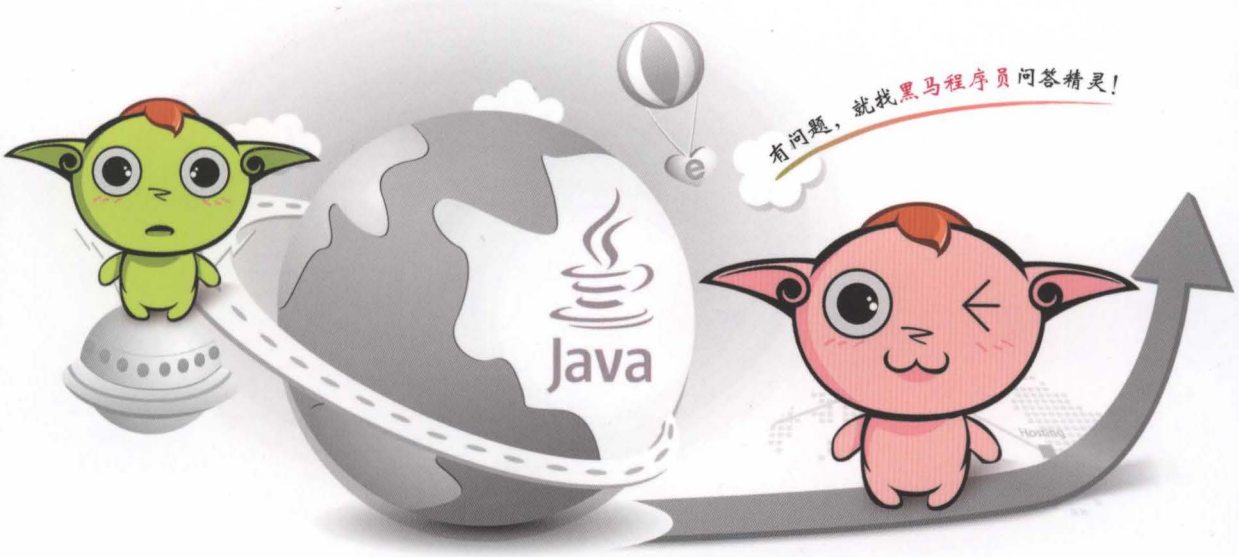
工业和信息化“十三五”
人才培养规划教材

国家信息技术紧缺人才培养工程
指定教材

Java EE企业级应用开发教程
(Spring+Spring MVC+MyBatis)

黑马程序员 / 编著



本书共 18 章，涵盖了轻量级 Java EE 框 架 Spring+Spring MVC+MyBatis 的实际应用技术，每个知识点都配备了实战案例，共 36 道思考题。

提供免费教学资源，包含 18 个精美教学 PPT、1 个客户管理项目、1000 道测试题、长达 35 小时的教学视频等。

添加 QQ 或微信号 208695827，获取教学答案、源码和“助学金红包”。

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

播妞

播妞—— IT技术女神，由传智播客旗下高端教育品牌黑马程序员推出，专门服务于计算机相关专业的大学生及IT爱好者；可随时提供教材源代码、习题答案、免费视频教程和就业宝典等。

播妞倾情寄语：

你加，或者不加我
我就在这里
不离 不弃
来我这里
或者
让我住进你的心里

★ 小心！此处常有“助学金红包”出没！★
快来领取！

播妞QQ：208695827

播妞微信：208695827

教师获取教材配套资源

添加微信/QQ

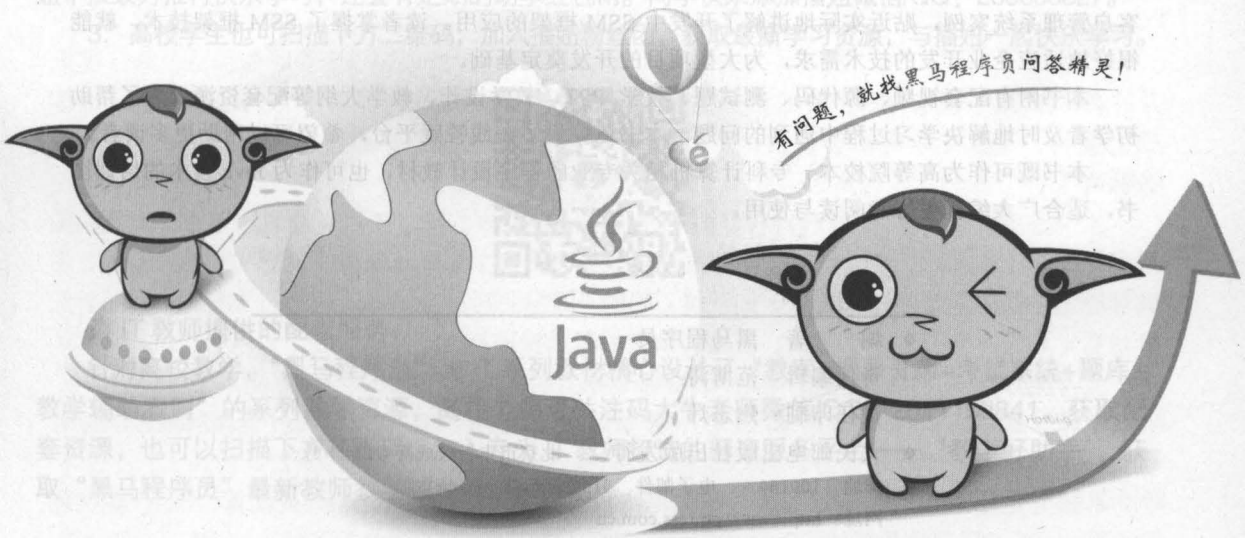
2011168841

工业和信息化“十三五”
人才培养规划教材

国家信息技术紧缺人才培养工程
指定教材

Java EE企业级应用开发教程
(Spring+Spring MVC+MyBatis)

黑马程序员 / 编著



人 民 邮 电 出 版 社
北 京

图书在版编目（CIP）数据

Java EE企业级应用开发教程 ：Spring+Spring MVC+MyBatis / 黑马程序员编著. -- 北京 ：人民邮电出版社，2017.9

工业和信息化“十三五”人才培养规划教材

ISBN 978-7-115-46102-5

Ⅰ. ①J… Ⅱ. ①黑… Ⅲ. ①JAVA语言－程序设计－高等学校－教材 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2017)第189003号

内 容 提 要

本书详细讲解了 Java EE 中 Spring、Spring MVC 和 MyBatis 三大框架（以下简称“SSM”）的基本知识和应用。本书在对知识点进行描述时采用了大量案例，可以更好地帮助读者学习和理解 SSM 的核心技术。

本书共 18 章，第 1~5 章主要讲解 Spring 的基本知识和应用，其中包括 Spring 的基本应用、Spring 中的 Bean、Spring AOP、Spring 的数据库开发以及 Spring 的事务管理。第 6~10 章主要讲解了 MyBatis 的相关知识，其中包含初识 MyBatis、MyBatis 的核心配置、动态 SQL、MyBatis 的关联映射以及 MyBatis 与 Spring 的整合。第 11～17 章主要讲解了 Spring MVC 的相关知识，其中包含 Spring MVC 入门，Spring MVC 的核心类和注解，数据绑定，JSON 数据交互和 RESTful 支持，拦截器，文件上传和下载以及 SSM 框架整合。第 18 章讲解整个 SSM 框架的总结与综合运用，全章通过一个 BOOT 客户管理系统案例，贴近实际地讲解了开发中 SSM 框架的应用。读者掌握了 SSM 框架技术，就能很好地适应企业开发的技术需求，为大型项目的开发奠定基础。

本书附有配套视频、源代码、测试题、教学 PPT、教学设计、教学大纲等配套资源。为了帮助初学者及时地解决学习过程中遇到的问题，本书还提供了在线答疑平台，希望可以帮助更多读者。

本书既可作为高等院校本、专科计算机相关专业的程序设计教材，也可作为 Java 技术的培训图书，适合广大编程爱好者阅读与使用。

◆ 编　　著　黑马程序员

责任编辑　范博涛

责任印制　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号

邮编　100164　　电子邮件　315@ptpress.com.cn

网址　http://www.ptpress.com.cn

三河市海波印务有限公司印刷

◆ 开本：787×1092　1/16

印张：21　　2017 年 9 月第 1 版

字数：525 千字　　2017 年 9 月河北第 1 次印刷

定价：49.80 元

读者服务热线：(010)81055256　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

序 言 PREFACE

江苏传智播客教育科技股份有限公司（简称传智播客）是一家致力于培养高素质软件开发人才的科技公司，“黑马程序员”是传智播客旗下高端 IT 教育品牌。

“黑马程序员”的学员多为大学毕业后，想从事 IT 行业，但各方面条件还不成熟的年轻人。“黑马程序员”的学员筛选制度非常严格，筛选制度包括了严格的技术测试、自学能力测试，还包括性格测试、压力测试、品德测试等。百里挑一的残酷筛选制度确保了学员质量，并降低了企业的用人风险。

自“黑马程序员”成立以来，教学研发团队一直致力于打造精品课程资源，不断在产、学、研 3 个层面创新自己的执教理念与教学方针，并集中“黑马程序员”的优势力量，有针对性地出版了计算机系列教材 50 多册，制作了教学视频数十套，发表各类技术文章数百篇。

“黑马程序员”不仅斥资研发 IT 系列教材，还为高校师生提供以下配套学习资源与服务。

为大学生提供的配套服务：

1. 专业的辅助学习平台“博学谷”（http://yx.boxuegu.com），专业老师在线为您答疑解惑。
2. 针对高校学生在学习过程中存在的压力等问题，我们还面向大学生量身打造了“播妞”。播妞不仅致力推行快乐学习，还会有定期的助学红包雨。同学快来添加播妞微信/QQ：208695827。
3. 高校学生也可扫描下方二维码，加入播妞粉丝团，获取最新学习资源，与播妞一起快乐学习。

为 IT 教师提供的配套服务：

针对高校教学，“黑马程序员”为 IT 系列教材精心设计了“教案+授课资源+考试系统+题库+教学辅助案例”的系列教学资源，高校老师可关注码大牛老师微信/QQ：2011168841，获取配套资源，也可以扫描下方二维码，加入专为 IT 教师打造的师资服务平台——“教学好助手”，获取“黑马程序员”最新教师教学辅助资源相关动态。

传智播客和黑马程序员

FOREWORD 前言

为什么要学习本书

当前轻量级 Java EE（Java 企业版）应用开发通常会采用两种方式：一种是以 SSH（Struts + Spring+Hibernate）框架为核心的组合方式，另一种是以 SSM（Spring+Spring MVC+ MyBatis）框架为核心的组合方式。使用这两种组合方式的项目都使 Java EE 架构具有高度的可维护性和可扩展性，同时极大地提高了项目的开发效率，降低了开发和维护的成本，因此，这两种组合方式已成为当前各个企业项目开发的首选。

两种组合框架的相同点在于都以 Spring 框架为核心，而两者的主要不同之处在于 MVC 的实现方式（Struts 与 Spring MVC），以及 ORM 持久化方面（Hibernate 与 Mybatis）。SSH 较注重配置开发，其中的 Hibernate 对 JDBC 的完整封装更加面向对象，对增、删、改、查的数据维护更自动化，但 SQL 优化方面较弱，且学习门槛稍高；SSM 更注重注解式开发，且 ORM 实现更加灵活，SQL 优化更简便，学习容易入门。目前来说，传统企业项目的开发，使用 SSH 框架比较多，而对性能要求较高的互联网项目，通常会选用 SSM 框架。因此，对于想从事互联网项目开发的人员来说，学好 SSM 框架，就显得比较重要了。

如何使用本书

本书适用于具有 Java 基础和一定的 Java Web 相关知识的读者学习，对于没有任何基础的读者，建议先学习本套系教材中的《Java 基础案例教程》和《Java Web 程序设计任务教程》。

本书在 Spring 4.3 + Spring MVC 4.3 + MyBatis 3 版本的基础上，详细讲解了这三大框架的基础知识和使用方法。在编写时，作者力求将一些非常复杂、难以理解的思想和问题简单化，使读者能够轻松理解并快速掌握这些知识点。同时，本书还对每个知识点都进行了深入的分析，并针对重要知识点精心设计了案例，以提高读者的实践操作能力。

全书共分为 18 个章节，接下来分别对每个章节的内容进行简单的介绍，具体如下。

- 第 1 章讲解了 Spring 框架入门的一些基础知识，主要内容包括 Spring 框架的概念、作用、优点、体系结构、下载和使用、核心容器、入门程序以及依赖注入等。
- 第 2 章对 Spring 中的 Bean 进行了详细讲解，主要内容包括 Bean 的配置、Bean 实例化的三种方式、Bean 的作用域、Bean 的生命周期以及 Bean 的三种装配方式。
- 第 3 章讲解了 Spring 框架中 AOP 的相关知识，主要内容包括 AOP 的介绍、动态代理、基于代理类的 AOP 实现以及如何使用 AspectJ 框架来进行 AOP 开发等。
- 第 4 章对 Spring 的数据库开发进行了详细讲解。主要包括 Spring JDBC 中的核心类和配置的介绍，以及 Spring JDBCTemplate 的常用方法。
- 第 5 章对 Spring 中的事务管理进行了详细讲解，主要内容包括 Spring 事务管理的核心接口、事务管理的方式，以及基于 XML 方式和基于 Annotation 方式的声明式事务处理的使用。

- 第 6 章对 MyBatis 框架的基础知识进行了讲解，主要内容包括 MyBatis 框架的概念，特点，下载和使用，工作原理以及一个简单的查、增、改、删案例。
- 第 7 章对 MyBatis 的核心配置进行了详细讲解。主要包括 MyBatis 中的两个重要核心对象 SqlSessionFactory 和 SqlSession，以及 MyBatis 配置文件和映射文件的详细讲解。
- 第 8 章对 MyBatis 框架的动态 SQL 知识进行了详细讲解，主要内容包括常用的动态 SQL 元素介绍以及常用动态 SQL 元素的使用。
- 第 9 章对 MyBatis 框架中的关联映射知识进行了详细讲解，主要内容包括关联关系中的一对一、一对多和多对多的处理。
- 第 10 章对 MyBatis 与 Spring 框架的整合使用进行了详细讲解，主要内容包括整合的环境搭建、传统 DAO 开发方式的整合和基于 Mapper 编程方式的整合。
- 第 11 章对 Spring MVC 框架的入门知识进行了详细讲解，主要内容包括 Spring MVC 的介绍、入门程序的编写以及 Spring MVC 框架的工作流程。
- 第 12 章对 Spring MVC 的核心类及其相关注解的使用进行了详细的讲解，主要内容包括对前端控制器 DispatcherServlet 的作用和配置的介绍，@Controller 注解和@RequestMapping 注解类型的使用，以及视图解析器的定义和配置。
- 第 13 章对 Spring MVC 中的数据绑定知识进行了详细讲解，主要内容包括数据绑定介绍、简单数据绑定和复杂数据绑定。
- 第 14 章对 Spring MVC 中的 JSON 数据交互和 RESTful 支持进行了详细的讲解，主要内容包括 JSON 的介绍、Spring MVC 中的 JSON 数据交互以及 RESTful 支持的使用。
- 第 15 章对 Spring MVC 中拦截器的使用进行了详细讲解，主要内容包括拦截器的定义和配置，单个拦截器和多个拦截器的执行流程，以及拦截器的实际应用。
- 第 16 章对 Spring MVC 环境下的文件上传和下载进行了详细讲解，主要内容包括如何实现文件上传和下载，以及如何实现中文名称文件的下载。
- 第 17 章对 SSM 框架的整合知识进行了详细讲解，主要内容包括 SSM 框架整合的环境搭建及 SSM 的整合过程。
- 第 18 章对 SSM 框架的实际应用（BOOT 客户管理系统）进行了详细讲解，主要内容包括系统概述、数据库设计、系统环境搭建，以及用户登录模块和客户管理模块的开发实现等。

在学习过程中，读者一定要亲自实践书中的案例代码，如果不能完全理解书中所讲的知识点，可以登录博学谷平台，通过平台中的教学视频进行辅助学习。学习完一个知识点后，要及时在博学谷平台上进行测试以巩固学习内容。另外，如果读者在理解知识点的过程中遇到困难，建议不要纠结于某个地方，可以先往后学习。通常来讲，随着对后面知识的不断深入了解，前面看不懂的知识点一般就能理解了。如果读者在动手练习的过程中遇到问题，建议多思考，理清思路，认真分析问题发生的原因，并在问题解决后多总结。

致谢

本书的编写和整理工作由传智播客教育科技股份有限公司完成，其中主要的参与人员有吕春林、陈欢、韩永蒙、石荣新、杜宏、梁桐、王友军、冯佳等。全体人员在近一年的编写过程中，付出了很多辛勤的汗水，在此一并表示衷心的感谢。

意见反馈

尽管我们尽了最大的努力，但教材中难免会有不妥之处，欢迎各界专家和读者朋友们来函给予宝贵意见，我们将不胜感激。您在阅读本书时，如发现任何问题或有不认同之处可以通过电子邮件与我们取得联系。

请发送电子邮件至：itcast_book@vip.sina.com。

黑马程序员

2017-6-12 于北京

目 录 CONTENTS

第 1 章 Spring 的基本应用 1
1.1 Spring 概述 2
1.1.1 什么是 Spring 2
1.1.2 Spring 框架的优点 2
1.1.3 Spring 的体系结构 3
1.1.4 Spring 的下载及目录结构 4
1.2 Spring 的核心容器 6
1.2.1 BeanFactory 6
1.2.2 ApplicationContext 6
1.3 Spring 的入门程序 8
1.4 依赖注入 11
1.4.1 依赖注入的概念 11
1.4.2 依赖注入的实现方式 12
1.5 本章小结 13

第 2 章 Spring 中的 Bean 15
2.1 Bean 的配置 16
2.2 Bean 的实例化 17
2.2.1 构造器实例化 17
2.2.2 静态工厂方式实例化 18
2.2.3 实例工厂方式实例化 20
2.3 Bean 的作用域 21
2.3.1 作用域的种类 21
2.3.2 singleton 作用域 22
2.3.3 prototype 作用域 23
2.4 Bean 的生命周期 23
2.5 Bean 的装配方式 25
2.5.1 基于 XML 的装配 25
2.5.2 基于 Annotation 的装配 28
2.5.3 自动装配 32
2.6 本章小结 33

第 3 章 Spring AOP 34
3.1 Spring AOP 简介 35
3.1.1 什么是 AOP 35
3.1.2 AOP 术语 36
3.2 动态代理 36
3.2.1 JDK 动态代理 36
3.2.2 CGLIB 代理 39
3.3 基于代理类的 AOP 实现 41
3.3.1 Spring 的通知类型 42
3.3.2 ProxyFactoryBean 42
3.4 AspectJ 开发 45
3.4.1 基于 XML 的声明式 AspectJ 45
3.4.2 基于注解的声明式 AspectJ 51
3.5 本章小结 55

第 4 章 Spring 的数据库开发 56
4.1 Spring JDBC 57
4.1.1 Spring JdbcTemplate 的解析 57
4.1.2 Spring JDBC 的配置 57
4.2 Spring JdbcTemplate 的常用方法 59
4.2.1 execute() 59
4.2.2 update() 63
4.2.3 query() 68
4.3 本章小结 71

第 5 章 Spring 的事务管理 72
5.1 Spring 事务管理概述 73
5.1.1 事务管理的核心接口 73
5.1.2 事务管理的方式 75
5.2 声明式事务管理 75
5.2.1 基于 XML 方式的声明式事务 75
5.2.2 基于 Annotation 方式的声明式事务 80
5.3 本章小结 83

第 6 章 初识 MyBatis 84
6.1 什么是 MyBatis 85
6.2 MyBatis 的下载和使用 86
6.3 MyBatis 的工作原理 87
6.4 MyBatis 入门程序 88
6.4.1 查询客户 88
6.4.2 添加客户 96
6.4.3 更新客户 97
6.4.4 删除客户 99
6.5 本章小结 100

第 7 章 MyBatis 的核心配置 ... 101
7.1 MyBatis 的核心对象 102
7.1.1 SqlSessionFactory 102
7.1.2 SqlSession 102
7.2 配置文件 105
7.2.1 主要元素 105
7.2.2 <properties>元素 106
7.2.3 <settings>元素 106
7.2.4 <typeAliases>元素 108
7.2.5 <typeHandler>元素 109
7.2.6 <objectFactory>元素 110
7.2.7 <plugins>元素 111
7.2.8 <environments>元素 111
7.2.9 <mappers>元素 113
7.3 映射文件 114
7.3.1 主要元素 114
7.3.2 <select>元素 115
7.3.3 <insert>元素 115
7.3.4 <update>元素和<delete>元素 117
7.3.5 <sql>元素 118
7.3.6 <resultMap>元素 119
7.4 本章小结 122

第 8 章 动态 SQL 123
8.1 动态 SQL 中的元素 124

8.2 <if>元素 124
8.3 <choose>、<when>、<otherwise>元素 126
8.4 <where>、<trim>元素 129
8.5 <set>元素 130
8.6 <foreach>元素 132
8.7 <bind>元素 134
8.8 本章小结 135

第 9 章 MyBatis 的关联映射 137

9.1 关联关系概述 138
9.2 一对一 139
9.3 一对多 146
9.4 多对多 151
9.5 本章小结 155

第 10 章 MyBatis 与 Spring 的整合 157

10.1 整合环境搭建 158
10.1.1 准备所需 JAR 包 158
10.1.2 编写配置文件 159
10.2 传统 DAO 方式的开发整合 161
10.3 Mapper 接口方式的开发整合 165
10.3.1 基于 MapperFactoryBean 的整合 165
10.3.2 基于 MapperScannerConfigurer 的整合 167
10.4 测试事务 168
10.5 本章小结 171

第 11 章 Spring MVC 入门.....172

11.1 Spring MVC 概述 173
11.2 第一个 Spring MVC 应用 ... 173
11.3 Spring MVC 的工作流程 177
11.4 本章小结 178

第 12 章 Spring MVC 的核心类和注解 179

12.1 DispatcherServlet 180
12.2 Controller 注解类型 180
12.3 RequestMapping 注解类型 181
12.3.1 @RequestMapping 注解的使用 181
12.3.2 @RequestMapping 注解的属性 182
12.3.3 组合注解 183
12.3.4 请求处理方法的参数类型和返回类型 184
12.4 ViewResolver（视图解析器） 186
12.5 应用案例——基于注解的 Spring MVC 应用 186
12.6 本章小结 188

第 13 章 数据绑定 190

13.1 数据绑定介绍 191
13.2 简单数据绑定 192
13.2.1 绑定默认数据类型 192
13.2.2 绑定简单数据类型 194
13.2.3 绑定 POJO 类型 195
13.2.4 绑定包装 POJO 198
13.2.5 自定义数据绑定 201
13.3 复杂数据绑定 205
13.3.1 绑定数组 205
13.3.2 绑定集合 207
13.4 本章小结 210

第 14 章 JSON 数据交互和 RESTful 支持 211

14.1 JSON 数据交互 212
14.1.1 JSON 概述 212

14.1.2 JSON 数据转换 213
14.2 RESTful 支持 221
14.2.1 什么是 RESTful 221
14.2.2 应用案例——用户信息查询 221
14.3 本章小结 224

第 15 章 拦截器 225

15.1 拦截器概述 226
15.1.1 拦截器的定义 226
15.1.2 拦截器的配置 227
15.2 拦截器的执行流程 227
15.2.1 单个拦截器的执行流程 227
15.2.2 多个拦截器的执行流程 230
15.3 应用案例——实现用户登录权限验证 233
15.4 本章小结 238

第 16 章 文件上传和下载 240

16.1 文件上传 241
16.1.1 文件上传概述 241
16.1.2 应用案例——文件上传 243
16.2 文件下载 247
16.2.1 实现文件下载 247
16.2.2 中文名称的文件下载 249
16.3 本章小结 251

第 17 章 SSM 框架整合 252

17.1 整合环境搭建 253
17.1.1 整合思路 253
17.1.2 准备所需 JAR 包 253
17.1.3 编写配置文件 254
17.2 整合应用测试 258
17.3 本章小结 262

第 18 章 BOOT 客户管理系统 263

18.1 系统概述 264
18.1.1 系统功能介绍 264
18.1.2 系统架构设计 264
18.1.3 文件组织结构 265
18.1.4 系统开发及运行环境 266
18.2 数据库设计 266
18.3 系统环境搭建 267
18.3.1 准备所需 JAR 包 267
18.3.2 准备数据库资源 269
18.3.3 准备项目环境 269
18.4 用户登录模块 274
18.4.1 用户登录 274
18.4.2 实现登录验证 281
18.4.3 退出登录 284
18.5 客户管理模块 286
18.5.1 查询客户 286
18.5.2 添加客户 307
18.5.3 修改客户 315
18.5.4 删除客户 322
18.6 本章小结 325

Java EE

Chapter 1

第1章 Spring的基本应用

学习目标

- 了解 Spring 的概念和优点
- 理解 Spring 中的 IoC 和 DI 思想
- 掌握 ApplicationContext 容器的使用
- 掌握属性 setter 方法注入的实现

Spring 是当前主流的 Java Web 开发框架，它是为了解决企业应用开发的复杂性问题而产生的。对于一个 Java 开发者来说，掌握 Spring 框架的使用，已是其必备的技能之一。本章将对 Spring 框架的基础知识进行详细的讲解。

1.1 Spring 概述

1.1.1 什么是 Spring

Spring 是由 Rod Johnson 组织和开发的一个分层的 Java SE/EE full-stack（一站式）轻量级开源框架，它以 IoC（Inversion of Control，控制反转）和 AOP（Aspect Oriented Programming，面向切面编程）为内核，使用基本的 JavaBean 来完成以前只可能由 EJB（Enterprise Java Beans，Java 企业 Bean）完成的工作，取代了 EJB 的臃肿、低效的开发模式。

Spring 致力于 Java EE 应用各层的解决方案，在表现层它提供了 Spring MVC 以及与 Struts 框架的整合功能；在业务逻辑层可以管理事务、记录日志等；在持久层可以整合 MyBatis、Hibernate、JdbcTemplate 等技术。因此，可以说 Spring 是企业应用开发很好的“一站式”选择。虽然 Spring 贯穿于表现层、业务逻辑层和持久层，但它并不想取代那些已有的框架，而是以高度的开放性与它们进行无缝整合。

1.1.2 Spring 框架的优点

Spring 具有简单、可测试和松耦合等特点，从这个角度出发，Spring 不仅可以用于服务器端开发，也可以应用于任何 Java 应用的开发中。关于 Spring 框架优点的总结，具体如下。

- 非侵入式设计

Spring 是一种非侵入式（non-invasive）框架，它可以使应用程序代码对框架的依赖最小化。

- 方便解耦、简化开发

Spring 就是一个大工厂，可以将所有对象的创建和依赖关系的维护工作都交给 Spring 容器管理，大大地降低了组件之间的耦合性。

- 支持 AOP

Spring 提供了对 AOP 的支持，它允许将一些通用任务，如安全、事务、日志等进行集中式处理，从而提高了程序的复用性。

- 支持声明式事务处理

只需要通过配置就可以完成对事务的管理，而无须手动编程。

- 方便程序的测试

Spring 提供了对 Junit4 的支持，可以通过注解方便地测试 Spring 程序。

- 方便集成各种优秀框架

Spring 不排斥各种优秀的开源框架，其内部提供了对各种优秀框架（如 Struts、Hibernate、MyBatis、Quartz 等）的直接支持。

- 降低 Java EE API 的使用难度

Spring 对 Java EE 开发中非常难用的一些 API（如 JDBC、JavaMail 等），都提供了封装，使这些 API 应用难度大大降低。

1.1.3　Spring 的体系结构

Spring 框架采用的是分层架构，它一系列的功能要素被分成 20 个模块，这些模块大体分为 Core Container、Data Access/Integration、Web、AOP（Aspect Oriented Programming）、Instrumentation、Messaging 和 Test，如图 1-1 所示。

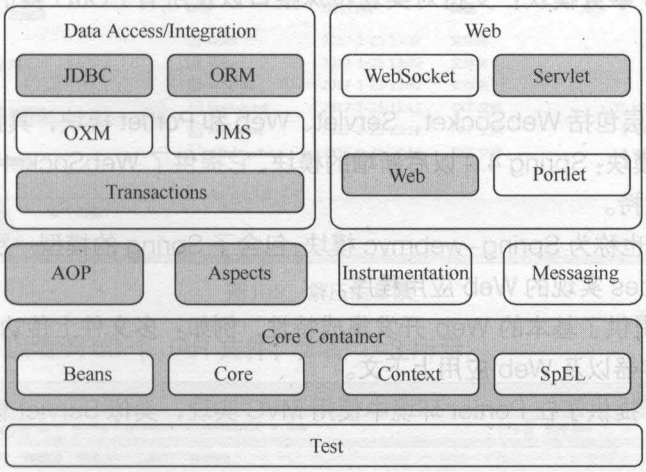


图1-1　Spring的体系结构

在图 1-1 中，包含了 Spring 框架的所有模块，其中，灰色背景模块为本书中所涉及的主要模块。接下来分别对体系结构中的模块作用进行简单介绍，具体如下。

1. Core Container（核心容器）

Spring 的核心容器是其他模块建立的基础，它主要由 Beans 模块、Core 模块、Context 模块、Context-support 模块和 SpEL（Spring Expression Language，Spring 表达式语言）模块组成，具体介绍如下。

- Beans 模块：提供了 BeanFactory，是工厂模式的经典实现，Spring 将管理对象称为 Bean。
- Core 核心模块：提供了 Spring 框架的基本组成部分，包括 IoC 和 DI 功能。
- Context 上下文模块：建立在 Core 和 Beans 模块的基础之上，它是访问定义和配置的任何对象的媒介。其中 ApplicationContext 接口是上下文模块的焦点。
- Context-support 模块：提供了对第三方库嵌入 Spring 应用的集成支持，比如缓存（EhCache、Guava、JCache）、邮件服务（JavaMail）、任务调度（CommonJ、Quartz）和模板引擎（FreeMarker、JasperReports、速率）。
- SpEL 模块：是 Spring 3.0 后新增的模块，它提供了 Spring Expression Language 支持，是运行时查询和操作对象图的强大的表达式语言。

2. Data Access/Integration（数据访问/集成）

数据访问/集成层包括 JDBC、ORM、OXM、JMS 和 Transactions 模块，具体介绍如下。

- JDBC 模块：提供了一个 JDBC 的抽象层，大幅度地减少了在开发过程中对数据库操作的编码。
- ORM 模块：对流行的对象关系映射 API，包括 JPA、JDO 和 Hibernate 提供了集成层

支持。

• OXM 模块：提供了一个支持对象/ XML 映射的抽象层实现，如 JAXB、Castor、XMLBeans、JiBX 和 XStream。

• JMS 模块：指 Java 消息传递服务，包含使用和产生信息的特性，自 4.1 版本后支持与 Spring-message 模块的集成。

• Transactions 事务模块：支持对实现特殊接口以及所有 POJO 类的编程和声明式的事务管理。

3. Web

Spring 的 Web 层包括 WebSocket、Servlet、Web 和 Portlet 模块，具体介绍如下。

• WebSocket 模块：Spring 4.0 以后新增的模块，它提供了 WebSocket 和 SockJS 的实现，以及对 STOMP 的支持。

• Servlet 模块：也称为 Spring-webmvc 模块，包含了 Spring 的模型—视图—控制器（MVC）和 REST Web Services 实现的 Web 应用程序。

• Web 模块：提供了基本的 Web 开发集成特性，例如：多文件上传功能、使用 Servlet 监听器来初始化 IoC 容器以及 Web 应用上下文。

• Portlet 模块：提供了在 Portlet 环境中使用 MVC 实现，类似 Servlet 模块的功能。

4. 其他模块

Spring 的其他模块还有 AOP、Aspects、Instrumentation 以及 Test 模块，具体介绍如下。

• AOP 模块：提供了面向切面编程实现，允许定义方法拦截器和切入点，将代码按照功能进行分离，以降低耦合性。

• Aspects 模块：提供了与 AspectJ 的集成功能，AspectJ 是一个功能强大且成熟的面向切面编程（AOP）框架。

• Instrumentation 模块：提供了类工具的支持和类加载器的实现，可以在特定的应用服务器中使用。

• Messaging 模块：Spring 4.0 以后新增的模块，它提供了对消息传递体系结构和协议的支持。

• Test 模块：提供了对单元测试和集成测试的支持。

1.1.4 Spring 的下载及目录结构

Spring 的第一个版本是在 2004 年发布的，经过 10 多年的发展，Spring 的版本也在不断地升级优化中。本书编写时，Spring 的最新版本为 4.3.6，本书的代码都是基于该版本编写实现的，建议读者也下载该版本。

Spring 开发所需的 JAR 包分为两个部分，具体如下。

1. Spring 框架包

Spring 4.3.6 版本的框架压缩包，名称为 spring-framework-4.3.6.RELEASE-dist.zip，此压缩包可以通过地址“http://repo.spring.io/simple/libs-release-local/org/springframework/spring/4.3.6.RELEASE/”下载。下载完成后，将压缩包解压到自定义的文件夹中，解压后的文件目录结构如图 1-2 所示。

在图 1-2 的目录中，docs 文件夹中包含 Spring 的 API 文档和开发规范；libs 文件夹中包含

开发需要的 JAR 包和源码；schema 文件夹中包含开发所需要的 schema 文件，这些文件中定义了 Spring 相关配置文件的约束。

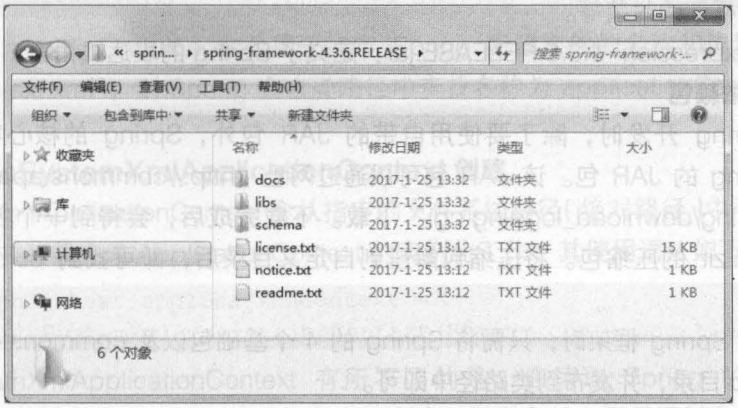


图1-2　解压后目录

打开 libs 目录可以看到 60 个 JAR 文件，如图 1-3 所示。

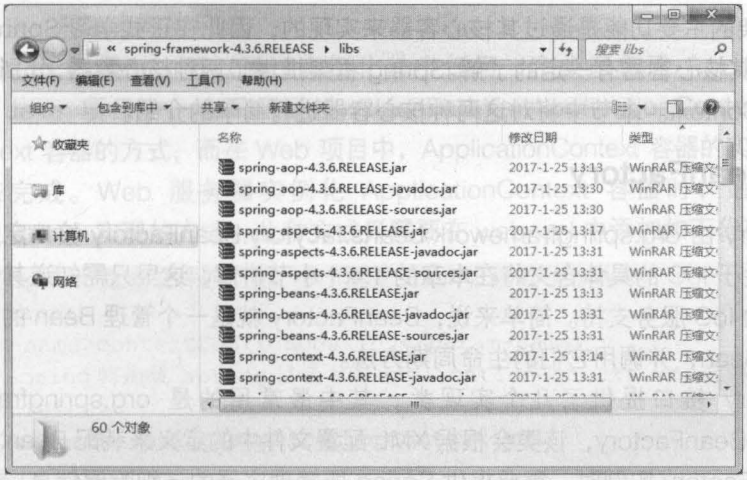


图1-3　libs目录

从图 1-3 可以看出，libs 目录中的 JAR 包分为三类，其中以 RELEASE.jar 结尾的是 Spring 框架 class 文件的 JAR 包；以 RELEASE-javadoc.jar 结尾的是 Spring 框架 API 文档的压缩包；以 RELEASE-sources.jar 结尾的是 Spring 框架源文件的压缩包。整个 Spring 框架由 20 个模块组成，该目录下 Spring 为每个模块都提供了这三类压缩包。

在 libs 目录中，有四个 Spring 的基础包，它们分别对应 Spring 核心容器的四个模块，具体介绍如下。

- spring-core-4.3.6.RELEASE.jar：包含 Spring 框架基本的核心工具类，Spring 其他组件都要用到这个包里的类，是其他组件的基本核心。
- spring-beans-4.3.6.RELEASE.jar：所有应用都要用到的 JAR 包，它包含访问配置文件、创建和管理 Bean 以及进行 Inversion of Control(IoC)或者 Dependency Injection(DI)操作相关的所有类。

• spring-context-4.3.6.RELEASE.jar：Spring 提供了在基础 IoC 功能上的扩展服务，还提供了许多企业级服务的支持，如邮件服务、任务调度、JNDI 定位、EJB 集成、远程访问、缓存以及各种视图层框架的封装等。

• spring-expression-4.3.6.RELEASE.jar：定义了 Spring 的表达式语言。

2. 第三方依赖包

在使用 Spring 开发时，除了要使用自带的 JAR 包外，Spring 的核心容器还需要依赖 commons.logging 的 JAR 包。该 JAR 包可以通过网址“http://commons.apache.org/proper/commons-logging/download_logging.cgi”下载。下载完成后，会得到一个名为 commons-logging-1.2-bin.zit 的压缩包。将压缩包解压到自定义目录后，即可找到 commons-logging-1.2.jar。

初学者学习 Spring 框架时，只需将 Spring 的 4 个基础包以及 commons-logging-1.2.jar 复制到项目的 lib 目录，并发布到类路径中即可。

1.2 Spring 的核心容器

Spring 框架的主要功能是通过其核心容器来实现的，因此在正式学习 Spring 框架的使用之前，有必要先对其核心容器有一定的了解。Spring 框架提供了两种核心容器，分别为 BeanFactory 和 ApplicationContext。本节中将对这两种核心容器进行简单的介绍。

1.2.1 BeanFactory

BeanFactory 由 org.springframework.beans.facytory.BeanFactory 接口定义，是基础类型的 IoC 容器（关于 IoC 的具体含义将在本章的 1.4.1 小节讲解，这里只需知道其表示控制反转），它提供了完整的 IoC 服务支持。简单来说，BeanFactory 就是一个管理 Bean 的工厂，它主要负责初始化各种 Bean，并调用它们的生命周期方法。

BeanFactory 接口提供了几个实现类，其中最常用的是 org.springframework.beans.factory.xml.XmlBeanFactory，该类会根据 XML 配置文件中的定义来装配 Bean。

创建 BeanFactory 实例时，需要提供 Spring 所管理容器的详细配置信息，这些信息通常采用 XML 文件形式来管理，其加载配置信息的语法如下。

```
BeanFactory beanFactory =
new XmlBeanFactory(new FileSystemResource("F: /applicationContext.xml"));
```

这种加载方式在实际开发中并不多用，读者了解即可。

1.2.2 ApplicationContext

ApplicationContext 是 BeanFactory 的子接口，也被称为应用上下文，是另一种常用的 Spring 核心容器。它由 org.springframework.context. ApplicationContext 接口定义，不仅包含了 BeanFactory 的所有功能，还添加了对国际化、资源访问、事件传播等方面的支持。

创建 ApplicationContext 接口实例，通常采用两种方法，具体如下。

1. 通过 ClassPathXmlApplicationContext 创建

ClassPathXmlApplicationContext 会从类路径 classPath 中寻找指定的 XML 配置文件，找

到并装载完成 ApplicationContext 的实例化工作，其使用语法如下。

```
ApplicationContext applicationContext =
    new ClassPathXmlApplicationContext(String configLocation);
```

上述代码中，configLocation 参数用于指定 Spring 配置文件的名称和位置。如果其值为“applicationContext.xml”，则 Spring 会去类路径中查找名称为 applicationContext.xml 的配置文件。

2. 通过 FileSystemXmlApplicationContext 创建

FileSystemXmlApplicationContext 会从指定的文件系统路径(绝对路径)中寻找指定的 XML 配置文件，找到并装载完成 ApplicationContext 的实例化工作，其使用语法如下。

```
ApplicationContext applicationContext =
    new FileSystemXmlApplicationContext(String configLocation);
```

与 ClassPathXmlApplicationContext 有所不同的是，在读取 Spring 的配置文件时，FileSystemXmlApplicationContext 不再从类路径中读取配置文件，而是通过参数指定配置文件的位置，例如“D:/workspaces/applicationContext.xml”。如果在参数中写的不是绝对路径，那么方法调用的时候，会默认用绝对路径来找。这种采用绝对路径的方式，会导致程序的灵活性变差，所以这个方法一般不推荐使用。

在使用 Spring 框架时，可以通过实例化其中任何一个类来创建 ApplicationContext 容器。通常在 Java 项目中，会采用通过 ClassPathXmlApplicationContext 类来实例化 ApplicationContext 容器的方式，而在 Web 项目中，ApplicationContext 容器的实例化工作会交由 Web 服务器来完成。Web 服务器实例化 ApplicationContext 容器时，通常会使用基于 ContextLoaderListener 实现的方式，此种方式只需要在 web.xml 中添加如下代码。

```
<!-- 指定 Spring 配置文件的位置，多个配置文件时，以逗号分隔-->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <!-- Spring 将加载 spring 目录下的 applicationContext.xml 文件 -->
    <param-value>
        classpath:spring/applicationContext.xml
    </param-value>
</context-param>
<!-- 指定以 ContextLoaderListener 方式启动 Spring 容器 -->
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

在本书后面章节讲解三大框架整合以及项目时，将采用基于 ContextLoaderListener 的方式由 Web 服务器实例化 ApplicationContext 容器。

创建 Spring 容器后，就可以获取 Spring 容器中的 Bean。Spring 获取 Bean 的实例通常采用以下两种方法。

- Object getBean(String name)：根据容器中 Bean 的 id 或 name 来获取指定的 Bean，获取之后需要进行强制类型转换。

• <T> T getBean(Class<T> requiredType)：根据类的类型来获取 Bean 的实例。由于此方法为泛型方法，因此在获取 Bean 之后不需要进行强制类型转换。

小提示

BeanFactory 和 ApplicationContext 两种容器都是通过 XML 配置文件加载 Bean 的。二者的主要区别在于，如果 Bean 的某一个属性没有注入，使用 BeanFacotry 加载后，在第一次调用 getBean()方法时会抛出异常，而 ApplicationContext 则在初始化时自检，这样有利于检查所依赖属性是否注入。因此，在实际开发中，通常都优先选择使用 ApplicationContext，而只有在系统资源较少时，才考虑使用 BeanFactory。

1.3 Spring 的入门程序

通过上一小节的学习，相信读者对 Spring 的核心容器已经有了一个初步的了解。为了帮助读者快速地学习 Spring，下面通过一个简单的入门程序来演示 Spring 框架的使用。

（1）在 Eclipse 中，创建一个名为 chapter01 的 Web 项目，将 Spring 的 4 个基础包以及 commons-logging 的 JAR 包复制到 lib 目录中，并发布到类路径下，如图 1-4 所示。

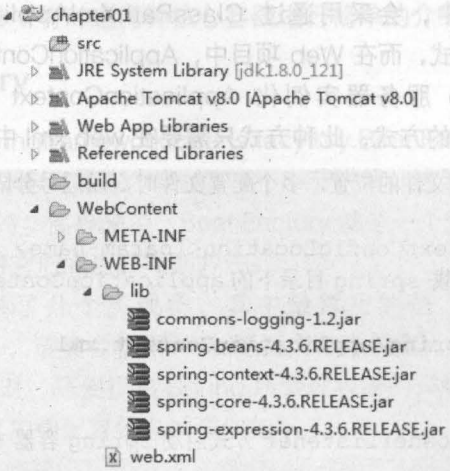


图1-4 导入JAR包

（2）在 src 目录下，创建一个 com.itheima.ioc 包，并在包中创建接口 UserDao，然后在接口中定义一个 say()方法，如文件 1-1 所示。

文件 1-1 UserDao.java

```
1  package com.itheima.ioc;
2  public interface UserDao {
3      public void say();
4  }
```

（3）在 com.itheima.ioc 包下，创建 UserDao 接口的实现类 UserDaoImpl，该类需要实现接口中的 say()方法，并在方法中编写一条输出语句，如文件 1-2 所示。

文件 1-2　UserDaoImpl.java

```
1  package com.itheima.ioc;
2  public class UserDaoImpl implements UserDao {
3      public void say() {
4          System.out.println("userDao say hello World !");
5      }
6  }
```

（4）在 src 目录下，创建 Spring 的配置文件 applicationContext.xml，并在配置文件中创建一个 id 为 userDao 的 Bean，如文件 1-3 所示。

文件 1-3　applicationContext.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://www.springframework.org/schema/beans
5      http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">
6      <!-- 将指定类配置给 Spring，让 Spring 创建其对象的实例 -->
7      <bean id="userDao" class="com.itheima.ioc.UserDaoImpl" />
8  </beans>
```

