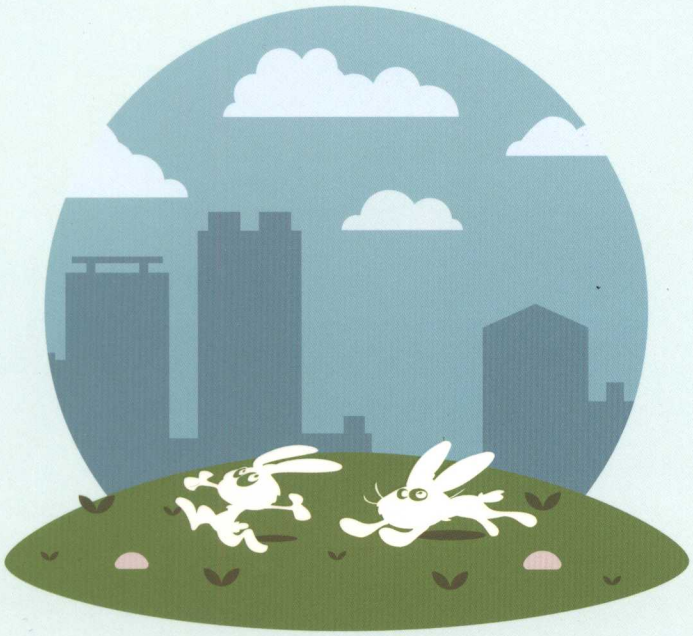
着重使用简单易用的方法，高效地使用Neo4j图数据库，
轻松发掘现实世界中关联数据的实用价值。

Broadview®
www.broadview.com.cn



Neo4j full stack development

罕有的中文原版Neo4j资料，一册在手，无忧使用Neo4j图数据库。

Neo4j全栈开发

陈韶健 / 著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

作者简介

陈韶健

具有15年以上IT从业经历的资深专家。在编程语言使用上，精通.NET和Java两大体系，尤其对Spring和Spring Boot有深入研究，并著有《深入实践Spring Boot》一书，于2016年11月在机械工业出版社出版。在数据库方面，熟悉SQL Server、Oracle、MySQL等传统关系型数据库，以及Redis、MongoDB、Neo4j等NoSQL数据库，并对Neo4j有更多的爱好和深入的研究。另外，在系统设计、服务器架构设计、数据安全和性能优化等方面都有丰富的实践经验。

Neo4j全栈开发

陈韶健 / 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内容简介

本书全面、系统地介绍了 Neo4j 这个独特而又高性能的 NoSQL 图数据库，从使用 Neo4j 进行程序开发，到 Neo4j 的管理和配置等层面全方位地阐释了 Neo4j 的整个生态体系。

本书不仅着重介绍了怎样以简单易用的方式来使用 Neo4j，更难能可贵的是，本书还分享了使用分布式 Neo4j 构建高可用的读/写分离负载均衡配置的实际操作过程和实现细节。

通过对本书的学习，读者将系统地掌握 Neo4j 的知识，并很快将其用于项目开发之中，为自己的应用提升访问性能，解决燃眉之急。

图书在版编目（CIP）数据

Neo4j 全栈开发 / 陈韶健著. —北京：电子工业出版社，2017.6

ISBN 978-7-121-31447-6

Ⅰ. ①N… Ⅱ. ①陈… Ⅲ. ①关系数据库系统 Ⅳ.①TP311.132.3

中国版本图书馆 CIP 数据核字（2017）第 095309 号

策划编辑：安　娜
责任编辑：徐津平
特约编辑：赵树刚
印　　刷：北京中新伟业印刷有限公司
装　　订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　　印张：19.75　　字数：411 千字
版　　次：2017 年 6 月第 1 版
印　　次：2017 年 6 月第 1 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

前　言

在高速发展的互联网应用中，业务需求的频繁变更和数据的快速增长都要求数据库必须具有很强的适应能力。Neo4j 图数据库正是一个能够适应这种业务需求不断变化和大规模数据增长而产生的数据库，它不但具有很强的适应能力，而且能够自始至终保持高效的查询性能。

现实世界中的一切事物都处在联系之中，如人际关系、电脑网络、地理数据、分子结构模型等，无一不处在纷繁复杂的联系之中。这种联系形成了一种互相关联的数据，联系才是数据的本质所在。传统的关系型数据库并不能很好地表现数据的联系，而一些NoSQL（Not Only SQL，非关系型数据库）数据库也不能表现数据之间的联系。同样是NoSQL的Neo4j图数据库是以图的结构形式来存储数据的，它所存储的就是联系的数据，是关联数据本身。

关联数据中的联系本来就很复杂，若要在关系型数据库中使用结构化形式来表现这种联系，则一般不能直接表示，处理起来既烦琐又费事，并且随着数据的不断增长，其访问性能将日趋下降。无数的开发人员和数据库管理人员都或多或少地使用过关系型数据库，在其应用的规模化进展过程中，对于数据库的性能优化往往捉襟见肘、陷入窘境。Neo4j 没有模式结构的定义，也不需要这些定义，它使用非结构化的方式来存储关联数据，所以能够直接表现数据的关联特性。

Neo4j 不管是与关系型数据库相比，还是与其他 NoSQL 数据库相比，都具有很多前所未有的优势，主要表现在以下几个方面。

1. 优越的性能表现

Neo4j 具有永久高效的读取和写入能力，这种能力与数据库的大小无关，不管是初始创建的数据库，还是用了很长时间、积累了大量数据的数据库，Neo4j 始终能保持闪电般的读/写速度。

2. 设计的灵活性

因为 Neo4j 没有模式结构定义的约束，并且由于图结构的自然伸展特性，都给 Neo4j 提供了无限广阔的灵活设计空间，因为无论是扩展设计，还是增加数据，都不会影响到原来数据的正常使用。

3. 迭代的敏捷性

正是由于 Neo4j 的灵活设计特性及其图结构数据的可伸缩性等特点，使其能追上业务需求变化发展的脚步，并且能适用于频繁迭代的敏捷开发方法。

4. 安全可靠的特性

Neo4j 不仅支持完整的事务管理特性，而且提供了实时在线备份功能，以及应对灾难事故进行日志恢复的方法，所有这些都充分说明了 Neo4j 是一个安全可靠的数据库。

5. 简单易用的特性

Neo4j 在使用上非常简单，不管是使用 Java 开发语言，还是使用其他开发语言，如 Python、Ruby、PHP、.NET、Node.js 等，都能够非常方便地访问 Neo4j。特别是 Spring Data Neo4j 开发包，更是提供了一整套非常简单易用的 Neo4j 数据库使用方法。

6. 丰富的学习资源

Neo4j 的社区版滋生了一个非常活跃的社区，在这个社区中，诸多开发者提供了非常丰富的使用 Neo4j 的案例——GraphGists，这是学习使用 Neo4j 的极好资源。通过对这些 GraphGists 的学习和交流，不仅能拓展你的思路，更能让你的开发工作变得更加简单和容易，而且还能帮助你快速构建应用的商业模型。

7. 大企业的考验

Neo4j 拥有广大而又有实力的用户群体，并且经过几年时间的运行实践，充分验证了它的稳定性和健壮性。如思科、沃尔玛、阿迪达斯等公司，都在使用 Neo4j 的过程中挖掘到了图数据库的巨大威力，并且创造出了蓬勃发展的商业模型。

综上所述，使用如此优秀的数据库，不仅可以提升一个应用的性能，而且可以适应大规模的数据增长，同时还能减轻开发人员和数据库管理人员的工作负担，给你和你的企业以及你的用户带来前所未有的优越体验。

读者对象

本书适合所有开发人员，特别是 Spring Boot 开发者阅读，同时适合数据库管理人员和系统设计人员学习使用，并可作为系统策划者进行数据库选型的参考资料。

实例代码下载

本书各章的实例代码下载在各个章节中都有明确说明，同时也可以通过以下网址选择不同项目进行下载或检出：

https://github.com/mr-csj?tab=repositories

勘误与反馈

如果有问题反馈则可以通过以下链接发起话题，而且如果因为编辑或排版出错需要勘误则也会首先在这里发表：

https://github.com/mr-csj/discuss/issues

由于时间仓促，加之作者水平所限，书中难免存在纰漏或错误之处，敬请读者批评指正！

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误**：您对书中内容的修改意见可在【提交勘误】处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **与作者交流**：在页面下方【读者评论】处留下您的疑问或观点，与作者和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/31447

目 录

第1章
Neo4j概述

什么是Neo4j呢？Neo4j是一个NoSQL的图数据库管理系统。这里所说的图是指图论中的图这种数据结构，图是一个比线性表和树更高级的数据结构。在Neo4j中，图表示为一些节点和连接这些节点的关系的集合，其中，节点表示实体，关系表示实体之间的连接方式。

在Neo4j中存储的关联数据表现为树状或网络状的形形色色的图，它更加形象和直观地表现了现实世界中的应用场景。Neo4j不但能给人一种耳目一新的感觉，更重要的是它能始终保持高效的查询性能，不会因为数据的增长而降低了查询的反应能力。

Neo4j是一个NoSQL数据库，像其他NoSQL数据库一样具有高效的查询性能。同时，Neo4j还具有完全事务管理特性，完全支持ACID（Atomicity，Consistency，Isolation，Durability）事务管理。

实践证明，图数据库具有很强的表现力。像Facebook中巨大的社交数据，Google搜索引擎的海量网页，或者现实世界中繁杂的交通网络，大至宏观世界的天文数据，小至微观世界的量子模型等，现实世界中不同领域的数据都可以使用图数据库来存储和访问。

Neo4j自2010年2月发布1.0.0版本，经过几年时间一些大中型企业的使用实践，可以充分证明Neo4j是一个成熟的数据库，同时也是一个安全可靠的数据库管理系统。

1.1 Neo4j 数据的特点

在 Neo4j 数据库中存储的图数据，相比于大家比较熟悉的关系型数据库来说，它没有模式结构（如表或视图等逻辑结构的定义），而是用节点和关系的属性来表现实体的内容。使用属性，可以让 Neo4j 的图数据具有更加出色的表现能力，使其像关系型数据库那样包含非常丰富的内容。

Neo4j 的图数据结构具有如下基本特征：

- 节点、关系和属性是构成图数据的三个基本要素。
- 节点和关系的属性是一个 Key-Value 的数据集合。
- 每个关系都有一个开始节点和一个结束节点相互连接。
- 大多数情况下关系可以不需要属性。

从以上特征中可以看出，Neo4j 存储的数据是一个属性图。其中，节点表示一个实体，实体可以是人、事物或者任何一种定义，节点的属性就是实体的内容。而实体之间的关联表现为节点的关系，比如朋友关系、从属关系等。

从总体来看，Neo4j 就是由无数相互联系的节点所组成的图形，它能很好并且形象地表现出现实世界中相互联系的事物。从这一方面来说，Neo4j 存储的数据更能体现事物之间相互联系的本质，这种关联数据表现出了现实场景中事物本来的样子。

1.2 Neo4j 数据的表现形式

由于 Neo4j 并不需要模式结构定义，因而非常适合用来存储非结构化或半结构化的数据，所以在数据表现形式上，与 RDBMS 或其他 NoSQL 数据库相比，都具有绝对的优势。

例如，在使用 RDBMS 设计数据库时，一般遵循这样一条规则：首先要做一些数据模型的分析，然后再进行实体-关系模型的设计，接下来才开始定义一些表结构。在这个

过程中，必不可少地需要考虑一些范式规范，以便让存储的数据可以避免过多的冗余。最后可能为了一些性能要求，还会反过来使用一些反范式设计，再增加一点冗余字段。经过所有这些工作之后，才能确定数据结构的定义。而当数据结构确定之后，就不能再有大的变更了，否则整个系统结构都要跟着更改。

纵观使用 RDBMS 的整个设计和开发过程，有时你会觉得，在这种情况下的程序开发者似乎显得太蹩脚了。虽然明知这些工作很烦琐，但还是要求你必须得这样做。

使用 Neo4j 来设计数据库，就不需要有这么多繁琐的过程，我们只需借助于在做业务需求时画出的一些简单框图，就可以用来作为数据模型，很容易地将其转化为图数据结构。

例如，如图 1-1 所示的简单框图，它可以表示有用户和部门两个实体，并且它们之间存在一种隶属关系。至于在框图中是使用方框还是圆框则完全根据个人喜好，并没有什么区别。在使用 Neo4j 来进行数据库设计时，使用这个简单的框图就足够了。要用 Neo4j 来表现这一数据，将会存储部门和用户这两个节点，以及一个连接这两个节点的隶属关系。

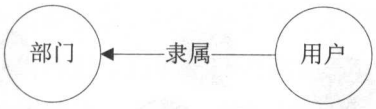


图 1-1　数据模型简单框图

如果我们按照这个数据模型在 Neo4j 中创建一些数据，实际的数据就可能表现为如图 1-2 所示的形式，它的样子跟原来的数据模型框图很相似。这些数据表明，现在有三个部门，分别是行政部、开发部和市场部，行政部有一个用户张三；开发部有两个用户，分别是李四和王一；市场部有三个用户，分别是小明、李红和小张。

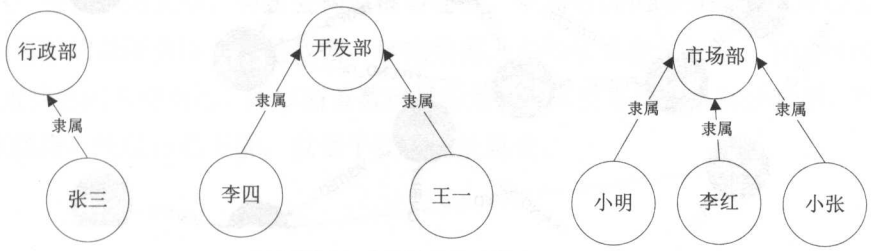


图 1-2　图数据实例

如果在 Neo4j 的 Web 控制台中查看这些数据，那么它的样子会显得更加漂亮和有趣，如图 1-3 所示。

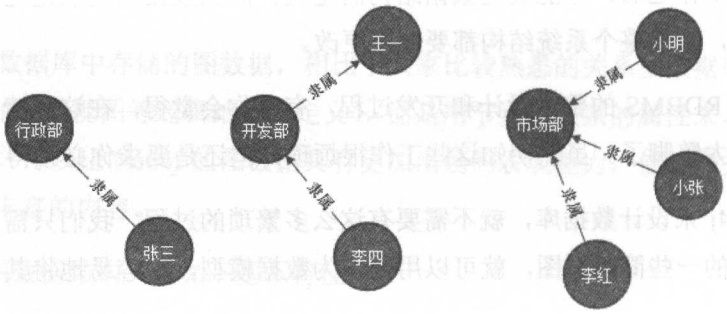


图 1-3　在 Neo4j 中的图数据实例

在这里，节点和关系的数量就体现了数据本身的数量，当我们再添加一定数量的数据之后，整个数据库就变得更加可观和有趣了。如图 1-4 所示是一个应用实例——电影社区推荐引擎的一些测试数据。从这幅图中可以看到星罗棋布的节点由各种关系连接在一起，很像一个网络图形。

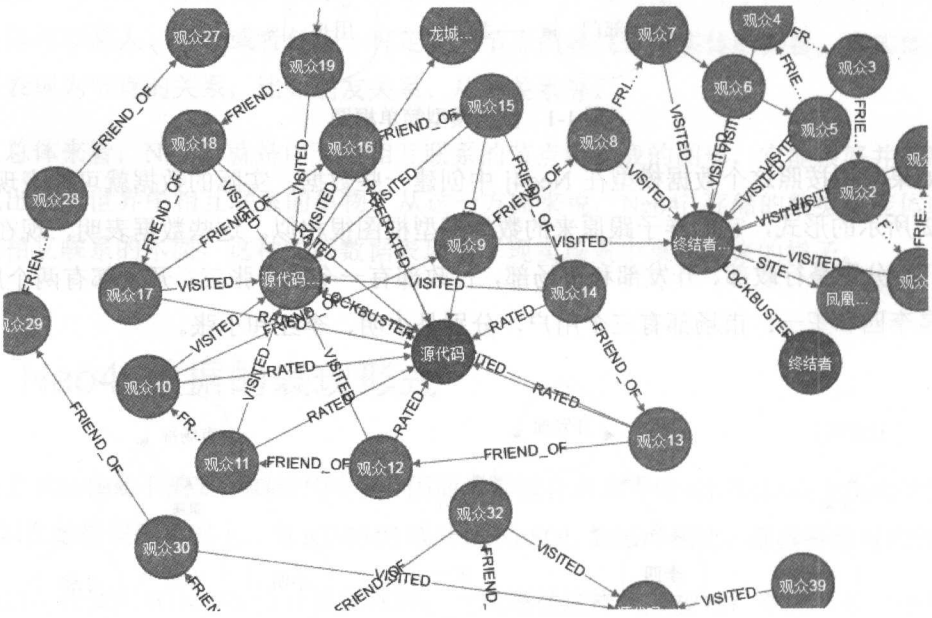


图 1-4　电影社区推荐引擎的图数据

在早期一些介绍 Neo4j 的文章中，也曾有人把图数据库形象地描述为网络数据库。如果数据量很大，那么 Neo4j 图数据的样子确实很像一个巨型网络。

对于一个能够如此活灵活现地表现现实世界中的关联数据的数据库，不要认为在使用上会很难、很复杂，相反，对 Neo4j 的管理和开发是非常简单并且充满趣味的。因为使用 Neo4j 图数据库，你并不需要去了解那些复杂的图数据结构理论，以及那些让人头痛的公式和符号，你只要能看懂如图 1-1 所示的简单框图的意义就可以了。

1.3　Neo4j 的优势

那么，如此引人入胜的 Neo4j，与其他数据库相比，具有哪些明显的优势呢？这可以从以下几个方面来分析，主要表现为查询的高性能、设计的灵活性和开发的敏捷性等。

1.3.1　查询的高性能

Neo4j 是一个原生的图数据库引擎，它存储了原生的图数据，因此，可以使用图结构的自然伸展特性来设计免索引邻近节点遍历的查询算法，即图的遍历算法设计。图的遍历是图数据结构所具有的独特算法，即从一个节点开始，根据其连接的关系，可以快速和方便地找出它的邻近节点。这种查找数据的方法并不受数据量的大小所影响，因为邻近查询查找的始终是有限的局部数据，而不会对整个数据库进行搜索。所以，Neo4j 具有非常高效的查询性能，相比于 RDBMS，它的查询速度可以提高数倍乃至数十倍。而且查询速度不会因数据量的增长而下降，即数据库可以经久耐用，并且始终保持最初的活力。不像 RDBMS 那样，因为不可避免地使用了一些范式设计，所以在查询时如果需要表示一些复杂的关系，势必会构造很多连接，从而形成很多复杂的运算。并且在查询中更加可怕的是还会涉及大量数据，这些数据大多与结果毫无关系，有的可能仅仅是通过 ID 查找它的名称而已，所以随着数据量的增长，即使查询一小部分数据，查询也会变得越来越慢，性能日趋下降，以至于让人无法忍受。

1.3.2 设计的灵活性

在日新月异的互联网应用中，业务需求会随着时间和条件的改变而发生变化，这对于以往使用结构化数据的系统来说，往往很难适应这种变化的需要。图数据结构的自然伸展特性及其非结构化的数据格式，让 Neo4j 的数据库设计可以具有很大的伸缩性和灵活性。因为随着需求的变化而增加的节点、关系及其属性并不会影响到原来数据的正常使用，所以使用 Neo4j 来设计数据库，可以更接近业务需求的变化，可以更快地赶上需求发展变化的脚步。

大多数使用关系型数据库的系统，为了应对快速变化的业务需求，往往需要采取推倒重来的方法重构整个应用系统。而这样做的成本是巨大的。使用 Neo4j 可以最大限度地避免这种情况发生。虽然有时候，也许是因为最初的设计考虑得太不周全，或者为了获得更好的表现力，数据库变更和迁移在所难免，但是使用 Neo4j 来做这项工作也是非常容易的，至少它没有模式结构定义方面的苦恼。

1.3.3 开发的敏捷性

图数据库设计中的数据模型，从需求的讨论开始，到程序开发和实现，以及最终保存在数据库中的样子，直观明了，似乎没有什么变化，甚至可以说本来就是一模一样的。这说明，业务需求与系统设计之间可以拉近距离，需求和实现结果之间越来越接近。这不但降低了业务人员与设计人员之间的沟通成本，也使得开发更加容易迭代，并且非常适合使用敏捷开发方法。

Neo4j 本身可伸缩的设计灵活性，以及直观明了的数据模型设计，以及其自身简单易用的特点等，所有这些优势都充分说明，使用 Neo4j 很适合以一种测试驱动的方法应用于系统设计和开发自始至终的过程之中，通过迭代来加深对需求的理解，并通过迭代来完善数据模型设计。

1.3.4 与其他数据库的比较

与当前一些主流的数据库相比，不管是传统的关系型数据库，还是 NoSQL 数据库，或者同类的图数据库，Neo4j 都是出类拔萃的。

在传统的 RDBMS 中，如果要表现一个部门的用户，即 1.2 节提到的例子，按照第三范式的设计要求，至少需要三张表格来表示，即部门表、用户表和部门-用户关系表，这样实体和关系就被人为地隔开了，它们是完全分离的，存在于不同的表中，这就给查询带来了一定的难度，从而影响了查询的性能。而 Neo4j 所表现的是实体的联系本身，它表现了现实世界中事物联系的本质，它的联系在节点创建时就已经建立，所以在查询中能以快捷的路径返回关联数据，从而表现出非常高效的查询性能。

Key-Value 的数据库虽然能提供高性能的查询，但它所能表示的内容是有限的。实际上，Neo4j 节点的属性就是一些 Key-Value 的数据集合。而 Neo4j 通过节点和关系的属性可以表现更为丰富多彩的内容，这是其他 Key-Value 的数据库所无法比拟的。

对于 Key-Document 文档数据库来说，相对于 Key-Value 数据库，内容是丰富了些，但美中不足的是，一个文档经不起内容的变更或修改。如果用 Neo4j 的节点及其属性来表示，处理类似的变更则轻而易举。

在图数据库领域，除 Neo4j 外，还有其他如 OrientDB、Giraph、AllegroGraph 等各种图数据库。与所有这些图数据库相比，Neo4j 的优势表现在以下两个方面。

（1）Neo4j 是一个原生图计算引擎，它存储和使用的数据自始至终都是使用原生的图结构数据进行处理的，不像有些图数据库，只是在计算处理时使用了图结构数据，而在存储时还将数据保存在关系型数据库中。

（2）Neo4j 是一个开源的数据库，其开源的社区版吸引了众多第三方的使用和推广，如开源项目 Spring Data Neo4j 就是一个做得很不错的例子，同时也得到了更多开发者的拥趸和支持，聚集了丰富的可供交流和学习的资源与案例。这些支持、推广和大量的使用，反过来会很好地推动 Neo4j 的发展。

1.3.5　综合表现

Neo4j 查询的高性能表现、易于使用的特性及其设计的灵活性和开发的敏捷性，以及坚如磐石般的事务管理特性，都充分说明了使用 Neo4j 是一个不错的选择。有关它的所有优点，总结起来，主要表现在以下几个方面。

（1）闪电般的读/写速度，无与伦比的高性能表现。

（2）非结构化数据存储方式，在数据库设计上具有很大的灵活性。

（3）能很好地适应需求变化，并适合使用敏捷开发方法。

（4）很容易使用，可以用嵌入式、服务器模式、分布式模式等方式来使用数据库。

（5）使用简单框图就可以设计数据模型，方便建模。

（6）图数据的结构特点可以提供更多更优秀的算法设计。

（7）完全支持 ACID 完整的事务管理特性。

（8）提供分布式高可用模式，可以支持大规模的数据增长。

（9）数据库安全可靠，可以实时备份数据，很方便恢复数据。

（10）图的数据结构直观而形象地表现了现实世界的应用场景。

1.4 哪些领域更适合使用 Neo4j

Neo4j 从推出至今，已经经历了几年时间的考验和锤炼，原则上可以应用于任何领域。从大量的应用实践中可以看出，Neo4j 在处理关联数据时更能体现出它的绝对优势。例如，社区网络、推荐引擎、地理（网络）数据、主数据存储、访问控制、欺诈检测、物流管理等，都在实际应用中表现出无与伦比的优势。

1.4.1 社区网络

在一个社区中，人与人之间具有亲属、同事、朋友等各种关系，而每个人又有不同的兴趣、爱好，并且从事各种不同的活动。如果社区庞大，各种关系又纷纭复杂，那么使用一般的关系型数据库或其他 NoSQL 数据库来管理，是很难厘清这些数据及其关系的。如果我们把这之中的人和各种事物及其关系使用节点和关系的形式存储在 Neo4j 的图数据库中并用它来进行数据管理，这不仅是一件极其容易的事情，并且对其中的各种

关系也能做到脉络清楚、有条不紊。而且在 Neo4j 中通过关联数据管理，还可以很容易地发现或发掘这些数据中各种各样的价值。例如，在一个社区网络中，通过每个人的爱好这一关系，人们可以认识更多的朋友，还可以由此组织一些让人更加感兴趣的活动等。

1.4.2　推荐引擎

由社区网络的数据累积和沉淀就可以引申出一个推荐引擎。比如在一个电影社区中，哪些观众在哪里看了哪些电影，并对哪些电影做了哪些评价。有了这些数据之后，我们就能做一个电影社区推荐引擎，让这些数据产生不可估量的商业价值。例如，当前有哪几部电影是被观众所追捧的，而这些电影还有哪些观众没有看过，在这些观众之中，哪些是比较活跃的，哪些是具有朋友关系的，这样就可以将这几部电影对各个群体进行有目的的推荐。这种推荐就是最有效的，对于电影院来说，这无疑是一种商业价值。

1.4.3　交通运输

庞大的交通系统，如航运、海运、铁路运输、公路运输等，任何一种运输系统都将连接世界各地，大到全球、全国，小到地区、城市运输，这里面的运输工具、线路、站点以及班次、调度等各种数据，庞大而又繁杂交错，这些数据使用 Neo4j 来存储和管理会显得脉络清楚、井然有序，因为图的数据结构正好体现了这种数据的特性。使用图的遍历算法能够很快地计算出从一个站点到另一个站点的最短路径。

1.4.4　物流管理

物流管理是交通网络的一个具体应用。对于一个大型的物流系统来说，几秒钟就有可能增加几百个甚至几千个包裹。如果是在一个城市中，那么这一个个包裹将分布于不同区域和街道之中，而要使包裹能够尽快地送达用户手中，在包裹分拣中心要找出每一个包裹配送的最短路径，再将它分配到不同的配送点，这时，Neo4j 就能发挥它的特长了。

1.4.5 主数据管理

对于一些结构化数据，例如客户资料、组织数据、产品数据等，使用 Neo4j 来存储和管理，可以很好地避免数据僵化，让数据具有实时价值，充满活力。不管是自上而下的查询，还是使用遍历，都能保持高效的查询性能。使用 Neo4j 灵活的属性管理还能让数据适应组织及产品结构的变迁和演化。

1.4.6 访问控制

如果有一个大型平台，它有成千上万的用户，并且有成倍的资源，那么对于这些资源的访问，必须要有一个有效的控制系统，以控制哪些用户能够访问哪些特定的资源，或者哪些用户不能访问哪些资源等。使用 Neo4j 来处理这些关联数据及实现访问控制正是它的特长。

1.4.7 欺诈检测

涉及银行卡、信用卡的欺诈交易，或者电信诈骗事项，使用关联数据可以厘清一个账号或一个电话号码的关系和行为，很容易在这些关系和行为之中找出某些异常的举动，从而能很快地检测出欺诈或诈骗行为。所以使用 Neo4j 来建立欺诈检测系统，或者使用关联数据来进行预测、推荐建议等做法都是不错的选择。

其他应用领域，如金融、教育、零售、科学、新闻调查、卫生保健等，Neo4j 都能发挥它的长处。总之，使用 Neo4j 不但可以很好地管理繁杂的关联数据，也能适应大规模的数据增长，并且能连接不同领域的数据，提供最全面和最快的实时反应速度。

1.5 哪些领域不适合使用 Neo4j

从上面的介绍中可以看出，Neo4j 的应用领域涵盖了很广的范围，几乎所有领域都可以使用。但是，世界上没有万能的东西，下面一些领域就不推荐使用 Neo4j。

（1）记录大量基于事件的数据。

（2）需要对大规模分布式数据进行处理（以 PB 级别计算的数据）。

（3）二进制数据存储。

（4）适合保存在关系型数据库中的结构化数据。

1.6　哪些企业在使用 Neo4j

从上面的分析可知，Neo4j 适用于主数据管理、身份识别和访问控制、社交网络、实时推荐引擎、基于图搜索、欺诈检测和网络与 IT 运营、运输和物流管理、连锁零售等领域。那么，当前有哪些知名的企业在使用 Neo4j 呢？要了解这些信息，可以通过下列链接查看 Neo4j 的官方网站介绍：

https://neo4j.com/customers/

打开上面的链接，可以看到正在使用 Neo4j 的一些知名客户及其案例，如图 1-5 所示。

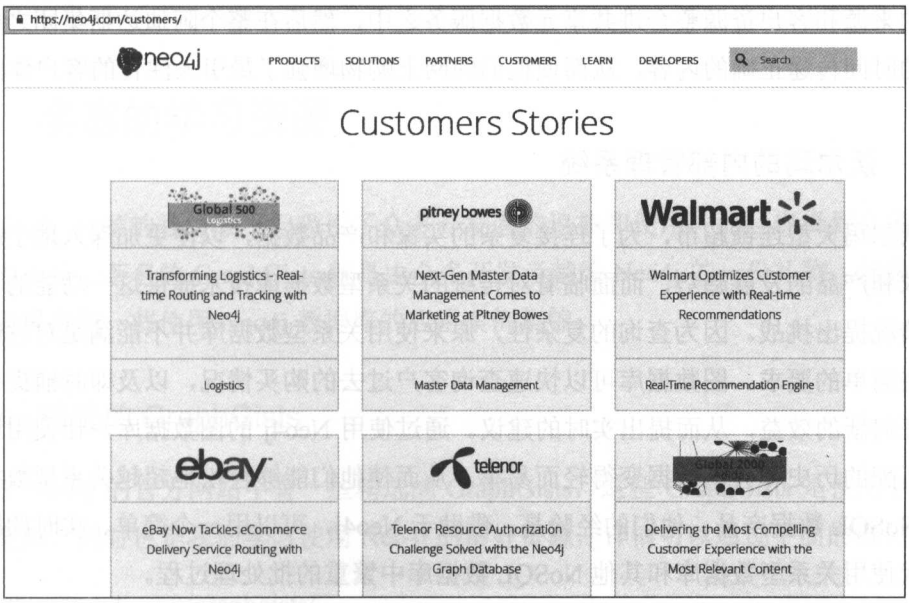


图 1-5　Neo4j 官网中的客户介绍

需要说明的是，这里列举出来的一些企业用户都是使用 Neo4j 企业版的用户，而社区版的使用情况并不包括在这里面。

一些世界知名企业都因为业务的发展，或者为了提高服务的访问性能，而选择了 Neo4j 来替代原来的关系型数据库，并从中获益。例如，阿迪达斯集团、沃尔玛、eBay 等大型企业，又如初创公司 Cobrain、Zephyr Health 和 Wanderu，甚至非盈利的 ICIJ 和世界经济论坛等，都在使用 Neo4j 图数据库。对这些使用 Neo4j 的企业案例，下面择要简单介绍一下。

1.6.1 阿迪达斯的购物网站

阿迪达斯集团，这样一个全球受众大型企业，希望在它的购物网站上能提供更加个性化的购物体验。然而，与许多大型零售商一样，该集团是由各种各样的信息，包括有关产品、市场、社会化媒体、主数据、数字资产、品牌内涵和其他关键领域的数据等组成的，而这些数据分布很零散，无法进行有效的整合。通过使用 Neo4j 的图数据库，终于实现了他们的要求。Neo4j 在这里被证明是理想的技术，它用于建立共享元数据服务，使新的来源和客户资源整合进共享元数据服务之中，然后在整个阿迪达斯集团之中，以正确的时间传递正确的内容，从而使他们的网上购物增强了最引人注目的客户体验。

1.6.2 沃尔玛的内部管理系统

沃尔玛大型连锁超市，为了连接复杂的买家和产品数据，以便更加深入地了解客户的需求和产品的发展趋势，而面临着对传统的关系型数据库技术提供这一功能的不容乐观的情况提出挑战。因为查询的复杂性，原来使用关系型数据库并不能满足对性能，甚至一些简单的要求。图数据库可以快速查询客户过去的购买情况，以及即时捕捉在线客户中任何新的效益，从而提出实时的建议。通过使用 Neo4j 的图数据库，让使用原来的方式匹配的历史和会话数据变得轻而易举，从而使他们能够轻松地超越关系型数据库和其他 NoSQL 数据产品。他们的经验是，借助于 Neo4j，可以用一个简单、实时的图数据库替代使用关系型数据库和其他 NoSQL 数据库中繁重的批处理过程。

1.6.3　eBay 的电子商务

总部位于伦敦的 eBay，它的使命一直是给人电子商务采购以最快的速度传递，为了支持数据和新功能的爆发性增长，送货服务平台需要进行改造。由于过去使用 MySQL，复杂的代码和缓慢的查询使他们无法忍受。选择使用 Neo4j 之后，由于它的设计灵活性、高效的查询性能和简单易用的特性，加快了开发的速度，克服了以前解决方案的速度和可扩展性限制。以至它的技术负责人最后说："我们的 Neo4j 解决方案比以前的 MySQL 解决方案快上千倍，而查询需要的代码量却至少减少了 10～100 倍。同时，Neo4j 还允许我们添加以前不可能实现的功能。"

另外，还有用于网络管理的思科、使用社交网络的 LinkedIn（领英）、使用身份验证和访问控制的 Telenor（挪威电信）、使用欺诈检测的 ICIJ 等，这里就不再一一列举说明了。

总之，在网络、电信、电子商务、信息服务、大型超市、物流、运动、社区、玩具行业等领域，都有来自不同国家和地区的大小公司和企业在使用 Neo4j 数据库，并且从中获益。

1.7　丰富的学习资源

Neo4j 开源的社区版不但吸引了众多开发者的追捧和使用，而且也得到众多开发者的鼎力支持。著名的 GraphGists 就是由众多开发者献给 Neo4j 的一份礼物，这是由众多开发者提供的一些使用 Neo4j 数据库的应用案例集锦。

1.7.1　精选的 GraphGists

在 Neo4j 的官方网站中有一些精选的 GraphGists，这些 GraphGists 是使用 Neo4j 的精彩案例，同时也是我们学习使用 Neo4j 的最好资源，详情可以通过下面的链接访问：

https://neo4j.com/graphgists/

打开上面的链接，可以看到一些精选的 GraphGists，如图 1-6 所示。

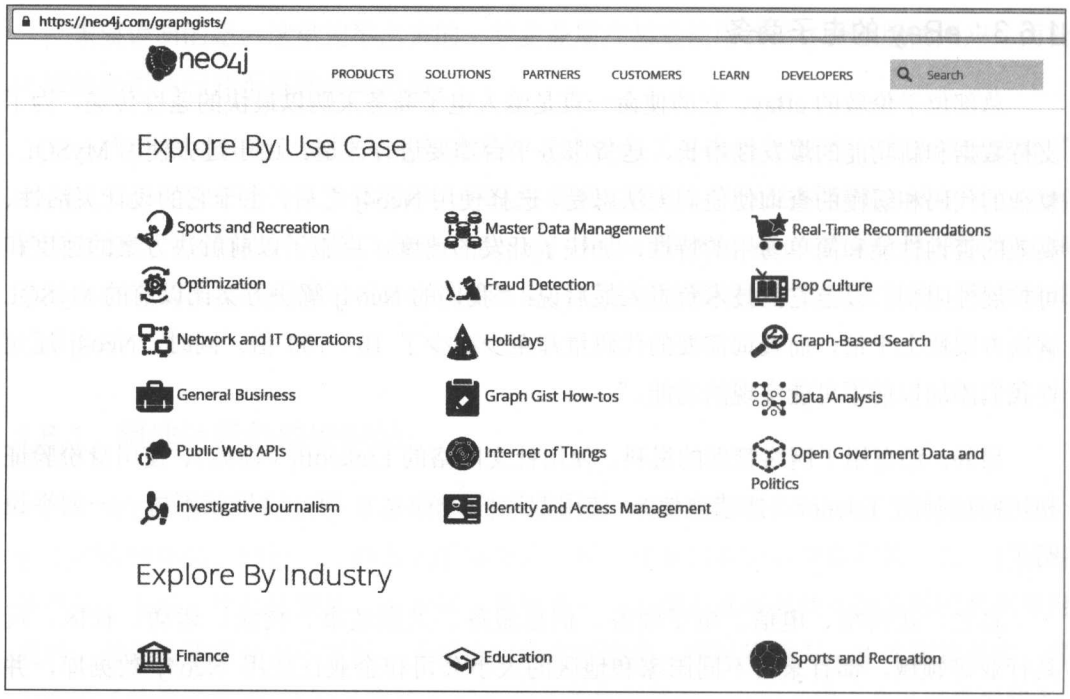


图 1-6 Neo4j 官网中的 GraphGists

这些案例琳琅满目，各行各业应有尽有，并且也都做了分类。

按行业分有金融、教育、运动和休闲、计算机科学与编程、零售、保健、制造业、运输和物流、社区网络等。

按用例分有主数据管理、实时建议、欺诈检测、流行文化、网络和 IT 运营、基于图搜索、数据分析、新闻调查、身份和访问控制管理等。

还有一些特色用例，在各自的应用领域具有非常出色的表现。

通过这些案例，我们可以学习如何在各个领域中利用关联数据进行建模，并利用各种独特的算法来设计查询，从而最好地表现数据的商业价值。

这些案例都连接了真实的数据库，并且都具有一些测试数据，非常真实地展示了设计的效果。如果你对其中的查询有自己的想法，则可以直接在界面上使用 Cypher 查询语言执行查询操作。

例如，我们打开一个 GraphGists，它呈现出如图 1-7 所示的界面。从这里可以看出，它的测试数据已经生成，并且一些查询设计也已经显示出查询结果。

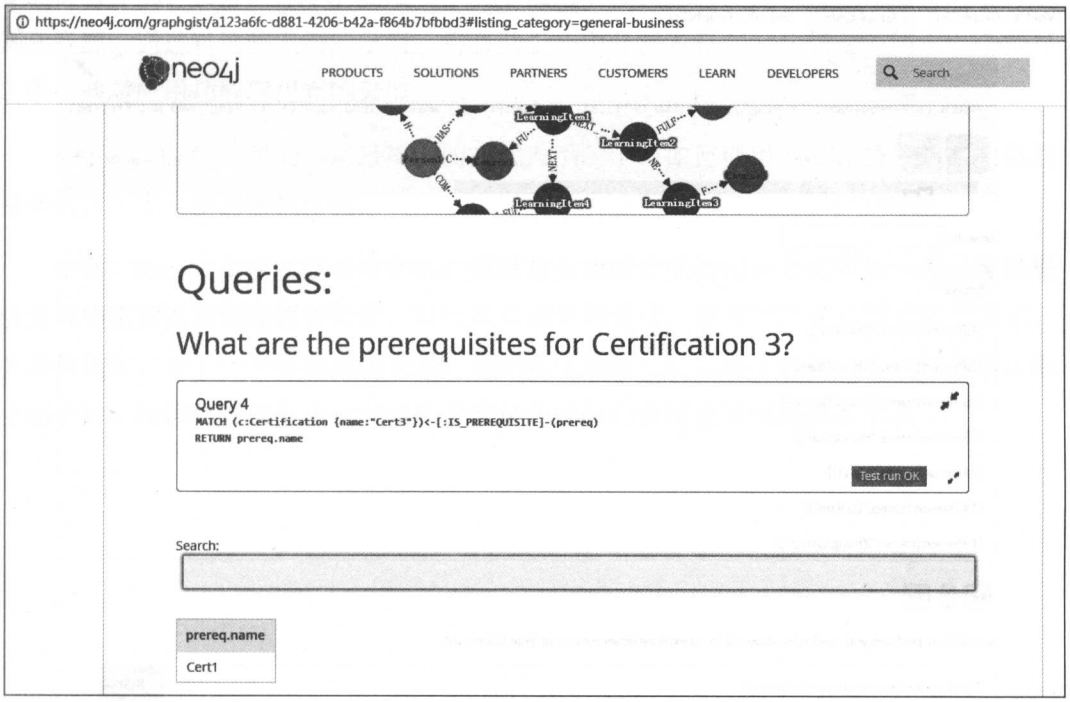


图 1-7　一个 GraphGists 实例

1.7.2　GraphGists 门户

除 Neo4j 的官网外，GraphGists 还有一个专门的门户网站：http://portal.graphgist. org/，这里有更多的使用 Neo4j 的案例，这些案例由全球开发者发布提供。

如果你有兴趣，则可以在上面发布你的 GraphGists。至于如何发布，这个网站也有介绍和说明。GraphGists 只是一个简单的 AsciiDoc 文本文件，通过连接托管在 GitHub 上的 Gist 或任何其他有效的 URL 进行访问。如本书第 7 章的电影社区推荐引擎的实例也发布在这个门户网站上，使用下面的链接就可以打开这个 GraphGists：

http://portal.graphgist.org/graph_gists/movie-recommended-community

如图 1-8 所示是打开这个案例的显示效果。

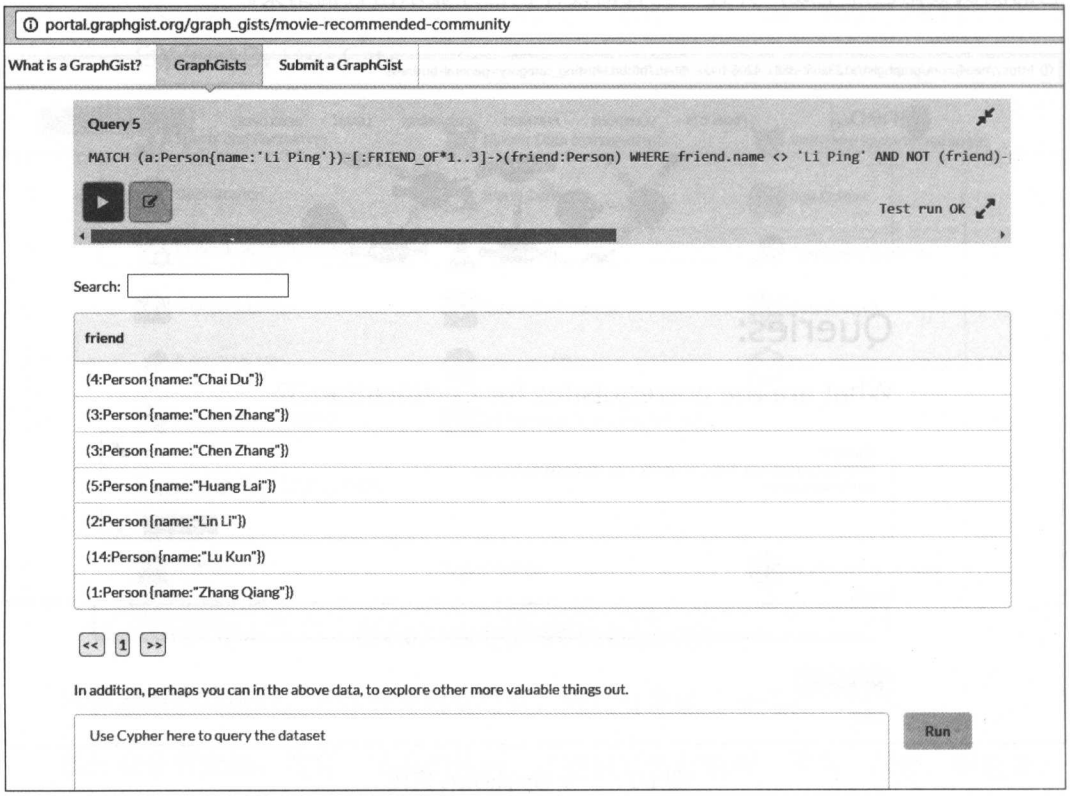


图 1-8　电影社区推荐引擎的 GraphGists

对于任何一个 GraphGists，你不但可以打开进行在线学习，还可以直接使用查询语言进行查询设计和体验。

1.8　小结

本章介绍了 Neo4j 是一个什么样的数据库，并说明了这个图数据库的一些特点和优势，以及在哪些领域使用这种数据库更能充分发挥它的优势。同时介绍了当前有哪些知名企业正在使用这种数据库，并从使用中获益和提升了应用的访问性能，使相应业务有了前所未有的进展。

通过上面的介绍可知，使用 Neo4j 不但能有效解决因为使用关系型数据库所处的困境，并且能获得高效的性能优势，而且还能使用一些新的特性，从而让应用项目获得更多的价值体现。另外，Neo4j 还是一个易于使用的数据库，并且具有非常丰富的学习案例和资源，通过对这些案例的学习，将有助于一个开发者将 Neo4j 快速应用于一个项目之中，并对应用功能取得新的突破。

从下一章开始，我们将从最简单的方式入手，学习如何使用 Neo4j 数据库，相信这将带给你一个全新的愉悦体验。

注意，Neo4j 企业版是要收费的，而只有企业版才能使用分布式安装，并且可以提供负载均衡和高可用配置等功能。社区版只能单机使用，最多可以使用数十亿个节点、关系和属性。对于一个小型应用来说，使用社区版就已经足够了，而当节点数量超过数十亿个时，相信可观的商业收益相比于支付企业版的费用也就变得微不足道了。

第2章
Neo4j API应用

Neo4j是用Java开发的，所以对于一个Java开发者来说，是近水楼台先得月，可以直接使用Neo4j API。开始使用Neo4j数据库，我们就从嵌入式的Neo4j开始吧，因为它不用安装服务器，只要引用Neo4j的开发包就能正常使用Neo4j了。

Neo4j API提供了原生的使用Neo4j的方法，不但可以执行一般的CRUD（Create，Retrieve，Update，Delete），即我们常说的增、删、改、查等操作，而且还能更好地设计遍历算法，以充分保证Neo4j的高效查询性能。只是，从易用性上来说，使用Neo4j API略显烦琐。在后面的章节中我们将着重介绍怎么使用SDN（Spring Data Neo4j）来访问Neo4j数据库，因为它提供了更加易于使用的规范建模、持久化设计和整套的接口方法，可以让使用Neo4j变得更加简单。

作为认识和理解Neo4j的入门，本章对Neo4j API的使用只是做一个简单的介绍，并不对其做太多的深入探讨和展开分析。实际上，不用了解Neo4j API，也能从更高的层次上（如上面提到的SDN）很好地使用Neo4j。

2.1 创建项目工程

在使用Neo4j进行开发之前，我们要创建一个项目工程，以方便项目的管理。对于Java开发者来说，推荐使用Spring Boot开发框架来创建工程，这是一个非常不错的选择，因为易于使用和高效的开发框架可以减少项目中一些烦琐的配置过程及其基础工作。同

时也极力推荐使用 Maven 作为项目管理的工具，这也能为项目管理，包括一些依赖的引用、项目打包和运行测试等，提供更加方便和友好的使用方法，从而简化和省略很多工作，并且这种做法与全球绝大多数开发者的步伐是一致的。

2.1.1　项目工程配置

使用 Spring Boot 开发框架可以很方便地在 Maven 管理中配置一个工程的一些依赖引用。Maven 使用项目对象模型（Project Object Model）来管理一个工程的配置，在开启一个新工程之后，将在工程的根目录中拥有一个配置文件：pom.xml。使用这个配置文件，可以很方便地进行依赖配置管理。

如代码清单 2-1 所示，我们使用了 JDK1.8 和 Spring Boot 1.4.2 的依赖配置。这样，这个项目工程就已经具有一般的开发所需要的依赖包了。在后续章节的一些实例中，也将使用这种基本的依赖配置。

代码清单 2-1　项目工程的依赖配置

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>1.8</java.version>
</properties>

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.2.RELEASE</version>
    <relativePath/>
</parent>
```

2.1.2　引用 Neo4j 开发包

除工程的开发环境配置外，现在我们要使用 Neo4j，还需要引用 Neo4j 的开发包。如代码清单 2-2 所示，在工程或模块的 Maven 管理中引用了 Neo4j 的开发包。现在就可以使用 Neo4j API 开始进行体验使用 Neo4j 数据库的历程了。

代码清单 2-2 引用 Neo4j 开发包依赖配置

```
<dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j</artifactId>
    <version>3.0.6</version>
</dependency>
```

2.2 使用 Neo4j API

使用 Neo4j API 的方式来访问 Neo4j 数据库是很简单的。需要注意的是，在进行任何有关数据库的存取操作过程中，都必须开启事务管理，因为 Neo4j 的数据存取都是在严格的事务管理中进行的。

2.2.1 使用嵌入式数据库

使用嵌入式的 Neo4j，只要在程序中打开一个文件目录，就可以用它作为 Neo4j 的数据库。如代码清单 2-3 所示，通过使用文件方式打开 Neo4j 数据库。在程序运行时，将在工程的 target 目录中创建数据库文件目录 neo4j-db，并生成一些数据文件。

代码清单 2-3 创建嵌入式数据库

```
private static final File DB_PATH = new File( "target/neo4j-db" );
private static GraphDatabaseService graphDb;
public Knows() throws IOException{
    //FileUtils.deleteRecursively( DB_PATH );
    graphDb = new GraphDatabaseFactory().newEmbeddedDatabase( DB_PATH );
    registerShutdownHook( graphDb );
}
```

在使用 Neo4j API 时，我们必须养成一个良好的习惯，即在完成数据库使用，将要退出程序时，不要忘记关闭数据库。因为是在使用 API，所以一切与数据库有关的行为都是由我们自己控制的。

如代码清单 2-4 所示，注册了一个 ShutdownHook 来实现数据库的关闭动作。因为

在执行了数据库的关闭指令之后，数据库并不能立即关闭，它还需要做一些清场的工作，所以必须使用一个线程来运行数据库的 shutdown()方法，才能完成数据库的关闭任务。

代码清单 2-4　关闭数据库

```
private static void registerShutdownHook( final GraphDatabaseService graphDb )
{
    Runtime.getRuntime().addShutdownHook( new Thread()
    {
        @Override
        public void run()
        {
            graphDb.shutdown();
        }
    } );
}
```

2.2.2　创建节点和关系

使用上面创建的嵌入式数据库，我们就可以体验创建节点和关系的操作了。

如代码清单 2-5 所示，是一个创建节点和关系的非常简单的实例。

这里给节点使用了标签 Person，这就相当于对节点进行了分组，可以方便以后节点的查询和管理，以及提高查询的性能。当然，如果不想使用标签也是可以的，只是查询起来就不那么方便了。

与 RDBMS 相比，标签就相当于表名，而一个节点相当于一行的数据，节点的属性相当于列的字段。与 RDBMS 不同的是，一个节点可以有多个标签。

代码清单 2-5　使用标签创建节点并为节点建立关系

```
try ( Transaction tx = graphDb.beginTx() )
{
    Label label = Label.label("Person");
    Node first = graphDb.createNode(label);
    first.setProperty("name", "first");
```

```
    Node second = graphDb.createNode(label);
    second.setProperty( "name", "second" );

    Relationship relationship = first.createRelationshipTo(second,
RelTypes.KNOWS);

    System.out.println("create node name is " + first.getProperty("name"));
    System.out.println("create node name is " + second.getProperty("name"));
    System.out.println("create relationship type is " +
relationship.getType());

    tx.success();
}
```

在这个程序中，我们创建了两个节点，并为每个节点都创建了一个 name 属性，且将属性值分别设置为 first 和 second，同时将它们与使用类型为 KNOWS 的关系连接起来。

运行上面的程序，将会输出如下结果：

```
create node name is first
create node name is second
create relationship type is KNOWS
```

即创建了两个节点，分别为 first 和 second，并为它们建立了 KNOWS 关系。这个数据在数据库中将表现为如图 2-1 所示的样子。

图 2-1　简单图数据

2.2.3　查询及更新

使用如代码清单 2-6 所示的方法，可以在数据库中查询标签为 Person、属性 name 的值为 second 的节点。因为我们使用了标签，所以可以很方便地使用 findNode()方法进行查询。在查询出节点之后，就可以对节点进行编辑了。例如，在这里我们更改节点的属性 name 的值为 My name，同时增加一个属性 sex 来代表性别，并将属性值设置为“男”。

最后使用 tx.success()方法提交事务，这个修改才能生效。

代码清单 2-6　使用属性查询节点并对节点进行更新

```
try ( Transaction tx = graphDb.beginTx() )
{
    Label label = Label.label("Person");
    Node node = graphDb.findNode(label, "name", "second");

    System.out.println("query node name is " + node.getProperty("name"));

    node.setProperty("name", "My name");
    node.setProperty("sex", "男");

    System.out.println("edit node name is " + node.getProperty("name"));
    System.out.println("add property sex is " + node.getProperty("sex"));

    tx.success();
}
```

2.2.4　删除关系和节点

如果要删除数据，则执行相关实体的 delete()方法即可。在删除数据时，要注意删除的执行规则：如果存在关系，则必须先删除关系，再删除节点，或者同时删除；如果不存在关系，则删除节点的操作将不能被成功执行。

如代码清单 2-7 所示，程序首先使用 findNode()方法查询出节点 first，然后通过这个节点查询出它的 KNOWS 关系，最后删除这个关系，同时删除这个关系所连接的两个节点。

代码清单 2-7　查询出节点之后删除节点及关系

```
try ( Transaction tx = graphDb.beginTx() )
{
    Label label = Label.label("Person");
    Node first = graphDb.findNode(label, "name", "first");
    Relationship relationship = first.getSingleRelationship(RelTypes.KNOWS,
Direction.OUTGOING);
```

```
    System.out.println("delete node name is " + first.getProperty("name")
            + ", relationship is KNOWS, end node name is "
            + relationship.getEndNode().getProperty("name"));

    relationship.delete();
    relationship.getEndNode().delete();
    first.delete();

    tx.success();
}
```

上面实例的完整源程序可以参看 https://github.com/mr-csj/neo4j-useapi/blob/master/base/src/main/java/com/demo/Knows.java。

或者从 GitHub 中通过下面的链接检出本章实例的整个工程到本地：

https://github.com/mr-csj/neo4j-useapi.git

然后通过配置 Application，选择运行的程序，即可在本地执行程序进行测试。如果你使用的开发工具是 IntelliJ IDEA，则有关 Application 的配置可以参考如图 2-2 所示的配置。

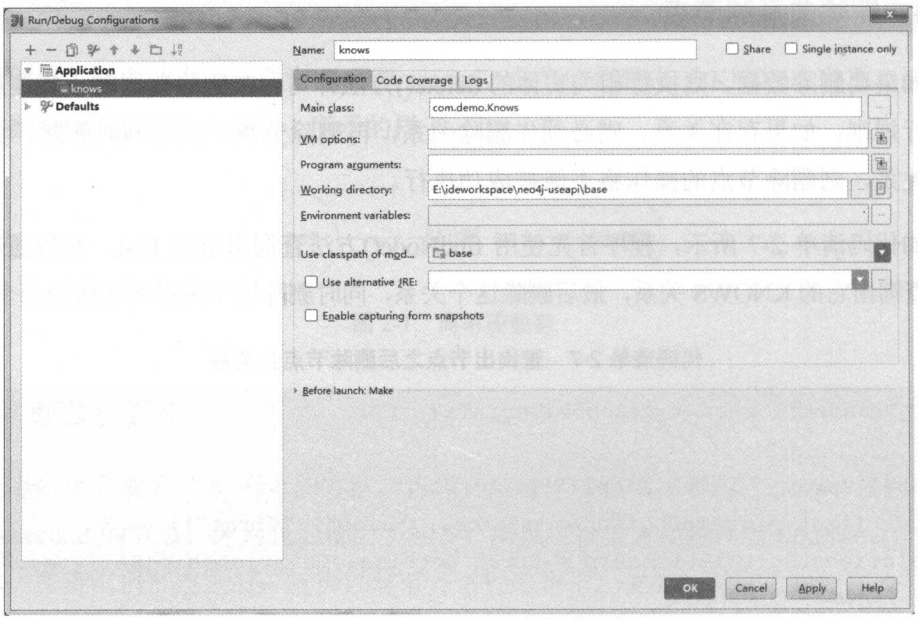


图 2-2 Application 配置

至此，我们完成了使用 Neo4j API 在嵌入式数据库中对节点和关系进行增、删、改、查的一般操作，并且初步认识了标签的使用，以及使用标签的必要性和重要性。

2.3　使用标签

通过前面的练习，我们已经认识到使用标签极其重要。标签不仅仅是对节点进行分门别类的标识，更为重要的是能对所有节点进行分组，从而更好地管理节点，并且提高了节点的查询性能。

例如，我们有两个节点，它们都具有 name 属性，其中一个节点的 name 属性的值为“小张”，另一个节点的 name 属性的值为“铁拳”。现在我们并不知道它们是什么类型，也不清楚这两个节点所表示的是什么事物。而且如果要查询这两个节点，就必须在整个数据库中进行搜索。下面我们为“小张”这个节点贴上一个标签 Person，用来表示观众，并为“铁拳”这个节点贴上标签 Movie，用来表示电影，如图 2-3 所示。

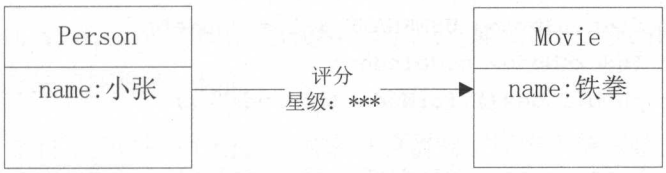


图 2-3　贴上标签的节点实例

这样一来，不但可以对这两个节点进行分类，也清楚了它们所表示的事物，管理起来十分方便，而查找的范围也大幅度缩小了。

既然为节点加上标签有这么多好处，所以即使 Neo4j 并没有强制创建节点时必须加上标签，但是根据实际使用经验，还是强烈建议为每个节点都加上标签。以后我们使用 SDN（Spring Data Neo4j）来建模时，SDN 将默认以类名为节点创建标签。

另外，一个节点也可以贴上多个标签，至于是否需要使用多标签，则主要依据实际使用情况来决定。

2.4 使用索引

如果在进行节点查询时使用了节点的属性作为查询条件，那么一种有效提高查询性能的方法就是为节点添加索引。这一点与使用 RDBMS 相同。

在 Neo4j 中使用索引有很多种方法，并且都是相当方便的。

2.4.1 手动索引

所谓手动索引，是指在创建节点时才能增加的索引。如代码清单 2-8 所示是一个创建手动索引的实例程序。程序中通过使用 Index，在创建节点并设定属性的时候，将属性增加为索引。

代码清单 2-8 创建手动索引

```
private static final File DB_PATH = new File( "target/neo4j-store" );
private static GraphDatabaseService graphDb;
private static final String USERNAME_KEY = "name";
private static Index<Node> nodeIndex;
nodeIndex = graphDb.index().forNodes( "nodes" );
......
private static Node createAndIndexUser( final String username )
{
    Node node = graphDb.createNode();
    node.setProperty( USERNAME_KEY, username );
    nodeIndex.add( node, USERNAME_KEY, username );
    return node;
}
```

创建手动索引之后，就可以使用这个索引来设计查询了，如代码清单 2-9 所示。程序中使用属性的值，通过 Index 查找一个节点。

代码清单 2-9 使用索引进行查询设计

```
String userName = "onename";
Node foundUser = nodeIndex.get( USERNAME_KEY, userName ).getSingle();
```

除了使用完全匹配的属性值执行一般的查询外，也可以使用一些匹配符号执行模糊查询，如代码清单 2-10 所示。这个程序通过使用通配符“*”查找出所有的节点，然后使用 Index 的 remove()方法删除索引，并使用 Node 的 delete()方法删除节点。

代码清单 2-10　使用索引进行模糊查询及删除数据

```
for ( Node user : nodeIndex.query( USERNAME_KEY, "*" ) )
{
    nodeIndex.remove(  user, USERNAME_KEY, user.getProperty( USERNAME_KEY ) );
    user.delete();
}
```

上面为了方便功能的说明，只列出了代码的片断，这个实例的完整代码可通过下面的链接查看：https://github.com/mr-csj/neo4j-useapi/blob/master/base/src/main/java/com/demo/ManualIndex.java。

2.4.2　模式索引

对于熟悉 RDBMS 索引的开发者来说，上面实例中使用的手动索引显然不太方便。相对于手动索引而言，在 Neo4j 中也可以使用自动索引，即使用模式索引，它与在 RDBMS 中使用的索引的方法和性质是一样的。

如代码清单 2-11 所示是一个创建模式索引的实例。程序中使用数据库模式 Schema 在标签为 User 的节点中为属性 name 创建模式索引，这样在 findNode()或 findNodes()等方法中就可以自动使用索引了。

代码清单 2-11　创建模式索引

```
private static void createIndex(GraphDatabaseService graphDb){
    try ( Transaction tx = graphDb.beginTx() )
    {
        Schema schema = graphDb.schema();
        schema.indexFor(Label.label("User")).on("name").create();
        tx.success();
    }
}
```

如果要删除模式索引，则可以使用如代码清单 2-12 所示的方法，即使用 IndexDefinition 的 drop()方法进行删除。

代码清单 2-12　删除模式索引

```
try ( Transaction tx = graphDb.beginTx() )
{
    Label label = Label.label( "User" );
    for(IndexDefinition indexDefinition :
graphDb.schema().getIndexes(label)){
        indexDefinition.drop();
    }
    tx.success();
}
```

Neo4j 除了有模式索引，还可以使用模式约束。因为使用模式索引还允许属性有重复值，如果要设定属性的唯一索引，则可以使用模式约束来实现。

2.4.3　模式约束

创建模式约束的方法与创建模式索引的方法有点相似，都是通过数据库模式 Schema 来创建的。如代码清单 2-13 所示是一个创建模式约束的实例。程序中使用 assertPropertyIsUnique()方法来设定属性值的唯一性。在创建了一个属性的模式约束之后，将会同时创建一个模式索引。

代码清单 2-13　创建模式约束并指定唯一属性

```
try ( Transaction tx = graphDb.beginTx() )
{
    Schema schema = graphDb.schema();

schema.constraintFor(Label.label( "User" )).assertPropertyIsUnique("name").
create();
    tx.success();
}
```

如果要删除模式约束，则可以使用如代码清单 2-14 所示的方法。这与删除模式索引

的方法有点相似，即使用 ConstraintDefinition 的 drop()方法来删除模式约束。

代码清单 2-14　删除模式约束

```
try ( Transaction tx = graphDb.beginTx() )
{
   Label label = Label.label( "User" );
   for(ConstraintDefinition constraintDefinition :
graphDb.schema().getConstraints(label)){
      constraintDefinition.drop();

   }
   tx.success();
}
```

下面我们使用一个测试用例来验证一下使用模式约束的情况。

首先，使用如代码清单 2-12 所示的方法，在标签为 User 的用户节点中，为 name 属性创建一个唯一约束。然后使用如代码清单 2-15 所示的方法创建几个用户节点。这里，我们使用一个循环创建了三个用户节点，并将用户的 name 属性分别设置为 user0、user1 和 user2。

代码清单 2-15　创建用户节点

```
private static void createUser(GraphDatabaseService graphDb){
   try ( Transaction tx = graphDb.beginTx() )
   {
      Label label = Label.label( "User" );
      for ( int id = 0; id < 3; id++ )
      {
         Node node = graphDb.createNode( label );
         node.setProperty( "name", "user" + id);
         System.out.println( "Create user id=[" + node.getId() + "] name=[" +
node.getProperty("name") + "]" );
      }
      tx.success();
   }
}
```

运行上面的测试程序，可以输出如下结果：

```
Create user id=[0] name=[user0]
Create user id=[1] name=[user1]
Create user id=[2] name=[user2]
```

接下来我们设计一个修改程序，如代码清单 2-16 所示。这个程序根据传入的参数来修改用户节点的 name 属性，如果修改成功则输出修改前后的属性值，否则输出异常的错误信息。

代码清单 2-16　修改用户节点属性

```
private static void updateUser(GraphDatabaseService graphDb, int id){
    String outstr = "";
    try ( Transaction tx = graphDb.beginTx() )
    {
        Label label = Label.label( "User" );
        String nameToFind = "user" + id;
        for ( Node node : loop(graphDb.findNodes(label, "name", nameToFind)) )
        {
            node.setProperty( "name", "user" + ( id + 1 ) );
            outstr += "Update user name from [" + nameToFind + "] to [" +
node.getProperty("name") + "]";
        }
        tx.success();
    }catch (Exception e){
        outstr = "Update error: " + e.getMessage();
    }
    System.out.println(outstr);
}
```

如果我们使用下述方法执行修改：

```
updateUser(graphDb, 0);
updateUser(graphDb, 1);
updateUser(graphDb, 2);
```

即分别尝试将 user0 修改成 user1，将 user1 修改成 user2，将 user2 修改成 user3，则运行程序将输出如下结果：

