板——在本例中为 PigPasswordRepository。

### 示例 12-28 配置 PigTemplate

```
<pig-factory properties-location="pig-server.properties"/>

<pig-template/>

<beans:bean id="passwordRepository"
    class="com.oreilly.springdata.hadoop.pig.PigPasswordRepository">
    <beans:constructor-arg ref="pigTemplate"/>
</beans:bean>
```

PigPasswordRepository（如示例 12-29 所示）可以在运行期传入输入文件。processPasswordFiles 方法展现了如何以编程的方式在 Java 中处理多个文件。例如，或许你想基于一套复杂的规则来选择一组输入文件，而这套规则又不能以 Pig Latin 或 Pig 用户自定义函数来指定。注意，PigTemplate 类实现了 PigOperations 接口。这是 Spring 模板类通用的实现方式，因为接口很容易模拟或提供存根实现，便于进行单元测试。

### 示例 12-29 基于 Pig 的 PasswordRepository

```
public class PigPasswordRepository implements PasswordRepository {

    private PigOperations pigOperations;
    private String pigScript = "classpath:password-analysis.pig";

    // constructor and setters omitted

    @Override
    public void processPasswordFile(String inputFile) {
        Assert.notNull(inputFile);
        String outputDir =
            PathUtils.format("/data/password-repo/output/%1$tY/%1$tm/%1$td/%1$tH/%1$tM/%1$tS");
        Properties scriptParameters = new Properties();
        scriptParameters.put("inputDir", inputFile);
        scriptParameters.put("outputDir", outputDir);
        pigOperations.executeScript(pigScript, scriptParameters);
    }

    @Override
    public void processPasswordFiles(Collection<String> inputFiles) {
        for (String inputFile : inputFiles) {
            processPasswordFile(inputFile);
        }
    }
}
```

Pig 脚本 password-analysis.pig 是通过 Spring 的资源抽象功能加载的，在这个例子中由类路径加载。若要运行使用 PigPasswordRepository 的应用程序，可使用如示例 12-30 所示的命令。

### 示例 12-30 构建并运行使用 PigPasswordRepository 的 Pig 脚本示例

```
$ cd hadoop/pig  
$ mvn clean package appassembler:assemble  
$ sh ./target/appassembler/bin/pigAppWithRepository
```

这个应用程序实际所执行的代码片段如示例 12-31 所示。

### 示例 12-31 使用 PigPasswordRepository

```
PasswordRepository repo = context.getBean(PigPasswordRepository.class);  
repo.processPasswordFile("/data/passwd/input")  
  
Collection<String> files = new ArrayList<String>();  
files.add("/data/passwd/input");  
files.add("/data/passwd/input2");  
repo.processPasswordFiles(files);
```

## 12.2.4 在 Spring Integration 数据管道中调用 Pig 脚本

要在 Spring Integration 数据管道中运行 Pig Latin 脚本，可以参考 Spring Integration 服务催化器（service activator）定义的 PigPasswordRepository，如示例 12-32 所示。

### 示例 12-32 在 Spring Integration 数据管道中调用 Pig 脚本

```
<bean id="passwordService" class="com.oreilly.springdata.hadoop.pig.PasswordService">  
    <constructor-arg ref="passwordRepository" />  
</bean>  
  
<int:service-activator input-channel="exampleChannel" ref="passwordService" />
```

以同步模式还是异步模式来执行服务催化器，取决于所使用的输入通道（channel）类型。如果它是 DirectChannel（默认）将会以同步方式执行；如果它是 ExecutorChannel，则以异步方式执行，并由 TaskExecutor 代理执行。服务类 PasswordService 如示例 12-33 所示。

### 示例 12-33 执行 Pig 分析 Job 的 Spring Integration 服务催化器

```
public class PasswordService {  
  
    private PasswordRepository passwordRepository;  
  
    // constructor omitted  
  
    @ServiceActivator  
    public void process(@Header("hdfs_path") String inputDir) {  
        passwordRepository.processPasswordFile(inputDir);  
    }  
}
```

这个处理方法的参数将从 Spring Integration 消息头中获取，方法参数的值是在消息头中与键 hdfs\_path 相关联的值。如同之前所使用的 FsShellWritingMessageHandler，消息值由 MessageHandler 实现进行填充，而且必须在数据处理管道中的服务催化器执行之前调用。

从创建一个简单的基于 Pig 的应用程序来执行脚本，到使用运行期参数替换来执行脚本，再到使用 Spring Integration 数据管道来执行脚本，本节的示例展示了一个循序渐进的演进过程。13.2 小节“Hadoop Workflows”中会介绍如何在步骤较多时使用 Spring Batch 来协调 Pig 脚本的执行过程。

## 12.2.5 使用 Pig 分析 Apache 日志文件

下面对 Apache HTTPD 日志文件执行简单的分析，并且展示如何为 Apache 日志文件使用自定义的加载器。如示例 12-34 所示，运行这个分析的配置类似于之前分析 password 文件时的配置。

### 示例 12-34 使用 Pig 分析 Apache HTTD 日志文件的配置

```
<context:property-placeholder location="hadoop.properties,pig-analysis.properties"/>

<configuration>
    fs.default.name=${hd.fs}
    mapred.job.tracker=${mapred.job.tracker}
</configuration>

<pig-factory/>

<script id="hdfsScript" location="copy-files.groovy">
    <property name="localSourceFile" value="${pig.localSourceFile}" />
    <property name="inputDir" value="${pig.inputPath}" />
    <property name="outputDir" value="${pig.outputPath}" />
</script>
```

*copy-files.groovy* 脚本的职责是将示例日志文件复制到 HDFS 中，并删除输出路径的内容。

Pig 脚本会产生一个文件，里面包含了每个 URL 的累计点击数，供更全面的分析使用。脚本会提取最少与最多的点击数并形成一个简单的表格用来显示简单的点击数分布图。Pig 简化了在各种条件下的数据过滤和预处理。例如，只保留成功的 GET 请求并删除针对图片的 GET 请求。Pig 脚本与 Spring 应用上下文启动时运行脚本的 PigRunner 配置分别如示例 12-35 与示例 12-36 所示。

### 示例 12-35 基本 Apache HTTPD 日志分析的 Pig 脚本

```
REGISTER $piggybanklib;
DEFINE LogLoader org.apache.pig.piggybank.storage.apachelog.CombinedLogLoader();
logs = LOAD '$inputPath' USING LogLoader AS (remoteHost, remoteLogname, user, time, \
method, uri, proto, status, bytes, referer, userAgent);

-- determine popular URLs (for caching purposes for example)
byUri = ORDER logs BY uri;
byUri = GROUP logs BY uri;

uriHits = FOREACH byUri GENERATE group AS uri, COUNT(logs.uri) AS numHits;
STORE uriHits into '$outputPath/pig_uri_hits';

-- create histogram data for charting, view book sample code for details
```

### 示例 12-36 用于分析 Apache HTTPD 文件的 PigRunner 配置

```
<pig-runner id="pigRunner"
    pre-action="hdfsScript"
    run-at-startup="true" >
<script location="apache-log-simple.pig">
<arguments>
    piggybanklib=${pig.piggybanklib}
    inputPath=${pig.inputPath}
    outputPath=${pig.outputPath}
</arguments>
</script>
</pig-runner>
```

Pig 脚本的参数指定了 jar 文件的位置，这个 jar 文件包含了用来读取与解析 Apache 日志文件的自定义加载器以及输入与输出的路径位置。为了解析 Apache HTTPD 日志文件，我们使用 Pig 发布版提供的自定义加载器，它作为 Piggybank 项目的一部分以源码的方式发布，编译后的 Piggybank jar 文件位于示例应用程序的 lib 目录。

## 12.3 使用 HBase

HBase 是一个分布式的、面向列的数据库。它将数据建模成表并存储在 HDFS 中，并且具有可伸缩性以支持拥有数十亿行以及数百万列的表。如同 Hadoop 的 HDFS 和 MapReduce，HBase 也是模仿 Google 研发的技术来设计的。HBase 来源于 Google 的 BigTable 技术，该技术起源于 2006 年的一篇研究论文(<http://research.google.com/archive/bigtable.html>)。与 MapReduce 不同，HBase 提供了基于键的接近实时的数据访问，因此可用于交互式的、非批量操作的应用程序。

HBase 数据模型是由表组成的，这个表由唯一的键进行标识，并包含相关联的列。这些列组合在一起形成了列族 (column family)，所以经常一起访问的数据可以在磁盘中存储在一起以增加 I/O 的性能。存储于列的数据是一个键/值对集合，而不是像关系型数据库中常见的单个值。列族需要使用模式 (schema) 来描述并且需要预先定义，但以键/值对集合形式存储的值则不需要，这使得系统在演化上具有了很大的灵活性。数据建模有很多更深入的细节，必须彻底理解才能高效地使用 HBase。《*HBase: The Definitive Guide*》(O'Reilly 出版) 是一本很好的 HBase 参考指南，它详细介绍了 HBase 的数据建模、架构、API 以及管理方式。

Spring for Apache Hadoop 为 HBase 应用的开发提供了一些基础却很便利的支持，可以很容易地配置 HBase 连接，它为 HBase 表提供了线程安全的数据访问方式，以及一个轻量级的对象-列数据映射功能。

### 12.3.1 Hello World

HBase 可从 HBase 主页 (<http://hbase.apache.org/>) 下载。安装完发布版之后，执行

bin 目录下的 start-hbase.sh 脚本即可启动 HBase 服务。如同 Pig 和 Hive，HBase 也附带了一个交互式控制台，可以在 bin 目录下执行 hbase shell 将它启动。

进入交互控制台之后，即可开始建立表、定义列族以及为特定的列族增加行数据。示例 12-37 展示了以下功能：创建用户表、插入示例数据、根据键获取数据以及删除一行数据。

### 示例 12-37 使用 HBase 交互式控制台

```
$ ./bin/hbase shell
> create 'users', { NAME => 'cfInfo' }, { NAME => 'cfStatus' }
> put 'users', 'row-1', 'cfInfo:qUser', 'user1'
> put 'users', 'row-1', 'cfInfo:qEmail', 'user1@yahoo.com'
> put 'users', 'row-1', 'cfInfo:qPassword', 'user1pwd'
> put 'users', 'row-1', 'cfStatus:qEmailValidated', 'true'
> scan 'users'
ROW                  COLUMN+CELL
row-1                column=cfInfo:qEmail, timestamp=1346326115599, value=user1
@ yahoo.com
row-1                column=cfInfo:qPassword, timestamp=1346326128125, value=us
er1pwd
row-1                column=cfInfo:qUser, timestamp=1346326078830, value=user1
row-1                column=cfStatus:qEmailValidated, timestamp=1346326146784,
value=true
1 row(s) in 0.0520 seconds
> get 'users', 'row-1'
COLUMN              CELL
cfInfo:qEmail       timestamp=1346326115599, value=user1@yahoo.com
cfInfo:qPassword    timestamp=1346326128125, value=user1pwd
cfInfo:qUser        timestamp=1346326078830, value=user1
cfStatus:qEmailValid timestamp=1346326146784, value=true
ated
4 row(s) in 0.0120 seconds

> deleteall 'users', 'row-1'
```

在这里创建了名为 cfInfo 和 cfStatus 的两个列族。键的名字在 HBase 被称为限定词 (qualifiers)，存储于 cfInfo 列族的是 username、email 和 password。cfStatus 列族存储其他的数据，这些数据通常不会与存储在 cfInfo 列族中的数据一起访问。例如，将 email 地址验证程序的状态放在 cfStatus 列族，但其他数据（例如使用者是否参与任何在线调查）也是可以包含进来的可选内容。deleteall 命令用来删除特定表和行中的所有数据。

### 12.3.2 使用 HBase Java 客户端

我们有很多可选的客户端 API 与 HBase 交互。本节所使用的是 Java 客户端，但也可以使用 REST、Thrift 以及 Avro 客户端。HTable 类是 Java 与 HBase 交互的主要方式。借助它，可以使用 Put 类将数据放入表中，使用 Get 类来根据键取出数据并使用 Delete 类来删除数据。查询数据则使用 Scan 类，可以指定键的范围以及过滤条件。示例 12-38 以 user1 作为键将一行数据放入之前建立的用户表中。

### 示例 12-38 HBase Put API

```
Configuration configuration = new Configuration(); // Hadoop configuration object
HTable table = new HTable(configuration, "users");

Put p = new Put(Bytes.toBytes("user1"));
p.add(Bytes.toBytes("cfInfo"), Bytes.toBytes("qUser"), Bytes.toBytes("user1"));
p.add(Bytes.toBytes("cfInfo"), Bytes.toBytes("qEmail"), Bytes.toBytes("user1@yahoo.com"));
p.add(Bytes.toBytes("cfInfo"), Bytes.toBytes("qPassword"), Bytes.toBytes("user1pwd"));
table.put(p);
```

HBase API 要求以字节数组的形式来使用数据，而不是其他的原生类型。HTable 也不是线程安全的类，需要小心管理它所使用的底层资源并捕获 HBase 的异常。Spring 的 HBaseTemplate 类为使用 HBase 提供了较高层次的抽象。与其他 Spring 模板类一样，一旦创建之后，它是线程安全的并且可将异常转换到 Spring 便利的数据访问异常体系。类似 JdbcTemplate，它提供了一系列的回调接口，如 TableCallback、RowMapper 与 ResultsExtractor，这些接口可以封装常用的功能，例如将 HBase 的结果集映射到 POJO。

TableCallback 回调接口为 HBaseTemplate 的功能奠定了基础。它可以执行表查询、应用配置设置（例如何时刷新数据）、关闭表以及将任何抛出的异常转换为 Spring DAO 异常体系。RowMapper 回调接口的用途是将 HBase 的 ResultScanner 查询得到的一行记录映射成 POJO。HBaseTemplate 有几个重载的 find 方法，它们使用附加的条件并自动迭代 HBase 的 ResultScanner “结果集” 对象，将每一行数据转换成一个 POJO，并且返回已映射对象的列表。请查看 Javadoc API (<http://docs.spring.io/spring-hadoop/docs/current/api/>) 以获取更详细的信息。若想要进一步控制映射程序（例如，数据行并不直接映射到一个 POJO 上时），ResultsExtractor 接口会传递给你一个 ResultScanner 对象，据此可以执行自定义的结果集迭代程序。

创建与配置 HBaseTemplate 的方式是先创建一个 HBaseConfiguration 对象并将它传到 HBaseTemplate。示例 12-39 演示了如何使用 Spring 的 Hadoop XML 命名空间来配置 HBaseTemplate，而在纯 Java 代码中，也很容易以编程式的方式实现。

### 示例 12-39 配置 HBaseTemplate

```
<configuration>
  fs.default.name=hdfs://localhost:9000
</configuration>

<hbase-configuration configuration-ref="hadoopConfiguration" />

<beans:bean id="hbaseTemplate" class="org.springframework.data.hadoop.hbase.HbaseTemplate">
  <beans:property name="configuration" ref="hbaseConfiguration" />
</beans:bean>
```

示例 12-40 说明了如何使用基于 HBaseTemplate 的 UserRepository 类来查找所有的用户，并将用户添加到 HBase 中。

## 示例 12-40 基于 HBaseTemplate 的 UserRepository 类

```
@Repository
public class UserRepository {

    public static final byte[] CF_INFO = Bytes.toBytes("cfInfo");

    private HbaseTemplate hbaseTemplate;
    private String tableName = "users";
    private byte[] qUser = Bytes.toBytes("user");
    private byte[] qEmail = Bytes.toBytes("email");
    private byte[] qPassword = Bytes.toBytes("password");

    // constructor omitted

    public List<User> findAll() {
        return hbaseTemplate.find(tableName, "cfInfo", new RowMapper<User>() {
            @Override
            public User mapRow(Result result, int rowNum) throws Exception {
                return new User(Bytes.toString(result.getValue(CF_INFO, qUser)),
                               Bytes.toString(result.getValue(CF_INFO, qEmail)),
                               Bytes.toString(result.getValue(CF_INFO, qPassword)));
            }
        });
    }

    public User save(final String userName, final String email, final String password) {
        return hbaseTemplate.execute(tableName, new TableCallback<User>() {
            public User doInTable(HTable table) throws Throwable {
                User user = new User(userName, email, password);
                Put p = new Put(Bytes.toBytes(user.getName()));
                p.add(CF_INFO, qUser, Bytes.toBytes(user.getName()));
                p.add(CF_INFO, qEmail, Bytes.toBytes(user.getEmail()));
                p.add(CF_INFO, qPassword, Bytes.toBytes(user.getPassword()));
                table.put(p);
                return user;
            }
        });
    }
}
```

在这个示例中，使用了匿名内部类来实现 TableCallback 与 RowMapper 接口，但是通常的实现模式是建立一个独立的类，这让你可以在应用程序的各个部分重用映射逻辑。可以使用 HBase 开发更多的功能，让它尽可能与 Spring 对 MongoDB 的支持那样拥有丰富的功能，然而在编写本书的时候，我们已经看到有些人开始研究 Spring Hadoop 与 HBase 的交互以简化 HBase 应用开发的过程。此外，HBase 支持一致的配置与编程模型，可以将其延伸到 Spring Data 和其他 Spring 相关的项目之中。

# 使用 Spring Batch 和 Spring Integration 创建大数据管道

Spring for Apache Hadoop 的目标在于简化 Hadoop 应用程序的开发。Hadoop 所包含的功能远远超出了在 wordcount 样例中所展示的那样——仅执行单一的 MapReduce Job 并将几个文件在 HDFS 中移进移出。搭建现实世界里的 Hadoop 应用还需要非常多的功能，包括收集事件驱动数据、使用像 Pig 这样的程序语言来编写数据分析 Job、调度、链接多个分析 Job，以及在 HDFS 与其他系统如数据库或传统的文件系统之间移动大量数据。

Spring Integration 提供了协调事件驱动活动的基础功能，例如，传送日志文件、处理事件流、实时分析或者触发批量数据处理分析 Job 的执行。Spring Batch 提供了协调工作流中粗粒度（coarse-grained）步骤（step）的框架，这其中包含了基于 Hadoop 的步骤以及 Hadoop 之外的步骤。同时 Spring Batch 也提供了高效的数据处理能力，可在不同的数据源与 HDFS 之间移动数据，例如 Flat 文件、关系型数据或 NoSQL 数据库。

Spring for Apache Hadoop 与 Spring Integration 和 Spring Batch 的结合提供了全面且一致的编程模型，以用于实现 Hadoop 应用程序，这类应用可以涵盖各种类型的功能。另一个产品 Splunk，也需要大量的功能来构建现实世界中大数据管道的解决方案。Spring 对 Splunk 的支持可帮助你创建复杂的 Splunk 应用，并且可融合这两种技术以提供解决方案。

## 13.1 收集并将数据加载到 HDFS

截至目前，我们所介绍的示例都是将一组固定存储在本地目录的数据文件复制到

HDFS 中。实际上，被加载到 HDFS 的文件会由其他程序持续生成，例如 Web 服务器。本地目录将被不断产生的日志文件所填充，日志文件一般遵循如 myapp-timestamp.log 的命名规范。日志文件通常会由远程的机器持续产生，如 Web 农场（Web Farm），这些日志文件需要传送到另外一台机器并加载到 HDFS 中。我们可以通过 Spring Integration 结合 Spring for Apache Hadoop 来实现这些场景。

本章首先简要地介绍了 Spring Integration，然后针对刚才所描述的每个场景实现应用程序。除此之外，本章还将介绍如何使用 Spring Integration 对来自事件流的数据进行处理并加载到 HDFS 之中。最后，本章介绍 Spring Integration 的特性：通过 JMX（Java Management Extensions，Java 管理扩展）和 HTTP 对这些应用程序启用丰富的运行时管理功能。

### 13.1.1 Spring Integration 介绍

Spring Integration 是一个基于 Apache 2 许可的开源项目，该项目起源于 2007 年，基于已建立的企业集成模式（Enterprise Integration Pattern，<http://www.eaipatterns.com/>）来编写应用程序。这些模式提供了关键的构件块来结合新系统和现有系统开发集成应用。这些模式是基于消息传递模型而建立的，消息在应用内部或外部系统之间进行交换。采用消息模型可带来许多好处，例如，可让组件之间实现逻辑和物理的解耦，消息的消费者不需要了解生产者。这种解耦使得构建集成应用变得更为容易，因为应用开发时，可以将这些独立的组件装配起来。消息模型也使得应用程序的测试变得更简便，因为可以先针对单个构件进行测试，这样在开发初期就可以发现错误，而不需要等到在分布式系统测试时才发现，到那个时候追踪错误根源将会变得非常困难。图 13-1 展示了 Spring Integration 应用的关键构件以及它们之间如何进行通信的。