在文件 1-3 中，第 2~5 行代码是 Spring 的约束配置。该配置信息不需要读者手写，可以在 Spring 的帮助文档中找到（参见本节中的多学一招）。第 7 行代码表示在 Spring 容器中创建一个 id 为 userDao 的 Bean 实例，其中 class 属性用于指定需要实例化 Bean 的类。

注意

Spring 配置文件的名称可以自定义，通常在实际开发中，都会将配置文件命名为 applicationContext.xml（有时也会命名为 beans.xml）。

（5）在 com.itheima.ioc 包下，创建测试类 TestIoC，并在类中编写 main()方法。在 main()方法中，需要初始化 Spring 容器，并加载配置文件，然后通过 Spring 容器获取 userDao 实例（即 Java 对象），最后调用实例中的 say()方法，如文件 1-4 所示。

文件 1-4　TestIoC.java

```
1  package com.itheima.ioc;
2  import org.springframework.context.ApplicationContext;
3  import
4      org.springframework.context.support.ClassPathXmlApplicationContext;
5  public class TestIoC {
6      public static void main(String[] args) {
7          //1.初始化 spring 容器，加载配置文件
8          ApplicationContext applicationContext =
9              new ClassPathXmlApplicationContext("applicationContext.xml");
10         //2.通过容器获取 userDao 实例
11         UserDao userDao = (UserDao) applicationContext.getBean("userDao");
12         //3.调用实例中的 say()方法
13         userDao.say();
14     }
15 }
```

执行程序后，控制台的输出结果如图 1-5 所示。

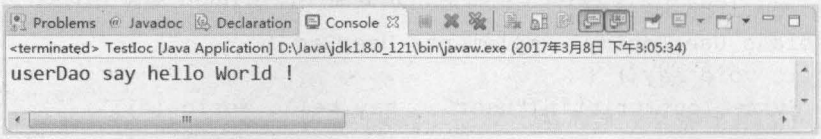
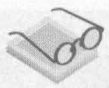


图1-5　运行结果

从图 1-5 可以看出，控制台已成功输出了 UserDaoImpl 类中的输出语句。在文件 1-4 的 main()方法中，并没有通过 new 关键字来创建 UserDao 接口的实现类对象，而是通过 Spring 容器来获取的实现类对象，这就是 Spring IoC 容器的工作机制。



多学一招：快速获取配置文件的约束信息

在 Spring 的配置文件中，包含了很多约束信息，初学者如果自己动手去编写，不但浪费时间，还容易出错。其实，在 Spring 的帮助文档中，就可以找到这些约束信息，具体的获取方法如下。

打开 Spring 解压文件夹中的 docs 目录，在 spring-framework-reference 文件夹下打开 html 文件夹，并找到 index.html 文件，如图 1-6 所示。

图1-6　Spring的参考文件目录

使用浏览器打开 index.html 后，在浏览器页面的 Overview of Spring Framework 下的 7.2.1 小节 Configuration metadata 中，即可找到配置文件的约束信息，如图 1-7 所示。

在图 1-7 中，标记处的配置信息就是 Spring 配置文件的约束信息。初学者只需将标注处的信息复制到项目的配置文件中使用即可。此外，由于使用的是 Spring 4.3 版本，所以还需要在复制后的 xsd 信息中加入版本号信息，其代码如下所示。

```
http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
```

学习本书时，建议读者先下载书中所有配套章节的源代码。在学习每一章时，如果涉及配置文件的约束信息，可以将相应章节源代码中的配置文件约束信息复制过来直接使用。

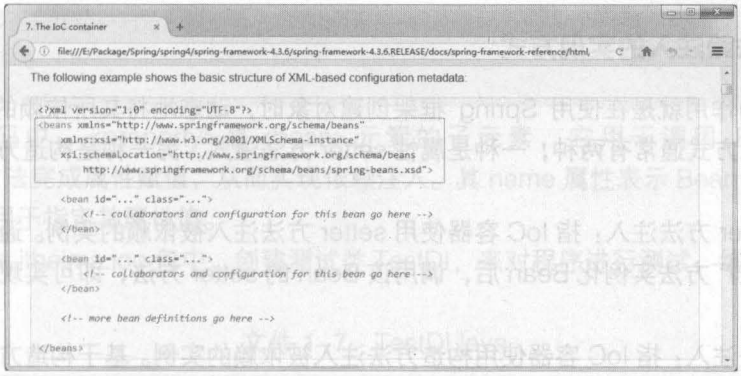


图1-7 配置文件的约束信息

1.4 依赖注入

1.4.1 依赖注入的概念

依赖注入（Dependency Injection，简称 DI）与控制反转（IoC）的含义相同，只不过这两个称呼是从两个角度描述的同一个概念。对于一个 Spring 初学者来说，这两种称呼很难理解，下面我们将通过简单的语言来描述这两个概念。

当某个 Java 对象（调用者）需要调用另一个 Java 对象（被调用者，即被依赖对象）时，在传统模式下，调用者通常会采用“new 被调用者”的代码方式来创建对象，如图 1-8 所示。这种方式会导致调用者与被调用者之间的耦合性增加，不利于后期项目的升级和维护。

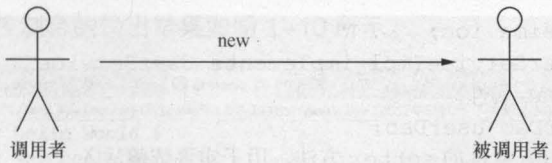


图1-8 调用者创建被调用者对象

在使用 Spring 框架之后，对象的实例不再由调用者来创建，而是由 Spring 容器来创建，Spring 容器会负责控制程序之间的关系，而不是由调用者的程序代码直接控制。这样，控制权由应用代码转移到了 Spring 容器，控制权发生了反转，这就是 Spring 的控制反转。

从 Spring 容器的角度来看，Spring 容器负责将被依赖对象赋值给调用者的成员变量，这相当于为调用者注入了它依赖的实例，这就是 Spring 的依赖注入，如图 1-9 所示。

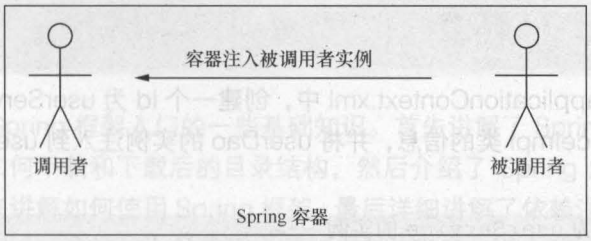


图1-9 将被调用者对象注入调用者对象

1.4.2 依赖注入的实现方式

依赖注入的作用就是在使用 Spring 框架创建对象时，动态地将其所依赖的对象注入 Bean 组件中，其实现方式通常有两种，一种是属性 setter 方法注入，另一种是构造方法注入，具体介绍如下。

- 属性 setter 方法注入：指 IoC 容器使用 setter 方法注入被依赖的实例。通过调用无参构造器或无参静态工厂方法实例化 Bean 后，调用该 Bean 的 setter 方法，即可实现基于 setter 方法的依赖注入。
- 构造方法注入：指 IoC 容器使用构造方法注入被依赖的实例。基于构造方法的依赖注入通过调用带参数的构造方法来实现，每个参数代表着一个依赖。

了解了两种注入方式后，下面以属性 setter 方法注入的方式为例，讲解一下 Spring 容器在应用中是如何实现依赖注入的。

（1）在 com.itheima.ioc 包中，创建接口 UserService，在接口中编写一个 say()方法，如文件 1-5 所示。

文件 1-5 UserService.java

```
1  package com.itheima.ioc;
2  public interface UserService {
3      public void say();
4  }
```

（2）在 com.itheima.ioc 包中，创建 UserService 接口的实现类 UserServiceImpl，在类中声明 userDao 属性，并添加属性的 setter 方法，如文件 1-6 所示。

文件 1-6 UserServiceImpl.java

```
1  package com.itheima.ioc;
2  public class UserServiceImpl implements UserService {
3      // 声明 UserDao 属性
4      private UserDao userDao;
5      // 添加 UserDao 属性的 setter 方法，用于实现依赖注入
6      public void setUserDao(UserDao userDao) {
7          this.userDao = userDao;
8      }
9      // 实现的接口中方法
10     public void say() {
11         //调用 userDao 中的 say() 方法，并执行输出语句
12         this.userDao.say();
13         System.out.println("userService say hello World !");
14     }
15 }
```

（3）在配置文件 applicationContext.xml 中，创建一个 id 为 userService 的 Bean，该 Bean 用于实例化 UserServiceImpl 类的信息，并将 userDao 的实例注入到 userService 中，其代码如下所示。

```
<!--添加一个 id 为 userService 的实例 -->
<bean id="userService" class="com.itheima.ioc.UserServiceImpl">
```

```
    <!-- 将 id 为 userDao 的 Bean 实例注入到 userService 实例中 -->
    <property name="userDao" ref="userDao" />
</bean>
```

在上述代码中，<property>是<bean>元素的子元素，它用于调用 Bean 实例中的 setUserDao()方法完成属性赋值，从而实现依赖注入。其 name 属性表示 Bean 实例中的相应属性名，ref 属性用于指定其属性值。

（4）在 com.itheima.ioc 包中，创建测试类 TestDI，来对程序进行测试，编辑后如文件 1-7 所示。

文件 1-7　TestDI.java

```
1  package com.itheima.ioc;
2  import org.springframework.context.ApplicationContext;
3  import
4      org.springframework.context.support.ClassPathXmlApplicationContext;
5  public class TestDI {
6      public static void main(String[] args) {
7          //1.初始化 spring 容器，加载配置文件
8          ApplicationContext applicationContext =
9              new ClassPathXmlApplicationContext("applicationContext.xml");
10         //2.通过容器获取 UserService 实例
11         UserService userService =
12           (UserService) applicationContext.getBean("userService");
13         //3.调用实例中的 say()方法
14         userService.say();
15     }
16 }
```

（5）执行程序后，控制台的输出结果如图 1-10 所示。

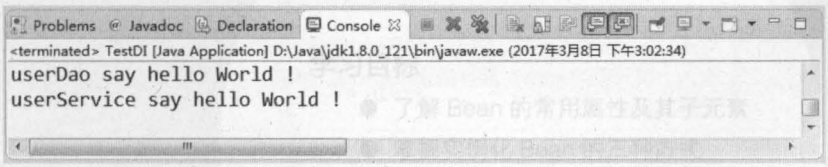


图1-10　运行结果

从图 1-10 可以看出，使用 Spring 容器通过 UserService 实现类中的 say()方法，调用了 UserDao 实现类中的 say()方法，并输出了结果。这就是 Spring 容器属性 setter 注入的方式，也是实际开发中最为常用的一种方式。

1.5 本章小结

本章主要讲解了 Spring 框架入门的一些基础知识。首先讲解了 Spring 框架的概念、作用、优点、体系结构以及如何下载和下载后的目录结构，然后介绍了 Spring 的两种核心容器。接下来通过一个入门程序来讲解如何使用 Spring 框架。最后详细讲解了依赖注入和控制反转的概念，并演示了依赖注入中 setter 方法注入的实现。通过本章的学习，读者可以对 Spring 框架及其体

系结构有一个初步的了解，能够初步地掌握 Spring 框架的使用，并能够理 Spring 框架中 IoC 和 DI 的思想，掌握属性 setter 方法注入的实现。

【思考题】

1. 请简述 Spring 框架的优点。
2. 请简述什么是 Spring 的 IoC 和 DI。

Java EE

2 Chapter

第 2 章 Spring 中的 Bean

学习目标

- 了解 Bean 的常用属性及其子元素
- 掌握实例化 Bean 的三种方式
- 熟悉 Bean 的作用域和生命周期
- 掌握 Bean 的三种装配方式

在第一章中，详细讲解了 Spring 的 IoC 思想及原理，并通过案例演示了 Spring 框架的基本使用。本章将在第一章节的基础上，针对 Spring 中 Bean 的相关知识进行详细的讲解。

2.1 Bean 的配置

Spring 可以被看作是一个大型工厂，这个工厂的作用就是生产和管理 Spring 容器中的 Bean。如果想要在项目中使用这个工厂，就需要开发者对 Spring 的配置文件进行配置。

Spring 容器支持 XML 和 Properties 两种格式的配置文件，在实际开发中，最常使用的就是 XML 格式的配置方式。这种配置方式通过 XML 文件来注册并管理 Bean 之间的依赖关系。接下来本小节将使用 XML 文件的形式对 Bean 的属性和定义进行详细的讲解。

在 Spring 中，XML 配置文件的根元素是<beans>，<beans>中包含了多个<bean>子元素，每一个<bean>子元素定义了一个 Bean，并描述了该 Bean 如何被装配到 Spring 容器中。

<bean>元素中同样包含了多个属性以及子元素，其常用属性及子元素如表 2-1 所示。

表 2-1 <bean>元素的常用属性及其子元素

属性或子元素名称	描述
id	是一个 Bean 的唯一标识符，Spring 容器对 Bean 的配置、管理都通过该属性来完成
name	Spring 容器同样可以通过此属性对容器中的 Bean 进行配置和管理，name 属性中可以为 Bean 指定多个名称，每个名称之间用逗号或分号隔开
class	该属性指定了 Bean 的具体实现类，它必须是一个完整的类名，使用类的全限定名
scope	用来设定 Bean 实例的作用域，其属性值有：singleton（单例）、prototype（原型）、request、session、global Session、application 和 websocket。其默认值为 singleton
constructor-arg	<bean>元素的子元素，可以使用此元素传入构造参数进行实例化。该元素的 index 属性指定构造参数的序号（从 0 开始），type 属性指定构造参数的类型，参数值可以通过 ref 属性或 value 属性直接指定，也可以通过 ref 或 value 子元素指定
property	<bean>元素的子元素，用于调用 Bean 实例中的 setter 方法完成属性赋值，从而完成依赖注入。该元素的 name 属性指定 Bean 实例中的相应属性名， ref 属性或 value 属性用于指定参数值
ref	<property>、<constructor-arg>等元素的属性或子元素，可以用于指定对 Bean 工厂中某个 Bean 实例的引用
value	<property>、<constructor-arg>等元素的属性或子元素，可以用于直接指定一个常量值
list	用于封装 List 或数组类型的依赖注入
set	用于封装 Set 类型属性的依赖注入
map	用于封装 Map 类型属性的依赖注入
entry	<map>元素的子元素，用于设置一个键值对。其 key 属性指定字符串类型的键值，ref 或 value 子元素指定其值，也可以通过 value-ref 或 value 属性指定其值

在配置文件中，通常一个普通的 Bean 只需要定义 id（或 name）和 class 两个属性即可，定义 Bean 的方式如下所示。

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

```
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!-- 使用 id 属性定义 bean1,其对应的实现类为 com.itheima.Bean1 -->
    <bean id="bean1" class="com.itheima.Bean1" />
    <!-- 使用 name 属性定义 bean2,其对应的实现类为 com.itheima.Bean2 -->
    <bean name="bean2" class="com.itheima.Bean2" />
</beans>
```

在上述代码中，分别使用 id 属性和 name 属性定义了两个 Bean，并使用 class 元素指定其对应的实现类。

注意

如果在 Bean 中未指定 id 和 name，则 Spring 会将 class 值当作 id 使用。

2.2 Bean 的实例化

在面向对象的程序中，想要使用某个对象，就需要先实例化这个对象。同样，在 Spring 中，要想使用容器中的 Bean，也需要实例化 Bean。实例化 Bean 有三种方式，分别为构造器实例化、静态工厂方式实例化和实例工厂方式实例化（其中最常用的是构造器实例化）。接下来的几个小节中，将分别对这三种实例化 Bean 的方式进行详细讲解。

2.2.1 构造器实例化

构造器实例化是指 Spring 容器通过 Bean 对应类中默认的无参构造方法来实例化 Bean。下面通过一个案例来演示 Spring 容器是如何通过构造器来实例化 Bean 的。

（1）在 Eclipse 中，创建一个名为 chapter02 的 Web 项目，在该项目的 lib 目录中加入 Spring 支持和依赖的 JAR 包。

（2）在 chapter02 项目的 src 目录下，创建一个 com.itheima.instance.constructor 包，在该包中创建 Bean1 类，如文件 2-1 所示。

文件 2-1 Bean1.java

```
1  package com.itheima.instance.constructor;
2  public class Bean1 {
3  }
```

（3）在 com.itheima.instance.constructor 包中，创建 Spring 的配置文件 beans1.xml，在配置文件中定义一个 id 为 bean1 的 Bean，并通过 class 属性指定其对应的实现类为 Bean1，如文件 2-2 所示。

文件 2-2 beans1.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://www.springframework.org/schema/beans
5      http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">
```

```
    <bean id="bean1" class="com.itheima.instance.constructor.Bean1" />
</beans>
```

（4）在 com.itheima.instance.constructor 包中，创建测试类 InstanceTest1，来测试构造器是否能实例化 Bean，编辑后如文件 2-3 所示。

文件 2-3　InstanceTest1.java

```
package com.itheima.instance.constructor;
import org.springframework.context.ApplicationContext;
import
    org.springframework.context.support.ClassPathXmlApplicationContext;
public class InstanceTest1 {
    public static void main(String[] args) {
        // 定义配置文件路径
        String xmlPath = "com/itheima/instance/constructor/beans1.xml";
        // ApplicationContext 在加载配置文件时，对 Bean 进行实例化
        ApplicationContext applicationContext =
                                new ClassPathXmlApplicationContext(xmlPath);
        Bean1 bean = (Bean1) applicationContext.getBean("bean1");
        System.out.println(bean);
    }
}
```

在文件 2-3 中，首先定义了配置文件的路径，然后 Spring 容器 ApplicationContext 会加载配置文件。在加载时，Spring 容器会通过 id 为 bean1 的实现类 Bean1 中默认的无参构造方法对 Bean 进行实例化。执行程序后，控制台的输出结果如图 2-1 所示。

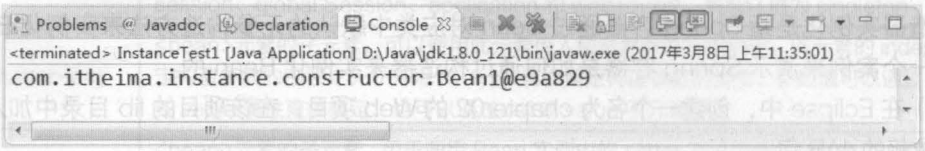


图2-1　运行结果

从图 2-1 可以看出，Spring 容器已经成功实例化了 Bean1，并输出了结果。

为了方便读者的学习，本章中的所有配置文件和类文件（包括测试类）都根据知识点放置在同一个包中。在实际开发中，为了方便管理和维护，建议将这些文件根据类别放置在不同目录中。

2.2.2　静态工厂方式实例化

使用静态工厂是实例化 Bean 的另一种方式。该方式要求开发者创建一个静态工厂的方法来创建 Bean 的实例，其 Bean 配置中的 class 属性所指定的不再是 Bean 实例的实现类，而是静态工厂类，同时还需要使用 factory-method 属性来指定所创建的静态工厂方法。下面通过一个案例来演示如何使用静态工厂方式实例化 Bean。

（1）在 chapter02 项目的 src 目录下，创建一个 com.itheima.instance.static_factory 包，在该包中创建一个 Bean2 类，该类与 Bean1 一样，不需添加任何方法。

（2）在 com.itheima.instance.static_factory 包中，创建一个 MyBean2Factory 类，并在类中创建一个静态方法 createBean()来返回 Bean2 实例，如文件 2-4 所示。

文件 2-4 MyBean2Factory.java

```
1  package com.itheima.instance.static_factory;
2  public class MyBean2Factory {
3      //使用自己的工厂创建 Bean2 实例
4      public static Bean2 createBean(){
5          return new Bean2();
6      }
7  }
```

(3) 在 com.itheima.instance.static_factory 包中，创建 Spring 配置文件 beans2.xml，编辑后如文件 2-5 所示。

文件 2-5 beans2.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://www.springframework.org/schema/beans
5      http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">
6      <bean id="bean2"
7            class="com.itheima.instance.static_factory.MyBean2Factory"
8            factory-method="createBean" />
9  </beans>
```

在上述配置文件中，首先通过<bean>元素的 id 属性定义了一个名称为 bean2 的 Bean，然后由于使用的是静态工厂方法，所以需要通过 class 属性指定其对应的工厂实现类为 MyBean2Factory。由于这种方式配置 Bean 后，Spring 容器不知道哪个是所需要的工厂方法，所以增加了 factory-method 属性来告诉 Spring 容器，其方法名称为 createBean。

(4) 在 com.itheima.instance.static_factory 包中，创建一个测试类 InstanceTest2，来测试使用静态工厂方式是否能实例化 Bean，编辑后如文件 2-6 所示。

文件 2-6 InstanceTest2.java

```
1  package com.itheima.instance.static_factory;
2  import org.springframework.context.ApplicationContext;
3  import
4      org.springframework.context.support.ClassPathXmlApplicationContext;
5  public class InstanceTest2 {
6      public static void main(String[] args) {
7          // 定义配置文件路径
8          String xmlPath =
9                  "com/itheima/instance/static_factory/beans2.xml";
10         // ApplicationContext 在加载配置文件时，对 Bean 进行实例化
11         ApplicationContext applicationContext =
12                 new ClassPathXmlApplicationContext(xmlPath);
13         System.out.println(applicationContext.getBean("bean2"));
14     }
15 }
```

执行程序后，控制台的输出结果如图 2-2 所示。

从图 2-2 可以看到，使用自定义的静态工厂方法，已成功实例化了 Bean2。

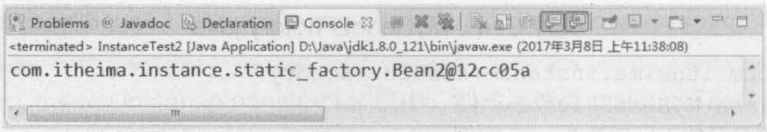


图2-2　运行结果

2.2.3　实例工厂方式实例化

还有一种实例化 Bean 的方式就是采用实例工厂。此种方式的工厂类中，不再使用静态方法创建 Bean 实例，而是采用直接创建 Bean 实例的方式。同时，在配置文件中，需要实例化的 Bean 也不是通过 class 属性直接指向的实例化类，而是通过 factory-bean 属性指向配置的实例工厂，然后使用 factory-method 属性确定使用工厂中的哪个方法。下面通过一个案例来演示实例工厂方式的使用。

（1）在 chapter02 项目的 src 目录下，创建一个 com.itheima.instance.factory 包，在该包中创建 Bean3 类，该类与 Bean1 一样，不需添加任何方法。

（2）在 com.itheima.instance.factory 包中，创建工厂类 MyBean3Factory，在类中使用默认无参构造方法输出“bean3 工厂实例化中”语句，并使用 createBean()方法创建 Bean3 对象，如文件 2-7 所示。

文件 2-7　MyBean3Factory.java

```
1  package com.itheima.instance.factory;
2  public class MyBean3Factory {
3      public MyBean3Factory() {
4          System.out.println("bean3 工厂实例化中");
5      }
6      //创建 Bean3 实例的方法
7      public Bean3 createBean(){
8          return new Bean3();
9      }
10 }
```

（3）在 com.itheima.instance.factory 包中，创建 Spring 配置文件 beans3.xml，设置相关配置后，如文件 2-8 所示。

文件 2-8　beans3.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://www.springframework.org/schema/beans
5      http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">
6      <!-- 配置工厂 -->
7      <bean id="myBean3Factory"
8              class="com.itheima.instance.factory.MyBean3Factory" />
9      <!-- 使用 factory-bean 属性指向配置的实例工厂，
10         使用 factory-method 属性确定使用工厂中的哪个方法-->
11     <bean id="bean3" factory-bean="myBean3Factory"
12             factory-method="createBean" />
13 </beans>
```

在上述配置文件中，首先配置了一个工厂 Bean，然后配置了需要实例化的 Bean。在 id 为 bean3 的 Bean 中，使用 factory-bean 属性指向配置的实例工厂，该属性值就是工厂 Bean 的 id。使用 factory-method 属性来确定使用工厂中的 createBean()方法。

（4）在 com.itheima.instance.factory 的包中，创建测试类 InstanceTest3，来测试实例工厂方式能否实例化 Bean，编辑后如文件 2-9 所示。

文件 2-9　InstanceTest3.java

```
1  package com.itheima.instance.factory;
2  import org.springframework.context.ApplicationContext;
3  import
4      org.springframework.context.support.ClassPathXmlApplicationContext;
5  public class InstanceTest3 {
6      public static void main(String[] args) {
7          // 指定配置文件路径
8          String xmlPath = "com/itheima/instance/factory/beans3.xml";
9          // ApplicationContext 在加载配置文件时，对 Bean 进行实例化
10         ApplicationContext applicationContext =
11                 new ClassPathXmlApplicationContext(xmlPath);
12         System.out.println(applicationContext.getBean("bean3"));
13     }
14 }
```

执行程序后，控制台的输出结果如图 2-3 所示。

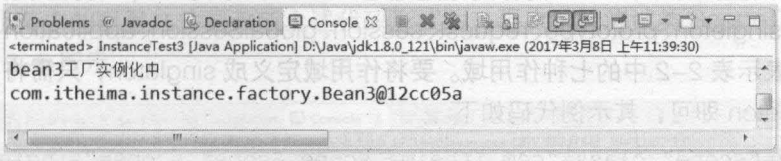


图2-3　运行结果

从图 2-3 可以看到，使用实例工厂的方式，同样成功实例化了 Bean3。

2.3 Bean 的作用域

通过 Spring 容器创建一个 Bean 的实例时，不仅可以完成 Bean 的实例化，还可以为 Bean 指定特定的作用域。本节将主要围绕 Bean 的作用域知识进行讲解。

2.3.1 作用域的种类

Spring 4.3 中为 Bean 的实例定义了 7 种作用域，这 7 种作用域及其说明如表 2-2 所示。

表 2-2　Bean 的作用域

作用域名称	说明
singleton（单例）	使用 singleton 定义的 Bean 在 Spring 容器中将只有一个实例，也就是说，无论有多少个 Bean 引用它，始终将指向同一个对象。这也是 Spring 容器默认的作用域
prototype（原型）	每次通过 Spring 容器获取的 prototype 定义的 Bean 时，容器都将创建一个新的 Bean 实例

续表

作用域名称	说明
request	在一次 HTTP 请求中，容器会返回该 Bean 的同一个实例。对不同的 HTTP 请求则会产生一个新的 Bean，而且该 Bean 仅在当前 HTTP Request 内有效
session	在一次 HTTP Session 中，容器会返回该 Bean 的同一个实例。对不同的 HTTP 请求则会产生一个新的 Bean，而且该 Bean 仅在当前 HTTP Session 内有效
globalSession	在一个全局的 HTTP Session 中，容器会返回该 Bean 的同一个实例。仅在使用 portlet 上下文时有效
application	为每个 ServletContext 对象创建一个实例。仅在 Web 相关的 ApplicationContext 中生效
websocket	为每个 websocket 对象创建一个实例。仅在 Web 相关的 ApplicationContext 中生效

在表 2-2 的 7 种作用域中，singleton 和 prototype 是最常用的两种，在接下来的两个小节中，将会对这两种作用域进行详细的讲解。

2.3.2 singleton 作用域

singleton 是 Spring 容器默认的作用域，当 Bean 的作用域为 singleton 时，Spring 容器就只会存在一个共享的 Bean 实例，并且所有对 Bean 的请求，只要 id 与该 Bean 的 id 属性相匹配，就会返回同一个 Bean 实例。singleton 作用域对于无会话状态的 Bean（如 Dao 组件、Service 组件）来说，是最理想的选择。

在 Spring 配置文件中，Bean 的作用域是通过<bean>元素的 scope 属性来指定的，该属性值可以设置为 singleton、prototype、request、session、globalSession、application 和 websocket 七个值，分别表示表 2-2 中的七种作用域。要将作用域定义成 singleton，只需将 scope 的属性值设置为 singleton 即可，其示例代码如下。

```
<bean id="scope" class="com.itheima.scope.Scope" scope="singleton"/>
```

在项目 chapter02 中，创建一个 com.itheima.scope 包，在包中创建 Scope 类，该类不需要写任何方法。然后在该包中创建一个配置文件 beans4.xml，将上述示例代码写入配置文件中。最后在包中创建测试类 ScopeTest，来测试 singleton 作用域，编辑后如文件 2-10 所示。

文件 2-10　ScopeTest.java

```
1  package com.itheima.scope;
2  import org.springframework.context.ApplicationContext;
3  import
4      org.springframework.context.support.ClassPathXmlApplicationContext;
5  public class ScopeTest {
6      public static void main(String[] args) {
7          // 定义配置文件路径
8          String xmlPath = "com/itheima/scope/beans4.xml";
9          // 加载配置文件
10         ApplicationContext applicationContext =
11                     new ClassPathXmlApplicationContext(xmlPath);
12         // 输出获得实例
13         System.out.println(applicationContext.getBean("scope"));
14         System.out.println(applicationContext.getBean("scope"));
```

```
15    }
16 }
```

执行程序后，控制台的输出结果如图 2-4 所示。

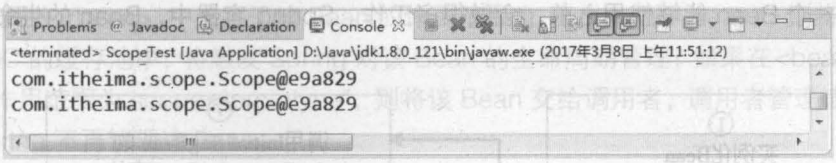


图2-4 运行结果

从图 2-4 可以看出，两次输出的结果相同，这说明 Spring 容器只创建了一个 Scope 类的实例。

需要注意的是，如果不设置 scope="singleton"，其输出结果也是一个实例，因为 Spring 容器默认的作用域就是 singleton。

2.3.3 prototype 作用域

对需要保持会话状态的 Bean（如 Struts2 的 Action 类）应该使用 prototype 作用域。在使用 prototype 作用域时，Spring 容器会为每个对该 Bean 的请求都创建一个新的实例。

要将 Bean 定义为 prototype 作用域，只需在配置文件中将<bean>元素的 scope 属性值设置为 prototype 即可，其示例代码如下。

```
<bean id="scope" class="com.itheima.scope.Scope" scope="prototype" />
```

将 2.3.2 小节中的配置文件更改成上述代码形式后，再次运行测试类 ScopeTest，控制台的输出结果如图 2-5 所示。

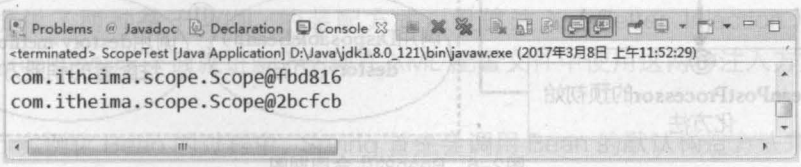


图2-5 运行结果

从图 2-5 可以看到，两次输出的 Bean 实例并不相同，这说明在 prototype 作用域下，创建了两个不同的 Scope 实例。

2.4 Bean 的生命周期

Spring 容器可以管理 singleton 作用域的 Bean 的生命周期，在此作用域下，Spring 能够精确地知道该 Bean 何时被创建，何时初始化完成以及何时被销毁。对于 prototype 作用域的 Bean，Spring 只负责创建，当容器创建了 Bean 实例后，Bean 的实例就交给客户端代码来管理，Spring 容器将不再跟踪其生命周期。每次客户端请求 prototype 作用域的 Bean 时，Spring 容器都会创建一个新的实例，并且不会管那些被配置成 prototype 作用域的 Bean 的生命周期。

了解 Bean 的生命周期的意义就在于，可以在某个 Bean 生命周期的某些指定时刻完成一些相关操作。这种时刻可能有很多，但在一般情况下，常会在 Bean 的 postinitiation（初始化后）

和 predestruction（销毁前）执行一些相关操作。

在 Spring 中，Bean 生命周期的执行是一个很复杂的过程，读者可以利用 Spring 提供的方法来定制 Bean 的创建过程。当一个 Bean 被加载到 Spring 容器时，它就具有了生命，而 Spring 容器在保证一个 Bean 能够使用之前，会做很多工作。Spring 容器中，Bean 的生命周期流程如图 2-6 所示。

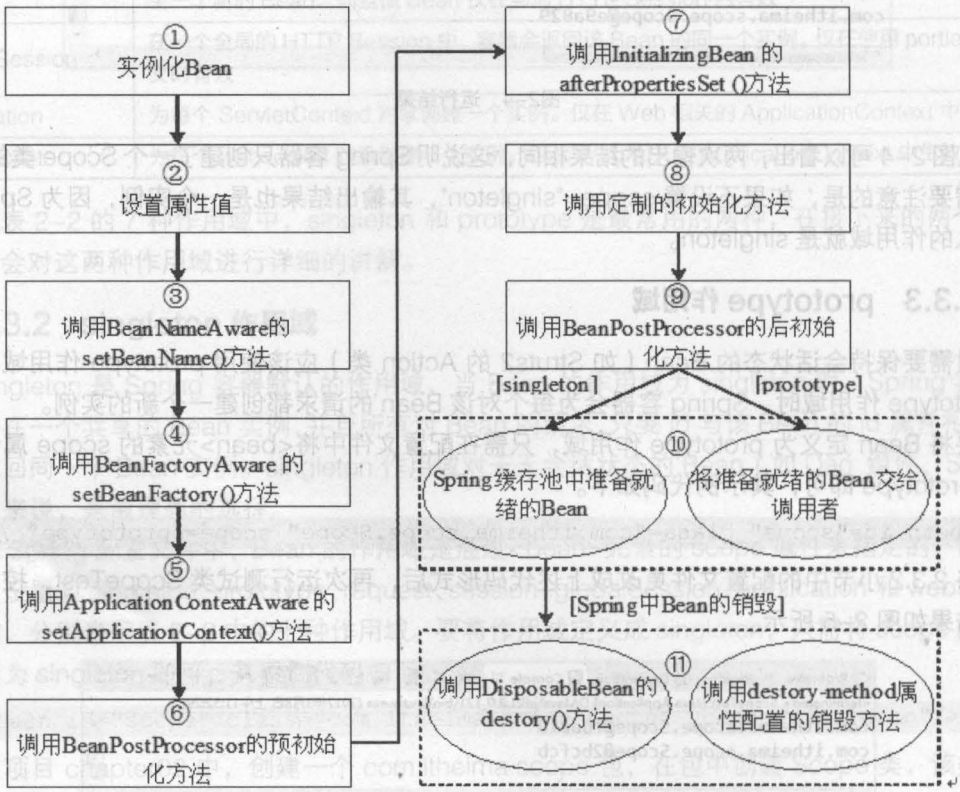


图2-6　Bean的生命周期图

在图 2-6 中，Bean 的生命周期的整个执行过程描述如下。

（1）根据配置情况调用 Bean 构造方法或工厂方法实例化 Bean。

（2）利用依赖注入完成 Bean 中所有属性值的配置注入。

（3）如果 Bean 实现了 BeanNameAware 接口，则 Spring 调用 Bean 的 setBeanName()方法传入当前 Bean 的 id 值。

（4）如果 Bean 实现了 BeanFactoryAware 接口，则 Spring 调用 setBeanFactory()方法传入当前工厂实例的引用。

（5）如果 Bean 实现了 ApplicationContextAware 接口，则 Spring 调用 setApplicationContext()方法传入当前 ApplicationContext 实例的引用。

（6）如果 BeanPostProcessor 和 Bean 关联，则 Spring 将调用该接口的预初始化方法 postProcessBeforeInitialzation()对 Bean 进行加工操作，这个非常重要，Spring 的 AOP 就是用它实现的。

（7）如果 Bean 实现了 InitializingBean 接口，则 Spring 将调用 afterPropertiesSet()方法。

（8）如果在配置文件中通过 init-method 属性指定了初始化方法，则调用该初始化方法。

（9）如果有 BeanPostProcessor 和 Bean 关联，则 Spring 将调用该接口的初始化方法 postProcessAfterInitialization()。此时，Bean 已经可以被应用系统使用了。

（10）如果在<bean> 中指定了该 Bean 的作用范围为 scope="singleton"，则将该 Bean 放入 Spring IoC 的缓存池中，将触发 Spring 对该 Bean 的生命周期管理；如果在<bean>中指定了该 Bean 的作用范围为 scope="prototype"，则将该 Bean 交给调用者，调用者管理该 Bean 的生命周期，Spring 不再管理该 Bean。

（11）如果 Bean 实现了 DisposableBean 接口，则 Spring 会调用 destory()方法将 Spring 中的 Bean 销毁；如果在配置文件中通过 destory-method 属性指定了 Bean 的销毁方法，则 Spring 将调用该方法进行销毁。

Spring 为 Bean 提供了细致全面的生命周期过程，通过实现特定的接口或通过<bean>的属性设置，都可以对 Bean 的生命周期过程产生影响。我们可以随意地配置<bean>的属性，但是在这里建议不要过多地使用 Bean 实现接口，因为这样会使代码和 Spring 聚合比较紧密。

2.5 Bean 的装配方式

Bean 的装配可以理解为依赖关系注入，Bean 的装配方式即 Bean 依赖注入的方式。Spring 容器支持多种形式的 Bean 的装配方式，如基于 XML 的装配、基于注解（Annotation）的装配和自动装配等（其中最常用的是基于注解的装配）。本节将主要讲解这三种装配方式的使用。

2.5.1 基于 XML 的装配

Spring 提供了两种基于 XML 的装配方式：设值注入（Setter Injection）和构造注入（Constructor Injection）。下面就讲解下如何在 XML 配置文件中使用这两种注入方式来实现基于 XML 的装配。

在 Spring 实例化 Bean 的过程中，Spring 首先会调用 Bean 的默认构造方法来实例化 Bean 对象，然后通过反射的方式调用 setter 方法来注入属性值。因此，设值注入要求一个 Bean 必须满足以下两点要求。

- Bean 类必须提供一个默认的无参构造方法。
- Bean 类必须为需要注入的属性提供对应的 setter 方法。

使用设值注入时，在 Spring 配置文件中，需要使用<bean>元素的子元素<property>来为每个属性注入值；而使用构造注入时，在配置文件里，需要使用<bean>元素的子元素<constructor-arg>来定义构造方法的参数，可以使用其 value 属性（或子元素）来设置该参数的值。下面通过一个案例来演示基于 XML 方式的 Bean 的装配。

（1）在项目 chapter02 的 src 目录下，创建一个 com.itheima.assemble 包，在该包中创建 User 类，并在类中定义 username、password 和 list 集合三个属性及其对应的 setter 方法，如文件 2-11 所示。

文件 2-11 User.java

```
1  package com.itheima.assemble;
2  import java.util.List;
```

```
public class User {
    private String username;
    private Integer password;
    private List<String> list;
    /**
     * 1.使用构造注入
     * 1.1 提供带所有参数的有参构造方法。
     */
    public User(String username, Integer password, List<String> list) {
        super();
        this.username = username;
        this.password = password;
        this.list = list;
    }
    /**
     * 2.使用设值注入
     * 2.1 提供默认空参构造方法;
     * 2.2 为所有属性提供 setter 方法。
     */
    public User() {
        super();
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public void setPassword(Integer password) {
        this.password = password;
    }
    public void setList(List<String> list) {
        this.list = list;
    }
    @Override
    public String toString() {
        return "User [username=" + username + ", password=" + password +
                ", list=" + list + "]";
    }
}
```

在文件 2-11 中，由于要使用构造注入，所以需要其有参和无参的构造方法。同时，为了输出时能够看到结果，还重写了其属性的 toString()方法。

（2）在 com.itheima.assemble 包中，创建配置文件 beans5.xml，在配置文件中通过构造注入和设值注入的方式装配 User 类的实例，如文件 2-12 所示。

文件 2-12　beans5.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">
```

```
    <!--1.使用构造注入方式装配 User 实例 -->
    <bean id="user1" class="com.itheima.assemble.User">
        <constructor-arg index="0" value="tom" />
        <constructor-arg index="1" value="123456" />
        <constructor-arg index="2">
            <list>
                <value>"constructorvalue1"</value>
                <value>"constructorvalue2"</value>
            </list>
        </constructor-arg>
    </bean>
    <!--2.使用设值注入方式装配 User 实例 -->
    <bean id="user2" class="com.itheima.assemble.User">
        <property name="username" value="张三"></property>
        <property name="password" value="654321"></property>
        <!-- 注入 list 集合 -->
        <property name="list">
            <list>
                <value>"setlistvalue1"</value>
                <value>"setlistvalue2"</value>
            </list>
        </property>
    </bean>
</beans>
```

在上述配置文件中，<constructor-arg >元素用于定义构造方法的参数，其属性 index 表示其索引（从 0 开始），value 属性用于设置注入的值，其子元素<list>来为 User 类中对应的 list 集合属性注入值。然后又使用了设值注入方式装配 User 类的实例，其中<property>元素用于调用 Bean 实例中的 setter 方法完成属性赋值，从而完成依赖注入，而其子元素<list>同样是为 User 类中对应的 list 集合属性注入值。

（3）在 com.itheima.assemble 包中，创建测试类 XmlBeanAssembleTest，在类中分别获取并输出配置文件中的 user1 和 user2 实例，如文件 2-13 所示。

文件 2-13　XmlBeanAssembleTest.java

```
package com.itheima.assemble;
import org.springframework.context.ApplicationContext;
import
    org.springframework.context.support.ClassPathXmlApplicationContext;
public class XmlBeanAssembleTest {
    public static void main(String[] args) {
        // 定义配置文件路径
        String xmlPath = "com/itheima/assemble/beans5.xml";
        // 加载配置文件
        ApplicationContext applicationContext =
                        new ClassPathXmlApplicationContext(xmlPath);
        // 构造方式输出结果
        System.out.println(applicationContext.getBean("user1"));
        // 设值方式输出结果
        System.out.println(applicationContext.getBean("user2"));
```

```
16      }
17  }
```

执行程序后，控制台的输出结果如图 2-7 所示。

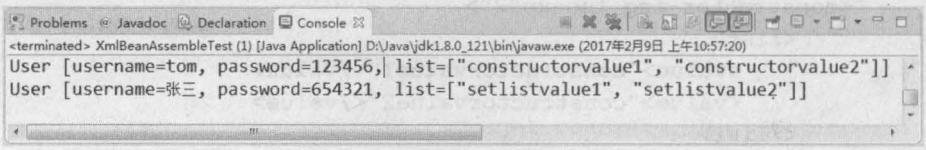


图2-7　运行结果

从图 2-7 可以看出，已经成功地使用基于 XML 装配的构造注入和设值注入两种方式装配了 User 实例。

2.5.2　基于 Annotation 的装配

在 Spring 中，尽管使用 XML 配置文件可以实现 Bean 的装配工作，但如果应用中有很多 Bean 时，会导致 XML 配置文件过于臃肿，给后续的维护和升级工作带来一定的困难。为此，Spring 提供了对 Annotation（注解）技术的全面支持。

Spring 中定义了一系列的注解，常用的注解如下所示。

- @Component：可以使用此注解描述 Spring 中的 Bean，但它是一个泛化的概念，仅仅表示一个组件（Bean），并且可以作用在任何层次。使用时只需将该注解标注在相应类上即可。
- @Repository：用于将数据访问层（DAO 层）的类标识为 Spring 中的 Bean，其功能与 @Component 相同。
- @Service：通常作用在业务层（Service 层），用于将业务层的类标识为 Spring 中的 Bean，其功能与@Component 相同。
- @Controller：通常作用在控制层（如 Spring MVC 的 Controller），用于将控制层的类标识为 Spring 中的 Bean，其功能与@Component 相同。
- @Autowired：用于对 Bean 的属性变量、属性的 setter 方法及构造方法进行标注，配合对应的注解处理器完成 Bean 的自动配置工作。默认按照 Bean 的类型进行装配。
- @Resource：其作用与 Autowired 一样。其区别在于@Autowired 默认按照 Bean 类型装配，而@Resource 默认按照 Bean 实例名称进行装配。@Resource 中有两个重要属性：name 和 type。Spring 将 name 属性解析为 Bean 实例名称，type 属性解析为 Bean 实例类型。如果指定 name 属性，则按实例名称进行装配；如果指定 type 属性，则按 Bean 类型进行装配；如果都不指定，则先按 Bean 实例名称装配，如果不能匹配，再按照 Bean 类型进行装配；如果都无法匹配，则抛出 NoSuchBeanDefinitionException 异常。
- @Qualifier：与@Autowired 注解配合使用，会将默认的按 Bean 类型装配修改为按 Bean 的实例名称装配，Bean 的实例名称由@Qualifier 注解的参数指定。

在上面几个注解中，虽然@Repository、@Service 与@Controller 功能与@Component 注解的功能相同，但为了使标注类本身用途更加清晰，建议在实际开发中使用@Repository、@Service 与@Controller 分别对实现类进行标注。

下面，通过一个案例来演示如何通过这些注解来装配 Bean。

（1）在chapter02项目的src目录下，创建一个com.itheima.annotation包，在该包中创建接口UserDao，并在接口中定义一个save()方法，如文件2-14所示。