```
Update error: Node 1 already exists with label User and property "name"=[user1]
Update error: Node 2 already exists with label User and property "name"=[user2]
Update user name from [user2] to [user3]
```

从运行结果中可以看出，前面两个修改都因为属性中有重复值而出现了错误，只有最后一个修改是成功的。

以上实例的完整程序请参照 https://github.com/mr-csj/neo4j-useapi/blob/master/base/src/main/java/com/demo/Indexs.java。

2.5　图的遍历

Neo4j 数据库的高性能查询表现就是根据图数据结构的自然伸展特性，使用免索引邻近查询算法，即图的遍历来实现的。图的遍历是图数据结构所具有的独特算法。下面我们通过一个简单的实例来见证一下这一算法的威力。

现在，假如我们拥有如图 2-4 所示的这些关联数据，我们就可以按照其关联关系，设计出各种遍历算法来执行数据查询。根据遍历时查找数据的路径不同，遍历算法可以分为广度优先遍历和深度优先遍历。

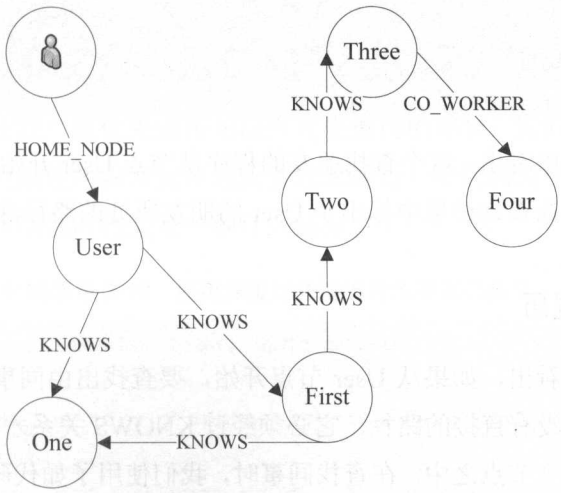


图 2-4　使用遍历算法的测试数据

2.5.1 广度优先遍历

如代码清单 2-17 所示，这是一个通过 KNOWS 关系来查找朋友的遍历算法设计。代码中使用 breadthFirst()设定了使用广度优先遍历算法，关系函数 relationships()设定了关系的类型为 KNOWS，同时设定查找的方向为单向，并从查找节点出发，通过 Evaluators.excludeStartPosition()设定了评估函数 evaluator()从开始节点进行查找。执行程序时，将从查找节点开始，找出由 KNOWS 关系连接的所有节点。

代码清单 2-17 查找朋友的遍历算法

```
private Traverser getFriends( final Node person )
{
    TraversalDescription td = graphDb.traversalDescription()
            .breadthFirst()
            .relationships( RelTypes.KNOWS, Direction.OUTGOING )
            .evaluator( Evaluators.excludeStartPosition() );
    return td.traverse( person );
}
```

运行上面的测试程序，将输出如下结果：

```
User's friends:
At depth 1 => First
At depth 1 => One
At depth 2 => Two
At depth 3 => Three
Number of friends found: 4
```

从运行结果中可以看出，这个查找朋友的程序从节点 User 开始，查找出了它的所有朋友。这里我们特意在查询结果中标出了 User 的朋友所处的路径深度。

2.5.2 深度优先遍历

从图 2-4 中可以看出，如果从 User 节点开始，要查找出由同事关系 CO_WORKER 所连接的节点，则将没有直接的路径，它必须经过 KNOWS 关系才能找到，并且这个同事在最右边的最后一个节点之中。在查找同事时，我们使用了如代码清单 2-18 所示的遍历算法，即使用了两个关系函数 relationships()来设计查找的路径，并在评估函数

evaluator()中设定查找最终节点的 CO_WORKER 关系，所以使用这个查找算法可以查找出朋友中最后节点的同事。这里我们通过 depthFirst()设定了使用深度优先遍历算法。

代码清单 2-18　查找同事的遍历算法

```
private Traverser findWorker( final Node startNode )
{
    TraversalDescription td = graphDb.traversalDescription()
            .depthFirst()
            .relationships( RelTypes.CO_WORKER, Direction.OUTGOING )
            .relationships( RelTypes.KNOWS, Direction.OUTGOING )
            .evaluator(Evaluators.includeWhereLastRelationshipTypeIs(RelTypes.
CO_WORKER ) );
    return td.traverse( startNode );
}
```

对于上面这个测试数据来说，其实要查找同事，不管是使用广度优先遍历，还是使用深度优先遍历，其查找结果并没有什么不同，但查找的方法还是有所不同的。如果使用广度优先遍历，则会首先查找第一条路径 User→One，然后再查找第二条路径 User→First→Two→Three→Four。如果使用深度优先遍历，则只要查找第二条路径就可以了。

执行上面的测试程序，将打印出如下结果：

```
co-worker:
At depth 4 => Four
Number of co-worker found: 1
```

为了更加清楚地比较广度优先遍历与深度优先遍历的差异，我们将代码清单 2-17 中查找朋友的算法设计改成使用深度优先遍历算法来执行，如代码清单 2-19 所示。现在我们再来尝试一下在查找朋友时将会是一种什么结果。

代码清单 2-19　使用深度优先遍历查找朋友的算法

```
private Traverser getFriends( final Node person )
{
    TraversalDescription td = graphDb.traversalDescription()
            .depthFirst()
            .relationships( RelTypes.KNOWS, Direction.OUTGOING )
            .evaluator( Evaluators.excludeStartPosition() );
```

```
    return td.traverse( person );
}
```

使用深度优先遍历算法，查找朋友的结果如下：

```
User's friends:
At depth 1 => First
At depth 2 => Two
At depth 3 => Three
At depth 2 => One
Number of friends found: 4
```

这个结果跟使用广度优先遍历不同的是，One 在最后才被找到。即首先从 User→First→Two→Three 这条路径进行遍历，再从 User→One 这条路径进行遍历。

具体在应用中应该使用哪种算法，可根据实际情况来决定。例如，对于上述测试实例来说，如果要查找 One，则可以使用广度优先遍历；如果要查找 Three，则使用深度优先遍历会快一点；而如果要查找 First，则使用任何一种遍历算法的结果都是一样的。

上面实例的完整程序请参照 https://github.com/mr-csj/neo4j-useapi/blob/master/traversal/src/main/java/com/test/Friends.java。

2.5.3 遍历的路径

当遍历节点经过多条路径时，为了优化遍历的算法，可以设定遍历执行的顺序路径。例如，现在有如图 2-5 所示的数据，我们来看看怎样通过设定顺序路径来优化遍历的算法。

首先创建节点和关系，如代码清单 2-20 所示。即按如图 2-5 所示的形式，创建 5 个节点，并在各个节点之间创建一些关系。

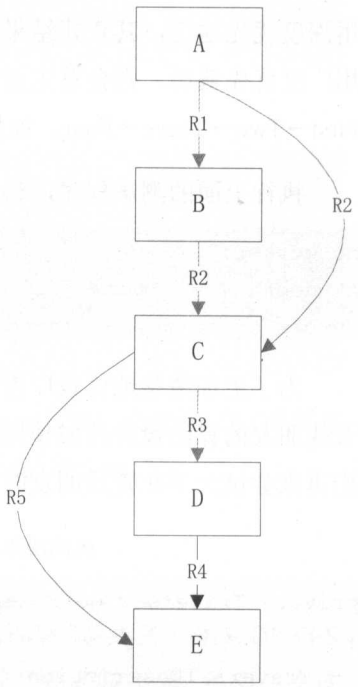


图 2-5 遍历的顺序路径示例数据

代码清单 2-20　创建顺序路径的节点和关系

```
public Node createTheGraph()
{
    try ( Transaction tx = db.beginTx() )
    {

        Node A = db.createNode();
        Node B = db.createNode();
        Node C = db.createNode();
        Node D = db.createNode();
        Node E = db.createNode();

        A.setProperty( "name", "A" );
        B.setProperty( "name", "B" );
        C.setProperty( "name", "C" );
        D.setProperty( "name", "D" );
        E.setProperty("name", "E");

        A.createRelationshipTo( B, R1 );
        B.createRelationshipTo( C, R2 );
        C.createRelationshipTo( D, R3 );
        A.createRelationshipTo( C, R2 );
        C.createRelationshipTo( E, R5 );
        D.createRelationshipTo(E, R4);

        tx.success();
        return A;
    }
}
```

现在，如果把遍历的顺序路径设定为 R1→R2→R3，则将返回节点 A,B,C,D。这个设定如代码清单 2-21 所示。程序中使用列表 ArrayList 指定了遍历的顺序路径，而在遍历的评估函数 Evaluation()的设置中使用 Uniqueness.NODE_PATH 设定了访问节点的路径唯一性。

代码清单 2-21 按顺序路径遍历

```
public TraversalDescription findPaths()
{
    final ArrayList<RelationshipType> orderPaths = new
ArrayList<RelationshipType>();
    orderPaths.add(R1);
    orderPaths.add(R2);
    orderPaths.add(R3);
    TraversalDescription td = db.traversalDescription().evaluator( new
Evaluator(){
            @Override
            public Evaluation evaluate( final Path path )
            {
                if ( path.length() == 0 )
                {
                    return Evaluation.EXCLUDE_AND_CONTINUE;
                }
                RelationshipType expectedType =
orderPaths.get( path.length() - 1 );
                boolean isExpectedType = path.lastRelationship()
                        .isType( expectedType );
                boolean included = path.length() == orderPaths.size() &&
isExpectedType;
                boolean continued = path.length() < orderPaths.size() &&
isExpectedType;
                return Evaluation.of( included, continued );
            }
        } )
        .uniqueness( Uniqueness.NODE_PATH );
    return td;
}
```

运行上面的测试程序，将看到如下输出结果：

```
(A)--[R1]-->(B)--[R2]-->(C)--[R3]-->(D)
```

这个结果与我们预期的结果是一样的。如果将遍历的顺序路径设定为 R2→R5，则

将只返回节点 A,C,E。也就是说，通过设定遍历的顺序路径，我们优化了遍历的算法，不像前面查找朋友那样，需要对整幅图进行遍历。

一种更加高效的遍历算法是使用最短路径的遍历算法。与使用顺序路径算法的单方向查找不同，最短路径一般从两个节点开始进行双向查找，即同时从两个方向出发向中间靠拢。有关最短路径的使用这里就不再举例了，在后续的章节中，我们将使用 Cypher 查询语言来设计使用最短路径的查询算法。

上面测试实例的完整程序请参照 https://github.com/mr-csj/neo4j-useapi/blob/master/traversal/src/main/java/com/test/OrderedPath.java。

2.6　使用 Cypher 查询语言

像关系型数据库使用 SQL 查询语言一样，Neo4j 也可以使用 CQL（Cypher Query Language），即 Cypher 查询语言来存取数据。实际应用中的查询设计，更多的是使用 Cypher 查询语言来实现的。因为 Cypher 查询语言的语法更接近于业务逻辑的内涵，所以也能更好地理解和维护。而且使用 Cypher 查询语言，也像使用 Neo4j API 一样，可以对数据库执行增、删、改、查等所有操作，并且实现起来更加方便和简单。

下面我们使用 Cypher 查询语言来设计一个分页查询，如代码清单 2-22 所示。

程序中使用 Cypher 查询语言中的关键字 SKIP 指定页码，并使用关键字 LIMIT 指定每页返回的行数，这样就完成了一个分页查询的设计。这里只是简单地使用 Map 集合类来构造返回的页对象 page。

代码清单 2-22　查询分页设计

```
public Map<String, Object> findPage(Map<String, Object> params){
    Map<String, Object> page = new HashMap<>();
    try ( Transaction tx = graphDb.beginTx() ) {
        String query = "MATCH (n) WHERE n.name =~ {name} RETURN n SKIP {skip}
LIMIT {limit}";
        Result result = graphDb.execute(query, params);
        Iterator<Node> n_column = result.columnAs("n");
```

```
        List<Map<String, Object>> content = new ArrayList<>();
        while(n_column.hasNext()){
            Node node = n_column.next();

            Map<String, Object> data = new HashMap<>();
            data.put("id",node.getId());
            data.put("name", node.getProperty("name"));
            data.put("create", new Date((Long) node.getProperty("create")));

            content.add(data);
        }

        query = "MATCH (n) WHERE n.name =~ {name} RETURN count(n) as count";
        result = graphDb.execute(query, params);

        if(result.hasNext()){
            Map<String,Object> row = result.next();
            page.put("totalElements", row.get("count"));
        }

        page.put("content", content);

        tx.close();
    }
    return page;
}
```

如果我们创建一些测试数据，就可以用来测试一下分页查询的效果。如代码清单 2-23 所示，程序中使用一个简单的 for 循环创建了 20 个节点，然后以 10 个节点为 1 页的方式查找出第 1 页的数据列表。

代码清单 2-23　查询分页测试程序

```
public static void main( String[] args ) throws IOException
{
    FileUtils.deleteRecursively(DB_PATH);
    //create data
```

```
    PageData pageData = new PageData();
    for(int i=0; i<20; i++) {
      pageData.createNode("test"+i);
    }
    //list page
    Map<String, Object> params = new HashMap<>();
    params.put( "name", "(?i)test.*");
    params.put("skip", 0*10);
    params.put("limit", 10);
    Map<String, Object> page = pageData.findPage(params);
    System.out.println("==page list==");
    for(Map<String, Object> list : (List<Map<String,
Object>>)page.get("content")){
      System.out.println("id="+list.get("id")+";name="+list.get("name")+";
create="+list.get("create"));
    }
    System.out.println("page total="+page.get("totalElements"));

    pageData.shutdown();
}
```

运行上面的程序，将输出如下结果：

```
create node id=0
create node id=1
create node id=2
create node id=3
create node id=4
create node id=5
create node id=6
create node id=7
create node id=8
create node id=9
create node id=10
create node id=11
create node id=12
create node id=13
create node id=14
```

```
create node id=15
create node id=16
create node id=17
create node id=18
create node id=19
==page list==
id=0;name=test0;create=Thu Nov 17 10:41:02 CST 2016
id=1;name=test1;create=Thu Nov 17 10:41:02 CST 2016
id=2;name=test2;create=Thu Nov 17 10:41:02 CST 2016
id=3;name=test3;create=Thu Nov 17 10:41:02 CST 2016
id=4;name=test4;create=Thu Nov 17 10:41:02 CST 2016
id=5;name=test5;create=Thu Nov 17 10:41:02 CST 2016
id=6;name=test6;create=Thu Nov 17 10:41:02 CST 2016
id=7;name=test7;create=Thu Nov 17 10:41:02 CST 2016
id=8;name=test8;create=Thu Nov 17 10:41:02 CST 2016
id=9;name=test9;create=Thu Nov 17 10:41:02 CST 2016
page total=20
```

Cypher 查询语言语法简洁，并且具有非常丰富的表现力。更多和更详细的有关 Cypher 查询语言的算法设计，我们将在第 4 章中进行介绍。上面测试的源程序请参照 https://github.com/mr-csj/neo4j-useapi/blob/master/traversal/src/main/java/com/test/PageData.java。

2.7 连接 Neo4j 服务器

Neo4j API 除了可以使用嵌入式数据库，也可以通过连接 Neo4j 服务器的方式来使用数据库。要想使用服务器方式，我们必须先安装 Neo4j 服务器。安装 Neo4j 服务器及其配置方法请参考本书第 3 章。

要想连接 Neo4j 服务器，必须使用 Neo4j 的驱动程序，这可以在工程的 Maven 管理中增加 Neo4j 的驱动依赖配置，如代码清单 2-24 所示。

代码清单 2-24　Neo4j 的驱动依赖配置

```
<dependency>
    <groupId>org.neo4j.driver</groupId>
    <artifactId>neo4j-java-driver</artifactId>
    <version>1.0.3</version>
</dependency>
```

下面通过一个简单的测试程序来演示如何使用 Neo4j API 连接 Neo4j 服务器，以实现数据的存取操作。如代码清单 2-25 所示，程序中使用 Bolt 驱动的方式来连接数据库，然后使用 Session 来运行 Cypher 查询语句，实现了创建和查询节点的操作。

代码清单 2-25　使用 Cypher 查询语言创建和查询节点

```
public static void main( final String[] args ) throws Exception{
    Driver driver = GraphDatabase.driver("bolt://192.168.1.214",
AuthTokens.basic("neo4j", "12345678"));
    Session session = driver.session();
    session.run( "CREATE (u:User {name:'one', email:'one@com.cn'})" );

    StatementResult result = session.run( "MATCH (u) WHERE u.name = 'one' RETURN
u" );
    while ( result.hasNext() )
    {
        Record record = result.next();
        Map<String, Object> map = record.get("u").asMap();
        Set<Map.Entry<String, Object>> set = map.entrySet();
        for(Map.Entry one : set){
            System.out.print(one.getKey() + "=" + one.getValue() + ";");
        }
        System.out.println();
    }
    session.close();
    driver.close();
}
```

通过上面的一些练习，会不会觉得使用 Cypher 查询语言来访问数据库比使用 Neo4j

API 的方法更加容易实现呢？是的，在后续的章节中，我们在使用 SDN 来访问数据库时，也会将大部分数据访问操作通过 Cypher 查询语言来实现。

使用 Neo4j API 的方法就介绍到这里。如果想了解更多有关 API 的使用方法，则可以参考官方的说明文档：https://neo4j.com/docs/java-reference/current/。

2.8 关于事务

从上面一些使用 Neo4j API 的例子中我们知道，不管是写入操作还是读取操作，都需要开启事务管理，即操作数据库的程序都必须包含在如代码清单 2-26 所示的程序段中，才能正常完成操作。而在使用 Cypher 查询语言时，在表面上好像看不到有关使用事务的痕迹，但是它的实现始终是由事务来管理执行的。

代码清单 2-26 事务管理

```
try ( Transaction tx = graphDb.beginTx() )
{
        ......
}
```

2.8.1 Neo4j 支持完整的事务管理特性

为了充分维护数据的完整性，并保证良好的事务行为，Neo4j 支持完整的 ACID（Atomicity，Consistency，Isolation，Durability）事务管理特性。

- 原子性：如果交易的任何部分发生故障，则数据库状态保持不变。
- 一致性：所有的交易数据在离开数据库时处于一致的状态。
- 隔离性：在一个事务期间，修改中的数据不能被其他操作访问。
- 持久性：在 DBMS 中可以始终恢复提交事务的结果。

对于这些事务特性，Neo4j 还有如下特别规定：

- 访问图数据、索引或架构等所有的数据库操作都必须在事务管理的范围内执行。

- 默认的隔离级别为 READ_COMMITTED。
- 通过遍历检索到的数据不会由其他事务所保护。
- 重复读可能会取得不同的数据结果（除非使用显式读锁）。
- 可以使用显式写锁来获取有关节点和关系，以实现更高水平的隔离级别（可串行化）。
- 锁在节点和关系的级别上获得。
- 死锁检测内置于核心事务管理之中。

2.8.2　交互周期

访问图数据、索引或架构等所有的数据库操作都必须在事务管理的范围内执行。这条特别规定说明了一个事务的交互周期，也就是说，有关数据的存取操作不能在事务的管理范围之外执行。

基于这一规定，我们使用 Neo4j API 设计出了如代码清单 2-27 所示的函数，如果在事务管理的范围之外调用，就不能成功返回节点的数据。即不管数据存在与否，如果想在事务之外调用这个函数，则只返回 Null。

代码清单 2-27　返回节点的函数设计

```
private Node findById(GraphDatabaseService graphDb, Long id){
    Node node = null;
    try(Transaction tx = graphDb.beginTx()){
        node = graphDb.getNodeById(id);
        tx.success();
    }
    return node;
}
```

另外，事务可以嵌套，所有嵌套事务都将被添加到顶层事务的范围之中。这样，当顶层的事务回滚时，整个事务都将回滚。

2.8.3 隔离级别

在一个事务中，重复读取数据，有可能会得到两个完全不同的结果（假如在这个时候有程序修改了某个属性）。使用显式读锁可以避免这种情况发生。

如果一个数据库的更新很频繁，那么在使用 Cypher 查询语言更新数据时也有可能出现更新失败的情况。要想避免这种情况的发生，可以使用显式写锁，如代码清单 2-28 所示。程序中使用了循环来更新节点的属性，为保证更新都被正常执行，这里使用了显式写锁。

代码清单 2-28 使用显式写锁的查询语句

```
MATCH (n:X {id: 42})
SET n._LOCK_ = true
WITH n.prop as p
// ... operations depending on p, producing k
SET n.prop = k + 1
REMOVE n._LOCK_
```

2.8.4 关于死锁

因为使用了锁，就有可能出现死锁的情况。所以，在 Neo4j 的核心程序中内置了死锁检测的机制。检测程序会在正在执行的所有事务中，在有可能抛出异常之前检测出死锁，并释放引起死锁的事务，而其他正在运行的事务将不会受到影响。被释放的事务将会回滚，即用户的操作将会终止，必须由用户重新提交操作。

死锁一般出现在并发的写请求之中，即在不能同时满足预期的隔离性和一致性时发生。为了避免出现死锁的情况，必须合理处理并发的写请求。例如，如果有多个事务同时进行，使用随机的顺序进行就比较容易出现死锁的情况；如果使用一种固定的顺序进行，就不容易发生死锁。另外，对于多线程处理程序，最好将写请求安排在一个单线程中执行，这也是一种很好的处理方法。

2.9 其他开发语言实例

对于使用其他编程语言的开发者来说，虽然不能直接使用 Neo4j API，但是使用 Neo4j

也是非常方便的。因为 Neo4j 是开源的，很多编程语言，如 Node.js、Python、PHP、Ruby、.NET 等，都提供了对 Neo4j 的支持。下面主要介绍 Node.js 和 Python 使用 Neo4j 的方法。

2.9.1　Node.js 访问 Neo4j

使用如下指令在 Node.js 中安装 Neo4j Driver：

```
npm install neo4j-driver@1.0.2
```

安装成功后，可以编写如代码清单 2-29 所示的脚本，实现对 Neo4j 的调用，并将脚本文件保存为 getdata.js。

代码清单 2-29　Node.js 使用 Neo4j 实例

```
var neo4j = require('neo4j-driver').v1;
var driver = neo4j.driver("bolt://192.168.1.214", neo4j.auth.basic("neo4j",
"12345678"));
var session = driver.session();
session
  .run( "MATCH (u:User) WHERE u.name = 'user' RETURN u" )
  .then(function( result ) {
        var record = result.records[0];
        var user = record['_fields'];
        console.log(JSON.stringify(user, null, 4));
    session.close();
    driver.close();
  }, function (err) {
  console.error(err);
})
```

这样就可以使用 Node.js 来运行脚本了。这个实例的运行情况和结果如下：

```
E:\test\nodejs>node getdata.js
[
    {
        "identity": {
            "low": 4268,
```

```
            "high": 0
        },
        "labels": [
            "User"
        ],
        "properties": {
            "password":
"$2a$10$EUuuQThVBDZErEk7eC9dd.df4AcEALeIY1f4o6ccI8BBVkdMOymSS",
            "sex": {
                "low": 1,
                "high": 0
            },
            "name": "user",
            "create": {
                "low": 1211561205,
                "high": 344
            },
            "email": "user@email.com"
        }
    }
]
```

2.9.2 Python 访问 Neo4j

使用如下指令安装 Python 的 Neo4j 依赖包 py2neo：

```
pip install py2neo
```

安装成功后，编写如代码清单 2-30 所示的脚本访问数据库，并将脚本文件保存为 p2n.py。

代码清单 2-30 Python 实例

```
[root@localhost ~]# vi p2n.py
from py2neo import Graph
graph = Graph(password="12345678")
print graph.data("MATCH (u:User{name:'user'}) RETURN u")
```

使用 Python 运行脚本文件 p2n.py 的结果如下：

```
[root@localhost ~]# python p2n.py
   [{u'u': (d718f28:User
{create:1478680311029,email:"user@email.com",name:"user",password:"$2a$10
$EUuuQThVBDZErEk7eC9dd.df4AcEALeIY1f4o6ccI8BBVkdMOymSS",sex:1})}]
```

其他如 Ruby、.NET、PHP 等编程语言的使用方法不再一一列举，有兴趣的读者请参考其官网指引：https://neo4j.com/developer/language-guides/。

2.10　小结

本章主要使用 Java 编程语言演示了如何使用 Neo4j API 来访问 Neo4j 数据库，并且通过对嵌入式数据库和服务器方式数据库的使用，初步了解了 Neo4j 的基本特性及其图数据库的一些使用方法。同时通过对标签、索引、遍历和事务管理等的实际操作和使用，加深了对 Neo4j 的理解。最后简要介绍了其他编辑语言如 Node.js、Python 等使用 Neo4j 的方法。从这些演示和练习中可知，Neo4j 是一个功能强大而又易于使用的图数据库。

通过本章的学习，相信你对 Neo4j 已经有了一个初步的了解。下一章我们将介绍如何安装和配置 Neo4j 服务器，以便更加直接地接触 Neo4j，从而更加形象和直观地理解 Neo4j，为后续章节的学习打下坚实的基础。

第3章
Neo4j的安装及使用

Neo4j 有两个版本，分别是社区版（Community Edition）和企业版（Enterprise Edition）。社区版是免费并且开源的，但只适用于单实例部署，即只能用于单机安装和使用。不过，社区版已经具备了 Neo4j 的全部功能，包括可编程的 API 调用、Cypher 查询语言使用和 ACID 事务管理等。社区版非常适合用来构建一个小型项目，或者用于项目的前期开发和调试。

Neo4j 的企业版扩展和丰富了社区版的功能，包括性能的大幅度提升和分布式服务的可扩展特性等。使用企业版，可以使用集群架构的解决方案，提供 7×24 小时不间断的高可用服务，支持在线的完整备份和增量备份功能，支持可能出现的灾难性数据恢复，充分保证数据的安全可靠，同时还能获得 Neo Technology 专业团队的技术支持服务。

下面介绍社区版的安装、配置和 Web 方式的客户端使用。企业版的安装、配置和使用方法请参照第 8 章的说明。

3.1 安装要求及推荐

为了保证 Neo4j 的安装能提供一台高性能的数据库服务器，我们需要了解一下 Neo4j 对服务器的要求，这些要求同样适用于企业版。

Neo4j 服务器的性能通常由内存大小、磁盘 I/O 速度、CPU 的计算能力等硬件配置所决定。下面列出这些配置的最低要求和推荐，以供参考。

CPU 要求：最低要求使用英特尔酷睿 i3。推荐使用英特尔酷睿 i7 或 IBM POWER 8。

内存要求：最低要求 2GB。推荐使用 16～32GB 或更多。

磁盘要求：最低要求 10GB SATA（Serial ATA）硬盘（当前大家使用的一般都是这种串口硬盘）。推荐使用固态硬盘（Solid State Drive，SSD），它相比于机械硬盘，可以提高 50 倍以上的读/写速度，并且具有更高的安全性。

文件系统：为了保证正确的 ACID 行为，文件系统必须能够支持磁盘缓冲同步机制，如使用 Linux 系统应该能够支持 FSYNC、fdatasync 等同步函数。

使用 Linux 系统，最低要求使用 EXT4 文件系统（或类似）。推荐使用 EXT4、ZFS 等文件系统。

CentOS 6.5 使用是的 EXT4 文件系统，CentOS 7 使用的是 XFS 文件系统。XFS 是一个比 EXT4 更加高级的文件系统，所以 CentOS 6.5 以上版本都符合要求。

软件要求：因为 Neo4j 是使用 Java 开发的，所以运行 Neo4j 需要一台 Java 虚拟机支持。Java 要求安装 JDK 1.8 及以上版本。

操作系统：Linux 操作系统可以使用 Ubuntu、Debian 或 CentOS 6.5 及以上版本。Windows 操作系统推荐使用 Windows 7 及 Windows Server 2012 等。

3.2　安装 Neo4j 服务器

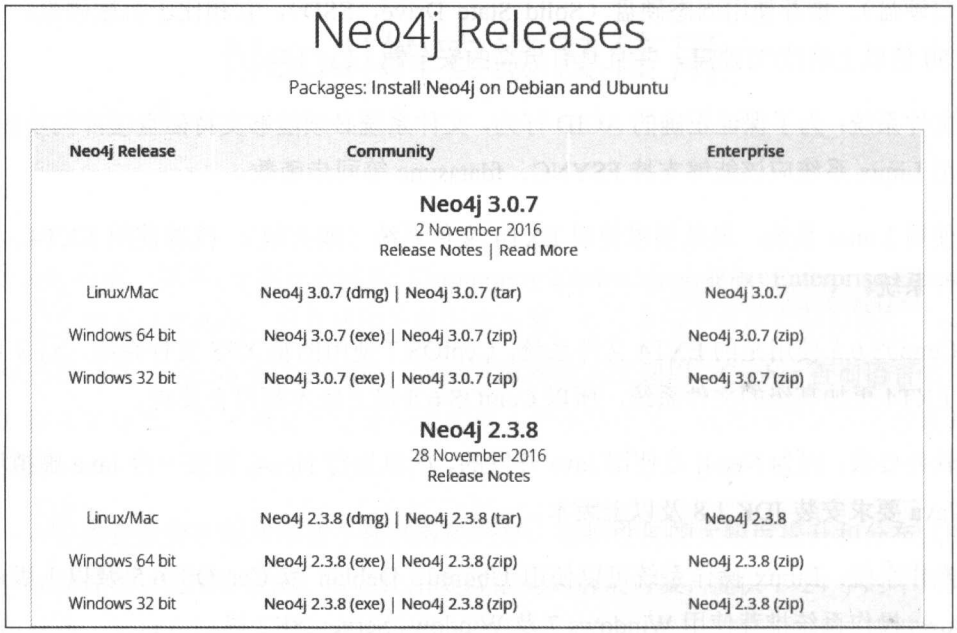
不管是 Linux、Windows 还是其他操作系统，在使用 Neo4j 之前，都必须先安装好 Java 虚拟机，因为运行 Neo4j 需要有 JVM（Java Virtual Machine）的支持。下面主要以 Linux 和 Windows 操作系统为例，介绍 Neo4j 的安装和配置方法。

3.2.1　下载 Neo4j

在 Neo4j 的官网中，可以下载各种版本的安装包。打开如下网址，可以选择合适的版本下载：

https://neo4j.com/download/other-releases/

图 3-1 是打开上述链接时的页面。当你打开上述链接的时候，可能上面提供的版本有些不一样，但只要选择接近版本即可。



Neo4j Releases

Packages: Install Neo4j on Debian and Ubuntu

Neo4j Release	Community	Enterprise
	Neo4j 3.0.7 2 November 2016 Release Notes | Read More	
Linux/Mac	Neo4j 3.0.7 (dmg) | Neo4j 3.0.7 (tar)	Neo4j 3.0.7
Windows 64 bit	Neo4j 3.0.7 (exe) | Neo4j 3.0.7 (zip)	Neo4j 3.0.7 (zip)
Windows 32 bit	Neo4j 3.0.7 (exe) | Neo4j 3.0.7 (zip)	Neo4j 3.0.7 (zip)
	Neo4j 2.3.8 28 November 2016 Release Notes	
Linux/Mac	Neo4j 2.3.8 (dmg) | Neo4j 2.3.8 (tar)	Neo4j 2.3.8
Windows 64 bit	Neo4j 2.3.8 (exe) | Neo4j 2.3.8 (zip)	Neo4j 2.3.8 (zip)
Windows 32 bit	Neo4j 2.3.8 (exe) | Neo4j 2.3.8 (zip)	Neo4j 2.3.8 (zip)

图 3-1　Neo4j 安装包下载

3.2.2　在 Linux 操作系统中安装 Neo4j

在 Linux 操作系统中安装，可以选择.tar 安装包下载。例如，我们选择 Neo4j 3.0.7 的 Linux 版本下载（Neo4j 3.x 的安装过程基本上都相同）。安装过程非常简单，只要解压缩，稍加配置就可以使用了。下面的安装实例是在 CentOS 7.0 中进行的。

解压缩安装包：

```
tar -xf neo4j-community-3.0.7-unix.tar.gz
```

启动服务：

```
./bin/neo4j start
```

如果没有修改过 Linux 的默认打开文件限制数，则在启动 Neo4j 时将有可能出现如下警告：

```
WARNING: Max 1024 open files allowed, minimum of 40000 recommended. See the
Neo4j manual.
```

Linux 的默认打开文件限制数是 1024，Neo4j 最低要求是 40 000，我们可以修改 Linux 的配置，将限制数修改为 65 536。

使用如下指令，查看打开文件限制数：

```
ulimit -a
```

如果显示结果有下列一行信息，则说明还是默认配置。

```
open files      (-n) 1024
```

使用下列指令临时修改并使之生效：

```
ulimit -n 65536
```

编辑下列文件，将修改写入系统配置之中，这样将可永久有效。

```
vi /etc/security/limits.conf
```

在文件末尾写入如下代码：

```
* - nofile 65536
* - nproc 65536
```

修改完成后重启 Neo4j 服务器，就不会出现警告了。

检查服务器是否启动成功。可以使用下列指令，如果能看到 Neo4j 的进程，就表明启动成功了。

```
ps -ef|grep neo4j
```

3.2.3　在 Windows 操作系统中安装 Neo4j

Neo4j 为 Windows 用户准备了安装版（.exe 文件）和压缩版（.zip 文件）安装包。

如果使用压缩版安装，则只要解压后就可以使用了。然后打开一个命令行窗口，将

路径切换到安装目录的 bin 下面，使用下列指令以控制台方式启动 Neo4j 服务器：

```
neo4j console
```

如果使用安装版安装，则安装完成后可在“开始”菜单中找到如图 3-2 所示的快捷菜单。在使用快捷菜单打开服务时，可以选择数据库的存放路径，一般使用默认选择就可以。

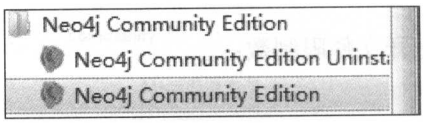


图 3-2　Neo4j 快捷菜单

3.3　Neo4j 基本配置

Neo4j 的默认配置只支持本地访问，为了使其能支持远程访问，可以修改相关的配置，即可在配置文件 neo4j.conf 中进行修改。

Neo4j 支持如下三种不同的连接方式：

- Bolt 连接方式。
- HTTP 连接方式。
- HTTPS 连接方式。

Neo4j 的 Web 客户端使用的是 HTTP 连接方式，在程序开发中一般可以使用 HTTP 和 Bolt 连接方式。HTTPS 的连接方式需要有安全证书支持，如果想让数据库开放在外网中访问，则可以使用这种连接方式。为了在局域网中支持远程连接方式，可以修改配置文件，将设置连接方式 IP 地址前面的注释符“#”去掉，并使用默认的 IP 设置（如 0.0.0.0）和端口设置即可。完成上述修改后的配置如下：

```
# Bolt connector
dbms.connector.bolt.type=BOLT
dbms.connector.bolt.enabled=true
```

```
dbms.connector.bolt.tls_level=OPTIONAL
# To have Bolt accept non-local connections, uncomment this line
dbms.connector.bolt.address=0.0.0.0:7687

# HTTP Connector
dbms.connector.http.type=HTTP
dbms.connector.http.enabled=true
# To accept non-local HTTP connections, uncomment this line
dbms.connector.http.address=0.0.0.0:7474

# HTTPS Connector
dbms.connector.https.type=HTTP
dbms.connector.https.enabled=true
dbms.connector.https.encryption=TLS
# To accept non-local HTTPS connection, change 'localhost' to '0.0.0.0'
dbms.connector.https.address= localhost:7473
```

这样，在启动 Neo4j 服务器后，我们就可以在局域网中通过安装服务器的 IP 地址来访问 Neo4j 数据库服务。

3.4　Neo4j 配置优化

在数据库的生产环境中，我们可以使用服务器的一些优化设置来保证 Neo4j 服务器能提供高性能的服务。服务器应该具备足够的内存，如普通的服务器都具有 32GB 或 64GB 的内存，这样就可以考虑对 Neo4j 的内存配置等做一些优化设置。

3.4.1　页面高速缓存

页面缓存被用来暂时存储操作数据，以提高数据的访问性能。所以，务必确保所有或至少大部分操作是从页面高速缓存中存取数据的，这将有助于避免使用昂贵的磁盘访问，从而产生最佳的访问性能。

那么，应该怎样设置页面缓存的大小呢？可以通过下面两种方法来计算。

第一种方法：如果不能精确地确定页面缓存的大小，则可以设置为系统内存的 50%减去最大 Java 堆大小之后剩余部分的大小。例如，操作系统有 32GB，堆大小设置为 8GB，那么页面缓存也设置为 8GB。

第二种方法：可以通过数据库的大小进行精确计算。比较合理的配置是，设置为数据库的大小加上 20%之后的 1/100。例如，数据库大小是 100GB，页面缓存设置为 1.2GB 比较合适。

然而数据库的大小又是怎么估算出来的呢？可以在数据库所在的目录（如 data/databases/graph.db）中执行下列指令进行估算：

```
du -hc *store.db*
```

确定了页面缓存的大小之后，可以在配置文件 neo4j.conf 中进行配置。例如，假定将页面缓存设为 8GB，配置完成后的结果如下：

```
# The default page cache memory assumes the machine is dedicated to running
# Neo4j, and is heuristically set to 50% of RAM minus the max Java heap size.
dbms.memory.pagecache.size=8g
```

3.4.2 堆大小

可用堆内存的大小是 Neo4j 性能的另一个重要配置。

一般而言，配置一个足够大的堆空间将有利于维持并行操作。对于许多设置而言，堆大小设置为 8～16GB 可以可靠地运行 Neo4j。

堆大小在 neo4j-wrapper.conf 配置文件中设置。如下所示是一个配置堆大小的实例。按照 Neo4j 的官方文档建议，初始大小和最大值设置为相同，可以避免垃圾回收出现不完整的情况。

```
dbms.memory.heap.initial_size=8g
dbms.memory.heap.max_size=8g
```

3.4.3 垃圾收集器

垃圾回收使用默认的配置就可以，下列配置被认为是比较适合大多数情况的：

```
dbms.jvm.additional=-XX:+UseG1GC
```

上面的一些配置优化可以根据实际情况进行设定。更多的安装配置说明可参考官方网站的帮助手册：http://neo4j.com/docs/operations-manual/current/。

3.5　使用 Neo4j 的 Web 控制台

Neo4j 的客户端不用安装任何程序，使用能够支持 HTML 5 的浏览器就可以访问服务器。为了达到最佳的访问效果，推荐使用 Google Chrome 浏览器。Neo4j 服务器启动后，就可以在浏览器中使用其 Web 客户端的控制台了。假设我们安装的服务器 IP 地址为 192.168.1.214，即可使用 http://192.168.1.214:7474 来打开客户端控制台。注意，如果服务器设置了防火墙，则必须开放相关端口。

第一次登录控制台的用户名和密码均为 neo4j，如图 3-3 所示。登录成功后，将被要求修改登录密码。

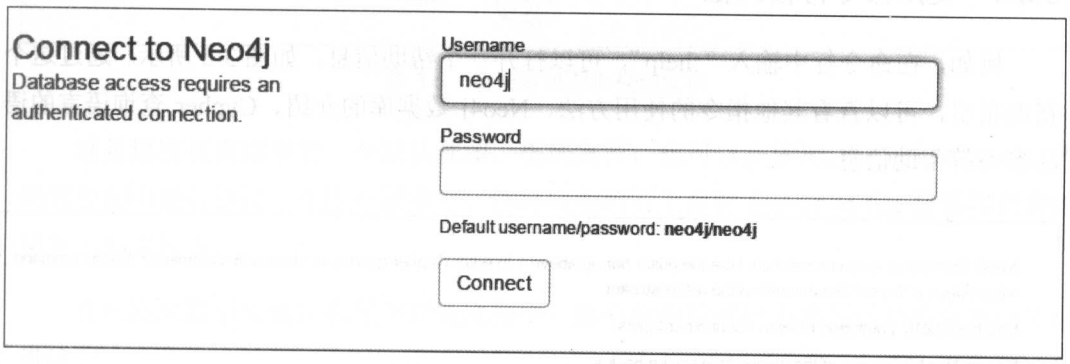


图 3-3　Neo4j 控制台登录界面

Neo4j 的数据库管理一般可以在它的客户端控制台上进行。使用 Web 控制台，还可以查看服务器的运行状态、执行 Cypher 查询语言创建节点和关系，或进行一些图形化的查询操作等。

如图 3-4 所示是进入控制台的首页界面。控制台的使用非常直观明了，下面进行简单介绍和说明。如果你觉得很容易掌握这些使用方法，则可以忽略下面这部分内容。

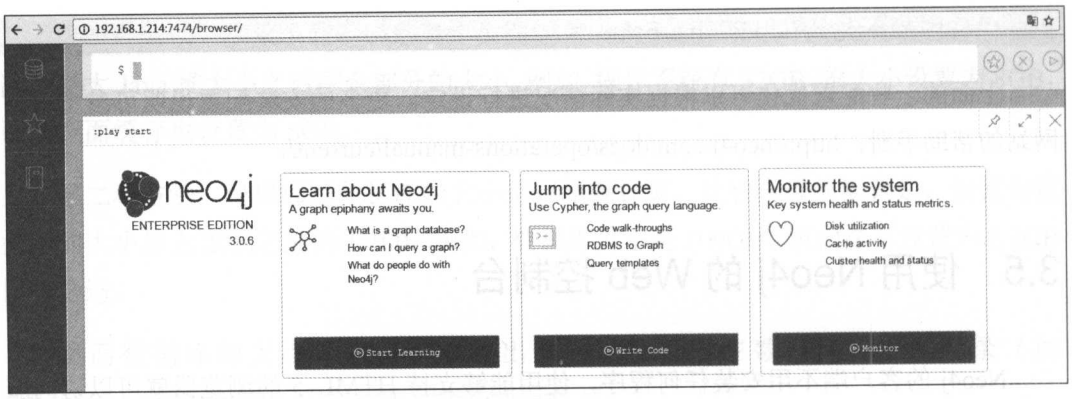


图 3-4　Neo4j 控制台首页

从图 3-4 中可以看出，页面左边是一个工具栏面板，面板上端有三个按钮，分别为数据库、收藏和文档，面板底部也有其他几个工具按钮；页面顶端是一个命令行输入框，在这里可执行一些指令，或运行 Cypher 查询语句，命令行右边的三个小按钮分别为收藏、清除和执行。

3.5.1　使用命令行输入框

例如，在命令行中输入“:help”，可以打开一个帮助信息，如图 3-5 所示。通过这个帮助指引，可以查看其他指令的使用方法、Neo4j 数据库的介绍、Cypher 查询语言的语法参考等帮助信息。

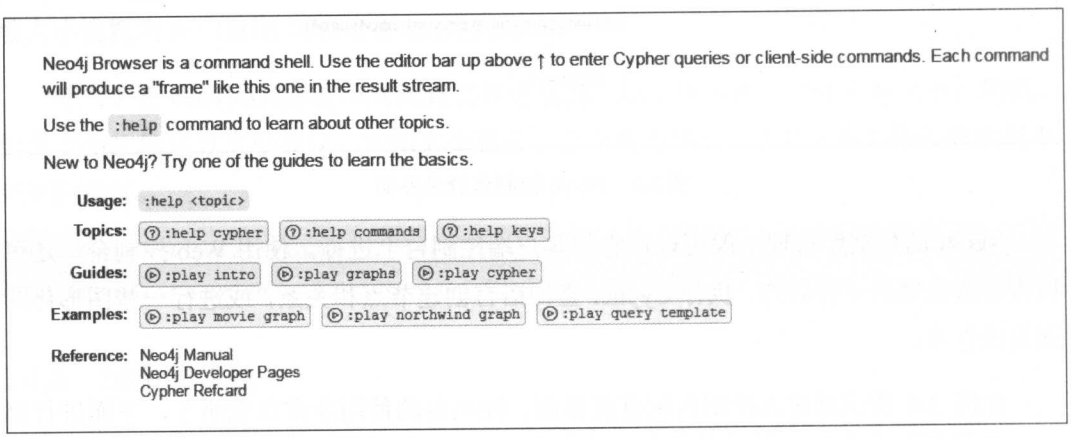


图 3-5　Neo4j 控制台帮助信息

在命令行中执行下列 Cypher 查询语句，即可任意返回 10 行数据。

```
MATCH (n) RETURN n LIMIT 10
```

如果数据库中有数据，如使用第 7 章中电影社区的数据，则将返回如图 3-6 所示的图形结果。

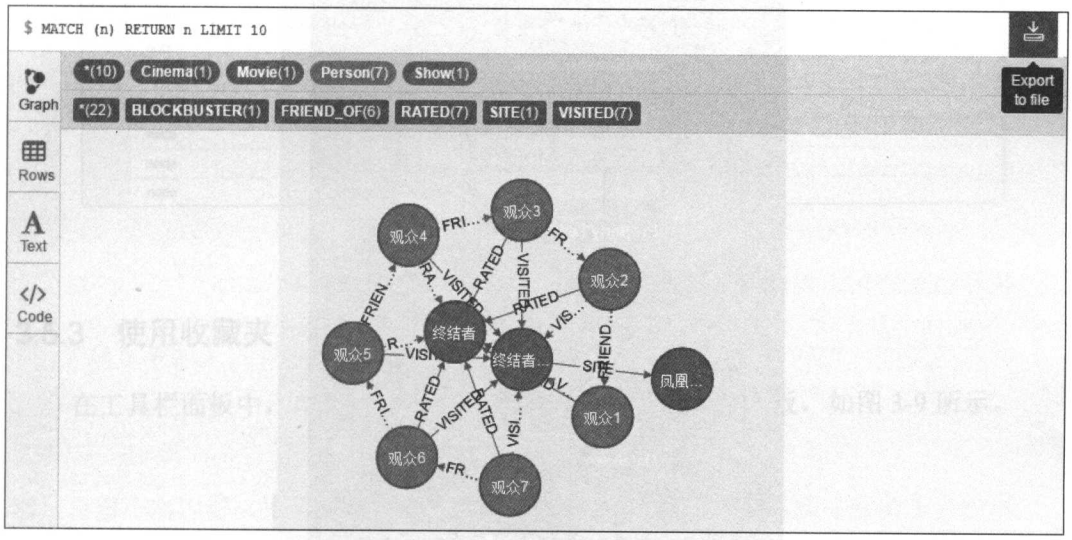


图 3-6　查询结果图形视图实例

这是返回查询结果的一个默认视图。根据需要，也可以在图形、记录行、文本、代码等视图中进行切换。在图形视图上，还可以使用右上角的 Export to file 按钮将查询结果导出到文件中。

在实际的数据库设计和程序开发过程中，这个命令行可以用来测试和验证查询算法设计。

3.5.2　数据库管理信息

在工具栏面板中，单击最上面的 Database 按钮，即可展开数据库信息面板，如图 3-7 所示。

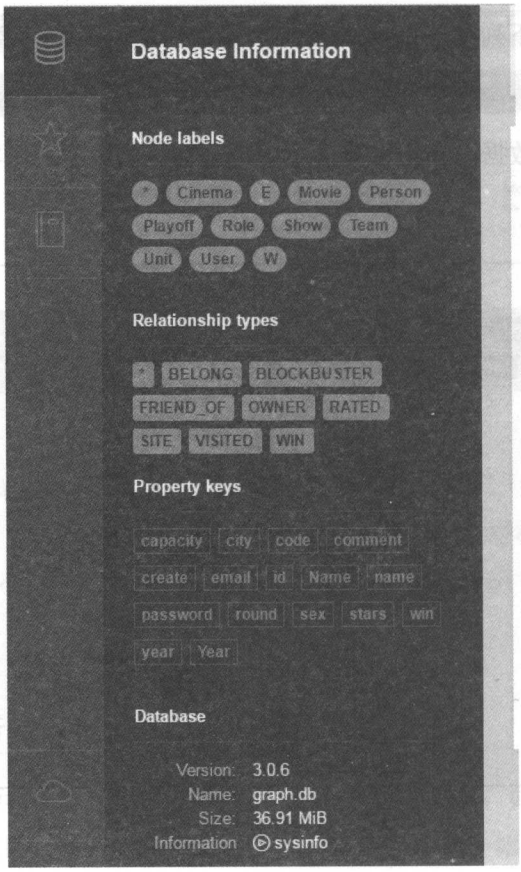


图 3-7　数据库信息面板

在数据库信息面板中包含如下栏目。

- 节点标签（Node labels）：包含数据库已有的标签。
- 关系类型（Relationship types）：包含数据库已有的关系。
- 属性关键字（Property keys）：包含数据库已有的属性关键字。
- 数据库（Database）：显示数据库的版本和大小等情况。

在上面这些栏目中，单击任何节点、关系和属性等对象，都将返回其相关的数据结果视图。例如，单击上面的属性关键字 name，即可返回如图 3-8 所示的查询结果记录行视图。

```
$ MATCH (n) WHERE EXISTS(n.name) RETURN DISTINCT "node" as element, n.name AS name LIMIT 25 UNION ALL MATCH (...
```

Rows | Text | Code

element	name
node	终结者
node	终结者第一场
node	观众1
node	凤凰影院
node	观众2
node	观众3
node	观众4
node	观众5
node	观众6
node	观众7
node	观众8
node	观众9
node	观众10

图 3-8　查询结果记录行视图实例

3.5.3　使用收藏夹

在工具栏面板中，单击 Favorites 按钮，则可展开收藏夹面板，如图 3-9 所示。

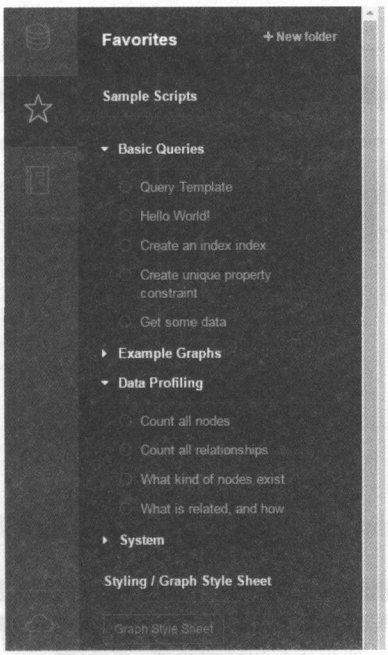


图 3-9　收藏夹面板

在收藏夹面板中包含如下栏目。

- 示例脚本（Sample Scripts）：包含基本查询（Basic Queries）、示例图（Example Graphs）、数据分析（Data Profiling）、系统（System）等子栏目。
- 样式/图表样式表（Styling/Graph Style Sheet）：可以设置图形视图显示的样式表。
- 输入（Import）：可以拖曳一个脚本文件到收藏夹中。

在收藏夹面板中，单击基本查询（Basic Queries）按钮，可以展开这个子栏目。如果单击其中的“Hello World!”，则可产生一条创建节点的语句，如下：

```
// Hello World!
CREATE (n {name:"World"}) RETURN "hello", n.name
```

执行这条语句，将创建一个简单节点，并返回“hello World”。

展开示例图（Example Graphs）子栏目，单击 Movie Graph，即可生成如下命令：

```
// Movie Graph
:play movie-graph
```

执行这个命令，将出现一个创建经典的电影图数据的操作指引。使用这个指引，即可创建一个电影实例的图数据，这是一个学习图数据的经典案例，如图 3-10 所示。

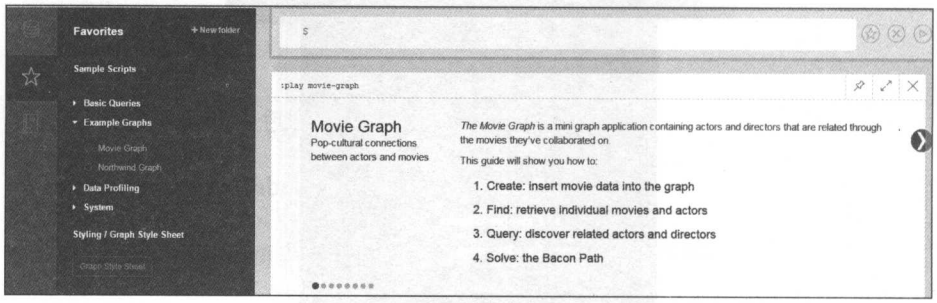


图 3-10　Neo4j 电影图数据经典案例

这个子栏目下面的 Northwind Graph 也是一个类似的图数据经典案例。

在数据分析（Data Profiling）子栏目中，提供了对当前数据库的一些统计工具按钮。例如，单击 Count all nodes，即可生成如下语句，这条语句可以用来计算当前数据库的节点数量。

```
// Count all nodes
MATCH (n)
RETURN count(n)
```

在这个子栏目中单击 What kind of nodes exist，即可统计节点的使用情况，如图 3-11 所示。

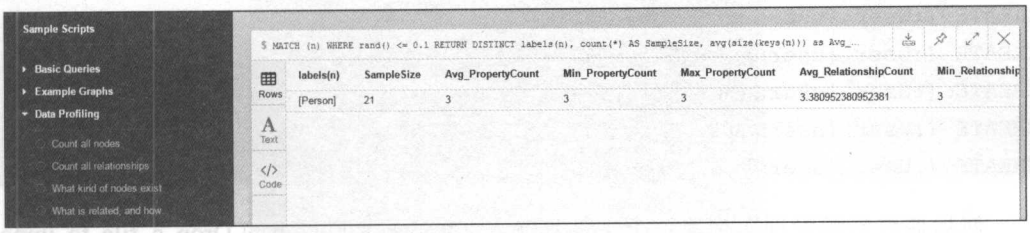


图 3-11　节点种类统计实例

在样式/图表样式表（Styling/Graph Style Sheet）子栏目中，单击 Graph Style Sheet 可以修改或者重置图形视图的显示风格。在查询结果的图形视图中，节点显示的彩色风格就是由这个样式定义的。如果有时发现节点显示都变成了灰色，则可以通过单击 Reset to default style 进行重置，如图 3-12 所示。

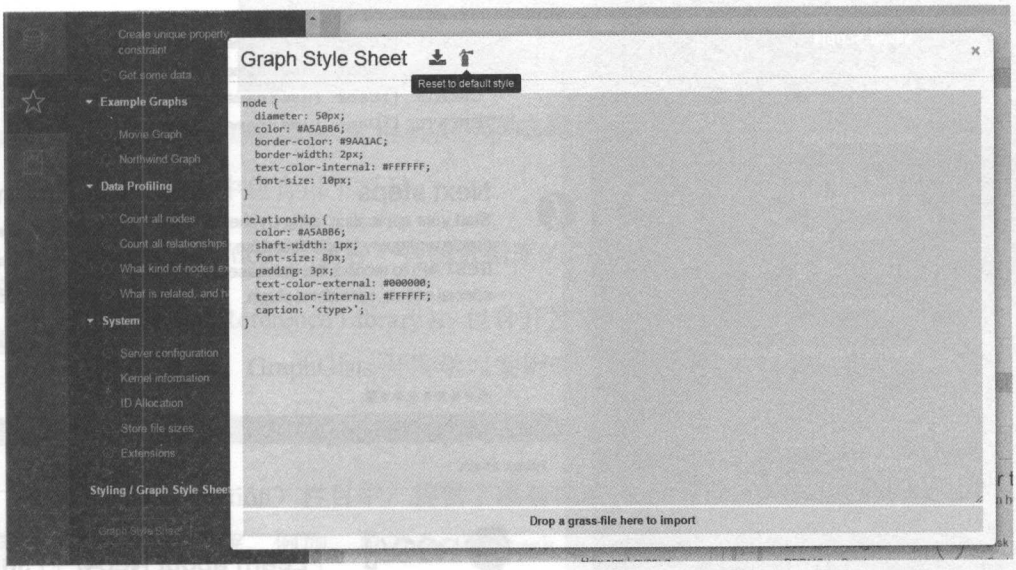


图 3-12　图形视图样式表设置

在输入（Import）栏目中，可以通过资源管理器拖曳一个查询脚本到收藏夹中，然后单击这个收藏就可以执行脚本。

例如，将下述代码保存为文件 users.cql。