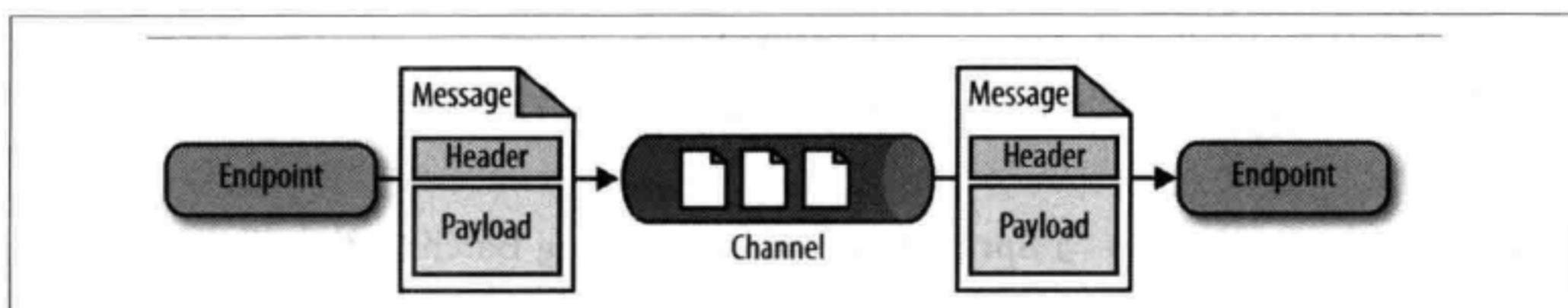


图 13-1 Spring Integration 应用的构件

端点（endpoint）是消息的生产者或是消费者，它们通过通道（channel）进行连接。消息是一个简单的数据结构，头信息（header）中包含了键/值对，负载（payload）中可以包含任意的对象类型。端点可能是与外部系统通信的适配器（adapter），例如 Email、FTP、TCP、JMS、RabbitMQ 或者 syslog，也可能是消息从一个通道转移到另一个通道时对消息所采取的操作。

Spring Integration 支持通用的消息操作，包括：根据消息头路由到一个或多个通道、将负载数据由字符串类型转换成复杂的数据类型并且支持对消息进行过滤，这样只有符合过滤条件的消息才会传递到下游通道。如图 13-2 所示的示例来自于金融服务行业的 Spring/C24 (<http://www.c24.biz/>) 合作项目，展示了 Spring Integration 可构建的数据处理通道类型。

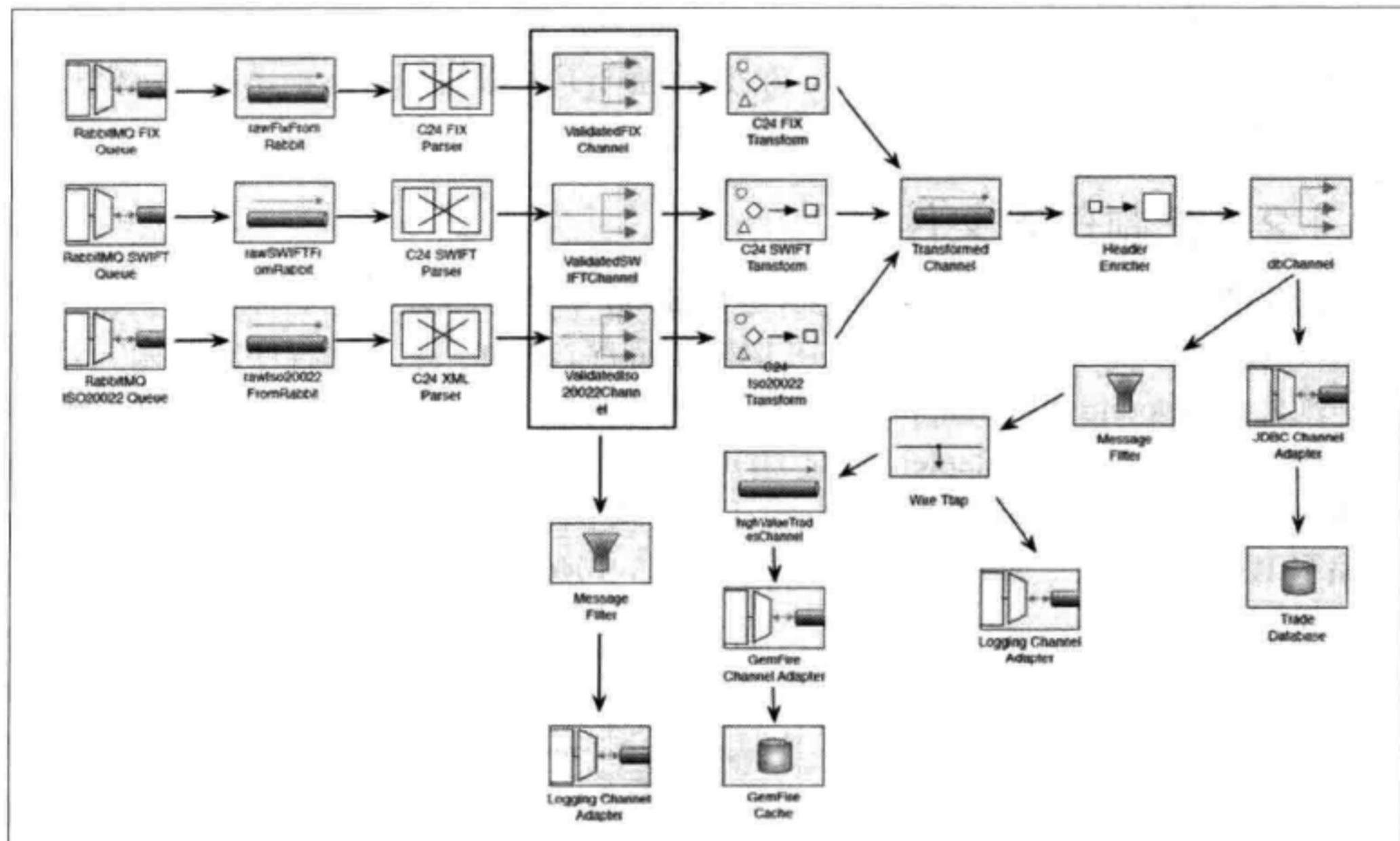


图 13-2 Spring Integration 处理管道

这张图的左半部分展示了金融交易消息会被 3 个 RabbitMQ 适配器接收，它们分别对应 3 个外部的交易数据来源。然后这些消息会被解析、验证并且转换成规范的数据格式。需要注意的是，这些格式并不一定是 XML，通常会是 POJO。接着消息头会被扩充，然后将这条交易数据储存到关系型数据库中，同时也传递到过滤器中。过滤器只会选择高额的交易，随后将其放入可进行实时处理的 GemFire 数据网格中。我们可以通过 XML 或者 Scala 以声明式定义这个处理管道，尽管大部分应用程序可以采用声明式配置，但是一些组件为了便于进行单元测试，需要编写成 POJO。

除了端点、通道以及消息，Spring Integration 的另一个重要组件是它的管理功能。可以很容易地通过 JMX 暴露位于数据管道中的所有组件，并且执行诸如停止与启动适配器的操作。控制总线（control bus）组件可以发送一小段代码（例如使用 Groovy）来进行复杂的操作以修改系统的状态，例如改变过滤规则或启动与停止适配器。控制总线随后会被连接到一个中间件适配器，这样它就可以接收要执行的代

码，常见的可选方案是 HTTP 和面向消息的中间件适配器。

我们不会对 Spring Integration 的内部原理进行太深入的探讨，也不会涉及适配器所提供的每一个功能，但是你应该深刻体会如何组合使用 Spring Integration 与 Spring for Apache Hadoop 创建复杂的数据管道解决方案。这里的示例程序中包含了使用 HDFS 时的一些自定义代码，根据计划这些代码将纳入 Spring Integration 项目中。如果想进一步了解 Spring Integration 的其他信息，可以访问项目网站 (<http://www.springsource.org/spring-integration>)，在这里包含了大量的参考文档链接、示例程序以及 Spring Integration 相关参考书籍的链接。

### 13.1.2 复制日志文件

在日志文件持续产生时，将它们复制到 Hadoop 中是一项很常见的任务。创建两个将持续生成的日志文件加载到 HDFS 的应用程序，其中一个应用程序会使用入站文件适配器 (inbound file adapter) 来轮询目录里的文件，另一个则轮询 FTP 站点。出站适配器(Outbound Adapter)会对 HDFS 进行写操作，它是使用 Spring for Apache Hadoop 所提供的 FsShell 类实现的，关于这个类的介绍请参考 11.5 小节的“在 JVM 中编写 HDFS 脚本”。这个数据管道图如图 13-3 所示。

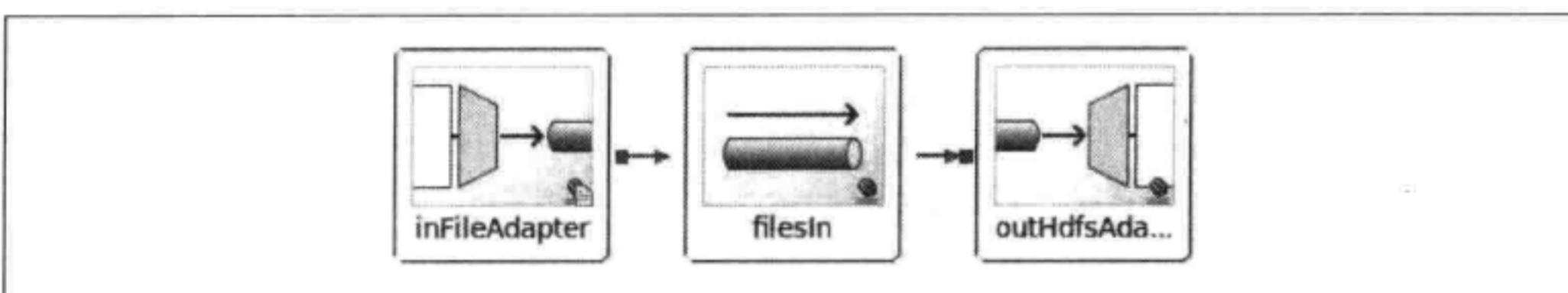


图 13-3 Spring Integration 数据管道轮询目录中的文件并将它们复制到 HDFS 中

在文件入站适配器中配置了用于轮询文件的目录以及确定哪些文件会被适配器检测到的文件名模式。这些值会被外部化到属性文件中，因此这些值可以很容易地改变以适应不同的运行时环境。由于文件系统不是事件驱动源，所以适配器使用轮询器来检测目录。可以使用多种方式配置轮询器，其中最常见的方式是使用固定延时 (fixed delay)、固定速率 (fixed rate) 或 cron 表达式。在这个示例中，没对这两个适配器之间的管道进行任何额外操作，但是如果需要的话，也可以很容易加上所需要的功能。示例 13-1 是配置这个数据管道的配置文件。

#### 示例 13-1 轮询目录中的文件并加载到 HDFS 的数据管道

```
<context:property-placeholder location="hadoop.properties,polling.properties"/>

<hdp:configuration id="hadoopConfiguration">fs.default.name=${hd.fs}</hdp:configuration>

<int:channel id="filesIn"/>

<file:inbound-channel-adapter id="inFileAdapter"
```

```

        channel="filesIn"
        directory="${polling.directory}"
        filename-pattern="${polling.fileNamePattern}">
    <int:poller id="poller" fixed-delay="${polling.fixedDelay}"/>
</file:inbound-channel-adapter>

<int:outbound-channel-adapter id="outHdfsAdapter"
    channel="filesIn"
    ref="fsShellWritingMessagingHandler" >

<bean id="fsShellWritingMessagingHandler"
    class="com.oreilly.springdata.hadoop.filepolling.FsShellWritingMessageHandler">
    <constructor-arg value="${polling.destinationHdfsDirectory}"/>
    <constructor-arg ref="hadoopConfiguration"/>
</bean>
```

这个管道的相关配置参数已外部化到 `polling.properties` 文件中，如示例 13-2 所示。

### 示例 13-2 轮询目录并将其加载到 HDFS 中的外部化属性配置

```

polling.directory=/opt/application/logs
polling.fixedDelay=5000
polling.fileNamePattern=*.txt
polling.destinationHdfsDirectory=/data/application/logs
```

这个配置会每隔 5 秒轮询一次 `/opt/application/logs` 目录，并搜索匹配 `*.txt` 模式的文件。在默认情况下，指定了 `filename-pattern` 时可避免复制文件，这个状态会被保存在内存中。之后会对这个文件适配器进行增强，使它能持久化保存应用程序状态。`FsShellWritingMessageHandler` 类负责将文件复制到 HDFS 中，它使用了 `FsShell` 的 `copyFromLocal` 方法。如果想在传输结束之后将文件从轮询目录中删除的话，需要将 `FsShellWritingMessageHandler` 类的 `deleteSourceFiles` 属性设置为 `true`。也可以锁住文件，这样当多个进程并发访问同一目录时，能够阻止它们同时读取这些文件。更多信息可以阅读 Spring Integration 参考指南。

要构建和运行这个应用，可以使用如示例 13-3 所示的命令。

### 示例 13-3 构建并运行文件轮询示例的命令

```

$ cd hadoop/file-polling
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/filepolling
```

相关的输出如示例 13-4 所示。

### 示例 13-4 文件轮询示例运行时的输出

```

03:48:44.187 [main] INFO
  c.o.s.hadoop.filepolling.FilePolling - File Polling Application Running
03:48:44.191 [task-scheduler-1] DEBUG o.s.i.file.FileReaderingMessageSource - \
Added to queue: [/opt/application/logs/file_1.txt]
03:48:44.215 [task-scheduler-1] INFO  o.s.i.file.FileReaderingMessageSource - \
Created message: [[Payload=/opt/application/logs/file_1.txt]
03:48:44.215 [task-scheduler-1] DEBUG o.s.i.e.SourcePollingChannelAdapter - \
Poll resulted in Message: [Payload=/opt/application/logs/file_1.txt]
03:48:44.215 [task-scheduler-1] DEBUG o.s.i.channel.DirectChannel - \
```

```

preSend on channel 'filesIn', message: [Payload=/opt/application/logs/file_1.txt]
03:48:44.310 [task-scheduler-1] INFO c.o.s.h.f.FsShellWritingMessageHandler - \
sourceFile = /opt/application/logs/file_1.txt
03:48:44.310 [task-scheduler-1] INFO c.o.s.h.f.FsShellWritingMessageHandler - \
resultFile = /data/application/logs/file_1.txt
03:48:44.462 [task-scheduler-1] DEBUG o.s.i.channel.DirectChannel - \
postSend (sent=true) on channel 'filesIn', \
message: [Payload=/opt/application/logs/file_1.txt]
03:48:49.465 [task-scheduler-2] DEBUG o.s.i.e.SourcePollingChannelAdapter - \
Poll resulted in Message: null
03:48:49.465 [task-scheduler-2] DEBUG o.s.i.e.SourcePollingChannelAdapter - \
Received no Message during the poll, returning 'false'
03:48:54.466 [task-scheduler-1] DEBUG o.s.i.e.SourcePollingChannelAdapter - \
Poll resulted in Message: null
03:48:54.467 [task-scheduler-1] DEBUG o.s.i.e.SourcePollingChannelAdapter - \
Received no Message during the poll, returning 'false'

```

在这个日志中，可以看到第一次轮询时检测到目录中有一个文件，接着会把该文件当作已处理过的文件，所以在第二次轮询的时候，文件入站适配器将不会再对它进行处理。FsShellWritingMessageHandler 额外的可选项还包括能够生成一个带有内嵌日期或 UUID（通用唯一标识码）的目录路径。将属性 generateDestinationDirectory 设置为 true，会使用默认的路径格式（年/月/日/小时/分钟/秒），也就是输出带有日期的路径。若将 generateDestinationDirectory 设置为 true，文件将被写入到 HDFS 中，如示例 13-5 所示。

### 示例 13-5 将 generateDestinationDirectory 设置为 true 来运行文件轮询示例的部分输出

```

03:48:44.187 [main] INFO c.o.s.hadoop.filepolling.FilePolling - \
File Polling Application Running
...
04:02:32.843 [task-scheduler-1] INFO c.o.s.h.f.FsShellWritingMessageHandler - \
sourceFile = /opt/application/logs/file_1.txt
04:02:32.843 [task-scheduler-1] INFO c.o.s.h.f.FsShellWritingMessageHandler - \
resultFile = /data/application/logs/2012/08/09/04/32/file_1.txt

```

将文件移入到 HDFS 的另外一种方法是通过 FTP 从远程机器来采集它们，如图 13-4 所示。

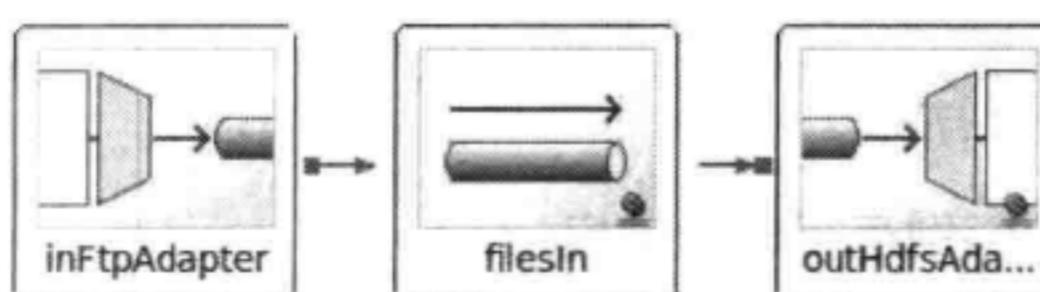


图 13-4 轮询 FTP 站点文件，并将它们复制到 HDFS 的 Spring Integration 数据管道

示例 13-6 的配置类似于文件轮询的配置，唯一不同的是入站适配器的配置发生了变化。

### 示例 13-6 定义在 FTP 站点轮询文件并将其加载到 HDFS 的数据管道

```
<context:property-placeholder location="ftp.properties,hadoop.properties"/>

<hdp:configuration>fs.default.name=${hd.fs}</hdp:configuration>

<bean id="ftpClientFactory"
      class="org.springframework.integration.ftp.session.DefaultFtpSessionFactory">
    <property name="host" value="${ftp.host}"/>
    <property name="port" value="${ftp.port}"/>
    <property name="username" value="${ftp.username}"/>
    <property name="password" value="${ftp.password}"/>
</bean>

<int:channel id="filesIn"/>

<int-ftp:inbound-channel-adapter id="inFtpAdapter"
    channel="filesIn"
    cache-sessions="false"
    session-factory="ftpClientFactory"
    filename-pattern="*.txt"
    auto-create-local-directory="true"
    delete-remote-files="false"
    remote-directory="${ftp.remoteDirectory}"
    local-directory="${ftp.localDirectory}">
    <int:poller fixed-rate="5000"/>
</int-ftp:inbound-channel-adapter>

<int:outbound-channel-adapter id="outHdfsAdapter"
    channel="filesIn" ref="fsShellWritingMessagingHandler"/>

<bean id="fsShellWritingMessagingHandler"
      class="com.oreilly.springdata.hadoop.ftp.FsShellWritingMessageHandler">
    <constructor-arg value="${ftp.destinationHdfsDirectory}"/>
    <constructor-arg ref="hadoopConfiguration"/>
</bean>
```

可以使用命令构建并运行这个应用程序，如示例 13-7 所示。

### 示例 13-7 构建并运行文件轮询示例的命令

```
$ cd hadoop/ftp
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/ftp
```

这个配置文件假定在 FTP 主机上有一个 *testuser* 账户。一旦在 FTP 的传出目录中放置了文件，就会看到数据管道开始活动，首先会将文件复制到本地目录，然后会复制到 HDFS 中。

## 13.1.3 事件流

流（Stream）是另一种常见的数据来源，你可能希望将其存储到 HDFS 中，并在它流入系统时执行实时的分析。为了满足这个需求，Spring Integration 提供了几个入站适配器来处理数据流。一旦数据进入 Spring Integration，它们会通过处理链（processing chain）传送并存储到 HDFS 中。管道也可以参与数据流的处理并将数

据写入到其他数据库，包括关系型数据库和 NoSQL，并且可以使用众多出站适配器中的一个来实现将数据流转发给其他系统。图 13-2 是这类数据管道的示例。接下来通过 TCP(Transmission Control Protocol, 传输控制协议) 和 UDP(User Datagram Protocol, 用户数据报协议) 入站适配器来使用 syslog (<http://en.wikipedia.org/wiki/Syslog>) 产生的数据，并将其写入到 HDFS。