文件2-14 UserDao.java

```
1  package com.itheima.annotation;
2  public interface UserDao {
3      public void save();
4  }
```

（2）在com.itheima.annotation包中，创建UserDao接口的实现类UserDaoImpl，该类需要实现接口中的save()方法，如文件2-15所示。

文件2-15 UserDaoImpl.java

```
1  package com.itheima.annotation;
2  import org.springframework.stereotype.Repository;
3  @Repository("userDao")
4  public class UserDaoImpl implements UserDao{
5      public void save(){
6          System.out.println("userdao...save...");
7      }
8  }
```

在文件2-15中，首先使用@Repository注解将UserDaoImpl类标识为Spring中的Bean，其写法相当于配置文件中<bean id="userDao" class="com.itheima.annotation.UserDaoImpl"/>的编写。然后在save()方法中输出打印一句话，用于验证是否成功调用了该方法。

（3）在com.itheima.annotation包中，创建接口UserService，在接口中同样定义一个save()方法，如文件2-16所示。

文件2-16 UserService.java

```
1  package com.itheima.annotation;
2  public interface UserService {
3      public void save();
4  }
```

（4）在com.itheima.annotation包中，创建UserService接口的实现类UserServiceImpl，该类需要实现接口中的save()方法，如文件2-17所示。

文件2-17 UserServiceImpl.java

```
1  package com.itheima.annotation;
2  import javax.annotation.Resource;
3  import org.springframework.stereotype.Service;
4  @Service("userService")
5  public class UserServiceImpl implements UserService{
6      @Resource(name="userDao")
7      private UserDao userDao;
8      public void save() {
9          //调用userDao中的save方法
10         this.userDao.save();
11         System.out.println("userservice....save...");
```

```
12      }
13  }
```

在文件 2-17 中，首先使用@Service 注解将 UserServiceImpl 类标识为 Spring 中的 Bean，这相当于配置文件中<bean id="userService" class="com.itheima.annotation.UserServiceImpl" />的编写；然后使用@Resource 注解标注在属性 userDao 上，这相当于配置文件中<property name="userDao" ref="userDao"/>的写法；最后在该类的 save()方法中调用 userDao 中的 save()方法，并输出一句话。

（5）在 com.itheima.annotation 包中，创建控制器类 UserController，编辑后如文件 2-18 所示。

文件 2-18　UserController.java

```
1  package com.itheima.annotation;
2  import javax.annotation.Resource;
3  import org.springframework.stereotype.Controller;
4  @Controller("userController")
5  public class UserController {
6      @Resource(name="userService")
7      private UserService userService;
8      public void save(){
9          this.userService.save();
10         System.out.println("userController...save...");
11     }
12 }
```

在文件 2-18 中，首先使用@Controller 注解标注了 UserController 类，这相当于在配置文件中编写<bean id="userController" class="com.itheima.annotation.UserController"/>；然后使用了@Resource 注解标注在 userService 属性上，这相当于在配置文件中编写<property name="userService" ref="userService" />；最后在其 save()方法中调用了 userService 中的 save()方法，并输出一句话。

（6）在 com.itheima.annotation 包中，创建配置文件 beans6.xml，在配置文件中编写基于 Annotation 装配的代码，如文件 2-19 所示。

文件 2-19　beans6.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xmlns:context="http://www.springframework.org/schema/context"
5       xsi:schemaLocation="http://www.springframework.org/schema/beans
6       http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
7       http://www.springframework.org/schema/context
8    http://www.springframework.org/schema/context/spring-context-4.3.xsd">
9       <!-- 使用 context 命名空间 ,在配置文件中开启相应的注解处理器 -->
10      <context:annotation-config />
11      <!--分别定义 3 个 Bean 实例  -->
12      <bean id="userDao" class="com.itheima.annotation.UserDaoImpl" />
13      <bean id="userService"
14          class="com.itheima.annotation.UserServiceImpl" />
```

```
15      <bean id="userController"
16          class="com.itheima.annotation.UserController" />
17 </beans>
```

从上述代码可以看出，文件 2-19 与之前的配置文件有很大不同。首先，在<beans>元素中，增加了第 4 行，第 7 行和第 8 行中包含有 context 的约束信息；然后通过配置<context:annotation-config />来开启注解处理器；最后分别定义了 3 个 Bean 对应所编写的 3 个实例。与 XML 装备方式有所不同的是，这里不再需要配置子元素<property>。

上述 Spring 配置文件中的注解方式虽然较大程度简化了 XML 文件中 Bean 的配置，但仍需要在 Spring 配置文件中一一配置相应的 Bean，为此 Spring 注解提供了另外一种高效的注解配置方式（对包路径下的所有 Bean 文件进行扫描），其配置方式如下。

```
<context:component-scan base-package="Bean 所在的包路径"/>
```

所以可以将上述文件 2-19 中第 9～16 行代码进行如下替换（推荐）。

```
<!--使用 context 命名空间 ,通知 Spring 扫描指定包下所有 Bean 类，进行注解解析-->
<context:component-scan base-package="com.itheima.annotation" />
```

注意

Spring 4.0 以上版本使用上面的代码对指定包中的注解进行扫描前，需要先向项目中导入 Spring AOP 的 JAR 包 spring-aop-4.3.6.RELEASE.jar，否则程序在运行时会报出“java.lang.NoClassDefFoundError: org/springframework/aop/TargetSource”错误。

（7）在 com.itheima.annotation 包中，创建测试类 AnnotationAssembleTest，在类中编写测试方法并定义配置文件的路径，然后通过 Spring 容器加载配置文件并获取 UserController 实例，最后调用实例中的 save()方法，如文件 2-20 所示。

文件 2-20 AnnotationAssembleTest.java

```
1  package com.itheima.annotation;
2  import org.springframework.context.ApplicationContext;
3  import
4     org.springframework.context.support.ClassPathXmlApplicationContext;
5  public class AnnotationAssembleTest {
6      public static void main(String[] args) {
7          // 定义配置文件路径
8          String xmlPath = "com/itheima/annotation/beans6.xml";
9          // 加载配置文件
10         ApplicationContext applicationContext =
11                 new ClassPathXmlApplicationContext(xmlPath);
12         // 获取 UserController 实例
13         UserController userController =
14           (UserController) applicationContext.getBean("userController");
15         // 调用 UserController 中的 save()方法
16         userController.save();
17     }
18 }
```

执行程序后，控制台的输出结果如图 2-8 所示。

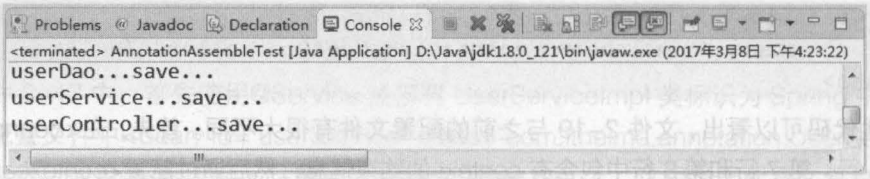


图2-8 运行结果

从图 2-8 可以看到，Spring 容器已成功获取了 UserController 的实例，并通过调用实例中的方法执行了各层中的输出语句，这说明已成功实现了基于 Annotation 装配 Bean。

小提示

上述案例中如果使用@Autowired 注解替换@Resource 注解，也可以达到同样的效果。

2.5.3 自动装配

虽然使用注解的方式装配 Bean，在一定程度上减少了配置文件中的代码量，但是也有企业项目中，是没有使用注解方式开发的，那么有没有什么办法既可以减少代码量，又能够实现 Bean 的装配呢?

答案是肯定的，Spring 的<bean>元素中包含一个 autowire 属性，我们可以通过设置 autowire 的属性值来自动装配 Bean。所谓自动装配，就是将一个 Bean 自动地注入到其他 Bean 的 Property 中。

autowire 属性有 5 个值，其值及说明如表 2-3 所示。

表 2-3 <bean>元素的 autowire 属性值及说明

属性值	说明
default（默认值）	由<bean>的上级标签<beans>的 default-autowire 属性值确定。例如<beans default-autowire="byName">，则该<bean>元素中的 autowire 属性对应的属性值就为 byName
byName	根据属性的名称自动装配。容器将根据名称查找与属性完全一致的 Bean，并将其属性自动装配
byType	根据属性的数据类型（Type）自动装配，如果一个 Bean 的数据类型，兼容另一个 Bean 中属性的数据类型，则自动装配
constructor	根据构造函数参数的数据类型，进行 byType 模式的自动装配
no	在默认情况下，不使用自动装配，Bean 依赖必须通过 ref 元素定义

下面通过修改 2.5.2 节中的案例来演示如何使用自动装配。

（1）修改 2.5.2 节中的文件 2-17 UserServiceImpl 和文件 2-18 UserController，分别在文件中增加类属性的 setter 方法。

（2）修改 2.5.2 节中的配置文件 beans6.xml，将配置文件修改成自动装配形式，如文件 2-21 所示。

文件 2-21 beans6.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xmlns:context="http://www.springframework.org/schema/context"
5         xsi:schemaLocation="http://www.springframework.org/schema/beans
6         http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
```

```
7        http://www.springframework.org/schema/context
8    http://www.springframework.org/schema/context/spring-context-4.3.xsd">
9      <!-- 使用 bean 元素的 autowire 属性完成自动装配 -->
10     <bean id="userDao"
11         class="com.itheima.annotation.UserDaoImpl" />
12     <bean id="userService"
13       class="com.itheima.annotation.UserServiceImpl" autowire="byName" />
14     <bean id="userController"
15       class="com.itheima.annotation.UserController" autowire="byName"/>
16 </beans>
```

上述配置文件中，用于配置 userService 和 userController 的<bean>元素中除了 id 和 class 属性外，还增加了 autowire 属性，并将其属性值设置为 byName。在默认情况下，配置文件中需要通过 ref 来装配 Bean，但设置了 autowire="byName"后，Spring 会自动寻找 userService Bean 中的属性，并将其属性名称与配置文件中定义的 Bean 做匹配。由于 UserServiceImpl 中定义了 userDao 属性及其 setter 方法，这与配置文件中 id 为 userDao 的 Bean 相匹配，所以 Spring 会自动地将 id 为 userDao 的 Bean 装配到 id 为 userService 的 Bean 中。

执行程序后，控制台的输出结果如图 2-9 所示。

图2-9 运行结果

从图 2-9 可以看出，使用自动装配同样完成了依赖注入。

2.6 本章小结

本章主要对 Spring 中的 Bean 进行了详细讲解。首先介绍了 Bean 的配置，然后通过案例讲解了 Bean 实例化的三种方式；接下来介绍了 Bean 的作用域和生命周期；最后讲解了 Bean 的三种装配方式。通过本章的学习，读者可以了解 Bean 的常用属性及其作用，可以掌握实例化 Bean 的三种方式，熟悉 Bean 作用域的种类及其生命周期，掌握 Bean 的三种装配方式。

【思考题】

1. 请简述 Bean 的生命周期。
2. 请简述 Bean 的几种装配方式的基本用法。

Java EE

3 Chapter

第3章 Spring AOP

学习目标

- 了解 AOP 的概念和作用
- 理解 AOP 中的相关术语
- 了解 Spring 中两种动态代理方式的区别
- 掌握基于代理类的 AOP 实现
- 掌握基于 XML 和注解的 AspectJ 开发

Spring 的 AOP 模块，是 Spring 框架体系结构中十分重要的内容，该模块中提供了面向切面编程实现。本章将对 Spring AOP 的相关知识进行详细讲解。

3.1 Spring AOP 简介

3.1.1 什么是 AOP

AOP 的全称是 Aspect-Oriented Programming，即面向切面编程（也称面向方面编程）。它是面向对象编程（OOP）的一种补充，目前已成为一种比较成熟的编程方式。

在传统的业务处理代码中，通常都会进行事务处理、日志记录等操作。虽然使用 OOP 可以通过组合或者继承的方式来达到代码的重用，但如果要实现某个功能（如日志记录），同样的代码仍然会分散到各个方法中。这样，如果想要关闭某个功能，或者对其进行修改，就必须要修改所有的相关方法。这不但增加了开发人员的工作量，而且提高了代码的出错率。

为了解决这一问题，AOP 思想随之产生。AOP 采取横向抽取机制，将分散在各个方法中的重复代码提取出来，然后在程序编译或运行时，再将这些提取出来的代码应用到需要执行的地方。这种采用横向抽取机制的方式，采用传统的 OOP 思想显然是无法办到的，因为 OOP 只能实现父子关系的纵向的重用。虽然 AOP 是一种新的编程思想，但却不是 OOP 的替代品，它只是 OOP 的延伸和补充。

在 AOP 思想中，类与切面的关系如图 3-1 所示。

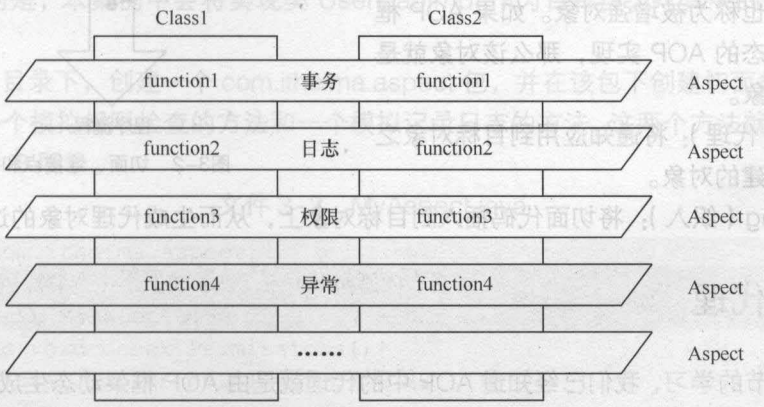


图3-1 类与切面的关系

从图 3-1 可以看出，通过 Aspect（切面）分别在 Class1 和 Class2 的方法中加入了事务、日志、权限和异常等功能。

AOP 的使用，使开发人员在编写业务逻辑时可以专心于核心业务，而不用过多地关注于其他业务逻辑的实现，这不但提高了开发效率，而且增强了代码的可维护性。

目前最流行的 AOP 框架有两个，分别为 Spring AOP 和 AspectJ。Spring AOP 使用纯 Java 实现，不需要专门的编译过程和类加载器，在运行期间通过代理方式向目标类织入增强的代码。AspectJ 是一个基于 Java 语言的 AOP 框架，从 Spring 2.0 开始，Spring AOP 引入了对 AspectJ 的支持，AspectJ 扩展了 Java 语言，提供了一个专门的编译器，在编译时提供横向代码的织入。

3.1.2 AOP 术语

在学习使用 AOP 之前，首先要了解一下 AOP 的专业术语。这些术语包括 Aspect、Joinpoint、Pointcut、Advice、Target Object、Proxy 和 Weaving，对于这些专业术语的解释，具体如下。

- Aspect（切面）：在实际应用中，切面通常是指封装的用于横向插入系统功能（如事务、日志等）的类，如图 3-1 中的 Aspect。该类要被 Spring 容器识别为切面，需要在配置文件中通过<bean>元素指定。
- Joinpoint（连接点）：在程序执行过程中的某个阶段点，它实际上是对象的一个操作，例如方法的调用或异常的抛出。在 Spring AOP 中，连接点就是指方法的调用。
- Pointcut（切入点）：是指切面与程序流程的交叉点，即那些需要处理的连接点，如图 3-2 所示。通常在程序中，切入点指的是类或者方法名，如某个通知要应用到所有以 add 开头的方法中，那么所有满足这一规则的方法都是切入点。
- Advice（通知/增强处理）：AOP 框架在特定的切入点执行的增强处理，即在定义好的切入点处所要执行的程序代码。可以将其理解为切面类中的方法，它是切面的具体实现。
- Target Object（目标对象）：是指所有被通知的对象，也称为被增强对象。如果 AOP 框架采用的是动态的 AOP 实现，那么该对象就是一个被代理对象。
- Proxy（代理）：将通知应用到目标对象之后，被动态创建的对象。

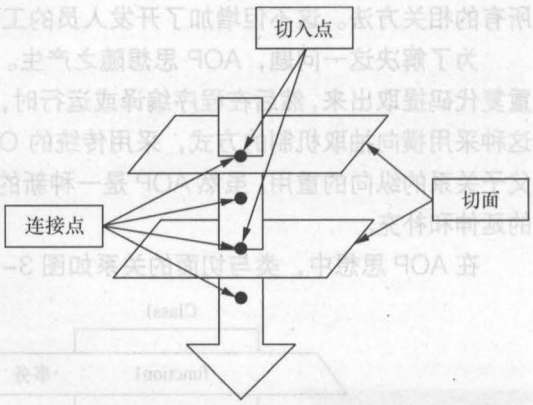


图3-2 切面、连接点和切入点

- Weaving（织入）：将切面代码插入到目标对象上，从而生成代理对象的过程。

3.2 动态代理

通过 3.1 节的学习，我们已经知道 AOP 中的代理就是由 AOP 框架动态生成的一个对象，该对象可以作为目标对象使用。Spring 中的 AOP 代理，可以是 JDK 动态代理，也可以是 CGLIB 代理。接下来的两个小节中，将结合相关案例，来演示这两种代理方式的使用。

3.2.1 JDK 动态代理

JDK 动态代理是通过 java.lang.reflect.Proxy 类来实现的，我们可以调用 Proxy 类的 newProxyInstance()方法来创建代理对象。对于使用业务接口的类，Spring 默认会使用 JDK 动态代理来实现 AOP。

接下来，通过一个案例来演示 Spring 中 JDK 动态代理的实现过程，具体步骤如下。

（1）创建一个名为 chapter03 的 Web 项目，导入 Spring 框架所需 JAR 包到项目的 lib 目录中，并发布到类路径下。

（2）在 src 目录下，创建一个 com.itheima.jdk 包，在该包下创建接口 UserDao，并在该接口中编写添加和删除的方法，如文件 3-1 所示。

文件 3-1 UserDao.java

```
1  package com.itheima.jdk;
2  public interface UserDao {
3      public void addUser();
4      public void deleteUser();
5  }
```

（3）在 com.itheima.jdk 包中，创建 UserDao 接口的实现类 UserDaoImpl，分别实现接口中的方法，并在每个方法中添加一条输出语句，如文件 3-2 所示。

文件 3-2 UserDaoImpl.java

```
1  package com.itheima.jdk;
2  // 目标类
3  public class UserDaoImpl implements UserDao {
4      public void addUser() {
5          System.out.println("添加用户");
6      }
7      public void deleteUser() {
8          System.out.println("删除用户");
9      }
10 }
```

需要注意的是，本案例中会将实现类 UserDaoImpl 作为目标类，对其中的方法进行增强处理。

（4）在 src 目录下，创建一个 com.itheima.aspect 包，并在该包下创建切面类 MyAspect，在该类中定义一个模拟权限检查的方法和一个模拟记录日志的方法，这两个方法就表示切面中的通知，如文件 3-3 所示。

文件 3-3 MyAspect.java

```
1  package com.itheima.aspect;
2  //切面类：可以存在多个通知 Advice（即增强的方法）
3  public class MyAspect {
4      public void check_Permissions(){
5          System.out.println("模拟检查权限...");
6      }
7      public void log(){
8          System.out.println("模拟记录日志...");
9      }
10 }
```

（5）在 com.itheima.jdk 包下，创建代理类 JdkProxy，该类需要实现 InvocationHandler 接口，并编写代理方法。在代理方法中，需要通过 Proxy 类实现动态代理，如文件 3-4 所示。

文件 3-4 JdkProxy.java

```
1  package com.itheima.jdk;
2  import java.lang.reflect.InvocationHandler;
3  import java.lang.reflect.Method;
```

```
import java.lang.reflect.Proxy;
import com.itheima.aspect.MyAspect;
/**
 * JDK 代理类
 */
public class JdkProxy implements InvocationHandler{
    // 声明目标类接口
    private UserDao userDao;
    // 创建代理方法
    public  Object createProxy(UserDao userDao) {
        this.userDao = userDao;
        // 1.类加载器
        ClassLoader classLoader = JdkProxy.class.getClassLoader();
        // 2.被代理对象实现的所有接口
        Class[] clazz = userDao.getClass().getInterfaces();
        // 3.使用代理类，进行增强，返回的是代理后的对象
        return  Proxy.newProxyInstance(classLoader,clazz,this);
    }
    /*
     * 所有动态代理类的方法调用，都会交由 invoke()方法去处理
     * proxy 被代理后的对象
     * method 将要被执行的方法信息（反射）
     * args 执行方法时需要的参数
     */
    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
                                                        throws Throwable {
        // 声明切面
        MyAspect myAspect = new MyAspect();
        // 前增强
        myAspect.check_Permissions();
        // 在目标类上调用方法，并传入参数
        Object obj = method.invoke(userDao, args);
        // 后增强
        myAspect.log();
        return obj;
    }
}
```

在文件 3-4 中，JdkProxy 类实现了 InvocationHandler 接口，并实现了接口中的 invoke()方法，所有动态代理类所调用的方法都会交由该方法处理。在创建的代理方法 createProxy()中，使用了 Proxy 类的 newProxyInstance()方法来创建代理对象。newProxyInstance()方法中包含 3 个参数，其中第 1 个参数是当前类的类加载器，第 2 个参数表示的是被代理对象实现的所有接口，第 3 个参数 this 代表的就是代理类 JdkProxy 本身。在 invoke()方法中，目标类方法执行的前后，会分别执行切面类中的 check_Permissions()方法和 log()方法。

（6）在 com.itheima.jdk 包中，创建测试类 JdkTest。在该类中的 main()方法中创建代理对象和目标对象，然后从代理对象中获得对目标对象 userDao 增强后的对象，最后调用该对象中的添加和删除方法，如文件 3-5 所示。

文件 3-5 JdkTest.java

```
1  package com.itheima.jdk;
2  public class JdkTest{
3      public static void main(String[] args) {
4          // 创建代理对象
5          JdkProxy jdkProxy = new JdkProxy();
6          // 创建目标对象
7          UserDao userDao= new UserDaoImpl();
8          // 从代理对象中获取增强后的目标对象
9          UserDao userDao1 = (UserDao) jdkProxy.createProxy(userDao);
10         // 执行方法
11         userDao1.addUser();
12         userDao1.deleteUser();
13     }
14 }
```

执行程序后，控制台的输出结果如图 3-3 所示。

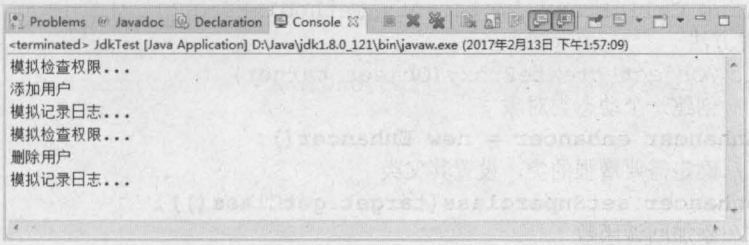


图3-3 运行结果

从图 3-3 可以看出，userDao 实例中的添加用户和删除用户的方法已被成功调用，并且在调用前后分别增加了检查权限和记录日志的功能。这种实现了接口的代理方式，就是 Spring 中的 JDK 动态代理。

3.2.2 CGLIB 代理

JDK 动态代理的使用非常简单，但它还有一定的局限性——使用动态代理的对象必须实现一个或多个接口。如果要对没有实现接口的类进行代理，那么可以使用 CGLIB 代理。

CGLIB (Code Generation Library) 是一个高性能开源的代码生成包，它采用非常底层的字节码技术，对指定的目标类生成一个子类，并对子类进行增强。在 Spring 的核心包中已经集成了 CGLIB 所需要的包，所以开发中不需要另外导入 JAR 包。

接下来，通过一个案例来演示 CGLIB 代理的实现过程，具体步骤如下。

(1) 在 src 目录下，创建一个 com.itheima.cglib 包，在包中创建一个目标类 UserDao，UserDao 不需要实现任何接口，只需定义一个添加用户的方法和一个删除用户的方法，如文件 3-6 所示。

文件 3-6 UserDao.java

```
1  package com.itheima.cglib;
2  //目标类
3  public class UserDao {
4      public void addUser() {
```

```
        System.out.println("添加用户");
    }
    public void deleteUser() {
        System.out.println("删除用户");
    }
}
```

（2）在 com.itheima.cglib 包中，创建代理类 CglibProxy，该代理类需要实现 MethodInterceptor 接口，并实现接口中的 intercept()方法，如文件 3-7 所示。

文件 3-7　CglibProxy.java

```
package com.itheima.cglib;
import java.lang.reflect.Method;
import org.springframework.cglib.proxy.Enhancer;
import org.springframework.cglib.proxy.MethodInterceptor;
import org.springframework.cglib.proxy.MethodProxy;
import com.itheima.aspect.MyAspect;
// 代理类
public class CglibProxy implements MethodInterceptor{
    // 代理方法
    public  Object createProxy(Object target) {
        // 创建一个动态类对象
        Enhancer enhancer = new Enhancer();
        // 确定需要增强的类，设置其父类
        enhancer.setSuperclass(target.getClass());
        // 添加回调函数
        enhancer.setCallback(this);
        // 返回创建的代理类
        return enhancer.create();
    }
    /**
     * proxy CGlib 根据指定父类生成的代理对象
     * method 拦截的方法
     * args 拦截方法的参数数组
     * methodProxy 方法的代理对象，用于执行父类的方法
     */
    @Override
    public Object intercept(Object proxy, Method method, Object[] args,
                            MethodProxy methodProxy) throws Throwable {
        // 创建切面类对象
        MyAspect myAspect = new MyAspect();
        // 前增强
        myAspect.check_Permissions();
        // 目标方法执行
        Object obj = methodProxy.invokeSuper(proxy, args);
        // 后增强
        myAspect.log();
        return obj;
    }
}
```

在文件3-7的代理方法中，首先创建了一个动态类对象Enhancer，它是CGLIB的核心类；然后调用了Enhancer类的setSuperclass()方法来确定目标对象；接下来调用了setCallback()方法添加回调函数，其中的this代表的就是代理类CglibProxy本身；最后通过return语句将创建的代理类对象返回。intercept()方法会在程序执行目标方法时被调用，方法运行时将会执行切面类中的增强方法。

（3）在com.itheima.cglib包中，创建测试类CglibTest。在该类的main()方法中首先创建代理对象和目标对象，然后从代理对象中获得增强后的目标对象，最后调用对象的添加和删除方法，如文件3-8所示。

文件3-8 CglibTest.java

```
1  package com.itheima.cglib;
2  // 测试类
3  public class CglibTest {
4      public static void main(String[] args) {
5          // 创建代理对象
6          CglibProxy cglibProxy = new CglibProxy();
7          // 创建目标对象
8          UserDao userDao = new UserDao();
9          // 获取增强后的目标对象
10         UserDao userDao1 = (UserDao)cglibProxy.createProxy(userDao);
11         // 执行方法
12         userDao1.addUser();
13         userDao1.deleteUser();
14     }
15 }
```

执行程序后，控制台的输出结果如图3-4所示。

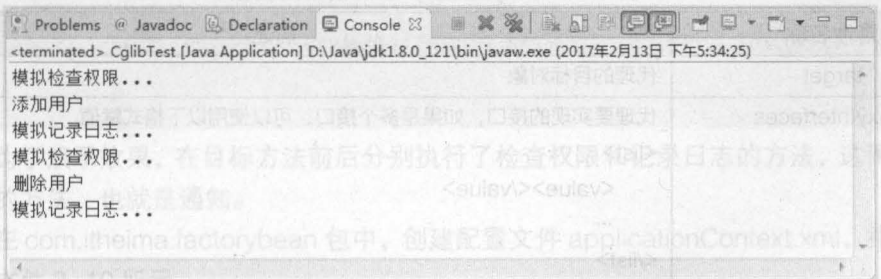


图3-4 运行结果

从图3-4可以看出，目标类UserDao中的方法被成功调用并增强了。这种没有实现接口的代理方式，就是CGLIB代理。

3.3 基于代理类的AOP实现

通过3.2小节的学习，读者对Spring中的两种代理模式已经有了一定的了解。实际上，Spring中的AOP代理默认就是使用JDK动态代理的方式来实现的。在Spring中，使用ProxyFactoryBean是创建AOP代理的最基本方式。接下来的两个小节中，将对Spring中基于代理类的AOP实现的相关知识进行详细讲解。

3.3.1 Spring 的通知类型

在讲解具体的代理类之前，我们需要先了解一下 Spring 的通知类型。Spring 中的通知按照在目标类方法的连接点位置，可以分为以下 5 种类型。

- org.aopalliance.intercept.MethodInterceptor（环绕通知）

在目标方法执行前后实施增强，可以应用于日志、事务管理等功能。

- org.springframework.aop.MethodBeforeAdvice（前置通知）

在目标方法执行前实施增强，可以应用于权限管理等功能。

- org.springframework.aop.AfterReturningAdvice（后置通知）

在目标方法执行后实施增强，可以应用于关闭流、上传文件、删除临时文件等功能。

- org.springframework.aop.ThrowsAdvice（异常通知）

在方法抛出异常后实施增强，可以应用于处理异常记录日志等功能。

- org.springframework.aop.IntroductionInterceptor（引介通知）

在目标类中添加一些新的方法和属性，可以应用于修改老版本程序（增强类）。

3.3.2 ProxyFactoryBean

ProxyFactoryBean 是 FactoryBean 接口的实现类，FactoryBean 负责实例化一个 Bean，而 ProxyFactoryBean 负责为其他 Bean 创建代理实例。在 Spring 中，使用 ProxyFactoryBean 是创建 AOP 代理的基本方式。

ProxyFactoryBean 类中的常用可配置属性如表 3-1 所示。

表 3-1 ProxyFactoryBean 的常用属性

属性名称	描述
target	代理的目标对象
proxyInterfaces	代理要实现的接口，如果是多个接口，可以使用以下格式赋值 <list> <value></value> ... </list>
proxyTargetClass	是否对类代理而不是接口，设置为 true 时，使用 CGLIB 代理
interceptorNames	需要织入目标的 Advice
singleton	返回的代理是否为单实例，默认为 true（即返回单实例）
optimize	当设置为 true 时，强制使用 CGLIB

对 ProxyFactoryBean 类有了初步的了解后，接下来通过一个典型的环绕通知案例，来演示 Spring 使用 ProxyFactoryBean 创建 AOP 代理的过程，具体步骤如下。

（1）在核心 JAR 包的基础上，再向 chapter03 项目的 lib 目录中导入 AOP 的 JAR 包 spring-aop-4.3.6.RELEASE.jar 和 aopalliance-1.0.jar，如图 3-5 所示。

关于这两个 JAR 包的介绍如下。

- spring-aop-4.3.6.RELEASE.jar：是 Spring 为 AOP 提供的实现包，Spring 的包中已经

提供。

• aopalliance-1.0.jar：是 AOP 联盟提供的规范包，该 JAR 包可以通过地址“http://mvnrepository.com/artifact/aopalliance/aopalliance/1.0”下载。

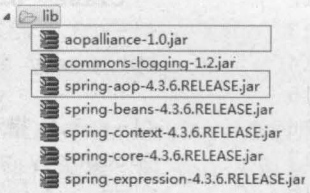


图3-5 添加的JAR包

（2）在 src 目录下，创建一个 com.itheima.factorybean 包，在该包中创建切面类 MyAspect。由于实现环绕通知需要实现 org.aopalliance.intercept. MethodInterceptor 接口，所以 MyAspect 类需要实现该接口，并实现接口中的 invoke()方法，来执行目标方法，如文件 3-9 所示。

文件 3-9 MyAspect.java

```
1  package com.itheima.factorybean;
2  import org.aopalliance.intercept.MethodInterceptor;
3  import org.aopalliance.intercept.MethodInvocation;
4  // 切面类
5  public class MyAspect implements MethodInterceptor {
6      @Override
7      public Object invoke(MethodInvocation mi) throws Throwable {
8          check_Permissions();
9          // 执行目标方法
10         Object obj = mi.proceed();
11         log();
12         return obj;
13     }
14     public void check_Permissions(){
15         System.out.println("模拟检查权限...");
16     }
17     public void log(){
18         System.out.println("模拟记录日志...");
19     }
20 }
```

这里为了演示效果，在目标方法前后分别执行了检查权限和记录日志的方法，这两个方法也就是增强的方法，也就是通知。

（3）在 com.itheima.factorybean 包中，创建配置文件 applicationContext.xml，并指定代理对象，如文件 3-10 所示。

文件 3-10 applicationContext.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://www.springframework.org/schema/beans
5      http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">
6      <!-- 1 目标类 -->
7      <bean id="userDao" class="com.itheima.jdk.UserDaoImpl" />
8      <!-- 2 切面类 -->
9      <bean id="myAspect" class="com.itheima.factorybean.MyAspect" />
10     <!-- 3 使用 Spring 代理工厂定义一个名称为 userDaoProxy 的代理对象 -->
11     <bean id="userDaoProxy"
```

```
12          class="org.springframework.aop.framework.ProxyFactoryBean">
13        <!-- 3.1 指定代理实现的接口-->
14        <property name="proxyInterfaces"
15                  value="com.itheima.jdk.UserDao" />
16        <!-- 3.2 指定目标对象 -->
17        <property name="target" ref="userDao" />
18        <!-- 3.3 指定切面,植入环绕通知 -->
19        <property name="interceptorNames" value="myAspect" />
20        <!-- 3.4 指定代理方式，true：使用 cglib，false(默认)：使用 jdk 动态代理 -->
21        <property name="proxyTargetClass" value="true" />
22    </bean>
23 </beans>
```

在文件 3-10 中，首先通过<bean>元素定义了目标类和切面，然后使用 ProxyFactoryBean 类定义了代理对象。在定义的代理对象中，分别通过<property>子元素指定了代理实现的接口、代理的目标对象、需要织入目标类的通知以及代理方式。

（4）在 com.itheima.factorybean 包中，创建测试类 ProxyFactoryBeanTest，在类中通过 Spring 容器获取代理对象的实例，并执行目标方法，如文件 3-11 所示。

文件 3-11　ProxyFactoryBeanTest.java

```
1  package com.itheima.factorybean;
2  import org.springframework.context.ApplicationContext;
3  import
4     org.springframework.context.support.ClassPathXmlApplicationContext;
5  import com.itheima.jdk.UserDao;
6  // 测试类
7  public class ProxyFactoryBeanTest {
8      public static void main(String args[]) {
9        String xmlPath = "com/itheima/factorybean/applicationContext.xml";
10       ApplicationContext applicationContext =
11                              new ClassPathXmlApplicationContext(xmlPath);
12       // 从 spring 容器获得内容
13       UserDao userDao =
14                          (UserDao) applicationContext.getBean("userDaoProxy");
15       // 执行方法
16       userDao.addUser();
17       userDao.deleteUser();
18     }
19 }
```

执行程序后，控制台的输出结果如图 3-6 所示。

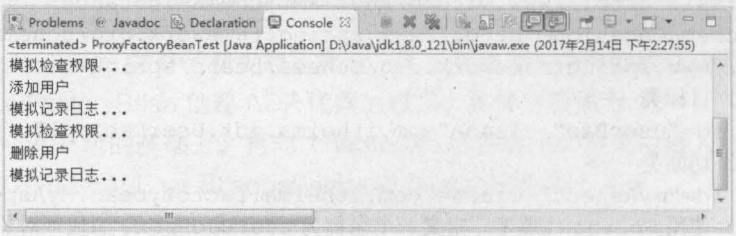


图3-6　运行结果

3.4 AspectJ 开发

AspectJ 是一个基于 Java 语言的 AOP 框架，它提供了强大的 AOP 功能。Spring 2.0 以后，Spring AOP 引入了对 AspectJ 的支持，并允许直接使用 AspectJ 进行编程，而 Spring 自身的 AOP API 也尽量与 AspectJ 保持一致。新版本的 Spring 框架，也建议使用 AspectJ 来开发 AOP。使用 AspectJ 实现 AOP 有两种方式：一种是基于 XML 的声明式 AspectJ，另一种是基于注解的声明式 AspectJ。接下来的两个小节中，将对这两种 AspectJ 的开发方式进行讲解。

3.4.1 基于 XML 的声明式 AspectJ

基于 XML 的声明式 AspectJ 是指通过 XML 文件来定义切面、切入点及通知，所有的切面、切入点和通知都必须定义在<aop:config>元素内。<aop:config>元素及其子元素如图 3-7 所示。

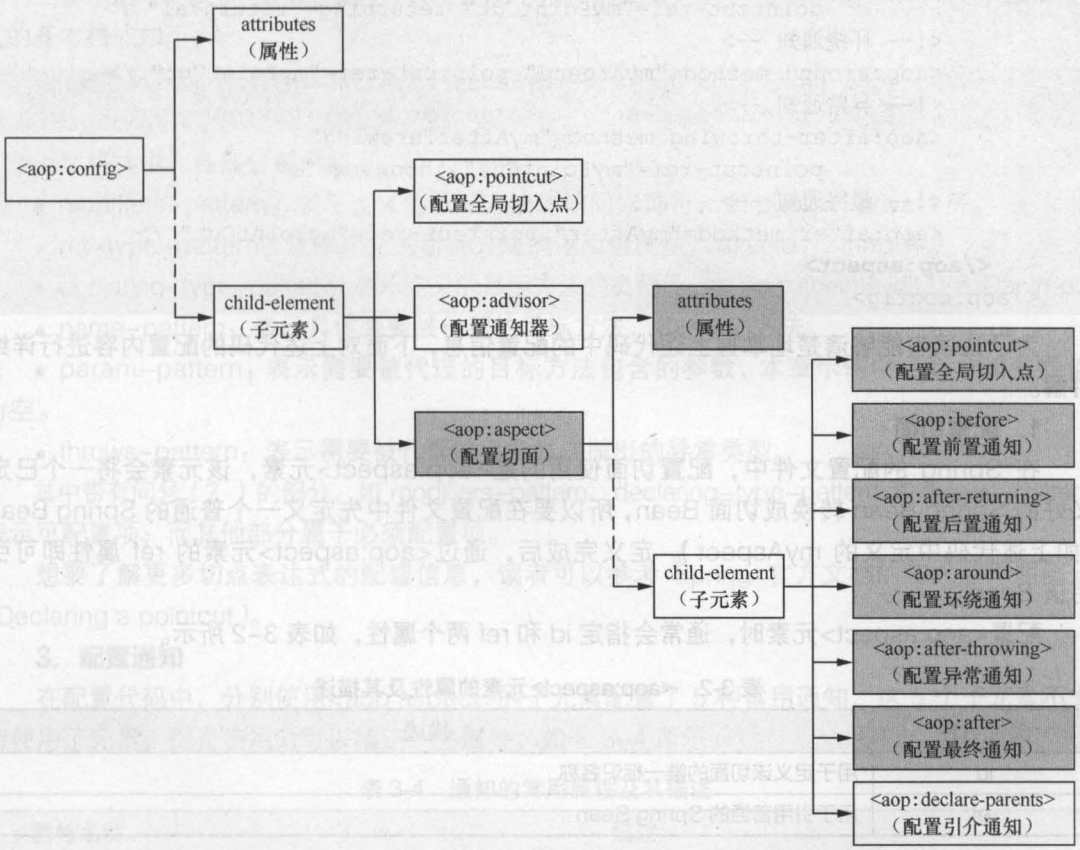


图3-7 <aop:config>元素及其子元素

在图 3-7 中，Spring 配置文件中的<beans>元素下可以包含多个<aop:config>元素，一个<aop:config>元素中又可以包含属性和子元素，其子元素包括<aop:pointcut>、<aop:advisor>和<aop:aspect> 。在配置时，这 3 个子元素必须按照此顺序来定义。在<aop:aspect>元素下，同样包含了属性和多个子元素，通过使用<aop:aspect>元素及其子元素就可以在 XML 文件中配

置切面、切入点和通知。图中灰色部分标注的元素即为常用的配置元素，这些常用元素的配置代码如下所示。

```
<!-- 定义切面 Bean -->
<bean id="myAspect" class="com.itheima.aspectj.xml.MyAspect" />
<aop:config>
    <!--1.配置切面 -->
    <aop:aspect  id="aspect"  ref="myAspect">
        <!--  2.配置切入点 -->
        <aop:pointcut expression="execution(* com.itheima.jdk.*.*(..))"
                                                id="myPointCut" />
        <!-- 3.配置通知 -->
        <!-- 前置通知 -->
        <aop:before method="myBefore" pointcut-ref="myPointCut" />
        <!-- 后置通知 -->
        <aop:after-returning method="myAfterReturning"
                pointcut-ref="myPointCut" returning="returnVal" />
        <!-- 环绕通知 -->
        <aop:around method="myAround" pointcut-ref="myPointCut" />
        <!-- 异常通知 -->
        <aop:after-throwing method="myAfterThrowing"
                pointcut-ref="myPointCut" throwing="e" />
        <!-- 最终通知 -->
        <aop:after method="myAfter" pointcut-ref="myPointCut" />
    </aop:aspect>
</aop:config>
```

为了让读者能够清楚地掌握上述代码中的配置信息，下面对上述代码的配置内容进行详细讲解。

1. 配置切面

在 Spring 的配置文件中，配置切面使用的是<aop:aspect>元素，该元素会将一个已定义好的 Spring Bean 转换成切面 Bean，所以要在配置文件中先定义一个普通的 Spring Bean（如上述代码中定义的 myAspect）。定义完成后，通过<aop:aspect>元素的 ref 属性即可引用该 Bean。

配置<aop:aspect>元素时，通常会指定 id 和 ref 两个属性，如表 3-2 所示。

表 3-2　<aop:aspect>元素的属性及其描述

属性名称	描述
id	用于定义该切面的唯一标识名称
ref	用于引用普通的 Spring Bean

2. 配置切入点

在 Spring 的配置文件中，切入点是通过<aop:pointcut>元素来定义的。当<aop:pointcut>元素作为<aop:config>元素的子元素定义时，表示该切入点是全局切入点，它可被多个切面所共享；当<aop:pointcut>元素作为<aop:aspect>元素的子元素时，表示该切入点只对当前切面有效。

在定义<aop:pointcut>元素时，通常会指定 id 和 expression 两个属性，如表 3-3 所示。

表 3-3 <aop:pointcut>元素的属性及其描述

属性名称	描述
id	用于指定切入点的唯一标识名称
expression	用于指定切入点关联的切入点表达式

在上述配置代码片段中，execution(* com.itheima.jdk.*.*(..))就是定义的切入点表达式，该切入点表达式的意思是匹配 com.itheima.jdk 包中任意类的任意方法的执行。其中 execution()是表达式的主体，第 1 个*表示的是返回类型，使用*代表所有类型；com.itheima.jdk 表示的是需要拦截的包名，后面第 2 个*表示的是类名，使用*代表所有的类；第 3 个*表示的是方法名，使用*表示所有方法；后面(..)表示方法的参数，其中的“..”表示任意参数。需要注意的是，第 1 个*与包名之间有一个空格。

上面示例中定义的切入点表达式只是开发中常用的配置方式，而 Spring AOP 中切入点表达式的基本格式如下：

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?
        name-pattern(param-pattern) throws-pattern?)
```

上述格式中，各部分说明如下。

- modifiers-pattern：表示定义的目标方法的访问修饰符，如 public、private 等。
- ret-type-pattern：表示定义的目标方法的返回值类型，如 void、String 等。
- declaring-type-pattern：表示定义的目标方法的类路径，如 com.itheima.jdk.UserDaoImpl。
- name-pattern：表示具体需要被代理的目标方法，如 add()方法。
- param-pattern：表示需要被代理的目标方法包含的参数，本章示例中目标方法参数都为空。
- throws-pattern：表示需要被代理的目标方法抛出的异常类型。

其中带有问号（？）的部分，如 modifiers-pattern、declaring-type-pattern 和 throws-pattern 表示可配置项；而其他部分属于必须配置项。

想要了解更多切点表达式的配置信息，读者可以参考 Spring 官方文档的切入点声明部分（Declaring a pointcut）。

3. 配置通知

在配置代码中，分别使用<aop:aspect>的子元素配置了 5 种常用通知，这 5 个子元素不支持使用子元素，但在使用时可以指定一些属性，如表 3-4 所示。

表 3-4 通知的常用属性及其描述

属性名称	描述
pointcut	该属性用于指定一个切入点表达式，Spring 将在匹配该表达式的连接点时织入该通知
pointcut-ref	该属性指定一个已经存在的切入点名称，如配置代码中的 myPointCut。通常 pointcut 和 pointcut-ref 两个属性只需要使用其中之一
method	该属性指定一个方法名，指定将切面 Bean 中的该方法转换为增强处理
throwing	该属性只对<after-throwing>元素有效，它用于指定一个形参名，异常通知方法可以通过该形参访问目标方法所抛出的异常

续表

属性名称	描述
returning	该属性只对<after-returning>元素有效，它用于指定一个形参名，后置通知方法可以通过该形参访问目标方法的返回值

了解了如何在 XML 中配置切面、切入点和通知后，接下来通过一个案例来演示如何在 Spring 中使用基于 XML 的声明式 AspectJ，具体实现步骤如下。

（1）导入 AspectJ 框架相关的 JAR 包，具体如下。

- spring-aspects-4.3.6.RELEASE.jar：Spring 为 AspectJ 提供的实现，Spring 的包中已经提供。
- aspectjweaver-1.8.10.jar：是 AspectJ 框架所提供的规范，读者可以通过网址“http://mvnrepository.com/artifact/org.aspectj/aspectjweaver/1.8.10”下载。