```
// Create Users
CREATE (:User {username: "ben"})
CREATE (:User {username: "brad"})
CREATE (:User {username: "adam"})
CREATE (:User {username: "mick"})
CREATE (:User {username: "matt"})
CREATE (:User {username: "kevin"})
```

然后通过资源管理器拖曳这个文件到输入（Import）栏目下的 Drop a file to import Cypher or Grass 上面，这个脚本就将被保存到收藏夹面板的 Saved Scripts 栏目下面，如图 3-13 所示。可以看出，这个脚本在收藏夹中被保存为 Create Users。如果单击 Create Users，就可以将该脚本导入命令行输入框中执行。

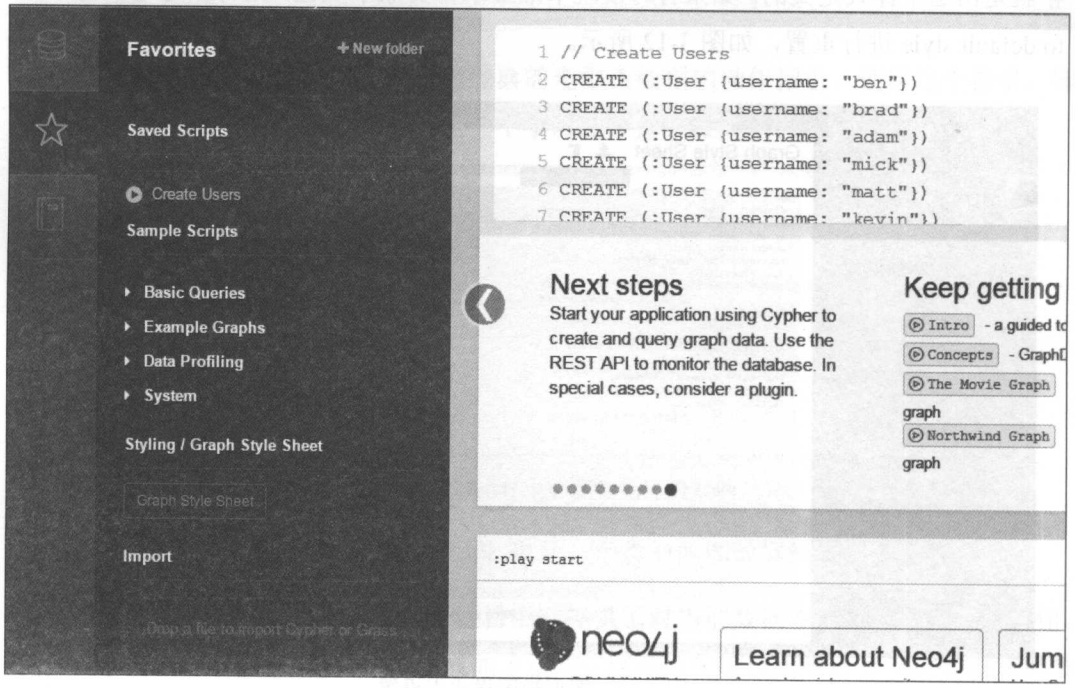


图 3-13　拖曳脚本实例

3.5.4　使用帮助手册

在工具栏面板中，单击 Documentation 按钮，即可展开如图 3-14 所示的帮助文档面板。

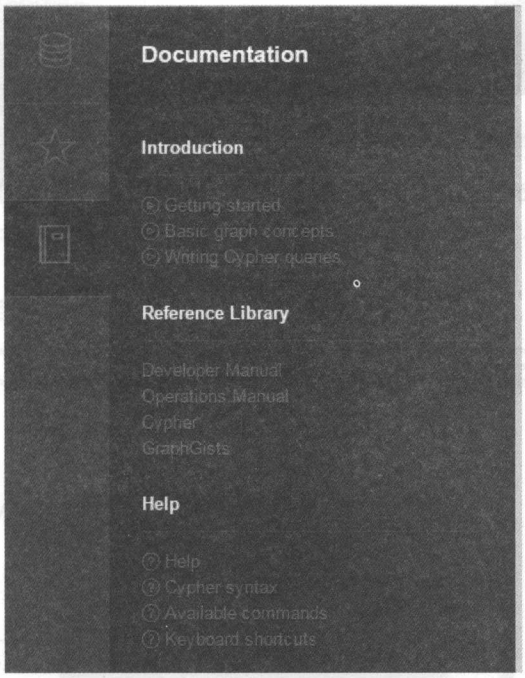


图 3-14　帮助文档面板

在帮助文档面板中包含以下栏目。

- 简介（Introduction）：包含一些图数据库的简要介绍。
- 参考资料库（Reference Library）：包含开发手册、数据库操作手册、Cypher 查询语言使用手册、GraphGists 开发学习案例等。
- 帮助（Help）：包含 Cypher 语法参考、可用命令、快捷键等帮助信息。

在简介（Introduction）栏目中，提供了图数据库的简明介绍，可以单击相关按钮对图数据库进行全面了解。例如，单击 Basic graph concepts，即可打开组成图的基本要素节点、关系和属性等概念介绍和说明文档，如图 3-15 所示。

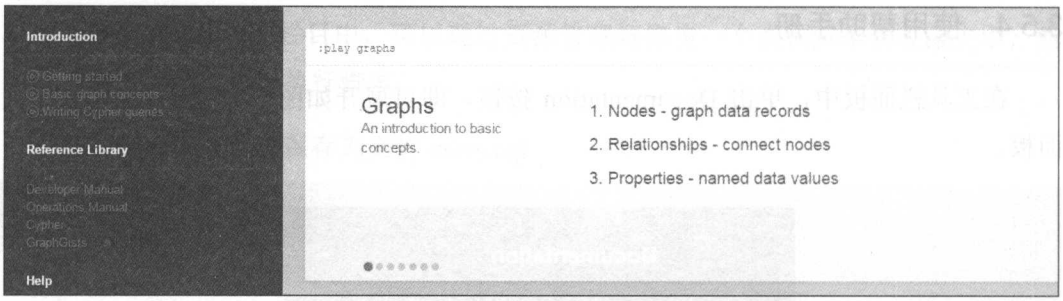


图 3-15　图数据库简介

在参考资料库（Reference Library）栏目中，提供了各个方面的帮助手册，包含开发手册、数据库操作手册、Cypher 查询语言使用手册、GraphGists 开发学习案例等。例如，单击 Developer Manual，将会打开如图 3-16 所示的开发手册。

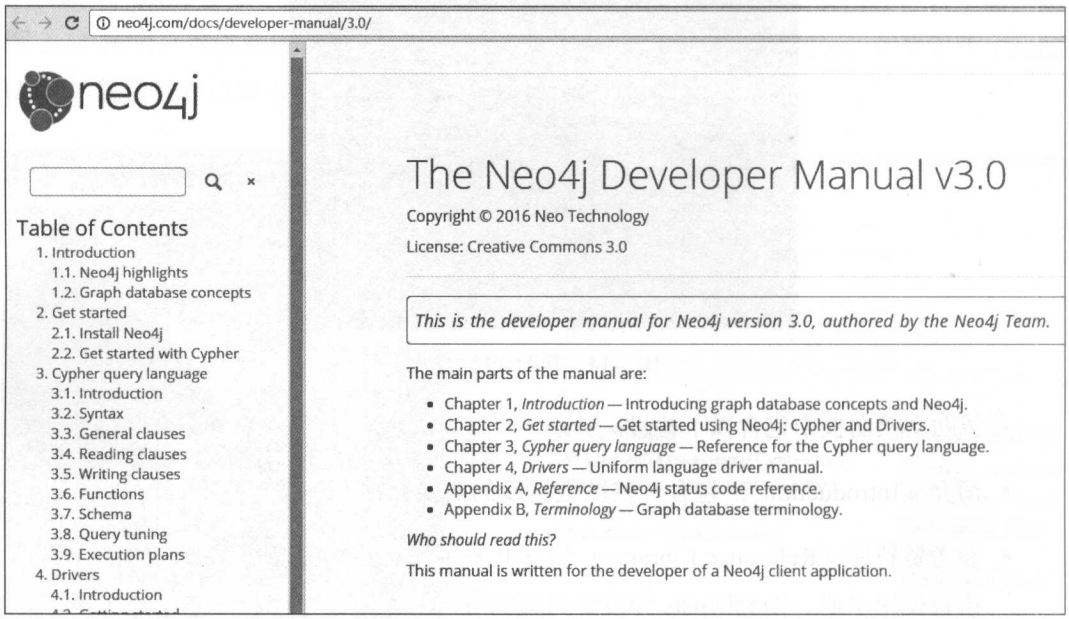


图 3-16　Neo4j 开发手册

在帮助（Help）栏目中，可以查看 Cypher 查询语言的语法、使用以“:”开头的命令以及一些快捷按钮等。例如，单击 Cypher syntax，可以查看 Cypher 查询语言的语法规则，如图 3-17 所示。

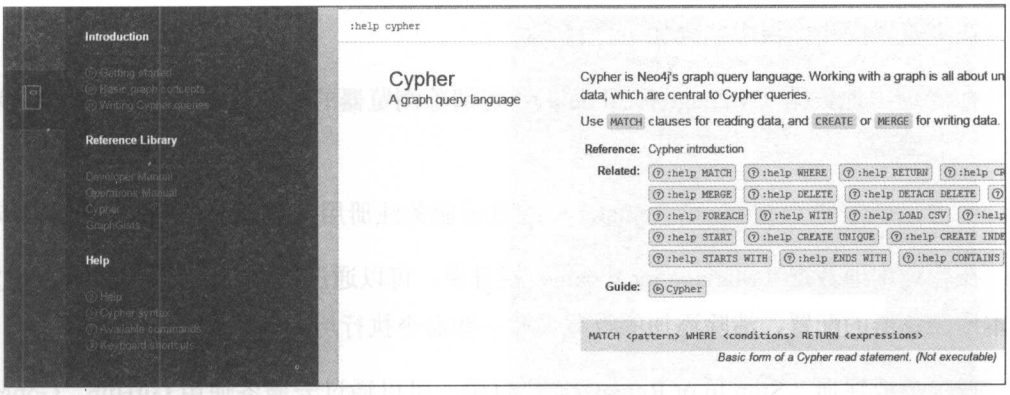


图 3-17　Cypher 查询语言语法规则帮助

3.5.5　使用浏览器同步功能

在工具栏面板底部单击 Cloud Service 按钮，即可展开浏览器同步面板，如图 3-18 所示。

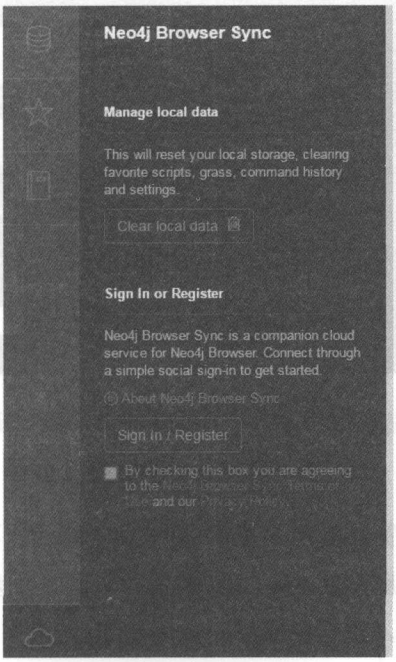


图 3-18　浏览器同步面板

在浏览器同步面板中包含如下栏目。

- 管理本地数据（Manage local data）：可清除浏览器的本地数据，包括一些设置和收藏等。
- 登录或注册（Sign In or Register）：通过云服务注册用户或登录，进行浏览器同步。

在管理本地数据（Manage local data）栏目中，可以通过单击 Clear local data 按钮重置本地浏览器的配置、清除添加的收藏夹和一些命令执行历史等。

在登录或注册（Sign In or Register）栏目中，可以通过云服务使用 GitHub、Google 或 Twitter 任一服务的账号进行登录，或使用以上服务注册账号后再进行登录。登录之后，即可实现浏览器同步。

在这个栏目中，单击 Sign In/Register 按钮，即可打开一个云服务登录界面，如图 3-19 所示。

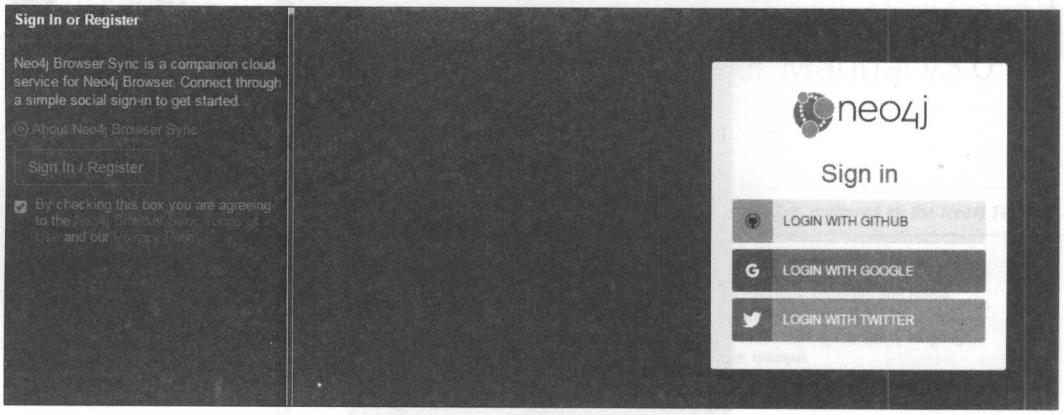


图 3-19　使用云服务登录

例如，我们使用 GitHub 的账号进行登录，即可打开如图 3-20 所示的登录认证界面。

登录成功后，将返回浏览器同步面板，如图 3-21 所示。不过，同步之后，暂时没有发现什么特别的功能。使用底端的 Sign out 按钮可以退出同步状态。需要注意的是，退出同步后，必须刷新浏览器，重新登录 Neo4j 服务器。

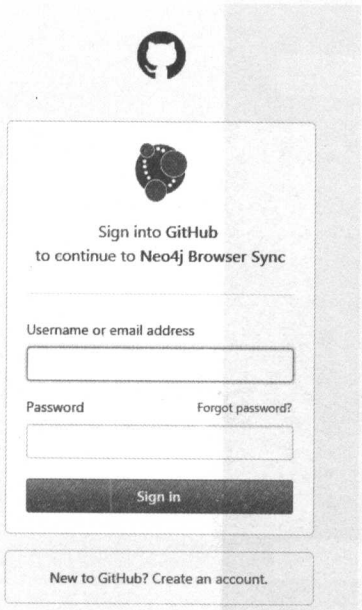


图 3-20　GitHub 登录认证

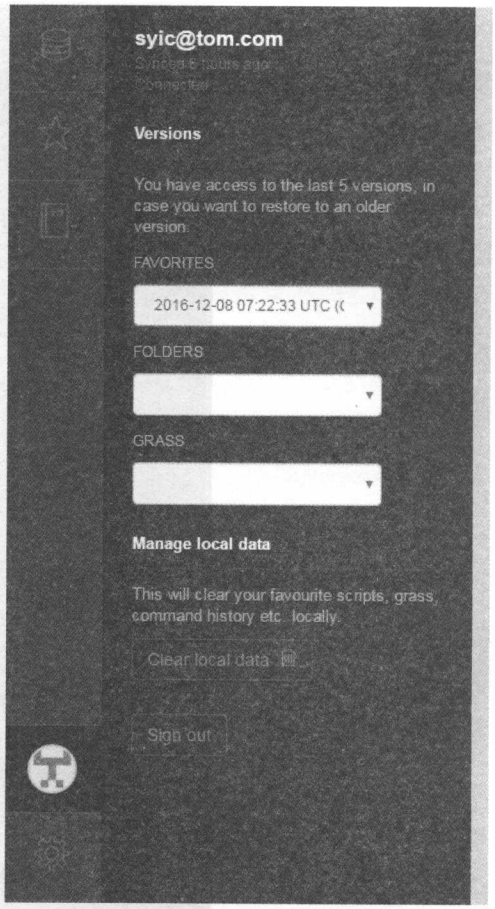


图 3-21　浏览器同步之后的界面

3.5.6　使用浏览器设置

在工具栏面板底部单击 Browser Settings 按钮，即可展开浏览器设置面板，如图 3-22 所示。

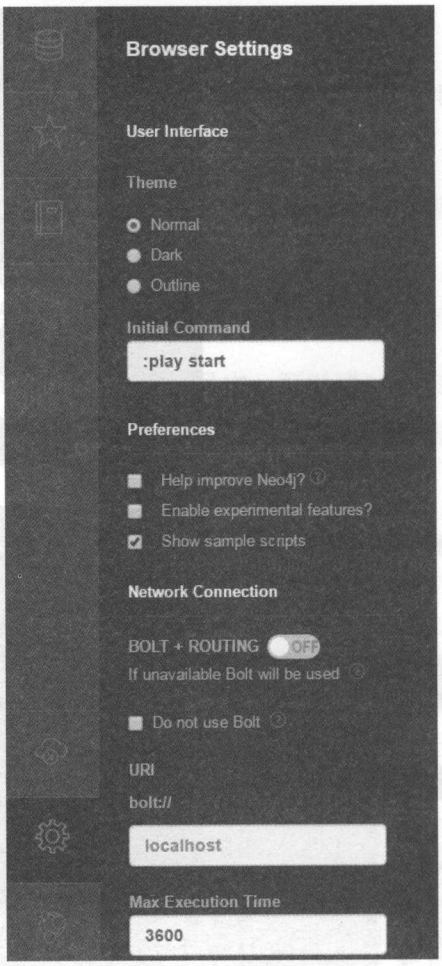


图 3-22　浏览器设置面板

如果使用 Neo4j 3.0 版本，则使用默认设置就可以了。我们发现 3.1 版的默认设置没有选择 Theme 下的 Normal 选项，这样在查询数据时将不返回图形视图界面。

3.5.7　关于 Neo4j

在工具栏面板底部单击 About Neo4j 按钮，即可展开关于 Neo4j 面板，在这里可以看到 Neo4j 的版权和许可等信息，如图 3-23 所示。

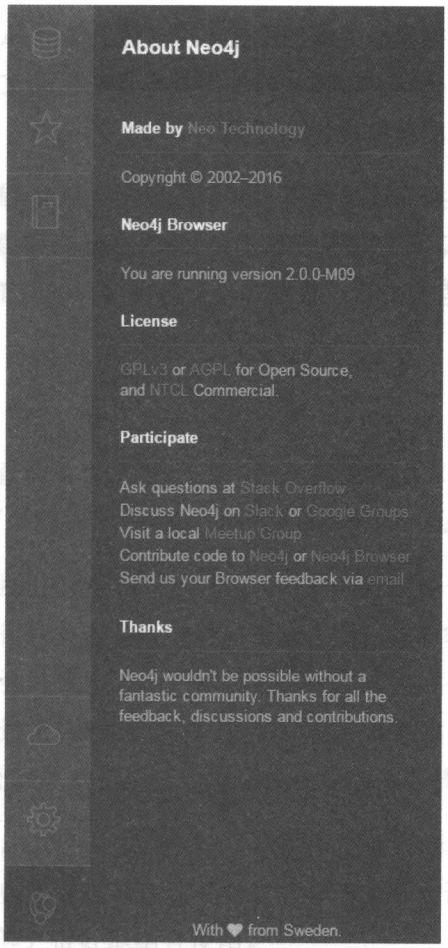


图 3-23　关于 Neo4j 面板

总之，Neo4j 的 Web 控制台不仅提供了全面且丰富的帮助文档或手册，还能非常方便地用来管理数据库和进行一些数据存取操作。控制台非常简单，可以边使用边学习，逐步熟悉基本的操作方法。

3.6　小结

本章主要介绍了 Neo4j 服务器的安装要求，演示了社区版的安装和配置，并说明了

如何在生产环境中优化数据库服务器的配置。同时对于使用 Neo4j 的 Web 客户端控制台也进行了一些简要说明和介绍，以及如何使用其丰富的帮助文档或手册来帮助我们更好地使用数据库及进行相关的程序设计和开发。

不管是管理 Neo4j 数据库还是使用 Neo4j 进行应用开发，都可以看出 Cypher 查询语言在这之中的重要地位。下一章我们将着重介绍 Cypher 查询语言的语法规则，以及如何更好地使用 Cypher 查询语言来存取数据和设计各种各样的查询算法。

第4章
Cypher查询语言简介

Neo4j 数据库使用的查询语言是 Cypher，这是在图数据库中比较实用的一种查询语言。Cypher 查询语言也可以简称为 CQL（Cypher Query Language）。

Cypher 是一种描述性的图数据库查询语言。Cypher 的语句结构基于简单的英语语法，整洁而又直观，这使得每条查询语句更加不言自明，非常容易理解。Cypher 的语句虽然结构简单，却富有很强的表现力，所以可以设计出非常高效的查询算法，即使很复杂的查询要求，也可以通过 Cypher 很容易地实现。Cypher 本身的设计遵循人性化的特点，它的设计目标是："让简单的事情变得容易，让复杂的事情成为可能。"

作为一种说明性语言，Cypher 重点表达了从图中能够清晰地检索出什么，而不用去关注它内部查询优化的一些实现细节。这使得它能够与命令式语言（如 Java）和脚本语言（如 Gremlin 和 JRuby 等）很好地结合使用。

Cypher 的一些关键字借鉴了以往一些优秀而既定的做法，如 WHERE 和 ORDER BY 等借鉴了 SQL 查询的做法，模式匹配 MATCH 借用了 SPARQL 表达的方法，一些集合语义也是从其他语言如 Haskell 和 Python 中借用过来的。

4.1 Cypher 语法基础

Cypher 查询语言的一条基本的查询语句可以使用如下方式来表示：

```
MATCH (n) WHERE id(n) = 1 RETURN n
```

这条查询语句的意思从字面上看就很好理解，它可以表述为：使用模式匹配的方式查询节点 n，通过条件表达式判断 n 的 ID 是否等于 1，然后返回符合条件的节点 n。

Cypher 查询语言对大小写不敏感，习惯上将关键字使用大写字母表示，这可以增强查询语句的可读性。

对于这里面所包含的语法，下面我们来逐步进行展开说明。如未加特别说明，下面的查询语句一般使用 Neo4j 的 Web 控制台来运行。

4.1.1 变量定义

在一条 Cypher 查询语句中，节点用小括号“()”来表示，关系用中括号“[]”来表示，而属性用大括号“{}”来表示。节点和关系可以定义变量，这种变量定义跟使用 AS 关键字定义的临时变量一样，可以很方便地在整条语句中引用。下面是一些节点和关系定义变量的方法。

定义一个节点变量：

```
(a)
```

表示节点之间的关系：

```
(a)-->(b)
```

显式地表示关系，并定义了关系变量：

```
(a)-[r]->(b)
```

在上面表示关系的表达式中，如果忽略符号“<”和“>”，则表示忽略关系的方向。

4.1.2 可用运算符

Cypher 的运算符是非常丰富的，除了具有一般查询语言使用的运算符，还增加了一些非常实用的列表运算符和正则表达式运算符等，为 Cypher 查询语言增加了更加丰富的运算能力和表达方法。

数学运算符：

```
+, - , *, /, %, ^
```

比较运算符：

```
=, <>, <, >, <=, > =
```

布尔值运算符：

```
AND, OR, XOR, NOT
```

字符串运算符：

```
+
```

列表运算符：

```
+, IN, [x], [x .. y]
```

正则表达式运算符：

```
=~
```

字符串匹配运算符：

```
STARTS WITH, ENDS WITH, CONTAINS
```

4.2　Cypher 读/写查询结构

Cypher 的读/写查询结构包含如下一些关键字：

```
[MATCH WHERE]
[OPTIONAL MATCH WHERE]
[WITH [ORDER BY] [SKIP] [LIMIT]]
(CREATE [UNIQUE] | MERGE)*
[SET|DELETE|REMOVE|FOREACH]*
[RETURN [ORDER BY] [SKIP] [LIMIT]]
```

其中，创建节点和关系不但可以使用 CREATE 关键字，也可以使用 MERGE 关键字。

4.2.1 用 CREATE 创建节点

使用关键字 CREATE 创建一个节点，如下查询语句所示：

```
CREATE (n:User {name: "Dav"})
```

在这条查询语句中，使用标签 User 创建了一个节点，并为节点增加了一个 name 属性，赋值为 Dav。执行查询后，将在数据库中创建一个包含如下内容的节点。

```
╒═══════════╕
│node       │
╞═══════════╡
│{name: Dav}│
└───────────┘
```

在创建节点时，还可以使用预定义的参数。如下语句表示使用参数创建多个节点。其中使用关键字 UNWIND 来展开集合“[{name:"a"},{name:"b"}]”的数据，使它成为一个列表。

```
UNWIND[{name:"a"},{name:"b"}] AS p CREATE (n:Test) SET n = p
```

执行上述查询语句后，将在数据库中创建如下所示的两个节点。

```
╒═══════╕
│node   │
╞═══════╡
│{name: a}│
├─────────┤
│{name: b}│
└─────────┘
```

4.2.2 用 CREATE 创建关系

下列代码演示了根据已有的节点创建一个 KNOWS 关系。

```
MATCH (n{name:"a"}),(m{name:"b"})
CREATE (n)-[r:KNOWS]->(m)
```

其中，使用匹配模式，通过节点的 name 属性查询节点，并把查询出来的节点分别使用变量 n 和 m 来表示，然后引用变量为它们创建关系，代码中的大于号“>”设定了

关系的方向。创建完成后，可以在数据库中查询到如图 4-1 所示的结果。

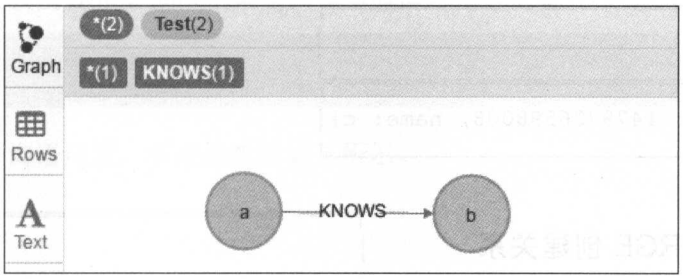


图 4-1　节点的关系

使用与上面相同的方法，可以在为节点建立关系时设定关系的属性，如下：

```
MATCH (n{name:"b"}),(m{name:"Dav"})
CREATE (n)-[:朋友 {亲密度: 3}]->(m)
```

即在为节点 n 创建“朋友”关系时，同时为关系创建一个属性“亲密度”，并设定属性的值为“3”。创建完成后的结果如下所示。

```
╒═══════════════════════════════════════╕
│p                                      │
╞═══════════════════════════════════════╡
│[{name: b}, {亲密度: 3}, {name: Dav}]  │
└───────────────────────────────────────┘
```

4.2.3　用 MERGE 创建节点

创建节点还可以使用关键字 MERGE，即以合并的方式来创建。在使用这种方式创建节点时，将会检查节点存在与否，如果节点已经存在，则使用已有的节点；如果节点不存在，则创建一个新节点。

下列查询语句使用 MERGE 来创建一个节点，并为其增加一个属性 created，赋值为当前时间。

```
MERGE (n:Test {name: "c"})
ON CREATE SET n.created = timestamp()
```

执行上述查询语句，将会首先检查标签为 Test、属性 name 的值等于 c 的节点是否存

在，如果存在即使用已有的节点，否则创建一个新节点。创建完成后的结果如下所示。

```
╒═══════════════════════════════════════╕
|node                                   |
╞═══════════════════════════════════════╡
|{created: 1479706598008, name: c}      |
└───────────────────────────────────────┘
```

4.2.4 用 MERGE 创建关系

除了可以用 CREATE 创建节点的关系，同样可以用关键字 MERGE，即使用合并的方式创建关系。在使用 MERGE 创建关系时，将会检查原来是否存在这种关系，如果存在则不修改任何数据，否则将为节点之间创建一个新关系。

下列所示的实例，使用 MERGE 为节点 a 和 b 创建一个 LOVES 关系，并将关系的方向设定为 a 指向 b。

```
MATCH (a:Person {name: 'b'}),(b:Person {name: 'c'})
MERGE (a)-[r:LOVES]->(b)
```

执行上述查询语句后，如果返回如下所示的结果，则说明新建了一个关系。

```
Created 1 relationship, statement executed in 191 ms.
```

如果关系已经存在，则不改变任何数据，并返回未做修改的提示结果。例如，我们使用合并的方式为节点 n 和 m 创建一个关系。

```
MATCH (n{name:"b"}),(m{name:"Dav"})
MERGE (n)-[:朋友 {亲密度: 3}]->(m)
```

在执行时，因为检查到上面的节点已经存在这个关系，所以将返回如下所示的结果。

```
(no changes, no rows)
```

4.2.5 用 SET 更新数据

在查询语句中使用关键字 SET，可以修改节点和关系的属性值，或者为节点和关系增加属性字段。

下列查询语句使用 SET 关键字修改节点 n 的属性 created，将其值设置为当前时间。其中使用了时间戳函数 timestamp()来返回当前时间。

```
MERGE (n:Test {name: "a"})
SET n.created = timestamp()
```

执行上述查询语句后，节点的数据如下所示。

```
╒═══════════════════════════════════════╕
│node                                   │
╞═══════════════════════════════════════╡
│{created: 1479707084760, name: a}      │
└───────────────────────────────────────┘
```

下列查询语句使用预定的参数为节点 n 增加两个属性。

```
UNWIND[{age:30},{addr:"sz"}] AS prop
MERGE (n:Test {name: "b"})
SET n += prop
```

成功执行上述查询语句后，原节点增加了两个属性，分别为 addr 和 age，结果如下所示。

```
╒═══════════════════════════════════╕
│node                               │
╞═══════════════════════════════════╡
│{name: b, addr: sz, age: 30}       │
└───────────────────────────────────┘
```

4.2.6　用 DELETE 删除数据

使用关键字 DELETE 可以删除节点和关系。在删除节点时，如果节点存在关系，则必须在删除关系后才能删除节点，或者同时删除。

下列查询语句将尝试删除所有节点。

```
MATCH (n) DELETE n
```

由于节点中还存在关系，所以在执行这条查询语句时，将会显示出如下错误信息，即提示必须删除关系后才能删除节点。

```
org.neo4j.kernel.api.exceptions.ConstraintViolationTransactionFailureException:
Cannot delete node<9>, because it still has relationships. To delete this node,
you must first delete its relationships.
```

下列查询语句使用模式匹配方式 MATCH 查找出“朋友”关系 r 及其朋友所指向的节点 m，并同时把 r 和 m 删除。

```
MATCH ()-[r:`朋友`]->(m) DELETE r,m
```

执行上述查询语句，将返回如下所示的结果，即同时删除了一个节点和一个关系。

```
Deleted 1 node, deleted 1 relationship, statement executed in 41 ms.
```

4.2.7 用 REMOVE 移除数据

使用关键字 REMOVE 来移除数据，可以移除一个标签中所有节点的数据，或者移除节点的一个属性。

下列查询语句将移除带有 Test 标签的所有节点，并且不管节点与其他节点是否存在关系。

```
MATCH (n) REMOVE n:Test
```

如果数据库中有三个带有 Test 标签的节点，则执行查询语句后将会返回如下所示的结果。

```
Removed 3 labels, statement executed in 100 ms.
```

如果被移除的节点中原来存在关系连接，那么这些节点被移除后，原来的关系就只能连接一些 NULL 节点。所以在使用 REMOVE 来移除节点时，需要谨慎操作。

下列查询语句可以用来移除节点的属性，执行查询将会移除节点 n 的属性 age。

```
MATCH (n{name:"b"})
REMOVE n.age
```

如果执行成功，则会返回如下所示的结果。

```
Set 1 property, statement executed in 46 ms.
```

如果执行不成功，则会返回如下所示的结果。

```
(no changes, no rows)
```

4.2.8　使用循环 FOREACH

使用关键字 FOREACH 可以在查询中设计一条循环语句。

下列查询语句使用 FOREACH 循环创建多个节点，其中使用 WITH 关键字创建一个集合 coll，然后在循环中使用集合 coll 的元素来创建一个节点。循环的前半部分使用变量 value 从集合 coll 中取值；后半部分创建一个标签为 Person 的节点，并将变量 value 的值赋给节点属性 name。

```
WITH ["a", "b", "c"] AS coll FOREACH (value IN coll | CREATE (:Person {name:
value}))
```

执行上述查询语句，将会返回如下所示的结果，即分别创建了三个标签、节点和属性。

```
Added 3 labels, created 3 nodes, set 3 properties, statement executed in 46 ms.
```

创建的节点将包含如下所示的数据。

```
╒═══════╕
|Person |
╞═══════╡
|{name: a}|
├─────────┤
|{name: b}|
├─────────┤
|{name: c}|
└─────────┘
```

4.3　使用索引

在第 2 章中，我们通过 Neo4j API 演示了如何使用索引。现在我们使用 Cypher 查询语言来使用索引，这比起 API 调用，将会更加容易。

4.3.1 创建和使用索引

下列查询语句将标签为 Person 的节点使用关键字 INDEX ON 为节点的属性 name 创建一个普通索引。

```
CREATE INDEX ON :Person(name)
```

执行上述查询语句后，将会返回如下所示的结果。

```
Added 1 index, statement executed in 175 ms.
```

这表明索引已经创建成功。在下列查询语句中，上面创建的索引将被自动调用。

```
MATCH (n:Person) WHERE n.name IN ["a", "b"]
RETURN n as Person
```

执行上述查询语句，可返回如下所示的数据。

```
╒══════════╕
│Person    │
╞══════════╡
│{name: a}│
├──────────┤
│{name: b}│
└──────────┘
```

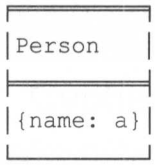
如果在查询中使用如下所示的语句，即先改变属性的大小写，再使用属性进行查找，则将不会自动使用上面的索引。

```
lower(n.name) = {value}
```

除了在查询中自动使用索引，也可以像下列查询语句那样，通过关键字 USING 显式指定使用的索引。

```
MATCH (n:Person)
USING INDEX n:Person(name)
WHERE n.name = 'a'
RETURN n as Person
```

执行上述查询语句，可返回如下所示的数据。



4.3.2　删除索引

下列查询语句使用 DROP 关键字删除一个已经存在的索引。

```
DROP INDEX ON :Person(name)
```

执行上述查询语句，如果索引存在，则将返回如下所示的结果。

```
Removed 1 index, statement executed in 105 ms.
```

4.4　使用约束

如果要创建具有唯一属性的索引，则必须使用约束。

4.4.1　创建约束

下列查询语句在标签为 Person 的节点中创建一个约束，并将节点的属性 name 指定为唯一属性。在创建约束时，将会同时创建索引，这条查询语句等同于创建了一个具有唯一属性约束的索引。

```
CREATE CONSTRAINT ON (p:Person)
ASSERT p.name IS UNIQUE
```

执行上述查询语句，如果返回如下所示的结果，则说明成功创建了一个约束。

```
Added 1 constraint, statement executed in 233 ms.
```

4.4.2　删除约束

使用下列查询语句，可以通过关键字 DORP 删除上面创建的唯一约束。

```
DROP CONSTRAINT ON (p:Person)
ASSERT p.name IS UNIQUE
```

执行上述查询语句，如果删除成功，则将返回如下所示的结果。

```
Removed 1 constraint, statement executed in 74 ms.
```

4.5 使用标签

正如在第 2 章中介绍的那样，使用标签非常重要，它相当于给节点进行分组，而且能提高查询的性能。标签就相当于关系型数据库的表名，而且比表名更加灵活。标签可以设置多个。

下面列举了一些常用的标签使用方法。

- 创建节点时使用标签。

```
CREATE (n:Person {name: {value}})
```

- 合并节点时使用标签。

```
MERGE (n:Person {name: {value}})
```

- 使用多标签的方法。

```
SET n:Spouse:Parent:Employee
```

- 查找数据时使用标签。

```
MATCH (n:Person)
WHERE n.name = {value}
```

- 移除数据时使用标签。

```
REMOVE n:Person
```

4.6　Cypher 只读查询结构

Cypher 只读查询结构包含如下一些关键字：

```
[MATCH WHERE]
[OPTIONAL MATCH WHERE]
[WITH [ORDER BY] [SKIP] [LIMIT]]
RETURN [ORDER BY] [SKIP] [LIMIT]
```

下列是一个基本的只读查询设计，使用了最基本的关键字 MATCH、WHERE 和 RETURN。由于在模式匹配中指定了节点及其关系，所以运行查询语句将返回一个完整的图数据。

```
MATCH p=(n:Test)-[:KNOWS]->()
WHERE n.name = "a"
RETURN p
```

执行上述查询语句后，将返回如图 4-2 所示的视图界面结果。

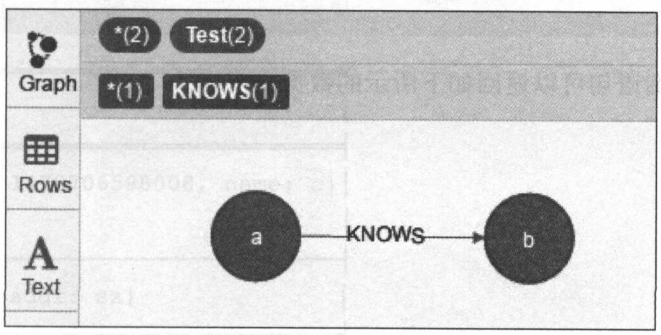


图 4-2　查询结果

4.6.1　条件过滤 WHERE

下列查询语句使用 WHERE 关键字设定查询的条件，即查询节点 n 的属性 name 不等于 a 的所有节点。

```
MATCH (n:Test)
WHERE n.name <> "a"
RETURN n
```

运行查询语句将返回如下所示的结果。

```
╒═══════════════════════════════════════╕
│n                                      │
╞═══════════════════════════════════════╡
│{name: b, addr: sz}                    │
├───────────────────────────────────────┤
│{created: 1479706598008, name: c}      │
└───────────────────────────────────────┘
```

4.6.2 联合查询 UNION

使用关键字 UNION 可以将两个查询的结果联合起来，如下：

```
MATCH (a)-[:KNOWS]->(b)
RETURN b.name
UNION
MATCH (a)-[:LOVES]->(b)
RETURN b.name
```

执行上述查询语句可以返回如下所示的数据。

```
╒══════╕
│b.name│
╞══════╡
│b     │
├──────┤
│c     │
└──────┘
```

4.6.3 使用链接 WITH

使用 WITH 可以将上一条查询语句的结果链接起来，用于下一条查询语句之中，即起到一种管道的功能。下列查询语句使用 WITH 引用了 MATCH 子句的 a 和 b。

```
MATCH (a)-[:KNOWS]->(b)
WITH a, count(b) AS knows
WHERE knows > 0
RETURN a
```

执行上述查询语句可返回如下所示的结果。

```
╒═══════════════════════════════════════╕
│a                                      │
╞═══════════════════════════════════════╡
│{created: 1479707084760, name: a}      │
└───────────────────────────────────────┘
```

4.6.4　返回结果 RETURN

下列查询语句在RETURN子句中使用DISTINCT过滤重复值，并使用关键字ORDER BY 对节点 n 的属性 name 进行倒序排序。

```
MATCH (n:Test)
RETURN DISTINCT n as test
ORDER BY test.name DESC
```

执行上述查询语句可返回如下所示的结果。

```
╒═══════════════════════════════════════╕
│test                                   │
╞═══════════════════════════════════════╡
│{created: 1479706598008, name: c}      │
│                                       │
├───────────────────────────────────────┤
│{name: b, addr: sz}                    │
├───────────────────────────────────────┤
│{created: 1479707084760, name: a}      │
│                                       │
└───────────────────────────────────────┘
```

下列查询语句在 RETURN 子句中使用关键字 SKIP 和 LIMIT 对查询结果进行分页。

```
MATCH (a:Test)
RETURN a as Test SKIP 2 LIMIT 2
```

执行上述查询语句可返回如下所示的结果。

```
╒═══════════════════════════════════════╕
│Test                                   │
╞═══════════════════════════════════════╡
│{created: 1479706598008, name: c}      │
└───────────────────────────────────────┘
```

4.7 使用 CASE 子句

CASE 条件判断子句的使用方法如下：

```
MATCH (n:Test) RETURN
CASE n.name
 WHEN "a" THEN 1
 WHEN "b" THEN 2
 ELSE 3
END as test
```

执行上述查询语句可返回如下所示的结果。

```
╒══════╕
│test  │
╞══════╡
│1     │
├──────┤
│2     │
├──────┤
│3     │
└──────┘
```

4.8 遍历的路径

一系列连接的节点和关系构成一条路径。路径也可以理解为一个关系的深度。在第

2 章的练习中，我们使用过路径来设计遍历算法。现在我们使用 Cypher 查询语言练习在遍历中使用路径来设计高效的查询算法。

4.8.1　最短路径

假如现有一部分电影社区的数据，如图 4-3 所示。在这些数据中，包含 4 个节点，分别是影院（Cinema）、电影（Movie）、观众（Person）和节目（Show）。这些节点具有如下关系：

- 观众与观众之间的朋友关系（FRIEND_OF）。
- 观众观看节目的关系（VISITED）。
- 观众给电影评分的关系（RATED）。
- 节目放映电影的关系（BLOCKBUSTER）。
- 节目在哪个影院上演的地点关系（SITE）。

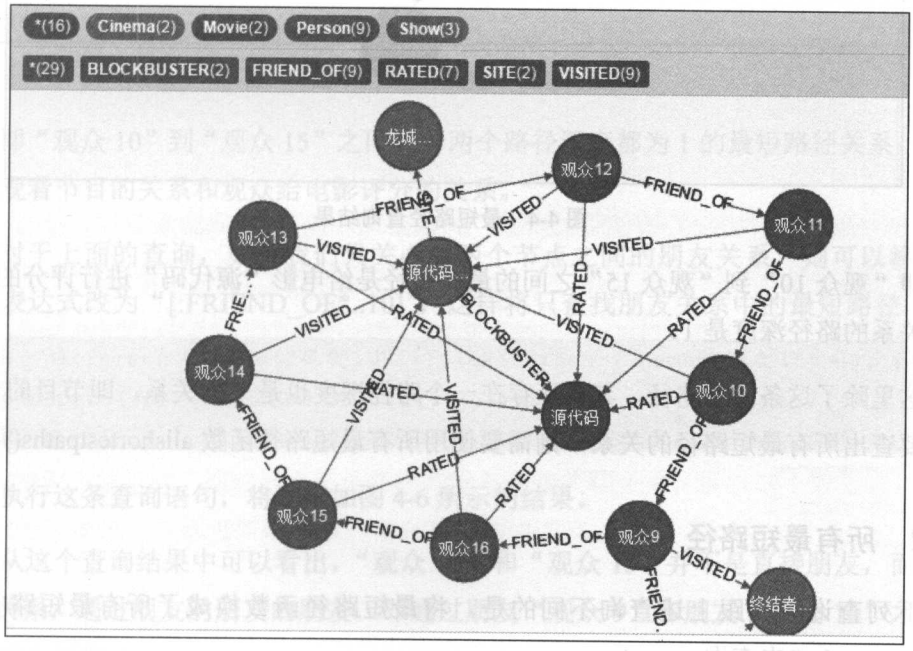


图 4-3　电影社区实例数据

下列查询语句使用最短路径遍历算法，查询从“观众 10”到“观众 15”之间的最短路径。其中用到了最短路径函数 shortestpath()，参数中使用的关系表达式“[*..10]”表示在路径深度 10 以内查找所有存在的关系中最短路径的关系。

```
MATCH (p1:Person { name:"观众 10" }),(p2:Person { name:"观众 15" }), p =
shortestpath((p1)-[*..10]-(p2))
RETURN p
```

执行上述查询语句，将返回如图 4-4 所示的结果。

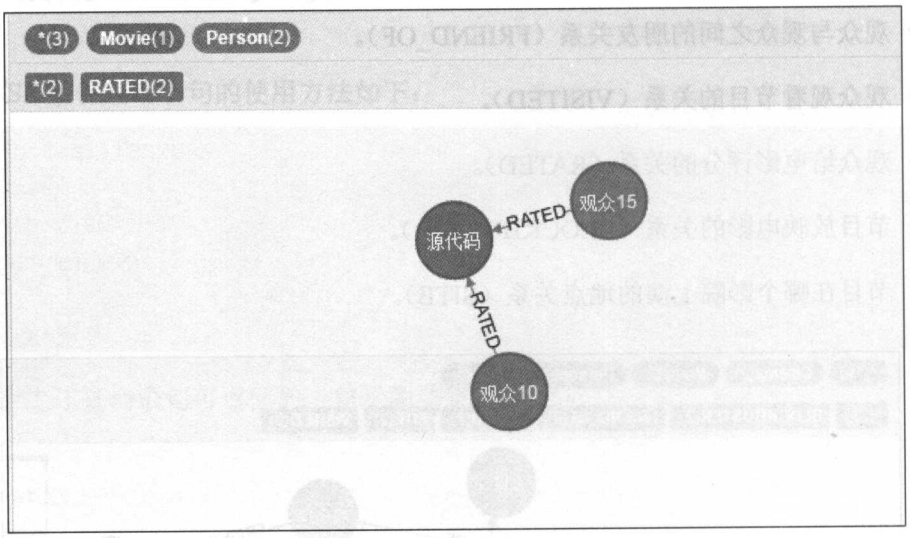


图 4-4　最短路径查询结果

即“观众 10”到“观众 15”之间的最短路径是给电影“源代码”进行评分的关系，这个关系的路径深度是 1。

这里除了这条最短路径，其实还存在一个路径深度也是 1 的关系，即节目的观看关系。要查出所有最短路径的关系，则需要使用所有最短路径函数 allshortestpaths()。

4.8.2　所有最短路径

下列查询算法跟上述查询不同的是，将最短路径函数换成了所有最短路径函数 allshortestpaths()来查询。

```
MATCH (p1:Person { name:"观众 10" }),(p2:Person { name:"观众 15" }), p =
allshortestpaths((p1)-[*..10]-(p2))
RETURN p
```

执行上述查询语句，将返回如图 4-5 所示的结果。

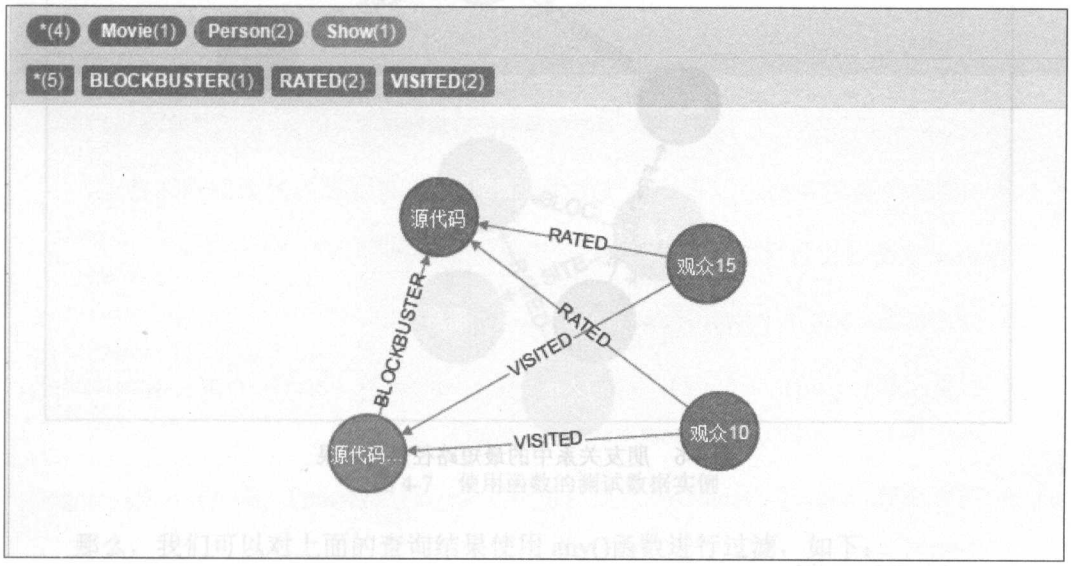


图 4-5　所有最短路径查询结果

即“观众 10”到“观众 15”之间包含两个路径深度都为 1 的最短路径关系，分别是观众观看节目的关系和观众给电影评分的关系。

对于上面的查询，如果我们只关心这两个节点之间的朋友关系，则可以将其中的关系表达式改为“[:FRIEND_OF*..10]”，这样将只查找朋友关系中的最短路径，如下：

```
MATCH (p1:Person { name:"观众 10" }),(p2:Person { name:"观众 15" }), p =
shortestpath((p1)-[:FRIEND_OF*..10]-(p2))
RETURN p
```

执行这条查询语句，将返回如图 4-6 所示的结果。

从这个查询结果中可以看出，“观众 10”和“观众 15”并不是直接朋友，而是隔了 3 个关系，通过朋友的朋友的朋友，即通过朋友“观众 9”和朋友“观众 16”才使他们二人成为朋友。

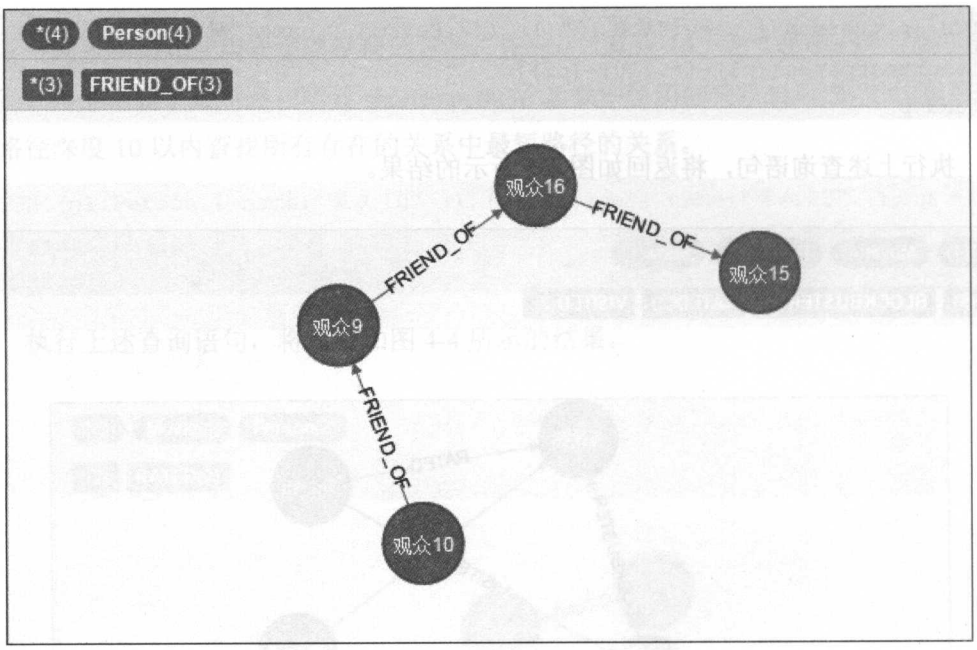


图 4-6　朋友关系中的最短路径查询结果

4.9　使用函数

Cypher 查询语言包含丰富的系统函数可供调用，其中一些函数具有很强的数据处理能力。在上面的练习中我们就使用过函数，如最短路径函数 shortestpath()。

有些函数的使用比较简单。例如，下列查询语句使用函数 size()来计算列表 coll 的大小。

```
WITH ["a", "b", "c"] AS coll RETURN size(coll)
```

执行这条查询语句将返回如下所示的结果。

```
+-----------+
| size(coll) |
+-----------+
| 3          |
+-----------+
```

有些函数的使用稍微复杂一些。例如，下面使用谓词函数 any()就是一个典型的例子。

假如通过下列查询语句可以返回如图 4-7 所示的数据。

```
MATCH p=(n:Person)-[*1..3]->(b)
WHERE n.name='观众 1'
RETURN p
```

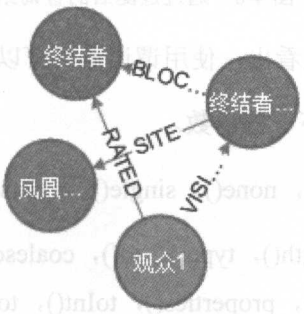


图 4-7　使用函数的测试数据实例

那么，我们可以对上面的查询结果使用 any()函数进行过滤，如下：

```
MATCH p=(n:Person)-[*1..3]->(b)
WHERE n.name='观众 1' AND any (x IN nodes(p) WHERE x.name = '凤凰影院')
RETURN p
```

在这里，首先使用模式匹配方式，设定只需查找观众的所有关系中路径深度为 1～3 的关系；然后使用过滤条件查找观众的名字为“观众 1”的节点，并使用谓词函数 any()进行过滤。即在查找出的关系所连接的节点中，通过过滤条件，查找节点的名字为“凤凰影院”的数据。其中使用了“x IN nodes(p)”找出上面查询结果中的节点数据。

执行这条查询语句，将返回如图 4-8 所示的结果。相比于如图 4-7 所示的查询结果，因为电影“终结者”跟影院“凤凰影院”没有直接关联关系，所以这个数据被过滤掉了，只保留了放映“终结者”的节目，它跟影院有直接的关联关系 SITE。

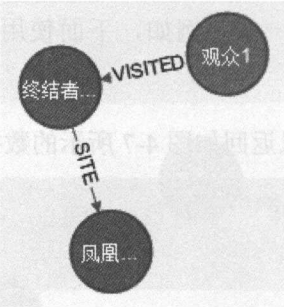


图 4-8　通过过滤后的查询结果

从上面这个使用实例可以看出，使用谓词函数可以表现更为复杂的查询要求。

Cypher 查询语言包含如下一些函数。

- 谓词函数：all()，any()，none()，single()，exists()。
- 标量函数：size()，length()，type()，id()，coalesce()，head()，last()，timestamp()，startNode()，endNode()，properties()，toInt()，toFloat()。
- 列表函数：nodes()，relationships()，labels()，keys()，extract()，filter()，tail()，range()，reduce()。
- 字符串函数：replace()，substring()，left()，right()，ltrim()，rtrim()，trim()，lower()，upper()，split()，reverse()，toString()。
- 数学函数：abs()，ceil()，floor()，round()，sign()，rand()。

另外还有对数函数、三角函数等。

如果要了解这些函数的使用方法，则可以参考官方的文档说明：http://neo4j.com/docs/developer-manual/current/cypher/functions/。

4.10　使用 CALL 调用存储过程

Cypher 查询语言也可以使用存储过程。下列查询语句使用关键字 CALL 调用了存储过程 dbms.procedures()，这可以列出系统可用的存储过程列表。

```
CALL dbms.procedures()
```

下列查询语句使用 db.labels()存储过程查询数据库中可用的标签。

```
CALL db.labels() YIELD label
RETURN label
```

执行上述查询语句，根据数据库中使用标签的实际情况，可返回如下所示的结果。

```
┌─────┐
|label|
├─────┤
|Movie|
├─────┤
|Actor|
├─────┤
|Unit |
├─────┤
|Role |
├─────┤
|User |
├─────┤
|Goods|
└─────┘
```

4.11　查询语句性能分析

我们所设计的查询语句和算法，都可以在前面加上一个关键字 PROFILE 来进行性能分析。

例如，下列查询语句在没有使用标签的情况下，查询属性 name 等于“观众 40”的节点。

```
PROFILE MATCH (p { name:"观众 40" }) RETURN p
```

执行上述查询语句将返回如图 4-9 所示的结果。从这个结果中可以看出，它将对整个数据库的所有节点即 313 个节点进行扫描，而最终只有一个节点符合要求，并返回了

一行记录。因为没有使用标签，如果数据量很大，那么这个查询将会很耗费性能。

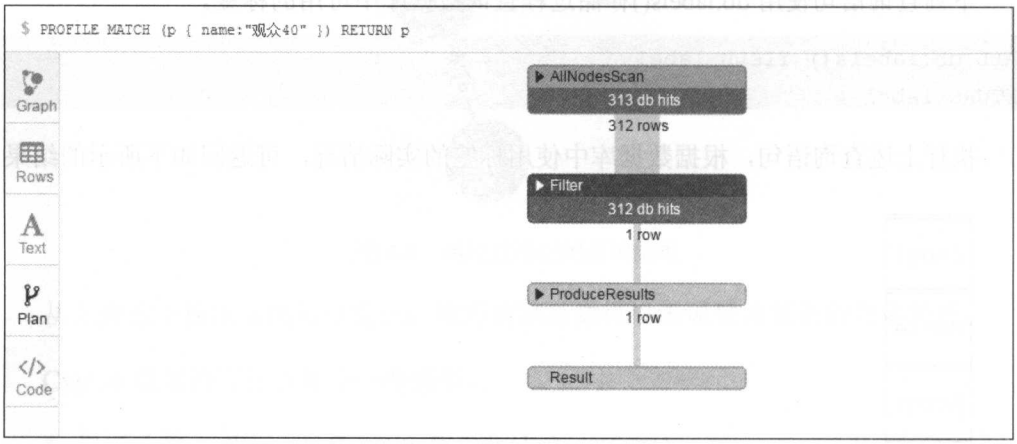


图 4-9　未使用标签的查询性能分析结果

现在再来看看使用标签的查询。如下列查询语句所示，我们在节点加上了标签 Person。

```
PROFILE MATCH (p:Person { name:"观众 40" }) RETURN p
```

执行这条查询语句，将返回如图 4-10 所示的结果。现在可以看出扫描的范围缩小了，它只在标签为 Person 的节点中执行查询。

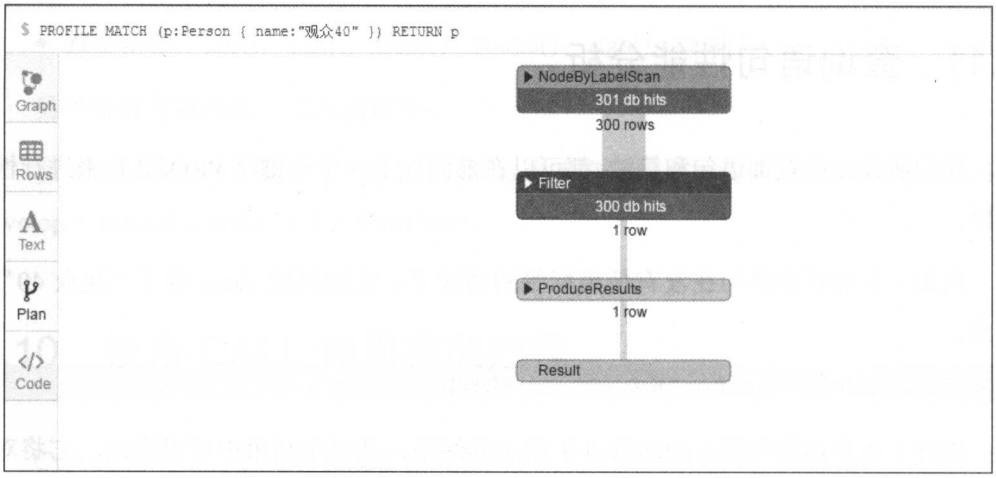


图 4-10　使用标签的查询性能分析结果

上面使用节点的 name 属性作为查询条件，现在我们再为 name 属性增加一个唯一属性约束，如下：

```
CREATE CONSTRAINT ON (p:Person)
ASSERT p.name IS UNIQUE
```

创建约束之后，将会同时创建索引。现在再使用上面的查询语句，如下：

```
PROFILE MATCH (p:Person { name:"观众 40" }) RETURN p
```

这条查询语句并没有改变什么，执行后，将返回如图 4-11 所示的结果。加上索引后，查询性能又提高了，它甚至不用从整个标签中搜索节点了，从属性的索引中直接就得到了查找的记录。

从上面的性能分析中可以看出，未加标签时，查询性能是最差的，将对整个数据库的所有节点进行检索；加上标签后，缩小了检索的范围，明显提高了查询性能；如果使用属性进行查询，加上索引后，查询性能又进一步提升了。

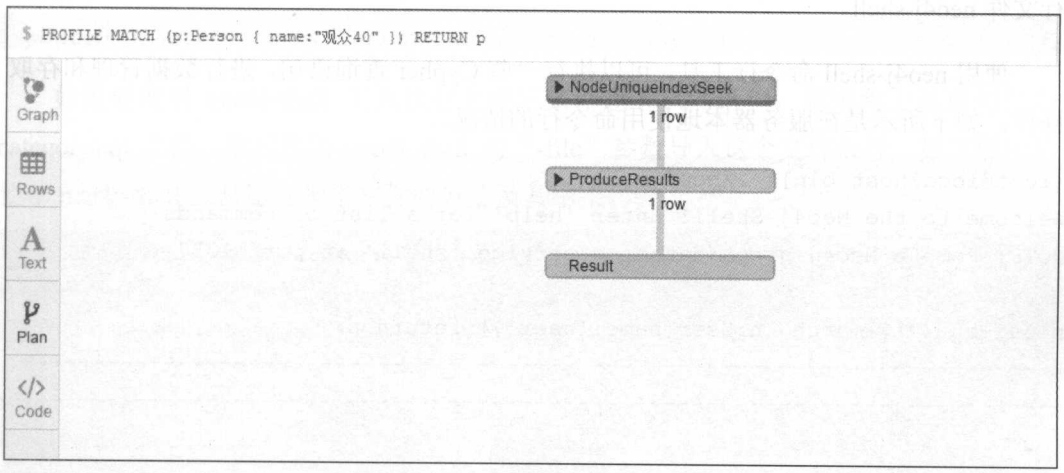


图 4-11　加了索引的查询性能分析结果

4.12　Cypher 的使用范围

Cypher 查询语言可在下列范围中使用：

- neo4j-shell。
- Web 控制台。
- Neo4j API。
- Rest API。

在实际应用开发中，我们也将大部分使用 Cypher 查询语言执行数据的存取操作。使用 Cypher 查询语言，还可以设计出一些比较复杂的查询算法，以满足一些复杂的功能需求。

下面简要介绍一下怎样在 neo4j-shell 和 Rest API 中使用 Cypher 查询语言。

4.12.1 在 neo4j-shell 中使用 Cypher 查询语言

Neo4j 自带一个命令行工具 neo4j-shell，即在安装服务器的 bin 目录中包含一个可执行文件 neo4j-shell。

使用 neo4j-shell 命令行工具，可以执行一些 Cypher 查询语句，进行数据管理和存取操作。如下所示是在服务器本地使用命令行的情况。

```
[root@localhost bin]# ./neo4j-shell
Welcome to the Neo4j Shell! Enter 'help' for a list of commands
NOTE: Remote Neo4j graph database service 'shell' at port 1337

neo4j-sh (?)$ match (u:User{name:'user'}) return u;
+-------------------------------------------------------------------------
-------------------------------------------------------------------------+
| u
|
+-------------------------------------------------------------------------
-------------------------------------------------------------------------+
|
Node[1081]{password:"$2a$10$Aud9xu9w4A7v3F95ZeKDCOWkTEVzZxLHulDN7GIEGv3F6
OwpQc.82",email:"user@email.com",name:"user",create:1476078664375,sex:1} |
+-------------------------------------------------------------------------
-------------------------------------------------------------------------+
```

```
1 row
15 ms
```

其中，使用查询语句“match (u:User{name:'user'}) return u;”返回了一条用户记录。这里必须注意的是，在使用 neo4j-shell 的命令行时，不能省略语句结束符“;”，即在一条语句结束准备敲回车键之前需要输入语句结束符，否则按回车键后命令行将会开始一个新行，并不开始运行语句。

不管是 Web 控制台的命令行，还是 neo4j-shell 的命令行，一次只能运行一条查询语句。如果需要多条语句一起执行，则可以通过 neo4j-shell 工具使用文件的方式来执行。

例如，有下列三条语句：

```
CREATE (:Category {name: "Best Sellers"});
CREATE (:Category {name: "Weight Loss"});
CREATE (:Category {name: "Advanced"});
```

如果在命令行上执行，例如，在 Web 控制台的命令行中执行，则会返回如下错误：

```
Expected exactly one statement per query but got: 3
```

如果要使用 neo4j-shell 工具执行上面三条创建节点的语句，则可将上述语句存为 category.cql 文件，然后使用 neo4j-shell 的“-file”参数导入这个文件执行。如下所示是使用 neo4j-shell 工具运行脚本文件的方法及其结果。

```
[root@localhost bin]# ./neo4j-shell -file /tmp/category.cql
+-------------------+
| No data returned. |
+-------------------+
Nodes created: 1
Properties set: 1
Labels added: 1
17 ms
+-------------------+
| No data returned. |
+-------------------+
Nodes created: 1
Properties set: 1
Labels added: 1
```

```
8 ms
+-------------------+
| No data returned. |
+-------------------+
Nodes created: 1
Properties set: 1
Labels added: 1
5 ms
```

从运行结果中可以看出，执行脚本文件后，在本地数据库中创建了三个节点。

4.12.2 在 Rest API 中使用 Cypher 查询语言

安装 Neo4j 服务器后，就已经具有 Rest API 的功能，可以支持 RESTful 调用。

在 Rest API 中使用 Cypher 查询语言，可以使用 JSON 格式的文本来传递参数，通过 HTTP 提交查询语句。如下所示是一个通过 curl 使用 Rest API 的实例。

```
curl -l -H "Content-type:application/json; charset=UTF-8" -X POST -d
'{"query" :"match (n{name:{name}}) return n","params": {"name" : "user"}}'
http://192.168.1.214:7474/db/data/cypher -H "Authorization: Basic
bmVvNGo6MTIzNDU2Nzg="
```

其中，字符串“bmVvNGo6MTIzNDU2Nzg=”是数据库的用户名和密码“neo4j:12345678”，通过 Base64 算法生成的数据库服务器认证码，提供给数据库服务器进行安全认证。这里使用了一个简单的查询“match (n{name:{name}}) return n”，其中“{name}”将通过“params”来使用参数“{"name" : "user"}}”。查询成功将返回一个节点。

如下所示是在 Linux 系统中执行上述指令的情况。

```
[root@localhost /]# curl -l -H "Content-type:application/json; charset=UTF-8"
-X POST -d '{"query" :"match (n{name:{name}}) return n","params": {"name" :
"user"}}' http://192.168.1.214:7474/db/data/cypher -H "Authorization: Basic
bmVvNGo6MTIzNDU2Nzg="
{
  "columns" : [ "n" ],
  "data" : [ [ {
    "metadata" : {
```

```
    "id" : 4268,
    "labels" : [ "User" ]
  },
  "data" : {
    "password" :
"$2a$10$EUuuQThVBDZErEk7eC9dd.df4AcEALeIY1f4o6ccI8BBVkdMOymSS",
    "sex" : 1,
    "name" : "user",
    "create" : 1478680311029,
    "email" : "user@email.com"
  },
  "paged_traverse" :
"http://192.168.1.214:7474/db/data/node/4268/paged/traverse/{returnType}{
?pageSize,leaseTime}",
  "outgoing_relationships" :
"http://192.168.1.214:7474/db/data/node/4268/relationships/out",
  "outgoing_typed_relationships" :
"http://192.168.1.214:7474/db/data/node/4268/relationships/out/{-list|&|
types}",
  "labels" : "http://192.168.1.214:7474/db/data/node/4268/labels",
  "create_relationship" :
"http://192.168.1.214:7474/db/data/node/4268/relationships",
  "traverse" :
"http://192.168.1.214:7474/db/data/node/4268/traverse/{returnType}",
  "extensions" : { },
  "all_relationships" :
"http://192.168.1.214:7474/db/data/node/4268/relationships/all",
  "all_typed_relationships" :
"http://192.168.1.214:7474/db/data/node/4268/relationships/all/{-list|&|
types}",
  "property" :
"http://192.168.1.214:7474/db/data/node/4268/properties/{key}",
  "self" : "http://192.168.1.214:7474/db/data/node/4268",
  "incoming_relationships" :
"http://192.168.1.214:7474/db/data/node/4268/relationships/in",
  "properties" :
"http://192.168.1.214:7474/db/data/node/4268/properties",
```