示例 13-8 展示了如何搭建处理链，使其能够通过 TCP 将 syslog 的数据转移到 HDFS 中。

### 示例 13-8 定义数据管道，通过 TCP 接收 syslog 文件并将它加载到 HDFS 之中

```
<context:property-placeholder location="hadoop.properties, syslog.properties"/>

<hdp:configuration register-url-handler="false">
    fs.default.name=${hd.fs}
</hdp:configuration>

<hdp:file-system id="hadoopFs"/>

<int-ip:tcp-connection-factory id="syslogListener"
    type="server"
    port="${syslog.tcp.port}"
    deserializer="lfDeserializer"/>

<bean id="lfDeserializer"
    class="com.oreilly.springdata.integration.ip.syslog.ByteArrayLfSerializer"/>
<int-ip:tcp-inbound-channel-adapter id="tcpAdapter"
    channel="syslogChannel"
    connection-factory="syslogListener"/>

<!-- processing chain -->
<int:chain input-channel="syslogChannel">
    <int:transformer ref="sysLogToMapTransformer"/>
    <int:object-to-string-transformer/>
    <int:outbound-channel-adapter ref="hdfsWritingMessageHandler"/>
</int:chain>

<bean id="sysLogToMapTransformer"
    class="com.oreilly.springdata.integration.ip.syslog.SyslogToMapTransformer"/>

<bean id="hdfsWritingMessageHandler"
    class="com.oreilly.springdata.hadoop.streaming.HdfsWritingMessageHandler">
    <constructor-arg ref="hdfsWriterFactory"/>
</bean>

<bean id="hdfsWriterFactory"
    class="com.oreilly.springdata.hadoop.streaming.HdfsTextFileWriterFactory">
    <constructor-arg ref="hadoopFs"/>
    <property name="basePath" value="${syslog.hdfs.basePath}"/>
    <property name="baseFilename" value="${syslog.hdfs.baseFilename}"/>
    <property name="fileSuffix" value="${syslog.hdfs.fileSuffix}"/>
    <property name="rolloverThresholdInBytes"
        value="${syslog.hdfs.rolloverThresholdInBytes}"/>
</bean>
```

这个通道的相关配置参数外部化到 streaming.properties 文件，如示例 13-9 所示。

### 示例 13-9 从 syslog 到 HDFS 的数据流所需要的外部化属性

```
syslog.tcp.port=1514  
syslog.udp.port=1513  
syslog.hdfs.basePath=/data/  
syslog.hdfs.baseFilename=syslog  
syslog.hdfs.fileSuffix=log  
syslog.hdfs.rolloverThresholdInBytes=500
```

图 13-5 是这个数据管道的示意图。

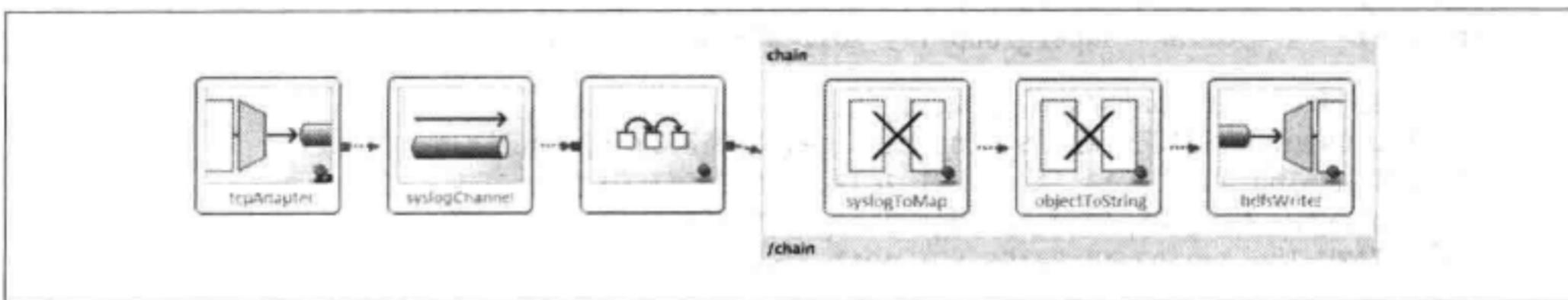


图 13-5 数据流从 syslog 流向 HDFS 的 Spring Integration 数据管道

这个配置将会创建连接工厂，它会在 1514 端口监听外部传入的 TCP 连接。为了将传入的 syslog 流分解为事件，序列化程序会根据换行符来将传入的字节流分段。需要注意的是，为了简化配置，未来底层的序列化配置将会被封装在 syslog XML 命名空间中。入站通道适配器会从 TCP 数据流中读取 syslog 消息，并把它解析成字节数组，数组会被设定为传入消息的有效负载。

Spring Integration 的链组件会自动将一系列端点 (endpoint) 组合在一起，我们不需要显式声明连接它们的管道。链中的第一个元素会解析 byte[] 数组，并将其转换成 java.util.Map 对象，这个 Map 中包含了 syslog 消息的键/值对。在这个阶段，可以对数据执行其他的操作，例如过滤、加工、实时分析或者路由到其他数据库。在这个示例中，使用内置的对象到字符串 (object-to-string) 转换器将负载数据（也就是 Map）转换成字符串。接着字符串会被传入到 HdfsWritingMessageHandler 中，并将其写入到 HDFS。HdfsWritingMessageHandler 可以让你直接配置写入文件的 HDFS 目录、文件命名规则，以及文件大小滚动 (rollover) 策略。在这个示例中，文件滚动的阈值设置较低 (500 字节，默认值为 10MB)，以便于在简单的测试示例中展现文件滚动的效果。

可以使用如示例 13-10 所示的命令来构建并运行这个应用程序。

### 示例 13-10 构建并运行 syslog 流示例的命令

```
$ cd hadoop/streaming  
$ mvn clean package appassembler:assemble  
$ sh ./target/appassembler/bin/streaming
```

若要发送测试消息，可以使用日志工具，如示例 13-11 所示。

### 示例 13-11 发送消息到 syslog

```
$ logger -p local3.info -t TESTING "Test Syslog Message"
```

由于我们已经将 HdfsWritingMessageHandler 的 rolloverThresholdInBytes 属性值设得很低，所以在发送几条消息或者等待消息从操作系统进入后，在 HDFS 中可以看到如示例 13-12 所示的文件。

### 示例 13-12 HDFS 内的 syslog 数据

```
$ hadoop dfs -ls /data
-rw-r--r-- 3 mpollack supergroup 711 2012-08-09 13:19 /data/syslog-0.log
-rw-r--r-- 3 mpollack supergroup 202 2012-08-09 13:22 /data/syslog-1.log
-rw-r--r-- 3 mpollack supergroup 240 2012-08-09 13:22 /data/syslog-2.log
-rw-r--r-- 3 mpollack supergroup 119 2012-08-09 15:04 /data/syslog-3.log

$ hadoop dfs -cat /data/syslog-2.log
{HOST=ubuntu, MESSAGE=Test Syslog Message, SEVERITY=6, FACILITY=19, \
TIMESTAMP=Thu Aug 09 13:22:44 EDT 2012, TAG=TESTING}
{HOST=ubuntu, MESSAGE=Test Syslog Message, SEVERITY=6, FACILITY=19, \
TIMESTAMP=Thu Aug 09 13:22:55 EDT 2012, TAG=TESTING}
```

使用 UDP 取代 TCP，需删除 TCP 相关定义，并添加如示例 13-13 所示的命令。

### 示例 13-13 配置以 UDP 来使用 syslog 数据

```
<int-ip:udp-inbound-channel-adapter id="udpAdapter"
    channel="syslogChannel" port="${syslog.udp.port}"/>
```

## 13.1.4 事件转发

当需要处理来自不同机器的大量数据时，将数据从生成者转发到另一个服务器来处理是非常实用的（与在本地处理相比较）操作。TCP 入站和出站适配器可以成对出现在应用程序中，以便将数据从某个服务器转发到另一个服务器。连接两个适配器的通道可以与多个持久化消息的存储组合使用。在 Spring Integration 中，消息存储以 MessageStore 接口来表示，可使用 JDBC、Redis、MongoDB 和 GemFire 的实现。因为入站和出站适配器成对出现在应用程序中会影响消息处理流程，所以消息在发送到接收的应用程序前，会先保存在发送者所在应用程序的消息存储中。一旦确认消息接收者已接收到消息，消息发送者就会将消息删除。接收者成功将接收到的消息放到自己的消息存储支撑（message-store-backed）通道中，然后发送确认消息。这个配置通过 TCP 保证了额外级别的“存储转发（store and forward）”，这种机制通常出现在像 JMS 或 RabbitMQ 这样的消息中间件之中。

示例 13-14 简单演示了 TCP 流量信息转发，并使用 Spring 所提供的支持功能启动了嵌入式 HSQL 数据库作为消息存储。

### 示例 13-14 使用 TCP 适配器跨进程存储并转发数据

```
<int:channel id="dataChannel">
    <int:queue message-store="messageStore"/>
</int:channel>

<int-jdbc:message-store id="messageStore" data-source="dataSource"/>

<jdbc:embedded-database id="dataSource"/>
```

```

<int-ip:tcp-inbound-channel-adapter id="tcpInAdapter"
    channel="dataChannel" port="${syslog.tcp.in.port}"/>

<int-ip:tcp-outbound-channel-adapter id="tcpOutAdapter"
    channel="dataChannel" port="${syslog.tcp.out.port}"/>

```

### 13.1.5 管理

Spring Integration 提供了两项重要功能，能够在运行时管理数据管道：将通道和端点导出到 JMX 以及控制总线。非常类似于 JMX，控制总线允许你调用操作指令并查看每个组件相关的信息，但是它的用途更加广泛，因为它允许在运行中的应用程序内执行一段小程序来改变其状态与行为。

将通道和端点导出到 JMX 很简单，只需要添加如示例 13-15 所示的 XML 配置即可。

#### 示例 13-15 将通道和端点导出到 JMX

```

<int-jmx:mbean-export default-domain="streaming-syslog"/>

<context:mbean-server/>

```

请运行前一节的 TCP 流示例，之后启动 JConsole 将会显示 JMX 度量(JMX metrics)信息以及可用的操作，如图 13-6 所示。这些例子是用来启动和停止 TCP 适配器，并且取得 MessageHandler 的最小、最大以及平均处理时间的。

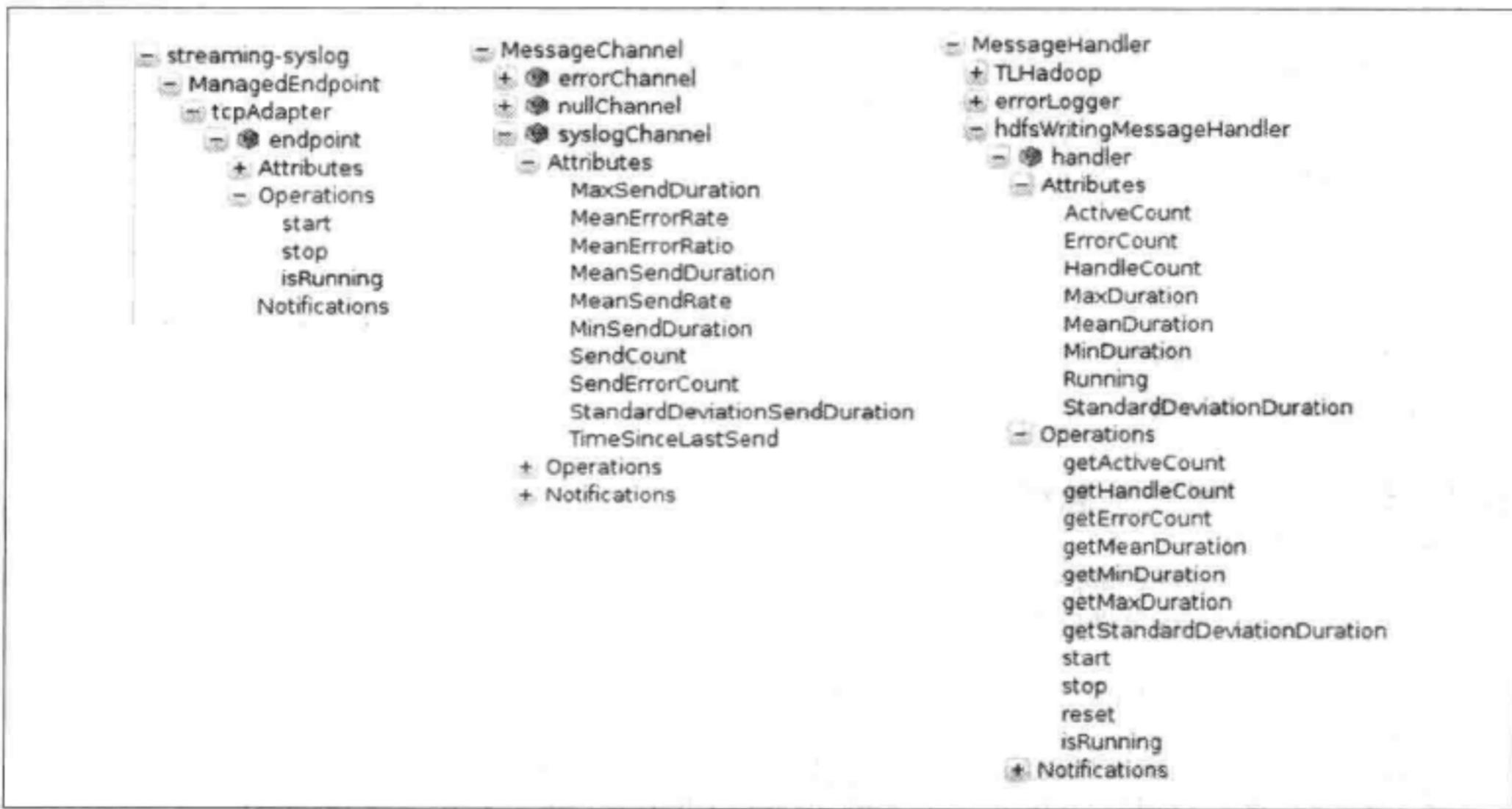


图 13-6 JConsole JMX 应用程序截屏界面，展示了在 TopAdapter、通道和 HdfsWritingMessage Handler 中可用的操作和属性

控制总线可以执行 Groovy 脚本或者 Spring 表达式语言 (SpEL)，可以在应用程序中以编程的方式来管理组件的状态。在默认情况下，Spring Integration 会通过控制总线暴露所有可存取的组件。停止 TCP 入站适配器的 SpEL 表达式语法为

`@tcpAdapter.stop()`。`@`前缀是一个可从 Spring 应用上下文中根据名称检索对象的操作符。在这个例子中，名称为 `tcpAdapter`，要调用的方法是 `stop`。执行相同动作的 Groovy 脚本不需要`@`前缀符。为了声明控制总线，请加上如示例 13-16 所示的配置。

### 示例 13-16 配置基于 Groovy 的控制总线

```
<int:channel id="inOperationChannel"/>

<int-groovy:control-bus input-channel="inOperationChannel"/>
```

通过将入站通道适配器或网关连接到控制总线的输入通道，可以远程执行脚本。也可以创建 Spring MVC 应用程序并让控制器发送消息到控制总线的输入通道，如示例 13-17 所示，如果想要提供更多诸如安全管理或者其他视图的 Web 应用功能，这是一种更为自然的方法。示例 13-17 展示了 Spring MVC 控制器将传入的 Web 请求体转发给控制总线并返回 String 类型的响应。

### 示例 13-17 在 Spring MVC 控制器中发送消息到控制总线

```
@Controller
public class ControlBusController {

    private @Autowired MessageChannel inOperationChannel;

    @RequestMapping("/admin")
    public @ResponseBody String simple(@RequestBody String message) {

        Message<String> operation = MessageBuilder.withPayload(message).build();
        MessagingTemplate template = new MessagingTemplate();
        Message response = template.sendAndReceive(inOperationChannel, operation);
        return response != null ? response.getPayload().toString() : null;
    }
}
```

重新运行示例应用程序，可以通过 HTTP 使用 curl 与控制总线交互并查询和修改入站 TCP 适配器的状态，如示例 13-18 所示。

### 示例 13-18 配置控制总线

```
$ cd streaming
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/streaming
... output omitted ...
$ curl -X GET --data "tcpAdapter.isRunning()" http://localhost:8080/admin
true
$ curl -X GET --data "tcpAdapter.stop()" http://localhost:8080/admin
$ curl -X GET --data "tcpAdapter.isRunning()" http://localhost:8080/admin
false
$ curl -X GET --data "tcpAdapter.start()" http://localhost:8080/admin
$ curl -X GET --data "tcpAdapter.isRunning()" http://localhost:8080/admin
true
```

## 13.1.6 Spring Batch 简介

Spring Batch 项目起源于 2007 年，它是由 SpringSource 和埃森哲（Accenture）合作

开发的，它有一个综合的批处理框架用来开发健壮的批处理应用程序。这些批处理应用程序需要对大量的数据进行处理，这些数据对业务操作至关重要。Spring Batch (<http://static.springsource.org/spring-batch/>) 已经在世界各地成千上万的企业应用中广泛使用。批处理 Job 有它自己的最佳实践和领域概念，这是埃森哲长年累月的咨询业务所构建起来的，并且已经封装到 Spring Batch 项目中。因此，Spring Batch 可以通过许多功能来支持大量数据的处理，例如失败后自动重试、跳过记录、从最后一次失败地方重新启动工作、定期批量提交给事务型数据库、可重用组件（如解析器、映射器、读取器、处理器、写入器和校验器）以及工作流定义。作为 Spring 生态系统一部分，Spring Batch 项目基于 Spring 框架的核心功能构建，例如 Spring 表达式语言的使用。Spring Batch 也延续了 Spring 框架的设计理念，强调基于 POJO 的开发方式并且促进创建可维护、可测试的代码。

Spring Batch 中工作流的概念在 Spring Batch 中转换成了 Job（注意不要和 MapReduce Job 混淆）。批处理 Job 是一个有向图，图中每个节点代表一个数据处理的 Step。Step 可以串行或者并行执行，这取决于 Step 的配置。Job 可以被启动、停止和重新启动。Job 已执行的 Step 进度将通过 JobRepository 存储在数据库中，所以 Job 可以重新启动。Job 也可以进行组合，所以可以在 Job 中包含 Job。图 13-7 展示了 Spring Batch 应用中的基本组件。JobLauncher 负责启动工作，通常是通过调度器（scheduler）来触发。Spring 框架提供了基本的调度功能，并支持与 Quartz 集成，但企业通常会采用它们自己的调度引擎，例如 Tivoli 或者 Control-M。其他启动 Job 的方式有：通过 RESTful 管理 API、Web 应用程序或者通过编程的方式对外部事件进行响应。最后一种方式，通常会使用 Spring Integration 项目以及它的众多通道适配器与外部系统通信。可以在《Spring Integration in Action》[Fisher12]这本书中看到更多关于 Spring Integration 结合 Spring Batch 使用的资料。如果想更多地了解 Spring Batch，可以访问项目网站 (<http://static.springsource.org/spring-batch>)，这里包含各种参考文档的链接、应用示例以及一些参考书籍的链接。