（2）在 chapter03 项目的 src 目录下，创建一个 com.itheima.aspectj.xml 包，在该包中创建切面类 MyAspect，并在类中分别定义不同类型的通知，如文件 3-12 所示。

文件 3-12　MyAspect.java

```
1  package com.itheima.aspectj.xml;
2  import org.aspectj.lang.JoinPoint;
3  import org.aspectj.lang.ProceedingJoinPoint;
4  /**
5   *切面类，在此类中编写通知
6   */
7  public class MyAspect {
8      // 前置通知
9      public void myBefore(JoinPoint joinPoint) {
10         System.out.print("前置通知 : 模拟执行权限检查...,");
11         System.out.print("目标类是: "+joinPoint.getTarget() );
12         System.out.println(",被植入增强处理的目标方法为: "
13                            +joinPoint.getSignature().getName());
14     }
15     // 后置通知
16     public void myAfterReturning(JoinPoint joinPoint) {
17         System.out.print("后置通知: 模拟记录日志...," );
18         System.out.println("被植入增强处理的目标方法为: "
19                         + joinPoint.getSignature().getName());
20     }
21     /**
22      * 环绕通知
23      * ProceedingJoinPoint 是 JoinPoint 子接口，表示可以执行目标方法
24      * 1.必须是 Object 类型的返回值
25      * 2.必须接收一个参数，类型为 ProceedingJoinPoint
26      * 3.必须 throws Throwable
27      */
28     public Object myAround(ProceedingJoinPoint proceedingJoinPoint)
29         throws Throwable {
30         // 开始
31         System.out.println("环绕开始: 执行目标方法之前，模拟开启事务...");
```

```
32          // 执行当前目标方法
33          Object obj = proceedingJoinPoint.proceed();
34          // 结束
35          System.out.println("环绕结束：执行目标方法之后，模拟关闭事务...");
36          return obj;
37      }
38      // 异常通知
39      public void myAfterThrowing(JoinPoint joinPoint, Throwable e) {
40          System.out.println("异常通知：" + "出错了" + e.getMessage());
41      }
42      // 最终通知
43      public void myAfter() {
44          System.out.println("最终通知：模拟方法结束后的释放资源...");
45      }
46  }
```

在文件 3-12 中，分别定义了 5 种不同类型的通知，在通知中使用了 JoinPoint 接口及其子接口 ProceedingJoinPoint 作为参数来获得目标对象的类名、目标方法名和目标方法参数等。

需要注意的是，环绕通知必须接收一个类型为 ProceedingJoinPoint 的参数，返回值也必须是 Object 类型，且必须抛出异常。异常通知中可以传入 Throwable 类型的参数来输出异常信息。

（3）在 com.itheima.aspectj.xml 包中，创建配置文件 applicationContext.xml，并编写相关配置，如文件 3-13 所示。

文件 3-13　applicationContext.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xmlns:aop="http://www.springframework.org/schema/aop"
5         xsi:schemaLocation="http://www.springframework.org/schema/beans
6         http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
7         http://www.springframework.org/schema/aop
8         http://www.springframework.org/schema/aop/spring-aop-4.3.xsd">
9      <!-- 1 目标类 -->
10     <bean id="userDao" class="com.itheima.jdk.UserDaoImpl" />
11     <!-- 2 切面 -->
12     <bean id="myAspect" class="com.itheima.aspectj.xml.MyAspect" />
13     <!-- 3 aop 编程 -->
14     <aop:config>
15        <!-- 配置切面 -->
16        <aop:aspect ref="myAspect">
17           <!-- 3.1 配置切入点，通知最后增强哪些方法 -->
18           <aop:pointcut expression="execution(* com.itheima.jdk.*.*(..))"
19                                                   id="myPointCut" />
20              <!-- 3.2 关联通知 Advice 和切入点 pointCut -->
21              <!-- 3.2.1 前置通知 -->
22              <aop:before method="myBefore" pointcut-ref="myPointCut" />
23              <!-- 3.2.2 后置通知,在方法返回之后执行，就可以获得返回值
24               returning 属性：用于设置后置通知的第二个参数的名称,类型是 Object -->
25              <aop:after-returning method="myAfterReturning"
```

```
26                  pointcut-ref="myPointCut" returning="returnVal" />
27              <!-- 3.2.3 环绕通知 -->
28              <aop:around method="myAround" pointcut-ref="myPointCut" />
29              <!-- 3.2.4 抛出通知：用于处理程序发生异常-->
30              <!-- * 注意：如果程序没有异常，将不会执行增强 -->
31              <!-- * throwing 属性：用于设置通知第二个参数的名称，类型 Throwable -->
32              <aop:after-throwing method="myAfterThrowing"
33                  pointcut-ref="myPointCut" throwing="e" />
34              <!-- 3.2.5 最终通知：无论程序发生任何事情，都将执行 -->
35              <aop:after method="myAfter" pointcut-ref="myPointCut" />
36          </aop:aspect>
37      </aop:config>
38  </beans>
```

在文件 3-13 中，首先在第 4、7、8 行代码中，分别引入了 AOP 的 Schema 约束，然后在配置文件中分别定义了目标类、切面和 AOP 的配置信息。

小提示

在 AOP 的配置信息中，使用<aop:after-returning>配置的后置通知和使用<aop:after>配置的最终通知虽然都是在目标方法执行之后执行，但它们也是有所区别的。后置通知只有在目标方法成功执行后才会被织入，而最终通知不论目标方法如何结束（包括成功执行和异常中止两种情况），它都会被织入。

（4）在 com.itheima.aspectj.xml 包下，创建测试类 TestXmlAspectj，在类中为了更加清晰地演示几种通知的执行情况，这里只对 addUser()方法进行增强测试，如文件 3-14 所示。

文件 3-14　TestXmlAspectj.java

```
1  package com.itheima.aspectj.xml;
2  import org.springframework.context.ApplicationContext;
3  import
4      org.springframework.context.support.ClassPathXmlApplicationContext;
5  import com.itheima.jdk.UserDao;
6  // 测试类
7  public class TestXmlAspectj {
8      public static void main(String args[]) {
9          String xmlPath =
10                     "com/itheima/aspectj/xml/applicationContext.xml";
11         ApplicationContext applicationContext =
12                     new ClassPathXmlApplicationContext(xmlPath);
13         // 1 从 spring 容器获得内容
14         UserDao userDao = (UserDao) applicationContext.getBean("userDao");
15         // 2 执行方法
16         userDao.addUser();
17     }
18 }
```

执行程序后，控制台的输出结果如图 3-8 所示。

要查看异常通知的执行效果，可以在 UserDaoImpl 类的 addUser()方法中添加错误代码，如“int i = 10/0;”，重新运行测试类，将可以看到异常通知的执行，此时控制台的输出结果如图 3-9

所示。

图3-8 运行结果

图3-9 运行结果

从图 3-8 和图 3-9 可以看出，使用基于 XML 的声明式 AspectJ 已经实现了 AOP 开发。

3.4.2 基于注解的声明式 AspectJ

与基于代理类的 AOP 实现相比，基于 XML 的声明式 ApectJ 要便捷得多，但是它也存在着一些缺点，那就是要在 Spring 文件中配置大量的代码信息。为了解决这个问题，AspectJ 框架为 AOP 的实现提供了一套注解，用以取代 Spring 配置文件中为实现 AOP 功能所配置的臃肿代码。

关于 AspectJ 注解的介绍，如表 3-5 所示。

表 3-5 AspectJ 的注解及其描述

注解名称	描述
@Aspect	用于定义一个切面
@Pointcut	用于定义切入点表达式。在使用时还需定义一个包含名字和任意参数的方法签名来表示切入点名称。实际上，这个方法签名就是一个返回值为 void，且方法体为空的普通的方法
@Before	用于定义前置通知，相当于 BeforeAdvice。在使用时，通常需要指定一个 value 属性值，该属性值用于指定一个切入点表达式（可以是已有的切入点，也可以直接定义切入点表达式）
@AfterReturning	用于定义后置通知，相当于 AfterReturningAdvice。在使用时可以指定 pointcut/value 和 returning 属性，其中 pointcut/value 这两个属性的作用一样，都用于指定切入点表达式。returning 属性值用于表示 Advice 方法中可定义与此同名的形参，该形参可用于访问目标方法的返回值
@Around	用于定义环绕通知，相当于 MethodInterceptor。在使用时需要指定一个 value 属性，该属性用于指定该通知被植入的切入点

续表

注解名称	描述
@AfterThrowing	用于定义异常通知来处理程序中未处理的异常，相当于 ThrowAdvice。在使用时可指定 pointcut/value 和 throwing 属性。其中 pointcut/value 用于指定切入点表达式，而 throwing 属性值用于指定一个形参名来表示 Advice 方法中可定义与此同名的形参，该形参可用于访问目标方法抛出的异常
@After	用于定义最终 final 通知，不管是否异常，该通知都会执行。使用时需要指定一个 value 属性，该属性用于指定该通知被植入的切入点
@DeclareParents	用于定义引介通知，相当于 IntroductionInterceptor（不要求掌握）

为了使读者可以快速地掌握这些注解，接下来重新使用注解的形式来实现 3.4.1 小节的案例，具体步骤如下。

（1）在 chapter03 项目的 src 目录下，创建 com.itheima.aspectj.annotation 包，将文件 3-12 的切面类 MyAspect 复制到该包下，并对该文件进行编辑，如文件 3-15 所示。

文件 3-15　MyAspect.java

```
1  package com.itheima.aspectj.annotation;
2  import org.aspectj.lang.JoinPoint;
3  import org.aspectj.lang.ProceedingJoinPoint;
4  import org.aspectj.lang.annotation.After;
5  import org.aspectj.lang.annotation.AfterReturning;
6  import org.aspectj.lang.annotation.AfterThrowing;
7  import org.aspectj.lang.annotation.Around;
8  import org.aspectj.lang.annotation.Aspect;
9  import org.aspectj.lang.annotation.Before;
10 import org.aspectj.lang.annotation.Pointcut;
11 import org.springframework.stereotype.Component;
12 /**
13  * 切面类，在此类中编写通知
14  */
15 @Aspect
16 @Component
17 public class MyAspect {
18     // 定义切入点表达式
19     @Pointcut("execution(* com.itheima.jdk.*.*(..))")
20     // 使用一个返回值为 void、方法体为空的方法来命名切入点
21     private void myPointCut(){}
22     // 前置通知
23     @Before("myPointCut()")
24     public void myBefore(JoinPoint joinPoint) {
25         System.out.print("前置通知 : 模拟执行权限检查...,");
26         System.out.print("目标类是: "+joinPoint.getTarget() );
27         System.out.println(",被植入增强处理的目标方法为: "
28                 +joinPoint.getSignature().getName());
29     }
30     // 后置通知
31     @AfterReturning(value="myPointCut()")
32     public void myAfterReturning(JoinPoint joinPoint) {
```

```
33          System.out.print("后置通知：模拟记录日志...," );
34          System.out.println("被植入增强处理的目标方法为："
35                  + joinPoint.getSignature().getName());
36      }
37      // 环绕通知
38      @Around("myPointCut()")
39      public Object myAround(ProceedingJoinPoint proceedingJoinPoint)
40              throws Throwable {
41          // 开始
42          System.out.println("环绕开始：执行目标方法之前，模拟开启事务...");
43          // 执行当前目标方法
44          Object obj = proceedingJoinPoint.proceed();
45          // 结束
46          System.out.println("环绕结束：执行目标方法之后，模拟关闭事务...");
47          return obj;
48      }
49      // 异常通知
50      @AfterThrowing(value="myPointCut()",throwing="e")
51      public void myAfterThrowing(JoinPoint joinPoint, Throwable e) {
52          System.out.println("异常通知：" + "出错了" + e.getMessage());
53      }
54      // 最终通知
55      @After("myPointCut()")
56      public void myAfter() {
57          System.out.println("最终通知：模拟方法结束后的释放资源...");
58      }
59  }
```

在文件 3-15 中，首先使用@Aspect 注解定义了切面类，由于该类在 Spring 中是作为组件使用的，所以还需要添加@Component 注解才能生效。然后使用了@Poincut 注解来配置切入点表达式，并通过定义方法来表示切入点名称。接下来在每个通知相应的方法上添加了相应的注解，并将切入点名称“myPointCut”作为参数传递给需要执行增强的通知方法。如果需要其他参数（如异常通知的异常参数），可以根据代码提示传递相应的属性值。

（2）在目标类 com.itheima.jdk.UserDaoImpl 中，添加注解@Repository("userDao")。

（3）在 com.itheima.aspectj.annotation 包下，创建配置文件 applicationContext.xml，并对该文件进行编辑，如文件 3-16 所示。

文件 3-16　applicationContext.xml

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xmlns:aop="http://www.springframework.org/schema/aop"
5     xmlns:context="http://www.springframework.org/schema/context"
6     xsi:schemaLocation="http://www.springframework.org/schema/beans
7     http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
8     http://www.springframework.org/schema/aop
9     http://www.springframework.org/schema/aop/spring-aop-4.3.xsd
10    http://www.springframework.org/schema/context
```

```
    http://www.springframework.org/schema/context/spring-context-4.3.xsd">
        <!-- 指定需要扫描的包，使注解生效 -->
        <context:component-scan base-package="com.itheima" />
        <!-- 启动基于注解的声明式 AspectJ 支持 -->
        <aop:aspectj-autoproxy />
</beans>
```

在文件 3-16 中，首先引入了 context 约束信息，然后使用<context>元素设置了需要扫描的包，使注解生效。由于此案例中的目标类位于 com.itheima.jdk 包中，所以这里设置 base-package 的值为“com.itheima”。最后，使用<aop:aspectj-autoproxy />来启动 Spring 对基于注解的声明式 AspectJ 的支持。

（4）在 com.itheima.aspectj.annotation 包中，创建测试类 TestAnnotation，该类与文件 3-14 基本一致，只是配置文件的路径有所不同，如文件 3-17 所示。

文件 3-17　TestAnnotation.java

```
package com.itheima.aspectj.annotation;
import org.springframework.context.ApplicationContext;
import
   org.springframework.context.support.ClassPathXmlApplicationContext;
import com.itheima.jdk.UserDao;
// 测试类
public class TestAnnotationAspectj {
    public static void main(String args[]) {
        String xmlPath =
                "com/itheima/aspectj/annotation/applicationContext.xml";
        ApplicationContext applicationContext =
              new ClassPathXmlApplicationContext(xmlPath);
        // 1 从 spring 容器获得内容
        UserDao userDao = (UserDao) applicationContext.getBean("userDao");
        // 2 执行方法
        userDao.addUser();
    }
}
```

执行程序后，控制台的输出结果如图 3-10 所示。

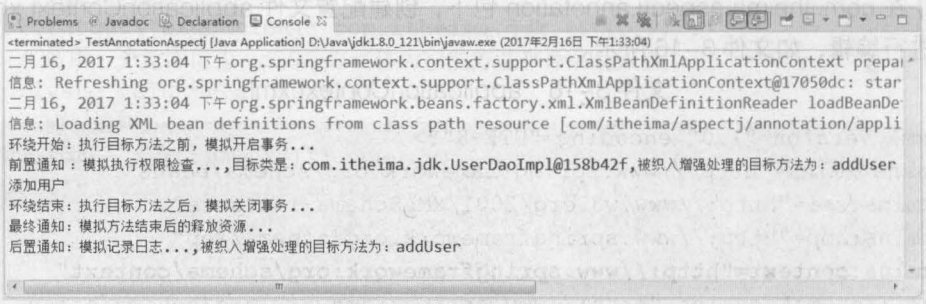


图3-10　运行结果

以 3.4.1 小节的方式来演示异常通知的执行，控制台的输出结果如图 3-11 所示。

从图 3-10 和图 3-11 可以看出，基于注解的方式与基于 XML 的方式的执行结果相同，只是

在目标方法前后通知的执行顺序发生了变化。相对来说，使用注解的方式更加简单、方便，所以在实际开发中推荐使用注解的方式进行 AOP 开发。

图3-11 运行结果

注意

如果在同一个连接点有多个通知需要执行，那么在同一切面中，目标方法之前的前置通知和环绕通知的执行顺序是未知的，目标方法之后的后置通知和环绕通知的执行顺序也是未知的。

3.5 本章小结

本章主要讲解了 Spring 框架中 AOP 的相关知识。首先对 AOP 进行了简单的介绍，然后讲解了 Spring 中的两种动态代理，接下来讲解了 Spring 中基于代理类的 AOP 实现，最后讲解了如何使用 AspectJ 框架来进行 AOP 开发。通过本章的学习，读者可以了解 AOP 的概念和作用，理解 AOP 中的相关常用术语，熟悉 Spring 中两种动态代理方式的区别，并能够掌握基于代理类和 AspectJ 框架的 AOP 开发方式。

【思考题】

1. 请列举你所知道的 AOP 专业术语并解释。
2. 请列举你所知道的 Spring 的通知类型并解释。

Java EE

4 Chapter

第 4 章 Spring 的数据库开发

学习目标

- 了解 Spring 中 JDBC 模块的作用
- 熟悉 Spring JDBC 的配置
- 掌握 JdbcTemplate 类中几个常用方法的使用

通过前几章的学习，读者对 Spring 框架核心技术中的几个重要模块已经有了一定的了解，并且通过学习，相信读者也逐渐地体会到了使用 Spring 框架的好处。Spring 框架降低了 Java EE API 的使用难度，其中就包括 JDBC 的使用难度。JDBC 是 Spring 数据访问/集成中的重要模块，本章将对 Spring 中的 JDBC 知识进行详细讲解。

4.1 Spring JDBC

Spring 的 JDBC 模块负责数据库资源管理和错误处理，大大简化了开发人员对数据库的操作，使得开发人员可以从烦琐的数据库操作中解脱出来，从而将更多的精力投入到编写业务逻辑中。接下来的两个小节，将针对 Spring 中的 JDBC 模块内容进行详细的讲解。

4.1.1 Spring JdbcTemplate 的解析

针对数据库的操作，Spring 框架提供了 JdbcTemplate 类，该类是 Spring 框架数据抽象层的基础，其他更高层次的抽象类却是构建于 JdbcTemplate 类之上。可以说，JdbcTemplate 类是 Spring JDBC 的核心类。

JdbcTemplate 类的继承关系十分简单。它继承自抽象类 JdbcAccessor，同时实现了 JdbcOperations 接口，如图 4-1 所示。

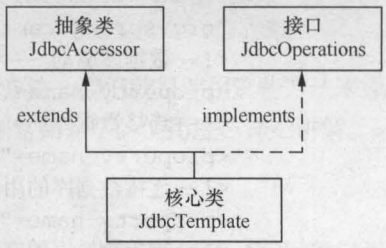


图4-1 JdbcTemplate继承关系

从图 4-1 可以看出，JdbcTemplate 类的直接父类是 JdbcAccessor，该类为子类提供了一些访问数据库时使用的公共属性，具体如下。

- DataSource：其主要功能是获取数据库连接，具体实现时还可以引入对数据库连接的缓冲池和分布式事务的支持，它可以作为访问数据库资源的标准接口。
- SQLExceptionTranslator：org.springframework.jdbc.support.SQLExceptionTranslator 接口负责对 SQLException 进行转译工作。通过必要的设置或者获取 SQLExceptionTranslator 中的方法，可以使 JdbcTemplate 在需要处理 SQLException 时，委托 SQLExceptionTranslator 的实现类来完成相关的转译工作。

JdbcOperations 接口定义了在 JdbcTemplate 类中可以使用的操作集合，包括添加、修改、查询和删除等操作。

4.1.2 Spring JDBC 的配置

Spring JDBC 模块主要由 4 个包组成，分别是 core（核心包）、dataSource（数据源包）、object（对象包）和 support（支持包），关于这 4 个包的具体说明如表 4-1 所示。

表 4-1 Spring JDBC 中的主要包及说明

包名	说明
core	包含了 JDBC 的核心功能，包括 JdbcTemplate 类、SimpleJdbcInsert 类、SimpleJdbcCall 类以及 NamedParameterJdbcTemplate 类
dataSource	访问数据源的实用工具类，它有多种数据源的实现，可以在 Java EE 容器外部测试 JDBC 代码

续表

包名	说明
object	以面向对象的方式访问数据库，它允许执行查询并将返回结果作为业务对象，可以在数据表的列和业务对象的属性之间映射查询结果
support	包含了 core 和 object 包的支持类，例如，提供异常转换功能的 SQLException 类

从表 4-1 可以看出，Spring 对数据库的操作都封装在了这几个包中，而想要使用 Spring JDBC，就需要对其进行配置。在 Spring 中，JDBC 的配置是在配置文件 applicationContext.xml 中完成的，其配置模板下所示。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">
    <!-- 1 配置数据源 -->
    <bean id="dataSource" class=
       "org.springframework.jdbc.datasource.DriverManagerDataSource">
       <!--数据库驱动 -->
       <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
       <!--连接数据库的 url -->
       <property name="url" value="jdbc:mysql://localhost:3306/spring"/>
       <!--连接数据库的用户名 -->
       <property name="username" value="root"/>
       <!--连接数据库的密码 -->
       <property name="password" value="root"/>
    </bean>
    <!-- 2 配置 JDBC 模板 -->
    <bean id="jdbcTemplate"
        class="org.springframework.jdbc.core.JdbcTemplate">
       <!-- 默认必须使用数据源 -->
       <property name="dataSource" ref="dataSource"/>
    </bean>
    <!-- 3 配置注入类 -->
    <bean id="xxx" class="Xxx">
        <property name="jdbcTemplate" ref="jdbcTemplate"/>
    </bean>
    ...
</beans>
```

在上述代码中，定义了 3 个 Bean，分别是 dataSource、jdbcTemplate 和需要注入类的 Bean。其中 dataSource 对应的 org.springframework.jdbc.datasource.DriverManagerDataSource 类用于对数据源进行配置，jdbcTemplate 对应的 org.springframework.jdbc.core.JdbcTemplate 类中定义了 JdbcTemplate 的相关配置。上述代码中 dataSource 的配置就是 JDBC 连接数据库时所需的 4 个属性，如表 4-2 所示。

表 4-2 中的 4 个属性，需要根据数据库类型或者机器配置的不同设置相应的属性值。例如，如果数据库类型不同，需要更改驱动名称；如果数据库不在本地，则需要将地址中的 localhost 替换成相应的主机 IP；如果修改过 MySQL 数据库的端口号（默认为 3306），则需要加上修改后

的端口号，如果未修改，则端口号可以省略；同时连接数据库的用户名和密码需要与数据库创建时设置的用户名和密码保持一致，本示例中 Spring 数据库的用户名和密码都是 root。

表 4-2 dataSource 的 4 个属性

属性名	含义
driverClassName	所使用的驱动名称，对应驱动 JAR 包中的 Driver 类
url	数据源所在地址
username	访问数据库的用户名
password	访问数据库的密码

定义 jdbcTemplate 时，需要将 dataSource 注入到 jdbcTemplate 中，而其他需要使用 jdbcTemplate 的 Bean，也需要将 jdbcTemplate 注入到该 Bean 中（通常注入到 Dao 类中，在 Dao 类中进行与数据库的相关操作）。

4.2 Spring JdbcTemplate 的常用方法

在 JdbcTemplate 类中，提供了大量的更新和查询数据库的方法，我们就是使用这些方法来操作数据库的。接下来的几个小节中，将对 JdbcTemplate 类中一些常用方法的使用进行详细讲解。

4.2.1 execute()

execute(String sql)方法能够完成执行 SQL 语句的功能。下面以创建数据表的 SQL 语句为例，来演示此方法的使用，具体步骤如下。

（1）在 MySQL 中，创建一个名为 spring 的数据库，创建方式如图 4-2 所示。

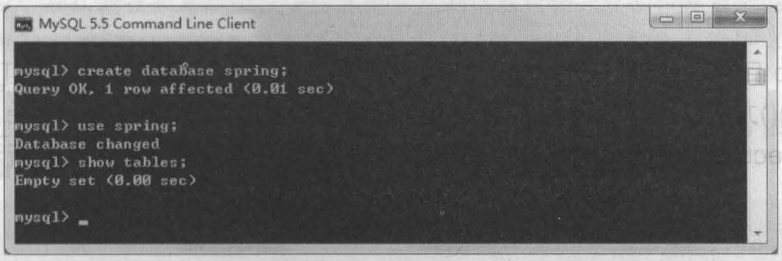


图4-2 创建spring数据库

在图 4-2 中，首先使用 SQL 语句创建了数据库 spring，然后选择使用 spring。为了便于后续验证数据表是通过 execute（String sql）方法执行创建的，这里使用了 show tables 语句查看数据库中的表，其结果显示为空。

（2）在 Eclipse 中，创建一个名为 chapter04 的 Web 项目，将运行 Spring 框架所需的 5 个基础 JAR 包、MySQL 数据库的驱动 JAR 包、Spring JDBC 的 JAR 包以及 Spring 事务处理的 JAR 包复制到项目的 lib 目录，并发布到类路径中。项目中所添加的 JAR 包如图 4-3 所示。

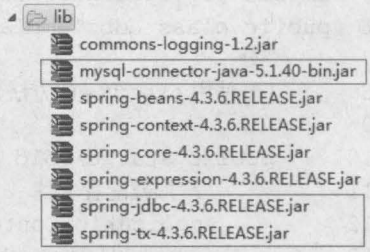


图4-3 Spring JDBC操作相关的JAR包

（3）在 src 目录下，创建配置文件 applicationContext.xml，在该文件中配置 id 为 dataSource 的数据源 Bean 和 id 为 jdbcTemplate 的 JDBC 模板 Bean，并将数据源注入到 JDBC 模板中，如文件 4-1 所示。

文件 4-1　applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">
    <!-- 1 配置数据源 -->
    <bean id="dataSource" class=
    "org.springframework.jdbc.datasource.DriverManagerDataSource">
        <!--数据库驱动 -->
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <!--连接数据库的 url -->
        <property name="url" value="jdbc:mysql://localhost/spring" />
        <!--连接数据库的用户名 -->
        <property name="username" value="root" />
        <!--连接数据库的密码 -->
        <property name="password" value="root" />
    </bean>
    <!-- 2 配置 JDBC 模板 -->
    <bean id="jdbcTemplate"
          class="org.springframework.jdbc.core.JdbcTemplate">
        <!-- 默认必须使用数据源 -->
        <property name="dataSource" ref="dataSource" />
    </bean>
</beans>
```

（4）在 src 目录下，创建一个 com.itheima.jdbc 包，在该包中创建测试类 JdbcTemplateTest。在该类的 main()方法中通过 Spring 容器获取在配置文件中定义的 JdbcTemplate 实例，然后使用该实例的 execute(String sql)方法执行创建数据表的 SQL 语句，如文件 4-2 所示。

文件 4-2　JdbcTemplateTest.java

```
package com.itheima.jdbc;
import org.springframework.context.ApplicationContext;
import
    org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.jdbc.core.JdbcTemplate;
public class JdbcTemplateTest {
    /**
     * 使用 execute()方法建表
     */
    public static void main(String[] args) {
        // 加载配置文件
        ApplicationContext applicationContext =
          new ClassPathXmlApplicationContext("applicationContext.xml");
        // 获取 JdbcTemplate 实例
```

```
        JdbcTemplate jdTemplate =
                (JdbcTemplate) applicationContext.getBean("jdbcTemplate");
        // 使用 execute()方法执行 SQL 语句，创建用户账户管理表 account
        jdTemplate.execute("create table account(" +
                           "id int primary key auto_increment," +
                           "username varchar(50)," +
                           "balance double)");
        System.out.println("账户表 account 创建成功！");
    }
}
```

成功运行程序后，再次查询 spring 数据库，其结果如图 4-4 所示。

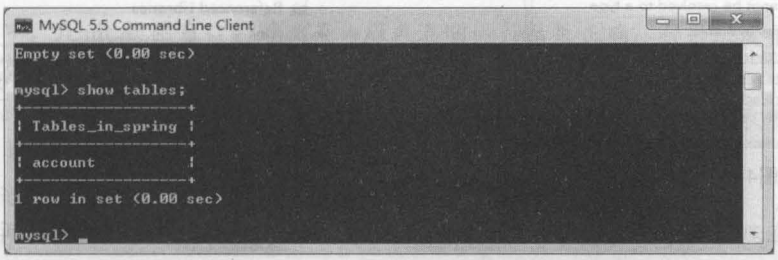


图4-4　spring数据库中的表

从图 4-4 可以看出，程序使用 execute（String sql）方法执行的 SQL 语句已成功创建了数据表 account。

多学一招：使用 JUnit 测试

在软件开发过程中，需要有相应的测试工作。依据测试目的不同，可以将软件测试分为单元测试、集成测试、确认测试和系统测试等。其中单元测试在软件开发阶段是最底层的测试，它易于及时发现并解决问题。JUnit 就是一个进行单元测试的开源框架，下面以文件 4-2 为例，来简单学习一下单元测试框架 JUnit4 的使用。

将文件 4-2 中的 main()方法，修改成名称为 mainTest()的普通方法，并在方法上添加单元测试的注解@Test，其代码如下所示。

```
@Test
public void mainTest() {
    // 加载配置文件
    ApplicationContext applicationContext =
            new ClassPathXmlApplicationContext("applicationContext.xml");
    // 获取 JdbcTemplate 实例
    JdbcTemplate jdTemplate =
            (JdbcTemplate) applicationContext.getBean("jdbcTemplate");
    // 使用 execute()方法执行 SQL 语句，创建用户账户管理表 account
    jdTemplate.execute("create table account(" +
                       "id int primary key auto_increment," +
                       "username varchar(50)," +
                       "balance double)");
    System.out.println("账户表 account 创建成功！");
}
```

@Test 就是 Junit4 用于测试的注解，要测试哪个方法，只需要在相应测试的方法上添加此注解即可。当在需要测试的方法上添加@Test 注解后，Eclipse 会在所添加的@Test 处报出 Test cannot be resolved to a type 的错误，将鼠标移到@Test 处，会显示出错误提示框如图 4-5 所示。

单击提示框中的 Add JUnit4 library to the build path 后，Eclipse 会自动将 JUnit4 的支持包加入到项目中，如图 4-6 所示。

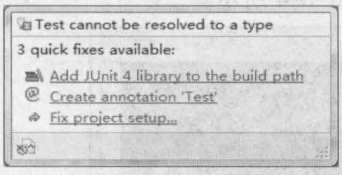


图4-5 添加JUnit4

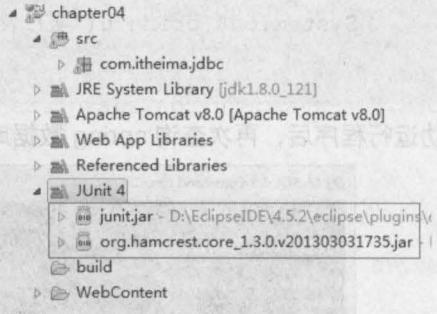


图4-6 JUnit框架支持包

加入后，在测试类中会自动导入 org.junit.Test 包，此时测试类中的代码将不再报错。在执行程序时，只需使用鼠标右键单击 mainTest()方法，在弹出的快捷菜单中选择 Run As→JUnit Test 选项来运行测试方法即可，如图 4-7 所示。

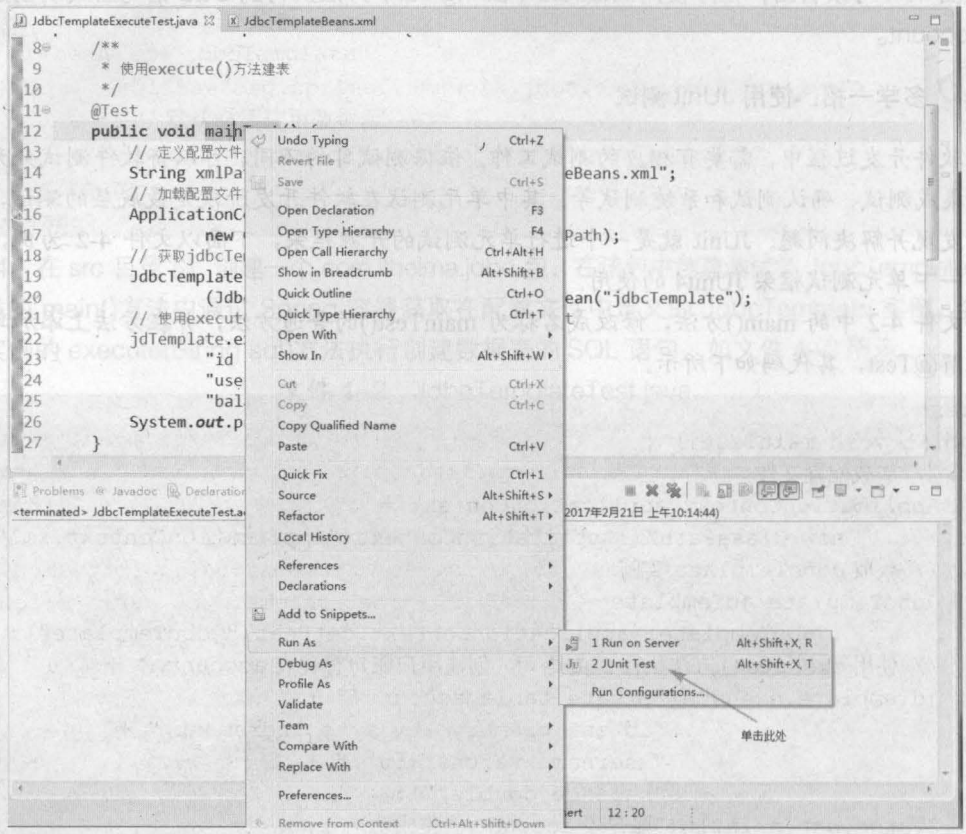


图4-7 运行JUnit

单击 JUnit Test 选项后，Eclipse 中会多出一个名为 JUnit 的视图窗口，其显示结果如图 4-8 所示。

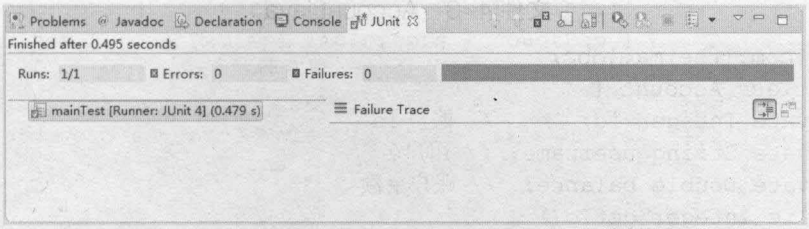


图4-8 JUnit控制台

在图 4-8 中，JUnit 视图窗口的进度条为绿色表明运行结果正确，如果进度条为红色则表示有错误，并且会在窗口中显示所报的错误信息。

需要注意的是，在运行此方法时，需要先将数据库中已创建好的 account 表删除，否则执行此方法时会报出 account 表已经存在的错误。

测试执行通过后，Console 控制台的输出结果如图 4-9 所示。

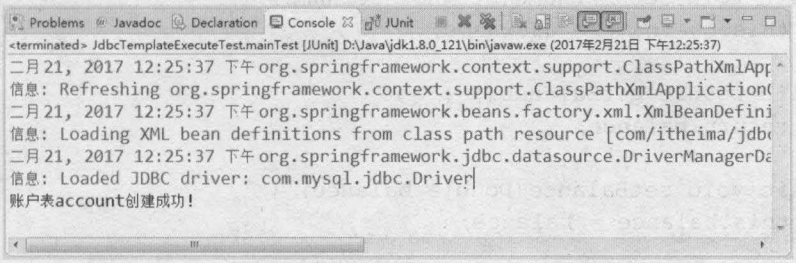


图4-9 运行结果

从图 4-9 可以看出，mainTest()方法已经执行成功，这就是单元测试的使用。

4.2.2 update()

update()方法可以完成插入、更新和删除数据的操作。在 JdbcTemplate 类中，提供了一系列的 update()方法，其常用方法如表 4-3 所示。

表 4-3 JdbcTemplate 类中常用的 update()方法

方法	说明
int update(String sql)	该方法是最简单的 update 方法重载形式，它直接执行传入的 SQL 语句，并返回受影响的行数
int update(PreparedStatementCreator psc)	该方法执行从 PreparedStatementCreator 返回的语句，然后返回受影响的行数
int update(String sql, PreparedStatementSetter pss)	该方法通过 PreparedStatementSetter 设置 SQL 语句中的参数，并返回受影响的行数
int update(String sql,Object... args)	该方法使用 Object...设置 SQL 语句中的参数，要求参数不能为 NULL，并返回受影响的行数

接下来，通过一个用户账户管理的案例来演示 update()方法的使用，具体步骤如下。

（1）在 chapter04 项目的 com.itheima.jdbc 包中，创建 Account 类，在该类中定义 id、username 和 balance 属性，以及其对应的 getter/setter 方法，如文件 4-3 所示。

文件 4-3　Account.java

```
package com.itheima.jdbc;
public class Account {
    private Integer id;       // 账户 id
    private String username; // 用户名
    private Double balance;  // 账户余额
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public Double getBalance() {
        return balance;
    }
    public void setBalance(Double balance) {
        this.balance = balance;
    }
    public String toString() {
        return "Account [id=" + id + ", "
                + "username=" + username +
                ", balance=" + balance + "]";
    }
}
```

（2）在 com.itheima.jdbc 包中，创建接口 AccountDao，并在接口中定义添加、更新和删除账户的方法，如文件 4-4 所示。

文件 4-4　AccountDao.java

```
package com.itheima.jdbc;
public interface AccountDao {
    // 添加
    public int addAccount(Account account);
    // 更新
    public int updateAccount(Account account);
    // 删除
    public int deleteAccount(int id);
}
```

（3）在 com.itheima.jdbc 包中，创建 AccountDao 接口的实现类 AccountDaoImpl，并在类中实现添加、更新和删除账户的方法，编辑后如文件 4-5 所示。

文件 4-5 AccountDaoImpl.java

```
package com.itheima.jdbc;
import org.springframework.jdbc.core.JdbcTemplate;
public class AccountDaoImpl implements AccountDao {
    // 声明 JdbcTemplate 属性及其 setter 方法
    private JdbcTemplate jdbcTemplate;
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
    // 添加账户
    public int addAccount(Account account) {
        // 定义 SQL
        String sql = "insert into account(username,balance) value(?,?)";
        // 定义数组来存储 SQL 语句中的参数
        Object[] obj = new Object[] {
                        account.getUsername(),
                        account.getBalance()
        };
        // 执行添加操作，返回的是受 SQL 语句影响的记录条数
        int num = this.jdbcTemplate.update(sql, obj);
        return num;
    }
    // 更新账户
    public int updateAccount(Account account) {
        // 定义 SQL
        String sql = "update account set username=?,balance=? where id = ?";
        // 定义数组来存储 SQL 语句中的参数
        Object[] params = new Object[] {
                            account.getUsername(),
                            account.getBalance(),
                            account.getId()
         };
        // 执行更新操作，返回的是受 SQL 语句影响的记录条数
        int num = this.jdbcTemplate.update(sql, params);
        return num;
    }
    // 删除账户
    public int deleteAccount(int id) {
        // 定义 SQL
        String sql = "delete  from account where id = ? ";
        // 执行删除操作，返回的是受 SQL 语句影响的记录条数
        int num = this.jdbcTemplate.update(sql, id);
        return num;
    }
}
```

从上述三种操作的代码可以看出，添加、更新和删除操作的实现步骤类似，只是定义的 SQL 语句有所不同。

（4）在 applicationContext.xml 中，定义一个 id 为 accountDao 的 Bean，该 Bean 用于将

jdbcTemplate 注入到 accountDao 实例中，其代码如下所示。

```
<!--定义 id 为 accountDao 的 Bean-->
<bean id="accountDao" class="com.itheima.jdbc.AccountDaoImpl">
    <!-- 将 jdbcTemplate 注入到 accountDao 实例中 -->
    <property name="jdbcTemplate" ref="jdbcTemplate" />
</bean>
```

（5）在测试类 JdbcTemplateTest 中，添加一个测试方法 addAccountTest()，该方法主要用于添加用户账户信息，其代码如下所示。

```
@Test
public void addAccountTest() {
    // 加载配置文件
    ApplicationContext applicationContext =
            new ClassPathXmlApplicationContext("applicationContext.xml");
    // 获取 AccountDao 实例
    AccountDao accountDao =
            (AccountDao) applicationContext.getBean("accountDao");
    // 创建 Account 对象，并向 Account 对象中添加数据
    Account account = new Account();
    account.setUsername("tom");
    account.setBalance(1000.00);
    // 执行 addAccount()方法，并获取返回结果
    int num = accountDao.addAccount(account);
    if (num > 0) {
        System.out.println("成功插入了" + num + "条数据！");
    } else {
        System.out.println("插入操作执行失败！");
    }
}
```

在上述代码中，获取了 AccountDao 的实例后，又创建了 Account 对象，并向 Account 对象中添加了属性值。然后调用了 AccountDao 对象的 addAccount()方法向数据表中添加一条数据。最后，通过返回的受影响的行数来判断数据是否插入成功。

使用 Junit4 测试运行后，控制台的输出结果如图 4-10 所示。

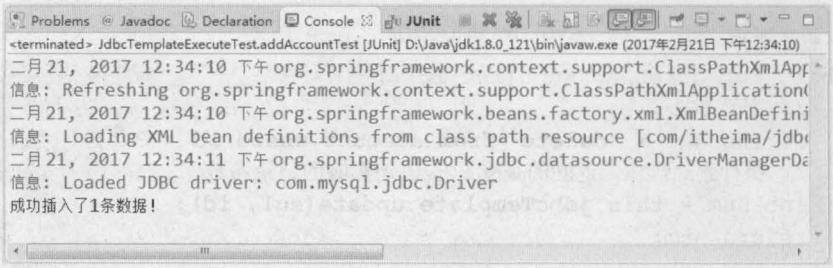


图4-10　运行结果

此时再次查询数据库中的 account 表，其结果如图 4-11 所示。

从图 4-11 可以看出，使用 JdbcTemplate 的 update()方法已成功地向数据表中插入了一条数据。

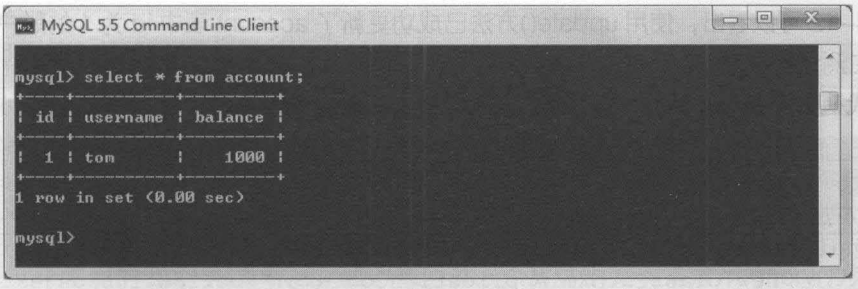


图4-11　account表

（6）执行完插入操作后，接下来使用 JdbcTemplate 类的 update()方法执行更新操作。在测试类 JdbcTemplateTest 中，添加一个测试方法 updateAccountTest()，其代码如下所示。

```
@Test
public void updateAccountTest() {
    // 加载配置文件
    ApplicationContext applicationContext =
            new ClassPathXmlApplicationContext("applicationContext.xml");
    // 获取 AccountDao 实例
    AccountDao accountDao =
            (AccountDao) applicationContext.getBean("accountDao");
    // 创建 Account 对象，并向 Account 对象中添加数据
    Account account = new Account();
    account.setId(1);
    account.setUsername("tom");
    account.setBalance(2000.00);
    // 执行 updateAccount()方法，并获取返回结果
    int num = accountDao.updateAccount(account);
    if (num > 0) {
        System.out.println("成功修改了" + num + "条数据！");
    } else {
        System.out.println("修改操作执行失败！");
    }
}
```

与 addAccountTest()方法相比，更新操作的代码增加了 id 属性值的设置，并将余额修改为 2000 后，调用了 AccountDao 对象中的 updateAccount()方法执行对数据表的更新操作。