```
   "incoming_typed_relationships" :
"http://192.168.1.214:7474/db/data/node/4268/relationships/in/{-list|&|
types}"
  } ] ]
```

从上面的执行结果中可以看出，RESTful 调用也可以使用类似“http://192.168.1.214:7474/db/data/node/4268”的链接在局域网中进行调用，即用节点的 ID“4268”直接在浏览器上使用 Rest API。

第一次打开上面的链接时，将会出现一个认证界面，要求输入数据库的用户名和密码，如图 4-12 所示。

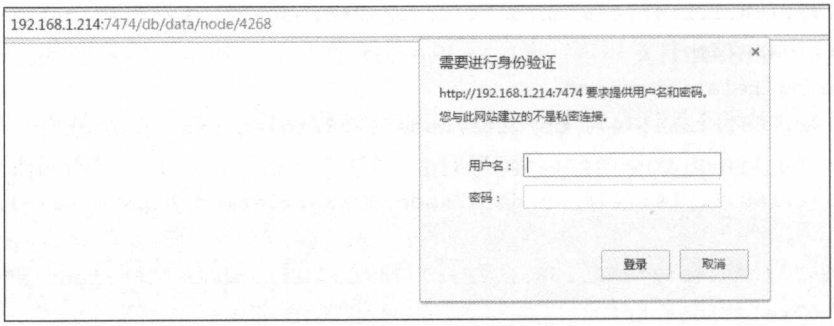


图 4-12　RESTful 调用身份认证

认证通过后，即可打开如图 4-13 所示的查询界面。

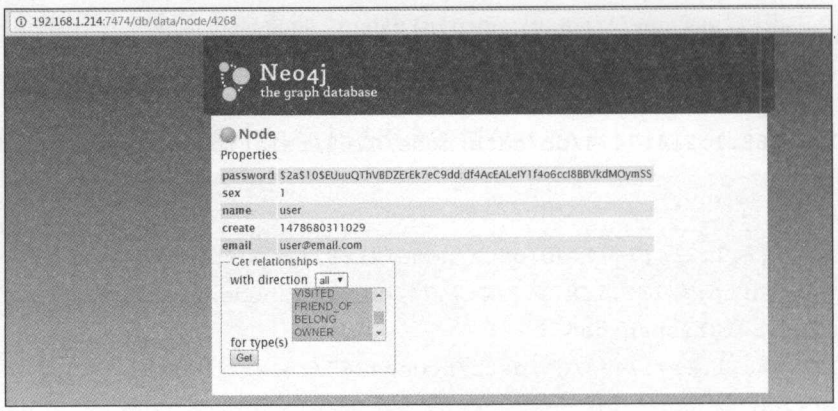


图 4-13　查询结果

有关 Cypher 查询语言的介绍，至此告一段落。

通过上面一些简要说明和操作练习，我们对 Cypher 查询语言有了一个比较全面和系统的了解。当然，对于 Cypher 查询语言，我们还有很多内容没有涉及。限于篇幅，这里并不能对整个 Cypher 查询语言进行面面俱到的讲解和说明，有关 Cypher 查询语言的更多更详细的说明，请参考 Neo4j 的官方文档：http://neo4j.com/docs/developer-manual/current/cypher/。

要熟练掌握和使用 Cypher 查询语言，还必须经过大量的练习和实践。通过实际应用来使用 Cypher 查询语言，是学习和提高的最好方法与途径。

4.13　小结

本章系统地介绍了 Cypher 查询语言的语法结构和特点，并通过一些实例进行了详细的讲解和练习，这些内容包括：

- 节点和关系的创建、修改、删除。
- 标签的创建、使用、删除。
- 索引的创建、使用、删除。
- 约束的创建、使用、删除。
- 查询中循环语句的使用方法。
- 在查询中如何使用参数和集合。
- 在查询中如何设计遍历算法。
- 在查询中如何使用函数。
- 如何使用存储过程。
- 如何对查询语句进行性能分析。
- 怎样在不同领域使用 Cypher 查询语言。

通过使用 Cypher 查询语言的一些练习，不但让我们对 Cypher 查询语言的语法有了

一个比较全面而系统的认识和理解，同时也学习到了在图数据库中一些特殊语法的使用方法。例如，通过标签、索引、约束、遍历、集合、函数等的具体使用，加深了对 Cypher 查询语言的理解程度。Cypher 查询语言虽然结构简单、语法简洁，但却有很强的表现力，可以表现任何复杂的业务逻辑，可以设计出功能丰富并且完善的查询算法。

在后续的章节中，我们将继续使用 Cypher 查询语言，通过一些现实应用场景来探讨 Cypher 查询语言更加精妙的使用方法。

下一章将介绍一个更加实用的工具——SDN（Spring Data Neo4j），这是一个相当优秀的设计，它可以在应用项目中提供更加简单易用的方法来使用 Neo4j 数据库。

第5章

使用SDN建模和设计存储库接口

SDN（Spring Data Neo4j）在2010年就发布了1.0.0版本，现在可以使用4.1.0及以上版本。SDN提供了通用的API接口，就像使用JPA（Java Persistence API）一样简单易用。我们只要使用POJO（Plain Ordinary Java Object，简单的Java对象），利用OGM（Object-Graph Mapping，对象-图映射）就能对Neo4j数据对象进行建模。建模之后，通过使用SDN提供的存储库接口，就能方便地使用标准的CURD方法，像使用JPA访问关系型数据库那样，以一种简单易用的方式来使用图数据库。

5.1 SDN简介

SDN是Spring Data的一个独立子项目，早期版本在性能上可能并不能达到令人满意的效果，所以并未得到很好的推广使用。在SDN 4之后，这种情况得到了很大的改观，并且在连接数据库的驱动方面，从以前只支持HTTP驱动的连接方式，扩展到同时支持嵌入式驱动和Bolt驱动等连接方式，这给开发者在使用数据库时增加了很大的灵活性。

5.1.1 SDN的特点

像JPA使用ORM一样，SDN使用OGM（对象-图映射）将域对象与图数据进行相互转换。使用这种转换机制，我们在对对象进行建模时，只要使用一些简单的注解，就可以让对象与图数据建立起映射关系。

SDN 也提供了对 OGM 映射的智能管理机制，增强了访问数据库的性能。这主要表现在两个方面：一方面，当一个应用在使用对象并且进行修改的过程中，并不需要直接对数据库进行操作，只有当保存对象时才连接数据库；另一方面，在保存一个对象时，其他与这一对象相关联的对象也能相应地得到保存。

出于存储和性能的考虑，Neo4j 没有日期、枚举等数据类型，但使用 OGM 可以提供这些数据的转换机制。通过这些类型转换，让我们在使用这些数据类型的过程中，并不觉得跟实际具有这些数据类型的数据库有什么区别。

5.1.2 SDN 存储库接口

当一个对象使用 SDN 建模之后，就可以使用 SDN 的存储库接口实现持久化和进行一些数据访问设计。通过简单继承 SDN 的存储库接口，就可以执行标准的 CURD 操作，同时还可以按接口的规范标准声明自定义方法，实现像使用查询语言一样的查询设计。而所有这些方法将由 SDN 智能实现，并不需要我们编写实现的代码。

另一个更具灵活性的设计是：在接口中通过注解使用自定义的 Cypher 查询语句。当然这也是由 OGM 的映射机制实现的，并且通过 OGM 优化处理使查询具有很高的效率，即提供了很好的性能表现。通过 Cypher 查询语言，我们可以设计出复杂的查询，以支持各种各样的业务需求。

SDN 还提供了隐式事务管理机制，即对数据库的每一项操作，包括查找数据、保存数据等，都可以使用隐式事务管理。通过使用隐式事务管理，每一个事务都将是自动提交的，这将不再需要我们编写任何事务管理的代码。当然，通过使用 OGM 的一些简单注解方法，我们也可以非常灵活地按业务需求来显式地管理事务。

SDN 存储库接口的一些标准方法，对于查找数据的路径深度默认都设计为 1，如果有需要，那么我们都能够在使用这些标准方法时指定路径的深度，以在对一个对象的访问中取得更多的关联对象和数据。这跟使用 Neo4j API 时遍历的设计和使用 Cypher 查询语言的关系深度的设定等使用方法是一致的。需要注意的是，使用路径的深度时必须要慎重考虑，因为过长的路径将会影响查询的性能。

如果上面这些设计还不能满足一些特殊的要求，则也可以使用 Neo4jTemplat 或

org.neo4j.ogm.session.Session 来访问数据库。这种访问类似于底层的数据库操作，可以为开发者提供更大、更灵活的设计空间。

5.2　数据模型设计

数据模型设计是数据建模的第一步，因为 Neo4j 不需要模式结构定义，所以使用简单框图就可以为一个项目或应用设计数据模型。创建数据模型之后，就可以使用 SDN 进行数据实体建模和一些数据访问的设计。

开始数据模型设计，一般通过分析业务需求就可以提取出需要建立的节点和关系，然后使用节点和关系画出框图，即可完成数据模型的设计。下面通过两个实例来简要说明数据模型的设计过程。

5.2.1　用户访问控制数据模型

在一个访问控制系统中，它的业务需求可以简单地描述为：怎样控制一个用户的访问权限。即一个用户登录系统后，他对系统的哪些资源具有访问权限。通过分析和结合以往的经验，我们可能需要四个节点，分别是用户、部门、角色和资源；三个关系，分别是隶属、拥有和权限。这样，我们就可以画出如图 5-1 所示的用户访问控制数据模型。

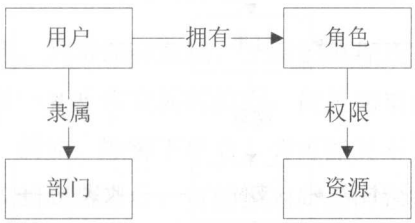


图 5-1　用户访问控制数据模型

这个数据模型是否合理、是否符合业务需求？我们可以用这个简单框图模拟一下业务流程，简单地测试一下它的合理性。首先看看从这个框图中能不能读出类似这样的信息：隶属于一个部门的一个用户拥有哪些角色就能对哪些资源具有访问权限。如果可以，就可以说明这个模型设计是可行的。

很明显，这个数据模型设计的业务流程是通顺的。因为对于这个框图，我们可以这样读出它的流程：部门具有一些隶属用户，用户拥有一些角色，角色对一些资源具有访问权限。

有了这个数据模型之后，就可以对节点和关系进行建模了。在建模中再来确定节点和关系的属性，例如，用户节点可能需要用户名、密码、性别、邮箱、创建日期等属性，同时还要确定关系的对等方式，例如，是一对一、一对多还是多对多等。对于这个实例来说，用户与部门的隶属关系是多对一关系，用户与角色的拥有关系和角色与资源的权限关系都是多对多关系。

5.2.2 购物网站数据模型

如果觉得上面的数据模型简单了一点，那么接下来我们使用一个业务需求比较复杂的实例来试一试，比如一个购物网站。购物网站的业务需求大概具有这样的流程：首先商家上架了商品，然后顾客浏览或查找商品，顾客找到自己需要的商品之后，确定购买，接着使用他的账户支付款项，商家收到货款后，将商品快递给顾客，从而完成一笔交易。根据这个业务流程，我们画出如图 5-2 所示的数据模型。

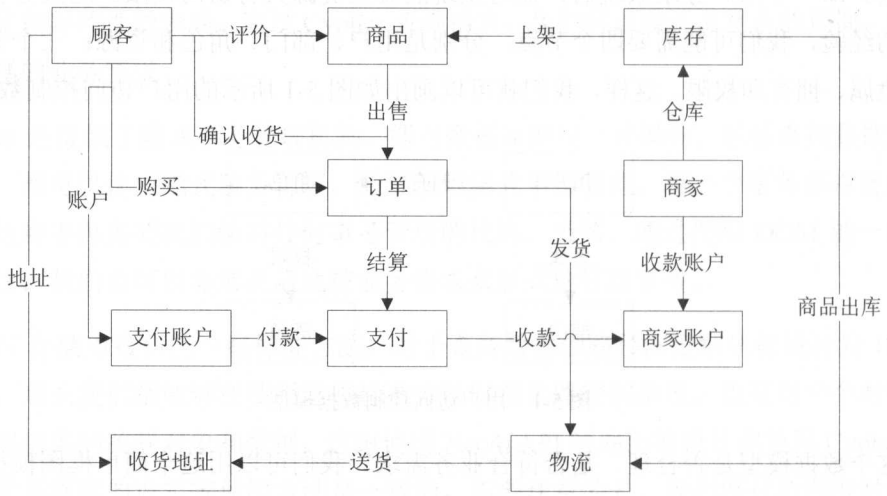


图 5-2 购物网站数据模型

使用这个数据模型，我们同样也可以先测试一下，即看一看它能不能通顺地读出一

个购物网站的基本流程。比如完成一个完整的购物流程，首先是商家的库存要上架商品，然后是顾客购买商品，即商品出售形成订单；接下来是顾客结算订单，使用账户付款，形成支付记录，同时商家账户收到款项，并且订单进入发货状态，同时生成物流记录；这时候，商家的库存办理商品出库，这样商品就通过快递进入送货过程之中；最后顾客从收货地址收到商品，并对订单执行确认收货操作，同时对商品进行评价，至此完成一次购物流程。这就可以说明，这个数据模型所表现的业务流程是通顺的，所以它的设计是合理的。

一般的购物网站还有购物车这一项，以满足顾客一次选购多个商品的需求，所以还必须设计一个购物车，即在上述流程中插入一个挑选商品到购物车的过程，如图 5-3 所示。其中购物车只是顾客与商品的一个关联关系。

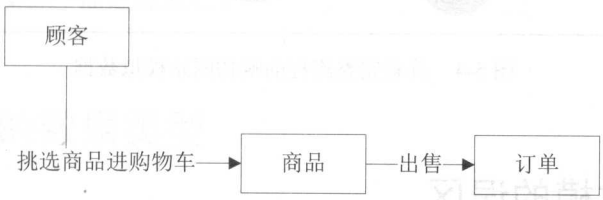


图 5-3　购物网站中购物车数据模型

这下应该很完整了吧？这个模型的整个流程可以通过数据库来表示。如图 5-4 所示，这是一个网上书店的模拟数据。其中“顾客 1”挑选了两本书到他的购物车中，“顾客 2”购买了一本小说，完成了一个完整的购物流程。

不过，如果再仔细想想，则可能会发现，上面的流程还需要更多的细化。比如，上面的数据模型虽然可以表现一个正常交易的流程，但是如果出现不正常的交易情况，那这个数据模型就走不通了。例如，顾客下单后，有可能又不要了，所以，这就需要有撤销订单流程。又如，顾客收到商品之后，可能因为质量问题需要退货和退款，所以，还需要增加相应的退货和退款处理流程。另外，商家售卖的一种商品中还有可能具有型号、颜色、价格和库存数量等不同分类，所以，对于商品节点还有必要进行细分。

不难看出，对上面的数据模型还必须再进行加工和细化。当然，除了这些，还可能有其他各种各样的情况。不过，不管是什么情况，都可以通过简单框图对数据模型进行细化和加工。至于最终怎么建立起一个完整的购物网站数据模型，这里就不再深入探索了。

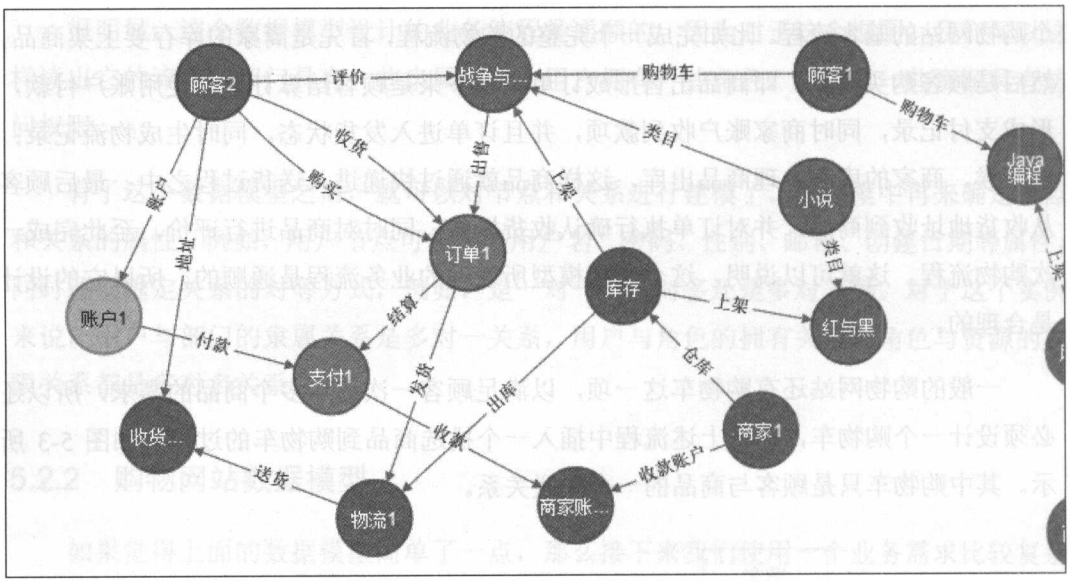


图 5-4　具有完整流程的购物网站模拟数据

5.3　数据建模的误区

Neo4j 数据建模的基本原则是：用节点来表示现实的事物，用关系来表示事物之间的联系。遵循这个基本原则来进行数据建模，将可避免出现错误。这个原则的道理其实很简单，但在实际设计的过程中，还必须注意一些事项，以免陷入数据建模的误区。

在进行数据建模时，必须结合整个系统开发的所有过程进行全盘考虑，要避免因为前期工作的一些疏忽或失误，影响到后期工作的正常开展。

在实际的对象建模过程中，大多数节点的关系都可以不需要属性。没有属性的关系通过节点实体建模就可以创建，即并不需要为关系实体进行单独建模。当且仅当一个关系需要属性时，才必须为其进行实体建模。当然，这并不是说关系不能定义属性，而应该这样考虑：如果一个关系的属性是可有可无的，那就可以不需要。另外，如果关系需要复杂的属性，就要考虑一下它是不是使用节点来表示会更加合理。

例如，在上面的用户访问控制数据模型中，对于隶属关系和拥有关系，也可以设定

它们的属性。比如对于隶属关系，可以设置一个属性，用来指定用户的来源，并说明用户是如何成为这种隶属关系的，是被安排的，还是自己申请加入的等。而对于拥有关系来说，也可以定义一个属性，来说明它是怎么拥有角色的；还可以定义一个属性，来表明什么时候用户拥有了这一角色等。但是，如果这些属性定义实际上并没有什么意义，这样做就完全没有必要了，因为它只会增加编辑和使用这种数据的难度而已。

根据大多数开发者的使用经验，建议创建节点的关系时尽量创建细粒度的关系。比如甲和乙是朋友关系，就不要创建成粗粒度的认识关系，因为如果是认识关系，那么是作为朋友认识的，还是作为同事认识的，就说不清楚了。如果既是同事又是朋友，那么可以为其创建两个关系，更加合乎逻辑。如果这时确实需要一个认识关系，那么也没有问题，可以在朋友关系和同事关系的基础上再创建一个认识关系，以表示他们既不是朋友又不是同事，仅仅有一面之缘而已。

5.4　Neo4j 的数据类型

在进行对象建模之前，我们很有必要需要全面了解一下 Neo4j 的数据类型。Neo4j 的数据类型如表 5-1 所示。

从表 5-1 中可以看出，虽然这些数据类型并不是面面俱到，但却显得精练而完善。比如对于数字类型来说，从字节、短整、整型、长整型到浮点和 double 类型等，应有尽有。正像上面提到的那样，虽然没有日期和枚举等数据类型，但是存储这些类型的数据是完全没有问题的。实践证明，一般应用的数据使用这些数据类型来存储是完全足够的。

对于数据类型的设计，Neo4j 的设计者在这方面的考虑应该是非常严谨的。因为为了提高数据的存取性能，Neo4j 的每一种数据类型都是按照定长的方式来存储的，而且不同类型的数据也是按照不同文件的方式分开存储的。这样对于任何数据的查询，Neo4j 都可以做到精准计算，从而可以提供非常高效的计算能力。

表 5-1 Neo4j 数据类型列表

类 型	描 述	取值范围
boolean	布尔	true/false
byte	8bit 整数	−128 to 127
short	16bit 整数	−32768 to 32767
int	32bit 整数	−2147483648 to 2147483647
long	64bit 整数	−9223372036854775808 to 9223372036854775807
float	32bit IEEE 754 浮点数	
double	64bit IEEE 754 浮点数	
char	16bit 表示 Unicode 字符的无符号整数	u0000 to uffff（0 to 65535）
String	Unicode 字符串	

5.5 在项目中使用 SDN

使用 SDN 给 Neo4j 的数据对象建模是一件非常简单的事情，它不同于关系型数据库，有范式和反范式等方面的思考，一般情况下，我们只需要使用简单框图设计的数据模型就可以为数据对象建模。

5.5.1 在项目工程中引用 SDN 依赖

SDN 的最新版本为 4.1.5，在一个项目工程中要使用 SDN，可以通过 Maven 管理进行配置。如代码清单 5-1 所示是 SDN 的依赖配置。如果使用 Spring Boot 开发框架，则可以不用指定版本号。一般情况下，Spring Boot 1.4.0 使用的 SDN 版本是 4.1.2，Spring Boot 1.4.2 使用的 SDN 版本是 4.1.5。

代码清单 5-1 SDN 依赖配置

```
<dependency>
    <groupId>org.springframework.data</groupId>
```

```
    <artifactId>spring-data-neo4j</artifactId>
    <!--<version>4.1.5.RELEASE</version>-->
</dependency>
```

5.5.2　建模中可用的 OGM 注解

使用 SDN 进行对象建模，可以使用如表 5-2 所示的 OGM 注解来实现。为了便于区分，我们将这些注解划分为两大类，即基本注解和类型转换注解。

表 5-2　OGM 注解列表

类　别	注　解	意　义	作 用 域	备　注
基本注解	@NodeEntity	节点实体	类	
	@GraphId	图标识	属性	必须使用长整型
	@Property	属性	属性	
	@Relationship	关系	属性	
	@Transient	临时字段	属性	不参与持久化
	@RelationshipEntity	关系实体	类	
	@StartNode	开始节点	属性	
	@EndNode	结束节点	属性	
类型转换	@Convert	类型转换	属性	
	@DateLong	日期长整型	属性	
	@DateString	日期字符串	属性	
	@EnumString	枚举字符串	属性	
	@NumberString	数字字符串	属性	

注意，与早期的 SDN 版本相比，当前版本已经不支持@Index 注解。SDN 的开发者认为，索引是数据库管理的事情。所以，如果要创建索引或约束，则可以通过数据库的客户端来创建。

在基本注解类别中，@NodeEntity 表示将一个对象设置成节点实体，@RelationshipEntity

表示将一个对象设置成关系实体。不管是节点实体还是关系实体，都必须在数据库中具有一个唯一标识，即实体的 ID，可以使用注解@GraphId 来设置。而对于一般属性，可以不使用注解，SDN 也能进行智能映射，即对于注解@Property，一般可以省略。另外有一个特别的注解@Transient，表示它所标注的对象属性将不参与持久化，即这个属性只能在对象中使用，数据库中的节点将不包含这个属性。

在类型转换注解类别中，可以将 Neo4j 中没有的数据类型进行转换，如日期类型可以转换为长整型或字符串，分别使用注解@DateLong 和@DateString 来实现；枚举类型可以使用注解@EnumString 转换为字符串来存储等。

5.5.3 日期类型转换实例

日期类型是一个经常使用的数据类型。对于 Neo4j 的日期类型，在进行对象建模时可以使用注解@DateLong 将其转换成长整型进行保存。而在 Web 应用的页面中，使用日期还必须注意其使用格式，这可以使用注解@DateTimeFormat 将日期类型进行格式化处理。例如，对于中文环境来说，可以使用“yyyy-MM-dd HH:mm:ss”格式来显示。日期类型转换实例如代码清单 5-2 所示。

代码清单 5-2 日期类型转换

```
@DateLong
@DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
private Date create;

public Date getCreate() {
        return create;
}

public void setCreate(Date create) {
        this.create = create;
}
```

5.6　使用 SDN 建模

如图 5-5 所示是一个电影社区的数据模型框图。这个数据模型有三个节点，分别是观众节点 Person、电影节点 Movie 和节目节点 Show。观众节点具有三个关系，分别是观众与观众之间的“朋友”关系、观众与节目之间的“观看”关系和观众与电影之间的“评分”关系。节目与电影之间也有一个“放映”关系，说明节目放映的是哪部电影。下面说明怎样对这些节点和关系进行建模。

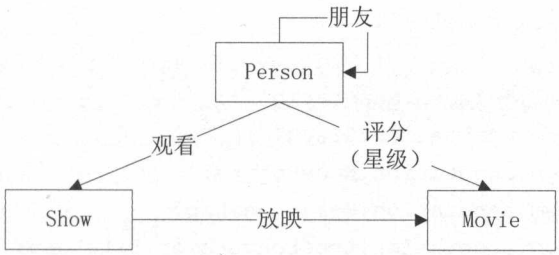


图 5-5　电影社区数据模型

5.6.1　节点建模

从图 5-5 所示的数据模型中可以看出，这里有三个节点需要建模，而在四个关系之中，因为“评分”关系具有属性“星级”，所以这个关系也要单独进行建模。

有关节点的建模，这里我们只选择观众和电影两个实体对象进行说明。

1. 观众节点建模

观众节点建模的实现方法如代码清单 5-3 所示。

程序中，使用注解@NodeEntity 标注类 Person 为一个节点实体，Person 具有 id、name、sex、create 等基本属性。使用注解@GraphId 将属性 id 作为节点的唯一标识，这样它的值将由数据库自动生成。属性 create 表示创建日期，并使用注解@DateLong 将其转换为长整型数据进行存储。

观众节点的三个关系也在这里建立起来，通过使用注解@Relationship 来设定，同时使用 type 指定了关系的类型，使用 direction 指定了关系的方向，并且三个关系都是多对多关系，使用 Java 的数据类型 Set 来引用关系的对象集合。注意，这里的“评分”关系没有指定方向，将默认使用发出方向。在“朋友”关系中还使用了一个 Json 的注解@JsonIgnore，以避免在数据访问中引起递归调用。

在观众节点建模中还为各个关系设计了生成函数。其中，函数 beFriend()用来生成“朋友”关系，函数 addVistiter()用来增加观众“观看”节目的关系，函数 rate()用来增加观众对一部电影进行“评分”的关系。

代码清单 5-3　观众节点建模

```
package com.test.data.domain;

import com.fasterxml.jackson.annotation.*;
import org.neo4j.ogm.annotation.GraphId;
import org.neo4j.ogm.annotation.NodeEntity;
import org.neo4j.ogm.annotation.Relationship;
import org.neo4j.ogm.annotation.typeconversion.DateLong;
import org.springframework.format.annotation.DateTimeFormat;

import java.util.*;

@NodeEntity
public class Person {
        @GraphId
        private Long id;
        private String name;
        private int sex;
        @DateLong
        @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
        private Date create;

        @Relationship(type = "朋友", direction = Relationship.OUTGOING)
        @JsonIgnore
        private Set<Person> friends = new HashSet<>();
```

```
    @Relationship(type = "评分")
    private Set<Rating> ratings = new HashSet<>();

    @Relationship(type = "观看", direction = Relationship.OUTGOING)
    private Set<Show> visiters = new HashSet<>();

    public void beFriend(Person person) {
            friends.add(person);
    }

    public void addVistiter(Show show){
            visiters.add(show);
    }

    public Rating rate(Movie movie, int stars, String comment) {
            Rating rating = new Rating(this, movie, stars, comment);
            ratings.add(rating);
            return rating;
    }
    ...
}
```

2. 电影节点建模

电影节点建模的实现方法如代码清单 5-4 所示。

程序中，使用注解@NodeEntity 设定类 Movie 为一个电影实体，Movie 只有两个属性，分别为 id 和 name，以及一个“评分”关系。注意，这里将“评分”关系的方向设定为进入方向。

代码清单 5-4　电影节点建模

```
package com.test.data.domain;

import org.neo4j.ogm.annotation.GraphId;
import org.neo4j.ogm.annotation.NodeEntity;
import org.neo4j.ogm.annotation.Relationship;
```

```
import java.util.HashSet;
import java.util.Set;

@NodeEntity
public class Movie{
    @GraphId
    private Long id;
    private String name;
    @Relationship(type = "评分", direction = Relationship.INCOMING)
    private Set<Rating> ratings = new HashSet<>();

    public Movie() {
    }
    ...
}
```

5.6.2 关系建模

除“评分”关系之外，电影社区数据模型中其他几个关系的建模已经在节点建模时实现了。“评分”关系因为本身具有属性，所以必须为其进行单独建模，如代码清单 5-5 所示。

程序中，注解@RelationshipEntity 指定类 Rating 是一个关系实体，并用 type 设定了关系的类型为“评分”。关系实体必须有开始节点和结束节点。程序中使用注解@StartNode 设定了“评分”关系的开始节点为 Person，使用注解@EndNode 设定了“评分”关系的结束节点为 Movie。其中，@JsonBackReference 注解是一个防止递归调用的设置。

函数 Rating(Person person, Movie movie, int stars, String comment)是“评分”关系的生成函数，当观众对一部电影进行评分时，将由观众对象对这个函数进行调用。

代码清单 5-5 评分关系建模

```
package com.test.data.domain;

import com.fasterxml.jackson.annotation.JsonBackReference;
import org.neo4j.ogm.annotation.EndNode;
import org.neo4j.ogm.annotation.GraphId;
```

```
import org.neo4j.ogm.annotation.RelationshipEntity;
import org.neo4j.ogm.annotation.StartNode;
import org.neo4j.ogm.annotation.typeconversion.DateLong;
import org.springframework.format.annotation.DateTimeFormat;

import java.util.Date;

@RelationshipEntity(type = "评分")
public class Rating{
    @GraphId
    private Long id;
    @StartNode
    @JsonBackReference
    private Person person;
    @EndNode
    @JsonBackReference
    private Movie movie;
    private int stars;
    private String comment;
    @DateLong
    @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    private Date create;

    public Rating() {}

    public Rating(Person person, Movie movie, int stars, String comment) {
        this.person = person;
        this.movie = movie;
        this.stars = stars;
        this.comment = comment;
    }
    ......
}
```

5.7 使用 SDN 设计存储库接口

对象建模完成之后，接下来的工作就是怎么实现实体对象的持久化和进行一些数据访问的设计。使用 SDN 来做这项工作，只要定义一个存储库接口，并通过继承 SDN 的 GraphRepository 接口，就可以轻松实现实体对象的持久化，并且同时拥有一些数据访问的方法。

5.7.1 创建存储库接口

对于上面完成建模的观众对象来说，我们可以为其创建一个观众实体对象的存储库接口，就可以实现观众对象的持久化设计，如代码清单 5-6 所示。观众实体存储库接口 PersonRepository 只是简单地继承了 SDN 的存储库接口 GraphRepository，就可以拥有观众实体的增、删、改、查方法，以及使用分页查询等方法。

代码清单 5-6 观众存储库接口定义

```
package com.test.data.repository;

import com.test.data.domain.Person;
import org.springframework.stereotype.Repository;

@Repository
public interface PersonRepository extends GraphRepository<Person> {

}
```

为什么只是简单地继承 GraphRepository 接口，就能够拥有访问数据的一般方法呢？这要看看 GraphRepository 接口是怎么定义的，也许就什么都清楚了。

从 GraphRepository 接口的源代码中可以看出，它不但本身已有一些声明方法，而且有其自己的继承关系，它的继承关系如图 5-6 所示。

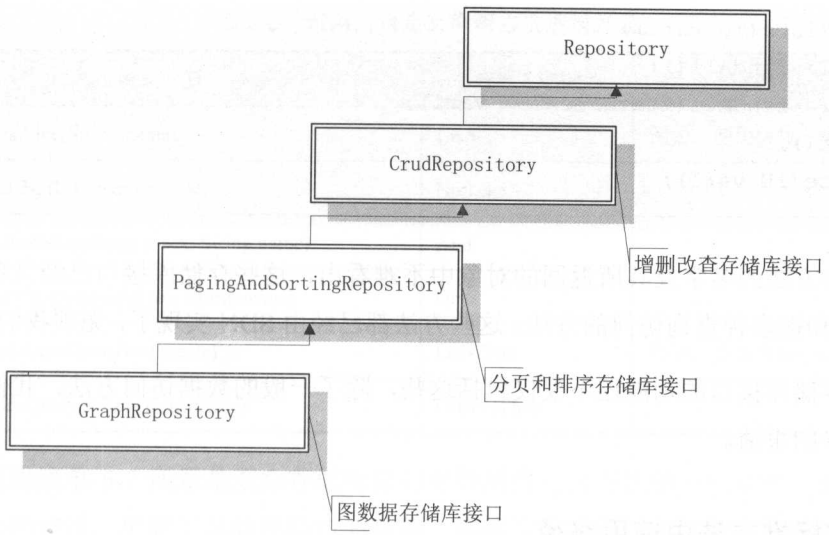


图 5-6　图存储库接口继承关系

从这个继承关系中可以看出，它所继承的这些接口中，除 Repository 接口之外，也都声明了一些访问数据库的方法。其中，PagingAndSortingRepostitory 接口声明了分页查询和排序的方法，CrudRepository 声明了 CURD 的基本方法。这就不难理解，为什么我们在观众实体存储库接口中只是简单地继承了 GraphRepository 接口，就一下子拥有了对观众实体进行增、删、改、查的所有方法了。

图 5-6 所示的这些存储库接口所声明的方法，我们可以列举一些来看看，大致如下：

```
Iterable<T> findAll();
Iterable<T> findAll(int var1);
Iterable<T> findAll(Sort var1);
Iterable<T> findAll(Sort var1, int var2);
Iterable<T> findAll(Iterable<Long> var1);
Iterable<T> findAll(Iterable<Long> var1, int var2);
Iterable<T> findAll(Iterable<Long> var1, Sort var2);
Iterable<T> findAll(Iterable<Long> var1, Sort var2, int var3);
Page<T> findAll(Pageable var1);
Page<T> findAll(Pageable var1, int var2);
<S extends T> S save(S var1);
<S extends T> Iterable<S> save(Iterable<S> var1);
T findOne(ID var1);
```

```
boolean exists(ID var1);
Iterable<T> findAll();
Iterable<T> findAll(Iterable<ID> var1);
long count();
void delete(ID var1);
.....
```

从这些方法的名字及其所返回的对象中不难看出，这些存储库接口已经具备了增、删、改、查和很多种查询访问的方法。这些方法都已经由 SDN 实现了，无须我们来实现。

使用存储库接口的功能还不仅仅包括这些，除了一般的数据访问方法，我们还可以用它做更多的事情。

5.7.2 在标准方法中使用路径

存储库接口 GraphRepository 中提供的一些标准查询方法的默认路径都是 1，如果有需要，那么我们随时可以指定访问的路径深度。当然，这必须要有所控制，因为过长的深度将会耗费昂贵的资源，从而影响访问数据库的性能。

例如，下列代码指定了路径的深度为 2，这意味着在返回的 Person 对象中，除了直接关联的数据，如观众的朋友、观众观看的节目和观众评分的电影等，还包含其他间接关联的数据，如观众的朋友的朋友、观众观看的节目中所放映的电影等。我们在实际使用中正是使用这样的查询来返回观众的节点数据中已经有“观看”关系但还未建立“评分”关系的那些电影节点，因为这些电影节点与观众之间隔着两层关系，即包含“观看”和“放映”两个关系。

```
Person person = personRepository.findOne(id, 2);
```

5.7.3 自定义声明方法

继承了 SDN 的存储库接口之后，就可以根据 Cypher 查询语言的规则，使用查询关键字自定义一些声明方法。如表 5-3 所示是一些通过测试验证的可用的自定义声明方法的规则，我们可以参照这些规则来声明一些自定义方法。

表 5-3　可用的自定义声明方法规则列表

实　例	查询关键字	备　注
findByNameLike(String name)	Like	例如，使用参数："*众*"
findByNameNotLike(String name)	NotLike	
findBySexAndName(Integer sex, String name)	And	
findBySexOrName(Integer sex, String name)	Or	
findByCreateLessThan(Date create)	LessThan	例如，使用参数：new Date()
findByCreateGreaterThan (Date create)	GreaterThan	

如代码清单 5-7 所示是观众存储库接口中使用自定义方法的一些实例。通过这些自定义方法的使用，扩展了存储库接口的功能。需要注意的是，在这些方法的参数中，我们大多省略了注解@Param 的使用。如果这些方法要提供给 RESTful 进行调用，则必须每个参数都使用注解@Param 进行设置，否则就有可能提示找不到输入参数的错误。

代码清单 5-7　使用自定义方法实例

```
package com.test.data.repository;

import com.test.data.domain.Person;
import org.springframework.data.neo4j.annotation.Query;
import org.springframework.data.neo4j.repository.GraphRepository;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

@Repository
public interface PersonRepository extends GraphRepository<Person> {
    Person findByName(String name);

    Iterable<Person> findByNameLike(String name);

    Iterable<Person> findByCreateLessThan(Date create);
}
```

如果在自定义方法中是通过注解@Query 的方式使用 Cypher 查询语句来实现的，则

不受这个规则的限制，完全可以按自己的喜好来定义方法的名字。不过为了程序的可读性考虑，应尽量按照这个规则来定义方法的名字，如下：

```
@Query("MATCH (n:Person) WHERE ID(n) <> {id} RETURN n;")
Iterable<Person> findByIdNot(@Param("id") Long id);
```

5.7.4 使用底层方法

在实际应用开发中，业务需求常常是很复杂的，上面使用 SDN 设计的一些查询方法有时还不能满足业务需求，这时候该怎么办呢？不用担心，SDN 也提供了一些底层调用的使用方法，能给数据访问提供更加灵活的设计空间。例如，可以使用 org.neo4j.ogm.session.Session 来访问数据库，这个 Session 由 Neo4jSession 所实现，提供了很多 load()、loadAll()、query()等方法。

例如，我们定义了一个通用的分页查询类，如代码清单 5-8 所示，就是使用 Session 的 loadAll()方法进行调用的。这个通用类还具有如下一些特点：

（1）使用泛型 Class<T>，表示它可以被任何实体调用，所以具有通用性。

（2）使用 Pageable 接口，可以传递分页参数和排序字段。

（3）使用 Filters 来传递输入参数，这可以根据需要传入任何参数。

代码清单 5-8 分页查询通用类

```
package com.test.data.service;
......
@Service
public class PagesService<T> {
    @Autowired
    private Session session;

    public Page<T> findAll(Class<T> clazz, Pageable pageable, Filters
filters){
        Collection data = this.session.loadAll(clazz, filters, convert
(pageable.getSort()), new Pagination(pageable.getPageNumber(),
pageable.getPageSize()), 1);
        int count = this.session.loadAll(clazz, filters, 1).size();
```

```
        return updatePage(pageable, new ArrayList(data), count);
    }

    private Page<T> updatePage(Pageable pageable, List<T> results, int total)
{
        return new PageImpl(results, pageable, (long)total);
    }

    private SortOrder convert(Sort sort) {
        SortOrder sortOrder = new SortOrder();
        if(sort != null) {
            Iterator var3 = sort.iterator();

            while(var3.hasNext()) {
                Sort.Order order = (Sort.Order)var3.next();
                if(order.isAscending()) {
                    sortOrder.add(new String[]{order.getProperty()});
                } else {
                    sortOrder.add(SortOrder.Direction.DESC, new
String[]{order. getProperty()});
                }
            }
        }
        return sortOrder;
    }
}
```

使用这个通用类的例子如代码清单 5-9 所示，目的是为观众节点实现灵活的分页查询设计。

程序中通过 Pageable 设定了页码和页大小，还通过 Sort 设定了排序字段，并通过 Filters 对象使用节点的属性字段作为查询参数。其中，属性 name 作为名称的模糊查询条件；属性 create 使用日期参数作为查询条件，在这个条件设定中，GREATER_THAN 设定查询的日期大于参数提供的日期进行查询，即只能查询指定日期之后的数据。而这些条件的使用还是可选的，即传入参数非空时才启用这个参数作为查询条件。

代码清单 5-9 使用分页通用类实例

```
package com.test.data.service;
......
@Service
@Transactional
public class PersonService {
    @Autowired
    private PersonRepository personRepository;
    @Autowired
    private PagesService<Person> personPagesService;

    public Page<Person> findPage(PersonQo personQo){
        Pageable pageable = new PageRequest(personQo.getPage(),
personQo. getSize(), new Sort(Sort.Direction.ASC, "id"));

        Filters filters = new Filters();
        if (!StringUtils.isEmpty(personQo.getName())) {
            Filter filter = new Filter("name", "*"+personQo.getName()+"*");
            filter.setComparisonOperator(ComparisonOperator.LIKE);
            filters.add(filter);
        }
        if (!StringUtils.isEmpty(personQo.getCreate())) {
            Filter filter = new Filter("create", personQo.getCreate().getTime());
            filter.setComparisonOperator(ComparisonOperator.GREATER_THAN);
            filter.setBooleanOperator(BooleanOperator.AND);
            filters.add(filter);
        }
        return personPagesService.findAll(Person.class, pageable, filters);
    }
}
```

5.8 SDN 配置

在一个项目工程中，为了让我们使用 SDN 建模的实体对象以及使用 SDN 定义的一

些存储库接口能够生效，还必须对 SDN 进行一些相关的配置，同时也需要使用 SDN 的一种驱动方式来配置连接 Neo4j 数据库的一些参数。在实现了这些配置之后，才能在一个项目工程中正常使用 SDN。

5.8.1　配置域对象和存储库接口

如果使用 Spring Boot 开发，则使用 SDN 的配置是很方便的。如代码清单 5-10 所示定义了一个配置类 Neo4jConfig 的实现方法。

这个配置类继承了 SDN 的配置类 Neo4jConfiguration，这样只要通过简单地重写 Neo4jConfiguration 的 SessionFactory 就可以加载域对象。程序中使用注解@EnableNeo4j Repositories 来激活我们设计的存储库接口。其中，注解@EnableTransactionManagement 激活了 SDN 的隐式事务管理。

代码清单 5-10　SDN 配置实例

```
package com.test.data.config;
......
@Configuration
@ComponentScan(basePackages = "com.test.data.services")
@EnableNeo4jRepositories(basePackages = { "com.test.data.repositories" })
@EnableTransactionManagement
public class Neo4jConfig extends Neo4jConfiguration {

    @Override
    public SessionFactory getSessionFactory() {
        return new SessionFactory("com.test.data.domain");
    }

}
```

5.8.2　使用 SDN 驱动连接数据库

SDN 可以使用如下三种驱动方式中的任何一种来连接数据库：

- HTTP 驱动。

- 嵌入式驱动。
- Bolt 驱动。

而连接数据库使用的配置文件是 ogm.properties，对于上面三种驱动方式的配置如代码清单 5-11 所示。这里我们使用的是 HTTP 驱动方式，而其他两种驱动方式都被注释掉了。如果使用嵌入式的驱动方式配置，则可以不需要用户名和密码。

代码清单 5-11　连接数据库配置实例

```
compiler=org.neo4j.ogm.compiler.MultiStatementCypherCompiler
#Http
driver=org.neo4j.ogm.drivers.http.driver.HttpDriver
URI=http://192.168.1.211:7474

username = neo4j
password = 12345678

#Bolt
#driver=org.neo4j.ogm.drivers.bolt.driver.BoltDriver
#URI=bolt://192.168.1.214

##Embedded
#driver=org.neo4j.ogm.drivers.embedded.driver.EmbeddedDriver
#URI=file:///test/graph.db
```

上面配置的 HTTP 驱动是 SDN 默认的驱动方式，如果要使用其他两种驱动方式，还必须加载其相应的驱动程序，即依赖包。如果使用 Maven 管理项目，则可以使用如代码清单 5-12 所示的方式引用依赖包。

代码清单 5-12　引用 Neo4j 驱动依赖包配置

```
<dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j-ogm-embedded-driver</artifactId>
    <version>2.0.5</version>
</dependency>
<dependency>
    <groupId>org.neo4j</groupId>
```

```
    <artifactId>neo4j-ogm-bolt-driver</artifactId>
    <version>2.0.5</version>
</dependency>
```

5.9　小结

本章简要介绍了 SDN 的历史及其发展情况，并详细说明了 SDN 的特点和优势。

SDN 是 Java 开发者，特别是 Spring Boot 开发者在应用工程中使用 Neo4j 数据库的一个实用而优秀的工具组件。使用 SDN 将为开发者提供更加简单易用的方法来访问 Neo4j 数据库。

Neo4j 的数据建模是非常容易的，使用 SDN 来进行数据建模更加容易实现。并且 SDN 的存储库接口为开发者提供了相当丰富的功能，使用这些功能可以轻松地实现访问数据库的任何业务需求。

总之，在一个项目工程中使用 SDN，不但可以省略很多工作，而且还能使用非常丰富的功能和方法实现访问 Neo4j 数据库的设计。而更加难能可贵的是，使用 SDN 还能保持使用 Neo4j 高性能的优势。

在后续章节中，我们将结合一两个具体的应用工程实例，进一步说明 SDN 在使用 Neo4j 的开发实践中的优秀表现，并从中深刻地体会使用 SDN 的优势。

第6章

应用实例一：NBA季后赛预测

在 Neo4j 的官方网站中收集了由众多开发者提供的一些非常精彩的开发案例——GraphGists，这里有很多非常有价值的查询算法设计，这是学习使用 Neo4j 的极好资源。这些 GraphGists 都具有实际连接的数据库，可以在线进行学习和操作。为了加深对 GraphGists 的理解，现在我们挑选一个 GraphGists 案例——NBA 季后赛预测，将它转化到我们的工程之中来使用。如果想要了解更多有关这个 GraphGists 的功能，则可以访问 https://neo4j.com/graphgist/5740344d-8596-404b-9fa1-affdc78cabbc，如图 6-1 所示。

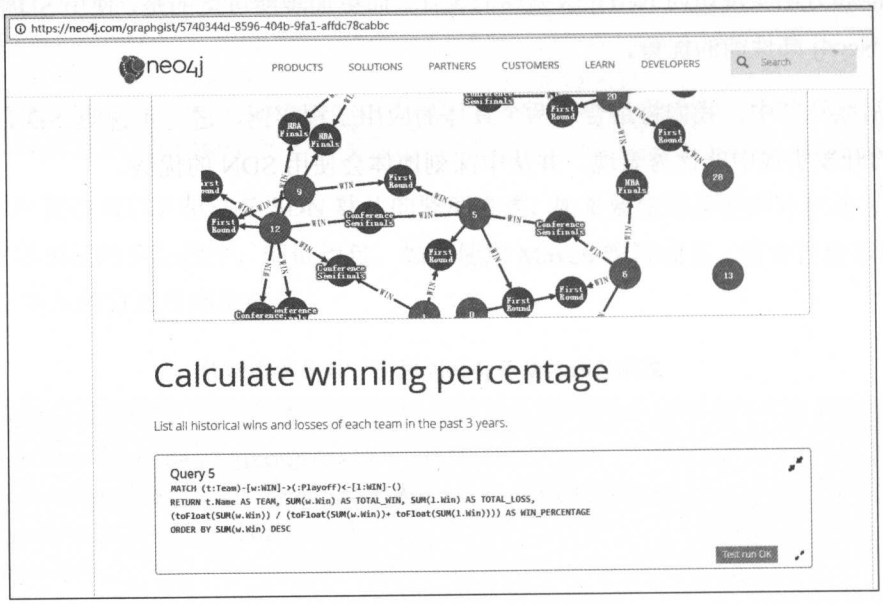


图 6-1　NBA 季后赛预测案例

6.1　应用背景分析

正像这个 GraphGists 本身所介绍的那样，每年的 NBA 季后赛都吸引了全世界数十亿的观众，并且每场比赛的输赢也为博彩公司创造了非常可观的营业利润。那么，博彩公司是否会有什么秘密算法来计算每场比赛的赔率呢？虽然这些赌博的输赢跟我们没有什么关系，但我们关心的是，对于这些球队，在这些比赛当中，有哪些秘密算法可以预测每场比赛的胜负呢？

6.1.1　胜负预测的依据

也许利用球队以往比赛的一些历史数据进行分析和比较，就可以看出或者预测两支球队在下一场比赛中谁的获胜概率会比较大一些，而这个 GraphGists 正是这样做的。虽然这个实例并没有从更多的层面进行更加细致的比较和综合分析，如球队的组成情况、球员的背景等信息，可以说它的预测并不十分准确，但是不管怎么样，对于这些算法的设计和运用，可以为我们学习使用 Neo4j 提供一个很好的实例。

6.1.2　NBA 季后赛数据模型

如果将 NBA 季后赛预测的 GraphGists 中生成测试数据的查询语句在我们的数据库中运行，则将生成如图 6-2 所示的数据。

我们可以从这些生成数据的查询语句中提取对象，建立数据模型。

下列代码是从这些查询语句中挑选出来的一些片断。

```
//create nba TEAM nodes
CREATE (BOS:Team:E{name: "Boston", code: "BOS"})
CREATE (DEN:Team:W{name: "Denver", code: "DEN"})
......
//create PLAYOFF nodes
CREATE (P201501:Playoff{year: "2015", round: "First round"})
CREATE (P201502:Playoff{year: "2015", round: "First round"})
```

```
......
//create PLAYOFF relationships between teams
CREATE (ATL)-[:WIN{win:4}]->(P201501)
CREATE (BKN)-[:WIN{win:2}]->(P201501)
......
```

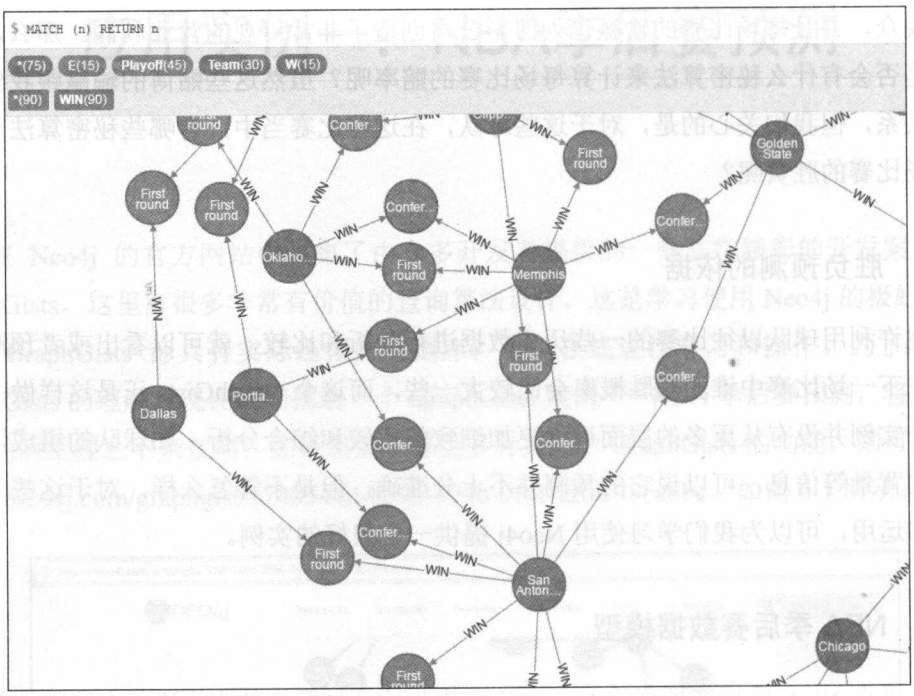


图 6-2　NBA 季后赛测试数据

分析这些创建节点和关系的查询语句，可以提取出节点和关系对象，然后建立这个案例的数据模型，如图 6-3 所示。

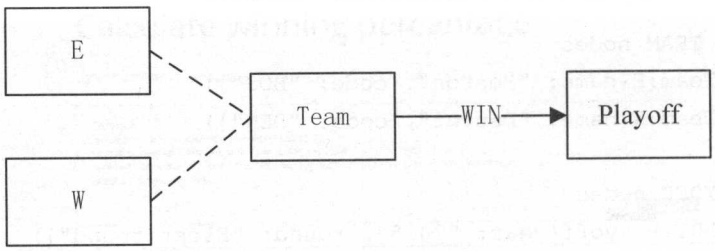


图 6-3　NBA 季后赛数据模型

这个数据模型有球队和比赛两个节点，球队和比赛之间有一个赢得关系。

其中，Team 为球队节点，Playoff 为比赛节点，WIN 为赢得关系。球队节点又细分为东部球队和西部球队，分别用标签 E 和 W 表示。

从对查询语句的分析中可以得出各个节点和关系的属性设置。

球队节点具有两个属性：name 和 code，分别表示球队名称和代码。比赛节点有两个属性：year 和 round，分别表示比赛的年代和回合。赢得关系的类型为 WIN，它有一个属性 win，用来表示在一轮比赛中赢得的场次。

这里顺便说明一下 NBA 季后赛的比赛规则。季后赛分首轮比赛、东西部半决赛、东西部决赛和总决赛四轮比赛，每轮比赛采用七场四胜制，即首先赢得四场比赛的球队为胜。这样，其中的比赛结果就有可能出现这样的比分情况：4∶3，4∶2，4∶1，4∶0。所以，数据模型中的赢得关系 WIN，只要记录比赛两队在每轮比赛中各自赢得的场次即可，赢得四场比赛的为胜方，其他都为负方。

6.2　实体对象建模

建立了 NBA 季后赛的数据模型之后，就可以使用 SDN 对各个实体对象进行建模。这里需要建模的实体有球队节点实体、比赛节点实体和赢得关系实体。

6.2.1　节点实体建模

节点实体建模包括球队节点实体建模和比赛节点实体建模。

1. 球队节点实体建模

尽管球队又细分为东部球队和西部球队两个分支，但是所有球队的属性是一样的，所以建模时可以用多标签的方式来实现。即将球队的标签设置为 Team，东部和西部球队的标签分别设置为 E 和 W。

在使用 SDN 建模时，可以用类继承的方式来实现节点的多标签设置。所以对于这个

数据模型的球队建模，实际上就需要先建模一个球队实体，然后再使用继承的方法，分别为东、西部球队建模。

球队节点实体建模的实现方法如代码清单 6-1 所示。程序中，使用注解@NodeEntity 设定类 Team 是一个节点实体，它的标签将默认使用类名即 Team 来表示。在属性定义中使用属性 id 作为节点的唯一标识，并使用注解@GraphId 进行设定，这个属性的值将由数据库自动生成；而属性 name 和 code 都使用字符串类型，可以省略注解，即在数据库中仍然使用字符串类型。注解@Relationship 建立了球队与比赛之间的赢得关系，关系类型设定为 WIN，这里省略关系的方向设定，将默认使用发出方向。这个关系是一个一对多关系，用集合类型 Set 来存放多个赢得关系。代码中还有一个 win()函数，用来添加一支球队在一轮比赛中的赢得关系，并设定赢得的场次。

代码清单 6-1　球队节点实体建模

```
package com.test.data.domain;

import org.neo4j.ogm.annotation.GraphId;
import org.neo4j.ogm.annotation.NodeEntity;
import org.neo4j.ogm.annotation.Relationship;

import java.util.HashSet;
import java.util.Set;

@NodeEntity
public class Team {
    @GraphId
    private Long id;
    private String name;
    private String code;
    @Relationship(type = "WIN")
    private Set<Winning> winnings = new HashSet<>();

    public Team() {
    }

    public Winning win(int win,Playoff playoff){
```

```
        Winning winning = new Winning(win, this, playoff);
        winnings.add(winning);
        return winning;
    }
    ......
}
```

东部球队节点实体建模，只要简单地继承球队对象即可，如代码清单 6-2 所示。程序中，在类的定义中不使用默认的类名称作为标签，而是使用 label = "E"的方式设定东部球队的标签。

代码清单 6-2　东部球队节点实体建模

```
package com.test.data.domain;

import org.neo4j.ogm.annotation.NodeEntity;

@NodeEntity(label = "E")
public class East extends Team {
}
```

西部球队节点实体建模的方法跟东部球队节点实体一样，不同的只是将标签设定为西部球队的标签 W，如代码清单 6-3 所示。

代码清单 6-3　西部球队节点实体建模

```
package com.test.data.domain;

import org.neo4j.ogm.annotation.NodeEntity;

@NodeEntity(label = "W")
public class West extends Team{
}
```

2. 比赛节点实体建模

比赛节点实体建模的实现方法如代码清单 6-4 所示。程序中使用注解@NodeEntity 设定类 Playoff 是一个节点实体，其中标签没有设定，将默认使用类名 Playoff 作为这个节点的标签。在属性设定中只有对节点的唯一标识，即属性 id 使用注解@GraphId 进行

设定，其他属性都使用了默认的方式，即使用 Java 的数据类型作为数据库的数据类型。

代码清单 6-4　比赛节点实体建模

```
package com.test.data.domain;

import org.neo4j.ogm.annotation.GraphId;
import org.neo4j.ogm.annotation.NodeEntity;

@NodeEntity
public class Playoff {
    @GraphId
    private Long id;
    private String year;
    private String round;

    public Playoff() {
    }
    ......
}
```

6.2.2　关系实体建模

在季后赛数据模型中，因为赢得关系具有属性，所以不能在节点实体建模中实现，必须为其进行单独建模。赢得关系实体建模的实现方法如代码清单 6-5 所示。程序中使用注解@RelationshipEntity 设定类 Winning 为一个关系实体，并将关系类型设定为 WIN。在属性中使用 id 作为关系的唯一标识，并使用注解@GraphId 进行设定，属性 win 将默认使用 Java 的数据类型作为数据库的数据类型，所以没有进行特别的设置。在关系实体建模中，必须指定一个关系的开始节点和结束节点。程序中使用注解@StartNode 设定赢得关系的开始节点为球队对象 Team，使用注解@EndNode 设定赢得关系的结束节点为比赛对象 Playoff。其中，注解@JsonBackReference 用来防止数据的递归调用。函数 Winning() 使用传入的参数，即赢得比赛的场次 win、球队对象 team 和比赛对象 playoff，创建一支球队在一轮比赛中的赢得关系。

代码清单 6-5　赢得关系实体建模

```
package com.test.data.domain;

import com.fasterxml.jackson.annotation.JsonBackReference;
import org.neo4j.ogm.annotation.EndNode;
import org.neo4j.ogm.annotation.GraphId;
import org.neo4j.ogm.annotation.RelationshipEntity;
import org.neo4j.ogm.annotation.StartNode;

@RelationshipEntity(type = "WIN")
public class Winning {
    @GraphId
    private Long id;
    private int win;
    @StartNode
    @JsonBackReference
    private Team team;
    @EndNode
    @JsonBackReference
    private Playoff playoff;

    public Winning() {
    }

    public Winning(int win, Team team, Playoff playoff) {
        this.win = win;
        this.team = team;
        this.playoff = playoff;
    }
    ......
}
```

6.3　实体持久化和查询设计

完成对象建模之后，就可以实现实体的持久化和进行一些查询设计。使用 SDN，可以通过定义一个实体的存储库接口来完成这些工作。

对于球队实体对象来说，因为需要使用东、西部不同的标签来分别管理东、西部球队，所以必须分别为东、西部实体对象各定义一个存储库接口。

另外，比赛实体对象和赢得关系实体对象都必须定义存储库接口，以实现实体的持久化和方便实现对实体的数据进行管理的一些设计。

6.3.1 东部球队存储库接口

东部球队存储库接口的设计如代码清单 6-6 所示。接口 EastTeamRepository 继承了 SDN 的存储库接口 GraphRepository。使用这个接口，就可以实现东部球队实体的持久化，在用它创建节点时，将会同时创建两个标签，分别为 E 和 Team。

程序中的声明方法 findEast()将返回东部球队的一个分页列表数据，这个方法使用注解@Query 设定了使用查询语句来实现数据的访问。其中，查询语句使用 Cypher 查询语言设计，查询语句中的过滤条件 WHERE 中的表达式“t.name =~ ('(?i).*'+{name}+'.*')”使用了正则表达式运行符“=~”，实现了使用节点的属性 name 进行模糊查询的方法。另外，使用关键字 SKIP 和 LIMIT 实现了分页查询的设计。程序中的另一个声明方法 findEastCount()用来在分页查询中返回记录的总数。

代码清单 6-6 东部球队存储库接口

```
package com.test.data.repository;

import com.test.data.domain.East;
import org.springframework.data.neo4j.annotation.Query;
import org.springframework.data.neo4j.repository.GraphRepository;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import java.util.Set;

@Repository
public interface EastTeamRepository extends GraphRepository<East> {
    @Query("MATCH (t:E) WHERE t.name =~ ('(?i).*'+{name}+'.*') " +
            "RETURN t ORDER BY t.name SKIP {skip} LIMIT {limit}")
    Set<East> findEast(@Param("name") String name, @Param("skip") Integer
```

```
skip, @Param("limit") Integer  limit);

    @Query("MATCH (t:E) WHERE t.name =~ ('(?i).*'+{name}+'.*') " +
           "RETURN COUNT(t) as count")
    Integer findEastCount(@Param("name") String name);
}
```

6.3.2　西部球队存储库接口

西部球队存储库接口的设计与东部球队存储库接口基本上相同，只是其中使用的实体对象不同而已，如代码清单 6-7 所示。

代码清单 6-7　西部球队存储库接口

```
package com.test.data.repository;

import com.test.data.domain.West;
import org.springframework.data.neo4j.annotation.Query;
import org.springframework.data.neo4j.repository.GraphRepository;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import java.util.Set;

@Repository
public interface WestTeamRepository extends GraphRepository<West> {
    @Query("MATCH (t:W) WHERE t.name =~ ('(?i).*'+{name}+'.*') " +
           "RETURN t ORDER BY t.name SKIP {skip} LIMIT {limit}")
    Set<West> findWest(@Param("name") String name, @Param("skip") Integer
skip, @Param("limit") Integer  limit);

    @Query("MATCH (t:W) " +
           "WHERE t.name =~ ('(?i).*'+{name}+'.*') " +
           "RETURN COUNT(t) as count")
    Integer findWestCount(@Param("name") String name);
}
```

6.3.3 比赛存储库接口

比赛存储库接口的定义如代码清单 6-8 所示。程序中声明了几个使用 Cypher 查询语言的方法，分别说明如下。

- 方法 findPlayoffByTeamName()：使用球队的名称作为参数，用来查询一支球队可以参与的比赛（这里指可以为球队录入赢得比赛场次的比赛，因为一场比赛如果已经分别给两支球队录入了比赛积分，就不能给其他球队使用了）。在查询设计中使用关键字 UNION 联合了两个查询结果。前一个查询使用过滤条件“Not (p)<-[:WIN]-()”，返回所有没有球队参与的比赛；后一个查询使用过滤条件“count < 2 AND NOT {name} IN team”，返回参与的球队数量小于 2，并且球队名称不等于参数中球队名称的比赛。
- 方法 findPlayoff()：返回没有球队参与的所有可用的比赛。
- 另一个 findPlayoff()方法：使用年代和分页参数，以分页方式返回比赛列表，并可按比赛年代进行模糊查询。
- 方法 findPlayoffCount()：用于分页查询之中，用来返回比赛的记录总数。

代码清单 6-8 比赛存储库接口

```
package com.test.data.repository;

import com.test.data.domain.Playoff;
import org.springframework.data.neo4j.annotation.Query;
import org.springframework.data.neo4j.repository.GraphRepository;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import java.util.Set;

@Repository
public interface PlayoffRepository extends GraphRepository<Playoff> {
    //一支球队可参与的比赛
    @Query("MATCH (p:Playoff) WHERE Not (p)<-[:WIN]-() RETURN p UNION " +
            "MATCH (p:Playoff)<-[:WIN]-(t:Team)  WITH count(t) as count,
```

```
COLLECT(t.name) AS team, p " +
          "WHERE count < 2 AND NOT {name} IN team RETURN p")
    Iterable<Playoff> findPlayoffByTeamName(@Param("name") String name);

    //还没有球队参与的比赛
    @Query("MATCH (p:Playoff) WHERE Not (p)<-[:WIN]-() RETURN p")
    Iterable<Playoff> findPlayoff();

    //分页比赛列表
    @Query("MATCH (p:Playoff) WHERE p.year =~ ('(?i).*'+{year}+'.*') " +
          "RETURN p ORDER BY p.year DESC SKIP {skip} LIMIT {limit}")
    Set<Playoff> findPlayoff(@Param("year") String year, @Param("skip")
Integer  skip, @Param("limit") Integer  limit);

    //比赛总数
    @Query("MATCH (p:Playoff) WHERE p.year =~ ('(?i).*'+{year}+'.*') " +
          "RETURN COUNT(p) AS count")
    Integer findPlayoffCount(@Param("year") String year);
}
```

6.3.4　赢得关系存储库接口

按理说，关系实体并不需要为其设计存储库接口，因为当节点创建了关系之后，在保存节点时，使用 SDN 的智能保存机制就能将关系同时保存下来。创建赢得关系存储库接口是为了在进行数据编辑时方便使用这个接口来删除关系。

赢得关系的存储库接口设计如代码清单 6-9 所示。由于并不需要使用这个接口执行一些数据访问的复杂查询操作，所以这个接口没有声明其他任何方法，我们仅仅需要使用这个接口执行一些简单的 CURD 操作就可以了。

代码清单 6-9　赢得关系存储库接口

```
package com.test.data.repository;

import com.test.data.domain.Winning;
import org.springframework.data.neo4j.repository.GraphRepository;
import org.springframework.stereotype.Repository;
```

```
@Repository
public interface WinningRepository extends GraphRepository<Winning> {
}
```

6.4 预测算法设计

到目前为止，我们只是建立了 NBA 季后赛的数据模型并为其建模，以及进行了实体持久化和一些访问数据的查询设计，还未进行过任何有关预测算法的设计。下面我们开始考虑怎么使用这些数据进行各种预测算法的设计。

像上面的持久化设计一样，这些预测算法的设计也使用一个存储库接口来实现，如代码清单 6-10 所示。程序中使用实体对象 Team 来继承存储库接口 GraphRepository，可以方便返回球队对象，而这里也并不用区分东、西部球队。

NBA 季后赛各种预测算法的设计在 GraphGists 中已经实现了，现在我们要做的是将其转化为应用中能够使用的形式。转化后的方法声明都将使用注解@Query 的方式放入代码清单 6-10 所定义的接口之中。

代码清单 6-10　预测算法存储库接口

```
package com.test.data.repository;

import com.test.data.domain.Team;
import org.springframework.data.neo4j.annotation.Query;
import org.springframework.data.neo4j.repository.GraphRepository;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import java.util.Map;
import java.util.Set;

@Repository
public interface RatioRepository extends GraphRepository<Team> {
  ......
}
```

6.4.1　NBA 季后赛的年度历史查询

NBA 季后赛的年度历史查询设计如代码清单 6-11 所示。程序中使用注解@Query 设定了在接口的声明方法中引用 Cypher 查询语言进行查询。在查询中使用 MATCH 关键字，在匹配模式中使用表达式“(t:Team)-[r:WIN]->(p:Playoff)”，表示将返回球队、赢得关系和比赛的完整数据。由于返回数据由不同对象的属性字段所组成，所以方法 findHistory() 的返回对象使用集合 Map 来对数据进行包装。

代码清单 6-11　NBA 季后赛年度历史查询

```
//年度 NBA 季后赛
@Query("MATCH (t:Team)-[r:WIN]->(p:Playoff) WHERE p.year = {year} " +
       "RETURN t.name AS name, t.code AS code, p.year AS year, p.round AS round,
r.win AS win")
Set<Map<String,Object>> findHistory(@Param("year") String year);
```

6.4.2　一支球队的比赛历史查询

查看一支球队的比赛历史，可以看出一支球队以往的实力及其发展情况，这可为当前的比赛预测提供一定的参考价值。

一支球队的比赛历史查询设计如代码清单 6-12 所示。这个查询算法的设计跟上面年度历史查询的设计有点类似，不同的是使用了球队的名字作为查询条件，并且进行了分页设计。其中，方法 findHistoryByTeamName()返回列表数据，方法 findHistoryByTeamNameCount() 返回记录总数。

代码清单 6-12　一支球队的比赛历史查询

```
//一支球队的历史季后赛
   @Query("MATCH (t:Team {name: {name}})-[w:WIN]->(p:Playoff)<-[l:WIN]-() " +
          "RETURN ID(t) AS id, ID(w) as wid, ID(l) as lid, t.name AS name,
t.code AS code, w.win AS win, l.win AS loss," +
          "p.year AS year, p.round AS round ORDER BY p.year SKIP {skip} LIMIT
{limit}")
   Set<Map<String,Object>> findHistoryByTeamName(@Param("name") String
name,@Param("skip") Integer  skip, @Param("limit") Integer  limit);
```

```
    //一支球队的历史季后赛总数
    @Query("MATCH (t:Team {name: {name}})-[w:WIN]->(p:Playoff)<-[l:WIN]-() " +
            "RETURN COUNT(t) AS count")
    Integer findHistoryByTeamNameCount(@Param("name") String nam);
```

6.4.3 胜负比率排名算法

胜负比率是指在比赛历史中，一支球队比赛获胜的次数除以获胜与失败之和的比率，这个比率越大，说明球队获胜的机会越多。所以，使用胜负比率进行排名，可以为胜负预测提供可靠的参考价值。

胜负比率排名的算法设计如代码清单 6-13 所示。首先，匹配模式中的表达式“(t:Team)-[w:WIN]->(:Playoff)<-[l:WIN]-()”中的两个赢得关系 w 和 l 分别表示胜与负的赢得关系，然后在结果中通过表达式“(SUM(w.win)*1.0 / (SUM(w.win)+ SUM(l.win))) AS percentage”计算出胜负比率，最后使用“ORDER BY SUM(w.win) DESC”进行倒序排序。这样，在返回的列表中，就可以清楚地看出所有球队的历史比赛的胜负比率排名的列表数据。因为是针对所有球队进行查询，所以也使用了分页设计的方法。

代码清单 6-13　胜负比率排名算法设计

```
//计算胜率
@Query("MATCH (t:Team)-[w:WIN]->(:Playoff)<-[l:WIN]-() " +
       "RETURN t.name AS team, SUM(w.win) AS wins, SUM(l.win) AS losses," +
       "(SUM(w.win)*1.0 / (SUM(w.win)+ SUM(l.win))) AS percentage " +
       "ORDER BY SUM(w.win) DESC SKIP {skip} LIMIT {limit}")
Set<Map<String,Object>> findPercentage(@Param("skip") Integer  skip, @Param
("limit") Integer  limit);

//胜率总数
@Query("MATCH (t:Team)-[w:WIN]->(:Playoff)<-[l:WIN]-() WITH DISTINCT t AS p " +
       "RETURN  COUNT(p) AS count")
Integer findPercentageCount();
```

6.4.4　输赢预测算法

如果两支球队正在交战，那么现在最精彩的是怎么对它们孰胜孰负进行预测。这可以从以下几个方面进行考虑：

如果两支球队历史上有过交战，那么可以总结它们历次交战的场次及其输赢的情况，通过这些情况来推测哪支球队更有胜出的可能。

如果两支球队从未有交战记录，那么也没有关系，通过 Neo4j 特有的算法，可以找出它们的所有关系中最接近的关系的一些比赛的输赢情况，从而也可以推断出它们正面交战时有可能出现的输赢情况。

另外，不管两支球队是否有交战记录，我们都可以从其各自的历史数据中计算出它们各自的平均净赢比率，比率越高说明这支球队获胜的概率就越大。所谓平均净赢，指在交战记录或其他历史数据中，一支球队获胜的平均数减去失败的平均数所得的差。

基于上面分析的情况，设计了输赢预测的查询算法，如代码清单 6-14 所示。其中，各个方法的意义说明如下。

- 方法 findWinAndLoss()：可以直接用来处理有交战记录的两支球队的输赢计算。其中，r1 表示交战记录中获胜的赢得关系，r2 表示交战记录中失败的赢得关系。
- 方法 findNeverMetPaths()：用来处理从未有交战记录的两支球队中关系最接近的比赛情况，并将这些数据列举出来。其中，使用所有最短路径函数 AllshortestPaths() 来查询两支球队在所有比赛关系中路径深度为 0～14 中的所有最接近的与其他球队的比赛记录。
- 方法 findAvgNetWin()：用来对正在进行比赛的两支球队，计算出其中一支球队的平均净赢比率。其中，不管两队是否有交战记录，都使用了所有最短路径函数 AllShortestPaths()，列出两队在赢得关系的路径深度为 0～14 的范围中关系最接近的比赛记录，这将包括两队的交战记录，然后使用这些记录计算出球队 t1 所有获胜的平均数减去所有失败的平均数所得的差，这个差值就是球队 t1 平均净赢的数值。算法中使用了列表函数 extract()和 range()来取得列表中的数据，并用数学函数 avg()来计算平均值。

代码清单 6-14 输赢预测查询算法设计