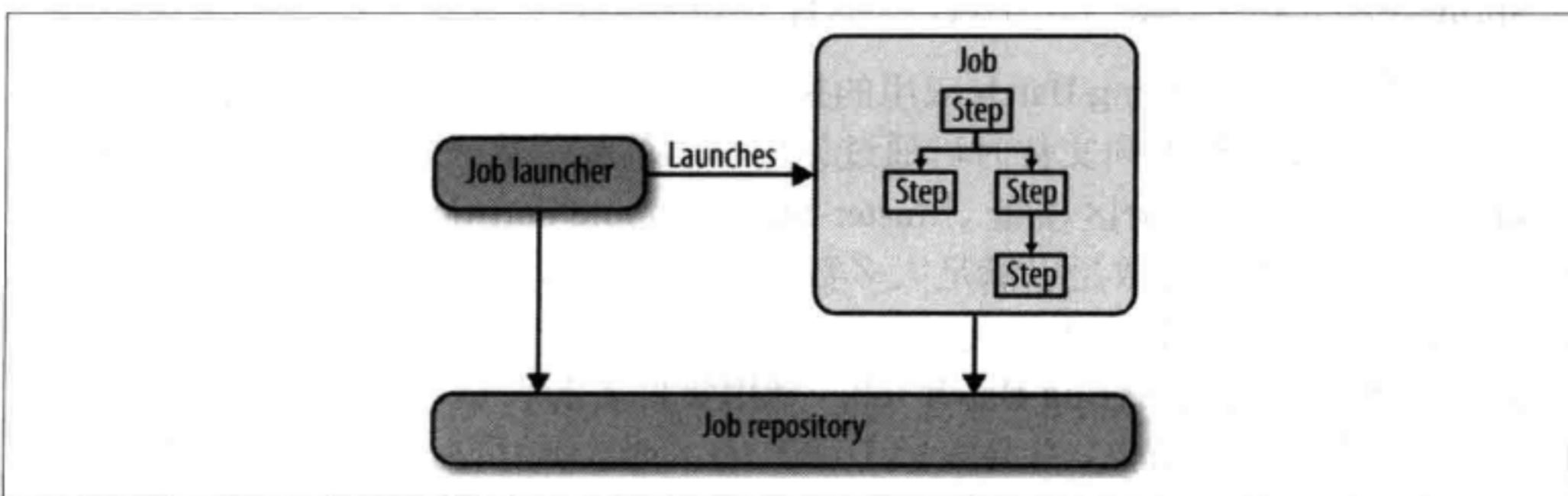


图 13-7 Spring Batch 概览

在每个 Step 中数据处理的执行可以分为 3 个阶段：ItemReader、ItemProcessor 和

ItemWriter，如图 13-8 所示，其中 ItemProcessor 是可选的。

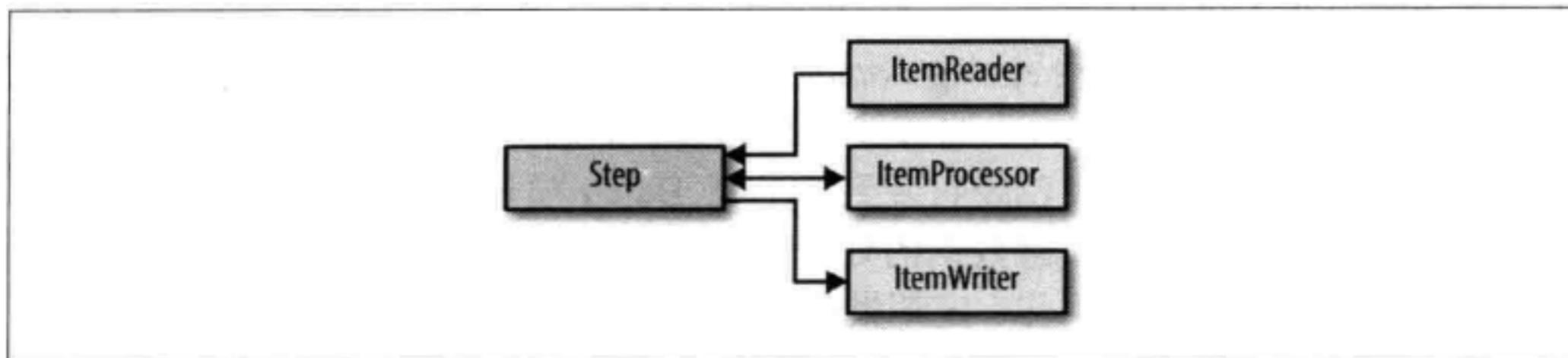


图 13-8 Spring Batch Step 组件

Spring Batch 的主要使用场景之一就是处理大型文件的内容并将数据加载到关系型数据库中。在这个示例中，FlatFileItemReader 和 JdbcItemWriter 与自定义逻辑一起使用，既可以以声明的方式来配置也可以直接在 ItemProcessor 里编写代码。为了提高性能，Step 会被“分块（chunked）”，即有一个数据块，例如 100 行数据，会被聚集在一起然后传送到 ItemWriter，这样就可以高效地使用许多数据库所提供的批处理 API 来插入数据。示例 13-19 展示了使用 Spring Batch XML 命名空间的配置片段，它可以从文件中读取数据并将它们写入到数据库中。在下文中，我们将深入探讨读取器、写入器和处理器的配置。

#### 示例 13-19 配置 Spring Batch Step 来处理 Flat 文件数据并将其复制到数据库中

```
<step id="simpleStep">
  <tasklet>
    <chunk reader="flatFileItemReader" processor="itemProcessor" writer="jdbcItemWriter"
           commit-interval="100"/>
  </tasklet>
</step>
```

Spring Batch 的其他特性可以扩展（scale up and out）执行 Job，以处理高容量和高效率的批处理 Job 需求。这些主题的相关信息，可以参考 Spring Batch 参考指南或者 Spring Batch 相关书籍 ([CoTeGreBa11], [Minella11])。

非常重要的一点是，Spring Batch 应用的执行模型独立于 Hadoop 集群。Spring Batch 应用程序在并发处理不同文件时可通过使用不同的线程数来扩展，或者使用 Spring Batch 自带的主从远程分区模型（Master-Slave Remote Partitioning Model）来实现扩展。实际上，加大线程数足以满足大多数用户的性能需求。在使用远程分区前，应该先试着将增加线程数作为扩展的首选策略。另外一个将要开发的执行模型是在 Hadoop 集群内部运行 Spring Batch Job，利用集群资源管理功能的优势将数据处理扩展到集群中的各个节点，并且兼顾 HDFS 的本地数据存储。这两种模型都各有优劣，在这里性能并不是决定选择使用哪个执行模型的唯一标准。在 Hadoop 集群外部执行批处理 Job，通常更便于数据在不同系统以及多个 Hadoop 集群之间进行移动。

在下一节使用 Spring Batch 框架来处理数据，并且从关系型数据库中将数据加载到 HDFS 中。在 13.3 小节的“从 HDFS 中导出数据”中，会将数据从 HDFS 导出到关系型数据库和 MongoDB 文档数据库。

### 13.1.7 从数据库中加载并处理数据

为了对数据进行处理，并将其从关系型数据库加载到 HDFS 中，需要使用 JdbcItemReader 和 HdfsTextItemWriter 来配置 Spring Batch Tasklet。本章节的示例应用程序位于 `./hadoop/batch-import` 目录中，它基于《*Spring Batch in Action*》(<http://code.google.com/p/springbatch-in-action/>) 一书中的示例应用程序。示例应用的领域是一个在线商店，需要维护所销售产品的目录。我们已经对原有示例进行了一些微调，以取代 Flat 文件系统来对 HDFS 进行写入操作。Spring Batch Tasklet 的配置如示例 13-20 所示。

#### 示例 13-20 从数据库中读取数据并写入到 HDFS 的 Spring Batch Step 配置

```
<job id="importProducts" xmlns="http://www.springframework.org/schema/batch">
    <step id="readWriteProducts">
        <tasklet>
            <chunk reader="jdbcReader" writer="hdfsWriter" commit-interval="100"/>
        </tasklet>
    </step>
</job>

<bean id="jdbcReader" class="org.springframework.batch.item.database.JdbcCursorItemReader">
    <property name="dataSource" ref="dataSource"/>
    <property name="sql" value="select id, name, description, price from product"/>
    <property name="rowMapper" ref="productRowMapper"/>
</bean>

<bean id="productRowMapper" class="com.oreilly.springdata.domain.ProductRowMapper"/>
```

我们使用标准的 JDBC DataSource 以及从 `product` 表中查询数据的 SQL 语句来配置 `JdbcCursorItemReader`，这些数据会被加载到 HDFS 中。执行示例 13-21 中的命令来启动数据库并初始化数据库的样例数据，将会弹出浏览器以浏览数据库内容，其中包含了 `product` 表和 Spring Batch 用来实现 Job Repository 的表。

#### 示例 13-21 初始化并运行 H2 数据库的命令，以提供给 Spring Batch 应用程序使用

```
$ cd hadoop/batch-import
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/start-database
```

提交间隔（interval）设置为 100 条，这已经超过了这个简单示例中可用的数据量，但通常推荐使用这个数量。每从数据库中读取 100 条记录，更新 Job 执行元数据的事务就会被提交到数据库。这样当执行失败时可以在停止的地方重新启动 Job。

`JdbcCursorItemReader` 的 `rowMapper` 属性是 Spring 的 `RowMapper` 接口实现，它也是 Spring JDBC 功能集合的一部分。当 `ResultSet` 中单条记录与某个 POJO 实例匹配

时，RowMapper 接口可便捷地将 JDBC ResultSet 转换成 POJO 对象。Spring 封装了 ResultSet 的迭代和异常处理（通常这很冗长且容易出错），你只需专注于映射代码的编写。在示例 13-22 中，应用程序中使用 ProductRowMapper 将 ResultSet 对象的每一条记录转换为 Product Java 对象。Product 类是一个简单的 POJO，包含 product 表中所选择的列对应的 getter 和 setter 方法。

#### 示例 13-22 将 ResultSet 中的一行记录转换成 Product 对象的 ProductRowMapper

```
public class ProductRowMapper implements RowMapper<Product> {  
  
    public Product mapRow(ResultSet rs, int rowNum) throws SQLException {  
        Product product = new Product();  
        product.setId(rs.getString("id"));  
        product.setName(rs.getString("name"));  
        product.setDescription(rs.getString("description"));  
        product.setPrice(rs.getBigDecimal("price"));  
        return product;  
    }  
}
```

JdbcCursorItemReader 类依赖于底层 JDBC 驱动的流功能，从而能够以高效的方式遍历结果集。可以设置 fetchSize 属性，指定驱动程序只加载一定数量的数据，这些数据将会被加载到运行在客户端进程的驱动之中。fetchSize 值的设置取决于 JDBC 驱动。例如，在 MySQL 中，官方文档建议将 fetchSize 的值设置为 Integer.MIN\_VALUE，但对于大数据结果集的处理这样设置对于效率提升并不是很明显。值得注意的是，Spring Batch 也提供了 JdbcPagingItemReader 类，作为另一种策略来控制由数据库加载到客户端程序的数据量，并且具有从存储过程加载数据的功能。

这个应用程序的最后一部分配置是 hdfsWriter，如示例 13-23 所示。

#### 示例 13-23 HdfsTextItemWriter 的配置

```
<context:property-placeholder location="hadoop.properties"/>  
  
<hdp:configuration>fs.default.name=${hd.fs}</hdp:configuration>  
<hdp:file-system id="hadoopFs"/>  
  
<bean id="hdfsWriter" class="com.oreilly.springdata.batch.item.HdfsTextItemWriter">  
    <constructor-arg ref="hadoopFs"/>  
    <property name="basePath" value="/import/data/products"/>  
    <property name="baseFilename" value="product"/>  
    <property name="fileSuffix" value="txt"/>  
    <property name="rolloverThresholdInBytes" value="100"/>  
    <property name="lineAggregator">  
        <bean class="org.springframework.batch.item.file.transform.PassThroughLineAggregator"/>  
    </property>  
</bean>
```

Hadoop 的配置和我们在前一节中所看到的差不多，但新增了<hdp:file-system/>，它负责根据 Hadoop 的配置创建相应的 org.apache.hadoop.fs.FileSystem。可选的实现

会通过 (*dfs://*)、HFTP (*hftp://*) 或 WebHDFS (*webhdfs://*) 与 HDFS 通信。HdfsTextItemWriter 使用 FileSystem 将纯文本文件写入到 HDFS。HdfsTextItemWriter 的属性配置 `basePath`、`baseFileName` 和 `file Suffix` 会将文件以类似 *product-0.txt* 和 *product-1.txt* 的命名方式写入到 */import/data/products* 目录。为了展示滚动的效果，我们将 `rolloverThresholdInBytes` 设置成非常低的值。

ItemWriters 通常需要一个协作对象，即 LineAggregator 接口的实现类，用来将正在处理的条目转换成字符串。在这个示例中，使用 Spring Batch 提供的 PassThroughFieldExtractor，它会委托 Product 类的 `toString()` 方法来创建字符串。Product 类的 `toString()` 方法用逗号分割来连接 ID、名称、说明以及价格。

为了运行应用程序并将数据由数据库导入到 HDFS，应执行如示例 13-24 所示的命令。

#### 示例 13-24 将数据由数据库导入到 HDFS 的命令

```
$ cd hadoop/batch-import  
$ mvn clean package appassembler:assemble  
$ sh ./target/appassembler/bin/import
```

示例 13-25 显示了在 HDFS 中的结果内容。

#### 示例 13-25 HDFS 中已导入 product 数据

```
$ hadoop dfs -ls /import/data/products
```

```
Found 6 items  
-rw-r--r-- 3 mpollack supergroup 114 2012-08-21 11:40 /import/data/products/product-0.txt  
-rw-r--r-- 3 mpollack supergroup 113 2012-08-21 11:40 /import/data/products/product-1.txt  
-rw-r--r-- 3 mpollack supergroup 122 2012-08-21 11:40 /import/data/products/product-2.txt  
-rw-r--r-- 3 mpollack supergroup 119 2012-08-21 11:40 /import/data/products/product-3.txt  
-rw-r--r-- 3 mpollack supergroup 136 2012-08-21 11:40 /import/data/products/product-4.txt  
-rw-r--r-- 3 mpollack supergroup 51 2012-08-21 11:40 /import/data/products/product-5.txt  
  
$ hadoop dfs -cat /import/data/products/  
  
PR1...210,BlackBerry 8100 Pearl,,124.6  
PR1...211,Sony Ericsson W810i,,139.45  
PR1...212,Samsung MM-A900M Ace,,97.8
```

在 Spring Batch 中还有其他大量的 LineAggregator 接口实现类来提供声明式控制，用以确定要将哪些字段写入到文件中以及使用什么字符来分割每个字段。示例 13-26 展示了其中一种实现。

#### 示例 13-26 对于 Product 对象，指定 JavaBean 的属性名以创建写入到 HDFS 的字符串

```
<property name="lineAggregator">  
  <bean class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">  
    <property name="fieldExtractor">  
      <bean class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor">  
        <property name="names" value="id,price,name"/>  
      </bean>  
    </property>
```

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

在默认情况下，DelimitedLineAggregator 会使用逗号分隔字段，JavaBean 属性名称会传入到 BeanWrapperFieldExtractor，这样 BeanWrapperFieldExtractor 将选取的 Product 对象写成一行输出到 HDFS。

使用 LineAggregator 的配置再次运行应用程序，将会在 HDFS 中创建文件，文件内容如示例 13-27 所示。

### 示例 13-27 使用另一种格式导入 HDFS 的 Product 数据

```
$ hadoop dfs -cat /import/data/products/products-0.txt
```

```
PR1...210,124.6,BlackBerry 8100 Pearl
PR1...211,139.45,Sony Ericsson W810i
PR1...212,97.8,Samsung MM-A900M Ace
```

## 13.2 Hadoop 工作流

Hadoop 应用程序极少由单个 MapReduce Job 或 Hive 脚本构成。分析逻辑通常会分解成几个步骤，再组合成一个执行链来执行完整的分析任务。在之前 MapReduce 与 Apache Web 日志的示例中，我们使用 JobRunners、HiveRunners 和 PigRunners 来执行 HDFS 操作以及 MapReduce、Hive 或 Pig 的 Job，但这并不是一个完全令人满意的解决方案。随着分析链中步骤数量的增加，将会发现执行流程将难以可视化，并且不能在 XML 命名空间中自然地形成图形结构。当我们在重用不同的 Runner 类时，也不能在分析链中跟踪执行步骤。这就意味着在分析链中如果有某个步骤失败了，就必须重新启动（手动）整个分析链，这样使得分析任务的整体执行时间（wall clock）会显著增加并且会影响到效率。本节介绍 Spring Batch 项目的扩展，它提供了将多个 Hadoop 操作链接在一起的构件，通常称为工作流（*workflows*）。

### 13.2.1 Spring Batch 对 Hadoop 的支持

由于 Hadoop 是面向批处理的系统，所以 Spring Batch 的领域概念和工作流为构造基于 Hadoop 的分析任务奠定了坚实的基础。我们利用组成 Spring Batch Step 的处理动作是可插拔的优势使得 Spring Batch 可被 Hadoop 自动发现。Step 的插入点称为 Tasklet。Spring for Apache Hadoop 提供了自定义的 Tasklet 供 HDFS，以及所有类型的 Hadoop Job：MapReduce、Streaming、Hive 和 Pig 使用。这样就可以创建如图 13-9 所示的工作流。

基于 Eclipse 的 Spring Tool Suite (STS) 支持 Spring Batch Job 可视化编程。图 13-10 展示了在 STS 中与图 13-9 对应的图。

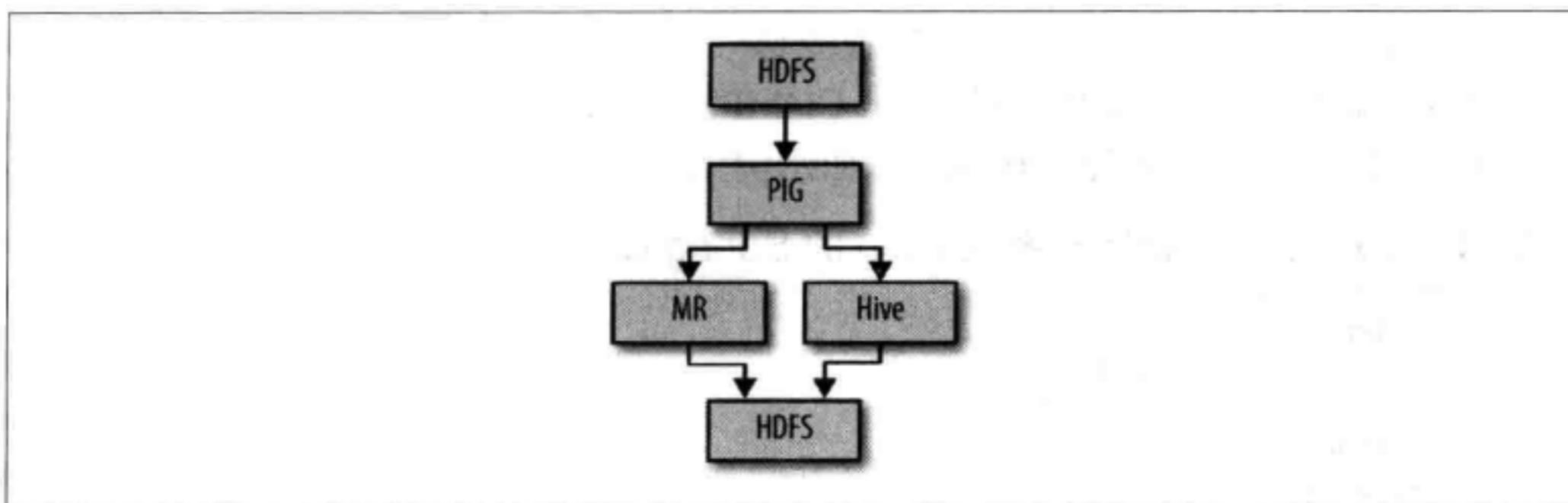


图 13-9 在 Spring Batch 应用程序中，执行 Hadoop HDFS 操作及运行 Pig、MapReduce 和 Hive Job 的 Step

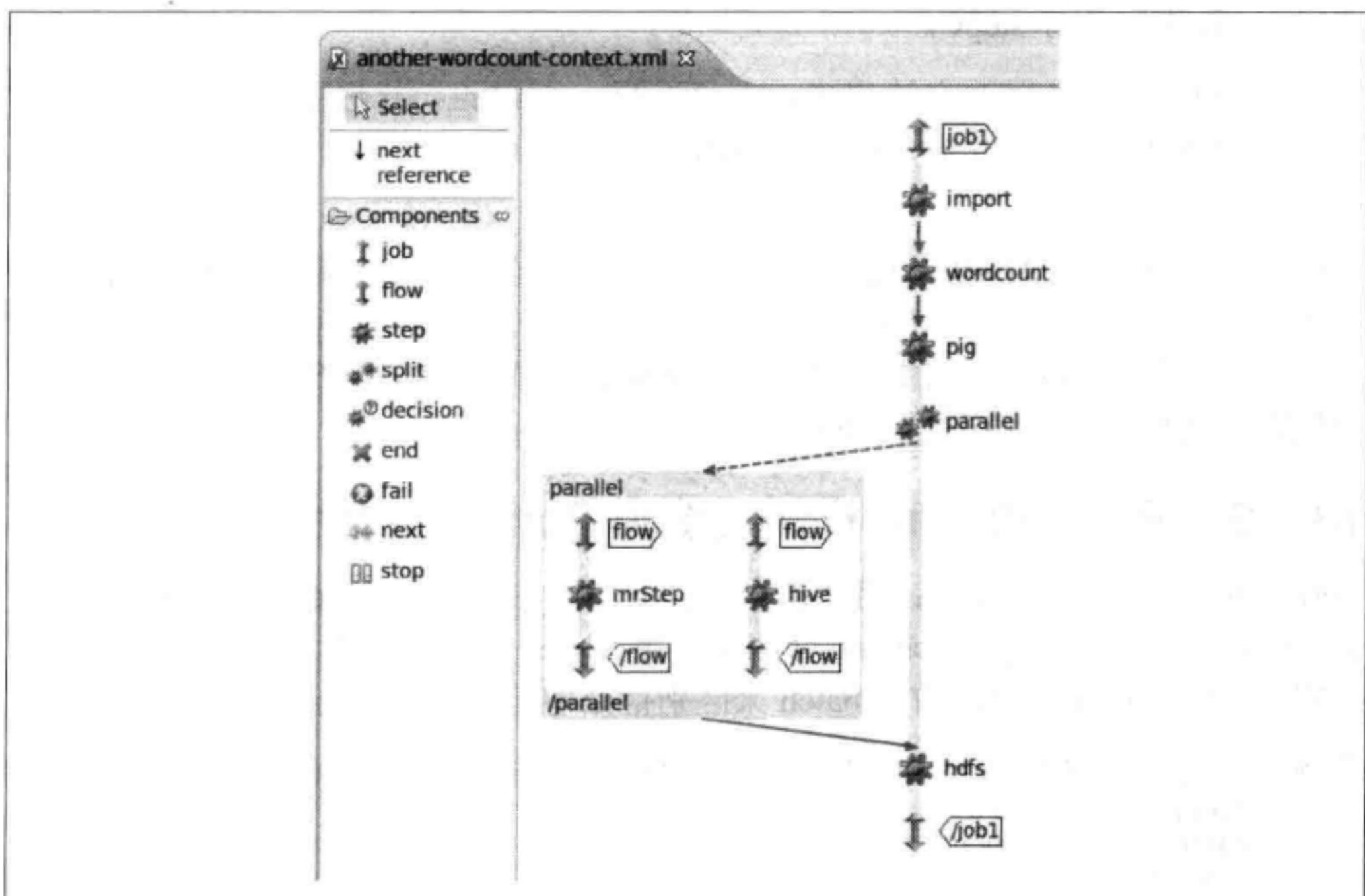


图 13-10 在 Eclipse 中创建 Spring Batch Job

当 Spring Batch Job 的 Step 为线性流程的普通模式时，其底层 XML 配置如示例 13-28 所示。

#### 示例 13-28 按顺序执行 Step 的 Spring Batch Job 定义

```

<job id="job">
    <step id="stepA" next="stepB"/>
    <step id="stepB" next="stepC"/>
    <step id="stepC"/>
</job>
  