使用 Junit4 运行方法后，再次查询数据库中的 account 表，其结果如图 4-12 所示。

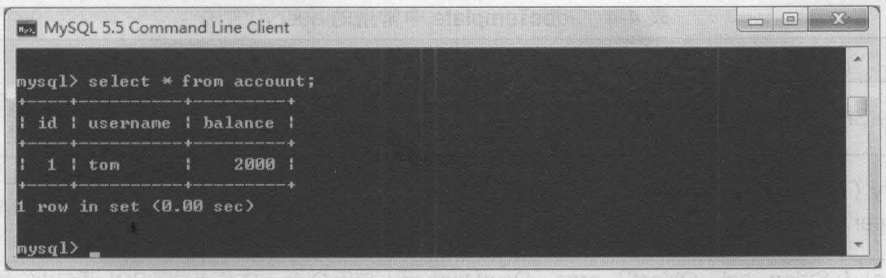


图4-12　account表

从图 4-12 可以看出，使用 update()方法已成功更新了 account 表中 id 为 1 的账户余额信息。

（7）在测试类 JdbcTemplateTest 中，添加一个测试方法 deleteAccountTest()，来执行删除操作，其代码如下所示。

```
@Test
public void deleteAccountTest() {
    // 加载配置文件
    ApplicationContext applicationContext =
            new ClassPathXmlApplicationContext("applicationContext.xml");
    // 获取 AccountDao 实例
    AccountDao accountDao =
            (AccountDao) applicationContext.getBean("accountDao");
    // 执行 deleteAccount()方法，并获取返回结果
    int num = accountDao.deleteAccount(1);
    if (num > 0) {
        System.out.println("成功删除了" + num + "条数据！");
    } else {
        System.out.println("删除操作执行失败！");
    }
}
```

在上述代码中，获取了 AccountDao 的实例后，执行了实例中的 deleteAccount()方法来删除 id 为 1 的数据。

使用 Junit4 测试运行方法后，查询 account 表中数据，其结果如图 4-13 所示。

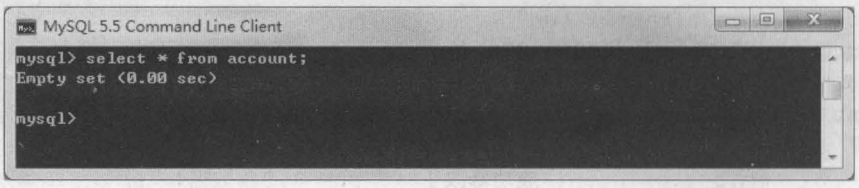


图4-13　account表

从图 4-13 可以看出，已成功通过 update()方法删除了 id 为 1 的数据。由于 account 表中只有一条数据，所以删除后表中数据为空。

4.2.3　query()

JdbcTemplate 类中还提供了大量的 query()方法来处理各种对数据库表的查询操作。其中，常用的几个 query()方法如表 4-4 所示。

表 4-4　JdbcTemplate 中常用的 query()方法

方法	说明
List query(String sql, RowMapper rowMapper)	执行 String 类型参数提供的 SQL 语句，并通过 RowMapper 返回一个 List 类型的结果
List query（String sql, PreparedStatementSetter pss, RowMapper rowMapper）	根据 String 类型参数提供的 SQL 语句创建 PreparedStatement 对象，通过 RowMapper 将结果返回到 List 中
List query（String sql, Object[] args, RowMapper rowMapper）	使用 Object[]的值来设置 SQL 语句中的参数值，采用 RowMapper 回调方法可以直接返回 List 类型的数据

续表

方法	说明
queryForObject(String sql, RowMapper rowMapper, Object... args)	将 args 参数绑定到 SQL 语句中，并通过 RowMapper 返回一个 Object 类型的单行记录
queryForList (String sql,Object[] args, class<T> elementType)	该方法可以返回多行数据的结果，但必须是返回列表，elementType 参数返回的是 List 元素类型

了解了几个常用的 query()方法后，接下来通过一个具体的案例来演示 query()方法的使用，其实现步骤如下。

（1）向数据表 account 中插入几条数据（也可以使用数据库图形化工具手动向表中插入数据），插入后 account 表中的数据如图 4-14 所示。

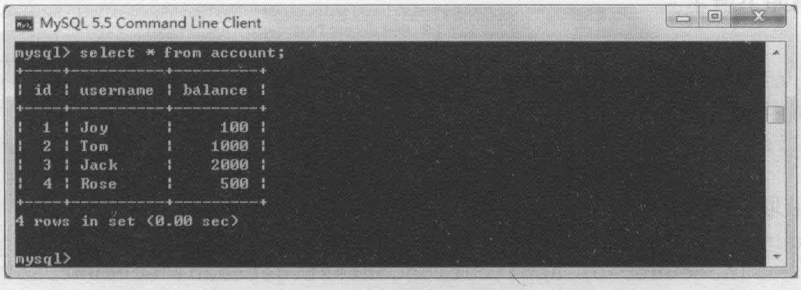


图4-14　account表

（2）在 AccountDao 中，分别创建一个通过 id 查询单个账户和查询所有账户的方法，其代码如下所示。

```
// 通过 id 查询
public Account findAccountById(int id);
// 查询所有账户
public List<Account> findAllAccount();
```

（3）在 AccountDao 接口的实现类 AccountDaoImpl 中，实现接口中的方法，并使用 query()方法分别进行查询，其代码如下所示。

```
// 通过 id 查询账户数据信息
public Account findAccountById(int id) {
    //定义 SQL 语句
    String sql = "select * from account where id = ?";
    // 创建一个新的 BeanPropertyRowMapper 对象
    RowMapper<Account> rowMapper =
            new BeanPropertyRowMapper<Account>(Account.class);
    // 将 id 绑定到 SQL 语句中，并通过 RowMapper 返回一个 Object 类型的单行记录
    return this.jdbcTemplate.queryForObject(sql, rowMapper, id);
}
// 查询所有账户信息
public List<Account> findAllAccount() {
    // 定义 SQL 语句
    String sql = "select * from account";
    // 创建一个新的 BeanPropertyRowMapper 对象
```

```
    RowMapper<Account> rowMapper =
                    new BeanPropertyRowMapper<Account>(Account.class);
    // 执行静态的 SQL 查询，并通过 RowMapper 返回结果
    return this.jdbcTemplate.query(sql, rowMapper);
}
```

在上面两个方法代码中，BeanPropertyRowMapper 是 RowMapper 接口的实现类，它可以自动地将数据表中的数据映射到用户自定义的类中（前提是用户自定义类中的字段要与数据表中的字段相对应）。创建完 BeanPropertyRowMapper 对象后，在 findAccountById()方法中通过 queryForObject()方法返回了一个 Object 类型的单行记录，而在 findAllAccount()方法中通过 query()方法返回了一个结果集合。

（4）在测试类 JdbcTemplateTest 中，添加一个测试方法 findAccountByIdTest()来测试条件查询，其代码如下所示。

```
@Test
public void findAccountByIdTest() {
    // 加载配置文件
    ApplicationContext applicationContext =
            new ClassPathXmlApplicationContext("applicationContext.xml");
    // 获取 AccountDao 实例
    AccountDao accountDao =
            (AccountDao) applicationContext.getBean("accountDao");
    // 执行 findAccountById()方法
    Account account = accountDao.findAccountById(1);
    System.out.println(account);
}
```

上述代码通过执行 findAccountById()方法获取了 id 为 1 的对象信息，并通过输出语句输出。使用 JUnit4 测试运行后，控制台的输出结果如图 4-15 所示。

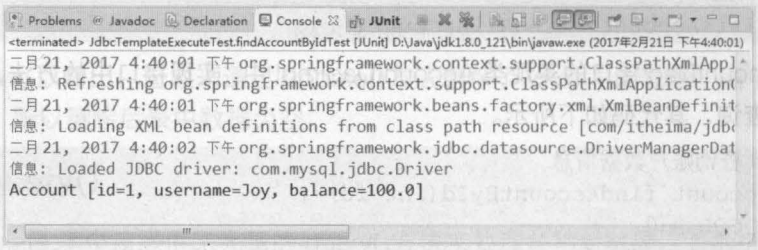


图4-15　运行结果

（5）测试完条件查询单个数据的方法后，接下来测试查询所有用户账户信息的方法。在测试类 JdbcTemplateTest 中，添加一个测试方法 findAllAccountTest()，其代码如下所示。

```
@Test
public void findAllAccountTest() {
    // 加载配置文件
    ApplicationContext applicationContext =
            new ClassPathXmlApplicationContext("applicationContext.xml");
    // 获取 AccountDao 实例
    AccountDao accountDao =
```

```
            (AccountDao) applicationContext.getBean("accountDao");
        // 执行 findAllAccount()方法,获取 Account 对象的集合
        List<Account> account = accountDao.findAllAccount();
        // 循环输出集合中的对象
        for (Account act : account) {
            System.out.println(act);
        }
    }
```

在上述代码中，调用了 AccountDao 对象的 findAllAccount()方法查询所有用户账户信息集合，并通过 for 循环输出查询结果。

使用 JUnit4 成功运行 findAllUserTest()方法后，控制台的显示信息如图 4-16 所示。

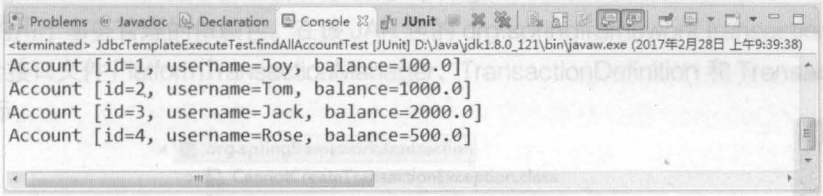


图4-16　运行结果

从图 4-16 可以看出，数据表 account 中的 4 条记录都已经被查询出来。

4.3 本章小结

本章对 Spring 框架中使用 JDBC 进行数据操作的知识进行了详细讲解。首先讲解了 Spring JDBC 中的核心类以及如何在 Spring 中配置 JDBC，然后通过案例讲解了 Spring JDBC 核心类 JdbcTemplate 中常用方法的使用。通过本章的学习，读者能够学会如何使用 Spring 框架进行数据库开发，并能深切地体会到 Spring 框架的强大。

【思考题】

1. 请简述 Spring JDBC 是如何进行配置的。
2. 请简述 Spring JdbcTemplate 类中几个常用方法的作用。

Java EE

5 Chapter

第 5 章
Spring 的事务管理

学习目标

- 熟悉 Spring 事务管理的 3 个核心接口
- 了解 Spring 事务管理的两种方式
- 掌握基于 XML 和 Annotation 的声明式事务的使用

通过上一章的学习，读者已经掌握了如何使用 Spring 来操作数据库，但是在实际开发中，操作数据库时还会涉及事务管理问题，为此 Spring 提供了专门用于事务处理的 API。Spring 的事务管理简化了传统的事务管理流程，并且在一定程度上减少了开发者的工作量。本章将针对 Spring 的事务管理功能进行详细讲解。

5.1 Spring 事务管理概述

5.1.1 事务管理的核心接口

在 Spring 的所有 JAR 包中，包含一个名为 spring-tx-4.3.6.RELEASE 的 JAR 包，该包就是 Spring 提供的用于事务管理的依赖包。在该 JAR 包的 org.springframework.transaction 包中，我们可以找到 3 个接口文件 PlatformTransactionManager、TransactionDefinition 和 TransactionStatus，如图 5-1 所示。

图5-1 事务管理核心接口

在图 5-1 中，方框标注的 3 个接口文件就是 Spring 事务管理所涉及的 3 个核心接口，接下来对这 3 个接口的作用分别进行讲解。

1. PlatformTransactionManager

PlatformTransactionManager 接口是 Spring 提供的平台事务管理器，主要用于管理事务。该接口中提供了 3 个事务操作的方法，具体如下。

- TransactionStatus getTransaction (TransactionDefinition definition)：用于获取事务状态信息。
- void commit (TransactionStatus status)：用于提交事务。
- void rollback (TransactionStatus status)：用于回滚事务。

在上面的 3 个方法中，getTransaction (TransactionDefinition definition) 方法会根据 TransactionDefinition 参数返回一个 TransactionStatus 对象，TransactionStatus 对象就表示一个事务，它被关联在当前执行的线程上。

PlatformTransactionManager 接口只是代表事务管理的接口，它并不知道底层是如何管理事务的，它只需要事务管理提供上面的 3 个方法，但具体如何管理事务则由它的实现类来完成。

PlatformTransactionManager 接口有许多不同的实现类，常见的几个实现类如下。

- org.springframework.jdbc.datasource.DataSourceTransactionManager：用于配置 JDBC 数据源的事务管理器。
- org.springframework.orm.hibernate4.HibernateTransactionManager：用于配置 Hibernate 的事务管理器。
- org.springframework.transaction.jta.JtaTransactionManager：用于配置全局事务管理器。

当底层采用不同的持久层技术时，系统只需使用不同的 PlatformTransactionManager 实现类即可。

2. TransactionDefinition

TransactionDefinition 接口是事务定义（描述）的对象，该对象中定义了事务规则，并提供了获取事务相关信息的方法，具体如下。

- String getName()：获取事务对象名称。
- int getIsolationLevel()：获取事务的隔离级别。
- int getPropagationBehavior()：获取事务的传播行为。
- int getTimeout()：获取事务的超时时间。
- boolean isReadOnly()：获取事务是否只读。

上述方法中，事务的传播行为是指在同一个方法中，不同操作前后所使用的事务。传播行为有很多种，具体如表 5-1 所示。

表 5-1 传播行为的种类

属性名称	值	描述
PROPAGATION_REQUIRED	REQUIRED	表示当前方法必须运行在一个事务环境当中，如果当前方法已处于事务环境中，则可以直接使用该方法；否则会开启一个新事务后执行该方法
PROPAGATION_SUPPORTS	SUPPORTS	如果当前方法处于事务环境中，则使用当前事务，否则不使用事务
PROPAGATION_MANDATORY	MANDATORY	表示调用该方法的线程必须处于当前事务环境中，否则将抛异常
PROPAGATION_REQUIRES_NEW	REQUIRES_NEW	要求方法在新的事务环境中执行。如果当前方法已在事务环境中，则先暂停当前事务，在启动新的事务后执行该方法；如果当前方法不在事务环境中，则会启动一个新的事务后执行方法
PROPAGATION_NOT_SUPPORTED	NOT_SUPPORTED	不支持当前事务，总是以非事务状态执行。如果调用该方法的线程处于事务环境中，则先暂停当前事务，然后执行该方法
PROPAGATION_NEVER	NEVER	不支持当前事务。如果调用该方法的线程处于事务环境中，将抛异常
PROPAGATION_NESTED	NESTED	即使当前执行的方法处于事务环境中，依然会启动一个新的事务，并且方法在嵌套的事务里执行；即使当前执行的方法不在事务环境中，也会启动一个新事务，然后执行该方法

在事务管理过程中，传播行为可以控制是否需要创建事务以及如何创建事务，通常情况下，数据的查询不会影响原数据的改变，所以不需要进行事务管理，而对于数据的插入、更新和删除操作，必须进行事务管理。如果没有指定事务的传播行为，Spring 默认传播行为是 REQUIRED。

3. TransactionStatus

TransactionStatus 接口是事务的状态，它描述了某一时间点上事务的状态信息。该接口中包含 6 个方法，具体如下。

- void flush()：刷新事务。
- boolean hasSavepoint()：获取是否存在保存点。
- boolean isCompleted()：获取事务是否完成。
- boolean isNewTransaction()：获取是否是新事务。
- boolean isRollbackOnly()：获取是否回滚。
- void setRollbackOnly()：设置事务回滚。

5.1.2 事务管理的方式

Spring 中的事务管理分为两种方式：一种是传统的编程式事务管理，另一种是声明式事务管理。

- 编程式事务管理：是通过编写代码实现的事务管理，包括定义事务的开始、正常执行后的事务提交和异常时的事务回滚。
- 声明式事务管理：是通过 AOP 技术实现的事务管理，其主要思想是将事务管理作为一个“切面”代码单独编写，然后通过 AOP 技术将事务管理的“切面”代码织入到业务目标类中。

声明式事务管理最大的优点在于开发者无须通过编程的方式来管理事务，只需在配置文件中进行相关的事务规则声明，就可以将事务规则应用到业务逻辑中。这使得开发人员可以更加专注于核心业务逻辑代码的编写，在一定程度上减少了工作量，提高了开发效率，所以在实际开发中，通常都推荐使用声明式事务管理。本书主要讲解的就是 Spring 的声明式事务管理。

5.2 声明式事务管理

Spring 的声明式事务管理可以通过两种方式来实现，一种是基于 XML 的方式，另一种是基于 Annotation 的方式。接下来的两个小节中，将对这两种声明式事务管理方式进行详细讲解。

5.2.1 基于 XML 方式的声明式事务

基于 XML 方式的声明式事务管理是通过在配置文件中配置事务规则的相关声明来实现的。Spring 2.0 以后，提供了 tx 命名空间来配置事务，tx 命名空间下提供了<tx:advice>元素来配置事务的通知（增强处理）。当使用<tx:advice>元素配置了事务的增强处理后，就可以通过编写的 AOP 配置，让 Spring 自动对目标生成代理。

配置<tx:advice>元素时，通常需要指定 id 和 transaction-manager 属性，其中 id 属性是配置文件中的唯一标识，transaction-manager 属性用于指定事务管理器。除此之外，还需要配置一个<tx:attributes>子元素，该子元素可通过配置多个<tx:method>子元素来配置执行事务的细节。<tx:advice>元素及其子元素如图 5-2 所示。

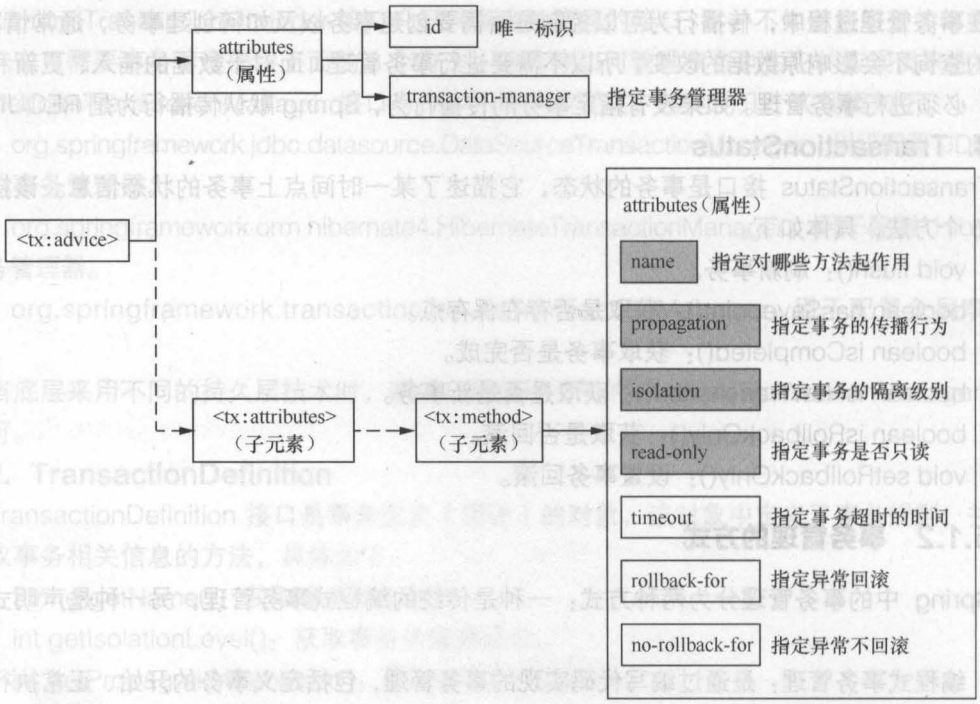


图5-2 <tx:advice>元素及其子元素

在图 5-2 中，配置<tx:advice>元素的重点是配置<tx:method>子元素，图中使用灰色标注的几个属性是<tx:method>元素中的常用属性。

关于<tx:method>元素的属性描述如表 5-2 所示。

表 5-2 <tx:method>元素的属性

属性名称	描述
name	该属性为必选属性，它指定了与事务属性相关的方法名。其属性值支持使用通配符，如'*'、'get*'、'handle*'、'*Order'等
propagation	用于指定事务的传播行为，其属性值就是表 5-1 中的值，它的默认值为 REQUIRED
isolation	该属性用于指定事务的隔离级别，其属性值可以为 DEFAULT、READ_UNCOMMITTED、READ_COMMITTED、REPEATABLE_READ 和 SERIALIZABLE，其默认值为 DEFAULT
read-only	该属性用于指定事务是否只读，其默认值为 false
timeout	该属性用于指定事务超时的时间，其默认值为-1，即永不超时
rollback-for	该属性用于指定触发事务回滚的异常类，在指定多个异常类时，异常类之间以英文逗号分隔
no-rollback-for	该属性用于指定不触发事务回滚的异常类，在指定多个异常类时，异常类之间以英文逗号分隔

了解了如何在 XML 文件中配置事务后，接下来通过一个案例来演示如何通过 XML 方式来实现 Spring 的声明式事务管理。本案例以上一章的项目代码和数据表为基础，编写一个模拟银行转账的程序，要求在转账时通过 Spring 对事务进行控制，其具体实现步骤如下。

（1）在 Eclipse 中，创建一个名为 chapter05 的 Web 项目，在项目的 lib 目录中导入 chapter04 项目中的所有 JAR 包，并将 AOP 所需 JAR 包也导入到 lib 目录中。导入后的 lib 目录如图 5-3 所示。

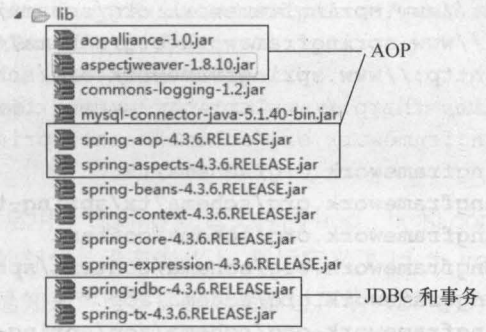


图5-3 项目所需JAR包

（2）将 chapter04 项目中的代码和配置文件复制到 chapter05 项目的 src 目录下，并在 AccountDao 接口中，创建一个转账方法 transfer()，其代码如下所示。

```
// 转账
public void transfer(String outUser,String inUser,Double money);
```

（3）在其实现类 AccountDaoImpl 中实现 transfer()方法，编辑后的代码如下所示。

```
/**
 * 转账
 * inUser: 收款人
 * outUser: 汇款人
 * money: 收款金额
*/
public void transfer(String outUser, String inUser, Double money) {
    // 收款时，收款用户的余额=现有余额+所汇金额
    this.jdbcTemplate.update("update account set balance = balance +? "
            + "where username = ?",money, inUser);
    // 模拟系统运行时的突发性问题
    int i = 1/0;
    // 汇款时，汇款用户的余额=现有余额-所汇金额
    this.jdbcTemplate.update("update account set balance = balance-? "
            + "where username = ?",money, outUser);
}
```

在上述代码中，使用了两个 update()方法对 account 表中的数据执行收款和汇款的更新操作。在两个操作之间，添加了一行代码“int i = 1/0;”来模拟系统运行时的突发性问题。如果没有事务控制，那么在转账操作执行后，收款用户的余额会增加，而汇款用户的余额会因为系统出现问题而不变，这显然是有问题的；如果增加了事务控制，那么在转账操作执行后，收款用户的余额和汇款用户的余额在问题出现前后都应该保持不变。

（4）修改配置文件 applicationContext.xml，添加命名空间并编写事务管理的相关配置代码，如文件 5-1 所示。

文件 5-1 applicationContext.xml

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

```
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-4.3.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.3.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-4.3.xsd">
    <!-- 1.配置数据源 -->
    <bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <!--数据库驱动 -->
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <!--连接数据库的 url -->
        <property name="url" value="jdbc:mysql://localhost/spring" />
        <!--连接数据库的用户名 -->
        <property name="username" value="root" />
        <!--连接数据库的密码 -->
        <property name="password" value="root" />
    </bean>
    <!-- 2.配置 JDBC 模板 -->
    <bean id="jdbcTemplate"
         class="org.springframework.jdbc.core.JdbcTemplate">
         <!-- 默认必须使用数据源 -->
         <property name="dataSource" ref="dataSource" />
    </bean>
    <!--3.定义 id 为 accountDao 的 Bean -->
    <bean id="accountDao" class="com.itheima.jdbc.AccountDaoImpl">
         <!-- 将 jdbcTemplate 注入到 AccountDao 实例中 -->
         <property name="jdbcTemplate" ref="jdbcTemplate" />
    </bean>
    <!-- 4.事务管理器，依赖于数据源 -->
    <bean id="transactionManager" class=
    "org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>
    <!-- 5.编写通知：对事务进行增强(通知)，需要编写对切入点和具体执行事务细节 -->
    <tx:advice id="txAdvice" transaction-manager="transactionManager">
        <tx:attributes>
            <!-- name：*表示任意方法名称 -->
            <tx:method name="*" propagation="REQUIRED"
                       isolation="DEFAULT" read-only="false" />
        </tx:attributes>
     </tx:advice>
     <!-- 6.编写 aop，让 spring 自动对目标生成代理，需要使用 AspectJ 的表达式 -->
     <aop:config>
        <!-- 切入点 -->
```

```
54          <aop:pointcut expression="execution(* com.itheima.jdbc.*.*(..))"
55              id="txPointCut" />
56          <!-- 切面：将切入点与通知整合 -->
57          <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointCut" />
58      </aop:config>
59  </beans>
```

在文件 5-1 中，首先启用了 Spring 配置文件的 aop、tx 和 context 3 个命名空间（从配置数据源到声明事务管理器的部分都没有变化），然后定义了 id 为 transactionManager 的事务管理器，接下来通过编写的通知来声明事务，最后通过声明 AOP 的方式让 Spring 自动生成代理。

（5）在 com.itheima.jdbc 包中，创建测试类 TransactionTest，并在类中编写测试方法 xmlTest()，如文件 5-2 所示。

文件 5-2　TransactionTest.java

```
1  package com.itheima.jdbc;
2  import org.junit.Test;
3  import org.springframework.context.ApplicationContext;
4  import
5  org.springframework.context.support.ClassPathXmlApplicationContext;
6  //测试类
7  public class TransactionTest {
8      @Test
9      public void xmlTest(){
10         ApplicationContext applicationContext =
11             new ClassPathXmlApplicationContext("applicationContext.xml");
12         // 获取 AccountDao 实例
13         AccountDao accountDao =
14             (AccountDao)applicationContext.getBean("accountDao");
15         // 调用实例中的转账方法
16         accountDao.transfer("Jack", "Rose", 100.0);
17         // 输出提示信息
18         System.out.println("转账成功！");
19     }
20 }
```

在文件 5-2 中，获取了 AccountDao 实例后，调用了实例中的转账方法，由 Jack 向 Rose 的账户中转入 100 元。如果在配置文件中所声明的事务代码能够起作用，那么在整个转账方法执行完毕后，Jack 和 Rose 的账户余额应该都是原来的数值。

在执行转账操作前，先查看 account 表中的数据，如图 5-4 所示。

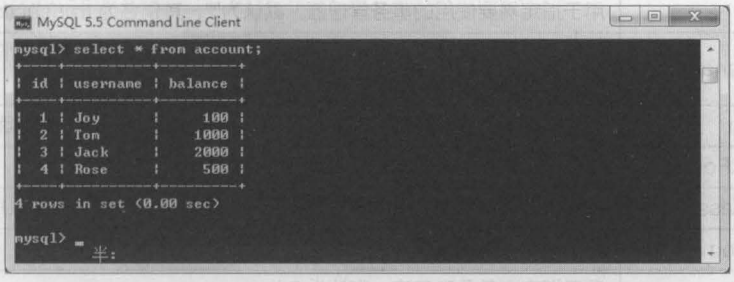


图5-4　account表

从图 5-4 可以看出，此时 Jack 的账户余额是 2000，而 Rose 的账户余额是 500。执行完文件 5-2 中的测试方法后，Junit 的控制台的显示结果如图 5-5 所示。

图5-5　运行结果

从图 5-5 可以看到，Junit 控制台中报出了“/by zero”的算术异常信息。此时如果再次查询数据表 account，会发现表中 Jack 和 Rose 的账户余额并没有发生任何变化（与图 5-4 中的显示结果一样），这说明 Spring 中的事务管理配置已经生效。

5.2.2　基于 Annotation 方式的声明式事务

Spring 的声明式事务管理还可以通过 Annotation（注解）的方式来实现。这种方式的使用非常简单，开发者只需做两件事情：

① 在 Spring 容器中注册事务注解驱动，其代码如下。

```
<tx:annotation-driven transaction-manager="transactionManager"/>
```

② 在需要使用事务的 Spring Bean 类或者 Bean 类的方法上添加注解@Transactional。如果将注解添加在 Bean 类上，则表示事务的设置对整个 Bean 类的所有方法都起作用；如果将注解添加在 Bean 类中的某个方法上，则表示事务的设置只对该方法有效。

使用@Transactional 注解时，可以通过其参数配置事务详情。@Transactional 注解可配置的参数信息如表 5-3 所示。

表 5-3　@Transactional 注解的参数及其描述

参数名称	描述
value	用于指定需要使用的事务管理器，默认为""，其别名为 transactionManager
transactionManager	指定事务的限定符值，可用于确定目标事务管理器，匹配特定的限定值（或者 Bean 的 name 值），默认为""，其别名为 value
isolation	用于指定事务的隔离级别，默认为 Isolation.DEFAULT（即底层事务的隔离级别）
noRollbackFor	用于指定遇到特定异常时强制不回滚事务
noRollbackForClassName	用于指定遇到特定的多个异常时强制不回滚事务。其属性值可以指定多个异常类名
propagation	用于指定事务的传播行为，默认为 Propagation.REQUIRED
read-only	用于指定事务是否只读，默认为 false

续表

参数名称	描述
rollbackFor	用于指定遇到特定异常时强制回滚事务
rollbackForClassName	用于指定遇到特定的多个异常时强制回滚事务。其属性值可以指定多个异常类名
timeout	用于指定事务的超时时长，默认为 TransactionDefinition.TIMEOUT_DEFAULT（即底层事务系统的默认时间）

从表 5-3 可以看出，@Transactional 注解与<tx:method>元素中的事务属性基本是对应的，并且其含义也基本相似。

为了让读者更加清楚地掌握@Transactional 注解的使用，接下来对上一小节的案例进行修改，以 Annotation 方式来实现项目中的事务管理，具体实现步骤如下。

（1）在 src 目录下，创建一个 Spring 配置文件 applicationContext-annotation.xml，在该文件中声明事务管理器等配置信息，如文件 5-3 所示。

文件 5-3 applicationContext-annotation.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-4.3.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.3.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-4.3.xsd">
    <!-- 1.配置数据源 -->
    <bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <!--数据库驱动 -->
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <!--连接数据库的 url -->
        <property name="url" value="jdbc:mysql://localhost/spring" />
        <!--连接数据库的用户名 -->
        <property name="username" value="root" />
        <!--连接数据库的密码 -->
        <property name="password" value="root" />
    </bean>
    <!-- 2.配置 JDBC 模板 -->
    <bean id="jdbcTemplate"
          class="org.springframework.jdbc.core.JdbcTemplate">
        <!-- 默认必须使用数据源 -->
        <property name="dataSource" ref="dataSource" />
    </bean>
```

```
    <!--3.定义 id 为 accountDao 的 Bean -->
    <bean id="accountDao" class="com.itheima.jdbc.AccountDaoImpl">
        <!-- 将 jdbcTemplate 注入到 AccountDao 实例中 -->
        <property name="jdbcTemplate" ref="jdbcTemplate" />
    </bean>
    <!-- 4.事务管理器，依赖于数据源 -->
    <bean id="transactionManager" class=
    "org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>
    <!-- 5.注册事务管理器驱动 -->
    <tx:annotation-driven transaction-manager="transactionManager"/>
</beans>
```

与基于 XML 方式的配置文件相比，文件 5-3 通过注册事务管理器驱动，替换了文件 5-1 中的第 5 步编写通知和第 6 步编写 aop，这样大大减少了配置文件中的代码量。

需要注意的是，如果案例中使用了注解式开发，则需要在配置文件中开启注解处理器，指定扫描哪些包下的注解。这里没有开启注解处理器是因为在配置文件中已经配置了 AccountDaoImpl 类的 Bean，而@Transactional 注解就配置在该 Bean 类中，所以可以直接生效。

（2）在 AccountDaoImpl 类的 transfer()方法上添加事务注解，添加后的代码如下所示。

```
@Transactional(propagation = Propagation.REQUIRED,
        isolation = Isolation.DEFAULT, readOnly = false)
public void transfer(String outUser, String inUser, Double money) {
    // 收款时，收款用户的余额=现有余额+所汇金额
    this.jdbcTemplate.update("update account set balance = balance +? "
        + "where username = ?",money, inUser);
    // 模拟系统运行时的突发性问题
    int i = 1/0;
    // 汇款时，汇款用户的余额=现有余额-所汇金额
    this.jdbcTemplate.update("update account set balance = balance-? "
        + "where username = ?",money, outUser);
}
```

上述方法已经添加了@Transactional 注解，并且使用注解的参数配置了事务详情，各个参数之间要用英文逗号“,”进行分隔。

小提示

在实际开发中，事务的配置信息通常是在 Spring 的配置文件中完成的，而在业务层类上只需使用@Transactional 注解即可，不需要配置@Transactional 注解的属性。

（3）在 TransactionTest 类中，创建测试方法 annotationTest()，编辑后的代码如下所示。

```
@Test
public void annotationTest(){
    ApplicationContext applicationContext =
new ClassPathXmlApplicationContext("applicationContext-annotation.xml");
    // 获取 AccountDao 实例
    AccountDao accountDao =
```

```
        (AccountDao)applicationContext.getBean("accountDao");
    // 调用实例中的转账方法
    accountDao.transfer("Jack", "Rose", 100.0);
    // 输出提示信息
    System.out.println("转账成功！");
}
```

从上述代码可以看出，与 XML 方式的测试方法相比，该方法只是对配置文件的名称进行了修改。程序执行后，会出现与 XML 方式同样的执行结果，这里就不再做重复演示，读者可自行测试。

5.3 本章小结

本章主要对 Spring 中的事务管理进行了详细讲解。首先讲解了 Spring 事务管理所涉及的 3 个核心接口，然后对 Spring 中事务管理的两种方式进行了介绍，最后通过案例分别对基于 XML 方式和基于 Annotation 方式的声明式事务处理的使用进行了详细讲解。通过本章的学习，读者可以对 Spring 的事务管理知识有一定的了解，并能够掌握 Spring 声明式事务管理的使用。

【思考题】

1. 请简述 Spring 中事务管理的两种方式。
2. 请简述如何使用 Annotation 方式进行声明式事务管理。

Java EE

6 Chapter

第 6 章 初识 MyBatis

学习目标

- 了解 MyBatis 的基础知识
- 熟悉 MyBatis 的工作原理
- 掌握 MyBatis 入门程序的编写

MyBatis 是当前主流的 Java 持久层框架之一，它与 Hibernate 一样，也是一种 ORM 框架。因其性能优异，且具有高度的灵活性、可优化性和易于维护等特点，所以受到了广大互联网企业的青睐，是目前大型互联网项目的首选框架。从本章开始，我们将对 MyBatis 框架的相关知识进行详细讲解。

6.1 什么是 MyBatis

MyBatis（前身是 iBatis）是一个支持普通 SQL 查询、存储过程以及高级映射的持久层框架，它消除了几乎所有的 JDBC 代码和参数的手动设置以及对结果集的检索，并使用简单的 XML 或注解进行配置和原始映射，用以将接口和 Java 的 POJO（Plain Old Java Object，普通 Java 对象）映射成数据库中的记录，使得 Java 开发人员可以使用面向对象的编程思想来操作数据库。

MyBatis 框架也被称之为 ORM（Object/Relational Mapping，即对象关系映射）框架。所谓的 ORM 就是一种为了解决面向对象与关系型数据库中数据类型不匹配的技术，它通过描述 Java 对象与数据库表之间的映射关系，自动将 Java 应用程序中的对象持久化到关系型数据库的表中。ORM 框架的工作原理如图 6-1 所示。

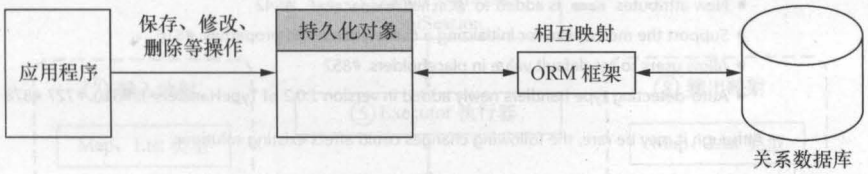


图6-1 ORM框架的工作原理

从图 6-1 可以看出，使用 ORM 框架后，应用程序不再直接访问底层数据库，而是以面向对象的方式来操作持久化对象（Persisent Object，PO），而 ORM 框架则会通过映射关系将这些面向对象的操作转换成底层的 SQL 操作。

当前的 ORM 框架产品有很多，常见的 ORM 框架有 Hibernate 和 MyBatis。这两个框架的主要区别如下。

- Hibernate：是一个全表映射的框架。通常开发者只需定义好持久化对象到数据库表的映射关系，就可以通过 Hibernate 提供的方法完成持久层操作。开发者并不需要熟练地掌握 SQL 语句的编写，Hibernate 会根据制定的存储逻辑，自动的生成对应的 SQL，并调用 JDBC 接口来执行，所以其开发效率会高于 MyBatis。然而 Hibernate 自身也存在着一些缺点，例如它在多表关联时，对 SQL 查询的支持较差；更新数据时，需要发送所有字段；不支持存储过程；不能通过优化 SQL 来优化性能等。这些问题导致其只适合在场景不太复杂且对性能要求不高的项目中使用。

- MyBatis：是一个半自动映射的框架。这里所谓的“半自动”是相对于 Hibernate 全表映射而言的，MyBatis 需要手动匹配提供 POJO、SQL 和映射关系，而 Hibernate 只需提供 POJO 和映射关系即可。与 Hibernate 相比，虽然使用 MyBatis 手动编写 SQL 要比使用 Hibernate 的工作量大，但 MyBatis 可以配置动态 SQL 并优化 SQL，可以通过配置决定 SQL 的映射规则，它还支持存储过程等。对于一些复杂的和需要优化性能的项目来说，显然使用 MyBatis 更加合适。

6.2 MyBatis 的下载和使用

在本书编写时，MyBatis 的最新版本是 mybatis-3.4.2，本书所讲解的 MyBatis 框架就是基于此版本的，所以希望读者也下载同样的版本，以便于学习。

该版本的 MyBatis 可以通过网址“https://github.com/mybatis/mybatis-3/releases”下载得到。在浏览器中输入网址后，可以在页面中看到 MyBatis 对应的下载链接，如图 6-2 所示。

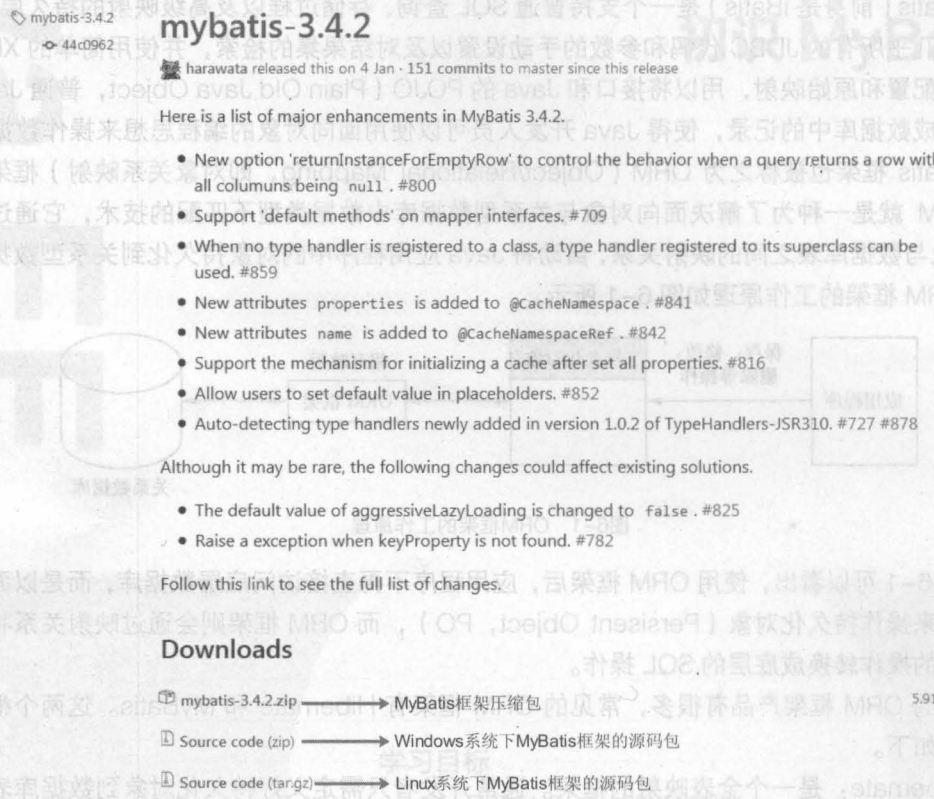


图6-2 MyBatis的下载

从图 6-2 可以看出，页面中包含了 3 个下载链接，这里我们只需下载 mybatis-3.4.2.zip 即可。下载并解压 mybatis-3.4.2.zip 压缩包，会得到一个名为 mybatis-3.4.2 的文件夹，该文件夹下包含的文件如图 6-3 所示。

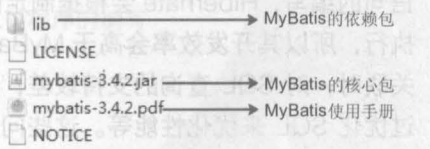


图6-3 MyBatis压缩包内的文件结构

使用 MyBatis 框架非常简单，只需在应用程序中引入 MyBatis 的核心包和 lib 目录中的依赖包即可。

注意

如果底层采用的是 MySQL 数据库，那么还需要将 MySQL 数据库的驱动 JAR 包添加到应用程序的类路径中；如果采用其他类型的数据库，则同样需要将对应类型的数据库驱动包添加到应用程序的类路径中。

6.3 MyBatis 的工作原理

为了使读者能够更加清晰地理解 MyBatis 程序，在正式讲解 MyBatis 入门案例之前，先来了解一下 MyBatis 程序的工作原理，如图 6-4 所示。

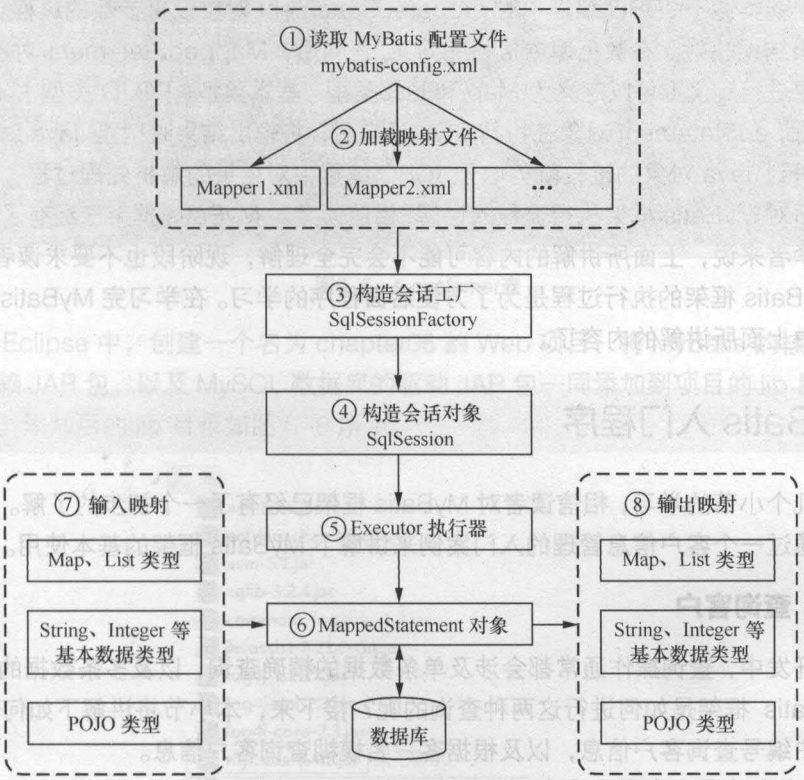


图6-4 MyBatis框架执行流程图

从图 6-4 可以看出，MyBatis 框架在操作数据库时，大体经过了 8 个步骤。下面就对图 6-4 中的每一步流程进行详细讲解，具体如下。

（1）读取 MyBatis 配置文件 mybatis-config.xml。mybatis-config.xml 作为 MyBatis 的全局配置文件，配置了 MyBatis 的运行环境等信息，其中主要内容是获取数据库连接。

（2）加载映射文件 Mapper.xml。Mapper.xml 文件即 SQL 映射文件，该文件中配置了操作数据库的 SQL 语句，需要在 mybatis-config.xml 中加载才能执行。mybatis-config.xml 可以加载多个配置文件，每个配置文件对应数据库中的一张表。

（3）构建会话工厂。通过 MyBatis 的环境等配置信息构建会话工厂 SqlSessionFactory。

（4）创建 SqlSession 对象。由会话工厂创建 SqlSession 对象，该对象中包含了执行 SQL 的所有方法。

（5）MyBatis 底层定义了一个 Executor 接口来操作数据库，它会根据 SqlSession 传递的参数动态地生成需要执行的 SQL 语句，同时负责查询缓存的维护。

（6）在 Executor 接口的执行方法中，包含一个 MappedStatement 类型的参数，该参数是

对映射信息的封装，用于存储要映射的 SQL 语句的 id、参数等。Mapper.xml 文件中一个 SQL 对应一个 MappedStatement 对象，SQL 的 id 即是 MappedStatement 的 id。

（7）输入参数映射。在执行方法时，MappedStatement 对象会对用户执行 SQL 语句的输入参数进行定义（可以定义为 Map、List 类型、基本类型和 POJO 类型），Executor 执行器会通过 MappedStatement 对象在执行 SQL 前，将输入的 Java 对象映射到 SQL 语句中。这里对输入参数的映射过程就类似于 JDBC 编程中对 preparedStatement 对象设置参数的过程。

（8）输出结果映射。在数据库中执行完 SQL 语句后，MappedStatement 对象会对 SQL 执行输出的结果进行定义（可以定义为 Map 和 List 类型、基本类型、POJO 类型），Executor 执行器会通过 MappedStatement 对象在执行 SQL 语句后，将输出结果映射至 Java 对象中。这种将输出结果映射到 Java 对象的过程就类似于 JDBC 编程中对结果的解析处理过程。

通过上面对 MyBatis 框架执行流程的讲解，相信读者对 MyBatis 框架已经有了一个初步的了解。对于初学者来说，上面所讲解的内容可能不会完全理解，现阶段也不要求读者能完全理解，这里讲解 MyBatis 框架的执行过程是为了方便后面程序的学习。在学习完 MyBatis 框架后，读者自然就会明白上面所讲解的内容了。

6.4 MyBatis 入门程序

通过前几个小节的学习，相信读者对 MyBatis 框架已经有了一个初步的了解。接下来的几个小节中，将通过一个客户信息管理的入门案例来讲解下 MyBatis 框架的基本使用。

6.4.1 查询客户

在实际开发中，查询操作通常都会涉及单条数据的精确查询，以及多条数据的模糊查询。那么使用 MyBatis 框架是如何进行这两种查询的呢？接下来，本小节将讲解下如何使用 MyBatis 框架根据客户编号查询客户信息，以及根据客户名模糊查询客户信息。

1. 根据客户编号查询客户信息

根据客户编号查询客户信息主要是通过查询客户表中的主键（这里表示唯一的客户编号）来实现的，其具体实现步骤如下。

（1）在 MySQL 数据库中，创建一个名为 mybatis 的数据库，在此数据库中创建一个 t_customer 表，同时预先插入几条数据。此操作所执行的 SQL 语句如下所示。