```
//计算输赢
 @Query("MATCH (t1:Team {name:
{t1}})-[r1:WIN]->(p:Playoff)<-[r2:WIN]-(t2:Team {name:{t2}}) " +
        "RETURN p.year AS year, r1.win AS win, r2.win AS loss " +
        "ORDER BY p.year DESC")
 Set<Map<String,Object>> findWinAndLoss(@Param("t1") String t1, @Param("t2")
String t2);

 //赢场比较
 @Query("MATCH (t1:Team {name:{t1}}),(t2:Team {name:{t2}}),\n" +
        "p = AllshortestPaths((t1)-[r:WIN*..14]-(t2))\n" +
        "WITH r,p,extract(r IN relationships(p)| r.win ) AS paths\n" +
        "RETURN paths")
 Set<Map<String,Object>> findNeverMetPaths(@Param("t1") String t1, @Param
("t2") String t2);

 //平均净赢
 @Query("MATCH p= AllShortestPaths((t1:Team {name:
{t1}})-[:WIN*0..14]-(t2:Team {name:{t2}})) " +
        "WITH extract(r IN relationships(p)| r.win) AS RArray, LENGTH(p)-1 AS
s " +
        "RETURN AVG(REDUCE(x = 0, a IN [i IN range(0,s) WHERE i % 2 = 0 | RArray[i] ]
| x + a)) " +
        "- AVG(REDUCE(x = 0, a IN [i IN range(0,s) WHERE i % 2 <> 0 | RArray[i] ]
| x + a)) AS NET_WIN")
float findAvgNetWin(@Param("t1") String t1, @Param("t2") String t2);
```

6.5 SDN 配置及数据库连接

为了让上面一些使用 SDN 的设计能够生效，在项目工程中还必须使用 SDN 配置进行启用。同时，连接数据库的方法也是使用 SDN 的驱动程序来实现的。

6.5.1　数据库连接配置

数据库连接配置使用 SDN 的默认配置文件 ogm.properties 来设定，完成后的文件内容如代码清单 6-15 所示。配置中使用了 SDN 默认的 HTTP 驱动来连接数据库，配置参数中使用 localhost 表示连接本地的数据库，其中用户名和密码可以根据数据库的设定进行配置。

代码清单 6-15　数据库连接配置

```
compiler=org.neo4j.ogm.compiler.MultiStatementCypherCompiler
driver=org.neo4j.ogm.drivers.http.driver.HttpDriver
URI=http://localhost:7474
username = neo4j
password = 12345678
```

6.5.2　SDN 配置

如果使用 Spring Boot 开发框架，则 SDN 配置可以使用一个配置类来实现，如代码清单 6-16 所示。

配置类 Neo4jConfig 继承了 SDN 的配置类 Neo4jConfiguration，并重写了 getSessionFactory()方法，设定了我们上面设计域对象的路径 com.test.data.domain。

程序中使用注解@EnableNeo4jRepositories 激活了我们上面的存储库接口设计，并使用注解@EnableTransactionManagement 启用了 SDN 的隐式事务管理机制。

代码清单 6-16　SDN 配置

```
package com.test.data.config;

import org.neo4j.ogm.session.SessionFactory;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.neo4j.config.Neo4jConfiguration;
import
org.springframework.data.neo4j.repository.config.EnableNeo4jRepositories;
import
org.springframework.transaction.annotation.EnableTransactionManagement;
```

```
@Configuration
@EnableNeo4jRepositories(basePackages = { "com.test.data.repository" })
@EnableTransactionManagement
public class Neo4jConfig extends Neo4jConfiguration {

    @Override
    public SessionFactory getSessionFactory() {
        return new SessionFactory("com.test.data.domain");
    }
}
```

6.6 数据库设计验证

对于上面的对象建模及其实体持久化和查询设计，我们可以通过一个测试用例进行测试，以验证这些设计是否正确和合理。

这个测试用例包含两个功能，分别为增加数据和查询数据。

测试用例中增加数据的设计如代码清单 6-17 所示。

程序中使用注解@Before，表示在其他测试开始之前首先执行这个测试。

程序中创建了一个比赛节点，设定为 2016 年的“第一回合”比赛；创建了一个东部球队节点，名字属性设定为“东部一队”，并在“第一回合”比赛中赢得了两场比赛，为这轮比赛的失败方，然后调用东部球队存储库接口的 save()方法保存节点；创建了一个西部球队节点，名字属性设定为“西部一队”，并在“第一回合”比赛中赢得了 4 场比赛，在这轮比赛中为获胜方，然后调用西部球队存储库接口的 save()方法保存节点。

其中，在保存节点后都使用 Assert 的 notNull()方法验证了节点的 ID 是否为非空。如果是则表示通过测试，否则表示不能通过测试。

代码清单 6-17　测试用例中增加数据的设计

```
@Before
public void add(){
    Playoff playoff = new Playoff();
```

```
    playoff.setYear("2016");
    playoff.setRound("第一回合");

    East east = new East();
    east.setName("东部一队");
    east.setCode("东一");
    east.win(2, playoff);
    eastTeamRepository.save(east);
    Assert.notNull(east.getId());

    West west = new West();
    west.setName("西部一队");
    west.setCode("西一");
    west.win(4, playoff);
    westTeamRepository.save(west);
    Assert.notNull(west.getId());
}
```

注意：上述程序创建的比赛节点，我们并没有使用它的存储库接口进行保存，猜一猜它会不会被保存下来呢？回顾一下我们在第 5 章中提到的 SDN 的智能机制，你也许就知道答案了。

测试用例中查询数据的设计如代码清单 6-18 所示。

程序中使用注解@Test 设定 get()方法是一个测试。

测试方法中调用了预测算法存储库接口的 findWinAndLoss()方法，使用上个测试创建的两支球队的名字作为参数，返回输赢的预测结果。然后使用 Assert 的 notEmpty()方法检查返回对象是否为非空。如果非空则表示通过测试。

代码清单 6-18　测试用例中查询数据的设计

```
@Test
public void get(){
    Set<Map<String,Object>> maps = ratioRepository.findWinAndLoss("东部一队",
"西部一队");
    Assert.notEmpty(maps);
    log.info("\n==========Year:{} ============",
```

```
maps.iterator().next().get("year"));
}
```

执行上面的测试程序，如果测试全部通过，就可以初步表明，我们上面的数据库设计基本上是正确的。

执行上面的测试程序之后，我们也可以在数据库的 Web 控制台上看到创建的节点、关系和属性等情况，如图 6-4 所示。

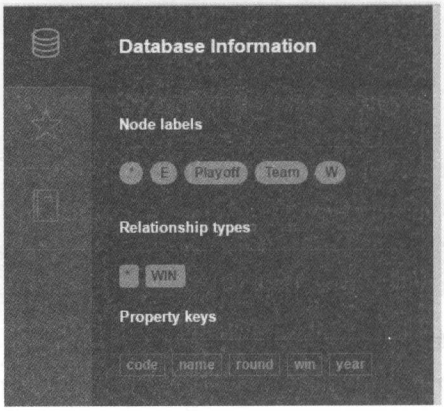


图 6-4　使用测试用例创建的数据对象

单击图 6-4 中的关系 WIN，也可以看到如图 6-5 所示的数据。

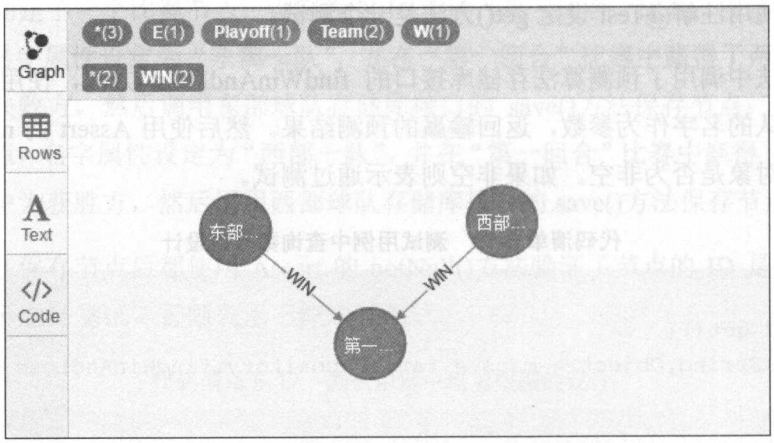


图 6-5　使用测试用例生成的数据

上面的一些预测算法的验证正像我们在 GraphGists 案例中看到的那样，都通过了测试并证明是可用的。当然，我们也可以使用 Neo4j 控制台，通过命令行输入框对这些 Cypher 查询语句进行验证。同时，SDN 的其他一些设计都可以参照这个测试用例的方法设计测试程序进行验证。

6.7　创建 Web 应用

通过 SDN 建模和一些数据访问的设计，我们已经建立了与 GraphGists 相同的 NBA 季后赛预测的数据库。但是，现在只能在数据库客户端中查看数据，还不能通过一个友好的操作界面很直观地管理数据的录入和编辑等操作，以及查看各种查询算法展示的效果。下面我们创建一个 Web 应用来使用上面用 SDN 创建的季后赛数据库。

在实例工程中，我们使用模块的方式来管理项目，上面设计的数据管理的模块名字设为 data。现在，我们使用模块的方式创建一个 Web 应用，模块的名字设为 web。web 模块将引用 data 模块的设计，这可以通过工程的 Maven 管理来设置。

Web 模块的依赖配置如代码清单 6-19 所示。配置中不但引用了 Spring Boot 的 Web 组件，也引用了页面设计的 Thymeleaf 模板，同时还引用了 data 模块。

代码清单 6-19　Web 模块的依赖配置

```
<dependencies>
      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-web</artifactId>
      </dependency>
      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-thymeleaf</artifactId>
      </dependency>
      <dependency>
         <groupId>com.test</groupId>
         <artifactId>data</artifactId>
         <version>${project.version}</version>
```

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

6.8 Web 前后端设计

一个 Web 应用大体上可以分为前端设计和后端设计两部分。后端设计包含数据访问和界面访问控制等内容，因为数据访问基本上已经在 data 模块中完成了设计，所以这里的主要工作就是界面访问控制设计方面的内容。前端设计主要是界面设计方面的内容，包含页面设计、显示样式设计和 JavaScript 脚本设计等。

对于 NBA 季后赛预测来说，在需要进行数据管理及操作的节点对象中，有东部球队节点、西部球队节点和比赛节点等。下面主要以东部球队节点的数据管理为例，来说明怎样进行 Web 前后端设计，其他节点的管理界面设计与此类似，可以参照这些方法来实现。

6.8.1 Web 后端设计

在一个 Web 应用中，操作界面的访问控制设计可以通过控制器的设计来实现。使用控制器，一般可以使用如下功能：

- 定义超链接地址，即提供给浏览器访问的 URL。
- 调用存储库接口，实现数据存取的请求。
- 返回页面或数据。

东部球队数据管理的 Web 后端设计包括主页访问控制设计、球队详情查看访问控制设计、新增数据访问控制设计、修改数据访问控制设计、删除数据访问控制设计等内容。

1. 主页访问控制设计

东部球队的主页访问控制设计如代码清单 6-20 所示。

程序中使用注解@Controller 将类 EastController 定义为一个控制器，并使用注解

@RequestMapping 定义了 URL 的前缀为“/east”，然后在 index()方法中定义了主页的 URL 为“/index”，这个 URL 与前缀连接起来构成了主页的 URL“/east/index”，这是一个相对位置的 URL。如果在浏览器上访问，加上运行应用的主机地址，就可以访问这个链接。例如，如果在本地上运行 Web 应用，则使用完整的链接 http://localhost/east/index 就将请求到这个 index()方法。

程序中 index()方法返回的字符串“east/index”指定了页面文件为目录 east 之下的 index.html，这是展示东部球队列表数据的主页设计文件。这种转换机制由 Thymeleaf 的配置所实现，这里我们使用它的默认配置，主要实现了在返回的字符串中加上文件扩展名“.html”的功能。

程序中的 getList()方法定义了 URL“/list”，并调用了东部球队存储库接口的 findEast()方法，请求了东部球队的分页列表数据，最后返回一个页对象 Page。其中，请求参数 TeamQo 是一个与实体类 Team 具有相同属性的简单类，并加上了几个分页属性，这样做的目的是可以方便组织参数的传递使用。

代码清单 6-20　主页访问控制设计

```
package com.test.web.controller;
......
@Controller
@RequestMapping("/east")
public class EastController {
   @Autowired
   EastTeamRepository eastTeamRepository;

        @RequestMapping("/index")
   public String index(ModelMap model) throws Exception{
      return "east/index";
   }

   @RequestMapping(value = "/list")
   @ResponseBody
   public Page<East> getList(TeamQo teamQo) {
      try {
         Pageable pageable = new PageRequest(teamQo.getPage(),
```

```
teamQo.getSize(), null);
            Set<East> easts = eastTeamRepository.findEast(teamQo.getName(),
teamQo.getPage() * teamQo.getSize(), teamQo.getSize());
            int count = eastTeamRepository.findEastCount(teamQo.getName());
            return new PageImpl(new ArrayList(easts), pageable, (long)count);
        }catch (Exception e){
            e.printStackTrace();
        }
        return null;
    }
}
```

2. 球队详情查看访问控制设计

在东部球队的主页上，列表数据展示的是每支球队的概要信息，通过在概要信息中提供的超链接，可以查看一支球队的详细信息。

东部球队详情查看访问控制设计如代码清单 6-21 所示。

程序中定义了查看详情的 URL 为“/{id}”，即访问这个 URL，将以球队的 ID 作为参数，调用东部球队存储库接口的 findOne()方法请求数据，然后返回字符串“east/show”，即指定使用页面设计文件 show.html 来显示查询结果。其中使用了控制器中的 ModelMap 将球队对象的数据传递给页面。

代码清单 6-21　球队详情查看访问控制设计

```
@RequestMapping(value="/{id}")
public String show(ModelMap model,@PathVariable Long id) {
    East east = eastTeamRepository.findOne(id);
    model.addAttribute("team",east);
    return "east/show";
}
```

3. 新增数据访问控制设计

东部球队的新增数据访问控制设计如代码清单 6-22 所示。

程序中 create()方法定义了新增数据的 URL 为“/new”，访问这个 URL 将返回字符

串“east/new”，即指定返回页面设计文件 new.html。

程序中 save()方法定义了一个保存数据的 URL 为“/save”，访问这个 URL 将调用东部球队存储库接口的 save()方法请求保存数据。其中使用了实体对象 East 作为参数，即从页面中传入保存对象的数据。

代码清单 6-22　新增数据访问控制设计

```
@RequestMapping("/new")
public String create(ModelMap model,East east){
    model.addAttribute("team", east);
    return "east/new";
}

@RequestMapping(value="/save", method = RequestMethod.POST)
@ResponseBody
public String save(East east) throws Exception{
    eastTeamRepository.save(east);
    logger.info("新增->ID="+east.getId());
    return "1";
}
```

4. 修改数据访问控制设计

东部球队的修改数据访问控制设计如代码清单 6-23 所示。

程序中的第一个 update()方法定义了修改数据的 URL 为“/edit/{id}”，即使用节点的 ID 作为传入参数，并使用这个 ID 通过东部球队存储库接口的 findOne()方法查找出东部球队实体对象 East，然后返回字符串“east/edit”，即指定返回页面设计文件 edit.html，展示一个具有节点数据的界面为用户提供修改操作。

程序中的另一个 update()方法定义了保存修改数据的 URL 为“/update”，访问这个 URL，根据传入的节点参数 East，调用东部球队存储库接口的 save()方法保存修改数据。

代码清单 6-23　修改数据访问控制设计

```
@RequestMapping(value="/edit/{id}")
public String update(ModelMap model,@PathVariable Long id){
```

```
    East east = eastTeamRepository.findOne(id);
    model.addAttribute("team",east);
    return "east/edit";
}

@RequestMapping(method = RequestMethod.POST, value="/update")
@ResponseBody
public String update(East east) throws Exception{
    eastTeamRepository.save(east);
    logger.info("修改->ID="+east.getId());
    return "1";
}
```

5. 删除数据访问控制设计

东部球队的删除数据访问控制设计如代码清单 6-24 所示。

程序中的 delete()方法定义了删除数据的 URL 为“/delete/{id}”，即删除数据时将使用节点的 ID 作为传入参数，然后调用东部球队存储库接口的 delete()方法删除节点数据。

代码清单 6-24　删除数据访问控制设计

```
@RequestMapping(value="/delete/{id}",method = RequestMethod.GET)
@ResponseBody
public String delete(@PathVariable Long id) throws Exception{
    eastTeamRepository.delete(id);
    logger.info("删除->ID=" + id);
    return "1";
}
```

6.8.2　Web 前端设计

Web 前端设计主要是界面设计方面的内容，包含页面设计、显示样式设计和 JavaScript 脚本设计等。其中，页面设计主要使用 HTML 语言来实现，并结合使用 Thymeleaf 模板及其标签语言，加强了页面设计的方法；显示样式设计主要使用 CSS 来实现；JavaScript 脚本设计主要使用 jQurey 工具来实现。

东部球队数据管理的 Web 前端设计主要包括主页界面设计、新增数据界面设计、修改数据界面设计和删除数据确认设计等内容。

1. 主页界面设计

东部球队的主页界面设计主要包括一个页面设计文件 index.html 和一个 JavaScript 脚本设计文件 list.js。

其中，页面设计文件的内容主要包括 4 个方面，分别为页头设计、查询表单设计、新增球队的超链接设计和列表设计。

东部球队主页的页头设计如代码清单 6-25 所示，主要实现了一些样式和脚本设计的引用，各个引用的功能说明如下。

- fragments/layout 引用了 Thymeleaf 的模板设计。
- jquery.pagination.js 和 pagination.css 分别是一个分页控件的插件和分页控件的样式设计。
- WdatePicker.js 和 WdatePicker.css 分别是一个日期控件的插件和日期控件的样式设计。
- artDialog.js 和 default.css 分别是一个对话框控件的插件和对话框控件的样式设计。
- list.js 是页面控件的一些事件响应和数据请求的实现方法设计。

代码清单 6-25　东部球队主页页头设计

```
<html xmlns:th="http://www.thymeleaf.org"
layout:decorator="fragments/layout">
<head>
   <title>东部球队管理</title>
   <link th:href="@{/scripts/pagination/pagination.css}" rel="stylesheet"
type="text/css" />
   <link th:href="@{/scripts/artDialog/default.css}" rel="stylesheet"
type="text/css" />
   <link th:href="@{/scripts/My97DatePicker/skin/WdatePicker.css}" rel=
"stylesheet" type="text/css" />
```

```
    <script th:src="@{/scripts/pagination/jquery.pagination.js}"></script>
    <script th:src="@{/scripts/jquery.smartselect-1.1.min.js}"></script>
    <script th:src="@{/scripts/artDialog/artDialog.js}" />
    <script th:src="@{/scripts/My97DatePicker/WdatePicker.js}"></script>
    <script th:src="@{/scripts/team/list.js}"></script>
</head>
```

东部球队主页的查询表单设计如代码清单 6-26 所示，主要实现了东部球队列表数据查询的功能。

表单中使用了一个标识为“名称”的控件，用来查询输入球队的名称。

超链接“查询”设定了控件的标识为 searchBtn，它的响应事件的实现将由脚本文件 list.js 来完成。

代码清单 6-26　东部球队主页的查询表单设计

```
<form id="queryForm" method="get">
       <div class="right-tab">
          <p class="tab-hover">东部球队</p>
          <p onclick="window.location.href='/west/index'">西部球队</p>
       </div>

       <div class="radiusGrayBox782">
       <div class="radiusGrayTop782"></div>
       <div class="radiusGrayMid782">
          <div class="dataSearchBox forUserRadius">
             <ul>
                <li>
                   <label class="preInpTxt f-left">名称</label>
                   <input type="text" class="inp-list f-left w-200"
value="" id="name" name="name"/>
                </li>
                <li>
                   <a href="javascript:void(0)" class="blueBtn-62X30
f-right" id="searchBtn">查询</a>
                </li>
             </ul>
          </div>
```

```
        </div>
    </div>
</form>
```

东部球队主页中的新增球队的超链接设计如代码清单 6-27 所示。这里主要创建了一个“新增”超链接，并将控件标识设定为 addInfo，控件的事件响应将由脚本文件 list.js 来设计。单击这个超链接将打开一个创建球队的操作界面。

代码清单 6-27　新增球队的超链接设计

```
<div class="newBtnBox">
  <a id="addInfo" class="blueBtn-62X30" href="javascript:void(0)">新增</a>
</div>
```

东部球队主页的表格控件设计如代码清单 6-28 所示，主要使用一个表格控件实现了球队列表数据显示的功能。

表格控件的表头部分设定了列表数据显示字段的名称，分别为“ID”、“名称”、“代码”和“操作”。其中“操作”字段将放置一些操作链接，可以执行对当条记录的查看、修改和删除等操作。

表格控件的表体部分只设定一个标识符 tbodyContent，这里的数据将由脚本文件 list.js 中设计的方法来填充。

表格控件下面使用分页插件 pagination 设置了一个列表工具条控件，它可以用来显示列表的页数和控制显示的分页列表数据。

代码清单 6-28　东部球队表格控件设计

```
<div class="dataDetailList mt-12">
      <table class="dataListTab">
          <thead>
          <tr>
              <th>ID</th>
              <th>名称</th>
              <th>代码</th>
              <th>操作</th>
          </tr>
          </thead>
```

```
            <tbody id="tbodyContent">
            </tbody>
        </table>
        <div class="tableFootLine">
            <div class="pagebarList pagination"/>
        </div>
</div>
```

东部球队主页设计中的脚本文件 list.js 的设计主要包括页面初始化设计、分页插件工具条设计、分页数据请求和回调函数设计、分页列表数据填充设计、对话框设计等内容。

脚本文件 list.js 中的页面初始化设计如代码清单 6-29 所示。这是一个页面初始化函数的设计。

在这个初始化函数中，设置了主页中标识为 searchBtn 的超链接“查询”控件的单击响应事件，即在页面上单击“查询”链接将执行 pageaction()函数，请求分页列表数据。

在初始化函数中，也设置了主页中标识为 addInfo 的超链接“新增”控件的单击响应事件，即在页面上单击“新增”链接将执行 create()函数，打开一个创建球队节点的对话框。

在初始化函数中，默认运行了 pageaction()函数，在打开东部球队主页时，将请求东部球队的分页列表数据进行显示。

在初始化函数中，还设置了一个 live 事件，为分页工具条插件 pagination 设置了 change 响应事件，用来执行跳转按钮的功能。

代码清单 6-29　东部球队主页的页面初始化设计

```
$(function () {
    $('#searchBtn').click(function(){
        pageaction();
    });
    $('#addInfo').click(function(){
        create();
    });
    //初始化分页
```

```
    pageaction();
    var pg = $('.pagination');
    $('#pageSelect').live("change",function(){
        pg.trigger('setPage', [$(this).val()-1]);
    });
});
```

脚本文件 list.js 中的分页插件工具条设计如代码清单 6-30 所示。

工具条设定了每页显示的记录条数为 10 条，并设定了“首页”、“上页”、“下页”、“尾页”等按钮，也设定了在这些按钮中间可以使用 3 个按页码显示的按钮，同时设定了可以显示页总数，并指定了回调函数的名称为 pageselectCallback。

代码清单 6-30　东部球队主页的分页插件工具条设计

```
var getOpt = function(){
    var opt = {
        items_per_page: 10,          //每页记录数
        num_display_entries: 3,      //中间显示的页数个数，默认为 10
        current_page:0,              //当前页
        num_edge_entries:1,          //头尾显示的页数个数，默认为 0
        link_to:"javascript:void(0)",
        prev_text:"上页",
        next_text:"下页",
        load_first_page:true,
        show_total_info:true ,
        show_first_last:true,
        first_text:"首页",
        last_text:"尾页",
        hasSelect:false,
        callback: pageselectCallback //回调函数
    }
    return opt;
}
```

脚本文件 list.js 中的分页数据请求和回调函数设计如代码清单 6-31 所示。

数据请求通过访问当前位置中的 URL“./list”向控制器请求列表数据，然后使用返回的数据更新分页插件工具条上的显示信息。

回调函数 pageselectCallback 通过调用 fillData()函数向页面表格控件填充数据，同时响应分页插件中工具条按钮的事件，通过传递页码等参数重新请求分页数据。

代码清单 6-31　东部球队主页的分页数据请求和回调函数设计

```
var currentPageData = null ;
var pageaction = function(){
    $.get('./list?t='+new Date().getTime(),{
        name:$("#name").val()
    },function(data){
        currentPageData = data.content;
        $(".pagination").pagination(data.totalElements, getOpt());
    });
}

var pageselectCallback = function(page_index, jq, size){
    var html = "" ;
    if(currentPageData!=null){
        fillData(currentPageData);
        currentPageData = null;
    }else
        $.get('./list?t='+new Date().getTime(),{
            size:size,page:page_index,name:$("#name").val()
        },function(data){
            fillData(data.content);
        });
}
```

脚本文件 list.js 中的分页列表数据填充设计如代码清单 6-32 所示。

填充数据时，首先清空了表格控件中标识为 tbodyContent 的表体中原来显示的内容，然后使用一个 each 循环展开数据对象进行数据填充，即使用球队对象的属性 id、name 和 code 分别表示“节点标识”、“球队名称”和“代码”，在最后一列数据中设置了“查看”、“修改”和“删除”等超链接，以方便对每个球队节点执行 CURD 等操作。

代码清单 6-32　东部球队主页的分页列表数据填充设计

```
function fillData(data){
```

```
    var $list = $('#tbodyContent').empty();
    $.each(data,function(k,v) {
        var html = "";
        html += '<tr> ' +
            '<td>' + (v.id == null ? '' : v.id) + '</td>' +
            '<td>' + (v.name == null ? '' : v.name) + '</td>' +
            '<td>' + (v.code == null ? '' : v.code) + '</td>';
        html += '<td><a class="c-50a73f mlr-6" href="javascript:void(0)"
onclick="showDetail(\'' + v.id + '\')">查看</a>';

        html += '<a class="c-50a73f mlr-6" href="javascript:void(0)" onclick=
"edit(\'' + v.id + '\')">修改</a>';

        html += '<a class="c-50a73f mlr-6" href="javascript:void(0)" onclick=
"del(\''+ v.id+'\')">删除</a>';

        html +='</td></tr>' ;

        $list.append($(html));
    });
}
```

脚本文件 list.js 中的对话框设计如代码清单 6-33 所示。

这是在列表页面中通过单击“查看”链接打开查看球队详情的一个对话框设计。程序通过在当前位置访问 URL“./{id}”，即使用东部球队节点的 ID 作为参数，向东部球队控制器请求数据。取得数据后，在当前界面上打开一个对话框来显示页面及其数据。对话框可以设置标题、窗口大小和显示位置等。其中，初始化函数 init 将对话框存入变量 artdialog 中，关闭对话框函数 close 可以通过变量 artdialog 来关闭对话框。

代码清单 6-33　东部球队主页的对话框设计

```
var artdialog ;
function showDetail(id){
    $.get("./"+id,{ts:new Date().getTime()},function(data){
        art.dialog({
            lock:true,
            opacity:0.3,
```

```
            title: "查看信息",
            width:'750px',
            height: 'auto',
            left: '50%',
            top: '50%',
            content:data,
            esc: true,
            init: function(){
                artdialog = this;
            },
            close: function(){
                artdialog = null;
            }
        });
    });
}
```

完成了东部球队主页界面设计之后，最终显示的效果如图 6-6 所示。图的上部是一个查询表单设计，中间是一个“新增”超链接设计，下部是数据列表和分页插件工具条设计。

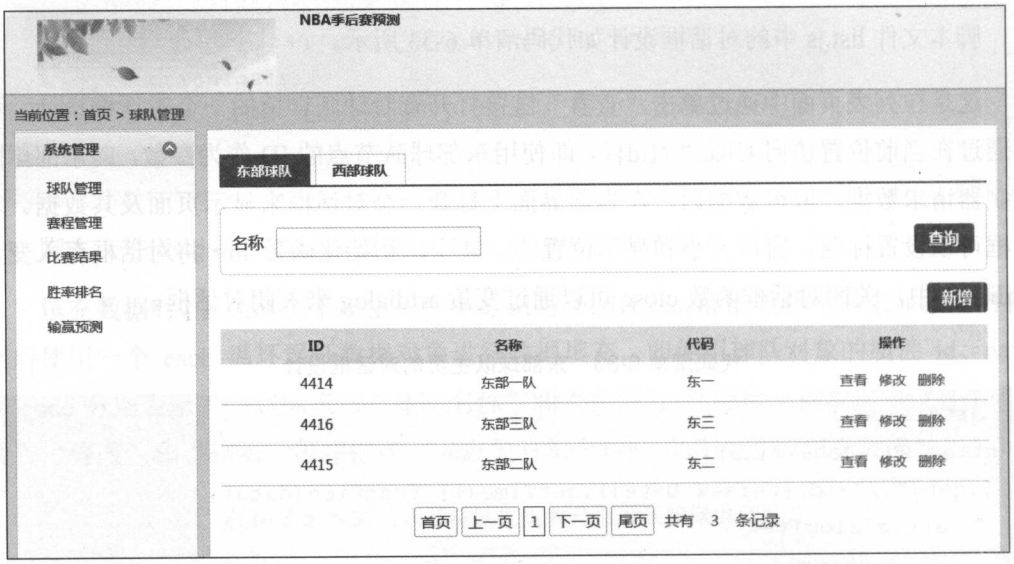


图 6-6　东部球队主页显示效果

2. 新增数据界面设计

东部球队新增数据的页面设计文件为 new.html，其内容如代码清单 6-34 所示。代码中引用了脚本文件 new.js，用来完成页面中表单提交数据的保存请求。页面中使用表单 form 设置了球队节点的两个属性“名称”和“代码”录入的相关控件。

代码清单 6-34　新增数据的页面设计

```
<script th:src="@{/scripts/team/new.js}"></script>
<form id="saveForm" action="./save" method="post">
    <table class="addNewInfList">
        <tr>
            <th>名称</th>
            <td width="240">
                <input class="inp-list w-200 clear-mr f-left" type="text"
name= "name" id="name" maxlength="32" />
                <span class="tipStar f-left">*</span>
            </td>
            <th>代码</th>
            <td width="240"><input class="inp-list w-200 clear-mr f-left"
type= "text" name="code" id="code"/></td>
        </tr>
    </table>
    <div class="bottomBtnBox">
        <a class="btn-93X38 saveBtn" href="javascript:void(0)">确定</a>
        <a class="btn-93X38 backBtn" href="javascript: closeDialog()">返回</a>
    </div>
</form>
```

上述页面引用的脚本文件 new.js 的设计如代码清单 6-35 所示。当上述页面提交表单时，将执行这个脚本。脚本中引用页面的表单对象“$('#saveForm')”对属性 name 进行非空验证，如果验证通过，则将在当前路径中访问 URL“/save”，执行保存数据的请求。其中，使用表单的系列化函数 serialize()将表单的控件转化为节点的属性参数。在发送保存数据请求时，如果请求成功则控制器将返回 1，否则返回错误提示。

代码清单 6-35　新增页面的验证和保存方法

```
$(function(){
   $('#saveForm').validate({
      rules: {
           name       :{required:true}
      },messages:{
           name :{required:"必填"}
        }
   });
$('.saveBtn').click(function(){
    if($('#saveForm').valid()){
        $.ajax({
           type: "POST",
           url: "./save",
           data: $("#saveForm").serialize(),
           headers: {"Content-type":
"application/x-www-form-urlencoded; charset=UTF-8"},
           success: function (data) {
              if (data == 1) {
                 alert("保存成功");
                 pageaction();
                 closeDialog();
              } else {
                 alert(data);
              }
           }
        });
      }else{
         alert('数据验证失败，请检查！');
      }
   });
});
```

完成设计后，新增页面的显示效果如图 6-7 所示。

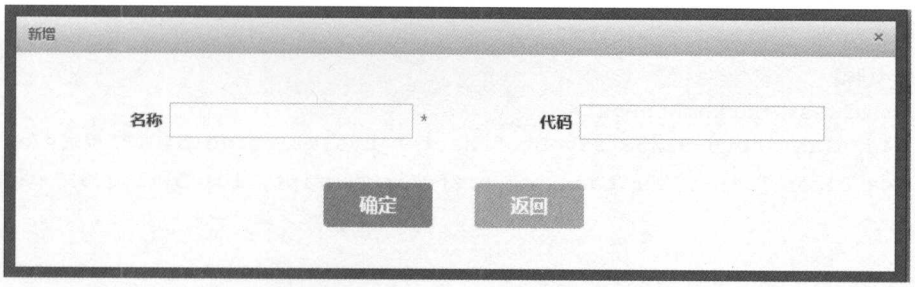


图 6-7　新增页面的显示效果

3. 修改数据界面设计

东部球队修改数据的页面设计文件为 edit.html，其内容如代码清单 6-36 所示。页面设计中引用了脚本文件 edit.js，用来实现表单数据的验证和发送保存数据的请求。在表单设计中，使用 type="hidden"的方式设计了一个隐藏控件，用来暂存节点的 ID，这样，在提交修改时将保证修改操作的正确性。代码中的 th:value 表示使用了 Thymeleaf 的标签语言为控件赋值。

代码清单 6-36　修改数据的页面设计

```
<script th:src="@{/scripts/team/edit.js}"></script>
<form id="saveForm" method="post">
   <input type="hidden" name="id" id="id" th:value="${team.id}"/>
<div class="addInfBtn" >
   <h3 class="itemTit"><span>编辑</span></h3>
   <table class="addNewInfList">
      <tr>
         <th>名称</th>
         <td width="240">
            <input class="inp-list w-200 clear-mr f-left" type="text"
th:value="${team.name}" id="name" name="name" maxlength="16" />
            <span class="tipStar f-left">*</span>
         </td>
         <th>代码</th>
         <td>
            <input class="inp-list w-200 clear-mr f-left" type="text"
th:value="${team.code}" id="code" name="code" maxlength="16" />
         </td>
```

```
        </tr>
    </table>
    <div class="bottomBtnBox">
        <a class="btn-93X38 saveBtn" href="javascript:void(0)">确定</a>
        <a class="btn-93X38 backBtn" href="javascript:closeDialog(0)">返回</a>
    </div>
</div>
</form>
```

上述页面引用的脚本文件 edit.js 的内容如代码清单 6-37 所示。它的使用方法跟新建节点时所使用的方法有点类似，即当页面提交表单时，将首先执行表单控件的验证，然后使用相应的 URL 发送保存数据的请求。不同的是，修改数据的保存请求在当前路径中使用了 URL “/update”。

代码清单 6-37　修改数据的验证和保存方法

```
$(function(){
    $('#saveForm').validate({
        rules: {
            name      :{required:true}
        },messages:{
            name :{required:"必填"}
        }
    });
    $('.saveBtn').click(function(){
        if($('#saveForm').valid()){
            $.ajax({
                type: "POST",
                url: "./update",
                data: $("#saveForm").serialize(),
                headers: {"Content-type":
"application/x-www-form-urlencoded;charset=UTF-8"},
                success: function (data) {
                    if (data == 1) {
                        alert("编辑成功");
                        pageaction();
                        closeDialog();
                    } else {
```

```
                    alert(data);
                }
            }
        });
    }else{
        alert('数据验证失败，请检查！');
    }
  });
});
```

完成设计后，修改数据界面显示效果如图 6-8 所示。

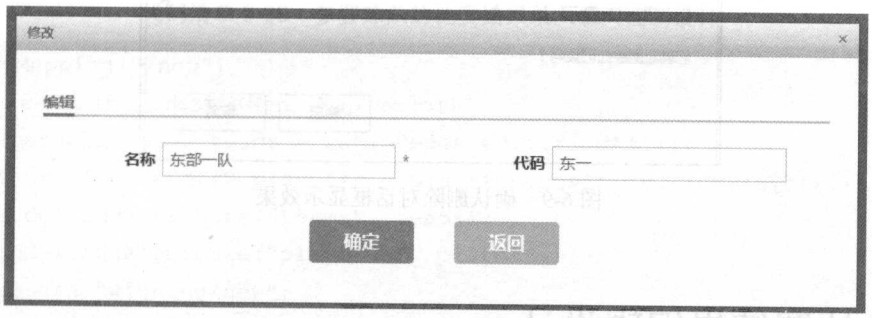


图 6-8　修改数据界面显示效果

4. 删除数据确认设计

删除节点数据时并不需要引用页面，但在执行删除请求的过程中需要有一个确认删除的提示对话框，以便用户进行确认。如代码清单 6-38 所示是设计删除确认对话框和发送删除请求的实现方法。其中，确认对话框使用 Java Script 的函数 confirm()来实现，发送删除数据请求的 URL 为当前位置中的“/delete/{id}”，其中使用了节点的 ID 来执行这个请求。

代码清单 6-38　删除数据确认设计

```
function del(id){
    if(!confirm("您确定删除此记录吗？")){
        return false;
    }
    $.get("./delete/"+id,{ts:new Date().getTime()},function(data){
```

```
        if(data==1){
            alert("删除成功");
            pageaction();
        }else{
            alert(data);
        }
    });
}
```

完成删除确认设计，最后的显示效果如图 6-9 所示。

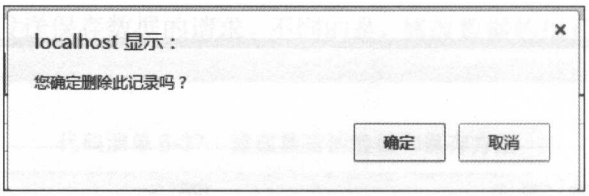


图 6-9　确认删除对话框显示效果

6.9　比赛结果编辑设计

参赛球队的比赛结果编辑是指球队的赢得关系编辑，一轮比赛结果的输赢将涉及两支参赛球队的赢得关系，所以赢得关系的编辑跟节点对象的编辑不大一样。

比赛结果编辑的 Web 设计包含后端的访问控制设计和前端的界面设计两部分内容。

6.9.1　比赛结果编辑的访问控制设计

参赛球队的比赛结果编辑包括数据录入、修改和删除等操作，这里只说明录入和删除设计方法。

1. 比赛结果录入访问控制设计

参赛球队的比赛结果录入访问控制设计如代码清单 6-39 所示。

程序中的 create()方法定义了一个当前位置的 URL “/new”，并返回字符串“winning/new”，即指定返回页面文件为 new.html，用来构建一个比赛结果录入的界面。其中，调用存储库接口 RatioRepository 的 findAll()方法返回所有球队的列表，调用存储库接口 PlayoffRepository 的 findPlayoff()方法返回可用的比赛列表，用来作为页面中赢得关系录入的参数。

程序中的 save()方法定义了一个当前位置的 URL “/save”，用来执行存储比赛录入结果的请求。在存储录入结果时，首先为参赛的两支球队各自创建赢得关系，然后分别调用存储库接口 RatioRepository 的 save()方法保存赢得关系。

代码清单 6-39　参赛球队的比赛结果录入访问控制设计

```
@RequestMapping("/new")
    public String create(ModelMap model){
        Iterable<Team> teams = ratioRepository.findAll();
        Iterable<Playoff> playoffs = playoffRepository.findPlayoff();
        model.addAttribute("teams", teams);
        model.addAttribute("playoffs", playoffs);
        return "winning/new";
    }

    @RequestMapping(value="/save", method = RequestMethod.POST)
    @ResponseBody
    public String save(HttpServletRequest request) throws Exception{
        String t1 = request.getParameter("t1");
        String t2 = request.getParameter("t2");
        String win = request.getParameter("win");
        String loss = request.getParameter("loss");
        String pid = request.getParameter("pid");

        Playoff playoff = playoffRepository.findOne(new Long(pid));

        Team team1 = ratioRepository.findOne(new Long(t1));
        team1.win(new Integer(win), playoff);
        ratioRepository.save(team1);

        Team team2 =  ratioRepository.findOne(new Long(t2));
```

```
        team2.win(new Integer(loss), playoff);
        ratioRepository.save(team2);

        logger.info("新增->ID="+playoff.getId());
        return "1";
    }
```

2. 比赛结果删除访问控制设计

参赛球队的比赛结果删除访问控制设计如代码清单 6-40 所示。

程序中的 delete()方法定义了当前位置的 URL “/delete/{wid}/{lid}”，表示使用两支球队的赢得关系 ID，分别调用赢得关系存储库接口 WinningRepository 的 delete()方法删除赢得关系。

代码清单 6-40　参赛球队的比赛结果删除访问控制设计

```
@RequestMapping(value="/delete/{wid}/{lid}",method = RequestMethod.GET)
    @ResponseBody
    public String delete(@PathVariable Long wid, @PathVariable Long lid) throws
Exception{
        if(wid != null) {
            winningRepository.delete(wid);
            logger.info("删除->WID=" + wid);
        }
        if(lid != null){
            winningRepository.delete(lid);
            logger.info("删除->LID=" + lid);
        }
        return "1";
    }
```

注意：这里仅仅删除节点的关系，并不删除节点本身。所以我们特地设计了一个赢得关系存储库接口来执行关系的删除操作。实际上，通过修改球队节点的关系也能达到这个要求，但是实现起来并不比这个简便。

6.9.2　比赛结果的录入界面设计

参赛球队的比赛结果录入界面设计主要由页面设计文件 new.html 和 JavaScript 脚本文件 new.js 组成。

页面设计文件 new.html 的内容如代码清单 6-41 所示。录入界面中使用了“甲队”和“乙队”来代表参赛的两支球队，然后通过下拉列表框的方式分别选择参赛的“甲队”和“乙队”，并通过“甲赢场数”输入“甲队”赢得的比赛场数，通过“乙赢场数”输入“乙队”赢得的比赛场数，最后通过下拉列表框“赛程”选择一轮比赛，完成一轮比赛结果的录入。注意上面的所有选项和输入都在后面带有一个“*”，表示都是必填的，否则将不能通过表单验证。

代码清单 6-41　比赛结果录入页面设计

```
<script th:src="@{/scripts/winning/new.js}"></script>
<form id="saveForm" method="post">
<div class="addInfBtn" >
   <h3 class="itemTit"><span>比赛结果</span></h3>
   <table class="addNewInfList">
      <tr>
         <th>甲队</th>
         <td width="240">
            <div class="selectDownMode f-left w-206">
               <input type="text"  value="请选择"/>
               <select class="selectMode" name="t1" id="t1">
                  <option value="0">请选择</option>
                  <option th:each="team:${teams}" th:value="${team.id}"
                          th:text="${#strings.length(team.name)>20?#
strings.substring(team.name,0, 20)+'...':team.name}"
                        ></option>
               </select>
            </div>
            <span class="tipStar f-left">*</span>
         </td>
         <th>乙队</th>
         <td width="240">
            <div class="selectDownMode f-left w-206">
               <input type="text"  value="请选择"/>
```

```
                <select class="selectMode" name="t2" id="t2">
                    <option value="0">请选择</option>
                    <option th:each="team:${teams}" th:value="${team.id}"
                            th:text="${#strings.length(team.name)>20?#
strings.substring(team.name,0, 20)+'...':team.name}"
                           ></option>
                </select>
            </div>
            <span class="tipStar f-left">*</span>
        </td>
    </tr>
    <tr>
        <th>甲赢场数</th>
        <td width="240">
            <input class="inp-list w-200 clear-mr f-left" type="text"
id="win" name="win" maxlength="16" />
            <span class="tipStar f-left">*</span>
        </td>
        <th>乙赢场数</th>
        <td width="240">
            <input class="inp-list w-200 clear-mr f-left" type="text"
id="loss" name="loss" maxlength="16" />
            <span class="tipStar f-left">*</span>
        </td>
    </tr>
    <tr>
        <th>赛程</th>
        <td width="240">
            <div class="selectDownMode f-left w-206">
                <input type="text"  value="请选择"/>
                <select class="selectMode" name="pid" id="pid">
                    <option value="">请选择</option>
                    <option th:each="playoff:${playoffs}" th:value=
"${playoff.id}"
th:text="${#strings.length(playoff.round)>20?# playoff. year+':'+strings.
substring(playoff.round,0,20)+'...':playoff.year+':'+playoff.round}"
                           ></option>
                </select>
```

```
            </div>
            <span class="tipStar f-left">*</span>
        </td>
    </tr>
</table>
<div class="bottomBtnBox">
    <a class="btn-93X38 saveBtn" href="javascript:void(0)">确定</a>
    <a class="btn-93X38 backBtn" href="javascript:closeDialog(0)">返回</a>
</div>
</div>

</form>
```

脚本文件 new.js 的内容如代码清单 6-42 所示。这个脚本实现了上述录入界面中表单提交的数据验证和保存请求的实现方法。其中，saveForm 是上述表单的 ID，saveBtn 是上述表单“确定”按钮的 ID。当在界面上单击“确定”按钮后，将调用当前路径的 URL “./save”，访问控制器请求保存数据，如果返回“1”则提示“保存成功”，否则将提示出错信息。

代码清单 6-42　比赛结果录入验证及请求方法设计

```
$(function(){
        $('#saveForm').validate({
                rules: {
          t1   :{required:true},
          t2 :{required:true},
          win:{required:true},
          loss:{required:true},
          pid:{required:true}
                },messages:{
          t1 :{required:"必填"},
          t2 :{required:"必填"},
          win :{required:"必填"},
          loss :{required:"必填"},
          pid :{required:"必填"}
      }
        });
$('.saveBtn').click(function(){
```

```
        if($('#saveForm').valid()){
      $.ajax({
          type: "POST",
          url: "./save",
          data: $("#saveForm").serialize(),
          headers: {"Content-type":
"application/x-www-form-urlencoded;charset=UTF-8"},
          success: function (data) {
              if (data == 1) {
                  alert("保存成功");
                  pageaction();
                  closeDialog();
              } else {
                  alert(data);
              }
          }
      });
                  }else{
                          alert('数据验证失败，请检查！');
                  }
       });
});
```

完成参赛球队的比赛结果录入设计后，录入界面的显示效果如图 6-10 所示。

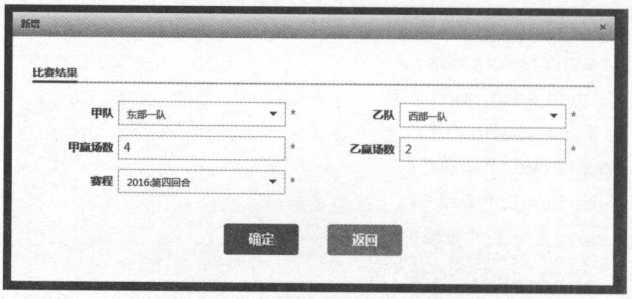


图 6-10　比赛结果录入界面显示效果

通过录入界面增加了一些比赛结果之后，就可以在比赛结果的主页上通过下拉列表框“球队名称”查询每支球队的历史参赛数据，并且可以进行分页查询。

比赛结果的列表查询效果如图 6-11 所示。

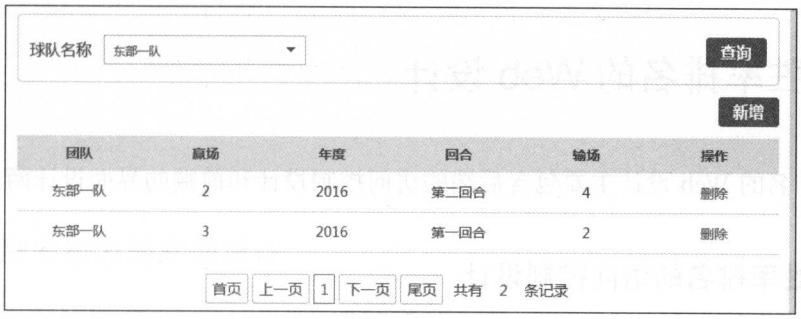


图 6-11　比赛结果列表查询显示效果

在比赛结果列表查询的显示界面上，通过单击“删除”超链接，即可删除一个比赛结果。对于比赛结果，这里没有提供修改的设计，读者如有兴趣，则可以考虑如何实现。

现在，在季后赛的 Web 管理后台录入一些球队、比赛和比赛结果等数据，在数据库客户端中通过单击所有关系，也可以看到如图 6-12 所示的图数据。

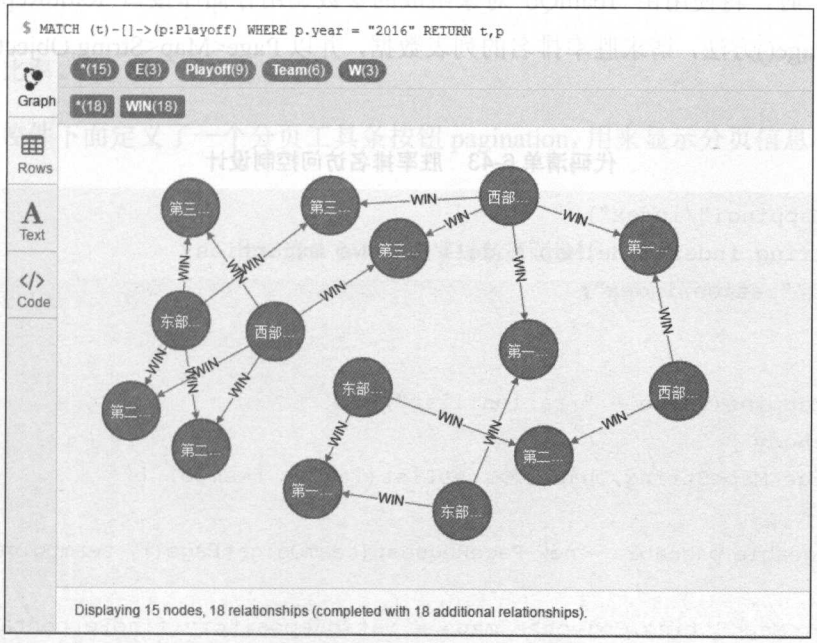


图 6-12　Web 录入数据后的图数据示例

有了这些数据，接下来实现其中一些预测算法的查询显示界面的设计就比较直观了。

6.10 胜率排名的 Web 设计

胜率排名的 Web 设计主要包含后端的访问控制设计和前端的界面设计两部分内容。

6.10.1 胜率排名的访问控制设计

胜率排名的访问控制设计如代码清单 6-43 所示。

程序中的 index()方法定义了一个当前位置的 URL 为“/index”，在页面上访问这个 URL 时，将返回一个字符串“ration/index”，这个字符串指定了使用当前目录 ration 下的 index.html 页面文件。

程序中的 getList()方法定义了一个当前位置的 URL 为“/ration_list”，在页面上访问这个 URL 时，将使用由 TeamQo 对象组成的参数调用存储库接口 RatioRepository 的 findPercentage()方法，请求胜率排名的列表数据，并以 Page<Map<String,Object>>对象返回数据。

代码清单 6-43 胜率排名访问控制设计

```
@RequestMapping("/index")
public String index(ModelMap model) throws Exception{
    return "ration/index";
}

@RequestMapping(value = "/ration_list")
@ResponseBody
public Page<Map<String,Object>> getList(TeamQo teamQo) {
    try {
        Pageable pageable = new PageRequest(teamQo.getPage(), teamQo.getSize(),
null);
        Set<Map<String,Object>> maps = ratioRepository.findPercentage
(teamQo.getPage() * teamQo.getSize(), teamQo.getSize());
```

```
        int count = ratioRepository.findPercentageCount();
        return new PageImpl(new ArrayList(maps), pageable, (long)count);
    }catch (Exception e){
        e.printStackTrace();
    }
    return null;
}
```

6.10.2　胜率排名的界面设计

胜率排名的界面设计包含一个页面设计文件 index.html 和一个脚本设计文件 ration_list.js。

页面设计文件 index.html 的内容如代码清单 6-44 所示。

代码的开头引用了 Thymeleaf 的模板 fragments/layout，页头中引用了分页插件、日期插件等公用插件，同时也引用了脚本文件 ration_list.js。代码中的主要内容显示区域定义了一个表格控件，用来显示列表数据，并预定义了表头字段显示名称，分别为“球队”、“总赢”、“总输”和“比率”，用来表示在一支球队的比赛历史中，所有比赛输赢场次的总计及其比率。

表格控件下面定义了一个分页工具条按钮 pagination，用来显示分页信息和控制分页数据显示。

代码清单 6-44　胜率排名页面设计

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
layout:decorator="fragments/layout">
<head>
    <title>胜率排名</title>

    <link th:href="@{/scripts/pagination/pagination.css}" rel="stylesheet"
type="text/css" />
    <link th:href="@{/scripts/artDialog/default.css}" rel="stylesheet"
type="text/css" />
    <link th:href="@{/scripts/My97DatePicker/skin/WdatePicker.css}" rel=
```

```
"stylesheet" type="text/css" />

    <script th:src="@{/scripts/pagination/jquery.pagination.js}"></script>
    <script th:src="@{/scripts/jquery.smartselect-1.1.min.js}"></script>
    <script th:src="@{/scripts/artDialog/artDialog.js}" />
    <script th:src="@{/scripts/My97DatePicker/WdatePicker.js}"></script>
    <script th:src="@{/scripts/ration/ration_list.js}"></script>
</head>
<body>
<div class="locationLine" layout:fragment="prompt">
    当前位置: 首页 &gt; <em >胜率排名</em>
</div>
<div class="statisticBox w-782"  layout:fragment="content">
    <div class="dataDetailList mt-12">
        <table class="dataListTab">
            <thead>
            <tr>
                <th>球队</th>
                <th>总赢</th>
                <th>总输</th>
                <th>比率</th>
            </tr>
            </thead>
            <tbody id="tbodyContent">
            </tbody>
        </table>
        <div class="tableFootLine">
            <div class="pagebarList pagination"/>
        </div>
    </div>
</div>
</body>
</html>
```

胜率排名的页面引用的脚本文件 ration_list.js 的设计包括页面初始化、分页插件工具条设计、分页数据请求和回调函数设计、分页列表数据填充设计等功能。这里只说明一下分页数据请求和回调函数设计、分页列表数据填充设计这两部分功能的实现。

胜率排名的分页数据请求和回调函数设计如代码清单 6-45 所示。

数据请求通过访问当前位置中的 URL “./ration_list” 向控制器请求列表数据，然后使用返回的数据更新分页工具条上的显示信息。

回调函数 pageselectCallback 通过调用 fillData()函数向页面表格控件填充数据，同时响应分页插件中工具条按钮的事件，通过传递页码等参数重新请求分页数据。

代码清单 6-45　胜率排名的分页数据请求和回调函数设计

```
var currentPageData = null ;
var pageaction = function(){
    $.get('./ration_list?t='+new Date().getTime(),{
    },function(data){
        currentPageData = data.content;
        $(".pagination").pagination(data.totalElements, getOpt());
    });
}

var pageselectCallback = function(page_index, jq, size){
    var html = "" ;
    if(currentPageData!=null){
        fillData(currentPageData);
        currentPageData = null;
    }else
        $.get('./ration_list?t='+new Date().getTime(),{
            size:size,page:page_index
        },function(data){
            fillData(data.content);
        });
}
```

胜率排名的分页列表数据填充设计如代码清单 6-46 所示。

填充数据时，首先清空了表格控件中标识为 tbodyContent 的表体原来显示的内容，然后使用一个 each 循环展开数据对象进行数据填充，列表中的数据分别为 team、wins、losses 和 percentage，分别表示“球队”、“总赢”、“总输”和“比率”。

代码清单 6-46　胜率排名的分页列表数据填充设计

```
function fillData(data){
    var $list = $('#tbodyContent').empty();
    $.each(data,function(k,v) {
        var html = "";
        html += '<tr> ' +
            '<td>' + (v.team == null ? '' : v.team) + '</td>' +
            '<td>' + (v.wins == null ? '' : v.wins) + '</td>' +
            '<td>' + (v.losses == null ? '' : v.losses) + '</td>' +
            '<td>' + (v.percentage == null ? '' : v.percentage.toFixed(3)) +
'</td>';
        html +='</tr>' ;
        $list.append($(html));
    });
}
```

完成胜率排名的界面设计后，显示效果如图 6-13 所示。

球队	总赢	总输	比率
东部三队	15	6	0.714
西部二队	9	15	0.375
西部一队	8	13	0.381
东部二队	6	4	0.600
西部三队	6	5	0.545
东部一队	5	6	0.455

首页　上一页　1　下一页　尾页　共有　6　条记录

图 6-13　胜率排名界面设计显示效果

6.11　输赢预测的 Web 设计

输赢预测的 Web 设计主要包含后端的访问控制设计和前端的界面设计两部分内容。

6.11.1　输赢预测的访问控制设计

输赢预测的访问控制设计如代码清单 6-47 所示。

程序中的 win_loss()方法定义了一个当前位置的 URL 为“/winloss”，在页面上访问这个 URL 时，将返回一个字符串“ration/win_loss”，这个字符串指定了使用当前目录 ration 下的页面文件 win_loss.html。其中，调用了存储库接口 RatioRepository 的 findAll()方法，即使用球队列表数据作为页面下拉列表框的参数。

程序中的 getWin_Loss()方法定义了一个当前位置的 URL 为“/win_loss”，在访问这个 URL 时，将首先调用存储库接口 RatioRepository 的 findWinAndLoss()方法，通过传入两支球队的名称作为参数，查询两支球队的交战记录。如果两队没有交战记录，则调用存储库接口 RatioRepository 的 findNeverMetPaths()方法查询与这两支球队关系最接近的球队的比赛情况。其次，调用存储库接口 RatioRepository 的 findAvgNetWin()方法计算出甲队的平均净赢数据。最后，将这些查询的数据通过一个 Map 对象返回给调用者。

代码清单 6-47　输赢预测访问控制设计

```
@RequestMapping("/winloss")
public String win_loss(ModelMap model) throws Exception{
    Iterable<Team> teams = ratioRepository.findAll();
    model.addAttribute("teams", teams);
    return "ration/win_loss";
}

@RequestMapping(value = "/win_loss")
@ResponseBody
public Map<String,Object> getWin_Loss(HttpServletRequest request) {
    try {
        String t1 = request.getParameter("t1");
        String t2 = request.getParameter("t2");
        Map<String,Object> data = new HashMap<>();
        Set<Map<String,Object>> maps = ratioRepository.findWinAndLoss(t1, t2);
        if(maps != null && maps.size() > 0){
            data.put("met", 1);
            data.put("content", maps);
        }else{
```

```
            maps = ratioRepository.findNeverMetPaths(t1, t2);
            data.put("met", 2);
            data.put("content", maps);
        }
        float netwin = ratioRepository.findAvgNetWin(t1, t2);
        data.put("netwin", netwin);
        return data;
    }catch (Exception e){
        e.printStackTrace();
    }
    return null;
}
```

6.11.2 输赢预测的界面设计

输赢预测的界面设计主要包括一个页面设计文件 win_loss.html 和一个脚本设计文件 win_loss.js。

页面设计文件 win_loss.html 的内容主要包括一个查询表单设计和一个显示列表数据的表格控件设计。

输赢预测的查询表单设计如代码清单 6-48 所示。

表单中设置了两个下拉列表框控件，分别表示甲队和乙队，可以用来选择两支球队作为输赢预测的查询条件。下拉列表框的数据都从球队列表对象中取得，并通过 Thymeleaf 的标签语言使用一条循环语句 th:each 输出。

超链接“查询”的控件标识设定为 searchBtn，它的事件响应的实现将由脚本文件 win_loss.js 来设计。

代码清单 6-48　输赢预测的查询表单设计

```
<form id="queryForm" method="get">
    <div class="radiusGrayBox782">
        <div class="radiusGrayTop782"></div>
        <div class="radiusGrayMid782">
            <div class="dataSearchBox forUserRadius">
                <ul>
```

```
                <li>
                    <label class="preInpTxt f-left">甲队</label>
                    <div class="selectDownMode f-left w-206">
                        <input type="text"  value="请选择"/>
                        <select class="selectMode" name="t1" id="t1">
                            <option value="0">请选择</option>
                            <option th:each="team:${teams}" th:value=
"${team.name}"
th:text="${#strings.length(team.name) >20?#strings.substring (team.name,
0,20)+'...':team.name}"
                                    ></option>
                        </select>
                    </div>
                </li>
                <li>
                    <label class="preInpTxt f-left">乙队</label>
                    <div class="selectDownMode f-left w-206">
                        <input type="text"  value="请选择"/>
                        <select class="selectMode" name="t2" id="t2">
                            <option value="">请选择</option>
                            <option th:each="team:${teams}" th:value=
"${team.name}"
th:text="${#strings.length(team.name) >20?#strings.substring (team.name,
0,20)+'...':team.name}"
                                    ></option>
                        </select>
                    </div>
                </li>
                <li>
                    <a href="javascript:void(0)" class="blueBtn-62X30 f-
right" id="searchBtn">查询</a>
                </li>
            </ul>
        </div>
    </div>
</div>
</form>
```

输赢预测的列表设计如代码清单 6-49 所示。

这里主要使用一个表格控件实现了输赢预测列表数据显示的功能。

表格控件的表头部分设定了一个标识符 theadContent，表体部分设定了一个标识符 tbodyContent，这里的数据都将由脚本文件 win_loss.js 中的设计方法来填充。

代码清单 6-49　输赢预测的表格控件设计

```
<div id="netContent"></div>
  <div class="dataDetailList mt-12">
      <table class="dataListTab">
          <thead id="theadContent">
          </thead>
          <tbody id="tbodyContent">
          </tbody>
      </table>
   </div>
</div>
```

输赢预测页面引用的脚本文件 win_loss.js 的内容如代码清单 6-50 所示，它实现了页面请求，并安排了数据在页面上的显示方式。有关这个脚本的实现功能解释如下：

在页面上单击“查询”按钮时，即触发超链接 searchBtn 的 click 事件，通过这个事件将调用 pageaction()方法。

pageaction()方法通过访问当前位置的 URL“./win_loss”请求数据，然后通过调用 fillData()方法将数据显示在页面上。

fillData()方法首先通过页面的控件 netContent 输出平均净赢的数据，然后通过页面控件 theadContent，根据两支球队是否有交战记录，分别进行不同的数据显示方式处理。如果两队有交战记录，则按年度、赢场、输场的列表格式显示；否则按甲队与其他球队的比赛历史方式显示。

代码清单 6-50　输赢预测页面请求方法设计

```
$(function () {
    $('#searchBtn').click(function(){
        pageaction();
```

```
    });
});

var currentPageData = null ;
var pageaction = function(){
    $.get('./win_loss?t='+new Date().getTime(),{
        t1:$("#t1").val(), t2:$("#t2").val()
    },function(data){
        currentPageData = data;
        fillData(currentPageData);
    });
}

function fillData(data){
    $('#netContent').text( '甲队平均净赢: '+data.netwin.toFixed(2));
    var $head = $('#theadContent').empty();
    var head = "";
    head += '<tr>';
    if(data.met == 1){
        head += '<th>年度</th>';
        head += '<th>赢场</th>';
        head += '<th>输场</th>';
    }else{
        head += '<th>比赛历史: [甲队,其他球队],...</th>';
    }
    head += '</tr>';
    $head.append($(head));

    var $list = $('#tbodyContent').empty();
    $.each(data.content,function(k,v) {
        var html = "";
        html += '<tr> ';
        if(data.met == 1) {
            html += '<td>' + (v.year == null ? '' : v.year) + '</td>' +
            '<td>' + (v.win == null ? '' : v.win) + '</td>' +
            '<td>' + (v.loss == null ? '' : v.loss) + '</td>';
        }else{
```

```
            html += '<td>' + (v.paths == null ? '' : v.paths) + '</td>';
        }
        html +='</tr>' ;

        $list.append($(html));
    });
}
```

完成了输赢预测的 Web 设计后，在进行查询时，将根据两支球队是否有交战记录，通过程序进行智能控制。如果两队有交战记录，则显示的效果如图 6-14 所示。

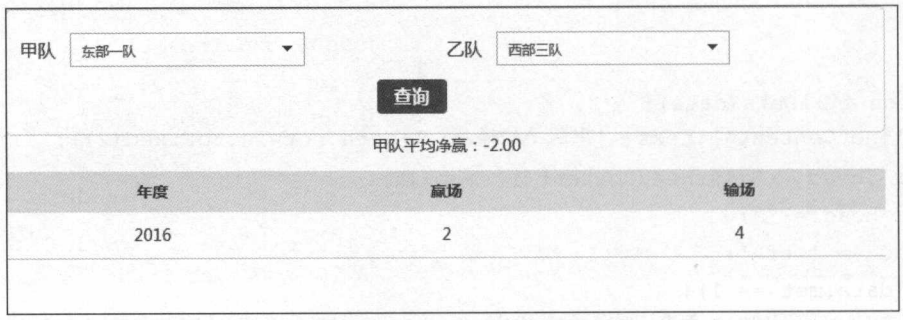


图 6-14　有交战记录的输赢预测显示效果

如果两队没有交战记录，则显示的效果如图 6-15 所示。图中的数据“1,4,4,3,4,2”表示甲队与其他球队有过三轮比赛，比赛的结果分别是 1 : 4、4 : 3、4 : 2，即输了一轮，赢了两轮。