```

也可以根据 Step 的结束状态设置流程条件。在 Step 的返回结果中有一些常见的 ExitStatus 编码，最常见的结束状态是 COMPLETED 和 FAILED。为了创建条件流程，可以使用 Step 的嵌套元素 next，如示例 13-29 所示。

#### 示例 13-29 按条件执行一系列 Step 的 Spring Batch Job 定义

```
<job id="job">
    <step id="stepA">
        <next on="FAILED" to="stepC"/>
        <next on="*" to="stepB"/>
    </step>
    <step id="stepB" next="stepC"/>
    <step id="stepC"/>
</job>
```

在这个例子中，如果结束代码匹配 FAILED，那下一步要执行的步骤是 stepC，否则在 stepB 之后执行 stepC。

在 Spring 中配置流程的方法很多，本节不再一一讨论。若想要学习更多配置高级工作流的方法，请参见 Spring Batch 参考文档或者之前提到的 Spring Batch 书籍。

可以使用 Spring for Apache Hadoop 提供的 XML 命名空间来配置与 Hadoop 相关的 Step。接下来介绍如何将 wordcount 示例配置成 Spring Batch 应用程序，我们会重用一部分之前 Hadoop 示例中使用过的 MapReduce 配置和 HDFS 脚本；然后介绍如何配置其他与 Hadoop 相关的 Step，例如 Hive 和 Pig。

### 13.2.2 将 wordcount 样例改造为 Spring Batch 应用

wordcount 示例有两个步骤：将数据导入到 HDFS 中并运行 MapReduce Job。如示例 13-30 所示的 Spring Batch Job 介绍了使用 Spring Batch XML 命名空间的工作流配置。我们使用命名空间前缀 batch 来区别批处理配置和 Hadoop 配置。

#### 示例 13-30 配置 Spring Batch Job 来执行 HDFS 和 MapReduce Step

```
<batch:job id="job1">
    <batch:step id="import" next="wordcount">
        <batch:tasklet ref="scriptTasklet"/>
    </batch:step>

    <batch:step id="wordcount">
        <batch:tasklet ref="wordcountTasklet"/>
    </batch:step>
</batch:job>

<tasklet id="wordcountTasklet" job-ref="wordcountJob"/>
<script-tasklet id="scriptTasklet" script-ref="hdfsScript">

<!-- MapReduce job and HDFS script as defined in previous examples -->
<job id="wordcountJob"
    input-path="${wordcount.input.path}"
    output-path="${wordcount.output.path}"
```

```

    mapper="org.apache.hadoop.examples.WordCount.TokenizerMapper"
    reducer="org.apache.hadoop.examples.WordCount.IntSumReducer"/>

<script id="hdfsScript" location="copy-data.groovy" >
    <property name="inputPath" value="${wordcount.input.path}"/>
    <property name="outputPath" value="${wordcount.output.path}"/>
    <property name="localResource" value="${local.data}"/>
</script>

```

通常来讲，批处理应用程序的参数化需要 Job 启动时所传入的参数。为了将批处理 Job 改为引用 Job 启动时传入的参数而不是引用静态属性文件，需要对配置进行一些修改。可以使用 SpEL (<http://static.springsource.org/spring/docs/current/spring-framework-reference/html/expressions.html>) 来获取批处理 Job 参数，这和现在所使用的\${...}语法引用变量的方式类似。

SpEL 语法和 Java 类似，它的表达式通常只是一行取值代码。我们使用语法#{...}对表达式进行求值来取代使用语法\${...}引用变量。为了访问 Spring Batch Job 参数，要使用表达式 #{jobParameters['mr.input']}。在默认情况下，当 Bean 在 Spring Batch 的 Step 作用域内时，变量 jobParameters 是可用的。MapReduce Job 和 HDFS 脚本的配置，如示例 13-31 所示。

### 示例 13-31 将 Spring Batch Tasklet 链接到 Hadoop Job 以及脚本组件中

```

<job id="wordcount-job" scope="step"
    input-path="#{jobParameters['mr.input']}"
    output-path="#{jobParameters['mr.output']}"
    mapper="org.apache.hadoop.examples.WordCount.TokenizerMapper"
    reducer="org.apache.hadoop.examples.WordCount.IntSumReducer"/>

<script id="hdfsScript" location="copy-files.groovy" scope="step">
    <property name="localSourceFile" value="#{jobParameters['localData']}"/>
    <property name="hdfsInputDir" value="#{jobParameters['mr.input']}"/>
    <property name="hdfsOutputDir" value="#{jobParameters['mr.output']}"/>
</script>

```

运行批处理程序时，应用的主程序会给这些参数传值，也可以使用其他方式为它们赋值以启动 Spring Batch 应用。最常见的选择是通过 CommandLineJobRunner、REST API 管理功能或者 Web 应用管理功能。示例 13-32 为示例应用的主驱动类。

### 示例 13-32 启动批处理工作主应用程序

```

ApplicationContext context =
    new ClassPathXmlApplicationContext("classpath:/META-INF/spring/*-context.xml");
JobLauncher jobLauncher = context.getBean(JobLauncher.class);
Job job = context.getBean(Job.class);
jobLauncher.run(job, new JobParametersBuilder()
    .addString("mr.input", "/user/gutenberg/input/word/")
    .addString("mr.output", "/user/gutenberg/output/word/")
    .addString("localData", "./data/nietzsche-chapter-1.txt")
    .addDate("date", new Date()).toJobParameters());

```

为了运行批处理应用，应执行如示例 13-33 所示的命令。

### 示例 13-33 运行 wordcount 批处理应用脚本

```
$ cd hadoop/batch-wordcount  
$ mvn clean package appassembler:assemble  
$ sh ./target/appassembler/bin/batch-wordcount
```

## 13.2.3 Hive 和 Pig 的步骤

要在 Spring Batch 工作流中执行 Hive 脚本，需使用 Hive Tasklet 元素，如示例 13-34 所示。

### 示例 13-34 配置 Hive Tasklet

```
<job id="job1" xmlns="http://www.springframework.org/schema/batch">  
    <step id="import" next="hive">  
        <tasklet ref="scriptTasklet"/>  
    </step>  
  
    <step id="hive">  
        <tasklet ref="hiveTasklet"/>  
    </step>  
</job>  
  
<hdp:hive-client-factory host="${hive.host}" port="${hive.port}"/>  
  
<hive-tasklet id="hiveTasklet">  
    <hdp:script location="analysis.hsql"/>  
</hive-tasklet>
```

要在 Spring Batch 工作流中执行 Pig 脚本，需使用 Pig Tasklet 元素，如示例 13-35 所示。

### 示例 13-35 配置 Pig Tasklet

```
<job id="job1" xmlns="http://www.springframework.org/schema/batch">  
    <step id="import" next="pig">  
        <tasklet ref="scriptTasklet"/>  
    </step>  
  
    <step id="pig">  
        <tasklet ref="pigTasklet"/>  
    </step>  
</job>  
  
<pig-factory/>  
  
<pig-tasklet id="pigTasklet">  
    <script location="analysis.pig">  
        <arguments>  
            piggybanklib=${pig.piggybanklib}  
            inputPath=${pig.inputPath}  
            outputPath=${pig.outputPath}  
        </arguments>  
    </script>  
</pig-tasklet>
```

### 13.3 从 HDFS 导出数据

Hadoop 中的分析结果通常会被复制到结构化数据存储中，例如关系型数据库或者 NoSQL 数据库，以便于展现或进一步分析。Spring Batch 的主要使用场景之一就是在文件和数据库之间移动数据并进行处理。本节将使用 Spring Batch 从 HDFS 中导出数据，并对数据执行一些基本操作，接下来将数据存储在 HDFS 之外的地方。数据存储的目标是关系型数据库或者 MongoDB。

#### 13.3.1 从 HDFS 到 JDBC

将 MapReduce Job 所产生的结果数据从 HDFS 移到关系型数据库是很常见的操作。Spring Batch 提供了很多内置组件，这样就可以通过配置来执行这个动作。本节的示例位于 `./hadoop/batch-extract` 目录下，示例代码是基于《*Spring Batch in Action*》一书的示例而创建的。示例应用程序的领域模型是一个在线商店，它需要维护所销售产品的目录。这个应用程序最初是从本地文件系统的 Flat 文件中读取产品数据，然后再将其写入到关系型数据库的 `product` 表中。我们已经修改了示例代码，让它从 HDFS 中读取，为展示 Spring Batch 额外的特性(<http://code.google.com/p/springbatch-in-action/>) 在代码中增加了错误处理逻辑。

为了将从本地文件系统读取数据替换成从 HDFS 中读取，需要在 Spring 中注册 `HdfsResource` 加载器，它会使用 Spring 的 `Resource` 抽象读取 HDFS 中的数据。因为 Spring Batch 的 `FlatFileItemReader` 类是基于 `Resource` 的抽象，所以在这里可以使用它。Spring 的 `Resource` 抽象提供了统一方式来从不同的来源中读取 `InputStream`，例如 URL (http 和 ftp)、Java 类路径或者标准文件系统。`Resource` 抽象 (<http://static.springsource.org/spring/docs/current/spring-framework-reference/html/resources.html>) 也支持通过使用 Ant 类型的正则表达式来读取多个资源的位置。为了配置 Spring 读取 HDFS 并将 HDFS 设置为默认的资源类型(如 `hdfs://hostname:port` 而不是 `file://`)，请将如示例 13-36 所示的 XML 添加到 Spring 配置文件中。

##### 示例 13-36 配置默认资源加载器使用 HDFS

```
<context:property-placeholder location="hadoop.properties"/>

<hdp:configuration>fs.default.name=${hd.fs}</hdp:configuration>
<hdp:resource-loader id="hadoopResourceLoader"/>

<bean id="defaultResourceLoader"
    class="org.springframework.data.hadoop.fs.CustomResourceLoaderRegistrar">
    <property name="loader" ref="hadoopResourceLoader"/>
</bean>
```

Spring Batch 基本概念，例如 `Job`、`Step`、`ItemReader`、处理器以及写入器，已在 13.1.6 小节“Spring Batch 简介”中介绍过了。在本节中，我们将配置这些组件并介绍它

们的一些配置属性。然而，我们不会太全面地探讨如何配置以及运行 Spring Batch 应用。在 Spring Batch 中包括非常丰富的内容，如异常处理、通知、数据校验、数据处理以及纵向和横向扩展，本书无法全部涵盖。如果需要了解更多的信息，可以参阅 Spring 参考手册或者之前提到过的任意一本与 Spring Batch 相关的图书。

示例 13-37 是创建 Spring Batch Job 所需的高级配置，在它所使用的 Step 中，会处理位于 HDFS 中的 MapReduce Job 输出文件并将其写入到数据库。

#### 示例 13-37 Spring Batch Job 的配置，由 HDFS 读取并写入到关系型数据库中

```
<job id="exportProducts">
    <step id="readWriteProducts">
        <tasklet>
            <chunk reader="hdfsReader" processor="processor" writer="jdbcWriter"
                   commit-interval="100" skip-limit="5">
                <skippable-exception-classes>
                    <include class="org.springframework.batch.item.file.FlatFileParseException"/>
                </skippable-exception-classes>
            </chunk>
            <listeners>
                <listener ref="jdbcSkipListener"/>
            </listeners>
        </tasklet>
    </step>
</job>
```

这个 Job 只定义了一个 Step，它包含了读取器、处理器和写入器。提交间隔要参考数据项的数量来决定，在提交数据库之前对数据项进行处理以及聚合。在实际使用中，会通过调整提交间隔的值以确定哪些值会产生最高的性能，这个属性的值应该是介于 10 到几百之间。这里也介绍了 Spring Batch 灵活的错误处理机制：使用 skip-limit 和 skippable-exception-classes 属性来设定在处理过程中，该 Step 失效前所允许发生特定错误的次数。skipLimit 属性决定了在 Job 失效前抛出异常次数。在这个例子中，允许抛出异常 FlatFileParseException 共 5 次。为了跟踪所有未正确处理的行，需要配置一个监听器，将错误数据写入到独立的数据库表中。这个监听器继承了 Spring Batch 的 SkipListenerSupport 类，我们重写了方法 onSkipInRead(Throwable t) 来提供失败行的信息。

因为要读取许多 MapReduce Job 生成的文件（例如，part-r-00001 和 part-r-00002），所以我们使用了 Spring Batch 的 MultiResourceItemReader，并传入 HDFS 目录名称作为 Job 参数，以及真正从 HDFS 中读取单个文件的 FlatFileItemReader 引用，如示例 13-28 所示。

#### 示例 13-38 Spring Batch HDFS 读取器的配置

```
<bean id="hdfsReader"
      class="org.springframework.batch.item.file.MultiResourceItemReader" scope="step">
    <property name="resources" value="#{jobParameters['hdfsSourceDirectory']}"/>
    <property name="delegate" ref="flatFileItemReader"/>
</bean>
```

```

<bean id="flatFileItemReader"
      class="org.springframework.batch.item.file.FlatFileItemReader">
    <property name="lineMapper">
      <bean class="org.springframework.batch.item.mapping.DefaultLineMapper">
        <property name="lineTokenizer">
          <bean
            class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer">
            <property name="names" value="id, name, description, price"/>
          </bean>
        </property>
        <property name="fieldSetMapper">
          <bean class="com.oreilly.springdata.batch.item.file.ProductFieldSetMapper"/>
        </property>
      </bean>
    </property>
  </bean>

```

当启动 Job 时，通过编程方式或者在使用 Spring Batch 管理功能时通过 REST API 或 web 应用方式来提供 job 参数 hdfsSourceDirectory。设置 bean 的作用域为 Step 可以解析 jobParameter 变量。示例 13-39 使用主 Java 类来加载 Spring Batch 配置并启动 Job。

### 示例 13-39 启动带参数的 Spring Batch job

```

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

    ApplicationContext ctx =
        new ClassPathXmlApplicationContext("classpath*/META-INF/spring/*.xml")
    JobLauncher jobLauncher = ctx.getBean(JobLauncher.class);
    Job job = ctx.getBean(Job.class);

    jobLauncher.run(job, new JobParametersBuilder()
        .addString("hdfsSourceDirectory", "/data/analysis/results/part-*")
        .addDate("date", new Date())
        .toJobParameters());
}

```

通过 MultiResourceItemReader 类，可以处理位于 HDFS 目录 */data/analysis/results* 中匹配表达式 part-\* 的多个文件。每一个 MultiResourceItemReader 发现的文件都会被 FlatFileItemReader 处理。文件中的内容样例都如示例 13-40 所示。在这个文件中有 4 个列，分别表示产品 ID、名称、说明和价格（在样例数据中描述为空）。

### 示例 13-40 导入数据库的 HDFS 文件的样本内容

```

PR1...210,BlackBerry 8100 Pearl,,124.60type
PR1...211,Sony Ericsson W810i,,139.45
PR1...212,Samsung MM-A900M Ace,,97.80
PR1...213,Toshiba M285-E 14,,166.20
PR1...214,Nokia 2610 Phone,,145.50
...
PR2...315,Sony BDP-S590 3D Blu-ray Disk Player,,86.99
PR2...316,GoPro HD HERO2,,241.14
PR2...317,Toshiba 32C120U 32-Inch LCD HDTV,,239.99

```

这里配置了两个 DefaultLineMapper 类型的协同对象来指定 FlatFileItemReader 的功能。第一个是 Spring Batch 提供的 DelimitedLineTokenizer 类，它会读取一行输入，

默认情况下会将逗号作为分隔符。字段名称和值都会被放置在 Spring Batch 的 FieldSet 对象中。FieldSet 类似 JDBC 的结果集，但它是从文件中读取数据。FieldSet 允许以名称或者位置来访问列，并且会将这些列的值转换为 Java 类型，如 String、Integer 或者 BigDecimal。第二个协作对象是 ProductFieldSetMapper，这个类是由我们提供的，它会将 FieldSet 转换成自定义的领域对象，如示例中的 Product 类。

当从数据库读取数据并写入到 HDFS 时，ProductFieldSetMapper 非常类似于前面章节中所使用的 ProductRowMapper，如示例 13-41 所示。

#### 示例 13-41 将 FieldSet 转换成 Product 领域对象

```
public class ProductFieldSetMapper implements FieldSetMapper<Product> {  
  
    public Product mapFieldSet(FieldSet fieldSet) {  
        Product product = new Product();  
        product.setId(fieldSet.readString("id"));  
        product.setName(fieldSet.readString("name"));  
        product.setDescription(fieldSet.readString("description"));  
        product.setPrice(fieldSet.readBigDecimal("price"));  
        return product;  
    }  
}
```

最后要配置的两个部分是 ItemProcessor 与 ItemWriter，如示例 13-42 所示。

#### 示例 13-42 Spring Batch 数据项处理与 JDBC 写入器的配置

```
<bean id="processor" class="com.oreilly.springdata.batch.item.ProductProcessor"/>  
  
<bean id="jdbcWriter" class="org.springframework.batch.item.database.JdbcBatchItemWriter">  
    <property name="dataSource" ref="dataSource"/>  
    <property name="sql"  
        value="INSERT INTO PRODUCT (ID, NAME, PRICE) VALUES (:id, :name, :price)"/>  
    <property name="itemSqlParameterSourceProvider">  
        <bean class="org.sfw.batch.item.database.BeanPropertyItemSqlParameterSourceProvider"/>  
    </property>  
</bean>
```

ItemProcessors 通常用于转换、过滤或者校验数据。在示例 13-43 中，使用了简单的过滤器，它会过滤掉产品描述 ID 由 PR1 开始的记录。根据协议，过滤掉符合规则的数据项时，ItemProcessor 会返回 null 值。需要注意的是，如果不想要进行任何处理而是希望直接将输入文件复制到数据库中可以从 tasklet 的 XML 配置块中将处理器属性删除。

#### 示例 13-43 简单的过滤器 ItemProcessor

```
public class ProductProcessor implements ItemProcessor<Product, Product> {  
  
    @Override  
    public Product process(Product product) throws Exception {  
        if (product.getId().startsWith("PR1")) {  
            return null;  
        } else {  
            return product;  
        }  
    }  
}
```

```
    }
}
}
```

JdbcBatchItemWriter 会将一批 SQL 语句组合一起提交到数据库。批次大小为之前所定义的提交间隔。我们使用标准的 JDBC DataSource 连接数据库，并且 SQL 语句为指定的内联方式。使用 SQL 语句的优点是可以使用命名参数来取代位置占位符。这是由 beanPropertyItemSqlParameterSourceProvider 提供的功能，这样可以使 Product 对象中属性的名称与 SQL 语句中 :name 的值关联。示例 13-44 为使用 H2 数据库的 product 表模式。

#### 示例 13-44 product 的模式定义

```
create table product (
    id character(9) not null,
    name character varying(50),
    description character varying(255),
    price float,
    update_timestamp timestamp,
    constraint product_pkey primary key (id)
);
```

启动数据库、将样例数据复制到 HDFS 中，然后创建 Spring Batch 模式并执行如示例 13-45 所示的命令。运行这些命令后，会启动与 H2 交互的 Web 控制台。

#### 示例 13-45 构建示例并启动数据库

```
$ cd hadoop/batch-extract
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/start-database &
```