```
# 创建一个名称为 mybatis 的数据库
CREATE DATABASE mybatis;
# 使用 mybatis 数据库
USE mybatis;
# 创建一个名称为 t_customer 的表
CREATE TABLE t_customer (
    id int(32) PRIMARY KEY AUTO_INCREMENT,
    username varchar(50),
    jobs varchar(50),
    phone varchar(16)
);
# 插入 3 条数据
```

```
INSERT INTO t_customer VALUES ('1', 'joy', 'doctor', '13745874578');
INSERT INTO t_customer VALUES ('2', 'jack', 'teacher', '13521210112');
INSERT INTO t_customer VALUES ('3', 'tom', 'worker', '15179405961');
```

完成上述操作后，数据库 t_customer 表中的数据如图 6-5 所示。

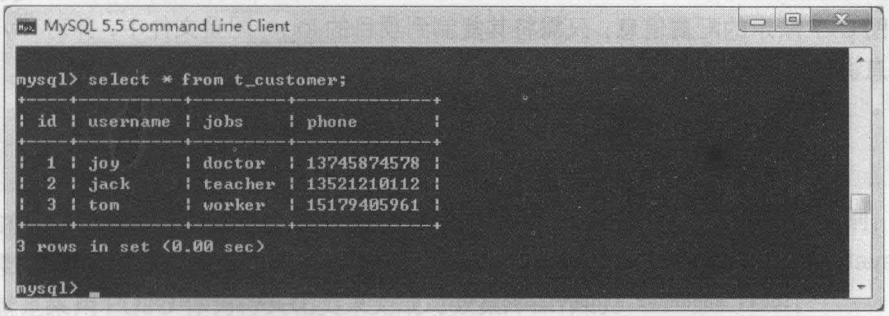


图6-5 t_customer表

（2）在 Eclipse 中，创建一个名为 chapter06 的 Web 项目，将 MyBatis 的核心 JAR 包、lib 目录中的依赖 JAR 包，以及 MySQL 数据库的驱动 JAR 包一同添加到项目的 lib 目录下，并发布到类路径中。添加后的 lib 目录如图 6-6 所示。

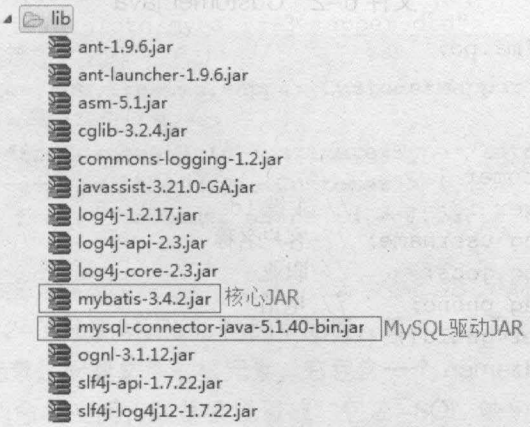


图6-6 MyBatis相关JAR包

（3）由于 MyBatis 默认使用 log4j 输出日志信息，所以如果要查看控制台的输出 SQL 语句，那么就需要在 classpath 路径下配置其日志文件。在项目的 src 目录下创建 log4j.properties 文件，编辑后的内容如文件 6-1 所示。

文件 6-1 log4j.properties

```
1  # Global logging configuration
2  log4j.rootLogger=ERROR, stdout
3  # MyBatis logging configuration...
4  log4j.logger.com.itheima=DEBUG
5  # Console output...
6  log4j.appender.stdout=org.apache.log4j.ConsoleAppender
7  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
8  log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```

在文件 6-1 中，包含了全局的日志配置、MyBatis 的日志配置和控制台输出，其中 MyBatis 的日志配置用于将 com.itheima 包下所有类的日志记录级别设置为 DEBUG。

*由于 log4j 文件中的具体内容已经超出了本书范围，所以这里不过多讲解，读者可自行查找资料学习。上述配置文件代码也不需要读者全部手写，在 MyBatis 使用手册中的 Logging 小节，*可以找到图 6-7 所示的配置信息，只需将其复制到项目的 log4j 配置文件中，并对 MyBatis 的日志配置信息进行简单修改即可使用。

Create a file called `log4j.properties` as shown below and place it in your classpath:

```
# Global logging configuration
log4j.rootLogger=ERROR, stdout
# MyBatis logging configuration...
log4j.logger.org.mybatis.example.BlogMapper=TRACE
# Console output...
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```

图6-7　MyBatis使用手册中的Logging配置

（4）在 src 目录下，创建一个 com.itheima.po 包，在该包下创建持久化类 Customer，并在类中声明 id、username、jobs 和 phone 属性，及其对应的 getter/setter 方法，如文件 6-2 所示。

文件 6-2　Customer.java

```
1  package com.itheima.po;
2  /**
3   * 客户持久化类
4   */
5  public class Customer {
6      private Integer id;       // 主键 id
7      private String username; // 客户名称
8      private String jobs;      // 职业
9      private String phone;     // 电话
10     public Integer getId() {
11         return id;
12     }
13     public void setId(Integer id) {
14         this.id = id;
15     }
16     public String getUsername() {
17         return username;
18     }
19     public void setUsername(String username) {
20         this.username = username;
21     }
22     public String getJobs() {
23         return jobs;
24     }
25     public void setJobs(String jobs) {
26         this.jobs = jobs;
27     }
28     public String getPhone() {
```

```
29          return phone;
30      }
31      public void setPhone(String phone) {
32          this.phone = phone;
33      }
34      @Override
35      public String toString() {
36          return "Customer [id=" + id + ", username=" + username +
37                          ", jobs=" + jobs + ", phone=" + phone + "]";
38      }
39  }
```

从上述代码可以看出，持久化类 Customer 与普通的 JavaBean 并没有什么区别，只是其属性字段与数据库中的表字段相对应。实际上，Customer 就是一个 POJO（普通 Java 对象），MyBatis 就是采用 POJO 作为持久化类来完成对数据库操作的。

（5）在 src 目录下，创建一个 com.itheima.mapper 包，并在包中创建映射文件 CustomerMapper.xml，编辑后如文件 6-3 所示。

文件 6-3　CustomerMapper.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
3      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
4  <!-- namespace 表示命名空间 -->
5  <mapper namespace="com.itheima.mapper.CustomerMapper">
6      <!--根据客户编号获取客户信息 -->
7      <select id="findCustomerById" parameterType="Integer"
8          resultType="com.itheima.po.Customer">
9          select * from t_customer where id = #{id}
10     </select>
11 </mapper>
```

在文件 6-3 中，第 2~3 行是 MyBatis 的约束配置，第 5~11 行是需要程序员编写的映射信息。其中，<mapper>元素是配置文件的根元素，它包含一个 namespace 属性，该属性为这个<mapper>指定了唯一的命名空间，通常会设置成“包名+SQL 映射文件名”的形式。子元素<select>中的信息是用于执行查询操作的配置，其 id 属性是<select>元素在映射文件中的唯一标识；parameterType 属性用于指定传入参数的类型，这里表示传递给执行 SQL 的是一个 Integer 类型的参数；resultType 属性用于指定返回结果的类型，这里表示返回的数据是 Customer 类型。在定义的查询 SQL 语句中，“#{}”用于表示一个占位符，相当于“?”，而“#{id}”表示该占位符待接收参数的名称为 id。

多学一招：快速获取配置文件的约束信息

在 MyBatis 的映射文件中，包含了一些约束信息，初学者如果自己动手去编写，不但浪费时间，还容易出错。其实，在 MyBatis 使用手册中，就可以找到这些约束信息，具体的获取方法如下。

打开 MyBatis 的使用手册 mybatis-3.4.2.pdf，在第 2 小节 Getting started（入门指南）下的 2.1.5 小节 Exploring Mapped SQL Statements 中，即可找到映射文件的约束信息，如图 6-8 所示。

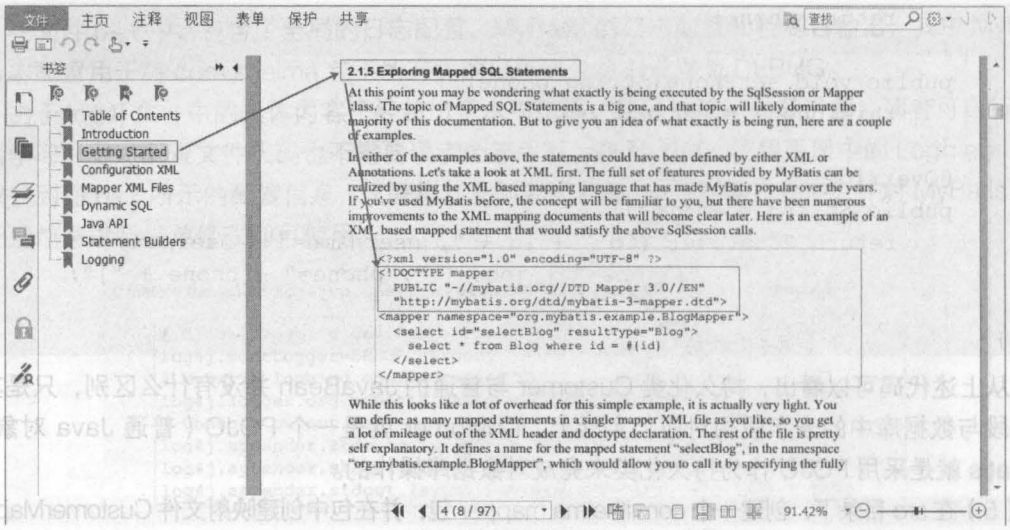


图6-8 映射文件的约束信息

在图 6-8 中，方框处标注的文件信息就是 MyBatis 映射文件的约束信息。初学者只需将信息复制到项目创建的 XML 文件中即可。

（6）在 src 目录下，创建 MyBatis 的核心配置文件 mybatis-config.xml，编辑后如文件 6-4 所示。

文件 6-4 mybatis-config.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <!--1.配置环境 ，默认的环境 id 为 mysql-->
    <environments default="mysql">
        <!--1.2.配置 id 为 mysql 的数据库环境 -->
        <environment id="mysql">
            <!-- 使用 JDBC 的事务管理 -->
            <transactionManager type="JDBC" />
            <!--数据库连接池 -->
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver" />
                <property name="url"
                          value="jdbc:mysql://localhost:3306/mybatis" />
                <property name="username" value="root" />
                <property name="password" value="root" />
            </dataSource>
        </environment>
    </environments>
    <!--2.配置 Mapper 的位置 -->
    <mappers>
        <mapper resource="com/itheima/mapper/CustomerMapper.xml" />
    </mappers>
</configuration>
```

在文件6-3中，第2~3行是MyBatis的配置文件的约束信息，下面<configuration>元素中的内容就是开发人员需要编写的配置信息。这里按照<configuration>子元素的功能不同，将配置分为了两个步骤：第1步配置了环境，第2步配置了Mapper的位置。关于上述代码中各个元素的详细配置信息将在下一章进行详细讲解，此案例中读者只需要按照上述代码配置即可。

小提示

上述配置文件同样不需要读者完全手动编写。在MyBatis使用手册mybatis-3.4.2.pdf的2.1.2小节中，已经给出了配置模板（包含约束信息），读者只需要复制过来，依照自己的项目需求修改即可。

（7）在src目录下，创建一个com.itheima.test包，在该包下创建测试类MybatisTest，并在类中编写测试方法findCustomerByIdTest()，如文件6-5所示。

文件6-5 MybatisTest.java

```
1  package com.itheima.test;
2  import java.io.InputStream;
3  import org.apache.ibatis.io.Resources;
4  import org.apache.ibatis.session.SqlSession;
5  import org.apache.ibatis.session.SqlSessionFactory;
6  import org.apache.ibatis.session.SqlSessionFactoryBuilder;
7  import org.junit.Test;
8  import com.itheima.po.Customer;
9  /**
10  * 入门程序测试类
11  */
12 public class MybatisTest {
13     /**
14      * 根据客户编号查询客户信息
15      */
16     @Test
17     public void findCustomerByIdTest() throws Exception {
18         // 1. 读取配置文件
19         String resource = "mybatis-config.xml";
20         InputStream inputStream =
21                 Resources.getResourceAsStream(resource);
22         // 2. 根据配置文件构建 SqlSessionFactory
23         SqlSessionFactory sqlSessionFactory =
24                 new SqlSessionFactoryBuilder().build(inputStream);
25         // 3. 通过 SqlSessionFactory 创建 SqlSession
26         SqlSession sqlSession = sqlSessionFactory.openSession();
27         // 4. SqlSession 执行映射文件中定义的 SQL，并返回映射结果
28         Customer customer = sqlSession.selectOne("com.itheima.mapper"
29                 + ".CustomerMapper.findCustomerById", 1);
30         // 打印输出结果
31         System.out.println(customer.toString());
32         // 5、关闭 SqlSession
33         sqlSession.close();
34     }
35 }
```

在文件 6-5 的 findCustomerByIdTest()方法中，首先通过输入流读取了配置文件，然后根据配置文件构建了 SqlSessionFactory 对象。接下来通过 SqlSessionFactory 对象又创建了 SqlSession 对象，并通过 SqlSession 对象的 selectOne()方法执行查询操作。selectOne()方法的第 1 个参数表示映射 SQL 的标识字符串，它由 CustomerMapper.xml 中<mapper>元素的 namespace 属性值+<select>元素的 id 属性值组成；第 2 个参数表示查询所需要的参数，这里查询的是客户表中 id 为 1 的客户。为了查看查询结果，这里使用了输出语句输出查询结果信息。最后，程序执行完毕时，关闭了 SqlSession。

使用 JUnit4 测试执行 findCustomerByIdTest()方法后，控制台的输出结果如图 6-9 所示。

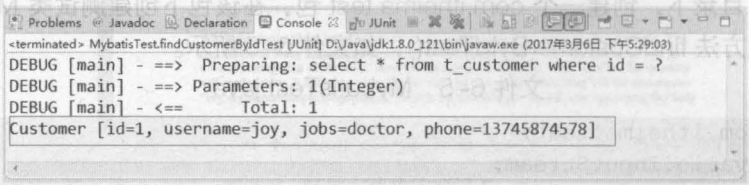


图6-9 运行结果

从图 6-9 可以看出，使用 MyBatis 框架已经成功查询出了 id 为 1 的客户信息。

2. 根据客户名模糊查询客户信息

了解了如何使用 MyBatis 根据客户编号查询客户信息后，接下来讲解下如何根据客户的名称来模糊查询相关的客户信息。

模糊查询的实现非常简单，只需要在映射文件中通过<select>元素编写相应的 SQL 语句，并通过 SqlSession 的查询方法执行该 SQL 即可。其具体实现步骤如下。

（1）在映射文件 CustomerMapper.xml 中，添加根据客户名模糊查询客户信息列表的 SQL 语句，具体实现代码如下。

```
<!--根据客户名模糊查询客户信息列表-->
<select id="findCustomerByName" parameterType="String"
    resultType="com.itheima.po.Customer">
    select * from t_customer where username like '%${value}%'
</select>
```

与根据客户编号查询相比，上述配置代码中的属性 id、parameterType 和 SQL 语句都发生相应变化。其中，SQL 语句中的“${}”用来表示拼接 SQL 的字符串，即不加解释的原样输出。“${value}”表示要拼接的是简单类型参数。

在使用“${}”进行 SQL 字符串拼接时，无法防止 SQL 注入问题。所以想要既能实现模糊查询，又要防止 SQL 注入，可以对上述映射文件 CustomerMapper.xml 中模糊查询的 select 语句进行修改，使用 MySQL 中的 concat()函数进行字符串拼接。具体修改示例如下所示。

```
select * from t_customer where username like concat('%',#{value},'%')
```

（2）在测试类 MybatisTest 中，添加一个测试方法 findCustomerByNameTest()，其代码如下所示。

```
/**
 * 根据用户名称来模糊查询用户信息列表
 */
@Test
public void findCustomerByNameTest() throws Exception{
    // 1. 读取配置文件
    String resource = "mybatis-config.xml";
    InputStream inputStream = Resources.getResourceAsStream(resource);
    // 2. 根据配置文件构建 SqlSessionFactory
    SqlSessionFactory sqlSessionFactory =
            new SqlSessionFactoryBuilder().build(inputStream);
    // 3. 通过 SqlSessionFactory 创建 SqlSession
    SqlSession sqlSession = sqlSessionFactory.openSession();
    // 4. SqlSession 执行映射文件中定义的 SQL，并返回映射结果
    List<Customer> customers = sqlSession.selectList("com.itheima.mapper"
            + ".CustomerMapper.findCustomerByName", "j");
    for (Customer customer : customers) {
        //打印输出结果集
        System.out.println(customer);
    }
    // 5. 关闭 SqlSession
    sqlSession.close();
}
```

从上述代码可以看出，findCustomerByNameTest()方法只是在第 4 步时与根据客户编号查询的测试方法有所不同，其他步骤都一致。在第 4 步时，由于可能查询出的是多条数据，所以调用的是 SqlSession 的 selectList()方法来查询返回结果的集合对象，并使用 for 循环输出结果集对象。

使用 JUnit4 执行 findCustomerByNameTest()方法后，控制台的输出结果如图 6-10 所示。

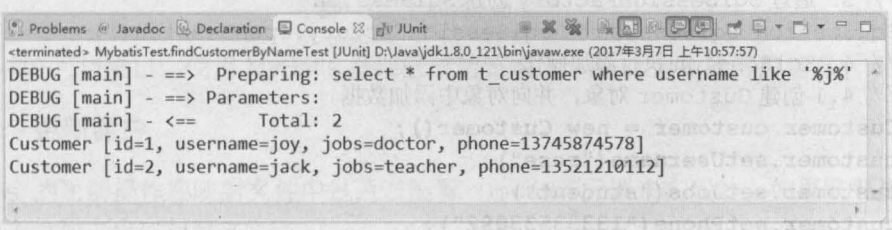


图6-10 运行结果

从图 6-10 可以看出，使用 MyBatis 框架已成功查询出了客户表中客户名称中带有“j”的两条客户信息。

至此，MyBatis 入门程序的查询功能就已经讲解完成。从上面两个查询方法中可以发现，MyBatis 的操作大致可分为以下几个步骤。

（1）读取配置文件。

（2）根据配置文件构建 SqlSessionFactory。

（3）通过 SqlSessionFactory 创建 SqlSession。

（4）使用 SqlSession 对象操作数据库（包括查询、添加、修改、删除以及提交事务等）。

（5）关闭 SqlSession。

6.4.2 添加客户

在 MyBatis 的映射文件中，添加操作是通过<insert>元素来实现的。例如，向数据库中的 t_customer 表中插入一条数据可以通过如下配置来实现。

```
<!-- 添加客户信息 -->
<insert id="addCustomer" parameterType="com.itheima.po.Customer">
    insert into t_customer(username,jobs,phone)
    values(#{username},#{jobs},#{phone})
</insert>
```

在上述配置代码中，传入的参数是一个 Customer 类型，该类型的参数对象被传递到语句中时，#{username}会查找参数对象 Customer 的 username 属性（#{jobs}和#{phone}也是一样），并将其属性值传入到 SQL 语句中。为了验证上述配置是否正确，下面编写一个测试方法来执行添加操作。

在测试类 MybatisTest 中，添加测试方法 addCustomerTest()，其代码如下所示。

```
/**
 * 添加客户
 */
@Test
public void addCustomerTest() throws Exception{
    // 1. 读取配置文件
    String resource = "mybatis-config.xml";
    InputStream inputStream = Resources.getResourceAsStream(resource);
    // 2. 根据配置文件构建 SqlSessionFactory
    SqlSessionFactory sqlSessionFactory =
            new SqlSessionFactoryBuilder().build(inputStream);
    // 3. 通过 SqlSessionFactory 创建 SqlSession
    SqlSession sqlSession = sqlSessionFactory.openSession();
    // 4. SqlSession 执行添加操作
    // 4.1 创建 Customer 对象，并向对象中添加数据
    Customer customer = new Customer();
    customer.setUsername("rose");
    customer.setJobs("student");
    customer.setPhone("13333533092");
    // 4.2 执行 SqlSession 的插入方法，返回的是 SQL 语句影响的行数
    int rows = sqlSession.insert("com.itheima.mapper"
            + ".CustomerMapper.addCustomer", customer);
    // 4.3 通过返回结果判断插入操作是否执行成功
    if(rows > 0){
        System.out.println("您成功插入了"+rows+"条数据！");
    }else{
        System.out.println("执行插入操作失败！！！");
    }
    // 4.4 提交事务
    sqlSession.commit();
    // 5. 关闭 SqlSession
```

```
        sqlSession.close();
    }
```

在上述代码的第 4 步操作中，首先创建了 Customer 对象，并向 Customer 对象中添加了属性值；然后通过 SqlSession 对象的 insert()方法执行插入操作，并通过该操作返回的数据来判断插入操作是否执行成功；最后通过 SqlSesseion 的 commit()方法提交了事务，并通过 close()方法关闭了 SqlSession。

使用 JUnit4 执行 addCustomerTest()方法后，控制台的输出结果如图 6-11 所示。

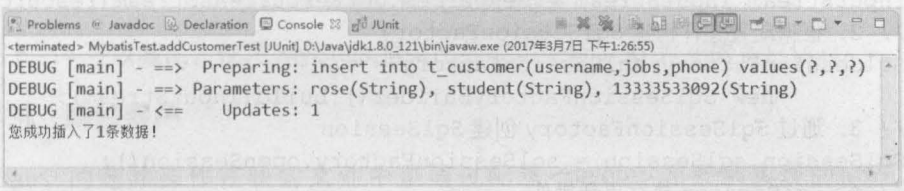


图6-11 运行结果

从图 6-11 可以看到，已经成功插入了 1 条数据。为了验证是否真的插入成功，此时查询数据库中的 t_customer 表，如图 6-12 所示。

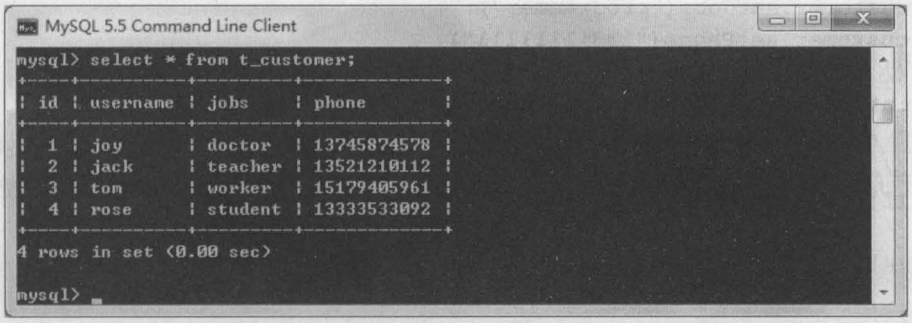


图6-12 t_customer表

从图 6-12 可以看出，使用 MyBatis 框架已成功新增了一条 id 为 4 的客户信息。

6.4.3 更新客户

MyBatis 的更新操作在映射文件中是通过配置<update>元素来实现的。如果需要更新用户数据，可以通过如下代码配置来实现。

```
<!-- 更新客户信息 -->
<update id="updateCustomer" parameterType="com.itheima.po.Customer">
    update t_customer set
    username=#{username},jobs=#{jobs},phone=#{phone}
    where id=#{id}
</update>
```

与插入数据的配置相比，更新操作配置中的元素与 SQL 语句都发生了相应变化，但其属性名却没有变。为了验证配置是否正确，下面以上一小节中新插入的数据为例，来进行更新客户测试。

在测试类 MybatisTest 中，添加测试方法 updateCustomerTest()，将 id 为 4 的用户职业修

改为“programmer”，电话修改为“13311111111”，其代码如下所示。

```
/**
 * 更新客户
 */
@Test
public void updateCustomerTest() throws Exception{
    // 1. 读取配置文件
    String resource = "mybatis-config.xml";
    InputStream inputStream = Resources.getResourceAsStream(resource);
    // 2. 根据配置文件构建 SqlSessionFactory
    SqlSessionFactory sqlSessionFactory =
            new SqlSessionFactoryBuilder().build(inputStream);
    // 3. 通过 SqlSessionFactory 创建 SqlSession
    SqlSession sqlSession = sqlSessionFactory.openSession();
    // 4. SqlSession 执行更新操作
    // 4.1 创建 Customer 对象，对对象中的数据进行模拟更新
    Customer customer = new Customer();
    customer.setId(4);
    customer.setUsername("rose");
    customer.setJobs("programmer");
    customer.setPhone("13311111111");
    // 4.2 执行 SqlSession 的更新方法，返回的是 SQL 语句影响的行数
    int rows = sqlSession.update("com.itheima.mapper"
          + ".CustomerMapper.updateCustomer", customer);
    // 4.3 通过返回结果判断更新操作是否执行成功
    if(rows > 0){
        System.out.println("您成功修改了"+rows+"条数据！");
    }else{
        System.out.println("执行修改操作失败！！！");
    }
    // 4.4 提交事务
    sqlSession.commit();
    // 5. 关闭 SqlSession
    sqlSession.close();
}
```

与添加客户的方法相比，更新操作的代码增加了 id 属性值的设置，并调用 SqlSession 的 update()方法对 id 为 1 客户的职业和电话进行修改。

使用 JUnit4 执行 updateCustomerTest()方法后，控制台的输出结果如图 6-13 所示。

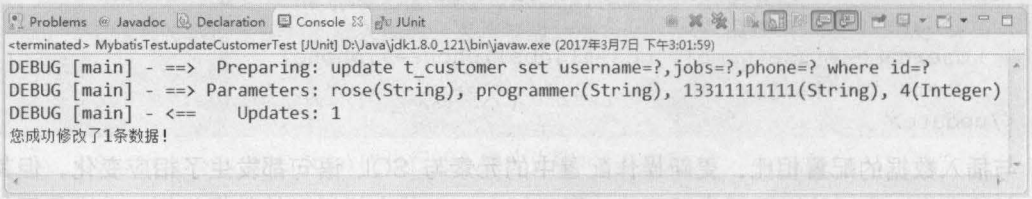


图6-13 运行结果

此时 t_customer 表中的数据如图 6-14 所示。

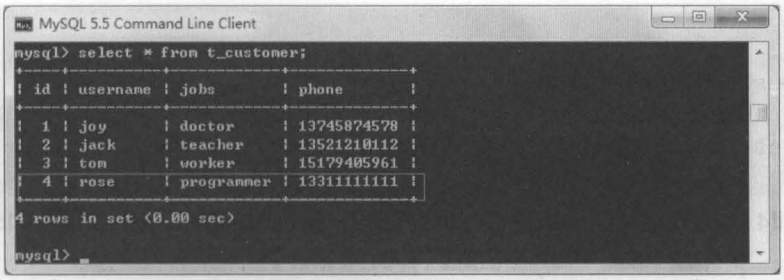


图6-14 t_customer表

从图6-14可以看出，使用MyBatis框架已经成功更新了id为4的客户信息。

6.4.4 删除客户

MyBatis 的删除操作在映射文件中是通过配置<delete>元素来实现的。在映射文件CustomerMapper.xml中添加删除客户信息的SQL语句，其示例代码如下。

```
<!-- 删除客户信息 -->
<delete id="deleteCustomer" parameterType="Integer">
   delete from t_customer where id=#{id}
</delete>
```

从上述配置的SQL语句中可以看出，我们只需要传递一个id值就可以将数据表中相应的数据删除。

要测试删除操作的配置十分简单，只需使用SqlSession对象的delete()方法传入需要删除数据的id值即可。在测试类MybatisTest中，添加测试方法deleteCustomerTest()，该方法用于将id为4的客户信息删除，其代码如下所示。

```
/**
 * 删除客户
 */
@Test
public void deleteCustomerTest() throws Exception{
   // 1. 读取配置文件
   String resource = "mybatis-config.xml";
   InputStream inputStream = Resources.getResourceAsStream(resource);
   // 2. 根据配置文件构建 SqlSessionFactory
   SqlSessionFactory sqlSessionFactory =
         new SqlSessionFactoryBuilder().build(inputStream);
   // 3. 通过 SqlSessionFactory 创建 SqlSession
   SqlSession sqlSession = sqlSessionFactory.openSession();
   // 4. SqlSession 执行删除操作
   // 4.1 执行 SqlSession 的删除方法，返回的是 SQL 语句影响的行数
   int rows = sqlSession.delete("com.itheima.mapper"
         + ".CustomerMapper.deleteCustomer", 4);
   // 4.2 通过返回结果判断删除操作是否执行成功
   if(rows > 0){
      System.out.println("您成功删除了"+rows+"条数据！");
   }else{
      System.out.println("执行删除操作失败！！！");
```

```
    }
    // 4.3 提交事务
    sqlSession.commit();
    // 5. 关闭 SqlSession
    sqlSession.close();
}
```

使用 JUnit4 执行 deleteCustomerTest()方法后，控制台的输出结果如图 6-15 所示。

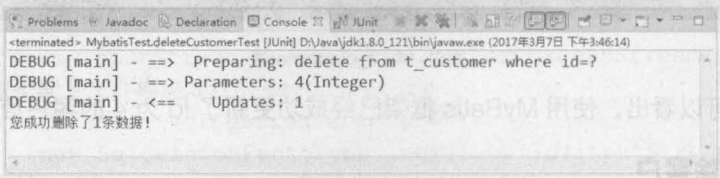


图6-15 运行结果

此时，再次查看表 t_customer 中的数据信息时，其结果如图 6-16 所示。

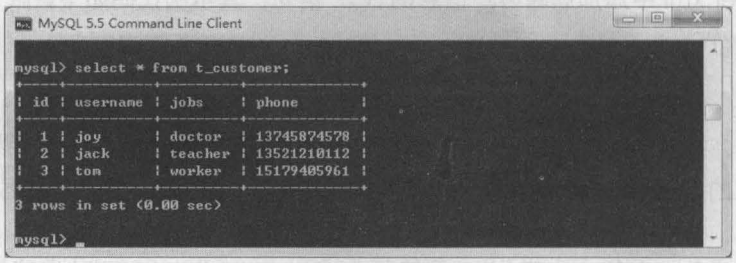


图6-16 t_customer表

从图 6-16 可以看出，使用 MyBatis 框架已经成功删除了 id 为 4 的客户信息。

至此，MyBatis 入门程序的增删改查操作已经讲解完成。关于程序中映射文件和配置文件中的元素信息，将在下一章详细讲解，本章入门程序读者只需要了解所使用的元素即可。

6.5 本章小结

本章首先对 MyBatis 框架的概念、特点和下载使用进行了讲解，然后对 MyBatis 框架的工作原理进行了流程分析，最后通过一个简单的增删改查案例来演示 MyBatis 框架的基本使用。通过本章的学习，读者可以了解 MyBatis 的概念和作用，熟悉 MyBatis 的工作原理，并能够使用 MyBatis 框架完成基本的数据库操作。

【思考题】

1. 请简述 MyBatis 框架与 Hibernate 框架的区别。
2. 请简述 MyBatis 的工作执行流程。

Java EE

7 Chapter

第 7 章 MyBatis 的核心配置

学习目标

- 了解 MyBatis 核心对象的作用
- 熟悉 MyBatis 配置文件中各个元素的作用
- 掌握 MyBatis 映射文件中常用元素的使用

通过上一章的学习，读者对 MyBatis 框架的使用已经有了一个初步的了解，但是要想熟练地使用 MyBatis 框架进行实际开发，只会简单的配置是不行的，我们还需要对框架中的核心对象，以及映射文件和配置文件有更加深入的了解。接下来，本章将对这些内容进行详细的讲解。

7.1 MyBatis 的核心对象

在使用 MyBatis 框架时，主要涉及两个核心对象：SqlSessionFactory 和 SqlSession，它们在 MyBatis 框架中起着至关重要的作用。本节将对这两个对象进行详细讲解。

7.1.1 SqlSessionFactory

SqlSessionFactory 是 MyBatis 框架中十分重要的对象，它是单个数据库映射关系经过编译后的内存镜像，其主要作用是创建 SqlSession。SqlSessionFactory 对象的实例可以通过 SqlSessionFactoryBuilder 对象来构建，而 SqlSessionFactoryBuilder 则可以通过 XML 配置文件或一个预先定义好的 Configuration 实例构建出 SqlSessionFactory 的实例。本书所讲解的就是通过 XML 配置文件构建出的 SqlSessionFactory 实例，其实现代码如下：

```
// 读取配置文件
InputStream inputStream = Resources.getResourceAsStream("配置文件位置");
// 根据配置文件构建 SqlSessionFactory
SqlSessionFactory sqlSessionFactory =
                    new SqlSessionFactoryBuilder().build(inputStream);
```

SqlSessionFactory 对象是线程安全的，它一旦被创建，在整个应用执行期间都会存在。如果我们多次地创建同一个数据库的 SqlSessionFactory，那么此数据库的资源将很容易被耗尽。为了解决此问题，通常每一个数据库都会只对应一个 SqlSessionFactory，所以在构建 SqlSessionFactory 实例时，建议使用单列模式。

7.1.2 SqlSession

SqlSession 是 MyBatis 框架中另一个重要的对象，它是应用程序与持久层之间执行交互操作的一个单线程对象，其主要作用是执行持久化操作。SqlSession 对象包含了数据库中所有执行 SQL 操作的方法，由于其底层封装了 JDBC 连接，所以可以直接使用其实例来执行已映射的 SQL 语句。

每一个线程都应该有一个自己的 SqlSession 实例，并且该实例是不能被共享的。同时，SqlSession 实例也是线程不安全的，因此其使用范围最好在一次请求或一个方法中，绝不能将其放在一个类的静态字段、实例字段或任何类型的管理范围（如 Servlet 的 HttpSession）中使用。使用完 SqlSession 对象之后，要及时地关闭它，通常可以将其放在 finally 块中关闭，代码如下所示。

```
SqlSession sqlSession = sqlSessionFactory.openSession();
try {
    // 此处执行持久化操作
} finally {
    sqlSession.close();
}
```

SqlSession 对象中包含了很多方法，其常用方法如下所示。

- <T> T selectOne (String statement);

查询方法。参数 statement 是在配置文件中定义的<select>元素的 id。使用该方法后，会返回执行 SQL 语句查询结果的一条泛型对象。

- <T> T selectOne (String statement，Object parameter);

查询方法。参数 statement 是在配置文件中定义的<select>元素的 id，parameter 是查询所需的参数。使用该方法后，会返回执行 SQL 语句查询结果的一条泛型对象。

- <E> List<E> selectList (String statement);

查询方法。参数 statement 是在配置文件中定义的<select>元素的 id。使用该方法后，会返回执行 SQL 语句查询结果的泛型对象的集合。

- <E> List<E> selectList (String statement，Object parameter);

查询方法。参数 statement 是在配置文件中定义的<select>元素的 id，parameter 是查询所需的参数。使用该方法后，会返回执行 SQL 语句查询结果的泛型对象的集合。

- <E> List<E> selectList (String statement，Object parameter，RowBounds rowBounds);

查询方法。参数 statement 是在配置文件中定义的<select>元素的 id，parameter 是查询所需的参数，rowBounds 是用于分页的参数对象。使用该方法后，会返回执行 SQL 语句查询结果的泛型对象的集合。

- void select (String statement，Object parameter，ResultHandler handler);

查询方法。参数 statement 是在配置文件中定义的<select>元素的 id，parameter 是查询所需的参数，ResultHandler 对象用于处理查询返回的复杂结果集，通常用于多表查询。

- int insert (String statement);

插入方法。参数 statement 是在配置文件中定义的<insert>元素的 id。使用该方法后，会返回执行 SQL 语句所影响的行数。

- int insert (String statement，Object parameter);

插入方法。参数 statement 是在配置文件中定义的<insert>元素的 id，parameter 是插入所需的参数。使用该方法后，会返回执行 SQL 语句所影响的行数。

- int update (String statement);

更新方法。参数 statement 是在配置文件中定义的<update>元素的 id。使用该方法后，会返回执行 SQL 语句所影响的行数。

- int update (String statement，Object parameter);

更新方法。参数 statement 是在配置文件中定义的<update>元素的 id，parameter 是更新所需的参数。使用该方法后，会返回执行 SQL 语句所影响的行数。

- int delete (String statement);

删除方法。参数 statement 是在配置文件中定义的<delete>元素的 id。使用该方法后，会返回执行 SQL 语句所影响的行数。

- int delete (String statement，Object parameter);

删除方法。参数 statement 是在配置文件中定义的<delete>元素的 id，parameter 是删除所需的参数。使用该方法后，会返回执行 SQL 语句所影响的行数。

- void commit();

提交事务的方法。

- void rollback();

回滚事务的方法。

- void close();

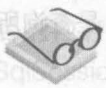
关闭 SqlSession 对象。

- <T> T getMapper(Class<T> type);

该方法会返回 Mapper 接口的代理对象，该对象关联了 SqlSession 对象，开发人员可以使用该对象直接调用方法操作数据库。参数 type 是 Mapper 的接口类型。MyBatis 官方推荐通过 Mapper 对象访问 MyBatis。

- Connection getConnection();

获取 JDBC 数据库连接对象的方法。



多学一招：使用工具类创建 SqlSession

在上一章的入门案例中，每个方法执行时都需要读取配置文件，并根据配置文件的信息构建 SqlSessionFactory 对象，然后创建 SqlSession 对象，这导致了大量的重复代码。为了简化开发，我们可以将上述重复代码封装到一个工具类中，然后通过工具类来创建 SqlSession，如文件 7-1 所示。

文件 7-1　MybatisUtils.java

```
1  package com.itheima.utils;
2  import java.io.Reader;
3  import org.apache.ibatis.io.Resources;
4  import org.apache.ibatis.session.SqlSession;
5  import org.apache.ibatis.session.SqlSessionFactory;
6  import org.apache.ibatis.session.SqlSessionFactoryBuilder;
7  /**
8   * 工具类
9   */
10 public class MybatisUtils {
11     private static SqlSessionFactory sqlSessionFactory = null;
12     // 初始化 SqlSessionFactory 对象
13     static {
14         try {
15             // 使用 MyBatis 提供的 Resources 类加载 MyBatis 的配置文件
16             Reader reader =
17                     Resources.getResourceAsReader("mybatis-config.xml");
18             // 构建 SqlSessionFactory 工厂
19             sqlSessionFactory =
20                     new SqlSessionFactoryBuilder().build(reader);
21         } catch (Exception e) {
22             e.printStackTrace();
23         }
24     }
25     // 获取 SqlSession 对象的静态方法
26     public static SqlSession getSession() {
```

```
27          return sqlSessionFactory.openSession();
28      }
29  }
```

这样，我们在使用时就只创建了一个 SqlSessionFactory 对象，并且可以通过工具类的 getSession()方法，来获取 SqlSession 对象。

7.2 配置文件

MyBatis 的核心配置文件中，包含了很多影响 MyBatis 行为的重要信息。这些信息通常在一个项目中只会在一个配置文件中编写，并且编写后也不会轻易改动。虽然在实际项目中需要开发人员编写或者修改的配置文件不多，但是熟悉配置文件中各个元素的功能还是十分重要的。接下来的几个小节中，将对 MyBatis 配置文件中的元素进行详细的讲解。

7.2.1 主要元素

在 MyBatis 框架的核心配置文件中，<configuration>元素是配置文件的根元素，其他元素都要在<configuration>元素内配置。在上一章的入门案例中，我们在配置文件内只使用了<environments>和<mapper>等几个元素，但在实际开发时，通常还会对其他一些元素进行配置。

MyBatis 配置文件中的主要元素如图 7-1 所示。

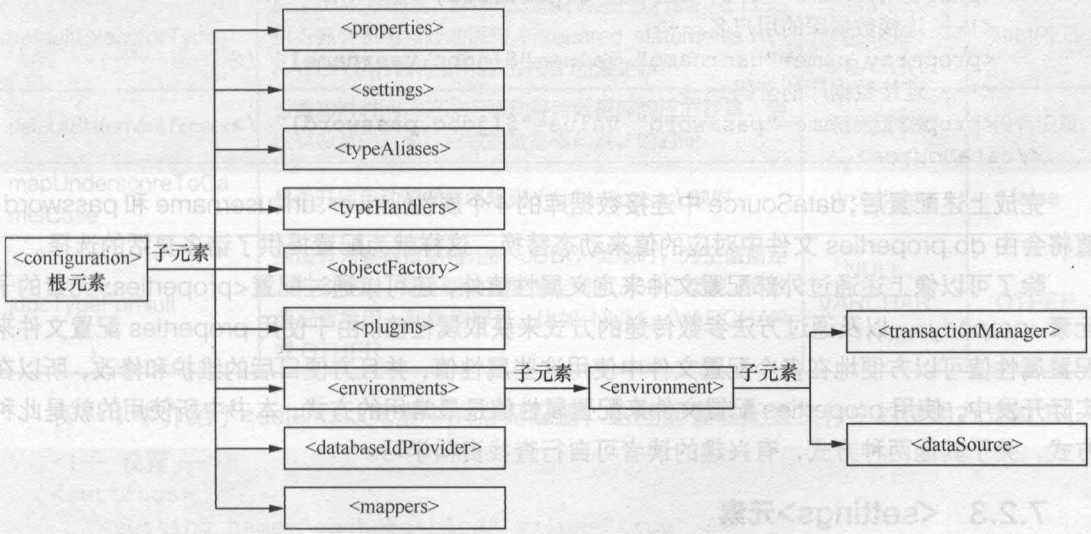


图7-1 MyBatis配置文件中的主要元素

从图 7-1 中可以看到，在 MyBatis 的配置文件中包含了多个元素，这些元素在配置文件中分别发挥着不同的作用。开发人员所需要熟悉的就是图中<configuration>元素各个子元素的配置。

注意

<configuration>的子元素必须按照图 7-1 中由上到下的顺序进行配置，否则 MyBatis 在解析 XML 配置文件的时候会报错。

7.2.2 <properties>元素

<properties>是一个配置属性的元素，该元素通常用于将内部的配置外在化，即通过外部的配置来动态地替换内部定义的属性。例如，数据库的连接等属性，就可以通过典型的 Java 属性文件中的配置来替换，具体方式如下。

（1）在项目的 src 目录下，添加一个全名为 db.properties 的配置文件，编辑后的代码如下所示。

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/mybatis
jdbc.username=root
jdbc.password=root
```

（2）在 MyBatis 配置文件 mybatis-config.xml 中配置<properties... />属性，具体如下。

```
<properties resource="db.properties" />
```

（3）修改配置文件中数据库连接的信息，具体如下。

```
<dataSource type="POOLED">
    <!-- 数据库驱动 -->
    <property name="driver" value="${jdbc.driver}" />
    <!-- 连接数据库的 url -->
    <property name="url" value="${jdbc.url}" />
    <!-- 连接数据库的用户名 -->
    <property name="username" value="${jdbc.username}" />
    <!-- 连接数据库的密码 -->
    <property name="password" value="${jdbc.password}" />
</dataSource>
```