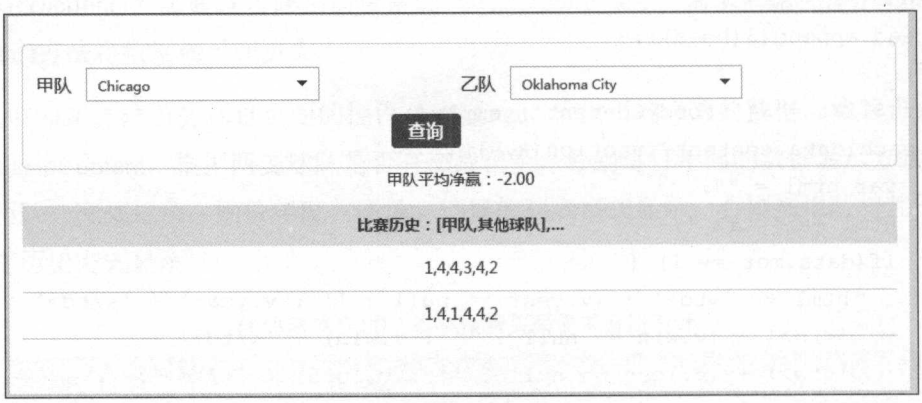


图 6-15　没有交战记录的输赢预测显示效果

从图 6-15 中还可以看出，甲队的平均净赢为–2.00，这表示甲队与乙队相比，输的可能性大一些。

6.12　使用 GraphGists 的测试数据

完成了 Web 设计之后，我们也可以在应用中使用原来 GraphGists 的数据看一看使用这些数据所展示的效果。

在 Neo4j 客户端的 Web 控制台上运行 GraphGists 生成测试数据的查询语句，即可生成相关的数据。GraphGists 生成测试数据的查询语句也可以从工程的如下路径中取得：

https://github.com/mr-csj/neo4j-prediction/blob/master/data/src/test/doc/TestData.cql

数据生成后，查询胜率排名的显示效果如图 6-16 所示。

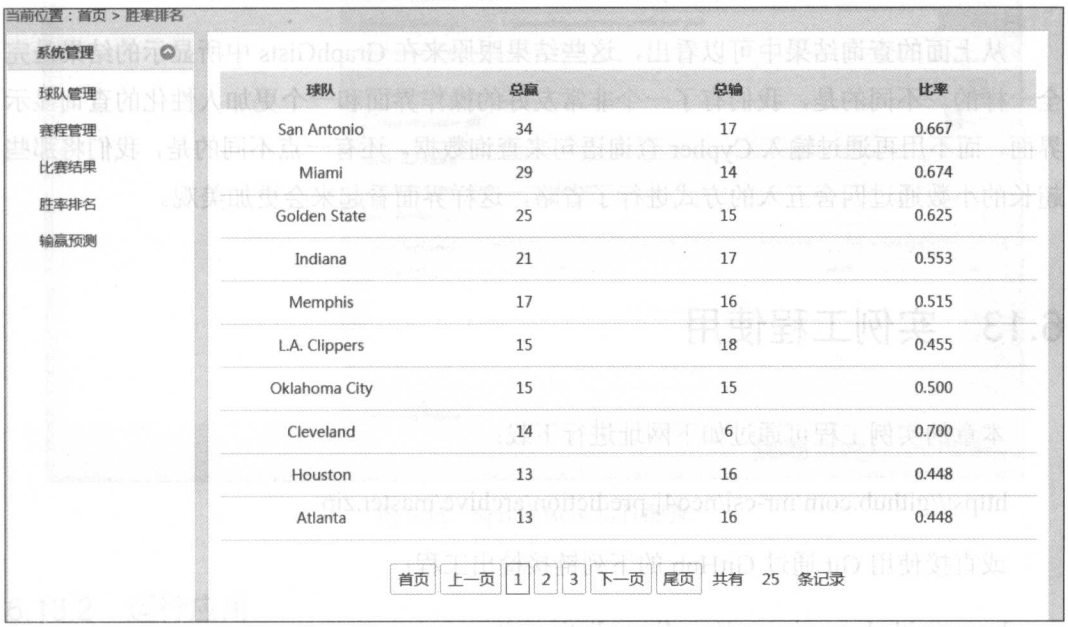


图 6-16　使用 GraphGists 的测试数据查询胜率排名的显示效果

查询输赢预测的显示效果如图 6-17 所示。

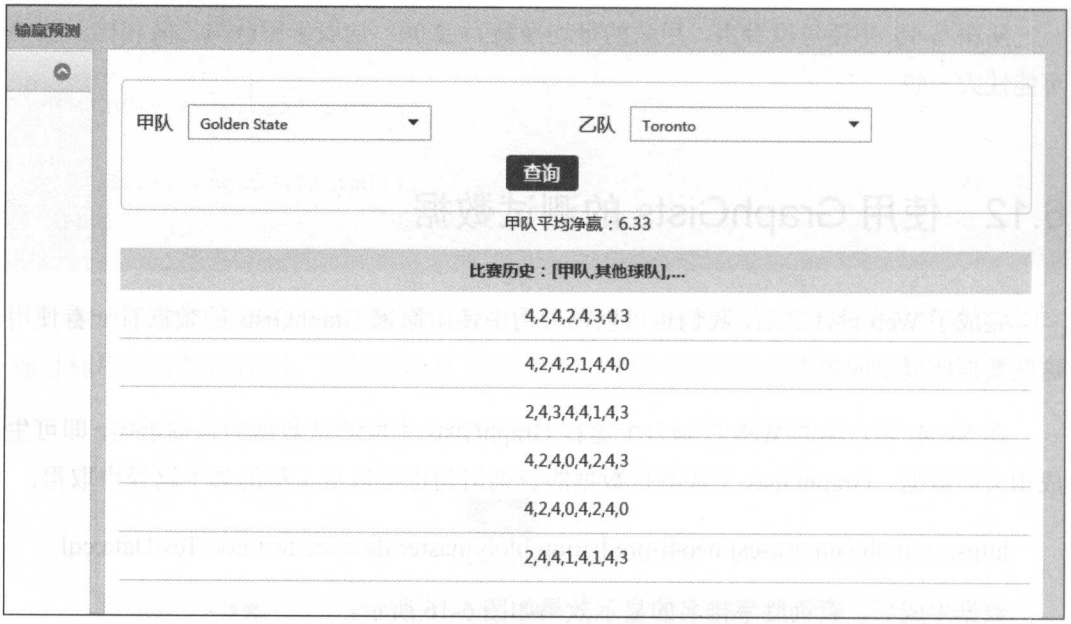


图 6-17　使用 GraphGists 的测试数据查询输赢预测的显示效果

从上面的查询结果中可以看出，这些结果跟原来在 GraphGists 中所显示的结果是完全一样的。不同的是，我们有了一个非常友好的操作界面和一个更加人性化的查询显示界面，而不用再通过输入 Cypher 查询语句来查询数据。还有一点不同的是，我们将那些超长的小数通过四舍五入的方式进行了省略，这样界面看起来会更加美观。

6.13　实例工程使用

本章的实例工程可通过如下网址进行下载：

https://github.com/mr-csj/neo4j-prediction/archive/master.zip

或直接使用 Git 通过 GitHub 的下列链接检出工程：

https://github.com/mr-csj/neo4j-prediction.git

6.13.1　工程配置

检出实例工程后，可以打开 dada 模块 resources 目录中的配置文件 ogm.properties，根据自己数据库的设置，配置连接数据库的 URL、用户名和密码等参数。

如果你使用的开发工具是 IntelliJ IDEA，则可以增加一个 Spring Boot 运行配置，用来启动和运行 web 模块，如图 6-18 所示。如果使用其他开发工具，则也可以参照这种方法进行配置，或者使用 Maven 工具对工程打包后再运行。

另外，对于 data 模块中的测试程序，也可以参照上面的方法增加一个 JUnit 配置，用来运行数据库设计的测试验证。

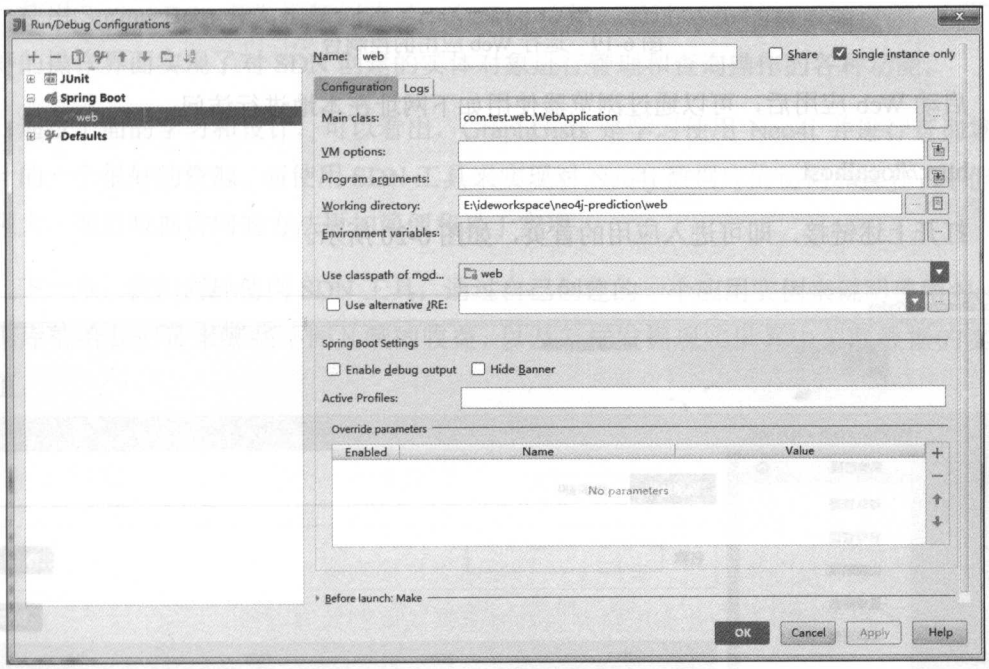


图 6-18　Spring Boot 运行配置

6.13.2　运行应用

运行上面的 Spring Boot 配置，即可启动 Web 应用，如图 6-19 所示。

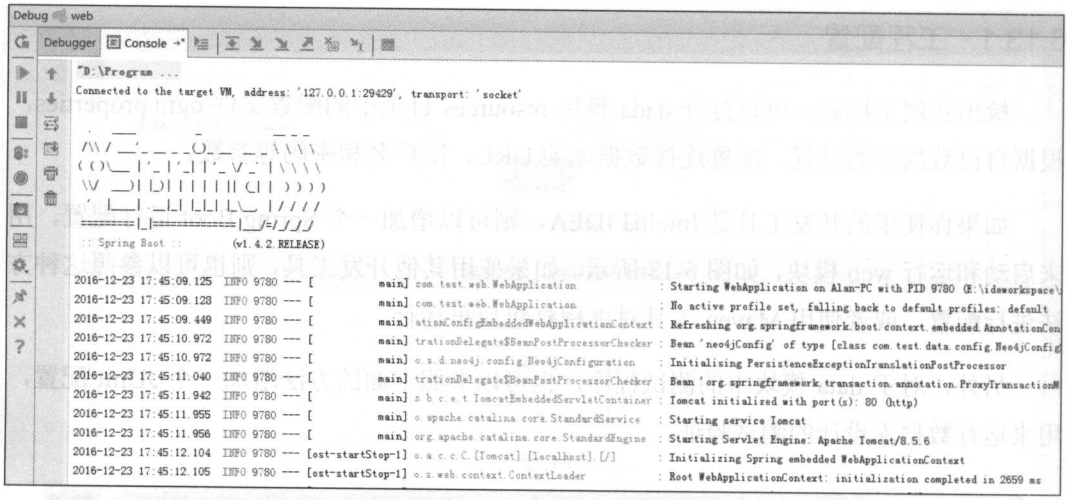


图 6-19　运行 Web 应用的控制台

启动 Web 应用后，可以通过浏览器使用如下网址在本地进行访问：

http://localhost

打开上述链接，即可进入应用的首页，如图 6-20 所示。

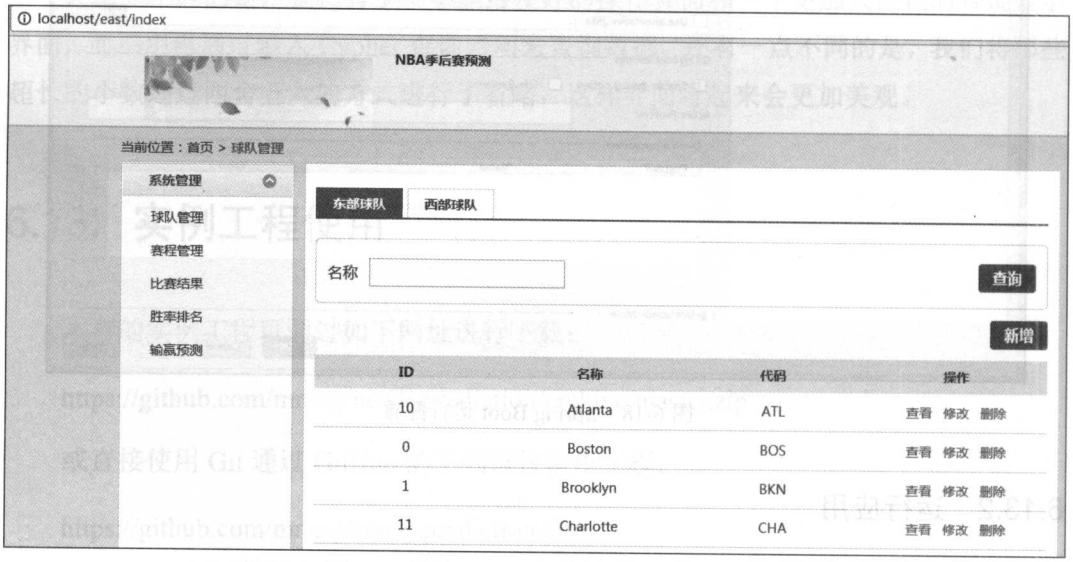


图 6-20　应用首页

6.14　小结

本章主要内容包括：

学习使用了 GraphGists 中一个典型的案例——NBA 季后赛预测。

从 GraphGists 案例中提取出数据模型，使用 SDN 进行节点和关系建模，并实现了建模的实体持久化。同时，学习使用了 GraphGists 中的查询算法，设计了各种存储库接口，实现了数据访问和各种预测算法设计。

使用 Spring Boot 开发框架创建了一个 Web 应用，通过 Web 的前后端设计，以一个友好的操作界面实现了对 SDN 创建的实体对象进行管理和查询操作的各种功能。

通过上面的学习和设计，可以看出，GraphGists 是学习使用 Neo4j 并进行查询算法设计的一个很好的资源。而使用 SDN 工具来实现对 Neo4j 数据库的访问和操作，不但功能强大，而且数据访问的方法更加简便和易于实现。

下一章，我们同样使用 SDN 工具，通过自己创建的一个应用实例来说明使用 Neo4j 数据库能给我们带来哪些不同凡响的收益，以及怎样发掘现实世界中关联数据的实用价值。

第7章
应用实例二：电影社区推荐引擎

在上一章中，我们通过 GraphGists 的一个典型案例学习了使用 Neo4j 进行项目工程的设计和开发。本章我们将根据实际生活中的一个应用场景，即根据一个电影社区来设计一个推荐引擎。

在现实生活的一个社区网络中存在各种各样的关系，如复杂的人际关系、各种商业关系等。使用 Neo4j 来存储和管理社区网络的关联数据，不但具有实时查询的高性能表现，而且可以很容易地发现数据的商业价值所在。

7.1 应用背景分析

电影社区是庞大的社区网络中一个很小的组成部分，为了节省篇幅，我们就使用这样一个小社区来演示和说明如何在实际应用中使用 Neo4j 数据库。通过这样一个小项目，希望能起到举一反三的效果。实际上，只要通过扩展数据模型，就可以将这个小项目扩展为一个能够表现复杂的业务逻辑并且具有丰富功能的大型项目。

在一个电影社区中存在很多电影观众，观众与观众之间或者观众与电影之间必定存在很多关系，比如观众之间的朋友关系、观众观看电影的关系、影院放映电影的关系、观众对电影进行评价的关系等。这些关系体现了现实世界中事物相互联系的本质。反过来说，现实世界中的关联数据本来就无所不在，就看你有没有去发现而已。搜集这些关联数据，然后通过 Neo4j 来管理这些数据，就可以很容易地发现关联数据的实用价值。

7.1.1　发现商业价值

电影社区的数据时刻发生着变化，哪些数据会具有价值呢？有变化就会有发展，变化的数据就最有价值。电影在不断地推出和上映，对于一名观众来说，在所有这些正在上映的电影中，他有几部未曾看过，在未看过的电影中，哪一部最先值得去观看？这就要看一看那些正在上映的电影中，观看过的观众是怎么对其做出评价的，然后选择一部大家都给了好评的电影来观看，肯定是最值得的。所以，观众对电影的评价关系就是最有价值的数据。

如果影院把一部观众评分最高的电影推荐给所有未看过这部电影的活跃观众，这对于一家影院来说，不就具有巨大的商业价值吗？这应该是时下所说的精准营销的做法。试想一下，如果你作为一个观众，观看了一部评分很高的电影之后觉得很不错，你也可以把它推荐给你的朋友来观看，那么，你的朋友肯定会对你满怀感激之情。所以，不但观众对电影的评分关系有价值，观众观看电影的关系和观众的朋友关系同样具有价值。

7.1.2　建立数据模型

根据上面的分析，我们抽象出电影社区的节点和关系对象，建立电影社区推荐引擎的数据模型，如图 7-1 所示。图中具有 4 个节点，分别是 Show、Cinema、Movie 和 Person，分别表示节目、影院、电影和观众；同时具有 5 个关系，分别是 SITE、BLOCKBUSTER、VISITED、RATED 和 FRIEND_OF，分别表示地点、放映、观看、评分和朋友。其中，节点和关系的连接方式是：

- 节目用地点关系连接影院，表示在哪家影院上映电影。
- 节目用放映关系连接电影，表示放映了哪部电影。
- 观众用观看关系连接节目，表示观看了哪个节目。
- 观众用评分关系连接电影，表示对哪部电影进行过评分。
- 观众用朋友关系连接自己，表示观众之间成为朋友。

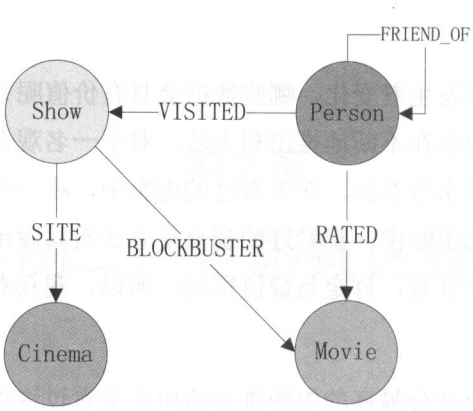


图 7-1　电影社区推荐引擎数据模型

7.2　数据对象建模

上面的数据模型设计可以很方便地使用 SDN 来建模。其中，4 个节点对象，即观众、影院、电影和节目都需要建模；在 5 个关系对象中，除评分关系因为需要保存评分的星级等属性，需要进行单独建模之外，其他 4 个关系，即地点、放映、观看、朋友，可以在创建节点时一起建模，无须为其进行单独建模。

7.2.1　节点建模

这里有观众、影院、电影和节目 4 个节点，下面使用 SDN 分别对这 4 个节点进行建模。

1. 观众节点建模

观众节点建模的实现方法如代码清单 7-1 所示。观众节点具有 id、name、sex、create 等属性，分别表示节点标识、观众名字、性别和创建时间。程序设计的方法说明如下：

程序中使用类作用域的注解@NodeEntity 设定类 Person 是一个节点实体对象，并将默认使用类的名字作为节点的标签。

属性注解@GraphId 设定了长整型的属性 id 为这个节点的唯一标识，它的值将由数据库自动生成。

属性注解@DateLong 设定了日期类型的属性 create 将使用长整型进行数据转换来保存数据，并且使用注解@DateTimeFormat 设定了数值在页面上进行转换的显示格式。

其他名字和性别属性省略了 OGM 的注解设定，将默认使用 Java 的数据类型作为数据库的数据类型。

程序中用对象 Set<Person>定义的集合对象 friends 使用注解@Relationship 设定为关系对象，其中关系类型指定为 FRIEND_OF，表示这是一种朋友关系，并把关系的方向设定为发出方向，同时这个关系还是一个多对多关系，使用注解@JsonIgnore 来防止数据的递归调用。

评分集合对象 ratings 的定义也使用注解@Relationship 来设定，并指定关系类型为 RATED，表示这是一个评分关系，关系的方向没有设定，将使用默认的发出方向。

观看集合对象 visitors 的定义也使用注解@Relationship 来设定，并指定关系类型为 VISITED，表示这是一个观看关系，同时把关系的方向指定为发出方向。

方法 beFriend()使用传入的另一个观众参数来创建观众的朋友关系。

方法 addVistiter()使用传入的节目参数来创建观众观看节目的关系。

方法 rate()使用传入的电影、星级和评论等参数来创建观众给电影评分的关系。

代码清单 7-1　观众节点建模

```
package com.test.data.domain;
......
@NodeEntity
public class Person {
        @GraphId
        private Long id;
        private String name;
        private int sex;
        @DateLong
        @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
```

```
    private Date create;

    @Relationship(type = "FRIEND_OF", direction = Relationship.OUTGOING)
    @JsonIgnore
    private Set<Person> friends = new HashSet<>();
    @Relationship(type = "RATED")
    private Set<Rating> ratings = new HashSet<>();
    @Relationship(type = "VISITED", direction = Relationship.OUTGOING)
    private Set<Show> visiters = new HashSet<>();

    public void beFriend(Person person) {
            friends.add(person);
    }

    public void addVistiter(Show show){
            visiters.add(show);
    }

    public Rating rate(Movie movie, int stars, String comment) {
            Rating rating = new Rating(this, movie, stars, comment);
            ratings.add(rating);
            return rating;
    }
.....
}
```

2. 影院节点建模

影院节点建模的实现方法如代码清单 7-2 所示。影院节点具有 id、name、city 和 capacity 属性，分别表示节点标识、名字、城市和影院容量。程序中使用注解@NodeEntity 设定了类 Cinema 为一个节点实体，并将默认使用类名作为节点的标签。属性注解 @GraphId 设定了属性 id 为节点的唯一标识，它的值将由数据库自动生成。其他属性都使用 Java 的数据类型作为数据库的数据类型，所以省略了类型的设定。

代码清单 7-2　影院节点建模

```
package com.test.data.domain;
```

```
import org.neo4j.ogm.annotation.GraphId;
import org.neo4j.ogm.annotation.NodeEntity;

@NodeEntity
public class Cinema {
        @GraphId
        private Long id;
        private String name;
        private String city;
        private int capacity;

        public Cinema() {
        }
......
}
```

3. 电影节点建模

电影节点建模的实现方法如代码清单 7-3 所示。电影节点具有 id 和 name 两个属性，分别表示节点标识和名字，同时还有一个集合关系对象 ratings，表示观众对电影的评分。程序中使用注解@NodeEntity 设定了类 Movie 为一个节点实体，并将默认使用类名作为节点的标签。属性注解@GraphId 设定了属性 id 为节点的唯一标识，它的值将由数据库自动生成。name 属性将使用 Java 的数据类型作为数据库的数据类型，所以省略了类型的设定。关系对象 ratings 使用注解@Relationship 来设定，并将类型指定为 RATED，表示这是一个评分关系，同时设定了关系的方向为进入方向。

代码清单 7-3　电影节点建模

```
package com.test.data.domain;

import org.neo4j.ogm.annotation.GraphId;
import org.neo4j.ogm.annotation.NodeEntity;
import org.neo4j.ogm.annotation.Relationship;

import java.util.HashSet;
import java.util.Set;
```

```
@NodeEntity
public class Movie{
    @GraphId
    private Long id;
    private String name;
    @Relationship(type = "RATED", direction = Relationship.INCOMING)
    private Set<Rating> ratings = new HashSet<>();

    public Movie() {
    }
    ......
}
```

4. 节目节点建模

节目节点建模的实现方法如代码清单 7-4 所示。节目节点具有 id、name 和 create 属性，分别表示节点标识、名字和日期。程序设计方法的说明如下：

程序中使用注解@NodeEntity 设定了类 Show 为一个节点实体，并将默认使用类名作为节点的标签。

属性注解@GraphId 设定了属性 id 为节点的唯一标识，它的值将由数据库自动生成。

name 属性将使用 Java 的数据类型作为数据库的数据类型，所以省略了类型的设定。

属性注解@DateLong 设定了日期类型的属性 create 将使用长整型进行数据转换来保存数据，并且使用注解@DateTimeFormat 设定了数值在页面上进行转换的显示格式。

影院对象 cinema 使用注解@Relationship 来设定它是一个关系，并将类型指定为 SITE，表示这是一个地点关系，同时设定了关系的方向为发出方向。

电影对象 movie 使用注解@Relationship 来设定它是一个关系，并将类型指定为 BLOCKBUSTER，表示这是一个放映关系，同时设定了关系的方向为发出方向。

代码清单 7-4　节目节点建模

```
package com.test.data.domain;
```

```
import org.neo4j.ogm.annotation.GraphId;
import org.neo4j.ogm.annotation.NodeEntity;
import org.neo4j.ogm.annotation.Relationship;
import org.neo4j.ogm.annotation.typeconversion.DateLong;
import org.springframework.format.annotation.DateTimeFormat;

import java.util.Date;

@NodeEntity
public class Show {
    @GraphId
    private Long id;
    private String name;
    @DateLong
    @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    private Date create;
    @Relationship(type = "SITE", direction = Relationship.OUTGOING)
    private Cinema cinema;
    @Relationship(type = "BLOCKBUSTER", direction = Relationship.OUTGOING)
    private Movie movie;

    public Show() {
    }
   ......
}
```

7.2.2　关系建模

电影社区推荐引擎数据模型中的 5 个关系只有评分关系需要单独建模。评分关系建模的实现方法如代码清单 7-5 所示。评分关系具有 id、stars、comment 和 create 4 个属性，分别表示关系实体标识、星级、评论和日期；同时还具有一个开始节点 person 和一个结束节点 movie。程序设计的方法说明如下：

程序中使用注解@RelationshipEntity 设定了类 Rating 为一个关系实体，并将关系类型指定为 RATED，表示评分关系。

属性注解@GraphId 设定了属性 id 为关系实体的唯一标识，它的值将由数据库自动生成。

属性注解@DateLong 设定了日期类型的属性 create 将使用长整型进行数据转换来保存数据，并且使用注解@DateTimeFormat 设定了数值在页面上进行转换的显示格式。

其他属性将使用 Java 的数据类型作为数据库的数据类型，所以省略了类型的设定。

注解@StartNode 设定了 Person 对象 person 为一个开始节点，同时使用注解@JsonBackReference 来防止数据的递归调用。

注解@EndNode 设定了 Movie 对象 movie 为一个结束节点，同时也使用注解@JsonBackReference 来防止数据的递归调用。

方法 Rating()使用的传入参数分别是观众、电影、星级和评论，用来生成一个观众对电影的评分关系。

代码清单 7-5　评分关系建模

```
package com.test.data.domain;

import com.fasterxml.jackson.annotation.JsonBackReference;
import org.neo4j.ogm.annotation.EndNode;
import org.neo4j.ogm.annotation.GraphId;
import org.neo4j.ogm.annotation.RelationshipEntity;
import org.neo4j.ogm.annotation.StartNode;
import org.neo4j.ogm.annotation.typeconversion.DateLong;
import org.springframework.format.annotation.DateTimeFormat;

import java.util.Date;

@RelationshipEntity(type = "RATED")
public class Rating{
    @GraphId
    private Long id;
    @StartNode
    @JsonBackReference
    private Person person;
```

```
    @EndNode
    @JsonBackReference
    private Movie movie;
    private int stars;
    private String comment;
    @DateLong
    @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    private Date create;

    public Rating() {}

    public Rating(Person person, Movie movie, int stars, String comment) {
        this.person = person;
        this.movie = movie;
        this.stars = stars;
        this.comment = comment;
    }
    ......
}
```

7.3　存储库接口设计

对实体对象进行建模之后，接下来要做的就是实体的持久化和数据访问的设计，这可以使用 SDN 通过定义存储库接口来实现。除关系实体可以不用创建存储库接口之外，上面创建的几个节点实体都需要创建存储库接口。

7.3.1　影院存储库接口设计

影院存储库接口设计如代码清单 7-6 所示。这个存储库接口的设计比较简单，只是继承了 SDN 的存储库接口 GraphRepository，实现了实体对象 Cinema 的持久化设计。其中，使用注解@Repository 设定了这个接口是一个存储库接口。

代码清单 7-6 影院存储库接口设计

```
package com.test.data.repository;

import com.test.data.domain.Cinema;
import org.springframework.data.neo4j.repository.GraphRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface CinemaRepository extends GraphRepository<Cinema> {
}
```

7.3.2 电影存储库接口设计

电影存储库接口设计如代码清单 7-7 所示。这个接口设计的方法说明如下：

使用注解@Repository 设定接口为一个存储库接口。

通过继承 GraphRepository 接口，实现实体对象 Movie 的持久化设计。

声明方法 findRatingMovie()，通过 Cypher 查询算法返回电影评分排名的列表数据。

声明方法 findRatingMovieCount()，通过 Cypher 查询算法返回电影评分排名的列表数据中的记录总数。

上面两个声明方法中的 Cypher 查询语句将在后面的查询算法设计中进行解释。

代码清单 7-7 电影存储库接口设计

```
package com.test.data.repository;

import com.test.data.domain.Movie;
import org.springframework.data.neo4j.annotation.Query;
import org.springframework.data.neo4j.repository.GraphRepository;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import java.util.Map;
import java.util.Set;
```

```
@Repository
public interface MovieRepository extends GraphRepository<Movie> {
    //电影评分排名
    @Query("MATCH (:Person)-[r:RATED]->(m:Movie) " +
           "WITH m, COLLECT(r.stars) AS ratings " +
           "WITH m, ratings, REDUCE(s = 0, i IN ratings | s + i)*1.0 / SIZE(ratings)
AS stars " +
           "RETURN ID(m) AS id, m.name AS name, stars, SIZE(ratings) AS num " +
           "ORDER BY stars DESC, num DESC SKIP {skip} LIMIT {limit}")
    Set<Map<String,Object>> findRatingMovie(@Param("skip") int skip,
@Param("limit") int limit);

    //电影评分总数
    @Query("MATCH (:Person)-[r:RATED]->(m:Movie) " +
           "WITH m, COLLECT(r.stars) AS ratings " +
           "RETURN COUNT(m) AS count ")
    int findRatingMovieCount();
}
```

7.3.3　节目存储库接口设计

节目存储库接口设计如代码清单 7-8 所示。这个存储库接口的设计跟影院存储库接口的设计一样，只是简单继承了 GraphRepository 接口，从而实现了实体对象 Show 的持久化设计以及普通的 CURD 数据访问方法。

代码清单 7-8　节目存储库接口设计

```
package com.test.data.repository;

import com.test.data.domain.Show;
import org.springframework.data.neo4j.repository.GraphRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface ShowRepository extends GraphRepository<Show> {
}
```

7.3.4 观众存储库接口设计

观众存储库接口设计如代码清单 7-9 所示。接口的设计通过继承 GraphRepository 实现了实体对象 Person 的持久化设计，声明方法 findByIdNot()，返回不包含参数 id 的观众列表，并通过 Cypher 查询语言实现了数据访问的方法。这个接口还有一些声明方法在这里没有列举出来，因为在这些方法中都涉及查询算法设计的内容，我们将在后面一起进行说明。

代码清单 7-9 观众存储库接口设计

```
package com.test.data.repository;

import com.test.data.domain.Person;
import org.springframework.data.neo4j.annotation.Query;
import org.springframework.data.neo4j.repository.GraphRepository;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import java.util.Collection;
import java.util.Date;
import java.util.Set;

@Repository
public interface PersonRepository extends GraphRepository<Person> {
    @Query("MATCH (n:Person) WHERE ID(n) <> {id} RETURN n;")
    Iterable<Person> findByIdNot(@Param("id") Long id);
    ......
}
```

7.4 Cypher 查询算法设计

在上面的存储库接口设计中包含一些查询算法的设计，这些设计都使用 Cypher 查询语言来实现，主要包括两个方面的内容，即电影排名查询算法设计和电影推荐查询算法设计。

7.4.1　电影排名查询算法设计

电影排名查询算法设计如代码清单 7-10 所示。这个查询设计的算法解释如下：

模式匹配 MATCH 设定了查询的关系为观众对电影进行评分，同时为了方便引用定义了两个变量，分别为评分关系 r 和电影节点 m。

使用两个链接关键字 WITH 实现了查询的管道功能。第一个管道主要实现了将评分关系中的属性 stars 转换为一个集合对象 ratings。第二个管道通过累加函数 REDUCE()取出集合 ratings 中的每条数值进行累加，然后将累加的和除以集合 ratings 的总数，计算出一部电影在所有评分中的平均值。

在返回结果中按评分数值的大小进行倒序排序，并使用分页的方式输出结果，其中 num 表示参与评分的观众人数。

代码清单 7-10　电影排名查询算法设计

```
MATCH (:Person)-[r:RATED]->(m:Movie)
WITH m, COLLECT(r.stars) AS ratings
WITH m, ratings, REDUCE(s = 0, i IN ratings | s + i)*1.0 / SIZE(ratings) AS
stars
RETURN ID(m) AS id, m.name AS name, stars, SIZE(ratings) AS num
ORDER BY stars DESC,num DESC SKIP 0 LIMIT 10
```

如果在数据库中创建一些数据，执行上面的查询，则可以看到如图 7-2 所示的结果。

id	name	stars	num
0	终结者	4.5	30
33	源代码	4.5	20

图 7-2　电影排名查询结果

7.4.2　电影推荐查询算法设计

电影推荐查询算法设计，按推荐的目标不同，可以分为两种，分别为推荐电影给观众的查询算法设计和推荐电影给朋友的查询算法设计。

推荐电影给观众的查询算法设计如代码清单 7-11 所示。这个设计主要使用了过滤条件过滤掉观看过所推荐电影的观众，然后以分页的方式返回观众列表。

代码清单 7-11　推荐电影给观众的查询算法设计

```
MATCH (o:Person) WHERE NOT (o)-[:VISITED]->()-[:BLOCKBUSTER]->({name:'终结者'})
RETURN o AS person SKIP 0 LIMIT 10
```

执行上述查询，可以看到如图 7-3 所示的结果，即以分页方式列出了未看过电影“终结者”的所有观众。

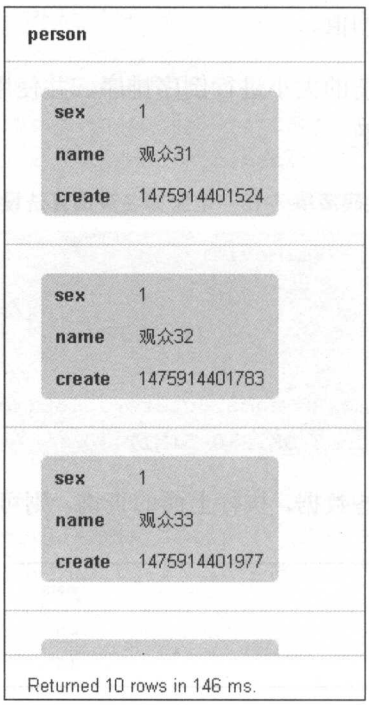


图 7-3　推荐电影给观众的查询结果

推荐电影给朋友的查询算法设计如代码清单 7-12 所示。这个算法设计在模式匹配中使用“[:FRIEND_OF*1..3]”限制了只找朋友关系中路径深度为 1～3 的朋友，而在过滤条件的设计中，跟上面推荐电影给观众的设计有点相似，这里只增加了一个条件“friend.name <> '观众 31'”，即避免在观众的朋友中返回自己。

代码清单 7-12　推荐电影给朋友的查询算法设计

```
MATCH (a:Person{name:'观众 31'})-[:FRIEND_OF*1..3]->(friend:Person) WHERE
friend.name <> '观众 31' AND NOT
(friend)-[:VISITED]->()-[:BLOCKBUSTER]->({name:' Titanic'}) RETURN friend
ORDER BY friend.name SKIP 0 LIMIT 10
```

执行上述查询，可返回如图 7-4 所示的结果，即实现了为“观众 31”比较接近的朋友推荐电影“Titanic”的查询方法。

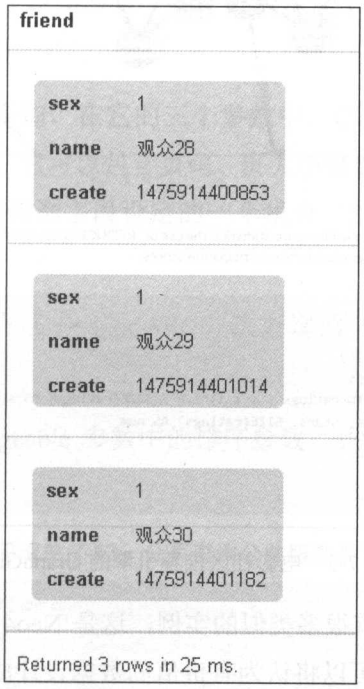


图 7-4　推荐电影给朋友的查询结果

除上面一些查询算法的设计之外，通过电影社区推荐引擎这个数据模型，也许你还可以发掘到其他更有价值的东西，从而设计出不同的查询算法。

本实例也发布在 GraphGists 门户网站上，可以通过下列链接查看：

http://portal.graphgist.org/graph_gists/movie-recommended-community

打开上述链接的效果如图 7-5 所示。

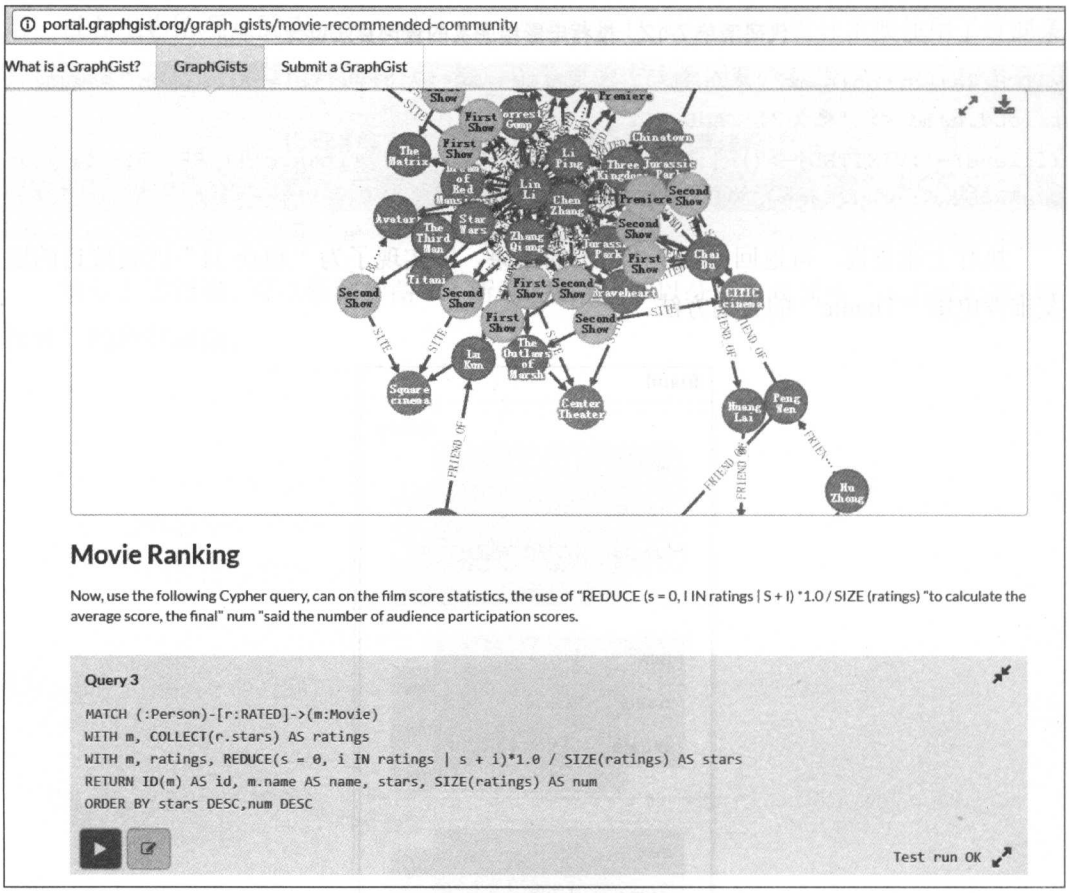


图 7-5　电影社区推荐引擎的 GraphGists

GraphGists 门户网站上有很多类似的实例，这是 Neo4j 爱好者和开发者极好的学习资源。如果你有兴趣，则也可以将认为有价值的数据设计成一个 GraphGists 发布在这个网站上。

7.5　数据访问服务类设计

有时候，我们可能需要使用一些特殊的手段来扩展 SDN 的设计，如使用 OGM 的会话提供类似于使用底层的方法来访问数据库，可以设计出功能更加强大的数据访问服务

类。同时对于上面我们创建的几个实体对象的数据访问方法，也可以为其创建一个服务类进行封装，提供统一的访问方法。

7.5.1　分页查询公共服务类

分页查询公共服务类的设计如代码清单 7-13 所示。这个公共服务类可以为任何节点提供功能更加强大的分页查询服务。这个类的设计方法解释如下：

在类和方法的定义中使用了泛型参数“<T>”，表示这个服务类可以为任何节点对象提供分页访问的服务。

findAll()方法实现了分页查询，在它的三个参数中，Class<T>为调用的实体对象；Pageable 是一个分页参数接口，它可以包含页码、页大小设定和排序字段等参数；Filters 是一个集合对象，它可以提供查询字段设定的一个集合。然后通过调用 OGM 会话的 loadAll()方法查询分页数据。

updatePage()方法将上述方法中 loadAll()方法查询的分页数据转换成一个页对象返回。

convert()方法用来转换 Pageable 参数中的排序参数，让它成为数据库能够识别的查询参数。

代码清单 7-13　分页查询公共服务类

```
package com.test.data.service;
......
@Service
public class PagesService<T> {
   @Autowired
   private Session session;
   public Page<T> findAll(Class<T> clazz, Pageable pageable, Filters
filters){
      Collection data = this.session.loadAll(clazz, filters, convert
(pageable.getSort()), new Pagination(pageable.getPageNumber(), pageable.
getPageSize()), 1);
      int count = this.session.loadAll(clazz, filters, 1).size();
```

```
        return updatePage(pageable, new ArrayList(data), count);
    }

    private Page<T> updatePage(Pageable pageable, List<T> results, int total) {
        return new PageImpl(results, pageable, (long)total);
    }

    private SortOrder convert(Sort sort) {
        SortOrder sortOrder = new SortOrder();
        if(sort != null) {
            Iterator var3 = sort.iterator();

            while(var3.hasNext()) {
                Sort.Order order = (Sort.Order)var3.next();
                if(order.isAscending()) {
                    sortOrder.add(new String[]{order.getProperty()});
                } else {
                    sortOrder.add(SortOrder.Direction.DESC, new String[]{order.
getProperty()});
                }
            }
        }
        return sortOrder;
    }
}
```

7.5.2 数据访问服务类

我们在上面定义的几个实体对象的数据访问方法都可以通过设计一个数据访问服务类进行封装，从而提供功能更加强大的数据访问服务。

电影对象数据访问服务类的设计如代码清单 7-14 所示。这个服务类设计的实现方法解释如下：

程序中几个增、删、改、查的方法如 findById()、save()、delete()等都调用了电影存储库接口的相关方法。

findPage()方法使用了上面设计的分页查询公共服务类进行分页查询，其中，将排序的字段设定为使用节点的 ID 进行排序。在查询参数设定中，使用电影的 name 属性通过指定查询条件为 LIKE 进行模糊查询；使用电影的 create 属性可以设定查询电影的创建时间范围，即通过指定查询条件 GREATER_THAN 设定为大于查询参数的时间范围。

代码清单 7-14　电影对象数据访问服务类

```
package com.test.data.service;
......
@Service
@Transactional
public class MovieService {
    @Autowired
    private MovieRepository movieRepository;
    @Autowired
    private PagesService<Movie> moviePagesService;
    public Movie findById(Long id) {
        return movieRepository.findOne(id);
    }
    public Movie save(Movie movie) {
        return movieRepository.save(movie);
    }
    public void delete(Long id) {
        movieRepository.delete(id);
    }
    public Page<Movie> findPage(MovieQo movieQo){
        Pageable pageable = new PageRequest(movieQo.getPage(),
movieQo.getSize(), new Sort(Sort.Direction.ASC, "id"));

        Filters filters = new Filters();
        if (!StringUtils.isEmpty(movieQo.getName())) {
            Filter filter = new Filter("name", "*"+movieQo.getName()+"*");
            filter.setComparisonOperator(ComparisonOperator.LIKE);
            filters.add(filter);
        }
        if (!StringUtils.isEmpty(movieQo.getCreate())) {
            Filter filter = new Filter("create", movieQo.getCreate());
```

```
            filter.setComparisonOperator(ComparisonOperator.GREATER_THAN);
            filter.setBooleanOperator(BooleanOperator.AND);
            filters.add(filter);
        }
        return moviePagesService.findAll(Movie.class, pageable, filters);
    }
}
```

7.6 数据库连接配置

上面使用 SDN 完成了对象建模和实体化设计的工作，为了能在项目工程中使用我们的设计，还需要做一些数据库连接配置和 SDN 配置等方面的工作。

7.6.1 SDN 驱动的依赖引用

使用 SDN 连接数据库，除了使用默认的 HTTP 驱动，还可以使用嵌入式驱动和 Bolt 驱动。如果要使用这两种驱动方式，则需要在工程的 Maven 配置中引入其相关的驱动程序依赖。如代码清单 7-15 所示是 SDN 及其驱动包的依赖配置。其中，neo4j-ogm-embedded-driver 为嵌入式驱动，neo4j-ogm-bolt-driver 为 Bolt 驱动。配置中的驱动包版本号 2.0.5 与 Spring Boot 的版本 1.4.2 相适应。

代码清单 7-15　Neo4j 及其驱动依赖

```
<dependencies>
        <dependency>
            <groupId>org.springframework.data</groupId>
            <artifactId>spring-data-neo4j</artifactId>
        </dependency>
        <dependency>
            <groupId>org.neo4j</groupId>
            <artifactId>neo4j-ogm-embedded-driver</artifactId>
            <version>2.0.5</version>
        </dependency>
        <dependency>
```

```
        <groupId>org.neo4j</groupId>
        <artifactId>neo4j-ogm-bolt-driver</artifactId>
        <version>2.0.5</version>
    </dependency>
</dependencies>
```

7.6.2　连接数据库配置

SDN 的三种驱动方式对应的连接数据库的配置大同小异，如代码清单 7-16 所示。这里给出了三种驱动方式的连接配置方法，可供选择使用。其中，嵌入式的连接配置可以不需要用户名和密码。

代码清单 7-16　SDN 数据库驱动连接配置

```
compiler=org.neo4j.ogm.compiler.MultiStatementCypherCompiler
#Http
driver=org.neo4j.ogm.drivers.http.driver.HttpDriver
URI=http://localhost:7474

username = neo4j
password = 12345678

##Bolt
#driver=org.neo4j.ogm.drivers.bolt.driver.BoltDriver
#URI=bolt://localhost

##Embedded
#driver=org.neo4j.ogm.drivers.embedded.driver.EmbeddedDriver
#URI=file:///test/graph.db
```

7.6.3　SDN 配置

为了启用 SDN，还必须在工程中增加一个 SDN 配置。如代码清单 7-17 所示是我们设计的 SDN 配置类。配置类 Neo4jConfig 继承了 SDN 的配置类 Neo4jConfiguration，启用了 SDN 的功能。其中，注解@EnableNeo4jRepositories 激活了我们定义的存储库接口，注解 @ComponentScan 扫描了我们定义的一些服务类，这里包括我们设计的分页查询公

共服务类和各个实体对象的数据访问服务类。程序中的一个重写方法 getSessionFactory() 通过指定包路径 com.test.data.domain，引用了我们定义的一些实体对象。

代码清单 7-17　SDN 配置

```
package com.test.data.config;

import org.neo4j.ogm.session.SessionFactory;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.neo4j.config.Neo4jConfiguration;
import
org.springframework.data.neo4j.repository.config.EnableNeo4jRepositories;
import
org.springframework.transaction.annotation.EnableTransactionManagement;

@Configuration
@ComponentScan(basePackages = "com.test.data.service")
@EnableNeo4jRepositories(basePackages = { "com.test.data.repository" })
@EnableTransactionManagement
public class Neo4jConfig extends Neo4jConfiguration {

    @Override
    public SessionFactory getSessionFactory() {
        return new SessionFactory("com.test.data.domain");
    }

}
```

7.7　数据库设计验证

使用 Neo4j 的开发过程可以很方便地通过测试驱动的方式进行。在上面使用 SDN 进行对象建模和存储库接口设计的过程中，如果你对某个功能的实现不是很确定，就可以使用测试进行验证。

现在，有关电影社区推荐引擎的数据库设计基本上完成了，我们很有必要来执行一个单元测试，以验证上面一些设计的合理性和正确性。

在这里，我们提供了一个测试用例，虽然不能全面覆盖上面的所有设计，但是大体上可以验证我们设计的工作成果。在这个实例中还未涉及一些功能的测试验证，如果需要，则也可以参照这个例子来设计。

这个测试用例包括三个方面的内容，分别为数据初始化、观众分页查询和观众模糊查询。

测试用例中数据初始化的设计如代码清单 7-18 所示。

程序中通过注解@Before 设定了 addData()方法将在其他所有测试方法开始之前执行，这个方法将生成电影社区的一些初始数据，即创建了一个影院节点“凤凰影院”、一个电影节点“终结者”、一个节目节点“终结者第一场”，同时通过创建关系，表明节目“终结者第一场”在地点“凤凰影院”放映了电影“终结者”；然后使用一条循环语句创建了 100 位观众，同时让每位观众都与前一位观众成为朋友，并观看了节目“终结者第一场”，而且给电影“终结者”进行了评分。其中，评分时指定电影的星级在 10 以内。

代码清单 7-18　测试用例中的数据初始化设计

```
@Before
   public void addData(){
      Cinema cinema = new Cinema();
      cinema.setName("凤凰影院");
      cinema.setCity("深圳");
      Movie movie = new Movie();
      movie.setName("终结者");
      Show show = new Show();
      show.setName("终结者第一场");
      show.setMovie(movie);
      show.setCinema(cinema);
      show.setCreate(new Date());
      for(int i=1; i<=100; i++){
         Person person = new Person();
         person.setName("观众" + i);
         person.setSex(1);
```

```
            person.setCreate(new Date());
            if(i > 1) {
                Person friend = personRepository.findByName("观众" + (i - 1));
                person.beFriend(friend);
            }
            person.addVistiter(show);
            person.rate(show.getMovie(),i % 10,"好看");
            personRepository.save(person);
        }
        Assert.notNull(show.getId());
}
```

测试用例中观众分页查询的设计如代码清单 7-19 所示。

观众分页查询首先使用 PersonQo 对象作为参数，分页参数使用默认的设定，即查询第 1 页，每页大小为 10 条数据，其他参数都不指定数值。然后调用观众数据访问服务类的 findPage()方法返回一个页对象 persons，并使用 Assert 验证对象 persons 是否非空，如果是则表示通过测试。最后使用一个 for 循环打印列表数据的简要信息。

代码清单 7-19　测试用例中的观众分页查询设计

```
@Test
public void getPage() {
    PersonQo personQo = new PersonQo();
    Page<Person> persons = personService.findPage(personQo);
    Assert.notNull(persons);
    for(Person person : persons.getContent()){
        log.debug("\n==========Person name={}, create={}", person. getName(),
person.getCreate());
    }
}
```

测试用例中观众模糊查询的设计如代码清单 7-20 所示。

程序中首先使用参数“*众*”调用观众存储库接口的 findByNameLike()方法执行对观众对象的模糊查询，即期望返回观众的名字中包含“众”字的所有观众。然后使用 Spring 的单元测试对象 Assert 对返回对象 persons 进行非空验证。

代码清单 7-20　测试用例中的观众模糊查询设计

```
@Test
public void gets() {
    Iterable<Person> persons = personRepository.findByNameLike("*众*");
    Assert.notNull(persons);
}
```

运行上面的测试用例，如果所有测试都通过，则说明我们上面的设计和配置大体上是正确的；否则可以根据错误提示，具体问题具体处理。为什么说大体上正确呢？这是因为这个测试用例并没有完全覆盖我们所有的数据库设计。

经过上面的测试，将在数据库中生成一些数据。如果通过 Neo4j 的 Web 控制台查看，就可以看到如图 7-6 所示的图数据。

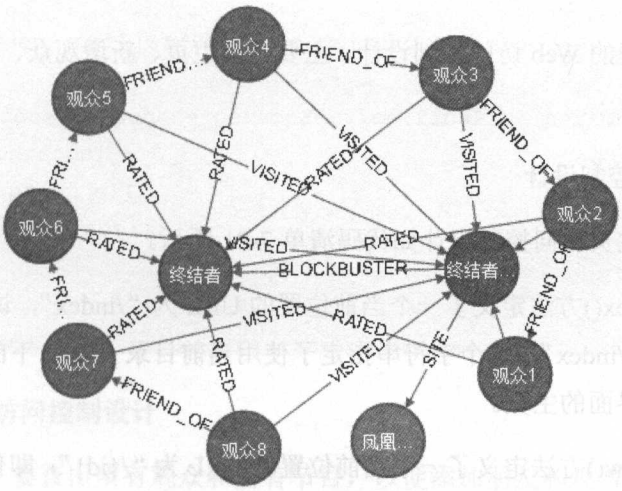


图 7-6　使用测试用例产生的图数据

7.8　Web 设计

上面通过 SDN 完成了电影社区推荐引擎的数据库设计，并通过一个测试用例完成了对数据库设计的验证。现在，我们使用一个 Web 应用来创建一个数据管理后台。通过管

理后台，可以模拟现实情况来录入和编辑电影社区中的一些数据。当然，真实运行的电影社区，它的数据来源可能具有各种不同的方式。例如，节目安排应该由各家影院来管理，而电影评分应该由观众自己来输入等。

在管理后台中，一般的操作就是数据实体对象的增、删、改、查，每个实体的数据管理操作方法基本上类似，这里我们将以观众实体为例，详细说明 Web 的访问控制设计和界面设计的方法与过程，其他几个实体对象如影院、节目、电影等的增、删、改、查设计可以参照这个设计来实现，就不再一一详述了。

而对于电影评分、电影排名和电影推荐等，由于其各具特殊性，所以编辑方法都有所不同，我们将对其进行单独说明。

7.8.1 访问控制设计

有关观众管理的 Web 访问控制设计，主要包括主页、新增观众、修改观众和删除观众等内容。

1. 主页访问控制设计

观众管理的主页访问控制设计如代码清单 7-21 所示。

程序中的 index()方法定义了一个当前位置的 URL 为“/index”，访问这个 URL 将返回字符串“person/index”，这个字符串指定了使用当前目录 person 下的 index.html 文件，即观众数据管理界面的主页。

程序中的 show()方法定义了一个当前位置的 URL 为“/{id}”，即使用观众节点的 ID 作为参数进行访问，访问这个 URL 将返回字符串“person/show”，这个字符串指定了使用当前目录 person 下的 show.html 文件，即查看一个观众的详情页面。其中，调用观众服务类的 findById()方法请求了一个观众对象的数据，用于在详情页面上进行显示。

程序中的 getList()方法定义了一个当前位置的 URL 为“/list”，访问这个 URL 将调用观众服务类的 findPage()方法，并返回一个包含观众列表数据的页对象。

代码清单 7-21　观众数据管理的主页访问控制设计

```
@RequestMapping("/index")
public String index() throws Exception{
    return "person/index";
}

@RequestMapping(value="/{id}")
public String show(ModelMap model,@PathVariable Long id) {
    Person person = personService.findById(id);
    model.addAttribute("person", person);
    return "person/show";
}

@RequestMapping(value = "/list")
@ResponseBody
public Page<Person> getList(PersonQo personQo) {
    try {
        Page<Person> persons = personService.findPage(personQo);
        return  persons;
    }catch (Exception e){
        e.printStackTrace();
    }
    return null;
}
```

2. 新增观众访问控制设计

新增观众时，要查出所有观众和所有节目，以便添加朋友和观看节目关系。保存时直接按返回的观众对象 Person 进行保存即可。

新增观众的访问控制设计如代码清单 7-22 所示。

程序中的 create()方法定义了一个相对路径的 URL“/new”，访问这个 URL 将返回字符串“person/new”，这个字符串指定了使用当前目录 person 下的 new.html 文件，即新增观众的页面设计。其中，分别调用观众存储库接口和节目存储库接口的 findAll()方法请求了观众列表和节目列表的数据，作为新增观众的参数。

程序中的 save()方法定义了一个当前位置的 URL “/save”，访问这个 URL 将调用观众服务类的 save()方法请求保存数据，如果保存成功则返回 1。

代码清单 7-22　新增观众的访问控制设计

```
@RequestMapping("/new")
public String create(ModelMap model,Person person){
    Iterable<Person> friends = personRepository.findAll();
    Iterable<Show> visiters = showRepository.findAll();
    model.addAttribute("friends",friends);
    model.addAttribute("visiters",visiters);
    model.addAttribute("person", person);
    return "person/new";
}

@RequestMapping(value="/save", method = RequestMethod.POST)
@ResponseBody
public String save(Person person, HttpServletRequest request) throws Exception{
    person.setCreate(new Date());
    personService.save(person);
    logger.info("新增->ID="+ person.getId());
    return "1";
}
```

注意：这里调用存储库接口的 findAll()方法中返回的对象都不使用集合 List 或 Set，而使用了 Iterable 接口。因为这个接口对内存的使用进行了优化，用于数据查询中，相当于延迟加载的效果。所以，在 SDN 存储库接口的所有 findAll()方法中，返回的集合对象几乎都使用了 Iterable 接口。

3. 修改观众访问控制设计

观众管理中修改观众的访问控制设计如代码清单 7-23 所示。

程序中的第一个 update()方法定义了一个当前位置的 URL “/edit/{id}”，即使用观众的 ID 作为参数，访问这个 URL 将返回字符串 “person/edit”，这个字符串指定了使用当前目录 person 下的 edit.html 文件，即修改观众的页面设计。其中，分别调用了观众服务类、观众存储库接口和节目存储库接口，为修改观众数据准备了几个参数。参数 fids 为

观众已有的朋友 ID 集合，参数 friends 为观众可以增加为新朋友的其他观众列表，参数 visiters 为所有节目列表，参数 vids 为观众看过的节目 ID 集合。

程序中的另一个 update()方法定义了一个当前位置的 URL“/update”，访问这个 URL 将调用观众服务类的 save()方法请求保存数据，如果保存成功则返回 1。

代码清单 7-23　修改观众的访问控制设计

```
@RequestMapping(value="/edit/{id}")
public String update(ModelMap model,@PathVariable Long id){
    Person person = personService.findById(id);
    Iterable<Person> friends = personRepository.findByIdNotIn(id);
    Iterable<Show> visiters = showRepository.findAll();

    //观众的朋友
    Set<Long> fids = new HashSet<>();
    for(Person friend : person.getFriends()){
        fids.add(friend.getId());
    }

    //观众看过的节目
    Set<Long> vids = new HashSet<>();
    for(Show show : person.getVisiters()){
        vids.add(show.getId());
    }

    model.addAttribute("friends",friends);
    model.addAttribute("visiters",visiters);
    model.addAttribute("person", person);
    model.addAttribute("fids",fids);
    model.addAttribute("vids",vids);
    return "person/edit";
}

@RequestMapping(method = RequestMethod.POST, value="/update")
@ResponseBody
public String update(Person person) throws Exception{
    personService.save(person);
```

```
    logger.info("修改->ID="+ person.getId());
    return "1";
}
```

4. 删除观众访问控制设计

观众管理中删除观众的访问控制设计如代码清单 7-24 所示。

程序中的 delete()方法定义了一个当前位置的 URL “/delete/{id}”，即使用观众的 ID 作为参数，访问这个 URL 将调用观众服务类的 delete()方法请求删除数据，如果删除成功则返回 1。

代码清单 7-24　删除观众的访问控制设计

```
@RequestMapping(value="/delete/{id}",method = RequestMethod.GET)
    @ResponseBody
    public String delete(@PathVariable Long id) throws Exception{
        personService.delete(id);
        logger.info("删除->ID="+id);
        return "1";
    }
```

7.8.2　界面设计

观众管理的界面设计包括主页界面设计、详情查看界面设计、新增数据界面设计、修改数据界面设计和删除数据确认设计等内容。

1. 主页界面设计

访问观众管理的主页界面设计主要包括一个页面设计文件 index.html 和一个 JavaScript 脚本文件 list.js。

其中，页面设计文件的内容主要包括 4 个方面，分别为页头设计、查询表单设计、新增观众的超链接设计和列表设计。

观众管理主页中的页头设计如代码清单 7-25 所示。这里主要实现了一些样式和脚本设计的引用，各个引用的功能说明如下：

- jquery.pagination.js 和 pagination.css 分别是一个分页控件的插件和分页控件的样式设计。
- WdatePicker.js 和 WdatePicker.css 分别是一个日期控件的插件和日期控件的样式设计。
- artDialog.js 和 default.css 分别是一个对话框控件的插件和对话框控件的样式设计。
- list.js 是页面控件的一些事件响应和数据请求的实现方法设计。

代码清单 7-25　观众管理主页中的页头设计

```
<head>
   <title>观众管理</title>
   <link th:href="@{/scripts/pagination/pagination.css}" rel="stylesheet"
type="text/css" />
   <link th:href="@{/scripts/artDialog/default.css}" rel="stylesheet"
type="text/css" />
   <link th:href="@{/scripts/My97DatePicker/skin/WdatePicker.css}"
rel="stylesheet" type="text/css" />
   <script th:src="@{/scripts/pagination/jquery.pagination.js}"></script>
   <script th:src="@{/scripts/jquery.smartselect-1.1.min.js}"></script>
   <script th:src="@{/scripts/artDialog/artDialog.js}" />
   <script th:src="@{/scripts/My97DatePicker/WdatePicker.js}"></script>
   <script th:src="@{/scripts/person/list.js}"></script>
</head>
```

观众管理主页中的查询表单设计如代码清单 7-26 所示。这里主要实现了观众列表数据查询的功能。

表单中使用了两个控件，分别为“名称”和“日期”，即可以通过输入观众的名称和创建观众的日期这两个参数来查询观众列表。其中，日期控件使用了 WdatePicker 的插件来实现。

“查询”超链接设定了控件标识为 searchBtn，它的执行事件的实现将由脚本文件 list.js 来完成。

代码清单 7-26　观众管理主页中的查询表单设计

```
<form id="queryForm" method="get">
   <div class="radiusGrayBox782">
      <div class="radiusGrayTop782"></div>
      <div class="radiusGrayMid782">
         <div class="dataSearchBox forUserRadius">
            <ul>
               <li>
                  <label class="preInpTxt f-left">名称</label>
                  <input type="text" class="inp-list f-left w-200" value= ""
id="name" name="name"/>
               </li>
               <li>
                  <label class="preInpTxt f-left">日期</label>
                  <input  onfocus="WdatePicker({dateFmt:'yyyy-MM-dd
HH:mm:ss'})" type="text" class="inp-list w-200 clear-mr f-left" id="create"
name="create"/>
               </li>
               <li>
                  <a href="javascript:void(0)" class="blueBtn-62X30
f- right" id="searchBtn">查询</a>
               </li>
            </ul>
         </div>
      </div>
   </div>
</form>
```

观众管理主页中的新增观众的超链接设计如代码清单 7-27 所示。这里主要创建了一个“新增”超链接，并将控件标识设定为 addInfo，控件的事件响应将由脚本文件 list.js 来完成。单击这个超链接将打开一个创建观众的操作界面。

代码清单 7-27　观众管理主页中的新增观众的超链接设计

```
<div class="newBtnBox">
  <a id="addInfo" class="blueBtn-62X30" href="javascript:void(0)">新增</a>
</div>
```

观众管理主页中的列表设计如代码清单 7-28 所示。这里主要使用一个表格控件实现了观众列表数据显示的功能。

表格控件的表头部分设定了列表数据显示各列的字段名称，分别为“ID”、“名称”、“日期”和“操作”。其中，“操作”这一列将放置一些操作链接，可以执行对当条记录的查看、修改和删除等操作。

表格控件的表体部分只设定了一个标识符 tbodyContent，这里的数据将由脚本文件 list.js 中的设计方法来填充。

表格控件下面使用分页插件 pagination 设置了一个列表工具条控件，可以用来显示列表的页数和控制显示的分页列表数据。

代码清单 7-28　观众管理主页中的表格控件设计

```
<div class="dataDetailList mt-12">
      <table class="dataListTab">
         <thead>
         <tr>
            <th>ID</th>
            <th>名称</th>
            <th>日期</th>
            <th>操作</th>
         </tr>
         </thead>
         <tbody id="tbodyContent">
         </tbody>
      </table>
      <div class="tableFootLine">
         <div class="pagebarList pagination"/>
      </div>
</div>
```

观众管理主页设计中的脚本文件 list.js 的设计包括页面初始化、分页插件工具条设计、分页数据请求和回调函数设计、分页列表数据填充设计、对话框设计等内容。

脚本文件 list.js 中的页面初始化设计如代码清单 7-29 所示。这是一个默认运行的初始化函数。

在初始化函数中为主页中标识为 searchBtn 的“查询”超链接设置了一个单击的响应事件，即在页面上单击“查询”链接将执行 pageaction()函数，请求分页列表数据。