接下来运行导出程序，如示例 13-46 所示。

#### 示例 13-46 运行导出 Job

```
$ sh ./target/appassembler/bin/export
INFO - Loaded JDBC driver: org.h2.Driver
INFO - Established shared JDBC Connection: conn0: \
url=jdbc:h2:tcp://localhost/mem:hadoop_export user=SA
INFO - No database type set, using meta data indicating: H2
INFO - No TaskExecutor has been set, defaulting to synchronous executor.
INFO - Job: [FlowJob: [name=exportProducts]] launched with the following parameters: \
[{:hdfsSourceDirectory=/data/analysis/results/part-*, date=1345578343793}]
INFO - Executing step: [readWriteProducts]
INFO - Job: [FlowJob: [name=exportProducts]] completed with the following parameters: \
[{:hdfsSourceDirectory=/data/analysis/results/part-*, date=1345578343793}] and the \
following status: [COMPLETED]
```

可以使用 H2 的 Web 控制台查看导入的数据。Spring Batch 也提供了管理控制台，可以用它来浏览哪些 Job 可以执行，也可以看到每个执行 Job 的状态。要启动管理控制台，请执行 `sh./target/appassembler/bin/launchSpringBatchAdmin`，打开浏览器并访问 `http://localhost:8080/springbatchadmin/jobs/executions`，并从表中选择最近执行 Job 的链接。点击特定 Job 执行的链接后，便可以查看 Job 状态的详细内容。

### 13.3.2 从 HDFS 到 MongoDB

如果要取代关系型数据库，将数据写入到 MongoDB，需要将 ItemWriter 的实现由 JdbcBatchItemWriter 改为 MongoItemWriter。示例 13-47 展示了简单的 MongoItemWriter 实现，它使用 MongoDB 的批处理功能来写入数据项列表，只需调用一次数据库操作即可将数据项插入到集合中。

#### 示例 13-47 写入 MongoDB 的 ItemWriter 实现

```
public class MongoItemWriter implements ItemWriter<Object> {

    private MongoOperations mongoOperations;
    private String collectionName = "/data";

    // constructor and setters omitted.

    @Override
    public void write(List<? extends Object> items) throws Exception {
        mongoOperations.insert(items, collectionName);
    }
}
```

Spring 的 MongoTemplate（实现了 MongoOperations 接口）提供了将 Java 类转换为 MongoDB 内部数据结构格式 DbObject 的功能。可以使用 Mongo XML 命名空间来指定到 MongoDB 的关联。示例 13-48 展示了 MongoItemWriter 的配置以及到 MongoDB 连接的底层依赖。

#### 示例 13-48 从 HDFS 中读取数据并写入到 MongoDB 的 Spring Batch Job 配置

```
<job id="exportProducts">
    <step id="readWriteProducts">
        <tasklet>
            <chunk reader="reader" writer="mongoWriter" commit-interval="100" skip-limit="5">
                <skippable-exception-classes>
                    <include class="org.springframework.batch.item.file.FlatFileParseException"/>
                </skippable-exception-classes>
            </chunk>
        </tasklet>
    </step>
</job>

<!-- reader configuration is the same as before --&gt;

&lt;bean id="mongoWriter" class="com.oreilly.springdata.batch.item.mongodb.MongoItemWriter"&gt;
    &lt;constructor-arg ref="mongoTemplate"/&gt;
    &lt;property name="collectionName" value="products"/&gt;
&lt;/bean&gt;

&lt;bean id="mongoTemplate" class="org.springframework.data.mongodb.core.MongoTemplate"&gt;
    &lt;constructor-arg ref="mongo"/&gt;
    &lt;constructor-arg name="databaseName" value="test"/&gt;
&lt;/bean&gt;

&lt;mongo:mongo host="localhost" port="27017"/&gt;</pre>
```

再次运行应用程序将会在测试数据库中产生 product 集合的内容，如示例 13-49 所示。

#### 示例 13-49 在 MongoDB Shell 中查看导出的 product 集合

```
$ mongo
> use test
> db.products.find()
{"_id" : "PR 210", "PRICE" : "124.60", "NAME" : "BlackBerry 8100 Pearl",
 "DESCRIPTION" : ""}
{"_id" : "PR 211", "PRICE" : "139.45", "NAME" : "Sony Ericsson W810i", "DESCRIPTION" : ""}
 {"_id" : "PR 212", "PRICE" : "97.80", "NAME" : "Samsung MM-A900M Ace", "DESCRIPTION" : ""}

(output truncated)
```

这个示例展示了如何基于不同的 Spring Data 项目以最少的代码构建重要的新功能。只要遵循 MongoItemWriter 的实现模式，就可以轻松为 Redis 或 GemFire 创建 ItemWriters，它的代码就如同本节所示的代码一样简单。

## 13.4 收集并加载数据到 Splunk

Splunk 可收集、索引、搜索并监控机器所产生的大数据，也可以处理实时数据流和历史数据。Splunk 的第一个版本在 2005 年发布，它的目的是分析数据中心内部所产生的数据，以帮助解决基础设施运维所带来的问题。因此，它的核心功能之一是将个机器产生的数据移到中央处理库，并且内置就能识别常用日志文件格式以及类似 syslog 的基础软件。Splunk 的基础架构包含一个用于处理并且索引数据流的 splunk 守护进程，以及可以允许用户搜索和创建报告的 Web 应用程序。Splunk 可在数据需求量增加时，通过增加独立的索引、搜索、转发实例来进行横向扩展。更多关于 Splunk 安装、运行以及开发的细节，可以参考产品网站 (<http://www.splunk.com/>) 与《Exploring Splunk》(<http://www.splunk.com/goto/book>) 一书。

虽然 Splunk 内置了收集日志文件和 syslog 数据的进程，但我们仍然需要从其他各种不同来源收集、转换以及加载数据到 Splunk，从而减少分析数据时使用正则表达式的需求。Splunk 也需要转换和提取数据到其他数据库和文件系统中。为了满足这些需求，创建了对应的 Spring Integration 入站和出站通道适配器，而且 Spring Batch 的支持也在规划中。在撰写本文时，支持 Splunk 的 Spring Integration 通道适配器位于 GitHub 资源库 (<https://github.com/SpringSource/spring-integration-extensions>) 中，它可以供 Spring Integration 扩展使用。适配器支持所有存取 Splunk 数据的方法。入站适配器支持阻塞和无阻塞的搜索、保存、实时搜索和导出模式。出站适配器支持将数据通过 RESTful API、流或 TCP 放入到 Splunk 中。Splunk 的所有功能都通过完整的 REST API 暴露出来，同时还有多种语言的 SDK，它们使得借助 REST API 进行开发尽可能简单。Spring Integration 适配器使用的 Splunk Java SDK，也可以使用 Github 获取。

为了介绍以 Spring Integration 的方式来使用 Splunk，我们将创建一个应用程序，将来自 Twitter 的搜索结果存储到 Splunk 中。这个应用程序的配置如示例 13-50 所示。

#### 示例 13-50 将 Twitter 搜索结果存储到 Splunk 应用程序的配置

```
<context:property-placeholder location="twitter.properties,splunk.properties" />

<bean id="twitterTemplate"
    class="org.springframework.social.twitter.api.impl.TwitterTemplate">
    <constructor-arg value="${twitter.oauth.consumerKey}" />
    <constructor-arg value="${twitter.oauth.consumerSecret}" />
    <constructor-arg value="${twitter.oauth.accessToken}" />
    <constructor-arg value="${twitter.oauth.accessTokenSecret}" />
</bean>

<int-splunk:server id="splunkServer"
    host="${splunk.host}" port="${splunk.port}"
    userName="${splunk.userName}" password="${splunk.password}"
    owner="${splunk.owner}"/>

<int:channel id="input"/>

<int:channel id="output"/>

<int-twitter:search-inbound-channel-adapter id="searchAdapter" channel="input"
    query="#voteobama OR #voteromney OR #votieberber">
    <int:poller fixed-rate="5000" max-messages-per-poll="50" />
</int-twitter:search-inbound-channel-adapter>

<int:chain input-channel="input" output-channel="output">
    <int:filter ref="tweetFilter"/>
    <int:transformer ref="splunkTransformer"/>
</int:chain>

<int-splunk:outbound-channel-adapter id="splunkOutboundChannelAdapter"
    channel="output" auto-startup="true"
    splunk-server-ref="splunkServer" pool-server-connection="true"
    sourceType="twitter-integration" source="twitter" ingest="SUBMIT"/>

<!-- tweetFilter and splunkTransformer bean definitions omitted -->
```

Spring Social 项目 (<http://www.springsource.org/spring-social>) 提供了以 OAuth 方式连接到 Twitter 的基础功能，也提供了与 Facebook、LinkedIn、TripIt、Foursquare 等其他数十个社交类软件即服务 (Software-as-a-Service) 提供商交互的支持。入站通道适配器使用 Spring Social 的 TwitterTemplate 类与 Twitter 进行交互。

为了访问 Twitter 提供的完整功能，需要在 Twitter 开发者网站 (Twitter Developer Website) 上创建一个新的应用程序。为了连接到 Splunk 服务器，需要使用 XML 命名空间的<int-splunk:server/>标签，以提供主机/端口以及用户验证信息（使用 Spring 占位符来外部化参数）。

Twitter 的入站通道适配器支持以时间轴更新 (Timeline Updates)、直接消息 (Direct Messages)、提及消息 (Mention Messages) 及搜索结果 (Search Results) 来接收 twitter

数据。请注意不久之后它将支持通过 Twitter garden hose 获取消费数据，它支持随机选择数据流，此数据流大小被限制为完整数据流的一个较小的百分比。在示例 13-50 中，查询将查找出带有美国 2012 总统大选标签（hashbag）的数据，并且还有一个标签用来展示贾斯汀·比伯（Justin Bieber）赢得各种奖项的投票情况。处理链会使用过滤器将 Twitter 负载对象转换成一个经过优化的数据格式，以帮助 Splunk 索引和搜索 Twitter。在示例应用中，过滤器被配置成一个简单的数据传递通道（pass-through）。出站通道适配器使用属性 inject = "SUBMIT" 来指定使用 REST API 写入 Splunk 资源。数据会被写入到名为 *twitter* 的 Splunk 资源并且使用默认索引。也可以设定索引属性来指定数据写入到非默认索引。

遵循目录 *splunk/tweets* 中的说明，执行示例并查看谁是最受欢迎的候选人（或虚拟候选人）。接着可以通过 Splunk Web 应用建立搜索，例如 source = "twitter" | regex tags = "^voteobama\$|^voteieber\$|^voteromney\$" | top tags，来获得显示这些标签相对热门程度的图表。



## 第六部分

---

# 数据网格



# 分布式数据网格：GemFire

vFabric™GemFire®（GemFire）是一个需要商业许可的数据管理平台，它提供了基于分布式体系结构的数据访问功能。它既可以作为独立的产品来使用，也可以作为 VMware vFabric 套件的一个组件。本章将对 Spring Data GemFire 进行概述，首先介绍 GemFire 以及使用 GemFire 进行开发所需的一些基本理念。如果你已经熟悉 GemFire，可以跳到 14.4 小节“使用 Spring XML 命名空间配置 GemFire”。

## 14.1 GemFire 简介

GemFire 提供了一个内存数据网格，它具备极高的吞吐量、低延迟的数据访问以及可扩展性。除了分布式缓存之外，GemFire 还提供了如下高级功能：

- 事件通知；
- OQL（对象查询语言，Object Query Language）查询语法；
- 持续查询；
- 事务支持；
- 远程方法调用；
- WAN 通信；
- 高效且便捷的对象序列化（PDX）；
- 为系统管理员提供管理和配置 GemFire 分布式系统的工具。

通过配置，GemFire 可以支持众多的分布式系统拓扑结构，并可以与 Spring 框架完美地集成。图 14-1 展示生产环境中局域网（LAN）内典型的客户机/服务器配置。GemFire 定位器（locator）作为分布式系统的中介者（broker）用来协助发现新的

成员节点，客户端应用程序使用定位器来获取缓存服务器的连接。此外，服务器节点使用定位器来进行相互查找。一旦有服务器联机在线，它会直接与对等服务器（peers）进行通信。同样的，一旦客户端完成初始化，它会直接与缓存服务器进行通信。由于定位器存在单点故障，因此需要有两个实例作为冗余。

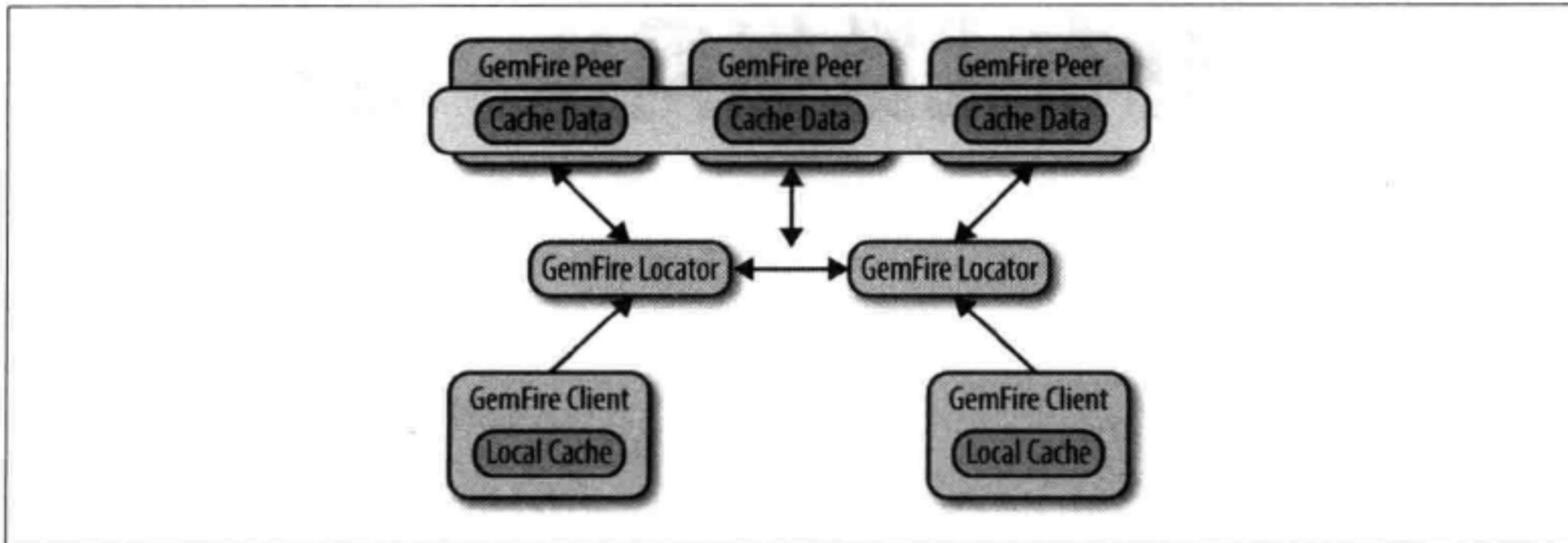


图 14-1 GemFire 客户机/服务器拓扑结构

GemFire 也可以进行简单的单机配置。需要说明的是本书代码示例的配置都比较简单，采用单进程模式且带有嵌入式缓存，以便于开发和集成测试。

在客户机/服务器场景中，应用程序使用连接池（如图 14-2 所示）来管理客户端缓存和服务器之间的连接。连接池管理网络连接、分配线程并提供了许多可调整的配置选项来平衡资源利用和性能。连接池的典型配置需要定位器的地址（没有在图 14-2 中标出）。一旦定位器提供了服务器连接，客户端即可直接与服务器进行通信。如果主服务器无法连接，并且还有其他可用的服务器，连接池将和备用服务器建立连接。

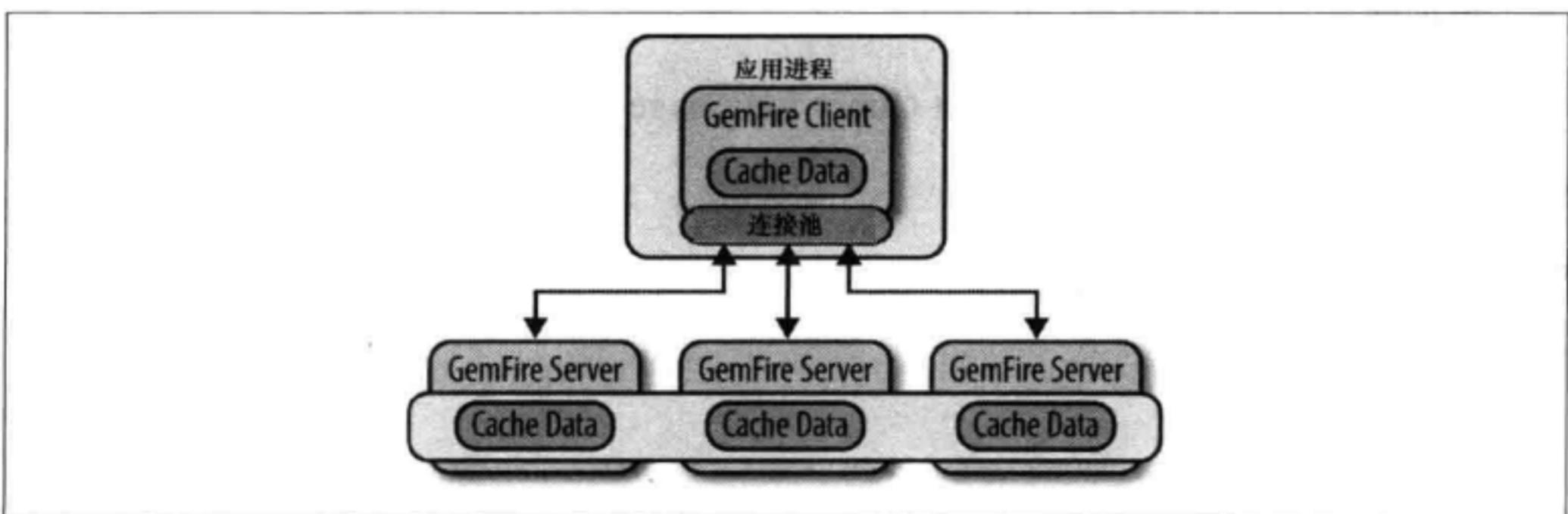


图 14-2 GemFire 连接池

## 14.2 缓存与域

从概念上讲，缓存就是单例的对象，可访问 GemFire 成员，并提供了许多内存调整、

网络连接以及其他功能的配置选项。缓存也可以作为域的容器，域提供了数据管理和存取的功能。

域必须从缓存中存储和检索数据。域接口扩展了 `java.util.Map`，从而可以使用熟悉的键/值语义来执行基本数据的存取。域接口装配到需要它的类之中，这样实际的域类型会与编程模型解耦合（这里有一些注意事项，留给读者作为思考练习）。通常每个域都会关联一个领域对象，类似于关系数据库中的表。观察示例代码，会看到 3 个域定义：`Customer`、`Product` 和 `Order`。需要注意的是 GemFire 不负责管理每个域之间关联关系的完整性。

GemFire 包含以下几个域类型。

#### 复制域 (*Replicated*)

数据会在域所定义的缓存成员之间复制，因此提供了高效的读取性能，但是因为需要进行数据复制，所以写入耗时较长。

#### 分区域 (*Partitioned*)

域会将数据分成桶（bucket）并分布存储在定义域的缓存成员之间，以提供高效的读写性能，适用于非常大型且无法在单个节点存储的数据集。

#### 本地域 (*Local*)

数据仅存放在本地节点上。

#### 客户域 (*Client*)

从技术上说客户域就是本地域，它作为复制域或分区域在缓存服务器上的代理。它可以持有本地创建或读取的数据，也可以是空的。在本地所做的更新会与缓存服务器同步。此外，客户域可以订阅事件，使得当存取相同域的远程进程发生变化时可保持同步。

希望以上简要的介绍能让你体验到 GemFire 灵活性和成熟度。对 GemFire 可选项与功能的详细讨论已经超出了本书范围，有兴趣的读者可以在产品网站 (<http://www.vmware.com/products/application-platform/vfabric-gemfire>) 找到更多详细资料。

## 14.3 如何获取 GemFire

vFabric GemFire 的网站提供了详细的产品信息、参考指南以及面向开发人员的免费下载链接，开发者版本只能使用三个节点连接。为了更全面地评估产品，可以使用为期 60 天的试用版。



要运行本书的示例代码，并不需要下载整个产品。在公共仓库中提供了包含免费开发许可证的 GemFire jar 文件，并且当声明了对 Spring Data GemFire 的依赖时，像 Maven 或 Gradle 这样的构建工具会自动将其下载。如果要使用定位器、管理工具等功能的话，需要安装完整的产品。

## 14.4 通过 Spring XML 命名空间配置 GemFire

Spring Data GemFire 包括一个专有的 XML 命名空间来全面地配置数据网格。事实上，使用 Spring 命名空间来配置 GemFire 是代替其内置 cache.xml 文件的首选方式。由于历史遗留原因，GemFire 将继续支持 cache.xml 配置，但这些都可以通过 Spring XML 来实现，并且可以充分利用 Spring 提供许多高级特性，如模块化的 XML 配置、属性占位符、SpEL 以及环境配置文件（profile）。在命名空间中，Spring Data GemFire 扩展了 Spring FactoryBean 模式来简化 GemFire 组件的创建与初始化。