完成上述配置后，dataSource 中连接数据库的 4 个属性（driver、url、username 和 password）值将会由 db.properties 文件中对应的值来动态替换。这样就为配置提供了诸多灵活的选择。

除了可以像上述通过外部配置文件来定义属性值外，还可以通过配置<properties>元素的子元素<property>，以及通过方法参数传递的方式来获取属性值。由于使用 properties 配置文件来配置属性值可以方便地在多个配置文件中使用这些属性值，并且方便日后的维护和修改，所以在实际开发中，使用 properties 配置文件来配置属性值是最常用的方式。本书中所使用的就是此种方式，关于其他两种方式，有兴趣的读者可自行查找资料学习。

7.2.3 <settings>元素

<settings>元素主要用于改变 MyBatis 运行时的行为，例如开启二级缓存、开启延迟加载等。虽然不配置<settings>元素，也可以正常运行 MyBatis，但是熟悉<settings>的配置内容以及它们的作用还是十分必要的。

<settings>元素中的常见配置及其描述如表 7-1 所示。

表 7-1 <settings>元素中的常见配置

设置参数	描述	有效值	默认值
cacheEnabled	该配置影响所有映射器中配置的缓存全局开关	true|false	false

续表

设置参数	描述	有效值	默认值
lazyLoadingEnabled	延迟加载的全局开关。开启时，所有关联对象都会延迟加载。特定关联关系中可以通过设置 fetchType 属性来覆盖该项的开关状态	true|false	false
aggressiveLazyLoading	关联对象属性的延迟加载开关。当启用时，对任意延迟属性的调用会使带有延迟加载属性的对象完整加载；反之，每种属性都会按需加载	true|false	true
multipleResultSetsEnabled	是否允许单一语句返回多结果集（需要兼容驱动）	true|false	true
useColumnLabel	使用列标签代替列名。不同的驱动在这方面有不同的表现。具体可参考驱动文档或通过测试两种模式来观察所用驱动的行为	true|false	true
useGeneratedKeys	允许 JDBC 支持自动生成主键，需要驱动兼容。如果设置为 true，则这个设置强制使用自动生成主键，尽管一些驱动不兼容但仍可正常工作	true|false	false
autoMappingBehavior	指定 MyBatis 应如何自动映射列到字段或属性。NONE 表示取消自动映射；PARTIAL 只会自动映射没有定义嵌套结果集映射的结果集；FULL 会自动映射任意复杂的结果集（无论是否嵌套）	NONE、PARTIAL、FULL	PARTIAL
defaultExecutorType	配置默认的执行器。SIMPLE 就是普通的执行器；REUSE 执行器会重用预处理语句（prepared statements）；BATCH 执行器将重用语句并执行批量更新	SIMPLE、REUSE、BATCH	SIMPLE
defaultStatementTimeout	设置超时时间，它决定驱动等待数据库响应的秒数。当没有设置的时候，它取的就是驱动默认的时间	任何正整数	没有设置
mapUnderscoreToCamelCase	是否开启自动驼峰命名规则（camel case）映射	true|false	false
jdbcTypeForNull	当没有为参数提供特定的 JDBC 类型时，为空值指定 JDBC 类型。某些驱动需要指定列的 JDBC 类型，多数情况直接用一般类型即可，比如 NULL、VARCHAR 或 OTHER	NULL、VARCHAR、OTHER	OTHER

表 7-1 中介绍了<settings>元素中的常见配置，这些配置在配置文件中的使用方式如下。

```
<!-- 设置 -->
<settings>
    <setting name="cacheEnabled" value="true" />
    <setting name="lazyLoadingEnabled" value="true" />
    <setting name="multipleResultSetsEnabled" value="true" />
    <setting name="useColumnLabel" value="true" />
    <setting name="useGeneratedKeys" value="false" />
    <setting name="autoMappingBehavior" value="PARTIAL" />
    ...
</settings>
```

上面所介绍的配置内容大多数都不需要开发人员去配置它，通常在需要时只配置少数几项即可。这里读者只需要了解这些可设置的参数值及其含义即可。

7.2.4 <typeAliases>元素

<typeAliases>元素用于为配置文件中的 Java 类型设置一个简短的名字，即设置别名。别名的设置与 XML 配置相关，其使用的意义在于减少全限定类名的冗余。

使用<typeAliases>元素配置别名的方法如下。

```
<!-- 定义别名 -->
<typeAliases>
    <typeAlias alias="user" type="com.itheima.po.User"/>
</typeAliases>
```

上述示例中，<typeAliases>元素的子元素<typeAlias>中的 type 属性用于指定需要被定义别名的类的全限定名；alias 属性的属性值 user 就是自定义的别名，它可以代替 com.itheima.po.User 使用在 MyBatis 文件的任何位置。如果省略 alias 属性，MyBatis 会默认将类名首字母小写后的名称作为别名。

当 POJO 类过多时，还可以通过自动扫描包的形式自定义别名，具体示例如下。

```
<!-- 使用自动扫描包来定义别名 -->
<typeAliases>
    <package name="com.itheima.po"/>
</typeAliases>
```

上述示例中，<typeAliases>元素的子元素<package>中的 name 属性用于指定要被定义别名的包，MyBatis 会将所有 com.itheima.po 包中的 POJO 类以首字母小写的非限定类名来作为它的别名，比如 com.itheima.po.User 的别名为 user，com.itheima.po.Customer 的别名为 customer 等。

需要注意的是，上述方式的别名只适用于没有使用注解的情况。如果在程序中使用了注解，则别名为其注解的值，具体如下。

```
@Alias(value = "user")
public class User {
    //User 的属性和方法
    ...
}
```

除了可以使用<typeAliases>元素自定义别名外，MyBatis 框架还默认为许多常见的 Java 类型（如数值、字符串、日期和集合等）提供了相应的类型别名，如表 7-2 所示。

表 7-2 MyBatis 默认别名

别名	映射的类型
_byte	byte
_long	long
_short	short
_int	int
_integer	int
_double	double

续表

别名	映射的类型
_float	float
_boolean	boolean
string	String
byte	Byte
long	Long
short	Short
int	Integer
integer	Integer
double	Double
float	Float
boolean	Boolean
date	Date
decimal	BigDecimal
bigdecimal	BigDecimal
object	Object
map	Map
hashmap	HashMap
list	List
arraylist	ArrayList
collection	Collection
iterator	Iterator

表 7-2 所列举的别名可以在 MyBatis 中直接使用，但由于别名不区分大小写，所以在使用时要注意重复定义的覆盖问题。

7.2.5 <typeHandler>元素

MyBatis 在预处理语句（PreparedStatement）中设置一个参数或者从结果集（ResultSet）中取出一个值时，都会用其框架内部注册了的 typeHandler（类型处理器）进行相关处理。typeHandler 的作用就是将预处理语句中传入的参数从 javaType（Java 类型）转换为 jdbcType（JDBC 类型），或者从数据库取出结果时将 jdbcType 转换为 javaType。

为了方便转换，MyBatis 框架提供了一些默认的类型处理器，其常用的类型处理器如表 7-3 所示。

表 7-3 常用的类型处理器

类型处理器	Java 类型	JDBC 类型
BooleanTypeHandler	java.lang.Boolean，boolean	数据库兼容的 BOOLEAN
ByteTypeHandler	java.lang.Byte，byte	数据库兼容的 NUMERIC 或 BYTE
ShortTypeHandler	java.lang.Short，short	数据库兼容的NUMERIC 或SHORT INTEGER

续表

类型处理器	Java 类型	JDBC 类型
IntegerTypeHandler	java.lang.Integer，int	数据库兼容的 NUMERIC 或 INTEGER
LongTypeHandler	java.lang.Long，long	数据库兼容的 NUMERIC 或 LONG INTEGE
FloatTypeHandler	java.lang.Float，float	数据库兼容的 NUMERIC 或 FLOAT
DoubleTypeHandler	java.lang.Double，double	数据库兼容的 NUMERIC 或 DOUBLE
BigDecimalTypeHandler	java.math.BigDecimal	数据库兼容的 NUMERIC 或 DECIMAL
StringTypeHandler	java.lang.String	CHAR，VARCHAR
ClobTypeHandler	java.lang.String	CLOB，LONGVARCHAR
ByteArrayTypeHandler	byte[]	数据库兼容的字节流类型
BlobTypeHandler	byte[]	BLOB，LONGVARBINARY
DateTypeHandler	java.util.Date	TIMESTAMP
SqlTimestampTypeHandler	java.sql.Timestamp	TIMESTAMP
SqlDateTypeHandler	java.sql.Date	DATE
SqlTimeTypeHandler	java.sql.Time	TIME

当 MyBatis 框架所提供的这些类型处理器不能够满足需求时，还可以通过自定义的方式对类型处理器进行扩展（自定义类型处理器可以通过实现 TypeHandler 接口或者继承 BaseTypeHandle 类来定义）。<typeHandler>元素就是用于在配置文件中注册自定义的类型处理器的。它的使用方式有两种，具体如下。

1. 注册一个类的类型处理器

```
<typeHandlers>
    <!-- 以单个类的形式配置 -->
    <typeHandler handler="com.itheima.type.CustomtypeHandler" />
</typeHandlers>
```

上述代码中，子元素<typeHandler>的 handler 属性用于指定在程序中自定义的类型处理器类。

2. 注册一个包中所有的类型处理器

```
<typeHandlers>
    <!-- 注册一个包中所有的 typeHandler，系统在启动时会自动扫描包下的所有文件-->
    <package name="com.itheima.type" />
</typeHandlers>
```

上述代码中，子元素<package>的 name 属性用于指定类型处理器所在的包名，使用此种方式后，系统会在启动时自动地扫描 com.itheima.type 包下所有的文件，并把它们作为类型处理器。

7.2.6 <objectFactory>元素

MyBatis 框架每次创建结果对象的新实例时，都会使用一个对象工厂（ObjectFactory）的实例来完成。MyBatis 中默认的 ObjectFactory 的作用就是实例化目标类，它既可以通过默认构造方法实例化，也可以在参数映射存在的时候通过参数构造方法来实例化。

在通常情况下，我们使用默认的 ObjectFactory 即可，MyBatis 中默认的 ObjectFactory 是

由 org.apache.ibatis.reflection.factory.DefaultObjectFactory 来提供服务的。大部分场景下都不用配置和修改，但如果想覆盖 ObjectFactory 的默认行为，则可以通过自定义 ObjectFactory 来实现，具体方式如下。

（1）自定义一个对象工厂。自定义的对象工厂需要实现 ObjectFactory 接口，或者继承 DefaultObjectFactory 类。由于 DefaultObjectFactory 类已经实现了 ObjectFactory 接口，所以通过继承 DefaultObjectFactory 类实现即可，示例代码如下所示。

```
// 自定义工厂类
public class MyObjectFactory extends DefaultObjectFactory {
    private static final long serialVersionUID = -4114845625429965832L;
    public <T> T create(Class<T> type) {
        return super.create(type);
    }
    public <T> T create(Class<T> type, List<Class<?>> constructorArgTypes,
                                       List<Object> constructorArgs) {
        return super.create(type, constructorArgTypes, constructorArgs);
    }
    public void setProperties(Properties properties) {
        super.setProperties(properties);
    }
    public <T> boolean isCollection(Class<T> type) {
        return Collection.class.isAssignableFrom(type);
    }
}
```

（2）在配置文件中使用<objectFactory>元素配置自定义的 ObjectFactory，如下所示。

```
<objectFactory type="com.itheima.factory.MyObjectFactory">
    <property name="name" value="MyObjectFactory"/>
</objectFactory>
```

由于自定义 ObjectFactory 在实际开发时不经常使用，这里读者只需要了解即可。

7.2.7 <plugins>元素

MyBatis 允许在已映射语句执行过程中的某一点进行拦截调用，这种拦截调用是通过插件来实现的。<plugins>元素的作用就是配置用户所开发的插件。如果用户想要进行插件开发，必须要先了解其内部运行原理，因为在试图修改或重写已有方法的行为时，很可能会破坏 MyBatis 原有的核心模块。关于插件的使用，本书不做详细讲解，读者只需了解<plugins>元素的作用即可，有兴趣的读者可以查找官方文档等资料自行学习。

7.2.8 <environments>元素

在配置文件中，<environments>元素用于对环境进行配置。MyBatis 的环境配置实际上就是数据源的配置，我们可以通过<environments>元素配置多种数据源，即配置多种数据库。

使用<environments>元素进行环境配置的示例如下。

```
<environments default="development">
    <environment id="development">
```

```
        <!-- 使用 JDBC 事务管理 -->
        <transactionManager type="JDBC" />
        <!-- 配置数据源 -->
        <dataSource type="POOLED">
            <property name="driver" value="${jdbc.driver}" />
            <property name="url" value="${jdbc.url}" />
            <property name="username" value="${jdbc.username}" />
            <property name="password" value="${jdbc.password}" />
        </dataSource>
    </environment>
    ...
</environments>
```

在上述示例代码中，<environments>元素是环境配置的根元素，它包含一个 default 属性，该属性用于指定默认的环境 ID。<environment>是<environments>元素的子元素，它可以定义多个，其 id 属性用于表示所定义环境的 ID 值。在<environment>元素内，包含事务管理和数据源的配置信息，其中<transactionManager>元素用于配置事务管理，它的 type 属性用于指定事务管理的方式，即使用哪种事务管理器；<dataSource>元素用于配置数据源，它的 type 属性用于指定使用哪种数据源。

在 MyBatis 中，可以配置两种类型的事务管理器，分别是 JDBC 和 MANAGED。关于这两个事务管理器的描述如下。

- JDBC：此配置直接使用了 JDBC 的提交和回滚设置，它依赖于从数据源得到的连接来管理事务的作用域。
- MANAGED：此配置从来不提交或回滚一个连接，而是让容器来管理事务的整个生命周期。在默认情况下，它会关闭连接，但一些容器并不希望这样，为此可以将 closeConnection 属性设置为 false 来阻止它默认的关闭行为。

注意

如果项目中使用的是 Spring+ MyBatis，则没有必要在 MyBatis 中配置事务管理器，因为实际开发中，会使用 Spring 自带的管理器来实现事务管理。

对于数据源的配置，MyBatis 框架提供了 UNPOOLED、POOLED 和 JNDI 三种数据源类型，具体如下。

1. UNPOOLED

配置此数据源类型后，在每次被请求时会打开和关闭连接。它对没有性能要求的简单应用程序是一个很好的选择。

UNPOOLED 类型的数据源需要配置 5 种属性，如表 7-4 所示。

表 7-4 UNPOOLED 数据源需要配置的属性

属性	说明
driver	JDBC 驱动的 Java 类的完全限定名（并不是 JDBC 驱动中可能包含的数据源类）
url	数据库的 URL 地址

续表

属性	说明
username	登录数据库的用户名
password	登录数据库的密码
defaultTransactionIsolationLevel	默认的连接事务隔离级别

2. POOLED

此数据源利用“池”的概念将 JDBC 连接对象组织起来，避免了在创建新的连接实例时所需要初始化和认证的时间。这种方式使得并发 Web 应用可以快速地响应请求，是当前流行的处理方式（本书中使用的就是此种方式）。

配置此数据源类型时，除了表 7-4 中的 5 种属性外，还可以配置更多的属性，如表 7-5 所示。

表 7-5 POOLED 数据源可额外配置的属性

属性	说明
poolMaximumActiveConnections	在任意时间可以存在的活动（也就是正在使用）连接数量，默认值：10
poolMaximumIdleConnections	任意时间可能存在的空闲连接数
poolMaximumCheckoutTime	在被强制返回之前，池中连接被检出(checked out)时间，默认值：20000 毫秒，即 20 秒
poolTimeToWait	如果获取连接花费的时间较长，它会给连接池打印状态日志并重新尝试获取一个连接（避免在误配置的情况下一直处于无提示的失败），默认值：20000 毫秒，即 20 秒
poolPingQuery	发送到数据库的侦测查询，用于检验连接是否处在正常工作秩序中。默认是“NO PING QUERY SET”，这会导致多数数据库驱动失败时带有一定的错误消息
poolPingEnabled	是否启用侦测查询。若开启，必须使用一个可执行的 SQL 语句设置 poolPingQuery 属性（最好是一个非常快的 SQL），默认值：false
poolPingConnectionsNotUsedFor	配置 poolPingQuery 的使用频度。可以被设置成匹配具体的数据库连接超时时间，来避免不必要的侦测，默认值：0（表示所有连接每一时刻都被侦测，只有 poolPingEnabled 的属性值为 true 时适用）

3. JNDI

此数据源可以在 EJB 或应用服务器等容器中使用。容器可以集中或在外部配置数据源，然后放置一个 JNDI 上下文的引用。

配置 JNDI 数据源时，只需要配置两个属性，如表 7-6 所示。

表 7-6 JNDI 数据源需要配置的属性

属性	说明
initial_context	此属性主要用于在 InitialContext 中寻找上下文（即 initialContext.lookup(initial_context)）。该属性为可选属性，在忽略时，data_source 属性会直接从 InitialContext 中寻找
data_source	此属性表示引用数据源实例位置的上下文的路径。如果提供了 initial_context 配置，那么程序会在其返回的上下文中进行查找；如果没有提供，则直接在 InitialContext 中查找

7.2.9 <mappers>元素

在配置文件中，<mappers>元素用于指定 MyBatis 映射文件的位置，一般可以使用以下 4

种方法引入映射器文件，具体如下所示。

1. 使用类路径引入

```
<mappers>
    <mapper resource="com/itheima/mapper/UserMapper.xml"/>
</mappers>
```

2. 使用本地文件路径引入

```
<mappers>
    <mapper url="file:///D:/com/itheima/mapper/UserMapper.xml"/>
</mappers>
```

3. 使用接口类引入

```
<mappers>
    <mapper class="com.itheima.mapper.UserMapper"/>
</mappers>
```

4. 使用包名引入

```
<mappers>
    <package name="com.itheima.mapper"/>
</mappers>
```

上述 4 种引入方式非常简单，读者可以根据实际项目需要选取使用。

7.3 映射文件

映射文件是 MyBatis 框架中十分重要的文件，可以说，MyBatis 框架的强大之处就体现在映射文件的编写上。接下来的几个小节中，将对 MyBatis 映射文件中的元素进行详细讲解。

7.3.1 主要元素

在映射文件中，<mapper>元素是映射文件的根元素，其他元素都是它的子元素。这些子元素及其作用如图 7-2 所示。

图7-2 映射文件中的主要元素

7.3.2 <select>元素

<select>元素用于映射查询语句，它可以帮助我们从数据库中读取出数据，并组装数据给业务开发人员。

使用<select>元素执行查询操作非常简单，其示例如下。

```
<select id="findCustomerById" parameterType="Integer"
    resultType="com.itheima.po.Customer">
    select * from t_customer where id = #{id}
</select>
```

上述语句中的唯一标识为 findCustomerById，它接收一个 Integer 类型的参数，并返回一个 Customer 类型的对象。

<select>元素中，除了上述示例代码中的几个属性外，还有其他一些可以配置的属性，如表 7-7 所示。

表 7-7 <select>元素的常用属性

属性	说明
id	表示命名空间中的唯一标识符，常与命名空间组合起来使用。组合后如果不唯一，MyBatis 会抛出异常
parameterType	该属性表示传入 SQL 语句的参数类的全限定名或者别名。它是一个可选属性，因为 MyBatis 可以通过 TypeHandler 推断出具体传入语句的参数。其默认值是 unset（依赖于驱动）
resultType	从 SQL 语句中返回的类型的类的全限定名或者别名。如果是集合类型，那么返回的应该是集合可以包含的类型，而不是集合本身。返回时可以使用 resultType 或 resultMap 之一
resultMap	表示外部 resultMap 的命名引用。返回时可以使用 resultType 或 resultMap 之一
flushCache	表示在调用 SQL 语句之后，是否需要 MyBatis 清空之前查询的本地缓存和二级缓存。其值为布尔类型（true|false），默认值为 false。如果设置为 true，则任何时候只要 SQL 语句被调用，都会清空本地缓存和二级缓存
useCache	用于控制二级缓存的开启和关闭。其值为布尔类型（true|false），默认值为 true，表示将查询结果存入二级缓存中
timeout	用于设置超时参数，单位为秒。超时时将抛出异常
fetchSize	获取记录的总条数设定，其默认值是 unset（依赖于驱动）
statementType	用于设置 MyBatis 使用哪个 JDBC 的 Statement 工作，其值为 STATEMENT、PREPARED（默认值）或 CALLABLE，分别对应 JDBC 中的 Statement、PreparedStatement 和 CallableStatement
resultSetType	表示结果集的类型，其值可设置为 FORWARD_ONLY、SCROLL_SENSITIVE 或 SCROLL_INSENSITIVE，它的默认值是 unset（依赖于驱动）

7.3.3 <insert>元素

<insert>元素用于映射插入语句，在执行完元素中定义的 SQL 语句后，会返回一个表示插入记录数的整数。

<insert>元素的配置示例如下。

```
<insert
    id="addCustomer"
```

```
    parameterType="com.itheima.po.Customer"
    flushCache="true"
    statementType="PREPARED"
    keyProperty=""
    keyColumn=""
    useGeneratedKeys=""
    timeout="20">
```

从上述示例代码中可以看出，<insert>元素的属性与<select>元素的属性大部分相同，但还包含了 3 个特有属性，这 3 个属性的描述如表 7-8 所示。

表 7-8 <insert>元素中的属性

属性	说明
keyProperty	（仅对 insert 和 update 有用）此属性的作用是将插入或更新操作时的返回值赋值给 PO 类的某个属性，通常会设置为主键对应的属性。如果需要设置联合主键，可以在多个值之间用逗号隔开
keyColumn	（仅对 insert 和 update 有用）此属性用于设置第几列是主键，当主键列不是表中的第一列时需要设置。在需要主键联合时，值可以用逗号隔开
useGeneratedKeys	（仅对 insert 和 update 有用）此属性会使 MyBatis 使用 JDBC 的 getGeneratedKeys() 方法来获取由数据库内部生产的主键，如 MySQL 和 SQL Server 等自动递增的字段，其默认值为 false

执行插入操作后，很多时候我们会需要返回插入成功的数据生成的主键值，此时就可以通过上面所讲解的 3 个属性来实现。

如果使用的数据库支持主键自动增长（如 MySQL），那么可以通过 keyProperty 属性指定 PO 类的某个属性接收主键返回值（通常会设置到 id 属性上），然后将 useGeneratedKeys 的属性值设置为 true，其使用示例如下。

```
<insert id="addCustomer" parameterType="com.itheima.po.Customer"
        keyProperty="id" useGeneratedKeys="true" >
    insert into t_customer(username,jobs,phone)
    values(#{username},#{jobs},#{phone})
</insert>
```

使用上述配置执行插入后，会返回插入成功的行数，以及插入行的主键值。为了验证此配置，可以通过如下代码测试。

```
@Test
public void addCustomerTest(){
    // 获取 SqlSession
    SqlSession sqlSession = MybatisUtils.getSession();
    Customer customer = new Customer();
    customer.setUsername("rose");
    customer.setJobs("student");
    customer.setPhone("13333533092");
    int rows = sqlSession.insert("com.itheima.mapper."
          + "CustomerMapper.addCustomer", customer);
    // 输出插入数据的主键 id 值
    System.out.println(customer.getId());
```

```
        if(rows > 0){
            System.out.println("您成功插入了"+rows+"条数据！");
        }else{
            System.out.println("执行插入操作失败！！！");
        }
        sqlSession.commit();
        sqlSession.close();
    }
```

执行程序后，控制台的输出结果如图 7-3 所示。

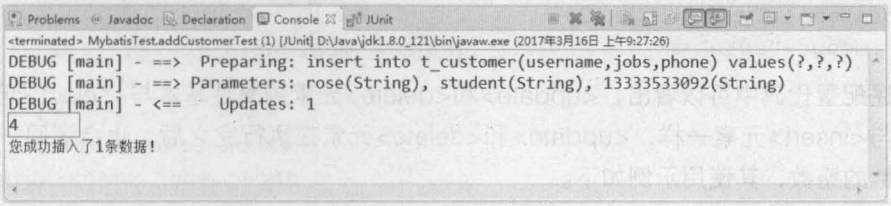


图7-3 运行结果

如果使用的数据库不支持主键自动增长（如 Oracle），或者支持增长的数据库取消了主键自增的规则时，也可以使用 MyBatis 提供的另一种方式来自定义生成主键，具体配置示例如下。

```
<insert id="insertCustomer" parameterType="com.itheima.po.Customer">
    <selectKey keyProperty="id" resultType="Integer" order="BEFORE">
      select if(max(id) is null, 1, max(id) +1) as newId from t_customer
    </selectKey>
    insert into t_customer(id,username,jobs,phone)
    values(#{id},#{username},#{jobs},#{phone})
</insert>
```

在执行上述示例代码时，<selectKey>元素会首先运行，它会通过自定义的语句来设置数据表中的主键（如果 t_customer 表中没有记录，则将 id 设置为 1，否则就将 id 的最大值加 1，来作为新的主键），然后再调用插入语句。

<selectKey>元素在使用时可以设置以下几种属性。

```
<selectKey
    keyProperty="id"
    resultType="Integer"
    order="BEFORE"
    statementType="PREPARED">
```

在上述<selectKey>元素的几个属性中，keyProperty、resultType 和 statementType 的作用与前面讲解的相同，这里不重复介绍。order 属性可以被设置为 BEFORE 或 AFTER。如果设置为 BEFORE，那么它会首先执行<selectKey>元素中的配置来设置主键，然后执行插入语句；如果设置为 AFTER，那么它会先执行插入语句，然后执行<selectKey>元素中的配置内容。

7.3.4 <update>元素和<delete>元素

<update>和<delete>元素的使用比较简单，它们的属性配置也基本相同（<delete>元素中不包含表 7-8 中的 3 个属性），其常用属性如下所示。

```
<update
    id="updateCustomer"
    parameterType="com.itheima.po.Customer"
    flushCache="true"
    statementType="PREPARED"
    timeout="20">
<delete
    id="deleteCustomer"
    parameterType="com.itheima.po.Customer"
    flushCache="true"
    statementType="PREPARED"
    timeout="20">
```

从上述配置代码中可以看出，<update>和<delete>元素的属性基本与<select>元素中的属性一致。与<insert>元素一样，<update>和<delete>元素在执行完之后，也会返回一个表示影响记录条数的整数，其使用示例如下。

```
<!-- 更新信息 -->
<update id="updateCustomer" parameterType="com.itheima.po.Customer">
    update t_customer
    set username=#{username},jobs=#{jobs},phone=#{phone}
    where id=#{id}
</update>
<!-- 删除信息 -->
<delete id="deleteCustomer" parameterType="Integer">
    delete from t_customer where id=#{id}
</delete>
```

7.3.5 <sql>元素

在一个映射文件中，通常需要定义多条 SQL 语句，这些 SQL 语句的组成可能有一部分是相同的（如多条 select 语句中都查询相同的 id、username、jobs 字段），如果每一个 SQL 语句都重写一遍相同的部分，势必会增加代码量，导致映射文件过于臃肿。那么有没有什么办法将这些 SQL 语句中相同的组成部分抽取出来，然后在需要的地方引用呢？答案是肯定的，我们可以在映射文件中使用 MyBatis 所提供的<sql>元素来解决上述问题。

<sql>元素的作用就是定义可重用的 SQL 代码片段，然后在其他语句中引用这一代码片段。例如，定义一个包含 id、username、jobs 和 phone 字段的代码片段如下。

```
<sql id="customerColumns">id,username,jobs,phone</sql>
```

这一代码片段可以包含在其他语句中使用，具体如下。

```
<select id="findCustomerById" parameterType="Integer"
        resultType="com.itheima.po.Customer">
    select <include refid="customerColumns"/>
    from t_customer
    where id = #{id}
</select>
```

在上述代码中，使用<include>元素的 refid 属性引用了自定义的代码片段，refid 的属性值

为自定义代码片段的 id。

上面示例只是一个简单的引用查询。在实际开发中，可以更加灵活地定义 SQL 片段，其示例如下。

```
<!--定义表的前缀名 -->
<sql id="tablename">
   ${prefix}customer
</sql>
<!--定义要查询的表 -->
<sql id="someinclude">
   from
   <include refid="${include_target}" />
</sql>
<!--定义查询列 -->
<sql id="customerColumns">
  id,username,jobs,phone
</sql>
<!--根据 id 查询客户信息 -->
<select id="findCustomerById" parameterType="Integer"
       resultType="com.itheima.po.Customer">
   select
   <include refid="customerColumns"/>
   <include refid="someinclude">
      <property name="prefix" value="t_" />
      <property name="include_target" value="tablename" />
   </include>
   where id = #{id}
</select>
```

上述代码中，定义了 3 个代码片段，分别为表的前缀名、要查询的表和需要查询的列。前两个代码片段中，分别获取了<include>子元素<property>中的值，其中第 1 个代码片段中的“${prefix}”会获取 name 为 prefix 的值“t_”，获取后所组成的表名为“t_customer”；而第 2 个代码片段中的“${include_target}”会获取 name 为 include_target 的值“tablename”，由于 tablename 为第 1 个 SQL 片段的 id 值，所以最后要查询的表为“t_customer”。所有的 SQL 片段在程序运行时，都会由 MyBatis 组合成 SQL 语句来执行需要的操作。

执行程序后，控制台的输出结果如图 7-4 所示。

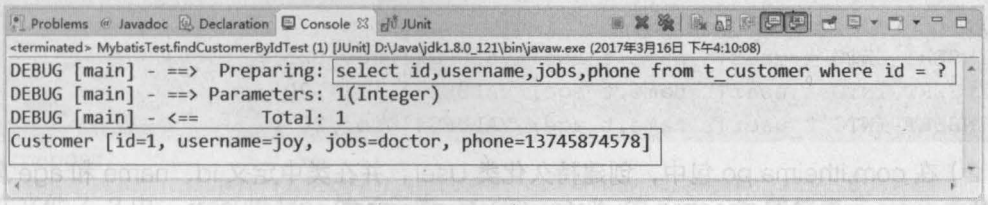


图7-4 运行结果

7.3.6 <resultMap>元素

<resultMap>元素表示结果映射集，是 MyBatis 中最重要也是最强大的元素。它的主要作用

是定义映射规则、级联的更新以及定义类型转化器等。

<resultMap>元素中包含了一些子元素，它的元素结构如下所示。

```
<!--resultMap 的元素结构 -->
<resultMap type="" id="">
    <constructor>   <!-- 类在实例化时,用来注入结果到构造方法中-->
        <idArg/>    <!-- ID 参数;标记结果作为 ID-->
        <arg/>      <!-- 注入到构造方法的一个普通结果-->
    </constructor>
    <id/>           <!-- 用于表示哪个列是主键-->
    <result/>       <!-- 注入到字段或 JavaBean 属性的普通结果-->
    <association property="" />   <!-- 用于一对一关联 -->
    <collection property="" />    <!-- 用于一对多关联 -->
    <discriminator javaType="">   <!-- 使用结果值来决定使用哪个结果映射-->
        <case value="" />         <!-- 基于某些值的结果映射 -->
    </discriminator>
</resultMap>
```

<resultMap>元素的 type 属性表示需要映射的 POJO，id 属性是这个 resultMap 的唯一标识。它的子元素<constructor>用于配置构造方法（当一个 POJO 中未定义无参的构造方法时，就可以使用<constructor>元素进行配置）。子元素<id>用于表示哪个列是主键，而<result>用于表示 POJO 和数据表中普通列的映射关系。<association>和<collection>用于处理多表时的关联关系，而<discriminator>元素主要用于处理一个单独的数据库查询返回很多不同数据类型结果集的情况。

在默认情况下，MyBatis 程序在运行时会自动地将查询到的数据与需要返回的对象的属性进行匹配赋值（需要表中的列名与对象的属性名称完全一致）。然而实际开发时，数据表中的列和需要返回的对象的属性可能不会完全一致，这种情况下 MyBatis 是不会自动赋值的。此时，就可以使用<resultMap>元素进行处理。

接下来，通过一个具体的案例来演示<resultMap>元素在此种情况的使用，具体步骤如下。

（1）在 mybatis 数据库中，创建一个 t_user 表，并插入几条测试数据。

```
USE mybatis;
CREATE TABLE t_user(
   t_id INT PRIMARY KEY AUTO_INCREMENT,
   t_name VARCHAR(50),
   t_age INT
);
INSERT INTO t_user(t_name,t_age) VALUES('Lucy',25);
INSERT INTO t_user(t_name,t_age) VALUES('Lili',20);
INSERT INTO t_user(t_name,t_age) VALUES('Jim',20);
```

（2）在 com.itheima.po 包中，创建持久化类 User，并在类中定义 id、name 和 age 属性，以及其 getter/setter 方法和 toString()方法，如文件 7-2 所示。

文件 7-2　User.java

```
1  package com.itheima.po;
2  public class User {
3      private Integer id;
```

```
    private String name;
    private Integer age;
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Integer getAge() {
        return age;
    }
    public void setAge(Integer age) {
        this.age = age;
    }
    @Override
    public String toString() {
        return "User [id=" + id + ", name=" + name + ", age=" + age + "]";
    }
}
```

（3）在 com.itheima.mapper 包下，创建映射文件 UserMapper.xml，并在映射文件中编写映射查询语句，如文件 7-3 所示。

文件 7-3 UserMapper.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper
    PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
    <mapper namespace="com.itheima.mapper.UserMapper">
    <resultMap type="com.itheima.po.User" id="resultMap">
        <id property="id" column="t_id"/>
        <result property="name" column="t_name"/>
        <result property="age" column="t_age"/>
    </resultMap>
    <select id="findAllUser" resultMap="resultMap">
        select * from t_user
    </select>
</mapper>
```

在文件 7-3 中，<resultMap>的子元素<id>和<result>的 property 属性表示 User 类的属性名，column 属性表示数据表 t_user 的列名。<select>元素的 resultMap 属性表示引用上面定义的 resultMap。

（4）在配置文件 mybatis-config.xml 中，引入 UserMapper.xml。

（5）在测试类中，编写测试方法 findAllUserTest()，代码如下所示。

```
@Test
public void findAllUserTest() {
    // 获取 SqlSession
    SqlSession sqlSession = MybatisUtils.getSession();
    // SqlSession 执行映射文件中定义的 SQL，并返回映射结果
    List<User> list =
     sqlSession.selectList("com.itheima.mapper.UserMapper.findAllUser");
    for (User user : list) {
        System.out.println(user);
    }
    // 关闭 SqlSession
    sqlSession.close();
}
```

使用 JUnit4 执行上述方法后，控制台的输出结果如图 7-5 所示。

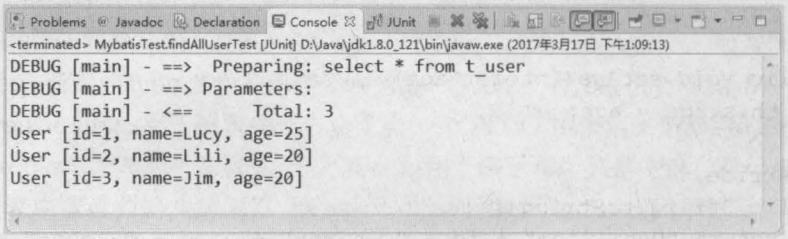


图7-5　运行结果

从图 7-5 可以看出，虽然 t_user 表的列名与 User 对象的属性名完全不一样，但查询出的数据还是被正确地封装到了 User 对象中。

除此之外，还可以通过<resultMap>元素中的<association>和<collection>处理多表时的关联关系。关于关联关系的内容，将在教材第 9 章中详细讲解，这里就不再叙述。

7.4 本章小结

本章主要对 MyBatis 中的核心对象和核心文件进行了详细讲解。首先讲解了 MyBatis 中的两个重要核心对象 SqlSessionFactory 和 SqlSession；然后介绍了配置文件中的元素及其使用；最后对映射文件中的几个主要元素进行了详细讲解。通过本章的学习，读者将能够了解 MyBatis 中两个核心对象的作用，熟悉配置文件中常用元素的使用，并掌握映射文件中常用元素的使用。

【思考题】

1. 请简述 MyBatis 核心对象 SqlSessionFactory 的获取方式。
2. 请简述 MyBatis 映射文件中的主要元素及其作用。

Java EE

8 Chapter

第 8 章 动态 SQL

学习目标

- 了解常用的动态 SQL 元素及其作用
- 掌握动态 SQL 中主要元素的使用

开发人员在使用 JDBC 或其他类似的框架进行数据库开发时，通常都要根据需求去手动拼装 SQL，这是一个非常麻烦且痛苦的工作，而 MyBatis 提供的对 SQL 语句动态组装的功能，恰能很好地解决这一麻烦工作。在本章中，我们将对 MyBatis 框架的动态 SQL 进行详细讲解。

8.1 动态 SQL 中的元素

动态 SQL 是 MyBatis 的强大特性之一，MyBatis 3 采用了功能强大的基于 OGNL 的表达式来完成动态 SQL，它消除了之前版本中需要了解的大多数元素，使用不到原来一半的元素就能完成所需工作。

MyBatis 动态 SQL 中的主要元素，如表 8-1 所示。

表 8-1 MyBatis 的动态 SQL 元素

元素	说明
<if>	判断语句，用于单条件分支判断
<choose>（<when>、<otherwise>）	相当于 Java 中的 switch...case...default 语句，用于多条件分支判断
<where>、<trim>、<set>	辅助元素，用于处理一些 SQL 拼装、特殊字符问题
<foreach>	循环语句，常用于 in 语句等列举条件中
<bind>	从 OGNL 表达式中创建一个变量，并将其绑定到上下文，常用于模糊查询的 sql 中

表 8-1 列举了 MyBatis 动态 SQL 的一些主要元素，并分别对其作用进行了简要介绍。为了帮助读者更好地掌握动态 SQL 的使用，接下来的几个小节将对这些动态 SQL 元素的使用进行详细讲解。

8.2 <if>元素

在 MyBatis 中，<if>元素是最常用的判断语句，它类似于 Java 中的 if 语句，主要用于实现某些简单的条件选择。

在实际应用中，我们可能会通过多个条件来精确地查询某个数据。例如，要查找某个客户的信息，可以通过姓名和职业来查找客户，也可以不填写职业直接通过姓名来查找客户，还可以都不填写而查询出所有客户，此时姓名和职业就是非必须条件。类似于这种情况，在 MyBatis 中就可以通过<if>元素来实现。下面就通过一个具体的案例，来演示这种情况下<if>元素的使用，具体实现步骤如下。

（1）在 Eclipse 中，创建一个名为 chapter08 的 Web 项目，将第 6 章 MyBatis 入门程序中的 JAR 包和文件复制到 chapter08 中，并将配置文件中的数据库信息修改为外部引入的形式，同时创建一个 com.itheima.utils 包，在该包下引入第 7 章编写的工具类 MybatisUtils，这样就完成了项目的创建和基本配置。搭建后的项目文件结构如图 8-1 所示。

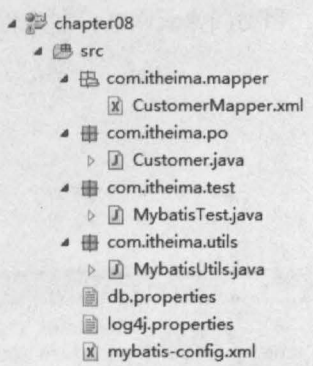


图8-1 chapter08项目文件结构

（2）修改映射文件 CustomerMapper.xml，在映射文件中使用<if>元素编写根据客户姓名和职业组合条件查询客户信息列表的动态 SQL，如文件 8-1 所示。

文件 8-1 CustomerMapper.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
   "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.itheima.mapper.CustomerMapper">
    <!-- <if>元素使用 -->
    <select id="findCustomerByNameAndJobs"
            parameterType="com.itheima.po.Customer"
            resultType="com.itheima.po.Customer">
        select * from t_customer where 1=1
        <if test="username !=null and username !=''">
            and username like concat('%',#{username}, '%')
        </if>
        <if test="jobs !=null and jobs !=''">
            and jobs= #{jobs}
        </if>
    </select>
</mapper>
```

在文件 8-1 中，使用<if>元素的 test 属性分别对 username 和 jobs 进行了非空判断（test 属性多用于条件判断语句中，用于判断真假，大部分的场景中都是进行非空判断，有时候也需要判断字符串、数字和枚举等），如果传入的查询条件非空就进行动态 SQL 组装。

（3）在测试类 MybatisTest 中，编写测试方法 findCustomerByNameAndJobsTest()，如文件 8-2 所示。

文件 8-2 MybatisTest.java

```
package com.itheima.test;
import java.util.List;
import org.apache.ibatis.session.SqlSession;
import org.junit.Test;
import com.itheima.po.Customer;
import com.itheima.utils.MybatisUtils;
public class MybatisTest {
    /**
     * 根据客户姓名和职业组合条件查询客户信息列表
     */
    @Test
    public void findCustomerByNameAndJobsTest(){
        // 通过工具类生成 SqlSession 对象
        SqlSession session = MybatisUtils.getSession();
        // 创建 Customer 对象，封装需要组合查询的条件
        Customer customer = new Customer();
        customer.setUsername("jack");
        customer.setJobs("teacher");
        // 执行 SqlSession 的查询方法，返回结果集
      List<Customer> customers = session.selectList("com.itheima.mapper"
```

```
21                  + ".CustomerMapper.findCustomerByNameAndJobs",customer);
22          // 输出查询结果信息
23          for (Customer customer2 : customers) {
24              // 打印输出结果
25              System.out.println(customer2);
26          }
27          // 关闭 SqlSession
28          session.close();
29      }
30  }
```

在文件 8-2 的 findCustomerByNameAndJobsTest()方法中，首先通过 MybatisUtils 工具类获取了 SqlSession 对象，然后使用 Customer 对象封装了用户名为 jack 且职业为 teacher 的查询条件，并通过 SqlSession 对象的 selectList()方法执行多条件组合的查询操作。为了查看查询结果，这里使用了输出语句输出查询结果信息。最后程序执行完毕时，关闭了 SqlSession 对象。

使用 JUnit4 执行 findCustomerByNameAndJobsTest()方法后，控制台的输出结果如图 8-2 所示。

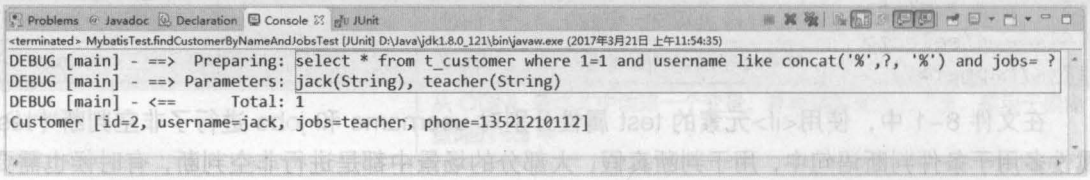


图8-2 运行结果

从图 8-2 可以看出，已经查询出了 username 为 jack，并且 jobs 为 teacher 的客户信息。如果将封装到 Customer 对象中的 jack 和 teacher 两行代码注释（即传入的客户姓名和职业都为空），然后再次使用 JUnit4 执行 findCustomerByNameAndJobsTest()方法后，控制台的输出结果如图 8-3 所示。

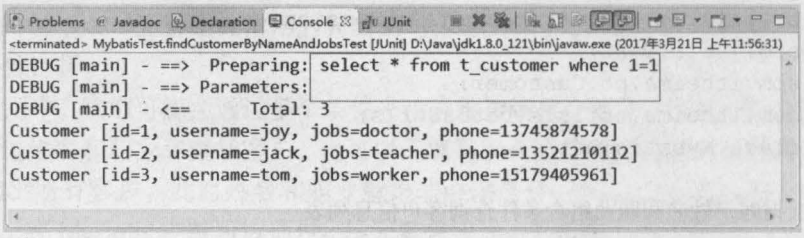


图8-3 运行结果

从图 8-3 可以看到，当未传递任何参数时，程序会将数据表中的所有数据查出。这就是<if>元素的使用。

8.3 <choose>、<when>、<otherwise>元素

在使用<if>元素时，只要 test 属性中的表达式为 true，就会执行元素中的条件语句，但是在实际应用中，有时只需要从多个选项中选择一个去执行。

例如下面的场景：

“当客户名称不为空，则只根据客户名称进行客户筛选；

当客户名称为空，而客户职业不为空，则只根据客户职业进行客户筛选。

当客户名称和客户职业都为空，则要求查询出所有电话不为空的客户信息。”

此种情况下，使用<if>元素进行处理是非常不合适的。如果使用的是 Java 语言，这种情况显然更适合使用 switch...case...default 语句来处理。那么在 MyBatis 中有没有类似的语句呢？答案是肯定的。针对上面情况，MyBatis 中可以使用<choose>、<when>、<otherwise>元素进行处理。下面让我们看一下如何使用<choose>、<when>、<otherwise>元素组合去实现上面的情况。