在初始化函数中为主页中标识为 addInfo 的“新增”超链接设置了一个单击的响应事件，即在页面上单击“新增”链接将执行 create()函数，打开一个创建观众节点的对话框。

在初始化函数中将默认运行 pageaction()函数，即在打开观众管理主页时，将默认显示观众的分页列表数据。

在初始化函数中设置了一个 live 事件，为分页工具条插件 pagination 设置了 change 响应事件，用来执行跳转页码按钮的功能。

代码清单 7-29　脚本中的页面初始化设计

```
$(function () {
    $('#searchBtn').click(function(){
        pageaction();
    });
    $('#addInfo').click(function(){
        create();
    });
    //初始化分页
    pageaction();
    var pg = $('.pagination');
    $('#pageSelect').live("change",function(){
        pg.trigger('setPage', [$(this).val()-1]);
    });
});
```

脚本文件 list.js 中的分页插件工具条设计如代码清单 7-30 所示。

工具条设定了每页显示的记录条数为 10 条，并设定了“首页”、“上页”、“下页”、“尾页”等按钮，也设定了在这些按钮中间可以使用 3 个按页码显示的按钮，同时设定可以显示页总数，并指定了回调函数的名称为 pageselectCallback。

代码清单 7-30　脚本中的分页插件工具条设计

```
var getOpt = function(){
    var opt = {
        items_per_page: 10,          //每页记录数
        num_display_entries: 3,      //中间显示的页数个数，默认为 10
        current_page:0,              //当前页
        num_edge_entries:1,          //头尾显示的页数个数，默认为 0
        link_to:"javascript:void(0)",
        prev_text:"上页",
        next_text:"下页",
        load_first_page:true,
        show_total_info:true ,
        show_first_last:true,
        first_text:"首页",
        last_text:"尾页",
        hasSelect:false,
        callback: pageselectCallback //回调函数
    }
    return opt;
}
```

脚本文件 list.js 中的分页数据请求和回调函数设计如代码清单 7-31 所示。

数据请求通过访问当前位置中的 URL“./list”向控制器请求列表数据，然后使用返回的数据更新分页插件工具条上的显示信息。

回调函数 pageselectCallback 通过调用 fillData()函数，向页面表格控件填充数据，同时响应分页插件中工具条按钮的事件，通过传递页码等参数重新请求分页数据。

代码清单 7-31　脚本中的分页数据请求和回调函数设计

```
var currentPageData = null ;
var pageaction = function(){
    $.get('./list?t='+new Date().getTime(),{
        name:$("#name").val(),create:$("#create").val()
    },function(data){
        currentPageData = data.content;
        $(".pagination").pagination(data.totalElements, getOpt());
```

```
    });
}
var pageselectCallback = function(page_index, jq, size){
    if(currentPageData!=null){
        fillData(currentPageData);
        currentPageData = null;
    }else
        $.get('./list?t='+new Date().getTime(),{
size:size,page:page_index,name:$("#name").val(),create:$("#create").val()
        },function(data){
            fillData(data.content);
        });
}
```

脚本文件 list.js 中的分页列表数据填充设计如代码清单 7-32 所示。

在填充数据时，首先清空表格控件中标识为 tbodyContent 的表体原来显示的内容，然后使用一个 each 循环展开数据对象进行数据填充，对于观众对象来说，在最后的“操作”一列数据中，设置了“查看”、“修改”、“删除”和“评分”等超链接，以方便对每个观众节点执行 CURD 等操作。其中，对日期数据调用了脚本函数 getSmpFormatDateByLong()，实现了日期的中文环境格式化输出。

代码清单 7-32　脚本中的分页列表数据填充设计

```
function fillData(data){
    var $list = $('#tbodyContent').empty();
    $.each(data,function(k,v) {
        var html = "";
        html += '<tr> ' +
            '<td>' + (v.id == null ? '' : v.id) + '</td>' +
            '<td>' + (v.name == null ? '' : v.name) + '</td>' +
            '<td>' + (v.create == null ? '' : getSmpFormatDateByLong(v.create,
true)) + '</td>';
        html += '<td><a class="c-50a73f mlr-6" href="javascript:void(0)"
onclick="showDetail(\'' + v.id + '\')">查看</a>';

        html += '<a class="c-50a73f mlr-6" href="javascript:void(0)" onclick=
"edit(\'' + v.id + '\')">修改</a>';
```

```
        html += '<a class="c-50a73f mlr-6" href="javascript:void(0)" onclick=
"del(\''+ v.id+'\')">删除</a>';

        html += '<a class="c-50a73f mlr-6" href="javascript:void(0)" onclick=
"rating(\''+ v.id+'\')">评分</a>';

        html +='</td></tr>' ;

        $list.append($(html));
    });
}
```

脚本文件 list.js 中的对话框设计如代码清单 7-33 所示。

这是在列表页面中通过单击“查看”链接打开查看观众详情的一个对话框设计。程序通过在当前位置中访问 URL“./{id}”，即使用观众节点的 ID 作为参数，向观众控制器请求数据。取得数据后，在当前界面上打开一个对话框来显示数据。对话框可以设置标题、窗口的大小和显示位置等。其中，初始化函数 init 将对话框存入变量 artdialog 中，关闭对话框函数 close 可以通过变量 artdialog 来关闭对话框。

代码清单 7-33　脚本中的对话框设计

```
var artdialog ;
function showDetail(id){
    $.get("./"+id,{ts:new Date().getTime()},function(data){
        art.dialog({
            lock:true,
            opacity:0.3,
            title: "查看信息",
            width:'750px',
            height: 'auto',
            left: '50%',
            top: '50%',
            content:data,
            esc: true,
            init: function(){
                artdialog = this;
```

```
            },
            close: function(){
                artdialog = null;
            }
        });
    });
}
```

完成观众管理的主页界面设计之后，显示效果如图 7-7 所示。图的上部是一个查询表单设计，中间是一个“新增”超链接设计，下部是列表控件和分页插件工具条设计。

图 7-7 观众管理主页界面显示效果

2. 详情查看界面设计

在观众管理主页中，通过单击数据列表中的“查看”链接，就可以打开一个查看观众详细信息的对话框。其中，对话框只是一个窗口设计而已，对话框上的显示信息还是通过界面设计来实现的。

观众管理的详情查看界面设计主要由一个页面设计文件 show.html 完成，它的内容如代码清单 7-34 所示。

这个页面设计包括两个方面的功能：一方面，使用一个表格控件来显示观众的详细信息；另一方面，使用一个超链接“返回”来关闭对话框。

表格控件设置了观众对象各个属性的显示方式。其中，“名称”和“性别”属性使用只读的输入控件来显示；“朋友”、“观看”和“评分”等关系都使用了只读的多行下拉列表框来显示，其中的列表数据使用 Thymeleaf 的标签语言的 each 循环语句来输出；“日期”属性使用 WdatePicker 插件来显示，并且设置了只读限制。

超链接“返回”的事件响应在脚本文件 list.js 中实现，通过调用对话框的 close()方法执行关闭对话框的操作。

代码清单 7-34　详情查看的页面设计

```
<div class="addInfBtn">
   <h3 class="itemTit"><span>观众信息</span></h3>
   <table class="addNewInfList">
      <tr>
         <th>名称</th>
         <td width="240"><input class="inp-list w-200 clear-mr f-left"
type="text" th:value="${person.name}" readonly="true"/></td>
         <th>性别</th>
         <td>
            <input class="inp-list w-200 clear-mr f-left" type="text"
th:if="${person.sex==0}" th:value="男" readonly="true"/>
            <input class="inp-list w-200 clear-mr f-left" type="text"
th:if="${person.sex==1}" th:value="女" readonly="true"/>
         </td>
      </tr>
      <tr>
         <th>朋友</th>
         <td>
            <select name="friends" id="friends" multiple="multiple"
readonly="true">
               <option th:each="friend:${person.friends}"
```

```
                        th:text="${#strings.length(friend.name)>20?#strings.
substring(friend.name,0,20)+'...':friend.name}"
                        th:selected="true"
                        ></option>
            </select>
        </td>
        <th>观看</th>
        <td>
            <select name="visiters" id="visiters" multiple="multiple"
readonly="true">
                <option th:each="visiter:${person.visiters}"
                        th:text="${#strings.length(visiter.name)>20?#strings.
substring(visiter.name,0,20)+'...':visiter.name}"
                        th:selected="true"
                        ></option>
            </select>
        </td>
    </tr>

    <tr>
        <th>日期</th>
        <td>
            <input  onfocus="WdatePicker({dateFmt:'yyyy-MM-dd HH:mm:ss'})"
type="text" class="inp-list w-200 clear-mr f-left" th:value="${person.create} ?
${#dates.format(person.create,'yyyy-MM-dd HH:mm:ss')} :''" readonly="true"/>
        </td>
        <th>评分</th>
        <td>
            <select name="ratings" id="ratings" multiple="multiple"
readonly="true">
                <option th:each="rating:${person.ratings}"
                        th:text="${rating.movie.name}+'-->'+${rating. stars}+
'*'+${rating.comment}"
                        th:selected="true"
                        ></option>
            </select>
        </td>
```

```
        </tr>
    </table>
    <div class="bottomBtnBox">
        <a class="btn-93X38 backBtn" href="javascript:closeDialog(0)">返回</a>
    </div>
</div>
```

完成观众管理的详情查看界面设计后，显示效果如图 7-8 所示。图中，内容显示的主要界面是表格控件设计的效果，底端的“返回”是关闭对话框的链接设计的显示效果。

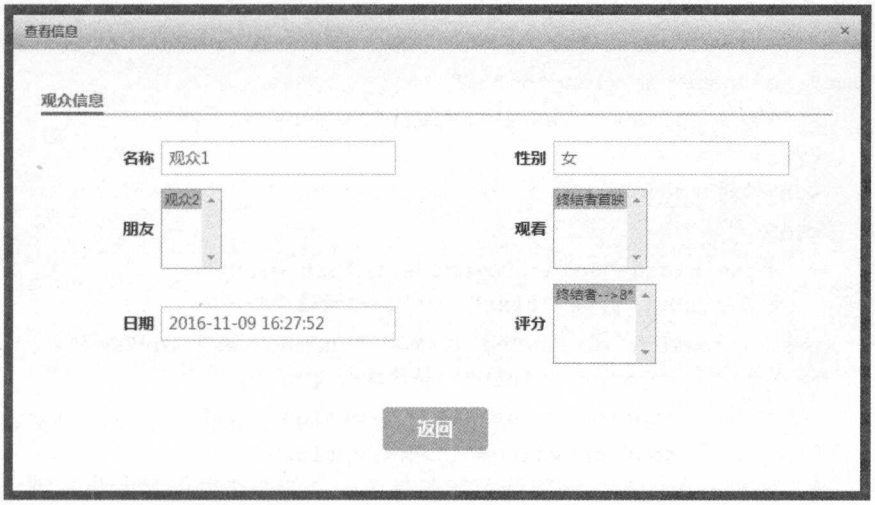


图 7-8　详情查看界面显示效果

3. 新增数据界面设计

观众管理中新增数据的界面设计主要包括一个页面设计文件 new.html 和一个 JavaScript 脚本文件 new.js。

页面设计文件 new.html 的内容如代码清单 7-35 所示。

这个页面设计主要使用一个表单来实现，表单中设置了观众的“名称”和“性别”属性的录入方式，分别使用普通的输入框和下拉列表框来实现；同时设置了“朋友”和“观看”关系的录入方式，都使用多行下拉列表框来实现；而观众的另一个属性“日期”将由系统使用当前时间来生成。

页面底端设置了两个超链接，分别是“确定”和“返回”，用来处理表单的提交和取消录入操作。

代码清单 7-35　新增数据的页面设计

```
<script th:src="@{/scripts/person/new.js}"></script>
<form id="saveForm" action="./save" method="post">
   <table class="addNewInfList">
      <tr>
         <th>名称</th>
         <td width="240">
            <input class="inp-list w-200 clear-mr f-left" type="text"
name="name" id="name" maxlength="32" />
            <span class="tipStar f-left">*</span>
         </td>
         <th>性别</th>
         <td>
            <div class="selectDownMode f-left w-206">
               <input type="text"  value="请选择"/>
               <select class="selectMode" name="sex" id="sex">
                  <option value="">请选择</option>
                  <option value="0">男</option>
                  <option value="1">女</option>
               </select>
            </div>
            <span class="tipStar f-left">*</span>
         </td>
      </tr
      <tr>
         <th>朋友</th>
         <td width="240">
            <div >
               <select name="friends" id="friends" multiple="multiple">
                  <option th:each="friend:${friends}"
th:value="${friend.id}"
                          th:text="${#strings.length(friend.name)>20?
#strings.substring(friend.name, 0,20)+'...':friend.name}"
                          ></option>
```

```
                </select>
            </div>
        </td>
        <th>观看</th>
        <td width="240">
            <select name="visiters" id="visiters" multiple="multiple">
                <option th:each="visiter:${visiters}"
th:value="${visiter.id}"
                        th:text="${#strings.length(visiter.name)>20?
#strings.substring(visiter. name,0,20)+'...':visiter.name}"
                        ></option>
            </select>
        </td>
    </tr>
</table>
<div class="bottomBtnBox">
    <a class="btn-93X38 saveBtn" href="javascript:void(0)">确定</a>
    <a class="btn-93X38 backBtn" href="javascript: closeDialog()">返回</a>
</div>
</form>
```

脚本文件 new.js 的内容如代码清单 7-36 所示。

这个脚本设计包括两个功能：一个是处理页面提交表单的数据验证；另一个是执行数据保存请求。

数据验证设定的规则为：观众的名称属性 name 和性别属性 sex 为必填字段。

当在新增页面上单击“确定”链接进行表单提交时，就将触发 saveBtn 的 click 事件。这个事件函数的设计将首先检查表单的数据验证是否通过，如果通过，则访问当前位置的 URL “./save”，向控制器发出保存数据的请求；否则提示数据验证失败。在数据保存请求中使用了表单的系列化函数 serialize()，将表单的控件数据系列化为观众的对象参数进行提交，如果请求成功，则将给出“保存成功”的提示；否则提示出错信息。

代码清单 7-36　新增数据的验证及保存调用设计

```
$(function(){
  $('#saveForm').validate({
```

```
    rules: {
          name :{required:true},
          sex :{required:true}
    },messages:{
          name :{required:"必填"},
          sex :{required:"必填"}
      }
  });
$('.saveBtn').click(function(){
   if($('#saveForm').valid()){
       $.ajax({
          type: "POST",
          url: "./save",
          data: $("#saveForm").serialize(),
          headers: {"Content-type":
"application/x-www-form-urlencoded; charset=UTF-8"},
          success: function (data) {
             if (data == 1) {
                alert("保存成功");
                pageaction();
                closeDialog();
             } else {
                alert(data);
             }
          }
       });
     }else{
       alert('数据验证失败，请检查！');
     }
  });
});
```

完成观众管理的新增数据界面设计后，显示效果如图 7-9 所示。

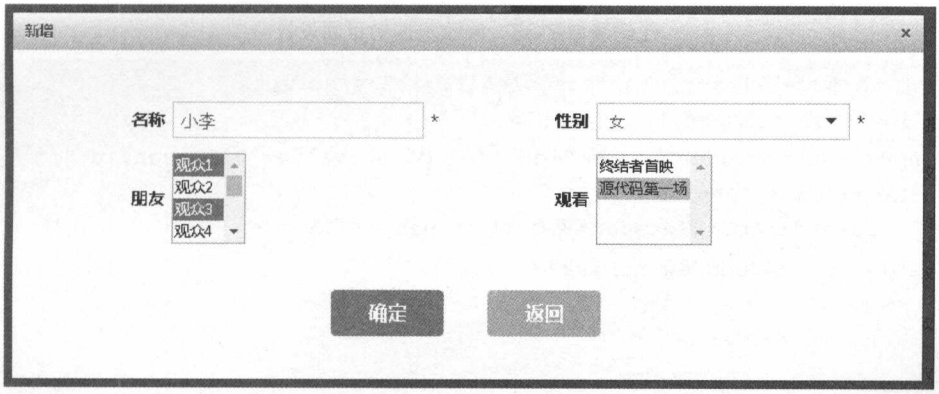


图 7-9　新增数据界面显示效果

4. 修改数据界面设计

观众管理中修改数据的界面设计主要包括一个页面设计文件 edit.html 和一个 JavaScript 脚本文件 edit.js。

页面设计文件 edit.html 的内容如代码清单 7-37 所示。

修改数据的页面设计主要使用一个表单控件来实现。这个表单设计跟新增页面的表单设计有点相似，不同的是，这个表单的各个控件中都已经具有了“旧数据”，所以对于下拉列表框的设计要在选项列表中进行判断，如果一个选项已经包含“旧数据”，则进行选中处理。

例如，“朋友”关系列表框的设计中包含这样的代码行“th:selected="${#lists.contains (fids, friend.id)}"”，即使用 Thymeleaf 的标签语言 lists 的包含函数 contains()对朋友的 ID 进行判断，如果选项列表的朋友 ID 集合 fids 包含 friend.id 则返回 true，这就实现了将下拉列表框的选中属性 selected 设置为选中的形式显示。

表单中有一个隐藏控件 id，存储了观众的节点 ID，以保证提交表单时只针对这个 id 的节点进行数据修改。

页面底端设置了两个超链接，分别是“确定”和“返回”，用来处理表单的提交和取消修改操作。

代码清单 7-37　修改数据的页面设计

```
<script th:src="@{/scripts/person/edit.js}"></script>
<form id="saveForm" method="post">
   <input type="hidden" name="id" id="id" th:value="${person.id}"/>
<div class="addInfBtn" >
   <h3 class="itemTit"><span>观众信息</span></h3>
   <table class="addNewInfList">
      <tr>
         <th>名称</th>
         <td width="240">
            <input class="inp-list w-200 clear-mr f-left" type="text"
th:value="${person.name}" id="name" name="name" maxlength="16" />
            <span class="tipStar f-left">*</span>
         </td>
         <th>性别</th>
         <td>
            <div class="selectDownMode f-left w-206">
               <input type="text" th:if="${person.sex==0}" th:value="男"/>
               <input type="text" th:if="${person.sex==1}" th:value="女"/>
               <input type="text" th:if="${person.sex==null}" th:value="
请选择"/>
               <select class="selectMode" name="sex" id="sex">
                  <option value="">请选择</option>
                  <option value="0" th:selected="${person.sex==0}">男
</option>
                  <option value="1" th:selected="${person.sex==1}">女
</option>
               </select>
            </div>
            <span class="tipStar f-left">*</span>
         </td>
      </tr>
      <tr>
         <th>朋友</th>
         <td width="240">
            <div >
               <select name="friends" id="friends" multiple="multiple">
```

```
                    <option th:each="friend:${friends}" th:value="${friend.id}"
                            th:text="${#strings.length(friend.name)>20?
#strings.substring(friend.name,0,20)+'...':friend.name}"
                            th:selected="${#lists.contains(fids, friend.id)}"
                            ></option>
                </select>
            </div>
        </td>
        <th>观看</th>
        <td width="240">
            <div >
                <select name="visiters" id="visiters" multiple="multiple">
                    <option th:each="visiter:${visiters}"
th:value= "$ {visiter.id}"
                            th:text="${#strings.length(visiter.name)>20?
#strings.substring(visiter.name,0,20)+'...':visiter.name}"
                            th:selected="${#lists.contains(vids, visiter. id)}"
                            ></option>
                </select>
            </div>
        </td>
    </tr>
    <tr>
        <th>日期</th>
        <td>
            <input  onfocus="WdatePicker({dateFmt:'yyyy-MM-dd HH:mm:ss'})"
type="text" class="inp-list w-200 clear-mr f-left" th:value="${person. create} ?
${#dates.format(person.create,'yyyy-MM-dd HH:mm:ss')} :''" id= "create"
name="create"/>
        </td>
    </tr>
</table>
<div class="bottomBtnBox">
    <a class="btn-93X38 saveBtn" href="javascript:void(0)">确定</a>
    <a class="btn-93X38 backBtn" href="javascript:closeDialog(0)">返回</a>
</div>
</div>
</form>
```

脚本文件 edit.js 的内容如代码清单 7-38 所示。

这个脚本设计包括两个功能：一个是处理页面提交表单的数据验证；另一个是执行数据保存请求。

数据验证设定的规则为：观众的名称属性 name 和性别属性 sex 为必填字段。

当在修改页面上单击“确定”链接进行表单提交时，将触发 saveBtn 的 click 事件。这个事件函数的设计将首先检查表单的数据验证是否通过，如果通过，则访问当前位置的 URL “./update”，向控制器发出保存数据的请求；否则提示数据验证失败。在数据保存请求中使用了表单的系列化函数 serialize()，将表单的控件数据系列化为观众的对象参数进行提交，如果请求成功，则将给出“编辑成功”的提示；否则提示出错信息。

代码清单 7-38　修改数据的表单验证和保存方法

```
$(function(){
  $('#saveForm').validate({
      rules: {
         name :{required:true},
         sex :{required:true}
      },messages:{
         name :{required:"必填"},
         sex :{required:"必填"}
      }
  });
  $('.saveBtn').click(function(){
    if($('#saveForm').valid()){
        $.ajax({
           type: "POST",
           url: "./update",
           data: $("#saveForm").serialize(),
           headers: {"Content-type":
"application/x-www-form-urlencoded; charset=UTF-8"},
           success: function (data) {
              if (data == 1) {
                 alert("编辑成功");
                 pageaction();
```

```
                    closeDialog();
                } else {
                    alert(data);
                }
            }
        });
    }else{
        alert('数据验证失败，请检查！');
    }
});

});
```

完成观众管理的修改数据界面设计后，显示效果如图 7-10 所示。

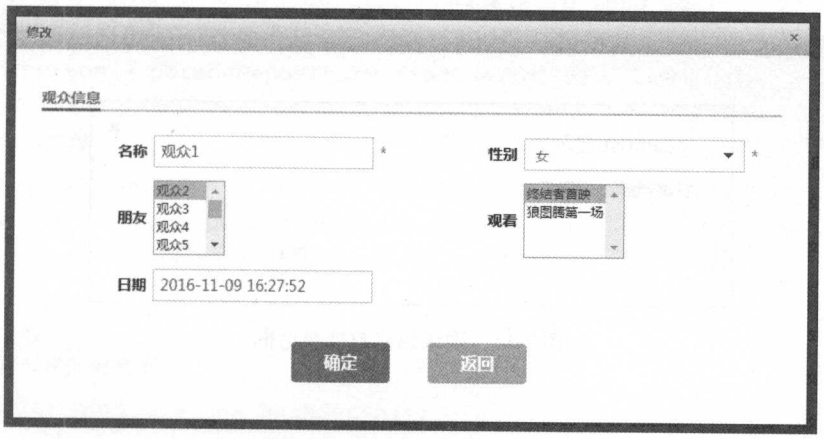


图 7-10　修改数据界面显示效果

5. 删除数据确认设计

在观众管理主页中，单击一条记录中的“删除”链接，将执行删除记录的请求。实现这个请求的方法如代码清单 7-39 所示。

在执行删除数据操作时，将显示一个对话框，由用户进行确认。当用户确定删除时，则访问当前位置的 URL“./delete/{id}”，向控制器提交删除数据的请求。如果请求成功，则将提示“删除成功”；否则返回错误提示。

代码清单 7-39　观众管理中的删除数据确认设计

```
function del(id){
    if(!confirm("您确定删除此记录吗？")){
        return false;
    }
    $.get("./delete/"+id,{ts:new Date().getTime()},function(data){
        if(data==1){
            alert("删除成功");
            pageaction();
        }else{
            alert(data);
        }
    });
}
```

完成观众管理的删除数据确认设计后，执行删除操作时显示的对话框如图 7-11 所示。

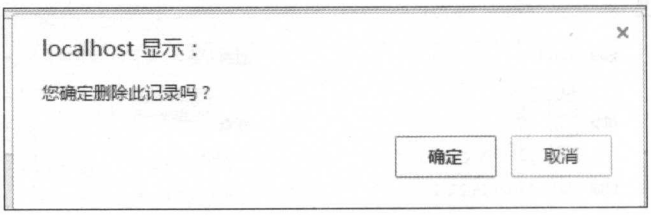


图 7-11　确认删除数据对话框

7.9　电影评分的 Web 设计

电影评分的 Web 设计主要实现一个观众对观看过的电影进行评分的功能，它的实现方法分为访问控制设计和界面设计两部分。

7.9.1　电影评分访问控制设计

电影评分的访问控制设计如代码清单 7-40 所示。

程序中的 rating()方法定义了一个当前位置的 URL 为“/rating/{id}”，即使用观众节

点的 ID 作为参数进行访问，访问这个 URL 将返回字符串“person/rating”，这个字符串指定了使用当前目录 person 下的 rating.html 文件，即电影评分的页面设计文件。其中，调用观众存储库接口的 findOne()方法中设定了路径深度为“2”，这样，在返回的观众对象中可以包含看过但未评分的电影对象。然后在观众看过的所有电影中找出未评分的电影组成一个集合对象传递给页面。

程序中的 rating_save()方法定义了一个当前位置的 URL 为“/rating/save”，访问这个 URL 时将设定观众的评分关系，然后调用观众数据管理服务类的 save()方法请求保存数据，如果请求成功则返回 1。

代码清单 7-40　电影评分访问控制设计

```
@RequestMapping(value="/rating/{id}")
public String rating(ModelMap model,@PathVariable Long id){
    //深度为 2 可返回看过的节目的电影对象：person.getVisiters().getMovie()
    Person person = personRepository.findOne(id, 2);

    //已评分的电影
    Set<Long> rating_sids = new HashSet<>();
    for(Rating rating : person.getRatings()){
        rating_sids.add(rating.getMovie().getId());
    }

    //看过的未评分的电影
    Set<Movie> movies = new HashSet<>();
    for(Show show : person.getVisiters()){
        if(!rating_sids.contains(show.getMovie().getId()))
            movies.add(show.getMovie());
    }

    model.addAttribute("movies",movies);
    model.addAttribute("person", person);

    return "person/rating";
}

@RequestMapping(method = RequestMethod.POST, value="/rating/save")
```

```
@ResponseBody
public String rating_save(Rating rating) throws Exception{
    Person person = rating.getPerson();
    person.rate(rating.getMovie(),rating.getStars(),rating.getComment());
    personService.save(person);
    logger.info("评分->ID="+ person.getId());
    return "1";
}
```

7.9.2 电影评分界面设计

电影评分的界面设计主要包括一个页面设计文件 rating.html 和一个 JavaScript 脚本文件 rating.js。

页面设计文件 rating.html 的内容如代码清单 7-41 所示。

这个页面设计主要使用一个表单来实现，表单中设置了“看过的电影”、“星级”、“评论”和“日期”4 个控件。其中，“看过的电影”即观众观看过但未进行评分的电影。

页面底端设置了两个超链接，分别是“确定”和“返回”，用来处理表单的提交和取消录入操作。

代码清单 7-41 电影评分的页面设计

```
<script th:src="@{/scripts/person/rating.js}"></script>
<form id="saveForm" method="post">
    <input type="hidden" name="person" id="person" th:value="${person.id}"/>
<div class="addInfBtn" >
    <h3 class="itemTit"><span>电影评分</span></h3>
    <table class="addNewInfList">
        <tr>
            <th>看过的电影</th>
            <td width="240">
                <div class="selectDownMode f-left w-206">
                    <input type="text"  value="请选择"/>
                    <select class="selectMode" name="movie" id="movie">
                        <option value="">请选择</option>
                        <option th:each="movie:${movies}" th:value="${movie.id}"
```

```
                        th:text="${#strings.length(movie.name)>20?
#strings.substring(movie.name,0,20)+'...':movie.name}"
                        ></option>
                </select>
            </div>
            <span class="tipStar f-left">*</span>
        </td>
        <th>星级</th>
        <td width="240">
            <input class="inp-list w-200 clear-mr f-left" type="text"
id="stars" name="stars" maxlength="16" />
            <span class="tipStar f-left">*</span>
        </td>
    </tr>
    <tr>
        <th>评论</th>
        <td width="240">
            <input class="inp-list w-200 clear-mr f-left" type="text"
id="comment" name="comment" maxlength="16" />
        </td>
        <th>日期</th>
        <td>
            <input onfocus="WdatePicker({dateFmt:'yyyy-MM-dd HH:mm: ss'})"
type="text" class="inp-list w-200 clear-mr f-left"  id="create" name=
"create"/>
        </td>
    </tr>
</table>
<div class="bottomBtnBox">
    <a class="btn-93X38 saveBtn" href="javascript:void(0)">确定</a>
    <a class="btn-93X38 backBtn" href="javascript:closeDialog(0)">返回</a>
</div>
</div>
</form>
```

脚本文件 rating.js 的内容如代码清单 7-42 所示。

这个脚本设计包括两个功能：一个是处理页面提交表单的数据验证；另一个是执行数据保存请求。

数据验证设定的规则为：评分中的电影 movie 和星级 stars 为必填字段。

当在界面上单击“确定”链接进行表单提交时，将触发 saveBtn 的 click 事件。这个事件函数的设计将首先检查表单的数据验证是否通过，如果通过，则访问当前位置的 URL“./rating/save”，向控制器发出保存数据的请求；否则提示数据验证失败。在数据保存请求中使用了表单的系列化函数 serialize()，将表单的控件数据系列化为评分关系的对象参数进行提交，如果请求成功，则将返回“评分成功”的提示；否则返回错误信息提示。

代码清单 7-42　电影评分的数据验证及访问方法

```
$(function(){
   $('#saveForm').validate({
      rules: {
           movie :{required:true},
           stars :{required:true}
        },messages:{
           movie :{required:"必填"},
           stars :{required:"必填"}
        }
   });
   $('.saveBtn').click(function(){
      if($('#saveForm').valid()){
          $.ajax({
             type: "POST",
             url: "./rating/save",
             data: $("#saveForm").serialize(),
             headers: {"Content-type":
"application/x-www-form-urlencoded; charset=UTF-8"},
             success: function (data) {
                if (data == 1) {
                   alert("评分成功");
                   pageaction();
                   closeDialog();
                } else {
                   alert(data);
                }
```

```
            }
        });
    }else{
        alert('数据验证失败，请检查！');
    }
});

});
```

完成电影评分的界面设计后，显示效果如图 7-12 所示。

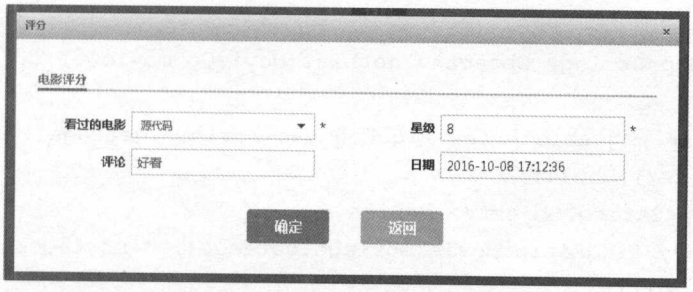


图 7-12　电影评分界面显示效果

7.10　电影排名的 Web 设计

电影排名的 Web 设计主要用来处理显示电影评分排名的列表数据，它的实现方法分为访问控制设计和界面设计两部分。

7.10.1　电影排名访问控制设计

电影排名的访问控制设计如代码清单 7-43 所示。

程序中的 index()方法定义了一个当前位置的 URL 为“/movie”，访问这个 URL 将返回字符串“worth/movie”，这个字符串指定了使用当前目录 worth 下的 movie.html 文件，即电影排名的页面设计文件。

程序中的 getList()方法定义了一个当前位置的 URL 为“/movie_list”，访问这个 URL

将调用电影存储库接口的 findRatingMovie()和 findRatingMovieCount()方法，请求电影排名的列表数据，然后使用 PageImpl()方法组成一个页对象返回给调用者。

代码清单 7-43　电影排名访问控制设计

```
@RequestMapping("/movie")
public String index() throws Exception{
    return "worth/movie";
}

@RequestMapping(value = "/movie_list")
@ResponseBody
public Page<Map<String, Object>> getList(MovieQo movieQo) {
    try {
        Pageable pageable = new PageRequest(movieQo.getPage(),
movieQo.getSize(), null);
        Set<Map<String,Object>> movies =
movieRepository.findRatingMovie(movieQo.getPage() * movieQo.getSize(),
movieQo.getSize());
        int count = movieRepository.findRatingMovieCount();
        return new PageImpl(new ArrayList(movies), pageable, (long)count);
    }catch (Exception e){
        e.printStackTrace();
    }
    return null;
}
```

7.10.2　电影排名界面设计

电影排名的界面设计主要包括一个页面设计文件 movie.html 和一个 JavaScript 脚本文件 movie_list.js。

页面设计文件 movie.html 的内容如代码清单 7-44 所示。

这个页面主要使用一个表格控件来显示电影排名的列表。

表格控件的表头部分设定了列表数据显示字段的名称，分别为“ID”、“名称”、“星级”和“评分人数”。

表格控件的表体部分设定了一个标识符 tbodyContent，表体的数据将由脚本文件 movie_list.js 中的设计方法来填充。

表格控件下面使用分页插件 pagination 设置了一个列表工具条控件，它可以用来显示列表的页数和控制显示的分页列表数据。

代码清单 7-44　电影排名的页面设计

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
layout:decorator="fragments/layout">
<head>
   <title>电影排名</title>
   ......
   <script th:src="@{/scripts/worth/movie_list.js}"></script>
</head>
<body>
<div class="locationLine" layout:fragment="prompt">
   当前位置：首页 &gt; <em >电影排名</em>
</div>
<div class="statisticBox w-782"  layout:fragment="content">
   <div class="dataDetailList mt-12">
      <table class="dataListTab">
         <thead>
         <tr>
            <th>ID</th>
            <th>名称</th>
            <th>星级</th>
            <th>评分人数</th>
         </tr>
         </thead>
         <tbody id="tbodyContent">
         </tbody>
      </table>
      <div class="tableFootLine">
         <div class="pagebarList pagination"/>
      </div>
   </div>
```

```
</div>
</body>
</html>
```

电影排名页面引用的脚本文件 movie_list.js 的设计主要包括页面初始化、分页插件工具条设计、分页数据请求和回调函数设计、分页列表数据填充设计等功能。这里只针对分页数据请求和回调函数设计、分页列表数据填充设计这两部分功能进行说明，其他功能的实现可参照观众管理的主页设计。

电影排名的分页数据请求和回调函数设计如代码清单 7-45 所示。

数据请求通过访问当前位置中的 URL “./movie_list” 向控制器请求列表数据，然后使用返回的数据更新分页插件工具条上的显示信息。

回调函数 pageselectCallback 通过调用 fillData()函数向页面表格控件填充数据，同时响应分页插件中工具条按钮的事件，通过传递页码等参数重新请求分页数据。

代码清单 7-45　电影排名分页数据请求和回调函数设计

```
var currentPageData = null ;
var pageaction = function(){
    $.get('./movie_list?t='+new Date().getTime(),function(data){
        currentPageData = data.content;
        $(".pagination").pagination(data.totalElements, getOpt());
    });
}

var pageselectCallback = function(page_index, jq, size){
    var html = "" ;
    if(currentPageData!=null){
        fillData(currentPageData);
        currentPageData = null;
    }else
        $.get('./movie_list?t='+new Date().getTime(),{
            size:size,page:page_index
        },function(data){
            fillData(data.content);
        });
}
```

电影排名的分页列表数据填充设计如代码清单 7-46 所示。

在填充数据时，首先清空表格控件中标识为 tbodyContent 的表体原来显示的内容，然后使用一个 each 循环展开数据对象进行数据填充，排名的各个字段分别为 id、name、starts 和 num，分别表示“电影节点标识”、“名称”、“星级”和“评分人数”。其中，星级使用函数 toFixed(2)使用四舍五入方式保留到小数点最后两位。

代码清单 7-46　电影排名分页列表数据填充设计

```
function fillData(data){
    var $list = $('#tbodyContent').empty();
    $.each(data,function(k,v) {
        var html = "";
        html += '<tr> ' +
            '<td>' + (v.id == null ? '' : v.id) + '</td>' +
            '<td>' + (v.name == null ? '' : v.name) + '</td>' +
            '<td>' + (v.stars == null ? '' : v.stars.toFixed(2)) + '</td>' +
            '<td>' + (v.num == null ? '' : v.num) + '</td>';
        html +='</tr>' ;
        $list.append($(html));
    });
}
```

完成电影排名的界面设计后，显示效果如图 7-13 所示。

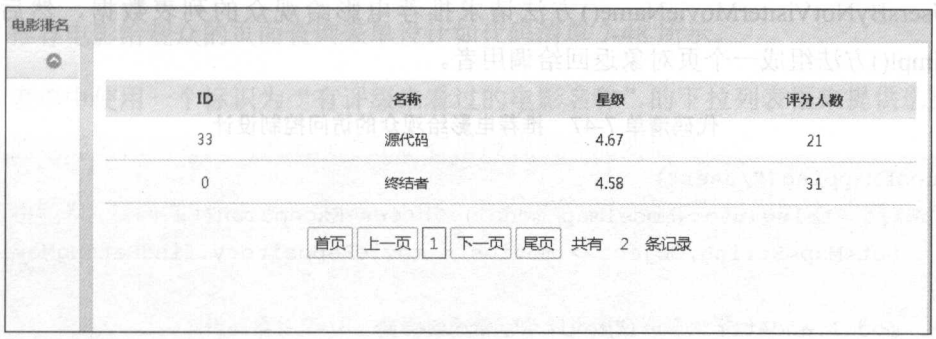


图 7-13　电影排名界面显示效果

7.11 电影推荐的 Web 设计

电影推荐的 Web 设计包括两个方面，分别为推荐电影给观众和推荐电影给朋友。

7.11.1 推荐电影给观众的 Web 设计

推荐电影给观众的Web设计主要实现了选择一部电影推荐给未看过这部电影的所有观众的界面操作功能，包含访问控制设计和界面设计两个方面的内容。

1. 推荐电影给观众的访问控制设计

推荐电影给观众的访问控制设计如代码清单 7-47 所示。

程序中的 user()方法定义了一个当前位置的 URL 为“/user”，访问这个 URL 将返回字符串“worth/user”，这个字符串指定了使用当前目录 worth 下的 user.html 文件，即推荐电影给观众的页面设计。其中，调用电影存储库接口的 findRatingMovie()方法返回排名靠前的 100 部电影，作为页面使用的查询参数。

程序中的 getUserList()方法定义了一个当前位置的 URL 为“/user_list”，访问这个 URL 将调用观众存储库接口的 findUsersByNotVisiterMovieNamePage()和 findUsersByNotVisiterMovieName()方法请求推荐电影给观众的列表数据，然后使用 PageImpl()方法组成一个页对象返回给调用者。

代码清单 7-47 推荐电影给观众的访问控制设计

```
@RequestMapping("/user")
    public String user(ModelMap model) throws Exception{
        Set<Map<String,Object>> movies = movieRepository.findRatingMovie(0,
100);
        model.addAttribute("movies", movies);
        return "worth/user";
    }

    @RequestMapping(value = "/user_list")
```

```
    @ResponseBody
    public Page<Person> getUserList(MovieQo movieQo) {
        try {
            Pageable pageable = new PageRequest(movieQo.getPage(),
movieQo.getSize(), null);
            Set<Person> list = personRepository.findUsersByNotVisiter
MovieNamePage(movieQo.getName(),
                    movieQo.getPage() * movieQo.getSize(), movieQo.getSize());
            int count = personRepository.findUsersByNotVisiterMovieName
(movieQo.getName()).size();
            return new PageImpl(new ArrayList(list), pageable, (long)count);
        }catch (Exception e){
            e.printStackTrace();
        }
        return null;
    }
```

2. 推荐电影给观众的界面设计

推荐电影给观众的界面设计主要包括一个页面设计文件 user.html 和一个 JavaScript 脚本文件 user_list.js。

页面设计文件 user.html 的内容包含页面查询表单设计、列表设计等方面的功能实现。

推荐电影给观众的页面查询表单设计如代码清单 7-48 所示。

表单中使用一个标识为“有评级未看过的电影名称”的下拉列表框来提供选择需要推荐的电影名称列表，列表选项中还附有星级说明等简要信息。

超链接“查询”的标识设定为 searchBtn，它的事件响应的实现将由脚本文件 user_list.js 来完成。

代码清单 7-48　推荐电影给观众的页面查询表单设计

```
<form id="queryForm" method="get">
    <div class="radiusGrayBox782">
        <div class="radiusGrayTop782"></div>
        <div class="radiusGrayMid782">
```

```
        <div class="dataSearchBox forUserRadius">
            <ul>
                <li>
                    <label class="preInpTxt f-left">有评级未看过的电影名称
</label>
                    <div class="selectDownMode f-left w-206">
                        <input type="text"  value="请选择"/>
                        <select class="selectMode" name="name" id="name">
                            <option value="">请选择</option>
                            <option th:each="movie:${movies}"
th:value= "${movie.name}"
                                    th:text="${#strings.length(movie.
name)>20?#strings.substring(movie.name,0,20)+'...':movie.name}+'(星级:
'+${#numbers.formatDecimal(movie.stars, 1, 2)}+')'"
                                    ></option>
                        </select>
                    </div>
                </li>
                <li>
                    <a href="javascript:void(0)" class="blueBtn-62X30
f-right" id="searchBtn">查询</a>
                </li>
            </ul>
        </div>
    </div>
</div>
</form>
```

推荐电影给观众的列表设计如代码清单 7-49 所示。

这里主要使用一个表格控件实现了观众列表数据显示的功能。

表格控件的表头部分设定了列表数据显示字段的名称，分别为“ID”、“名称”、“性别”和“日期”。

表格控件的表体部分设定了一个标识符 tbodyContent，这里的数据将由脚本文件 user_list.js 中的设计方法来填充。

表格控件下面使用分页插件 pagination 设置了一个列表工具条控件，用来显示列表的页数和控制显示的分页列表数据。

代码清单 7-49　推荐电影给观众的表格控件设计

```
<div class="dataDetailList mt-12">
      <table class="dataListTab">
         <thead>
         <tr>
            <th>ID</th>
            <th>名称</th>
            <th>性别</th>
            <th>日期</th>
         </tr>
         </thead>
         <tbody id="tbodyContent">
         </tbody>
      </table>
         <div class="tableFootLine">
            <div class="pagebarList pagination"/>
         </div>
   </div>
```

推荐电影给观众的脚本文件 user_list.js 的设计主要包括页面初始化、分页插件工具条设计、分页数据请求和回调函数设计、分页列表数据填充设计等功能。这里只针对分页数据请求和回调函数设计、分页列表数据填充设计这两部分功能进行说明。

推荐电影给观众的分页数据请求和回调函数设计如代码清单 7-50 所示。

数据请求通过访问当前位置中的 URL “./user_list” 向控制器请求列表数据，然后使用返回的数据更新分页插件工具条上的显示信息。

回调函数 pageselectCallback 通过调用 fillData()函数向页面表格控件填充数据，同时响应分页插件中工具条按钮的事件，通过传递页码等参数重新请求分页数据。

代码清单 7-50　推荐电影给观众的分页数据请求和回调函数设计

```
var currentPageData = null ;
var pageaction = function(){
```

```
    $.get('./user_list?t='+new Date().getTime(),{
        name:$("#name").val()
    },function(data){
        currentPageData = data.content;
        $(".pagination").pagination(data.totalElements, getOpt());
    });
}

var pageselectCallback = function(page_index, jq, size){
    var html = "" ;
    if(currentPageData!=null){
        fillData(currentPageData);
        currentPageData = null;
    }else
        $.get('./user_list?t='+new Date().getTime(),{
            size:size,page:page_index,name:$("#name").val()
        },function(data){
            fillData(data.content);
        });
}
```

推荐电影给观众的分页列表数据填充设计如代码清单 7-51 所示。

在填充数据时，首先清空表格控件中标识为 tbodyContent 的表体原来显示的内容，然后使用一个 each 循环展开数据对象进行数据填充，即使用观众对象的属性 id、name、sex 和 create，分别表示“观众节点标识”、“名称”、“性别”和“创建日期”。

代码清单 7-51　推荐电影给观众的分页列表数据填充设计

```
function fillData(data){
    var $list = $('#tbodyContent').empty();
    $.each(data,function(k,v) {
        var html = "";
        html += '<tr> ' +
            '<td>' + (v.id == null ? '' : v.id) + '</td>' +
            '<td>' + (v.name == null ? '' : v.name) + '</td>' +
            '<td>' + (v.sex == null ? '' : v.sex==0?'男':'女') + '</td>' +
            '<td>' + (v.create == null ? '' : getSmpFormatDateByLong(v.create,
```

```
true)) + '</td>';
        html +='</tr>' ;
        $list.append($(html));
    });
}
```

完成推荐电影给观众的界面设计后，显示效果如图 7-14 所示。

图 7-14　推荐电影给观众界面显示效果

7.11.2　推荐电影给朋友的 Web 设计

推荐电影给朋友的 Web 设计同样包含访问控制和界面设计两个方面的内容。这些设计跟推荐电影给观众的设计类似，这里不再赘述。

完成推荐电影给朋友的界面设计后，显示效果如图 7-15 所示。

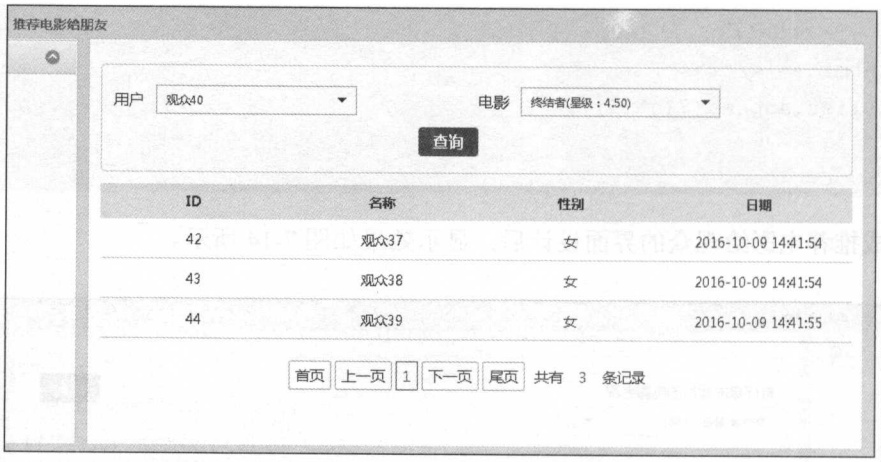


图 7-15 推荐电影给朋友界面显示效果

7.12 管理后台的导航栏设计

在上面各种数据管理的 Web 设计中，各个项目的数据管理在访问控制设计中都配置了主页的 URL，对于这些 URL 的访问，通过一个导航栏进行了统一管理和设置。

这个导航栏的设计如代码清单 7-52 所示。

导航栏主要包含以下几个方面的主页链接：

- 影院管理。
- 电影管理。
- 节目管理。
- 观众管理。
- 电影排名。
- 推荐给观众。
- 推荐给朋友。

代码清单 7-52 管理后台的导航栏设计

```
<!DOCTYPE html>
<html>
```

```
<body>
<div th:fragment="nav">
   <div class="columnLeftMenu">
      <h6 ><span class="f-right"></span>系统管理</h6>
      <ul >
         <li><a th:classappend="${page == 'cinema/index' ? 'currentPageNav':''}"
th:href="@{/cinema/index}">影院管理</a></li>
         <li><a th:classappend="${page == 'movie/index' ? 'currentPageNav':''}"
th:href="@{/movie/index}">电影管理</a></li>
         <li><a th:classappend="${page == 'show/index' ? 'currentPageNav':''}"
th:href="@{/show/index}">节目管理</a></li>
         <li><a th:classappend="${page == 'person/index' ? 'currentPageNav':''}"
th:href="@{/person/index}">观众管理</a></li>
         <li><a th:classappend="${page == 'worth/movie' ? 'currentPageNav':''}"
th:href="@{/worth/movie}">电影排名</a></li>
         <li><a th:classappend="${page == 'worth/user' ? 'currentPageNav':''}"
th:href="@{/worth/user}">推荐给观众</a></li>
         <li><a th:classappend="${page == 'worth/friend' ? 'currentPageNav':''}"
th:href="@{/worth/friend}">推荐给朋友</a></li>
      </ul>
   </div>
</div>
</body>
</html>
```

完成导航栏设计后，显示效果如图 7-16 所示。

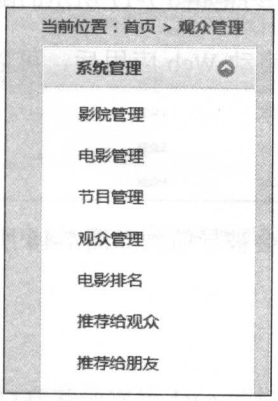


图 7-16　导航栏设计显示效果

7.13 实例工程使用

本章实例工程已发布在 GitHub 中，可以通过下列链接直接下载：

https://github.com/mr-csj/neo4j-social/archive/master.zip

也可以使用开发工具通过 Git 客户端使用如下链接检出：

https://github.com/mr-csj/neo4j-social.git

7.13.1 运行配置

检出工程后，配置好数据库连接和设置一个 Spring Boot 运行配置，就可以运行项目工程中的 web 模块，体验上面设计的所有细节和过程了。

工程的数据库连接使用如下的默认配置。如果这个配置跟你的数据库配置参数不同，则可以修改相关的参数。

```
compiler=org.neo4j.ogm.compiler.MultiStatementCypherCompiler
#Http
driver=org.neo4j.ogm.drivers.http.driver.HttpDriver
URI=http://localhost:7474
username = neo4j
password = 12345678
```

Spring Boot 的运行配置可以参照如图 7-17 所示的配置。

使用 Spring Boot 运行配置启动 Web 应用后，可以使用如下链接通过浏览器进行访问：

http://localhost

使用上述地址访问应用，将会被导航到如图 7-18 所示的管理后台首页。

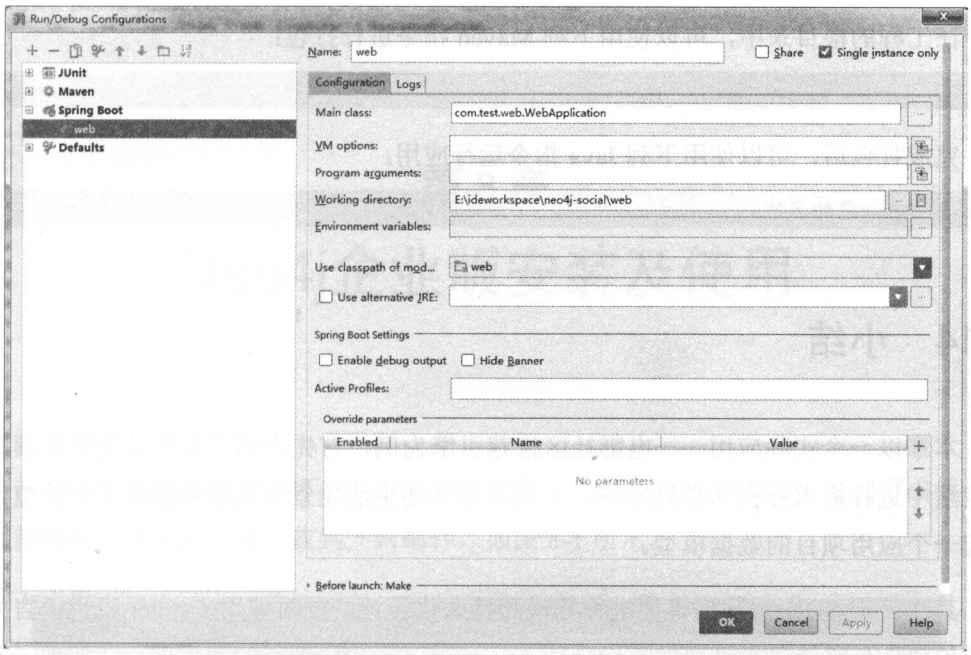


图 7-17　Web 应用运行配置

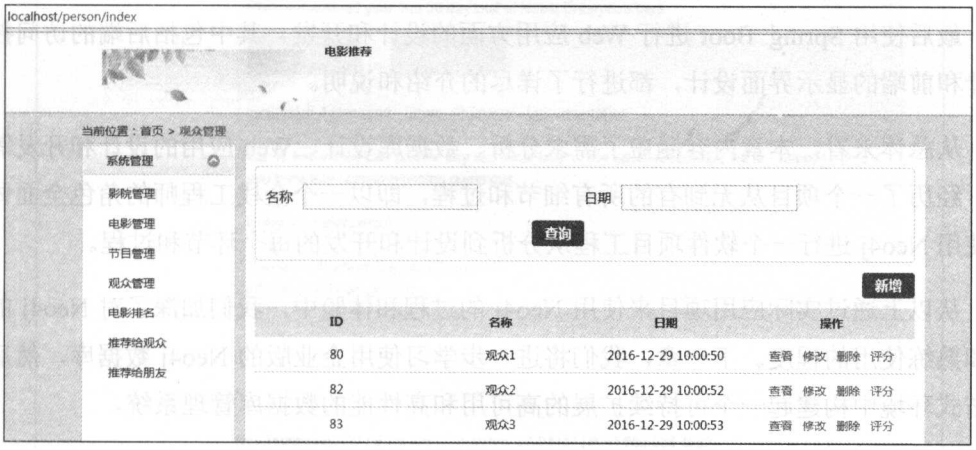


图 7-18　电影社区推荐引擎管理后台首页

7.13.2　应用发布

本章实例通过打包后，也可以在其他支持 Java 的机器上进行部署。

在工程的根目录中，可以使用下列 Maven 指令进行打包：

```
mvn clean package -D skipTests
```

完成打包后，可以使用下列 Java 指令运行应用：

```
java -jar <包的名字>.jar
```

7.14 小结

本章以一个实际应用——电影社区推荐引擎为例，首先介绍了怎样从司空见惯的现实生活中进行需求分析和数据发掘，从而发现普通生活场景中关联数据的实用价值，建立起一个应用项目的数据模型。

其次使用 SDN 对数据模型进行建模和持久化设计，同时使用 Cypher 查询语言为数据的价值体现设计了查询算法，即使用 Neo4j 建立了应用项目的数据库，并实现了业务需求的数据管理和查询等方法。

最后使用 Spring Boot 进行 Web 应用方面的设计和开发，其中包括后端的访问控制设计和前端的显示界面设计，都进行了详尽的介绍和说明。

从总体来看，本章内容涵盖了需求分析、数据库设计、Web 应用的设计和开发等阶段，经历了一个项目从无到有的所有细节和过程，即以一个全栈工程师的角色全面体验了使用 Neo4j 进行一个软件项目工程从分析到设计和开发的每个环节和过程。

从以上通过实际应用项目来使用 Neo4j 的过程和体验中，我们加深了对 Neo4j 的理解和熟练使用的程度。下一章，我们将进一步学习使用企业版的 Neo4j 数据库，然后在分布式环境中构建起一个可持续扩展的高可用和高性能的数据库管理系统。

第 8 章
Neo4j企业版安装及使用

在 Neo4j 官网中，可以下载企业版进行试用。在填写试用请求时，将被要求提供一个邮箱地址，提交后，将会收到一封邮件，如图 8-1 所示。

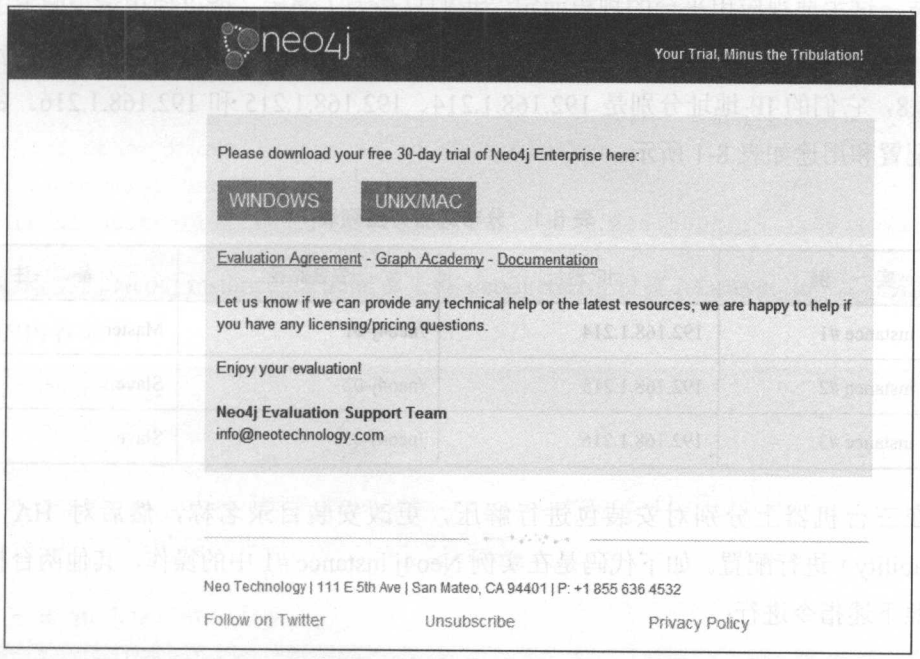


图 8-1　企业版下载邮件

在这封电子邮件中提供了试用企业版的下载链接。试用版本可以免费试用 30 天。成功下载安装包后，可以按照下面介绍的方法进行安装和使用。

8.1 分布式服务器安装

分布式服务器的安装方法也可以参考官方的安装手册进行：http://neo4j.com/docs/operations-manual/current/tutorial/#ha-setup-tutorial。

下面，我们将使用 Linux 安装包，分别介绍两种方法来安装分布式服务器，即在不同机器上安装和在同一台机器上安装。

8.1.1 在不同机器上安装分布式服务器

按照安装手册的介绍，分布式服务器的最佳配置可以使用三台、五台或更多的机器来安装，这主要视应用平台的规模而定。我们只是为了演示，所以使用最少配置的三台 Linux 机器来进行测试。假如三台测试机器都使用 CentOS 7.0 操作系统，并且都安装了 JDK 1.8，它们的 IP 地址分别是 192.168.1.214、192.168.1.215 和 192.168.1.216。各台机器的配置和用途如表 8-1 所示。

表 8-1 分布式服务器列表

实　例	IP 地址	安装路径	备　注
Neo4j instance #1	192.168.1.214	/neo4j-01	Master
Neo4j instance #2	192.168.1.215	/neo4j-02	Slave
Neo4j instance #3	192.168.1.216	/neo4j-03	Slave

在三台机器上分别对安装包进行解压，更改安装目录名称，然后对 HA（High Availability）进行配置。如下代码是在实例 Neo4j instance #1 中的操作，其他两台机器可以参照下述指令进行：

```
tar -xf neo4j-enterprise-3.0.6-unix.tar.gz
mv neo4j-enterprise-3.0.6 neo4j-01
cd neo4j-01
vi ./conf/neo4j.conf
```

1. 修改配置

修改实例 Neo4j instance #1 的配置文件 neo4j.conf，设置 ha.server_id = 1，完成修改部分的内容如下：

```
# HTTP Connector
dbms.connector.http.type=HTTP
dbms.connector.http.enabled=true
# To accept non-local HTTP connections, uncomment this line
dbms.connector.http.address=0.0.0.0:7474

# HA - High Availability
# SINGLE - Single mode, default.
dbms.mode=HA

# Unique server id for this Neo4j instance
# can not be negative id and must be unique
ha.server_id = 1

# List of other known instances in this cluster
# Alternatively, use IP addresses:
ha.initial_hosts = 192.168.1.214:5001,192.168.1.215:5001,192.168.1.216:5001
```

修改实例 Neo4j instance #2 的配置文件 neo4j.conf，设置 ha.server_id = 2，完成修改部分的内容如下：

```
# HTTP Connector
dbms.connector.http.type=HTTP
dbms.connector.http.enabled=true
# To accept non-local HTTP connections, uncomment this line
dbms.connector.http.address=0.0.0.0:7474

# HA - High Availability
# SINGLE - Single mode, default.
dbms.mode=HA

# Unique server id for this Neo4j instance
# can not be negative id and must be unique
```

```
ha.server_id = 2

# List of other known instances in this cluster
# Alternatively, use IP addresses:
ha.initial_hosts = 192.168.1.214:5001,192.168.1.215:5001,192.168.1.216:5001
```

修改实例 Neo4j instance #3 的配置文件 neo4j.conf，设置 ha.server_id = 3，完成修改部分的内容如下：

```
# HTTP Connector
dbms.connector.http.type=HTTP
dbms.connector.http.enabled=true
# To accept non-local HTTP connections, uncomment this line
dbms.connector.http.address=0.0.0.0:7474

# HA - High Availability
# SINGLE - Single mode, default.
dbms.mode=HA

# Unique server id for this Neo4j instance
# can not be negative id and must be unique
ha.server_id = 3

# List of other known instances in this cluster
# Alternatively, use IP addresses:
ha.initial_hosts = 192.168.1.214:5001,192.168.1.215:5001,192.168.1.216:5001
```

如图 8-2 所示是修改实例 Neo4j instance #3 的配置文件 neo4j.conf 的实际操作过程。

2. 更改打开文件限制数

别忘了在安装单机版时碰到过的问题，Linux 的默认打开文件数是 1024，我们可以将其增加 64 倍，修改成 65 536。

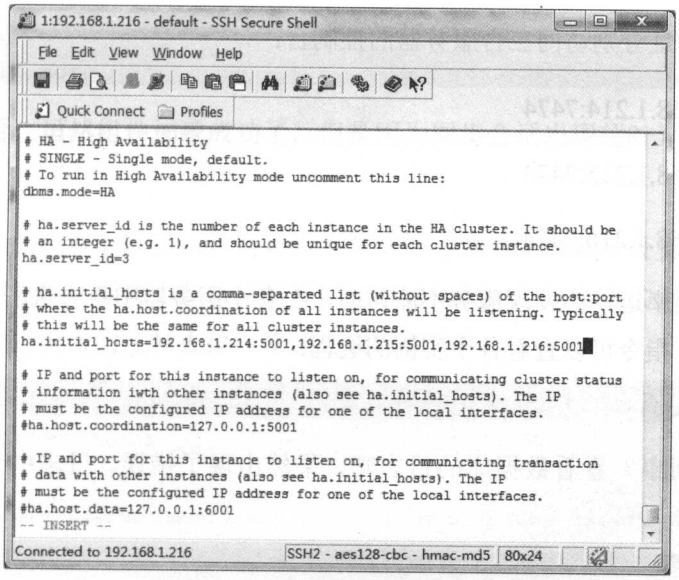


图 8-2　修改 neo4j.conf 的操作过程

临时改动，立即生效，可使用如下指令：

```
ulimit -n 65536
```

让修改永久生效，需要将下列两行代码加入系统配置文件/etc/security/limits.conf：

```
* - nofile 65536
* - nproc 65536
```

使用下列指令可以查看修改情况：

```
ulimit -a
```

如果能看到如下信息就可以了：

```
open files  (-n) 65536
```

3. 启动服务

像单机时启动服务器一样启动，注意启动的顺序，第一个启动的将默认作为 Master。

```
neo4j-01$ ./bin/neo4j start
neo4j-02$ ./bin/neo4j start
neo4j-03$ ./bin/neo4j start
```

使用如下网址分别访问三台服务器的控制台：

http://192.168.1.214:7474

http://192.168.1.215:7474

http://192.168.1.216:7474

如果不是很幸运，会发现服务不能访问，在各个服务器中查看进程，却发现服务不存在。使用如下指令可以查看各个实例的进程：

```
ps -ef|grep neo4j
```

这是怎么回事？看看数据库的日志吧，果然出现了错误。日志中记载的失败原因如下：

```
......
Caused by: org.neo4j.kernel.lifecycle.LifecycleException: Component
'org.neo4j.cluster.client.ClusterJoin@593b117a' was successfully
initialized, but failed to start. Please see attached cause exception.
        at org.neo4j.kernel.lifecycle.LifeSupport$LifecycleInstance.start
(LifeSupport.java:444)
        at org.neo4j.kernel.lifecycle.LifeSupport.start(LifeSupport.java: 107)
        at org.neo4j.kernel.lifecycle.LifeSupport$LifecycleInstance.start
(LifeSupport.java:434)
        at org.neo4j.kernel.lifecycle.LifeSupport.start(LifeSupport.java: 107)
        at org.neo4j.kernel.impl.factory.GraphDatabaseFacadeFactory.
newFacade(GraphDatabaseFacadeFactory.java:140)
        ... 10 more
Caused by: java.util.concurrent.TimeoutException: Conversation-response
mapping:
{2/13#=ResponseFuture{conversationId='2/13#', initiatedByMessageType=join,
response=null}}
        at org.neo4j.cluster.statemachine.StateMachineProxyFactory
$ResponseFuture.get(StateMachineProxyFactory.java:314)
        at org.neo4j.cluster.client.ClusterJoin.joinByConfig(ClusterJoin.
java:143)
        at org.neo4j.cluster.client.ClusterJoin.start(ClusterJoin.java: 82)
        at org.neo4j.kernel.lifecycle.LifeSupport$LifecycleInstance.start
```

```
(LifeSupport.java:434)
        ... 14 more
```

从日志上看，虽然组件加载成功了，但是因为同步会话出现超时异常，最终导致服务启动失败。

查看服务器的配置 neo4j.conf，可以看到其中有心跳消息的超时设置和集群成员之间的同步心跳超时设置。配置中默认注释掉了超时时间设定。在所有实例中，将这两项同步超时的时间设定前面的注释去掉，修改完成后如下：