GemFire 提供几个回调接口，如 CacheListener、CacheWriter 及 CacheLoader，从而允许开发者添加自定义的事件处理器。使用 Spring IoC 容器，它们可以配置成普通的 Spring Bean 并注入到 GemFire 组件之中。cache.xml 提供的配置选项则相对受限制，而且需要回调机制来实现 GemFire 的 Declarable 接口，因此这是对 cache.xml 的一个重大改进。

此外，诸如 Spring Tool Suite (STS) 这样的 IDE 对 XML 命名空间提供了良好的支持，例如代码补全、弹出式注解、实时验证等功能都使得它们更容易使用。

以下各节将使用 GemFire 的 Spring XML 命名空间。若需要更全面的信息，可参阅项目的 Spring Data GemFire 参考指南 (<http://www.springsource.org/spring-gemfire/>)。

### 14.4.1 缓存配置

配置 GemFire 缓存，需要创建一个 Spring Bean 定义文件，并添加 Spring GemFire 命名空间。在 STS 工具中，如图 14-3 所示，选择项目并打开上下文菜单（按右键），选择 New→Spring Bean Configuration file，然后填入文件名称并单击“Next”按钮。

在 XSD 命名空间视图窗口，选择 gfe 命名空间，如图 14-4 所示。

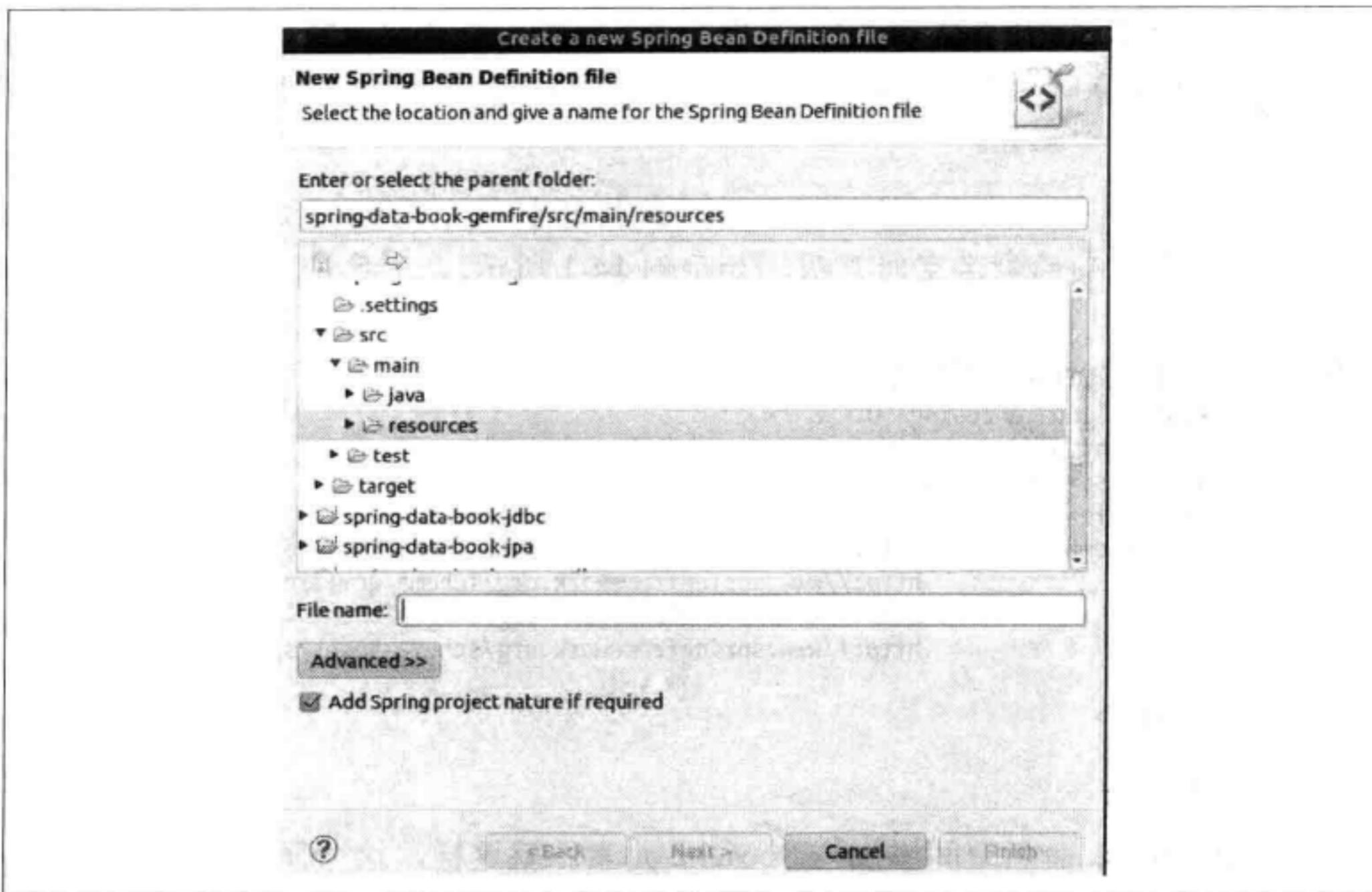


图 14-3 在 STS 创建 Spring Bean 定义文件

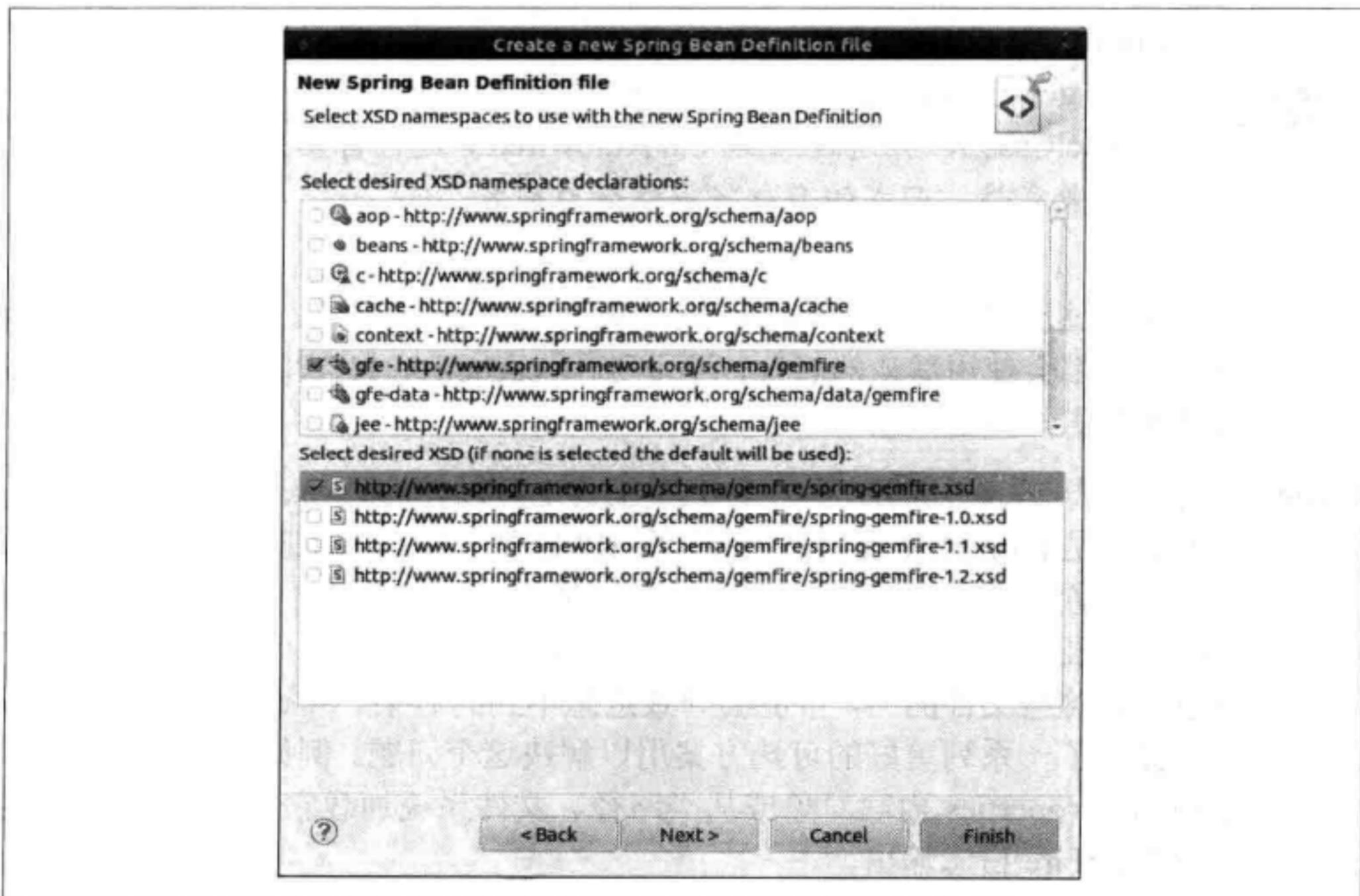


图 14-4 选择 Spring XML 命名空间



除了 gfe 命名空间之外，还有一个 gfe-data 命名空间可用于 Spring Data POJO 映射和以及对 Repository 的支持。gfe 命名空间则用于 GemFire 的核心配置。

单击“Finish”按钮，在 XML 编辑器中会打开 Bean 定义文件，它包含正确的命名空间声明，如示例 14-1 所示。

### 示例 14-1 在 Spring 配置文件声明一个 GemFire 缓存

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

    <gfe:cache/>

</beans>
```

现在可以使用 gfe 命名空间添加一个 cache 元素。就这样，这个简单的缓存声明将创建一个内嵌的缓存，在 Spring 的 ApplicationContext 注册为 gemfireCache，并在初始化上下文时创建它。



Spring Data GemFire 之前版本在创建 Bean 时，默认名称采用连字符（如 gemfire-cache）。在 1.2 版本之后，替换为采用驼峰名称（camelCase），并通过注解（@Autowired）进行自动装配。为了提供向后兼容性，旧式的名称将被注册为别名。

你可以很容易通过设置 cache 元素的 id 属性来改变 Bean 的名称。然而，所有其他的命名空间元素都会使用默认的名称，除非使用 cache-ref 属性来显式覆盖。所以可以依循惯例节省一些工作量。

cache 元素提供了一些额外的属性，在 STS 工具中可以按 Ctrl-Space 键获取友好的提示。最重要的是 properties-ref 属性。除了 API 之外，GemFire 通过外部属性暴露了一些全局的配置选项。在默认情况下，GemFire 会在所有常见的地方（用户的主目录、当前目录及类路径）查找一个名为 gemfire.properties 的文件。这虽然方便，但如果任意放置这些文件的话，可能会导致意想不到的后果。Spring 通过标准的属性加载机制提供了一系列更好的可选方案用以解决这个问题。例如，可以简单地构建一个 java.util.Properties 内联对象或从类路径、文件系统加载它。示例 14-2 使用属性来配置 GemFire 日志输出。

## 示例 14-2 引用属性来配置 GemFire

定义内联属性：

```
<util:properties id="props">
    <prop key="log-level">info</prop>
    <prop key="log-file">gemfire.log</prop>
</util:properties>

<gfe:cache properties-ref="props" />
```

或引用资源的位置：

```
<util:properties id="props" location="gemfire-cache.properties" />

<gfe:cache properties-ref="props" />
```



最好的选择通常是将系统管理员所关心的属性存储在文件系统中约定的位置，而不是将它们定义在 Spring XML 之中或是打包在 jar 文档里面。

注意使用 Spring 的 util 命名空间创建 Properties 对象的方法。这跟 Spring 的属性占位符机制有关系但是又不相同，它使用基于 token 的替换方式，使得可以通过各种来源在外部定义任意 Bean 里的属性。此外，cache 元素包含 cache-xml-location 属性，可以使用 GemFire 的内置配置模式来设置缓存。如前所述，这些主要是由历史遗留原因造成的。

cache 元素也提供了一些 pdx-\* 属性来启用和配置 GemFire 特有的序列化功能 (PDX)，在 14.6 小节“使用 Repository”中会讨论 PDX。

关于缓存的高级配置，cache 元素提供额外的属性来调整内存和网络通信配置（如图 14-5 所示），并且通过子元素注册回调，如 TransactionListers、TransactionWriters 等（如图 14-6 所示）。

**Attribute : lock-lease**  
The timeout, in seconds, for implicit and explicit object lock leases. This affects both automatic locking and manual locking. Once a lock is obtained, it can remain in force for the lock lease time period before being automatically cleared by the system.

**Data Type : string**  
**Default Value : 120**

图 14-5 在 STS 中显示缓存属性列表

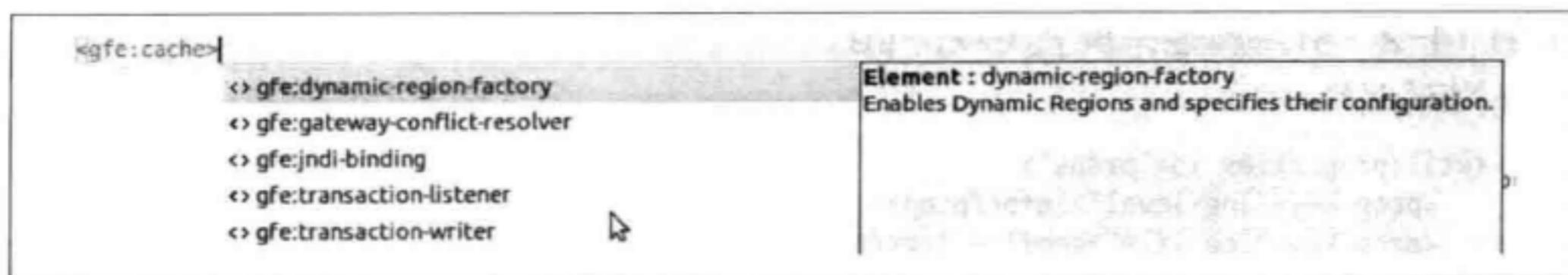


图 14-6 在 STS 中显示子元列表

use-bean-factory-locator 属性（以上图中未显示）是值得说明一下的，这个工厂 Bean 使用 Spring 内部的 BeanFactoryLocator 负责创建缓存，这样用户声明在 GemFire 内置 cache.xml 文件中的类就会注册为 Spring Bean。对于给定的 id 的 cache，BeanFactoryLocator 的实现只允许存在一个 Bean 的定义，在某些情况下，如在 Eclipse 中运行 JUnit 集成测试，则必须将这个值设置为 false 来禁用 BeanFactoryLocator 以防止异常。这个异常也可能在构建脚本中的 JUnit 测试运行期间发生。在这种情况下，测试运行器应该配置为每个测试启用新 JVM（在 Maven 中的配为：<forkmode> always </ forkmode>）。通常将这个值设置为 false 不会发生任何问题。

#### 14.4.2 域配置

如本章开头所述，GemFire 提供了多种类型的域。XML 命名空间定义 replicated-region、partitioned-region、local-region 以及 client-region 元素来创建域，再次强调，这并不能涵盖所有可用的功能，但是突出介绍了一些较为常见的功能。示例 14-3 所示是一个入门级的简单域声明示例。

##### 示例 14-3 基本域声明

```
<gfe:cache/>
<gfe:replicated-region id="Customer" />
```

该域需要依赖缓存。在内部，会通过缓存来创建域。依照惯例，命名空间会被隐式织入。默认缓存声明会创建一个名为 gemfireCache 的 Spring Bean。默认域声明使用相同的约定。换句话说，示例 14-3 等同于：

```
<gfe:cache id="gemfireCache" />
<gfe:replicated-region id="Customer" cache-ref="gemfireCache" />
```

如果愿意的话，可以更改为任何有效的 Bean 的名字，但要确保按需将 cache-ref 设置为相应 Bean 的名称。

通常 GemFire 部署为一个分布式数据网格，复制域或分区域托管在缓存服务器上。客户端应用程序使用客户域来访问数据。对于开发和集成测试，这是一种最佳实践，以消除任何依赖于外部的运行环境。如示例代码所示，可以通过简单声明内

嵌缓存的复制域或本地域。Spring 环境配置文件在配置 GemFire 的不同环境时非常有用。

在示例 14-4 中，dev 配置文件的目的是进行集成测试，而 prod 配置文件用于已部署的缓存配置。缓存和域的配置对应用程序代码是完全透明的。同时也要注意使用属性占位符从外部的属性文件指定定位器的主机和端口的方法。缓存客户端配置将在 14.4.3 小节“缓存客户端配置”中进一步探讨。

#### 示例 14-4 用于开发环境和生产环境的 XML 配置示例

```
<beans profile="dev">
    <gfe:cache/>
    <gfe:replicated-region id="Customer" />
</beans>

<beans profile="prod">
    <context:properties-placeholder location="client-app.properties" />
    <gfe:client-cache pool-name="pool" />

    <gfe:client-region id="Customer" />

    <gfe:pool id="pool">
        <gfe:locator host="${locator.host.1}" port="${locator.port.1}"/>
        <gfe:locator host="${locator.host.2}" port="${locator.port.2}"/>
    </gfe:pool>
</beans>
```



Spring 提供了一些方法来激活相应的环境配置文件，可以在系统属性中设置 spring.profiles.active 属性或 servlet 上下文参数或通过@ActiveProfiles 注解。

如图 14-7 所示，有一些共同的域配置选项以及针对每种不同类型域的特殊配置选项，例如，可以配置将所有域的数据以同步或异步的方式备份到本地磁盘存储。

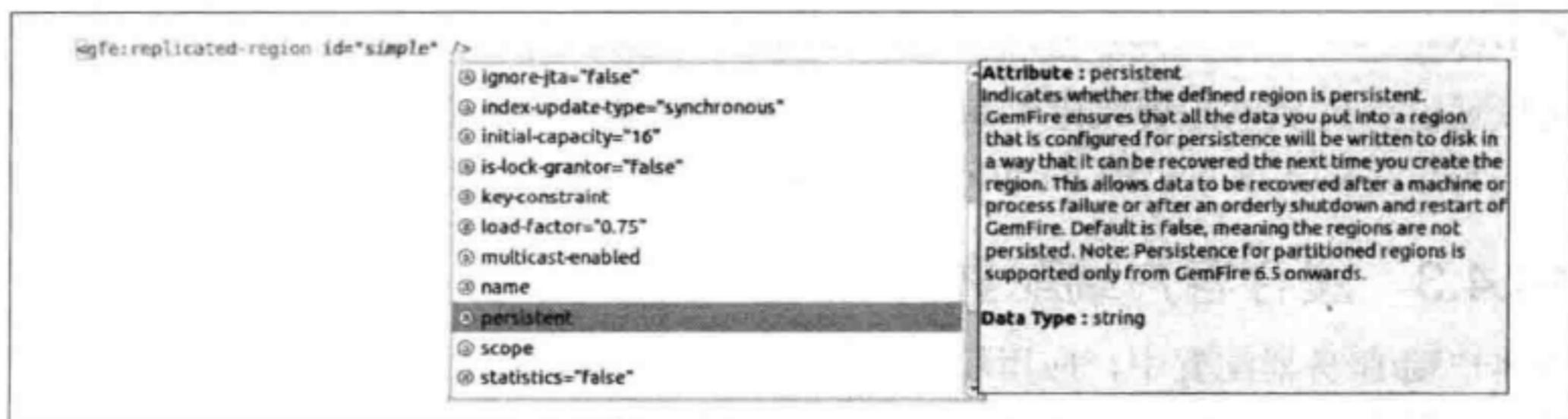


图 14-7 在 STS 工具中显示复制域属性

此外，可以对域进行配置，从而将所选的条目通过 WAN 网关同步分发到遥远的地理区域。也可以注册 CacheListener、CacheLoader 与 CacheWriter 来处理域事件。

这些接口是用来实现相应调用的回调。CacheListener 是通用的事件处理器，每当条目被创建、更新、销毁时会被调用。例如，可以编写一个简单的 CacheListener 来记录缓存事件，这在分布式环境中特别有用（参见示例 14-5）。缓存加载器当缓存未命中（即请求的条目不存在）时 CacheLoader 会被调用，允许到数据库或其他系统资源去读取数据。CacheWriter 当条目更新或创建时被调用，提供“写过程”或“写之后”的能力。

#### 示例 14-5 LoggingCacheListener 实现

```
public class LoggingCacheListener extends CacheListenerAdapter {

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

    @Override
    public void afterCreate(EntryEvent event) {
        String regionName = event.getRegion().getName();
        Object key = event.getKey();
        Object newValue = event.getNewValue();
        log.info("In region [" + regionName + "] created key ["
            + key + "] value [" + newValue + "]");
    }