```
# How often heartbeat messages should be sent. Defaults to ha.default_timeout.
ha.heartbeat_interval=5s

# Timeout for heartbeats between cluster members. Should be at least twice that of ha.heartbeat_interval.
ha.heartbeat_timeout=11s
```

重新启动集群中的所有成员，现在就全部正常运行了。

4. 登录控制台

通过浏览器打开 Neo4j 的 Web 控制台，把三台服务器的用户和密码都修改成一样，然后登录 Web 控制台，如图 8-3 所示。

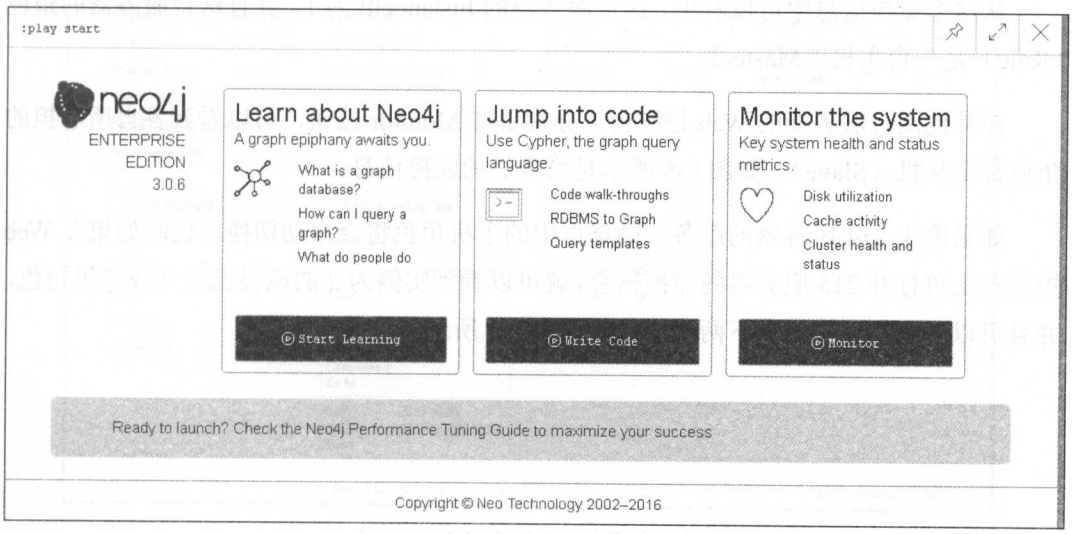


图 8-3　Neo4j 的 Web 控制台

在 214 服务器的 Web 控制台上单击如图 8-3 所示的 Monitor 按钮，相当于执行了指令“:play sysinfo”，这样就可以显示当前服务器的系统信息，如图 8-4 所示。

图 8-4　主机的服务器系统信息

从这个监控信息中可以看出，这台服务器的 InstanceId 为 1，并且这台服务器的角色（Role）是一台主机（Master）。

在其他两台服务器的 Web 控制台上分别单击 Monitor 按钮，可以看到各自所承担的角色都是从机（Slave）。如图 8-5 所示是实例 2 的监控信息。

如果停止 214 服务器的服务，则集群中的主机角色将会自动切换。这时如果在 Web 控制台上再打开 215 服务器的监控信息，就可以看到实例为 2 的成员已经变成主机角色，并且可以看到集群中只剩下两个成员，如图 8-6 所示。

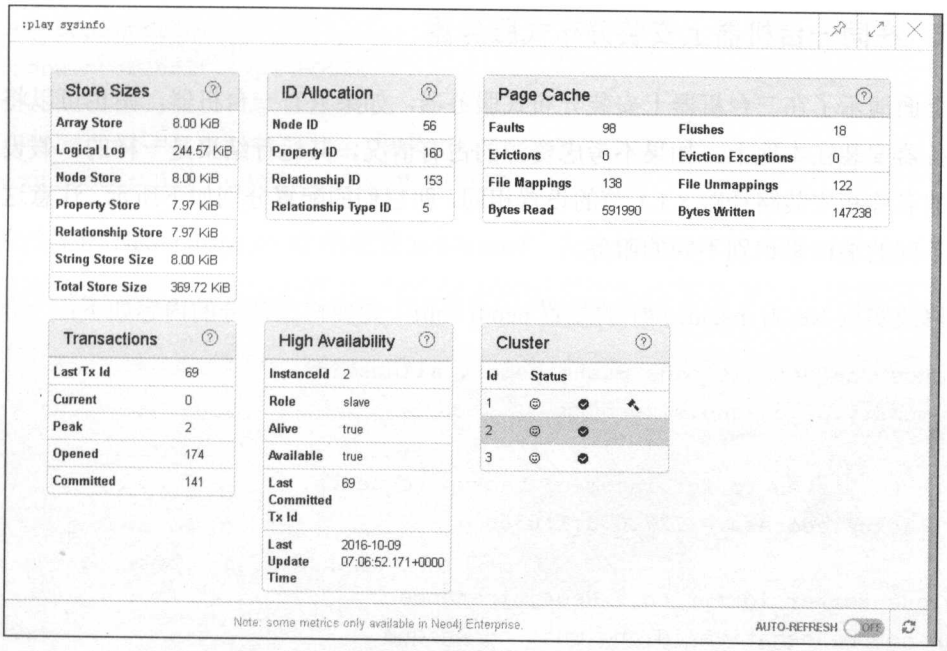


图 8-5　从机的服务器系统信息

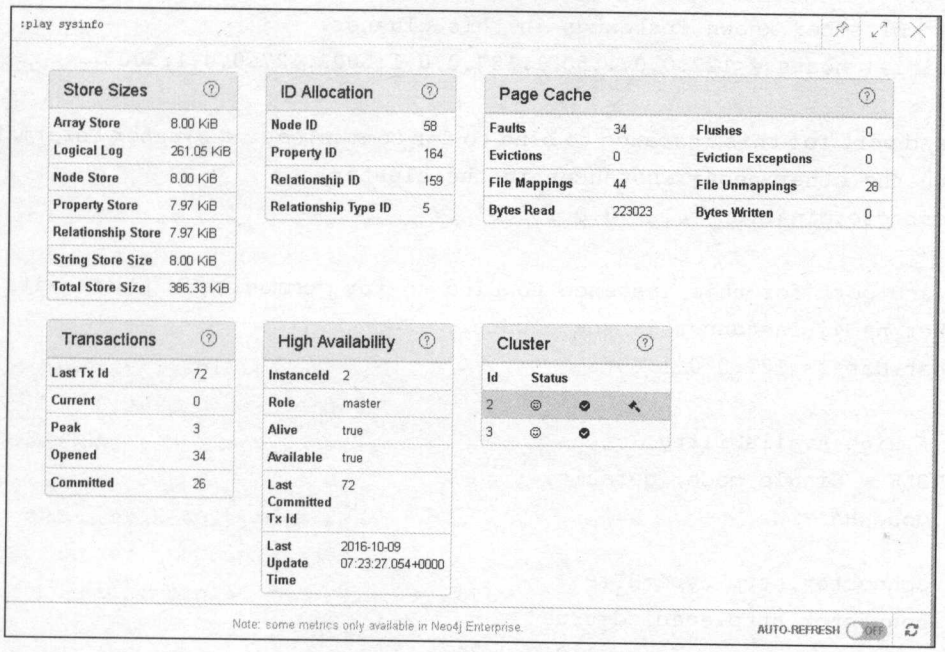


图 8-6　主从机切换后的监控信息

8.1.2 在同一台机器上安装分布式服务器

上面演示了在三台机器上安装分布式服务器，如果只有一台机器，则也可以将分布式服务器安装在本地上。如果不考虑资源的占有情况，其运行结果是一样的。假设三个实例的名称和安装路径跟 8.1.1 节的设置相同，即它们的配置分别如下所示。注意这里使用了不同的端口来识别不同的服务。

修改实例 Neo4j instance #1 的配置 neo4j.conf，完成修改部分的内容如下：

```
# Reduce the default page cache memory allocation
dbms.memory.pagecache.size=500m

# Port to listen to for incoming backup requests.
dbms.backup.address = 127.0.0.1:6366

# Unique server id for this Neo4j instance
# can not be negative id and must be unique
ha.server_id = 1

# List of other known instances in this cluster
ha.initial_hosts = 127.0.0.1:5001,127.0.0.1:5002,127.0.0.1:5003

# IP and port for this instance to bind to for communicating cluster information
# with the other neo4j instances in the cluster.
ha.host.coordination = 127.0.0.1:5001

# IP and port for this instance to bind to for communicating data with the
# other neo4j instances in the cluster.
ha.host.data = 127.0.0.1:6363

# HA - High Availability
# SINGLE - Single mode, default.
dbms.mode=HA

dbms.connector.http.type=HTTP
dbms.connector.http.enabled=true
dbms.connector.http.address=0.0.0.0:7474
```

```
# Bolt connector
dbms.connector.bolt.type=BOLT
dbms.connector.bolt.enabled=true
dbms.connector.bolt.tls_level=OPTIONAL
dbms.connector.bolt.address=0.0.0.0:7687
```

修改实例 Neo4j instance #2 的配置 neo4j.conf，完成修改部分的内容如下：

```
# Reduce the default page cache memory allocation
dbms.memory.pagecache.size=500m

# Port to listen to for incoming backup requests.
dbms.backup.address = 127.0.0.1:6367

# Unique server id for this Neo4j instance
# can not be negative id and must be unique
ha.server_id = 2

# List of other known instances in this cluster
ha.initial_hosts = 127.0.0.1:5001,127.0.0.1:5002,127.0.0.1:5003

# IP and port for this instance to bind to for communicating cluster information
# with the other neo4j instances in the cluster.
ha.host.coordination = 127.0.0.1:5002

# IP and port for this instance to bind to for communicating data with the
# other neo4j instances in the cluster.
ha.host.data = 127.0.0.1:6364

# HA - High Availability
# SINGLE - Single mode, default.
dbms.mode=HA

dbms.connector.http.type=HTTP
dbms.connector.http.enabled=true
dbms.connector.http.address=0.0.0.0:7475

# Bolt connector
```

```
dbms.connector.bolt.type=BOLT
dbms.connector.bolt.enabled=true
dbms.connector.bolt.tls_level=OPTIONAL
dbms.connector.bolt.address=0.0.0.0:7688
```

修改实例 Neo4j instance #3 的配置 neo4j.conf，完成修改部分的内容如下：

```
# Reduce the default page cache memory allocation
dbms.memory.pagecache.size=500m

# Port to listen to for incoming backup requests.
dbms.backup.address = 127.0.0.1:6368

# Unique server id for this Neo4j instance
# can not be negative id and must be unique
ha.server_id = 3

# List of other known instances in this cluster
ha.initial_hosts = 127.0.0.1:5001,127.0.0.1:5002,127.0.0.1:5003

# IP and port for this instance to bind to for communicating cluster information
# with the other neo4j instances in the cluster.
ha.host.coordination = 127.0.0.1:5003

# IP and port for this instance to bind to for communicating data with the
# other neo4j instances in the cluster.
ha.host.data = 127.0.0.1:6365

# HA - High Availability
# SINGLE - Single mode, default.
dbms.mode=HA

dbms.connector.http.type=HTTP
dbms.connector.http.enabled=true
dbms.connector.http.address=0.0.0.0:7476

# Bolt connector
dbms.connector.bolt.type=BOLT
```

```
dbms.connector.bolt.enabled=true
dbms.connector.bolt.tls_level=OPTIONAL
dbms.connector.bolt.address=0.0.0.0:7689
```

其启动和运行方法与安装在不同机器上的操作基本相同，这里不再赘述。

8.2　使用 Haproxy 实施负载均衡服务

在 Neo4j 分布式服务器安装好之后，我们当然不是一个一个地去使用集群中的成员。

在 Neo4j 的集群成员中，不管是主机还是从机，都可以进行读/写操作。所以，我们可以使用一种代理服务器工具对 Neo4j 的集群实施负载均衡配置。这样，对于客户端来说，就可以通过类似于广播地址的方式，使用一个统一的 IP 地址来访问集群中的所有成员。

Haproxy 是一台配置简便并且性能不错的代理服务器，而且它可以跟 Neo4j 的分布式服务器进行无缝整合，是 Neo4j 实施负载均衡配置的首选工具。

8.2.1　普通负载均衡配置

通过使用 Haproxy 就可以实现对上面安装的分布式数据库的负载均衡访问。现在我们使用另外一台机器来做这件事情，假如它的 IP 地址为 192.168.1.211，并且已经安装了 Haproxy 代理服务器。

Haproxy 只需一个配置文件就可以实现负载均衡服务。

普通的 Neo4j 分布式服务器的负载均衡配置如代码清单 8-1 所示。

在配置中，使用本机的 80 端口对外提供默认的 HTTP 服务。访问 80 端口，将被导向到后端服务 neo4j 中。

后端服务 neo4j 的配置使用了默认的负载均衡算法，即用普通的轮询算法均衡地访问 214、215、216 三台机器的 7474 端口提供的服务，即 Neo4j 服务器中提供的 HTTP 方式的服务。

在配置中，还使用本机的 8080 端口开启了 Haproxy 的监控管理后台，使用监控管理后台的用户名和密码分别设置为“admin”和“123456”。

代码清单 8-1　使用 Neo4j 分布式服务器的 Haproxy 负载均衡配置

```
global
    daemon
    maxconn 256
defaults
    mode http
    timeout connect 5000ms
    timeout client 50000ms
    timeout server 50000ms
frontend http-in
    bind *:80
    default_backend neo4j
backend neo4j
    option httpchk GET /db/manage/server/ha/master
    server s1 192.168.1.214:7474 maxconn 32
    server s2 192.168.1.215:7474 maxconn 32
    server s3 192.168.1.216:7474 maxconn 32
listen admin
    bind *:8080
    stats enable
    stats uri /
    stats auth admin:123456
```

使用以上配置启动 Haproxy 代理服务器，就可以使用由其提供的负载均衡服务了。

假如上面的配置文件保存为 haproxy.cfg，即可在文件所在目录使用下列指令启动 Haproxy 代理服务器：

```
/usr/local/haproxy/sbin/haproxy -f haproxy.cfg
```

启动 Haproxy 代理服务器之后，就可以在浏览器中使用链接 http://192.168.1.211 打开 Neo4j 的 Web 控制台。在 Web 控制台上，通过执行“:play sysinfo”指令，就可以查看连接的是哪台服务器实例。这时刷新浏览器可更新连接的服务器实例，如图 8-7 所示。

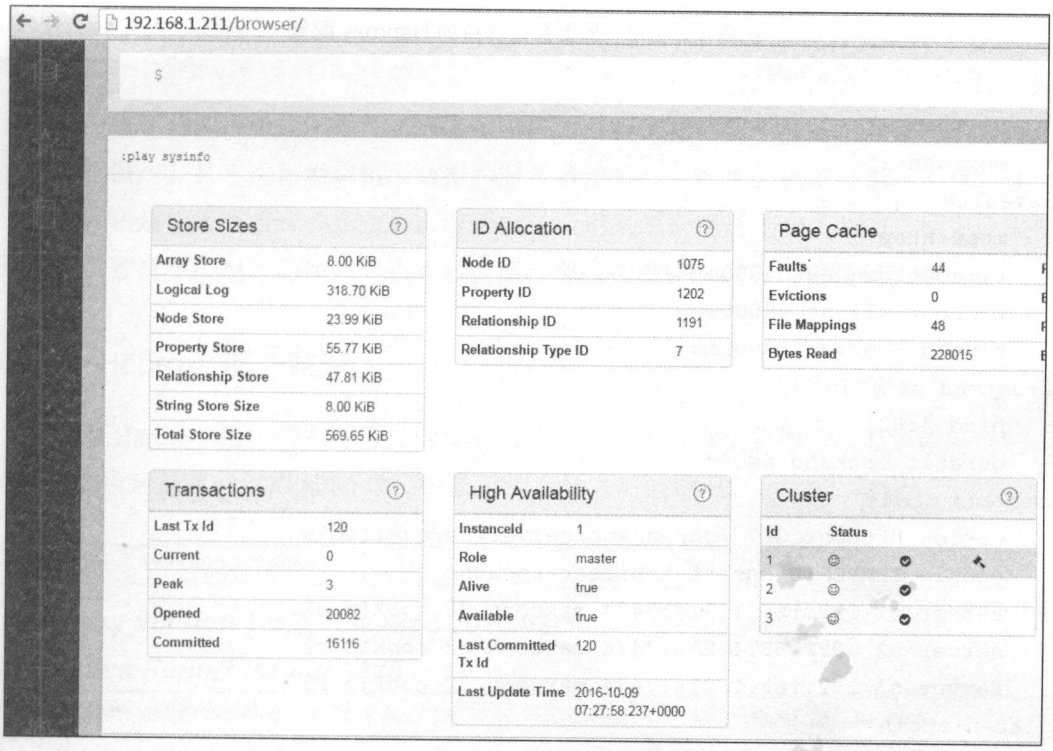


图 8-7　使用负载均衡之后 Neo4j 的 Web 控制台上的监控信息

不幸的是，这时候除了可以监控，不能正常存取数据。如果想在控制台中存取数据，则将会显示如下错误：

```
Unrecognized transaction id. Transaction may have timed out and been rolled back.
```

即使通过应用工程来访问也是一样的。为什么会这样呢？这是因为使用 HTTP 方式连接服务器不是一个长连接，而是一个短连接，这样，对于每个查询请求来说，从发送请求到返回结果至少要连接两次服务器，而每次连接服务器时都被 Haproxy 切换了服务器实例，从而导致一个请求不能在一个事务中处理完全，所以存取数据都会失败。

基于上述分析，我们修改 Haproxy 的配置，使用 Cookie 来保持会话，如代码清单 8-2 所示。

在配置中，增加了一行代码“cookie SERVERID insert indirect nocache”，即使用 Cookie 方式来保存每个客户端的会话 ID。

代码清单 8-2　加上会话保持的 Haproxy 配置

```
global
    daemon
    maxconn 256
defaults
    mode http
    timeout connect 5000ms
    timeout client 50000ms
    timeout server 50000ms
frontend http-in
    bind *:80
    default_backend neo4j
backend neo4j
    option httpchk GET /db/manage/server/ha/available
    cookie SERVERID insert indirect nocache
    server s1 192.168.1.214:7474 maxconn 32 cookie s1
    server s2 192.168.1.215:7474 maxconn 32 cookie s2
    server s3 192.168.1.216:7474 maxconn 32 cookie s3
listen admin
    bind *:8080
    stats enable
    stats uri /
    stats auth admin:123456
```

使用上述配置重启 Haproxy 代理服务器。

现在，在 Web 控制台上，如果再刷新浏览器并执行“:play sysinfo”指令查看监控信息，就可以发现始终停留在一个服务器实例上，再也不会进行自动切换了。这时候再进行任何数据存取操作，就可以正常执行了。

如果在应用项目中使用负载均衡服务，则可以使用如代码清单 8-3 所示的配置，即以 HTTP 驱动的方式来连接数据库。

代码清单 8-3　应用系统连接服务器配置

```
compiler=org.neo4j.ogm.compiler.MultiStatementCypherCompiler
driver=org.neo4j.ogm.drivers.http.driver.HttpDriver
URI=http://192.168.1.211
```

```
username = neo4j
password = 12345678
```

使用这个配置来连接数据库，对于一个应用系统来说，并不清楚正在使用的是哪个服务器实例，并且它跟使用单机版数据库服务器没有什么区别。也就是说，不管是使用单机版，还是使用分布式的企业版，对于一个应用系统来说是完全透明的。如果原来使用单机版数据库，现在切换到分布式数据库，那么在程序上也不用做任何改变。

8.2.2　Haproxy 服务监控

使用 Haproxy 提供的服务监控管理后台，可以实时查看负载均衡服务的使用情况。如图 8-8 所示是我们使用 8080 端口登录监控后台之后的情况。

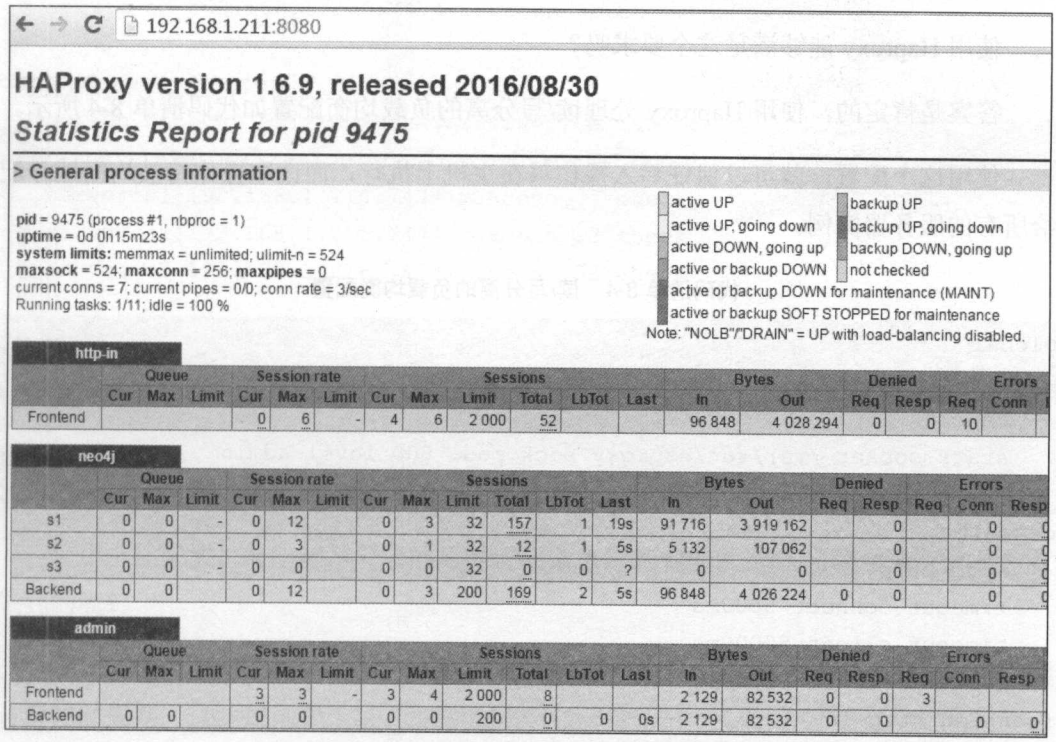


图 8-8　Haproxy 服务监控

从图 8-8 中的 Backend 服务 neo4j 中可以看出 Haproxy 配置中的三个服务 s1、s2 和

s3 的运行情况。其中，在 Sessions 的记录中可以看出服务 s1、s2 已经有访问记录，而 s3 处于空闲状态。

8.3 实现读/写分离的负载均衡服务

虽然 Neo4j 集群中的主从机都可以进行读/写操作，但是如果在从机中写入，数据同步到主机，则比在主机中写入同步到从机会有些许延迟。对于一般的应用来说，也许这样并没有什么影响，但对于一个实时性要求较高的系统来说，显然这样做是不太合理的。所以最好能控制数据写入只在主机上执行。至于数据读取的访问，不管在哪个实例上执行都无所谓。这就要求我们配置的负载均衡服务能够提供读/写分离的处理机制。

使用 Haproxy 能够满足这个要求吗？

答案是肯定的，使用 Haproxy 处理读/写分离的负载均衡配置如代码清单 8-4 所示。

使用这个配置，就可以保证写入操作只在主机上执行，而读取操作将被均衡地分配给所有的服务器实例。

代码清单 8-4　读/写分离的负载均衡配置

```
global
    daemon
    maxconn 256
    stats socket /var/run/haproxy.sock mode 600 level admin
    stats timeout 2m
defaults
    mode http
    timeout connect 5000ms
    timeout client 50000ms
    timeout server 50000ms
frontend http-in
    bind *:80
    acl write_method method POST DELETE PUT
    acl write_hdr hdr_val(X-Write) eq 1
    acl write_payload payload(0,0) -m reg -i CREATE|MERGE|SET|DELETE|REMOVE
```

```
    acl tx_cypher_endpoint path_beg /db/data/transaction
    use_backend neo4j-master if write_hdr
    use_backend neo4j-master if tx_cypher_endpoint write_payload
    use_backend neo4j-all if tx_cypher_endpoint
    use_backend neo4j-master if write_method
    default_backend neo4j-all

backend neo4j-all
    option httpchk GET /db/manage/server/ha/available
HTTP/1.0\r\nAuthorization:\Basic\ bmVvNGo6MTIzNDU2Nzg=
    #cookie SERVERID insert indirect nocache
    balance source
    server s1 192.168.1.214:7474 maxconn 32 check
    server s2 192.168.1.215:7474 maxconn 32 check
    server s3 192.168.1.216:7474 maxconn 32 check

backend neo4j-master
    option httpchk GET /db/manage/server/ha/master
HTTP/1.0\r\nAuthorization:\Basic\ bmVvNGo6MTIzNDU2Nzg=
    server s1 192.168.1.214:7474 maxconn 32 check
    server s2 192.168.1.215:7474 maxconn 32 check
    server s3 192.168.1.216:7474 maxconn 32 check

listen admin
    bind *:8080
    stats enable
    stats uri /
    stats auth admin:123456
```

使用上述配置，就能对客户端的数据访问请求进行智能的读/写分离处理。这是怎么做到的呢？

首先，使用 Haproxy 的 ACL（Access Control Lists）并结合正则表达式定义了一些访问控制规则。这些规则如下：

- 如果通过 Rest 方式访问，请求方法为 POST、DELETE、PUT 的则可以认为是写入请求。

- 如果请求头中有“X-Write”等于“1”的，则也可以归为写入请求。
- 如果使用的 Cypher 查询语言中包含 CREATE|MERGE|SET|DELETE| REMOVE 关键字，则也可以归为写入请求。

其次，配置两个后端（Backend）服务：一个为 neo4j-all，使用/db/manage/server/ha/available 设定了可以访问所有可用的实例；另一个为 neo4j-master，使用/db/manage/server/ha/master 设定了只能访问主机实例。

这样就可以根据访问规则进行判断了。如果是写入请求，则使用 neo4j-master 后端服务；其他请求都使用 neo4j-all 后端服务。这样就实现了读/写分离的要求。

这里，我们在 neo4j-all 后端服务中使用一个负载均衡算法 balance source 来记住访问的源，实现了记住请求客户端的方法。这样就可以保证一个客户端的请求不会因为被切换了访问的实例而不能保证一个事务的完整性。当然，这里也可以使用前面用过的方法，即使用 Cookie 的方式来记住一个客户端的方法。

另外，在两个后端服务中，我们使用了 HTTP/1.0\r\nAuthorization 的方式来完成 Neo4j 服务器的认证。配置中的这个字符串“bmVvNGo6MTIzNDU2Nzg=”是服务器认证的用户名和密码，它是通过 username:password 的格式，再使用 Base64 编码算法生成的。这个字符串是我们使用数据库的用户和密码 neo4j:12345678 生成的。如果你的数据库用户名和密码跟这个相同，则仍然可以使用这个字符串；否则就要使用 Base64 来生成了。

使用上面这个配置来启动 Haproxy 代理服务器，就可以实现智能的读/写分离处理机制。

如图 8-9 所示是使用读/写分离负载均衡配置之后打开 Haproxy 管理后台的监控情况。从图中可以看出，后端服务 neo4j-master 只有一个可用实例，即主机 s1；后端服务 neo4j-all 可以使用所有可用的实例。如果在应用工程或 Neo4j 的 Web 客户端控制台上执行一些读/写请求操作，就可以从 Session 的 Total 中看到客户端的访问情况。

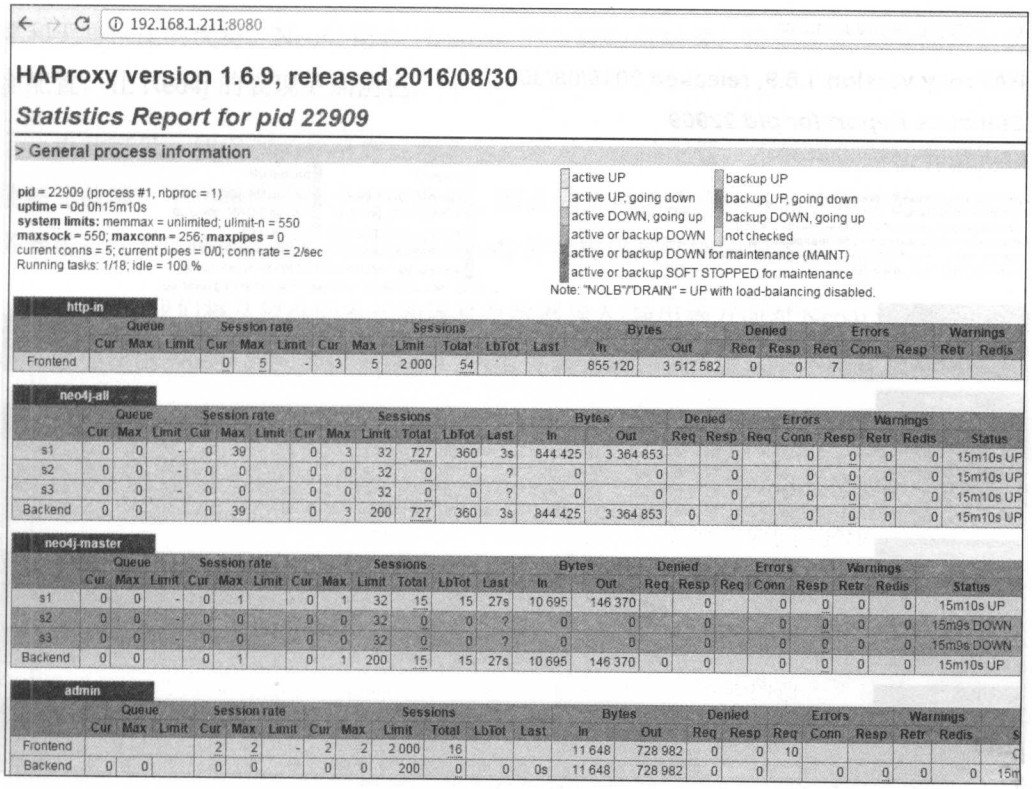
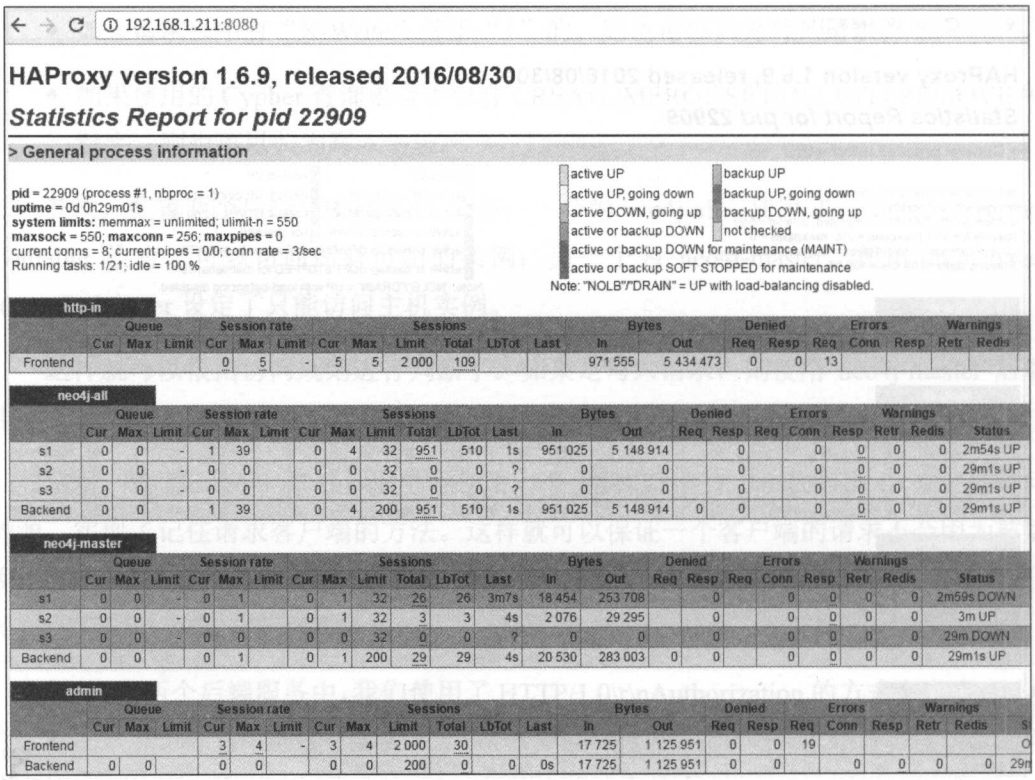


图 8-9　具有读/写分离处理机制的监控情况

这个读/写分离负载均衡配置的智能机制还体现在，如果主机出现故障，则后端服务 neo4j-master 中的主机将会进行自动切换。例如，我们重启一下实例 s1 的 Neo4j 服务器，刷新一下监控控制台，就会出现如图 8-10 所示的情况，即主机从实例 s1 切换到实例 s2 上。

对于上面的配置，如果在后端服务 neo4j-all 中将原来的配置/db/manage/server/ha/available 改为/db/manage/server/ha/slave，则可实现在后端服务 neo4j-all 中只能访问从机。这样更改之后，就可以控制读取操作只在从机上进行。

使用 Haproxy 智能的读/写分离负载均衡配置，能使 Neo4j 的分布式服务器发挥最优化的性能。



HAProxy version 1.6.9, released 2016/08/30

Statistics Report for pid 22909

> General process information

pid = 22909 (process #1, nbproc = 1)
uptime = 0d 0h29m01s
system limits: memmax = unlimited; ulimit-n = 550
maxsock = 550; maxconn = 256; maxpipes = 0
current conns = 8; current pipes = 0/0; conn rate = 3/sec
Running tasks: 1/21; idle = 100 %

active UP | backup UP
active UP, going down | backup UP, going down
active DOWN, going up | backup DOWN, going up
active or backup DOWN | not checked
active or backup DOWN for maintenance (MAINT)
active or backup SOFT STOPPED for maintenance
Note: "NOLB"/"DRAIN" = UP with load-balancing disabled.

http-in

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings	
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis
Frontend				0	5	-	5	5	2 000	109			971 555	5 434 473	0	0	13				

neo4j-all

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status
s1	0	0	-	1	39		0	4	32	951	510	1s	951 025	5 148 914		0		0	0	0	0	2m54s UP
s2	0	0	-	0	0		0	0	32	0	0	?	0	0		0		0	0	0	0	29m1s UP
s3	0	0	-	0	0		0	0	32	0	0	?	0	0		0		0	0	0	0	29m1s UP
Backend	0	0		1	39		0	4	200	951	510	1s	951 025	5 148 914	0	0		0	0	0	0	29m1s UP

neo4j-master

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status
s1	0	0	-	0	1		0	1	32	26	26	3m7s	18 454	253 708		0		0	0	0	0	2m59s DOWN
s2	0	0	-	0	1		0	1	32	3	3	4s	2 076	29 295		0		0	0	0	0	3m UP
s3	0	0	-	0	0		0	0	32	0	0	?	0	0		0		0	0	0	0	29m DOWN
Backend	0	0		0	1		0	1	200	29	29	4s	20 530	283 003	0	0		0	0	0	0	29m1s UP

admin

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	S
Frontend				3	4	-	3	4	2 000	30			17 725	1 125 951	0	0	19					O
Backend	0	0		0	0		0	0	200	0	0	0s	17 725	1 125 951	0	0		0	0	0	0	29m

图 8-10　主机自动切换的监控情况

上述配置文件可通过如下网址获得（包含在第 7 章的实例工程文档之中）：

https://github.com/mr-csj/neo4j-social/blob/master/data/src/main/doc/haproxy-rw.cfg。

使用 Haproxy 负载均衡配置和 Neo4j 的高可用分布式管理架构，即可配置出一个高可用并且可持续扩展的数据库服务管理系统，而这样一个管理系统实际上等同于一个数据库的私有云服务。

8.4　小结

本章通过 Neo4j 企业版服务器的安装和使用演示，使我们了解到，使用 Neo4j 的分布式服务器，不但安装和配置简单明了，而且跟其他任何数据库系统相比，具有得天独

厚和无与伦比的优势。Neo4j 结合 Haproxy 的负载均衡配置，特别是读/写分离的负载均衡配置，让 Neo4j 的表现更加出色。

由此可见，使用 Neo4j 分布式服务器，经过合理的配置，不但可以保证数据库服务能够无故障、不间断地提供全天候服务，而且完全可以适应业务高速发展的需要。对服务器进行动态的横向扩展，可以保障 Neo4j 持续提供高性能的服务。

下一章，我们将从数据的备份和恢复及数据导入/导出等方面对 Neo4j 进行深入了解，从中体会使用 Neo4j 的安全可靠保障，从而更加全面和深入地理解 Neo4j 数据库体系。

第9章
Neo4j的数据安全及备份

从前面一些章节的介绍和学习中我们知道，Neo4j 具有太多令人振奋的优势，不但在性能上是无与伦比的，在分布式的高可用和负载均衡服务中也具有绝对优势，并且因为简单易用的特性和数据库在设计上的灵活性，给开发者提供了更多的便利，提高了开发工作的效率。另外，图数据结构的特性也让 Neo4j 能够提供其他一些数据库所不能提供的功能，从而使之更加出类拔萃。

本章将从数据安全保障方面介绍数据的备份与恢复、数据库安全保障、数据的导入与导出、故障恢复与事务日志、数据库升级及数据迁移等内容。

9.1 数据的备份与恢复

在企业版中，我们使用分布式的安装方式，其实就已经给数据库提供了多个具有完全数据的副本，并且都是实时在线更新的。除此之外，在企业版中，还有一个数据备份的功能，并且同时支持完整和增量备份功能。

9.1.1 数据备份

在 Neo4j 中备份数据，最简单的方法是将整个数据目录复制到另一个地方存储。这种方法不管是社区版还是企业版都可以使用，因为这只需使用文件系统的复制功能就能轻松实现。

企业版可以提供在线实时增量备份功能，还可以配置成本地或者远程进行备份。

如果需要进行远程备份，则只需在配置文件中启用如下所示的设置。

```
# To allow remote backups, uncomment this line:
dbms.backup.address=0.0.0.0:6362
```

完整和增量备份都使用 neo4j-backup 工具来执行。首先创建一个备份目录，如下：

```
mkdir /mnt/backup/neo4j-backup
```

然后就可以使用 neo4j-backup 工具执行备份了，如下：

```
./bin/neo4j-backup -host 192.168.1.214 -to /mnt/backup/neo4j-backup
```

在第一次执行备份时，将进行完整备份，以后每次执行都只对有更改的地方进行备份，即只执行增量备份。

必须要注意的是，备份的存储目录必须使用上述目录结构，如果使用其他目录，则会报错。例如，我尝试使用存储目录/opt/neo4j-backup，在执行备份时就报了如下错误提示。这应该是 neo4j-backup 这个工具的问题。

```
[root@localhost neo4j-01]# ./bin/neo4j-backup -host 192.168.1.214 -to
/opt/neo4j-backup
Performing backup from '192.168.1.214:6362'
2016-10-12 09:57:15.750+0000 INFO  [o.n.b.BackupService] Previous backup
found, trying incremental backup.
Exception in thread "main" java.lang.RuntimeException: /opt/neo4j-backup
doesn't contain a database
      at org.neo4j.backup.BackupService.doIncrementalBackup(BackupService.
java:191)
      at org.neo4j.backup.BackupService.doIncrementalBackupOrFallback
ToFull(BackupService.java:240)
      at org.neo4j.backup.BackupTool.doBackup(BackupTool.java:241)
      at org.neo4j.backup.BackupTool.executeBackup(BackupTool.java:204)
      at org.neo4j.backup.BackupTool.runBackup(BackupTool.java:195)
      at org.neo4j.backup.BackupTool.run(BackupTool.java:131)
      at org.neo4j.backup.BackupTool.main(BackupTool.java:91)
```

为了实现按时增量备份，我们可以采用定时器的方式，定时执行下列指令。

```
./bin/neo4j-backup -host 192.168.1.214 -to /mnt/backup/neo4j-backup
```

使用如下指令可以编辑一个文件，构造一个定时器。

```
vi neo4j-cro
```

文件的内容如下所示，即每天在 6:00～22:00 每隔 30 分钟进行一次增量备份。

```
#30 minutes in 06 to 22
0,30 06-22 * * * /neo4j-01/bin/neo4j-backup -host 192.168.1.214 -to
/mnt/backup/neo4j-backup
```

最后，使用下列指令启用定时器。

```
crontab neo4j-cro
```

这样就实现了实时在线备份功能。

9.1.2 清理备份日志

在 Linux 系统中，定时任务每执行一次都会以邮件方式通知用户，如下：

```
“您在 /var/spool/mail/root 中有邮件”
```

使用指令“cat / var/spool/mail/root”，可以看到这是一个日志文件，里面包含很多如下所示的内容。

```
From root@localhost.localdomain  Wed Oct 12 16:30:03 2016
Return-Path: <root@localhost.localdomain>
X-Original-To: root
Delivered-To: root@localhost.localdomain
Received: by localhost.localdomain (Postfix, from userid 0)
      id 9612120000BB; Wed, 12 Oct 2016 16:30:03 +0800 (CST)
From: "(Cron Daemon)" <root@localhost.localdomain>
To: root@localhost.localdomain
Subject: Cron <root@localhost> /neo4j-01/bin/neo4j-backup -host 192.168.1.214
-to /mnt/backup/neo4j-backup
Content-Type: text/plain; charset=GBK
Auto-Submitted: auto-generated
Precedence: bulk
X-Cron-Env: <XDG_SESSION_ID=1867>
```

```
X-Cron-Env: <XDG_RUNTIME_DIR=/run/user/0>
X-Cron-Env: <LANG=zh_CN.GBK>
X-Cron-Env: <SHELL=/bin/sh>
X-Cron-Env: <HOME=/root>
X-Cron-Env: <PATH=/usr/bin:/bin>
X-Cron-Env: <LOGNAME=root>
X-Cron-Env: <USER=root>
Message-Id: <20161012083003.9612120000BB@localhost.localdomain>
Date: Wed, 12 Oct 2016 16:30:02 +0800 (CST)

Performing backup from '192.168.1.214:6362'
2016-10-12 08:30:02.249+0000 INFO  [o.n.b.BackupService] Previous backup
found, trying incremental backup.
Done
......
```

这个日志文件对我们来说并没有什么用处，而且随着时间的推移将会越来越大，占用和浪费了一定的磁盘空间。所以必须清理这封邮件或者禁止这个日志文件的输出。

通过使用指令“crontab –e”可以修改定时器的配置。在定时器中，我们可以在执行备份的指令中增加一个重定向输出的设定“>/dev/null 2>&1”，即可达到清理邮件的目的。

在重定向输出的设定“>/dev/null 2>&1”中，表示先将标准输出重定向到一个本来不存在的设备/dev/null 中，这将产生一个标准错误，然后再将这个标准错误重定向到标准输出中，这样就再也不会输出日志了。完成后的定时器配置如下：

```
#30 minutes in 06 to 22
0,30 06-22 * * * /neo4j-01/bin/neo4j-backup -host 192.168.1.214 -to
/mnt/backup/neo4j-backup >/dev/null 2>&1
```

9.1.3　数据恢复

数据恢复使用文件系统方式就可以进行，社区版和企业版的恢复方式有点差异。

对于单机安装的社区版来说，数据恢复可按下列顺序进行：

（1）停止服务器。

（2）复制备份文件到原数据库文件夹中。

（3）启动服务器。

对于分布式的企业版来说，数据恢复必须按下列顺序进行：

（1）停止分布式服务器中的所有实例。

（2）复制备份文件到任何一个实例的原数据库文件夹中，最好选择作为主机的实例。

（3）启动主机和其他所有实例服务器。

9.2 数据库安全保障

基于安全考虑，在安装 Neo4j 数据库时，默认的配置只允许在本地访问。只有将配置文件 neo4j.conf 中下列配置的最后一行的注释去掉，才能允许局域网的远程访问。

```
# HTTP Connector
dbms.connector.http.type=HTTP
dbms.connector.http.enabled=true
# To accept non-local HTTP connections, uncomment this line
# dbms.connector.http.address=0.0.0.0:7474
```

如果是在互联网环境中，那么还可以将 Neo4j 服务器配置成 HTTPS 的方式进行访问。

不管是本地访问，还是远程访问，包括 RESTful 方式的访问，都需要提供用户名和密码进行认证，只有认证通过才能正常使用数据库。这是最基本的数据库安全保障。

9.3 数据的导入与导出

Neo4j 提供了一些数据导入与导出的方法和工具，使用这些方法和工具，可以很方便地执行数据的导入与导出操作。

9.3.1　使用 neo4j-import 导入数据

在安装 Neo4j 服务器时，在可执行文件目录中有一个 neo4j-import 可执行文件，这是一个数据导入工具，使用这个工具，可以使用 CSV 数据文件通过导入的方式来新建一个数据库。

下面我们使用一个在 Windows 系统中安装的 Neo4j 服务器来执行这个测试。

例如，我们现在有 movies.csv、actors.csv、roles.csv 三个数据文件，可以使用这些文件来导入数据，从而新建一个数据库。这些文件的内容如下所示。

movies.csv 文件的内容：

```
movieId:ID,title,year:int,:LABEL
tt0133093,"The Matrix",1999,Movie
tt0234215,"The Matrix Reloaded",2003,Movie;Sequel
tt0242653,"The Matrix Revolutions",2003,Movie;Sequel
```

actors.csv 文件的内容：

```
personId:ID,name,:LABEL
keanu,"Keanu Reeves",Actor
laurence,"Laurence Fishburne",Actor
carrieanne,"Carrie-Anne Moss",Actor
```

roles.csv 文件的内容：

```
:START_ID,role,:END_ID
keanu,"Neo",tt0133093
keanu,"Neo",tt0234215
keanu,"Neo",tt0242653
laurence,"Morpheus",tt0133093
laurence,"Morpheus",tt0234215
laurence,"Morpheus",tt0242653
carrieanne,"Trinity",tt0133093
carrieanne,"Trinity",tt0234215
carrieanne,"Trinity",tt0242653
```

现在使用下列指令，即可导入上面三个文件的数据，然后在目录 D:\neo4j-community-3.0.3\data\databases\graph.db 中生成一个全新的数据库。

```
neo4j-import --into D:\neo4j-community-3.0.3\data\databases\graph.db -nodes
movies.csv -nodes  actors.csv --relationships:ACTED_IN
 roles.csv
```

需要注意的是，在使用上述指令之前必须先确认数据库已经停止运行，并且上面的目录结构不存在，或者虽然有这个目录结构，但不存在任何数据。这个目录是安装数据库时数据的默认存放目录，如果存在，则可以先将它删除，然后再执行上述测试指令。

当一切准备妥当之后，就可以执行上面的指令了。它的执行过程和情况如下：

```
D:\neo4j-community-3.0.3\bin>neo4j-import --into
D:\neo4j-community-3.0.3\data\databases\graph.db -nodes  movies.csv -nodes
actors.csv --relationships:ACTED_IN
 roles.csv
Neo4j version: 3.0.3
Importing the contents of these files into
D:\neo4j-community-3.0.3\data\databases\graph.db:
Nodes:
  D:\neo4j-community-3.0.3\bin\movies.csv

  D:\neo4j-community-3.0.3\bin\actors.csv
Relationships:
  :ACTED_IN
  D:\neo4j-community-3.0.3\bin\roles.csv

Available memory:
  Free machine memory: 1.88 GB
  Max heap memory : 1.75 GB

Nodes
[*>:??-------------------------------------------------------------------
--------------]    0
Done in 227ms
Prepare node index

Done in 240ms
Calculate dense nodes

Done in 39ms
```

```
Node --> Relationship Sparse

Done in 11ms
Relationship --> Relationship Sparse

Done in
Minority relationships

Done in 28ms
Node counts

Done in 41ms
Relationship counts

Done in 10ms

IMPORT DONE in 4s 12ms. Imported:
  6 nodes
  0 relationships
  15 properties
```

从执行结果中可以看出，其中增加了 6 个节点和 15 个属性。现在启动服务器，在数据库客户端的 Web 控制台上就可以查到如图 9-1 所示的数据。

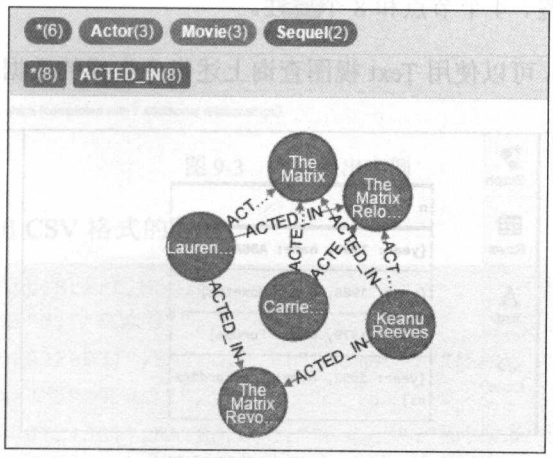


图 9-1　通过导入数据新建数据库的结果

9.3.2 使用 Cypher 导入数据

在实际使用中，通常我们并不需要创建一个全新的数据库，而是要在原有的数据库中导入另外一些数据。所以一般所说的导入数据可以使用 Cypher 查询语言的 LOAD 指令进行操作。只不过 LOAD 指令不能从文件系统中导入数据，它必须通过 HTTP 协议来执行数据导入操作。下面通过一个简单的例子来说明使用 LOAD 指令的方法。

例如，我们可以在一个 Web 服务中提供一个可以通过 HTTP 协议访问的数据文件 artists.csv，这个文件的内容如下：

```
"1","ABBA","1992"
"2","Roxette","1986"
"3","Europe","1979"
"4","The Cardigans","1992"
```

启动 Web 服务，即可使用如下所示的 Cypher 查询语句导入上述文件的数据，用来创建节点 Artist。其中，“line[1]”表示使用第 2 列的数据，“line[2]”表示使用第 3 列的数据。

```
LOAD CSV FROM 'http://localhost/artists.csv' AS line
CREATE (:Artist { name: line[1], year: toInt(line[2])})
```

执行上述指令的结果如下：

```
Added 4 labels, created 4 nodes, set 8 properties, statement executed in 1295 ms.
```

即创建了 4 个标签、4 个节点和 8 个属性。

在 Web 控制台上，可以使用 Text 视图查询上述指令生成的数据，结果如图 9-2 所示。

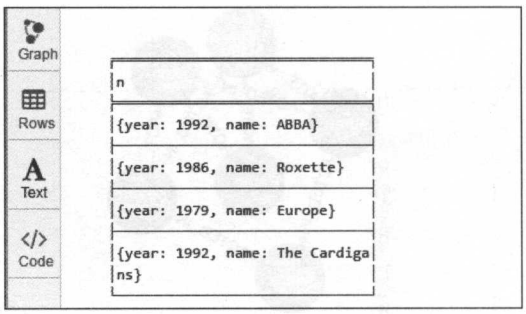


图 9-2 导入数据结果

9.3.3　导出数据

在 Neo4j 的 Web 客户端控制台中查询出来的数据，可以使用 CSV 或 JSON 等格式导出到文件中。例如，下列查询设计使用观众对电影进行评分的关系，返回了观众节点、关系类型、评分星级和电影节点等数据。

```
MATCH (n:Person)-[r:RATED]-(m:Movie) RETURN n as Person,type(r) as
Relationship,r.stars as Start,m as Movie LIMIT 5
```

执行查询的结果如图 9-3 所示。在结果视图中通过单击右上角的 Export to file 按钮，弹出导出选项，选择相应的选项就可以将查询结果导出到文件中。

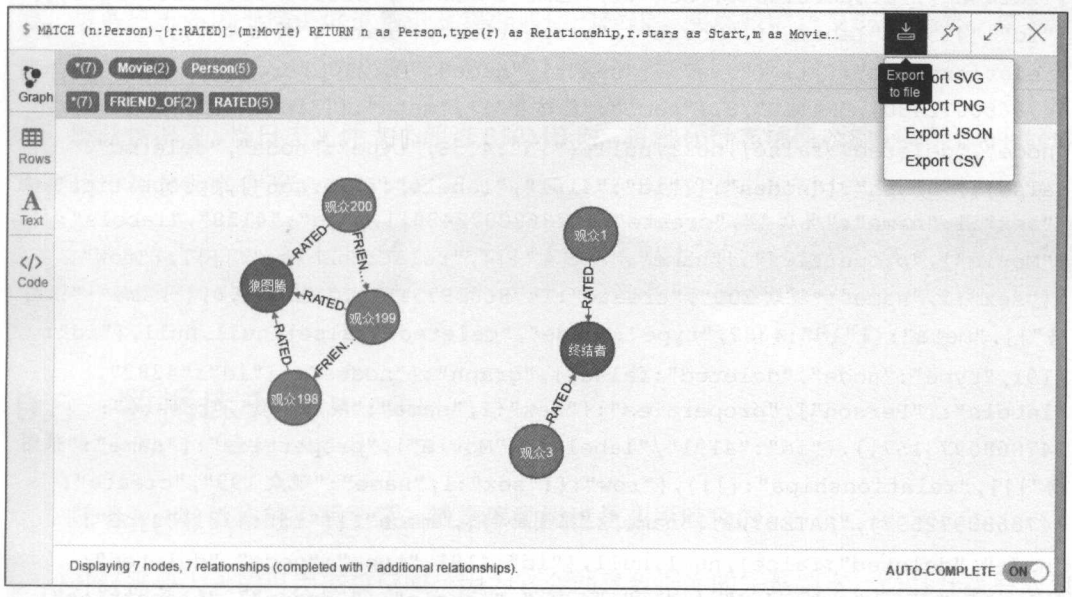


图 9-3　数据导出实例

如下所示是导出 CSV 格式的数据：

```
Person,Relationship,Start,Movie
"{""sex"":1,""name"":""观众 3"",
""create"":1478680072693}",RATED,9,"{""name"":""终结者""}"
"{""sex"":1,""name"":""观众 1"",
""create"":1478680072430}",RATED,8,"{""name"":""终结者""}"
"{""sex"":1,""name"":""观众 200"",
```

```
""create"":1478680973157}",RATED,0,"{""name"":""狼图腾""}"
"{""sex"":1,""name"":""观众 199"",
""create"":1478680972597}",RATED,9,"{""name"":""狼图腾""}"
"{""sex"":1,""name"":""观众 198"",
""create"":1478680972021}",RATED,8,"{""name"":""狼图腾""}"
```

如下所示是导出 JSON 格式的数据：

```
{"columns":["Person","Relationship","Start","Movie"],"data":[{"row":[{"sex":
1,"name":"观众 3","create":1478680072693},"RATED",9,{"name":"终结者"}],
"meta":[{"id":4163,"type":"node","deleted":false},null,null,{"id":4158,
"type":"node","deleted":false}],"graph":{"nodes":[{"id":"4163","labels":
["Person"],"properties":{"sex":1,"name":"观众 3","create":1478680072693}},
{"id":"4158","labels":["Movie"],"properties":{"name":"终结者"}}],
"relationships":[]}},{"row":[{"sex":1,"name":"观众 1","create":
1478680072430},"RATED",8,{"name":"终结者"}],"meta":[{"id":4161,"type":
"node","deleted":false},null,null,{"id":4158,"type":"node","deleted":
false}],"graph":{"nodes":[{"id":"4161","labels":["Person"],"properties":
{"sex":1,"name":"观众 1","create":1478680072430}},{"id":"4158","labels":
["Movie"],"properties":{"name":"终结者"}}],"relationships":[]}},{"row":
[{"sex":1,"name":"观众 200","create":1478680973157},"RATED",0,{"name":"狼图
腾"}],"meta":[{"id":4382,"type":"node","deleted":false},null,null,{"id":
4191,"type":"node","deleted":false}],"graph":{"nodes":[{"id":"4382",
"labels":["Person"],"properties":{"sex":1,"name":"观众 200","create":
1478680973157}},{"id":"4191","labels":["Movie"],"properties":{"name":"狼图
腾"}}],"relationships":[]}},{"row":[{"sex":1,"name":"观众 199","create":
1478680972597},"RATED",9,{"name":"狼图腾"}],"meta":[{"id":4381,"type":
"node","deleted":false},null,null,{"id":4191,"type":"node","deleted":
false}],"graph":{"nodes":[{"id":"4381","labels":["Person"],"properties":
{"sex":1,"name":"观众 199","create":1478680972597}},{"id":"4191","labels":
["Movie"],"properties":{"name":"狼图腾"}}],"relationships":[]}},{"row":
[{"sex":1,"name":"观众 198","create":1478680972021},"RATED",8,{"name":"狼图
腾"}],"meta":[{"id":4380,"type":"node","deleted":false},null,null,{"id":
4191,"type":"node","deleted":false}],"graph":{"nodes":[{"id":"4380",
"labels":["Person"],"properties":{"sex":1,"name":"观众 198","create":
1478680972021}},{"id":"4191","labels":["Movie"],"properties":{"name":"狼图
腾"}}],"relationships":[]}}],"stats":{"contains_updates":false,"nodes_
created":0,"nodes_deleted":0,"properties_set":0,"relationships_created":
```

```
0,"relationship_deleted":0,"labels_added":0,"labels_removed":0,"indexes_
added":0,"indexes_removed":0,"constraints_added":0,"constraints_removed":0}}
```

9.4　故障恢复与事务日志

当系统出现不可避免的故障时，比如系统崩溃、断电等事故，就需要使用 Neo4j 的事务日志恢复数据库。

事务日志记录了数据库中的所有操作，它是数据库需要恢复的情况下的真实数据来源。事务日志还用于提供增量备份及集群通信的操作。实现故障恢复的前提是，对于任何给定的配置，将至少保留最新的非空事务日志。

默认情况下，当日志文件大小超过 250MB 时，将进行日志切换。这可以在 neo4j.conf 中使用下列参数来更改。

```
# Retention policy for transaction logs needed to perform recovery and backups.
#dbms.tx_log.rotation.retention_policy=7 days
```

9.5　数据库升级

随着 Neo4j 版本的不断更新，数据库升级也是在所难免的。

升级需要大量的可用磁盘空间，因为它会创建数据库的完整副本，而且升级后的数据库也可能需要更大的数据文件。升级时，建议在现有数据库文件的大小之上提供额外 50%的磁盘空间。

9.5.1　从 2.x 升级到 3.0.3

首先安装 Neo4j 3.0.3。

如果需要使用原来配置文件的一些配置，则可以使用 config-migrator 工具迁移配置

文件。config-migrator 工具可以在安装路径的 tools 目录中找到。使用如下指令执行配置文件的迁移：

```
java -jar config-migrator.jar path/to/neo4j2.3 path/to/neo4j3.0
```

其中，“path/to/neo4j2.3”指 Neo4j 2.x 的安装路径，“path/to/neo4j3.0”指 Neo4j 3.x 的安装路径。

在迁移过程中，记下输出的任何警告，然后查看生成的配置文件，根据需要进行编辑。

其次，使用 neo4j-admin 工具导入数据。

在 Neo4j 的安装目录的可执行文件中有一个 neo4j-admin 工具，可以用来从旧数据库中导入数据到现有数据库中，它的指令方式如下：

```
neo4j-admin import -mode database  -database  <database-name>  -from
<source-directory>
```

下列代码是一个从 2.3.4 迁移到 3.0.3 的实际操作过程及其结果。

```
D:\neo4j-community-3.0.3\bin>neo4j-admin import -mode database -database
graph.db
-from D:\neo4j\ebuy.db
Imported data from D:\neo4j\ebuy.db to
D:\neo4j-community-3.0.3\data\databases\graph.db
```

默认的数据库设置将使用文件目录 graph.db 来存放数据。如果不能正常使用 graph.db，则可在配置文件 neo4j.conf 中通过数据库的名称来设定，如下：

```
# The name of the database to mount
dbms.active_database=graph.db
```

导入数据之后，在启动数据库之前，先在 neo4j.conf 配置中使用如下一行配置：

```
dbms.allow_format_migration = true
```

然后启动 Neo4j 3.0.3。这时可以看到数据库升级的过程，如下所示：

```
D:\neo4j-community-3.0.3\bin>neo4j console
2016-11-30 12:32:26.891+0000 INFO  Starting...
```

```
2016-11-30 12:32:27.580+0000 INFO  Bolt enabled on localhost:7687.
2016-11-30 12:32:27.761+0000 INFO  Starting upgrade of database
2016-11-30 12:32:27.818+0000 INFO  Migrating Indexes (1/3):
2016-11-30 12:32:27.822+0000 INFO    10% completed
2016-11-30 12:32:27.822+0000 INFO    20% completed
2016-11-30 12:32:27.822+0000 INFO    30% completed
......
2016-11-30 12:32:27.936+0000 INFO    80% completed
2016-11-30 12:32:27.936+0000 INFO    90% completed
2016-11-30 12:32:27.936+0000 INFO    100% completed
2016-11-30 12:32:28.511+0000 INFO  Successfully finished upgrade of database
2016-11-30 12:32:32.600+0000 INFO  Started.
2016-11-30 12:32:33.493+0000 INFO  Remote interface available at
http://localhost:7474/
```

有关升级和进度指示的信息都将记录到 debug.log 中。

升级完成后，应将 dbms.allow_format_migration 设置为 false，或直接将这条配置注释掉，如下：

```
dbms.allow_format_migration = false
```

现在重启服务器，就能正常使用升级后的数据库了。

按照官方操作手册的说明，最好在升级完成后立即进行一次完整备份。

9.5.2　在 3.x 之间升级

如果是在 3.x 之间升级就比较简单了，如从 3.0.3 升级到 3.0.6。在安装新数据库后，如果需要，则复制一些旧的配置到新数据库的配置中，然后直接复制整个数据库的目录 graph.db 到新安装数据库的目录中。这些工作完成之后，同样在启动新数据库之前要将 dbms.allow_format_migration 设置为 true，升级完成后再设置成 false，或者直接将该条配置注释掉。

9.6 小结

本章从数据库安全保障方面介绍了 Neo4j 数据的备份与恢复、数据库安全保障、数据的导入与导出、故障恢复与事务日志、数据库升级和数据迁移等内容。从中可以了解到，使用 Neo4j 数据库是相当安全可靠的，特别是使用企业版数据库，不但能使用分布式服务器，而且能使用实时在线备份功能，更能得到最大限度的安全保障。

结 束 语

非常高兴你花费了一些宝贵的时间来阅读本书，同时恭喜你，通过本书的学习，或许让你也喜欢上了 Neo4j。

本书从介绍 Neo4j 这个独特的图数据库开始，与你共同经历了从简单的 API 练习，到 Neo4j 服务器的安装配置，再到 Cypher 查询语言的语法设计及使用，以及在应用项目中的开发实践等过程，共同见证了这一独特的数据库的优秀表现。

如此优秀的数据库，还可以使用颇具绅士风度的 Spring Data Neo4j 来优雅地使用，更加让人感叹不止。而更为可贵的还是 Neo4j 的高可用和高性能表现。Neo4j 不但可以适应业务的高速发展，持续保持最初的活力，而且在安全可靠方面同样得到了有力保障。当我们使用关系型数据库陷入窘境时，是不是 Neo4j 给我们带来了令人喜悦的希望呢？

本书虽然尝试从全方位阐释 Neo4j 的整个生态体系，但是对 Neo4j 的介绍不可能做到面面俱到，有些方面和领域还需要你我在未来的实际应用中共同探索。如果你打算在你的下一个项目中使用 Neo4j 数据库，那么你将极愿意去发掘和体验使用 Neo4j 的乐趣。祝你成功！

附录A
参考资料

1. Neo4j 下载链接：https://neo4j.com/download/other-releases/。
2. Neo4j 开发手册：http://neo4j.com/docs/developer-manual/3.0/。
3. Neo4j 操作手册：http://neo4j.com/docs/operations-manual/3.0/。
4. Cypher 查询语言使用参考：http://neo4j.com/docs/cypher-refcard/3.0/。
5. 各种编程语言使用 Neo4j 指南：https://neo4j.com/developer/language-guides/。
6. Spring Data Neo4j 参考：https://projects.spring.io/spring-data-neo4j/。
7. Spring Boot 参考手册：http://docs.spring.io/spring-boot/docs/current/reference/html/。
8. Neo4j 精选 GraphGists：http://graphgist.neo4j.com/。
9. GraphGists 门户：http://portal.graphgist.org/。
10. Neo4j 企业版安装手册：http://neo4j.com/docs/operations-manual/current/tutorial/#ha-setup-tutorial。
11. Haproxy 配置参考：https://www.haproxy.com/doc/aloha/7.0/haproxy/healthchecks.html。

投稿联络：安 娜
微信&QQ：80303489
邮箱：anna@phei.com.cn

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

Neo4j全栈开发

Neo4j是一个高性能的NoSQL图数据库，现实世界中快速增长的数据要求有很高的查询性能，而Neo4j正是为适应这种大规模的数据增长而产生的数据库。

本书以Neo4j 3.x版本为基础，通过Neo4j API的开发实例、Cypher查询语言的算法设计、Spring Data Neo4j的对象建模和持久化设计等方面，详细分享了构建可扩展、高可用的Neo4j分布式架构的实操细节。同时使用Spring Boot开发框架，结合具体的应用实例，详细阐述了使用Neo4j进行前后端设计的详细过程和方法，从此让Neo4j提供全天候、无间断的高可靠服务。

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

策划编辑：安　娜
责任编辑：徐津平
封面设计：李　玲

上架建议：数据库

ISBN 978-7-121-31447-6

定价：69.00元