    @Override
    public void afterDestroy(EntryEvent event) {
        ...
    }

    @Override
    public void afterUpdate(EntryEvent event) {
        ...
    }
}
```

其他的选项包括缓存过期（expiration），也就是域或条目在缓存中保留的最长时间，以及缓存回收（eviction），也就是当达到所定义的最大内存或最大的缓存条数限制时，确定那些条目从缓存中删除的策略。移除的缓存条目可以存储在磁盘溢出区。

可以通过分区域的配置来限制每个分区节点本地内存的分配量、定义使用的桶数等。甚至可以实现自己的 PartitionResolver 来控制数据在分区节点之中如何分布。

### 14.4.3 缓存客户端配置

在客户端/服务器配置中，应用程序是缓存客户端（cache client），它们生产和消费数据，但不直接分发到其他程序。缓存客户端也不会看到远端程序所进行的更新，正如你现在所期望的那样，这完全是可配置的。示例 14-6 展示了使用 client-cache、client-region 及 pool 来设置基本的客户端（client-side）。client-cache 是一个轻量级实现，尤其适用于客户端服务，如管理一个或多个客户域。pool 表示连接池，作为分布式系统的桥梁，它在配置的时候需要任意数量的定位器。



通常情况下，两个定位器已经足够了。第一个是主定位器，另外一个在故障转移切换时使用。每一个分布式系统的成员应使用相同的定位器配置。定位器是一个独立的进程并运行在指定 JVM 中，但这没有严格要求。对于开发和测试，pool 也提供了直接访问缓存服务器的 server 子元素。这对设置简单的客户机/服务器环境非常有用（例如，在本地计算机上），但不建议用于生产系统。正如本章开头介绍的，需要安装完整 GemFire 才能使用定位器，但可以使用 gemfire.jar 所公开的 API 来直接连接服务器进行开发，它最多支持 3 个缓存成员。

#### 示例 14-6 配置缓存池

```
<gfe:client-cache pool-name="pool" />

<gfe:client-region id="Customer" />

<gfe:pool id="pool">
    <gfe:locator host="${locator.host.1}" port="${locator.port.1}"/>
    <gfe:locator host="${locator.host.2}" port="${locator.port.2}"/>
</gfe:pool>
```

可以配置 pool 来控制连接和网络通信的线程分配，值得注意的是 subscription-enabled 属性，必须将其设置为 true 以同步远程程序所发起的域条目事件，如示例 14-7 所示。

#### 示例 14-7 启用缓存池订阅

```
<gfe:client-region id="Customer">
    <gfe:key-interest durable="false" receive-values="true" />
</client-region>

<gfe:pool id="pool" subscription-enabled="true">
    <gfe:locator host="${locator.host.1}" port="${locator.port.1}"/>
    <gfe:locator host="${locator.host.2}" port="${locator.port.2}"/>
</gfe:pool>
```

启用订阅之后，client-region 就可以注册感兴趣的所有键或特定键。订阅可能是持久的，也就是说 client-region 要更新客户端离线期间有可能发生的任何事件。此外，在某些情况下，它还可以通过限制传输值来提高性能，除非对值的获取是显式执行的。比如，在这个示例中新键是可见的，但要取出值的话，必须显式地调用 region.get(key)。

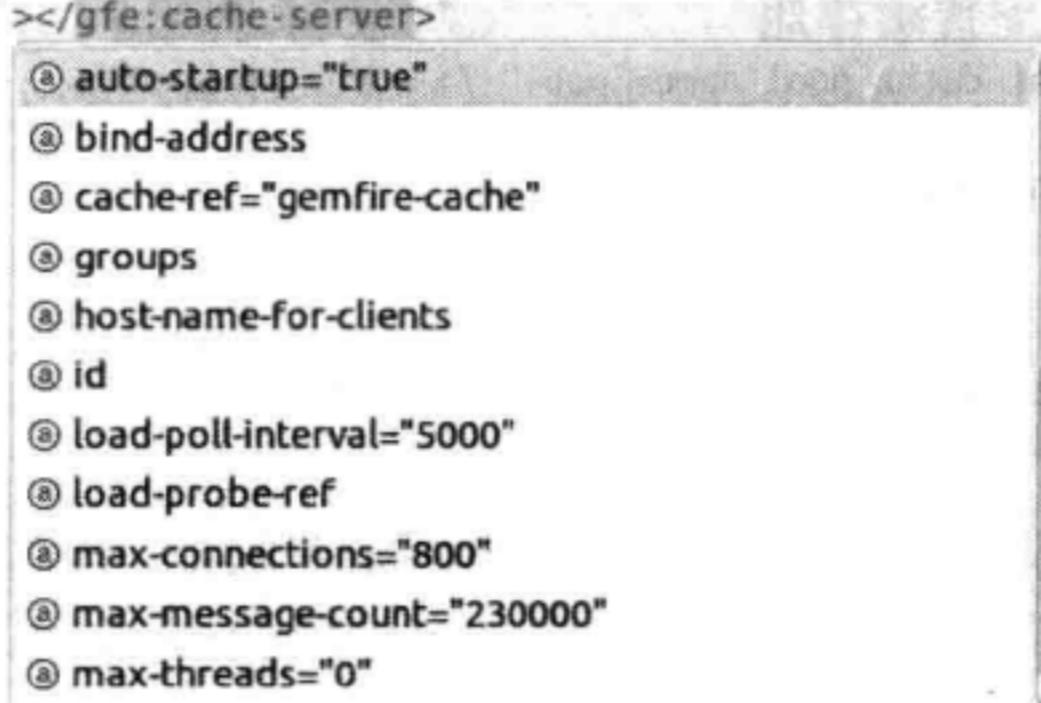
#### 14.4.4 缓存服务端配置

借助于 Spring，只需声明 cache 和 region 并使用额外的 cache-server 元素，即可创建与初始化缓存服务器进程。要启动缓存服务器，只需使用命名空间来配置它并启动应用程序上下文即可，如示例 14-8 所示。

### 示例 14-8 启动一个 Spring 应用程序上下文

```
public static void main(String args[]) {  
    new ClassPathXmlApplicationContext("cache-config.xml");  
}
```

图 14-8 显示 Spring 配置的缓存服务器，它管理了两个分区域和一个复制域。cache-server 暴露了许多参数，可用来调整网络通信、系统资源等。



```
<gfe:cache>  
<gfe:partitioned-region id="Customers"/>  
<gfe:partitioned-region id="Orders"/>  
<gfe:replicated-region id="Products"/>  
  
<gfe:cache-server></gfe:cache-server>  
</beans>  
⑧ auto-startup="true"  
⑧ bind-address  
⑧ cache-ref="gemfire-cache"  
⑧ groups  
⑧ host-name-for-clients  
⑧ id  
⑧ load-poll-interval="5000"  
⑧ load-probe-ref  
⑧ max-connections="800"  
⑧ max-message-count="230000"  
⑧ max-threads="0"
```

图 14-8 配置缓存服务器

#### 14.4.5 WAN 配置

对于地域分布式系统来说，需要 WAN 的配置。例如，一个全球的组织可能需要跨越东京、伦敦和纽约的办公室来共享数据。每个地点都在本地端管理它的事务，但是远程的地点需要与其进行同步。因为从性能和可靠性方面来说 WAN 通信是非常昂贵的，所以 GemFire 会将事件放入到队列中并通过广域网的网关处理，以达到最终的一致性。我们可以控制每一个远程位置要对哪个事件进行同步，也可以调整内部的队列大小、同步调度、持久化备份等。深入讨论 GemFire 的 WAN 网关架构已超出了本书的范围，但需要特别注意的是 WAN 同步必须在域级别启用，如示例 14-9 所示的配置。

### 示例 14-9 GemFire WAN 配置

```
<gfe:replicated-region id="region-with-gateway" enable-gateway="true" hub-id="gateway-hub" />  
  
<gfe:gateway-hub id="gateway-hub" manual-start="true">  
    <gfe:gateway gateway-id="gateway">  
        <gfe:gateway-listener>  
            <bean class="..."/>  
        </gfe:gateway-listener>  
        <gfe:gateway-queue maximum-queue-memory="5" batch-size="3" batch-time-interval="10" />
```

```
</gfe:gateway>

<gfe:gateway gateway-id="gateway2">
    <gfe:gateway-endpoint port="1234" host="host1" endpoint-id="endpoint1" />
    <gfe:gateway-endpoint port="2345" host="host2" endpoint-id="endpoint2" />
</gfe:gateway>
</gfe:gateway-hub>
```

这个示例展示了使用 GemFire 6 的 API 来启用域的 WAN 通信，必须设置 enable-gateway 属性为 true（或者通过 hub-id 属性来标示），而且 hub-id 必须引用 gateway-hub 元素。在这里，我们看到 gateway-hub 配置了两个网关。第一个配置了可选的 GatewayListener，用来处理网关事件，另外它还配置网关队列，第二个定义了两个远程网关端点。



即将发布 GemFire 7.0 将会修改 WAN 架构，会包括一些新功能以及 API，通常也会改变网关配置方式。Spring Data GemFire 计划同时发布一个支持所有 GemFire 7.0 新功能的版本，而目前的 WAN 架构不建议使用。

#### 14.4.6 磁盘存储配置

GemFire 可以配置磁盘存储来保存域备份、磁盘溢出时回收缓存、WAN 网关等。因为磁盘存储可以用于多种目的，所以它在命名空间中被定义为最顶层元素，使用它的组件可对它进行引用。磁盘写操作可选择同步或异步。

对于异步写入，条目保存在队列中，这是可配置的。其他可选项能够用来控制调度（例如，磁盘写操作之前可消耗的时间或以 MB 表示的最大文件大小）。在示例 14-10 中，配置了一个溢出磁盘存储来保存被移除的条目。对于异步写入，可在队列中最多存储 50 个条目，每隔 10 秒或者当队列达到容量时会进行刷新。该域配置为如果总的内存大小超过 2GB 就进行回收。自定义 ObjectSizer 用来估算每个条目所分配的内存。

##### 示例 14-10 磁盘存储配置

```
<gfe:partitioned-region id="partition-data" persistent="true" disk-store-ref="ds2">
    <gfe:eviction type="MEMORY_SIZE" threshold="2048" action="LOCAL_DESTROY">
        <gfe:object-sizer>
            <bean class="org.springframework.data.gemfire.SimpleObjectSizer" />
        </gfe:object-sizer>
    </gfe:eviction>
</gfe:partitioned-region>

<gfe:disk-store id="ds2" queue-size="50" auto-compact="true"
    max-oplog-size="10" time-interval="10000">
    <gfe:disk-dir location="/gemfire/diskstore" />
</gfe:disk-store>
```

## 14.5 使用 GemfireTemplate 进行数据访问

Spring Data GemFire 提供了一个类似于 JdbcTemplate、JmsTemplate 的数据访问模板类，GemfireTemplate 包装了一个域，并提供了简单的数据访问和查询方法以及回调接口来访问域操作。使用 GemfireTemplate 的主要原因之一就是它可以将 GemFire 检查异常转换为 Spring 的 PersistenceException 运行时异常体系。这简化了原生 Region API 所需的异常处理，并且还可以使用模板与 Spring 的声明式事务更加无缝结合，在这里会用到 GemfireTransactionManager，它与其他所有的 Spring 事务管理器类似，在默认情况下可以对运行时异常执行回滚（但不是检查型异常）。@Repository 组件也可进行异常转换，事务将运行在使用@Transactional 的方法上，该方法会直接使用 Region 接口，但是需要更进一步的处理。

示例 14-11 简单地展示了使用 GemfireTemplate 的数据访问对象。注意 findByLastName(...) 方法的实现需要调用 template.query()。GemFire 使用对象查询语言（Object Query Language, OQL）来进行查询。此方法只需要一个 boolean 断言来定义查询条件，查询的主体假定为 SELECT \* from [region name] WHERE...。该模板还实现了 find(...) 与 findUnique(...) 方法，接受参数化的查询字符串以及相关的参数，并且隐藏了 GemFire 底层的 QueryService API。

### 示例 14-11 使用 GemfireTemplate 实现 Repository

```
@Repository
class GemfireCustomerRepository implements CustomerRepository {
    private final GemfireTemplate template;

    @Autowired
    public GemfireCustomerRepository(GemfireTemplate template) {
        Assert.notNull(template);
        this.template = template;
    }

    /**
     * Returns all objects in the region. Not advisable for very large datasets.
     */
    public List<Customer> findAll() {
        return new ArrayList<Customer>((Collection<? extends Customer>) ←
            template.getRegion().values());
    }

    public Customer save(Customer customer) {
        template.put(customer.getId(), customer);
        return customer;
    }

    public List<Customer> findByLastname(String lastname) {
        String queryString = "lastname = '" + lastname + "'";
        ...
    }
}
```

```

        SelectResults<Customer> results = template.query(queryString);
        return results.asList();
    }

    public Customer findByEmailAddress(EmailAddress emailAddress) {

        String queryString = "emailAddress = ?1";
        return template.findUnique(queryString, emailAddress);
    }

    public void delete(Customer customer) {
        template.remove(customer.getId());
    }
}

```

可以将 GemfireTemplate 配置为普通的 Spring Bean，如示例 14-12 所示。

#### 示例 14-12 GemfireTemplate 配置

```

<bean id="template" class="org.springframework.data.gemfire.GemfireTemplate">
    <property name="region" ref="Customer" />
</bean>

```

## 14.6 使用 Repository

Spring Data GemFire 1.2.0 发布版本引入了 Spring Data Repository 对 GemFire 的基础支持。它支持第 2 章中所提到的 Repository 核心功能，但不包含分页和排序。示例代码将会演示这些功能。

### 14.6.1 POJO 映射

因为在 GemFire 域中，每个对象需要一个唯一键，所以顶级领域对象 Customer、Order 与 Product 都继承自 AbstractPersistentEntity，它定义了 id 属性，如示例 14-13 所示。

#### 示例 14-13 AbstractPersistentEntity 领域类

```

import org.springframework.data.annotation.Id;

public class AbstractPersistentEntity {

    @Id
    private final Long id;
}

```

每个领域对象都使用了@Region 注解。按照默认的约定，域名称与类的简单名称相同。但是，我们可以将注解值设为自定义的域名来覆盖它。如果已提供域元素的话，它必须与域名称相对应，也就是域元素的 id 属性或 name 属性的值。常见的属性，例如@PersistenceConstructor（如示例 14-14 所示）和@Transient 的工作方式与我们的预期相符。

#### 示例 14-14 Product 领域类

```
@Region
public class Product extends AbstractPersistentEntity {

    private String name, description;
    private BigDecimal price;
    private Map<String, String> attributes = new HashMap<String, String>();

    @PersistenceConstructor
    public Product(Long id, String name, BigDecimal price, String description) {

        super(id);
        Assert.hasText(name, "Name must not be null or empty!");
        Assert.isTrue(BigDecimal.ZERO.compareTo(price) < 0, "Price must be greater than zero!");

        this.name = name;
        this.price = price;
        this.description = description;
    }
}
```

#### 14.6.2 创建 Repository

GemFire Repository 支持基本的 CRUD 与查询操作，可以使用 Spring Data 通用的方法名查询映射机制来定义。此外，可以使用@Query 来配置 Repository 的方法执行任意的 OQL 查询，如示例 14-15 所示。

#### 示例 14-15 ProductRepository 接口

```
public interface ProductRepository extends CrudRepository<Product, Long> {

    List<Product> findByDescriptionContaining(String description);

    /**
     * Returns all {@link Product}s having the given attribute value.
     * @param attribute
     * @param value
     * @return
     */
    @Query("SELECT * FROM /Product where attributes[$1] = $2")
    List<Product> findByAttributes(String key, String value);

    List<Product> findByName(String name);
}
```

可以使用专有的 gfe-data 命名空间来启用 Repository 的查找功能，它与核心的 gfe 命名空间并没有在一起。另外，如果使用 Java 配置的话，只要在配置类简单地添加@EnableGemfireRepositories 注解即可，如示例 14-16 所示。

#### 示例 14-16 使用 XML 启用的 GemFire Repository

```
<gfe-data:repositories base-package="com.oreilly.springdata.gemfire" />
```

#### 14.6.3 PDX 序列化

PDX 是 GemFire 特有的序列化库，它具备高效率、可配置特性并且可以与 C# 或 C++

编写的 GemFire 客户端应用程序交互，还支持对象版本化的特性。在一般情况下，对象进行网络传输和磁盘持久化的操作时必须要被序列化。如果缓存条目已经序列化，就会以序列化的形式存储。对于分布式拓扑结构通常也是这样。没有持久化备份的独立缓存，通常不进行序列化。

如果没有启用 PDX，将会使用 Java 序列化。在这种情况下，领域类和所有非瞬态属性必须要实现 `java.io.Serializable`。但 PDX 并不需要这样做，此外，PDX 是高度可配置的，可以个性化定制从而对序列化进行优化或增强，以满足应用的需求。

示例 14-17 展示了如何使用 PDX 配置 GemFire Repository。

#### 示例 14-17 配置 MappingPdxSerializer

```
<gfe:cache pdx-serializer="mapping-pdx-serializer" />  
  
<bean id="mapping-pdx-serializer"  
      class="org.springframework.data.gemfire.mapping.MappingPdxSerializer" />
```

MappingPdxSerializer 会自动织入默认的映射上下文，这也是 Repository 所使用的。这里有一个限制需要注意，那就是每个缓存的实例只能有一个 PDX 序列器，因此如果正在使用 PDX Repository 的话，最好设置专用的缓存节点（即不要使用同样的程序来管理非 Repository 的域）。

## 14.7 支持持续查询

GemFire 一个非常强大的功能就是支持持续查询（Continuous Query, CQ），它提供了一个查询驱动（query-driven）事件通知功能。在传统的分布式应用中，如果数据消费者依赖于其他程序的准实时更新，就必须实现某种类型的轮询机制。这不是特别高效，也缺乏可伸缩性。而使用发布/订阅消息系统的话，应用程序收到一个事件后，通常会访问基于磁盘数据存储的相关数据。持续查询提供了极其高效的替代方案，通过 CQ，客户端应用程序可以在缓存服务器上注册一个定期执行的查询，客户端也可以提供回调，当域事件影响查询结果集状态时会调用这个方法。注意 CQ 需要客户机/服务器配置。

Spring Data GemFire 提供了 `ContinuousQueryListenerContainer`，它支持基于 Spring `DefaultMessageListenerContainer` 的编程模型，这个模型可以用于 JMS 消息驱动的 POJO。要配置 CQ，需要使用命名空间来创建 CQLC，并且为每一个持续查询注册监听器，如示例 14-18 所示。请注意，因为 CQ 使用 GemFire 的订阅机制，pool 必须将 `subscription-enabled` 属性设置为 `true`。

#### 示例 14-18 配置 ContinuousQueryListenerContainer

```
<gfe:client-cache pool-name="client-pool" />  
  
<gfe:pool id="client-pool" subscription-enabled="true">
```

```
<gfe:server host="localhost" port="40404" />
</gfe:pool>

<gfe:client-region id="Person" pool-name="client-pool" />

<gfe:cq-listener-container>
    <gfe:listener ref="cqListener" query="select * from /Person" />
</gfe:cq-listener-container>

<bean id="cqListener" class="org.springframework.data.gemfire.examples.CQListener" />
```

示例 14-19 为监听器的实现。

#### 示例 14-19 持续查询监听器的实现

```
public class CQListener {

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

    public void handleEvent(CqEvent event) {
        log.info("Received event " + event);
    }
}
```

在查询的过程中，任何程序发生改变时，`handleEvent()`方法都将被调用。请注意 `CQListener` 并不需要实现任何接口，也没有要求什么特别的方法名。持续查询容器会非常智能地自动调用只有 `CqEvent` 参数的方法。如果这样的方法不止一个，那就需要在监听器的配置中声明方法的名字了。

---

# 关于封面

*Spring Data*封面上的动物是巨松鼠（巨松鼠属），是世界上最大的松鼠。这种松鼠在亚洲的热带森林中都能找到，它们身体由两种显眼的颜色组成，两个耳朵之间有着独特的白色斑点。成年巨松鼠的头部和身体能够达到14英寸，而尾巴的长度则能接近2英尺。它们的耳朵是圆形的并具有便于爬行的利爪。

健康的成年巨松鼠重量能够达到4.5英镑。它们的颜色有褐色、铁锈色、棕色以及米黄色，最多可能达到280种。它们是食草性的，靠花朵、水果、蛋、昆虫甚至树皮为生。

巨松鼠是一种树上筑巢的物种，很少会离开树，会在很高的树枝上筑巢。它们在树之间跳跃，可以跳到20英尺高。当遇到危险时，巨松鼠通常会不动并借助树干保护自己，而不是逃跑。它主要的天敌是凶猛的鸟类或豹子。巨松鼠主要在每天的早晨或晚间活动，在白天休息。它是一种害羞且机警的动物，不太容易被发现。

封面图片来自*Shaw's Zoology*。