（1）在映射文件 CustomerMapper.xml 中，使用<choose>、<when>、<otherwise>元素执行上述情况的动态 SQL 代码如下所示。

```
<!--<choose>(<when>、<otherwise>)元素使用 -->
<select id="findCustomerByNameOrJobs"
        parameterType="com.itheima.po.Customer"
        resultType="com.itheima.po.Customer">
    select * from t_customer where 1=1
    <choose>
        <when test="username !=null and username !=''">
            and username like concat('%',#{username}, '%')
        </when>
        <when test="jobs !=null and jobs !=''">
            and jobs= #{jobs}
        </when>
        <otherwise>
            and phone is not null
        </otherwise>
    </choose>
</select>
```

在上述代码中，使用了<choose>元素进行 SQL 拼接，当第一个<when>元素中的条件为真，则只动态组装第一个<when>元素内的 SQL 片段，否则就继续向下判断第二个<when>元素中的条件是否为真，以此类推。当前面所有 when 元素中的条件都不为真时，则只组装<otherwise>元素内的 SQL 片段。

（2）在测试类 MybatisTest 中，编写测试方法 findCustomerByNameOrJobsTest()，其代码如下所示。

```
/**
 * 根据客户姓名或职业查询客户信息列表
 */
@Test
public void findCustomerByNameOrJobsTest(){
    // 通过工具类生成 SqlSession 对象
    SqlSession session = MybatisUtils.getSession();
    // 创建 Customer 对象，封装需要组合查询的条件
    Customer customer = new Customer();
    customer.setUsername("jack");
```

```
        customer.setJobs("teacher");
        // 执行 SqlSession 的查询方法，返回结果集
        List<Customer> customers = session.selectList("com.itheima.mapper"
                + ".CustomerMapper.findCustomerByNameOrJobs",customer);
        // 输出查询结果信息
        for (Customer customer2 : customers) {
            // 打印输出结果
            System.out.println(customer2);
        }
        // 关闭 SqlSession
        session.close();
    }
```

使用 JUnit4 执行 findCustomerByNameOrJobsTest()方法后，控制台的输出结果如图 8-4 所示。

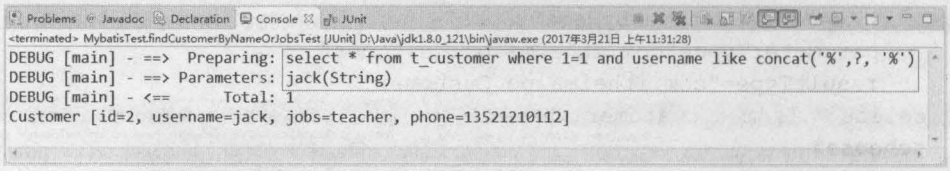


图8-4 运行结果

从图 8-4 可以看出，虽然同时传入了姓名和职业两个查询条件，但 MyBatis 所生成的 SQL 只是动态组装了客户姓名进行条件查询。

如果将上述代码中“customer.setUsername("jack");”删除或者注释掉，然后再次执行 findCustomerByNameOrJobsTest()方法时，控制台的输出结果如图 8-5 所示。

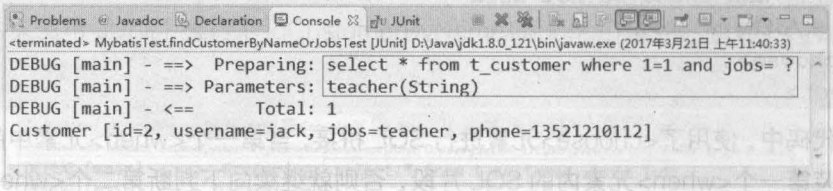


图8-5 运行结果

从图 8-5 可以看出，MyBatis 生成的 SQL 组装了客户职业进行条件查询，同样查询出了客户信息。

如果将设置客户姓名和职业参数值的两行代码都注释（即客户姓名和职业都为空），那么程序的执行结果如图 8-6 所示。

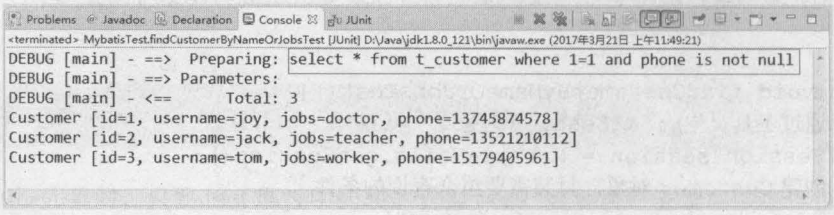


图8-6 运行结果

从图 8-6 可以看出，当姓名和职业参数都为空时，MyBatis 的 SQL 组装了<otherwise>元素中的 SQL 片段进行条件查询。

8.4 <where>、<trim>元素

在前两个小节的案例中，映射文件中编写的 SQL 后面都加入了“where 1=1”的条件，那么到底为什么要这么写呢？如果将 where 后“1=1”的条件去掉，那么 MyBatis 所拼接出来的 SQL 将会如下所示。

```
select * from t_customer where and username like concat('%',?, '%')
```

上面 SQL 中，where 后直接跟的是 and，这在运行时肯定会报 SQL 语法错误，而加入了条件“1=1”后，既保证了 where 后面的条件成立，又避免了 where 后面第一个词是 and 或者 or 之类的关键词。那么在 MyBatis 中，有没有什么办法不用加入“1=1”这样的条件，也能使拼接后的 SQL 成立呢？

针对这种情况，MyBatis 提供了<where>元素来处理这样的问题。

以 8.2 小节的案例为例，将映射文件中的“where 1=1”条件删除，并使用<where>元素替换后的代码如下所示。

```
<!-- <where>元素-->
<select id="findCustomerByNameAndJobs"
        parameterType="com.itheima.po.Customer"
        resultType="com.itheima.po.Customer">
    select * from t_customer
    <where>
        <if test="username !=null and username !=''">
          and username like concat('%',#{username}, '%')
        </if>
        <if test="jobs !=null and jobs !=''">
          and jobs= #{jobs}
        </if>
    </where>
</select>
```

上述配置代码中，使用<where>元素对“where 1=1”条件进行了替换，<where>元素会自动判断组合条件下拼装的 SQL 语句，只有<where>元素内的条件成立时，才会在拼接 SQL 中加入 where 关键字，否则将不会添加；即使 where 之后的内容有多余的“AND”或“OR”，<where>元素也会自动将它们去除。

除了使用<where>元素外，还可以通过<trim>元素来定制需要的功能，上述代码还可以修改为如下形式：

```
<!-- <trim>元素-->
<select id="findCustomerByNameAndJobs"
        parameterType="com.itheima.po.Customer"
        resultType="com.itheima.po.Customer">
    select * from t_customer
    <trim prefix="where" prefixOverrides="and">
```

```
        <if test="username !=null and username !=''">
            and username like concat('%',#{username}, '%')
        </if>
        <if test="jobs !=null and jobs !=''">
            and jobs= #{jobs}
        </if>
    </trim>
</select>
```

上述配置代码中，同样使用<trim>元素对“where 1=1”条件进行了替换，<trim>元素的作用是去除一些特殊的字符串，它的 prefix 属性代表的是语句的前缀（这里使用 where 来连接后面的 SQL 片段），而 prefixOverrides 属性代表的是需要去除的那些特殊字符串（这里定义了要去除 SQL 中的 and），上面的写法和使用<where>元素基本是等效的。

8.5 <set>元素

在 Hibernate 中，如果想要更新某一个对象，就需要发送所有的字段给持久化对象，然而实际应用中，大多数情况下都是更新的某一个或几个字段。如果更新的每一条数据都要将其所有的属性都更新一遍，那么其执行效率是非常差的。有没有办法让程序只更新需要更新的字段呢?

为了解决上述情况中的问题，MyBatis 中提供了<set>元素来完成这一工作。<set>元素主要用于更新操作，其主要作用是在动态包含的 SQL 语句前输出一个 SET 关键字，并将 SQL 语句中最后一个多余的逗号去除。

以入门案例中的更新操作为例，使用<set>元素对映射文件中更新客户信息的 SQL 语句进行修改的代码如下所示。

```
<!-- <set>元素 -->
<update id="updateCustomer"  parameterType="com.itheima.po.Customer">
    update t_customer
    <set>
        <if test="username !=null and username !=''">
            username=#{username},
        </if>
        <if test="jobs !=null and jobs !=''">
            jobs=#{jobs},
        </if>
        <if test="phone !=null and phone !=''">
            phone=#{phone},
        </if>
    </set>
    where id=#{id}
</update>
```

在上述配置的 SQL 语句中，使用了<set>和<if>元素相结合的方式来组装 update 语句。其中<set>元素会动态前置 SET 关键字，同时也会消除 SQL 语句中最后一个多余的逗号；<if>元素用于判断相应的字段是否传入值，如果传入的更新字段非空，就将此字段进行动态 SQL 组装，并更新此字段，否则此字段不执行更新。

为了验证上述配置，可以在测试类中编写测试方法 updateCustomerTest()，其代码如下所示。

```
/**
 * 更新客户
 */
@Test
public void updateCustomerTest(){
    // 获取 SqlSession
    SqlSession sqlSession = MybatisUtils.getSession();
    // 创建 Customer 对象，并向对象中添加数据
    Customer customer = new Customer();
    customer.setId(3);
    customer.setPhone("13311111234");
    // 执行 SqlSession 的更新方法，返回的是 SQL 语句影响的行数
    int rows = sqlSession.update("com.itheima.mapper"
          + ".CustomerMapper.updateCustomer", customer);
    // 通过返回结果判断更新操作是否执行成功
    if(rows > 0){
        System.out.println("您成功修改了"+rows+"条数据！");
    }else{
        System.out.println("执行修改操作失败！！！");
    }
    // 提交事务
    sqlSession.commit();
    // 关闭 SqlSession
    sqlSession.close();
}
```

从上述代码可以看出，与入门案例中的更新方法有所不同的是，这里只设置了其 id 和 Phone 的属性值，也就是需要修改 id 为 3 的客户电话信息。

使用 JUnit4 执行 updateCustomerTest()方法后，控制台的输出结果如图 8-7 所示。

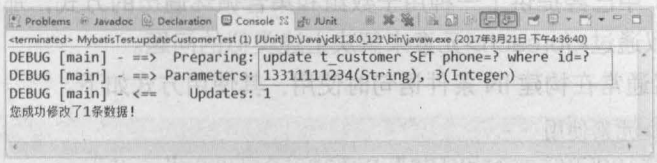


图8-7 运行结果

从图 8-7 可以看出，控制台中已经提示成功更新了 1 条数据。为了验证是否真的执行成功，此时查看数据库 t_customer 表中的数据，如图 8-8 所示。

从图 8-8 可以看出，使用<set>元素已成功对数据表中 id 为 3 的客户电话进行了修改。

由于篇幅有限，本案例中未演示不使用<set>元素时，执行单个字段更新的情况，读者可以自行修改入门案例进行测试。

注意

在映射文件中使用<set>和<if>元素组合进行 update 语句动态 SQL 组装时，如果<set>元素内包含的内容都为空，则会出现 SQL 语法错误。所以在使用<set>元素进行字段信息更新时，要确保传入的更新字段不能都为空。

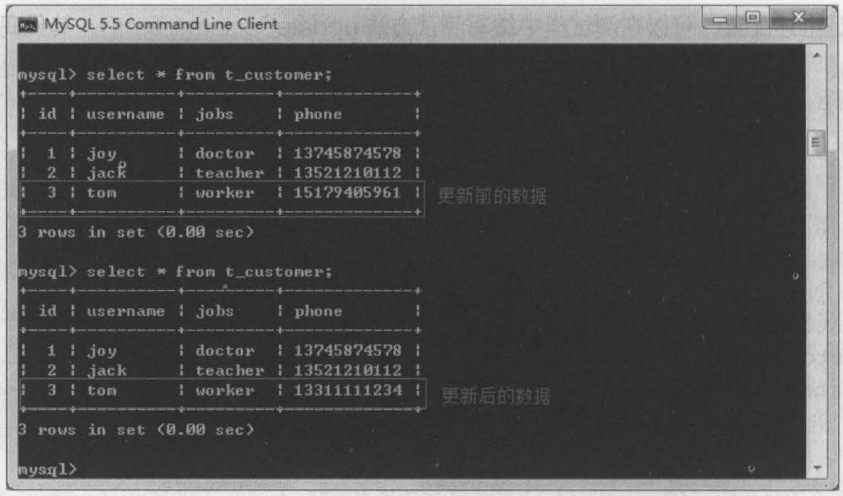


图8-8 t_customer表

8.6 <foreach>元素

在实际开发中，有时可能会遇到这样的情况：假设在一个客户表中有 1000 条数据，现在需要将 id 值小于 100 的客户信息全部查询出来，这要怎么做呢？有人也许会说，“我可以一条一条查出来”，那如果查询 200、300 甚至更多也一条一条查吗？这显然是不可取的。有的人会想到，可以在 Java 方法中使用循环，将查询方法放在循环语句中，然后通过条件循环的方式查询出所需的数据。这种查询方式虽然可行，但每执行一次循环语句，都需要向数据库中发送一条查询 SQL，其查询效率是非常低的。那么还有其他更好的方法吗？我们能不能通过 SQL 语句来执行这种查询呢？

其实，MyBatis 中已经提供了一种用于数组和集合循环遍历的方式，那就是使用<foreach>元素，我们完全可以通过<foreach>元素来解决上述类似的问题。

<foreach>元素通常在构建 IN 条件语句时使用，其使用方式如下。

```
<!--<foreach>元素使用 -->
<select id="findCustomerByIds" parameterType="List"
        resultType="com.itheima.po.Customer">
    select * from t_customer where id in
    <foreach item="id" index="index" collection="list"
             open="(" separator="," close=")">
        #{id}
    </foreach>
</select>
```

在上述代码中，使用了<foreach>元素对传入的集合进行遍历并进行了动态 SQL 组装。关于<foreach>元素中使用的几种属性的描述具体如下。

- item：配置的是循环中当前的元素。
- index：配置的是当前元素在集合的位置下标。
- collection：配置的 list 是传递过来的参数类型（首字母小写），它可以是一个 array、list

(或 collection)、Map 集合的键、POJO 包装类中数组或集合类型的属性名等。

- open 和 close：配置的是以什么符号将这些集合元素包装起来。
- separator：配置的是各个元素的间隔符。

注意

你可以将任何可迭代对象（如列表、集合等）和任何的字典或者数组对象传递给<foreach>作为集合参数。当使用可迭代对象或者数组时，index 是当前迭代的次数，item 的值是本次迭代获取的元素。当使用字典（或者 Map.Entry 对象的集合）时，index 是键，item 是值。

为了验证上述配置，可以在测试类 MybatisTest 中，编写测试方法 findCustomerByIdsTest()，其代码如下所示。

```
/**
 * 根据客户编号批量查询客户信息
 */
@Test
public void findCustomerByIdsTest(){
    // 获取 SqlSession
    SqlSession session = MybatisUtils.getSession();
    // 创建 List 集合，封装查询 id
    List<Integer> ids=new ArrayList<Integer>();
    ids.add(1);
    ids.add(2);
    // 执行 SqlSession 的查询方法，返回结果集
    List<Customer> customers = session.selectList("com.itheima.mapper"
            + ".CustomerMapper.findCustomerByIds", ids);
    // 输出查询结果信息
    for (Customer customer : customers) {
        // 打印输出结果
        System.out.println(customer);
    }
    // 关闭 SqlSession
    session.close();
}
```

在上述代码中，执行查询操作时传入了一个客户编号集合 ids。

使用 JUnit4 执行 findCustomerByIdsTest()方法后，控制台的输出结果如图 8-9 所示。

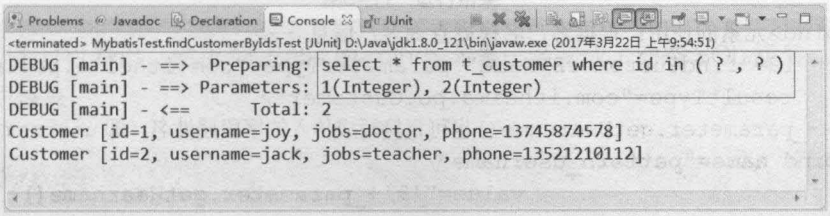


图8-9 运行结果

从图 8-9 可以看出，使用<foreach>元素已对传入的客户编号集合进行了动态 SQL 组装，最终成功批量查询出了对应的客户信息。

脚下留心：

在使用<foreach>时最关键也是最容易出错的就是 collection 属性，该属性是必须指定的，而且在不同情况下，该属性的值是不一样的。主要有以下 3 种情况。

（1）如果传入的是单参数且参数类型是一个数组或者 List 的时候，collection 属性值分别为 array 和 list（或 collection）。

（2）如果传入的参数是多个的时候，就需要把它们封装成一个 Map 了，当然单参数也可以封装成 Map 集合，这时候 collection 属性值就为 Map 的键。

（3）如果传入的参数是 POJO 包装类的时候，collection 属性值就为该包装类中需要进行遍历的数组或集合的属性名。

所以在设置 collection 属性值的时候，必须按照实际情况配置，否则程序就会出现异常。例如，将上述<foreach>元素中 collection 的属性值设置为 array，则程序执行后，将出现图 8-10 所示的异常。

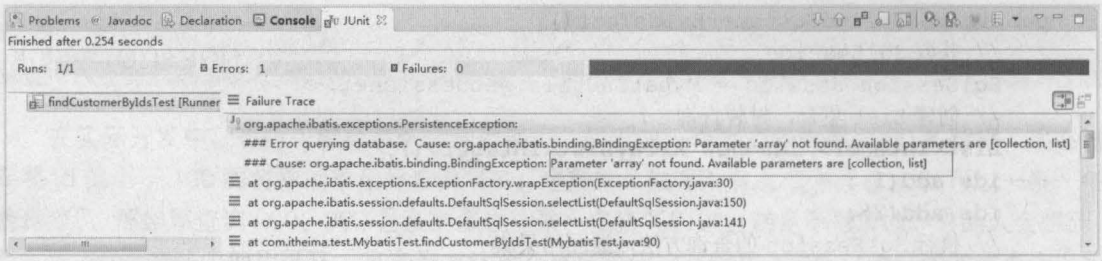


图8-10　运行结果

8.7 <bind>元素

在进行模糊查询编写 SQL 语句的时候，如果使用“${}”进行字符串拼接，则无法防止 SQL 注入问题；如果使用 concat 函数进行拼接，则只针对 MySQL 数据库有效；如果使用的是 Oracle 数据库，则要使用连接符号“||”。这样，映射文件中的 SQL 就要根据不同的情况提供不同形式的实现，这显然是比较麻烦的，且不利于项目的移植。为此，MyBatis 提供了<bind>元素来解决这一问题，我们完全不必使用数据库语言，只要使用 MyBatis 的语言即可与所需参数连接。

MyBatis 的<bind>元素可以通过 OGNL 表达式来创建一个上下文变量，其使用方式如下：

```
<!--<bind>元素的使用：根据客户名模糊查询客户信息 -->
<select id="findCustomerByName" parameterType="com.itheima.po.Customer"
        resultType="com.itheima.po.Customer">
    <!--_parameter.getUsername()也可直接写成传入的字段属性名，即 username-->
     <bind name="pattern_username"
                          value="'%'+_parameter.getUsername()+'%'" />
    select * from t_customer
    where
    username like #{pattern_username}
</select>
```

上述配置代码中，使用<bind>元素定义了一个 name 为 pattern_username 的变量，<bind>元素中 value 的属性值就是拼接的查询字符串，其中_parameter.getUsername()表示传递进来的参数（也可以直接写成对应的参数变量名，如 username）。在 SQL 语句中，直接引用<bind>元素的 name 属性值即可进行动态 SQL 组装。

为了验证上述配置是否能够正确执行，可以在测试类 MybatisTest 中，编写测试方法 findCustomerByNameTest()进行测试，其代码如下所示。

```
/**
 * <bind>元素的使用：根据客户名模糊查询客户信息
 */
@Test
public void findCustomerByNameTest(){
    // 通过工具类生成 SqlSession 对象
    SqlSession session = MybatisUtils.getSession();
    // 创建 Customer 对象，封装查询的条件
    Customer customer =new Customer();
    customer.setUsername("j");
    // 执行 SqlSession 的查询方法，返回结果集
    List<Customer> customers = session.selectList("com.itheima.mapper"
            + ".CustomerMapper.findCustomerByName", customer);
    // 输出查询结果信息
    for (Customer customer2 : customers) {
        // 打印输出结果
        System.out.println(customer2);
    }
    // 关闭 SqlSession
    session.close();
}
```

使用 JUnit4 执行 findCustomerByNameTest()方法后，控制台的输出结果如图 8-11 所示。

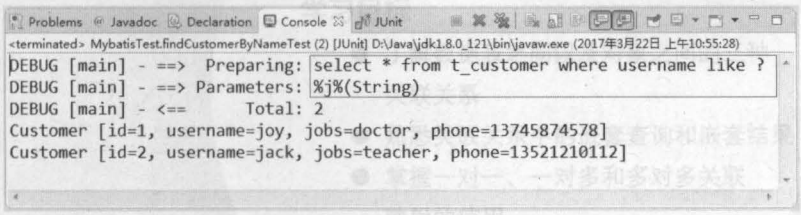


图8-11 运行结果

从图 8-11 可以看出，使用 MyBatis 的<bind>元素已经完成了动态 SQL 组装，并成功模糊查询出了客户信息。

8.8 本章小结

本章首先对 MyBatis 框架的动态 SQL 元素做了简要介绍，然后分别对这些主要的动态 SQL 元素进行了详细讲解。通过本章的学习，读者可以了解常用动态 SQL 元素的主要作用，并能够掌握这些元素在实际开发中如何使用。在 MyBatis 框架中，这些动态 SQL 元素的使用十分重要，

熟练地掌握它们能够极大地提高开发效率。

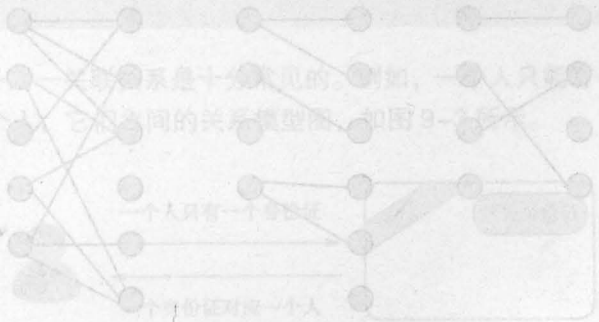
【思考题】

1. 请简述 MyBatis 框架动态 SQL 中的主要元素及其作用。
2. 请简述 MyBatis 框架动态 SQL 中<foreach>元素 collection 属性的注意事项。

Java EE

Chapter 9

第 9 章
MyBatis 的关联映射



学习目标

- 了解数据表之间以及对象之间的三种关联关系
- 熟悉关联关系中的嵌套查询和嵌套结果
- 掌握一对一、一对多和多对多关联映射的使用

通过前面几章的学习，读者已经熟悉了 MyBatis 的基本知识，并能够使用 MyBatis 以及面向对象的方式进行数据库操作，但这些操作只是针对单表实现的。在实际的开发中，对数据库的操作常常会涉及多张表，这在面向对象中就涉及了对象与对象之间的关联关系。针对多表之间的操作，MyBatis 提供了关联映射，通过关联映射就可以很好地处理对象与对象之间的关联关系。本章中，将对 MyBatis 的关联关系映射进行详细讲解。

9.1 关联关系概述

在关系型数据库中，多表之间存在着三种关联关系，分别为一对一、一对多和多对多，如图 9-1 所示。

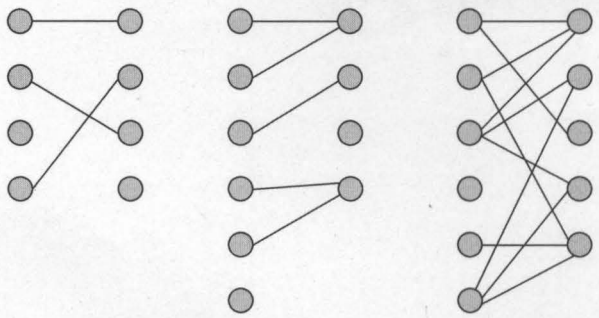


图9-1　关系型数据库中多表之间的三种关联关系

这三种关联关系的具体说明如下。

- 一对一：在任意一方引入对方主键作为外键。
- 一对多：在“多”的一方，添加“一”的一方的主键作为外键。
- 多对多：产生中间关系表，引入两张表的主键作为外键，两个主键成为联合主键或使用新的字段作为主键。

通过数据库中的表可以描述数据之间的关系，同样，在 Java 中，通过对象也可以进行关系描述，如图 9-2 所示。

```
class A{                class A{                class A{
   B b;                    List<B> b;              List<B> b;
}                       }                       }

class B{                class B{                class B{
   A a;                    A a;                    List<A> a;
}                       }                       }

  一对一                  一对多                  多对多
```

图9-2　Java对象描述数据表之间的关系

在图 9-2 中，三种关联关系的描述如下。

- 一对一的关系：就是在本类中定义对方类型的对象，如 A 类中定义 B 类类型的属性 b，B 类中定义 A 类类型的属性 a。
- 一对多的关系：就是一个 A 类类型对应多个 B 类类型的情况，需要在 A 类中以集合的方式引入 B 类类型的对象，在 B 类中定义 A 类类型的属性 a。
- 多对多的关系：在 A 类中定义 B 类类型的集合，在 B 类中定义 A 类类型的集合。

以上就是 Java 对象中，三种实体类之间的关联关系，那么使用 MyBatis 是如何处理 Java 对象中的三种关联关系呢？在接下来的 3 个小节中，将对 MyBatis 中的这几种关联关系的使用进行详细讲解。

9.2 一对一

在现实生活中，一对一关联关系是十分常见的。例如，一个人只能有一个身份证，同时一个身份证也只会对应一个人，它们之间的关系模型图，如图 9-3 所示。

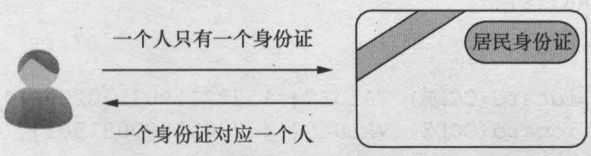


图9-3 人与身份证的关联关系

那么使用 MyBatis 是怎么处理图 9-3 中的这种一对一关联关系的呢？在本书第 7 章所讲解的<resultMap>元素中，包含了一个<association>子元素，MyBatis 就是通过该元素来处理一对一关联关系的。

在<association>元素中，通常可以配置以下属性。

- property：指定映射到的实体类对象属性，与表字段一一对应。
- column：指定表中对应的字段。
- javaType：指定映射到实体对象属性的类型。
- select：指定引入嵌套查询的子 SQL 语句，该属性用于关联映射中的嵌套查询。
- fetchType：指定在关联查询时是否启用延迟加载。fetchType 属性有 lazy 和 eager 两个属性值，默认值为 lazy（即默认关联映射延迟加载）。

<association>元素的使用非常简单，只需要参考如下两种示例配置即可，具体如下。

```
<!--方式一：嵌套查询-->
<association property="card" column="card_id"
             javaType="com.itheima.po.IdCard"
             select="com.itheima.mapper.IdCardMapper.findCodeById" />
<!--方式二：嵌套结果-->
<association property="card" javaType="com.itheima.po.IdCard">
   <id property="id" column="card_id" />
   <result property="code" column="code" />
</association>
```

小提示

MyBatis 在映射文件中加载关联关系对象主要通过两种方式：嵌套查询和嵌套结果。嵌套查询是指通过执行另外一条 SQL 映射语句来返回预期的复杂类型；嵌套结果是使用嵌套结果映射来处理重复的联合结果的子集。开发人员可以使用上述任意一种方式实现对关联关系的加载。

了解了 MyBatis 中处理一对一关联关系的元素和方式后，接下来就以个人和身份证之间的一对一关联关系为例，进行详细讲解。

查询个人及其关联的身份证信息是先通过查询个人表中的主键来获个人信息，然后通过表中的外键，来获取证件表中的身份证号信息。其具体实现步骤如下。

（1）创建数据表。在 mybatis 数据库中分别创建名为 tb_idcard 和 tb_person 的数据表，同时预先插入两条数据。其执行的 SQL 语句如下所示。

```
USE mybatis;
# 创建一个名称为 tb_idcard 的表
CREATE TABLE  tb_idcard(
    id INT PRIMARY KEY AUTO_INCREMENT,
    CODE VARCHAR(18)
);
# 插入两条数据
INSERT INTO tb_idcard(CODE) VALUES('152221198711020624');
INSERT INTO tb_idcard(CODE) VALUES('152201199008150317');
# 创建一个名称为 tb_person 的表
CREATE TABLE  tb_person(
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(32),
    age INT,
    sex VARCHAR(8),
    card_id INT UNIQUE,
    FOREIGN KEY(card_id) REFERENCES tb_idcard(id)
);
# 插入两条数据
INSERT INTO tb_person(name,age,sex,card_id) VALUES('Rose',29,'女',1);
INSERT INTO tb_person(name,age,sex,card_id) VALUES('tom',27,'男',2);
```

完成上述操作后，数据库 tb_idcard 和 tb_person 表中的数据如图 9-4 所示。

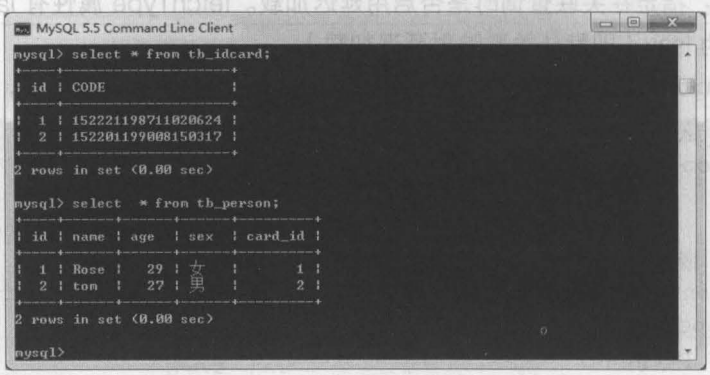


图9-4 tb_idcard和tb_person表

（2）在 Eclipse 中创建一个名为 chapter09 的 Web 项目，然后引入相关 JAR 包、log4j 日志文件、MybatisUtils 工具类以及 mybatis-config.xml 核心配置文件。项目环境搭建完成后的文件结构，如图 9-5 所示。

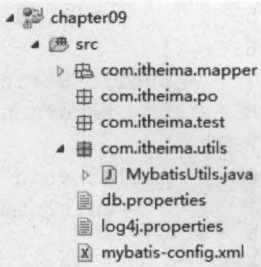


图9-5　项目文件结构

（3）在项目的 com.itheima.po 包下创建持久化类 IdCard 和 Person，编辑后的代码，如文件 9-1 和文件 9-2 所示。

文件 9-1　IdCard.java

```
1  package com.itheima.po;
2  /**
3   * 证件持久化类
4   */
5  public class IdCard {
6      private Integer id;
7      private String code;
8      public Integer getId() {
9          return id;
10     }
11     public void setId(Integer id) {
12         this.id = id;
13     }
14     public String getCode() {
15         return code;
16     }
17     public void setCode(String code) {
18         this.code = code;
19     }
20     @Override
21     public String toString() {
22         return "IdCard [id=" + id + ", code=" + code + "]";
23     }
24 }
```

文件 9-2　Person.java

```
1  package com.itheima.po;
2  /**
3   * 个人持久化类
4   */
5  public class Person {
6      private Integer id;
7      private String name;
8      private Integer age;
9      private String sex;
10     private IdCard card;  //个人关联的证件
11     public Integer getId() {
12         return id;
13     }
14     public void setId(Integer id) {
```

```
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Integer getAge() {
        return age;
    }
    public void setAge(Integer age) {
        this.age = age;
    }
    public String getSex() {
        return sex;
    }
    public void setSex(String sex) {
        this.sex = sex;
    }
    public IdCard getCard() {
        return card;
    }
    public void setCard(IdCard card) {
        this.card = card;
    }
    @Override
    public String toString() {
        return "Person [id=" + id + ", name=" + name + ", "
                + "age=" + age + ", sex=" + sex + ", card=" + card + "]";
    }
}
```

在上述两个文件中，分别定义了各自的属性以及对应的 getter/setter 方法，同时为了方便查看输出结果还重写了 toString()方法。

（4）在 com.itheima.mapper 包中，创建证件映射文件 IdCardMapper.xml 和个人映射文件 PersonMapper.xml，并在两个映射文件中编写一对一关联映射查询的配置信息，如文件 9-3 和文件 9-4 所示。

文件 9-3　IdCardMapper.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
   "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.itheima.mapper.IdCardMapper">
  <!-- 根据 id 查询证件信息 -->
  <select id="findCodeById" parameterType="Integer" resultType="IdCard">
      SELECT * from tb_idcard where id=#{id}
  </select>
</mapper>
```

文件 9-4 PersonMapper.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
   "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.itheima.mapper.PersonMapper">
    <!-- 嵌套查询：通过执行另外一条 SQL 映射语句来返回预期的特殊类型 -->
    <select id="findPersonById" parameterType="Integer"
                                resultMap="IdCardWithPersonResult">
        SELECT * from tb_person where id=#{id}
    </select>
    <resultMap type="Person" id="IdCardWithPersonResult">
        <id property="id" column="id" />
        <result property="name" column="name" />
        <result property="age" column="age" />
        <result property="sex" column="sex" />
        <!-- 一对一：association 使用 select 属性引入另外一条 SQL 语句 -->
        <association property="card" column="card_id" javaType="IdCard"
            select="com.itheima.mapper.IdCardMapper.findCodeById" />
    </resultMap>
</mapper>
```

在上述两个映射文件中，使用了 MyBatis 中的嵌套查询方式进行了个人及其关联的证件信息查询，因为返回的个人对象中除了基本属性外还有一个关联的 card 属性，所以需要手动编写结果映射。从映射文件 PersonMapper.xml 中可以看出，嵌套查询的方法是先执行一个简单的 SQL 语句，然后在进行结果映射时，将关联对象在<association>元素中使用 select 属性执行另一条 SQL 语句（即 IdCardMapper.xml 中的 SQL）。

（5）在核心配置文件 mybatis-config.xml 中，引入 Mapper 映射文件并定义别名，如文件 9-5 所示。

文件 9-5 mybatis-config.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
   "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <!-- 引入数据库连接配置文件 -->
    <properties resource="db.properties" />
    <!--使用扫描包的形式定义别名 -->
    <typeAliases>
        <package name="com.itheima.po" />
    </typeAliases>
    <!--配置环境 ，默认的环境 id 为 mysql -->
    <environments default="mysql">
        <!-- 配置 id 为 mysql 的数据库环境 -->
        <environment id="mysql">
            <!-- 使用 JDBC 的事务管理 -->
            <transactionManager type="JDBC" />
            <!--数据库连接池 -->
            <dataSource type="POOLED">
```

```
                <property name="driver" value="${jdbc.driver}" />
                <property name="url" value="${jdbc.url}" />
                <property name="username" value="${jdbc.username}" />
                <property name="password" value="${jdbc.password}" />
            </dataSource>
        </environment>
    </environments>
    <!--配置 Mapper 的位置 -->
    <mappers>
        <mapper resource="com/itheima/mapper/IdCardMapper.xml" />
        <mapper resource="com/itheima/mapper/PersonMapper.xml" />
    </mappers>
</configuration>
```

在上述核心配置文件中，首先引入了数据库连接的配置文件，然后使用扫描包的形式自定义别名，接下来进行环境的配置，最后配置了 Mapper 映射文件的位置信息。

（6）在 com.itheima.test 包中，创建测试类 MybatisAssociatedTest，并在类中编写测试方法 findPersonByIdTest()，如文件 9-6 所示。

文件 9-6　MybatisAssociatedTest.java

```
package com.itheima.test;
import org.apache.ibatis.session.SqlSession;
import org.junit.Test;
import com.itheima.po.Person;
import com.itheima.utils.MybatisUtils;
/**
 * Mybatis 关联查询映射测试类
 */
public class MybatisAssociatedTest {
    /**
     * 嵌套查询
     */
    @Test
    public void findPersonByIdTest() {
        // 1. 通过工具类生成 SqlSession 对象
        SqlSession session = MybatisUtils.getSession();
        // 2. 使用 MyBatis 嵌套查询的方式查询 id 为 1 的人的信息
        Person person = session.selectOne("com.itheima.mapper."
                                + "PersonMapper.findPersonById", 1);
        // 3. 输出查询结果信息
        System.out.println(person);
        // 4. 关闭 SqlSession
        session.close();
    }
}
```

在文件 9-6 的 findPersonByIdTest() 方法中，首先通过 MybatisUtils 工具类获取了 SqlSession 对象，然后通过 SqlSession 对象的 selectOne()方法获取了个人信息。为了查看结果，这里使用了输出语句输出查询结果信息。最后程序执行完毕时，关闭了 SqlSession。

使用 JUnit4 执行 findPersonByIdTest()方法后，控制台的输出结果如图 9-6 所示。

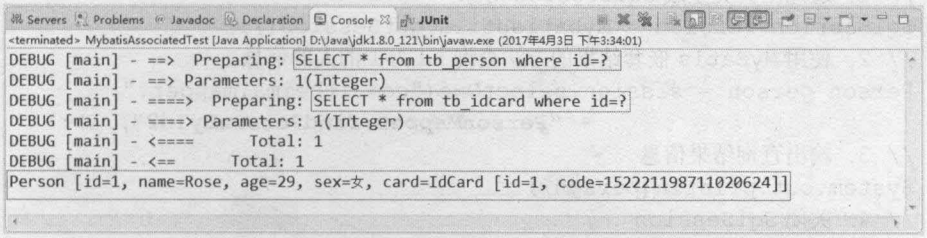


图9-6　运行结果

从图 9-6 可以看出，使用 MyBatis 嵌套查询的方式查询出了个人及其关联的身份证信息，这就是 MyBatis 中的一对一关联查询。

虽然使用嵌套查询的方式比较简单，但是从图 9-6 中可以看出，MyBatis 嵌套查询的方式要执行多条 SQL 语句，这对于大型数据集合和列表展示不是很好，因为这样可能会导致成百上千条关联的 SQL 语句被执行，从而极大地消耗数据库性能并且会降低查询效率。这并不是开发人员所期望的。为此，我们可以使用 MyBatis 提供的嵌套结果方式，来进行关联查询。

在 PersonMapper.xml 中，使用 MyBatis 嵌套结果的方式进行个人及其关联的证件信息查询，所添加的代码如下所示。

```
<!-- 嵌套结果：使用嵌套结果映射来处理重复的联合结果的子集 -->
<select id="findPersonById2" parameterType="Integer"
                             resultMap="IdCardWithPersonResult2">
    SELECT p.*,idcard.code
    from tb_person p,tb_idcard idcard
    where p.card_id=idcard.id
    and p.id= #{id}
</select>
<resultMap type="Person" id="IdCardWithPersonResult2">
    <id property="id" column="id" />
    <result property="name" column="name" />
    <result property="age" column="age" />
    <result property="sex" column="sex" />
    <association property="card" javaType="IdCard">
        <id property="id" column="card_id" />
        <result property="code" column="code" />
    </association>
</resultMap>
```

从上述代码中可以看出，MyBatis 嵌套结果的方式只编写了一条复杂的多表关联的 SQL 语句，并且在<association>元素中继续使用相关子元素进行数据库表字段和实体类属性的一一映射。

在测试类 MybatisAssociatedTest 中，编写测试方法 findPersonByIdTest2()，其代码如下所示。

```
/**
 * 嵌套结果
 */
@Test
```

```
public void findPersonByIdTest2() {
    // 1. 通过工具类生成 SqlSession 对象
    SqlSession session = MybatisUtils.getSession();
    // 2. 使用 MyBatis 嵌套结果的方法查询 id 为 1 的人的信息
    Person person = session.selectOne("com.itheima.mapper."
                        + "PersonMapper.findPersonById2", 1);
    // 3. 输出查询结果信息
    System.out.println(person);
    // 4. 关闭 SqlSession
    session.close();
}
```

使用 JUnit4 执行 findPersonByIdTest2()方法后，控制台的输出结果如图 9-7 所示。

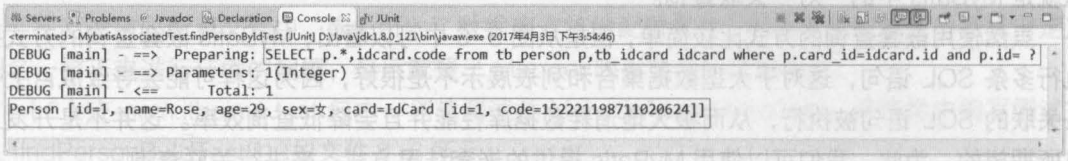
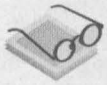


图9-7 运行结果

从图 9-7 可以看出，使用 MyBatis 嵌套结果的方式只执行了一条 SQL 语句，并且同样查询出了个人及其关联的身份证的信息。



多学一招：MyBatis 延迟加载的配置

在使用 MyBatis 嵌套查询方式进行 MyBatis 关联查询映射时，使用 MyBatis 的延迟加载在一定程度上可以降低运行消耗并提高查询效率。MyBatis 默认没有开启延迟加载，需要在核心配置文件 mybatis-config.xml 中的<settings>元素内进行配置，具体配置方式如下。

```
<settings>
    <!-- 打开延迟加载的开关 -->
    <setting name="lazyLoadingEnabled" value="true" />
    <!-- 将积极加载改为消息加载，即按需加载 -->
    <setting name="aggressiveLazyLoading" value="false"/>
</settings>
```

在映射文件中，MyBatis 关联映射的<association>元素和<collection>元素中都已默认配置了延迟加载属性，即默认属性 fetchType="lazy"（属性 fetchType="eager"表示立即加载），所以在配置文件中开启延迟加载后，无须在映射文件中再做配置。

9.3 一对多

与一对一的关联关系相比，开发人员接触更多的关联关系是一对多（或多对一）。例如一个用户可以有多个订单，同时多个订单归一个用户所有。用户和订单的关联关系如图 9-8 所示。

那么使用 MyBatis 是怎么处理这种一对多关联关系的呢？在本书第 7 章所讲解的<resultMap>元素中，包含了一个<collection>子元素，MyBatis 就是通过该元素来处理一对多关联关系的。<collection>子元素的属性大部分与<association>元素相同，但其还包含一个特殊属

性——ofType。ofType 属性与 javaType 属性对应，它用于指定实体对象中集合类属性所包含的元素类型。

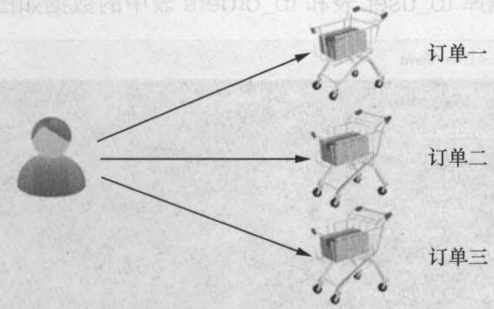


图9-8 用户和订单的关联关系

<collection>元素的使用也非常简单，同样可以参考如下两种示例进行配置，具体代码如下。

```
<!--方式一：嵌套查询 -->
<collection property="ordersList" column="id"
            ofType="com.itheima.po.Orders"
            select=" com.itheima.mapper.OrdersMapper.selectOrders" />
<!--方式二：嵌套结果 -->
<collection property="ordersList" ofType="com.itheima.po.Orders">
   <id property="id" column="orders_id" />
   <result property="number" column="number" />
</collection>
```

在了解了 MyBatis 处理一对多关联关系的元素和方式后，接下来以用户和订单之间的这种一对多关联关系为例，详细讲解如何在 MyBatis 中处理一对多关联关系，具体步骤如下。

（1）在 mybatis 数据库中，创建两个数据表，分别为 tb_user 和 tb_orders，同时在表中预先插入几条数据，执行的 SQL 语句如下所示。

```
# 创建一个名称为 tb_user 的表
CREATE TABLE tb_user (
  id int(32) PRIMARY KEY AUTO_INCREMENT,
  username varchar(32),
  address varchar(256)
 );
# 插入 3 条数据
INSERT INTO tb_user VALUES ('1', '詹姆斯', '克利夫兰');
INSERT INTO tb_user VALUES ('2', '科比', '洛杉矶');
INSERT INTO tb_user VALUES ('3', '保罗', '洛杉矶');
# 创建一个名称为 tb_orders 的表
CREATE TABLE tb_orders (
  id int(32) PRIMARY KEY AUTO_INCREMENT,
  number varchar(32) NOT NULL,
  user_id int(32) NOT NULL,
  FOREIGN KEY(user_id) REFERENCES tb_user(id)
);
# 插入 3 条数据
INSERT INTO tb_orders VALUES ('1', '1000011', '1');
```

```
INSERT INTO tb_orders VALUES ('2', '1000012', '1');
INSERT INTO tb_orders VALUES ('3', '1000013', '2');
```

完成上述操作后，数据库 tb_user 表和 tb_orders 表中的数据如图 9-9 所示。

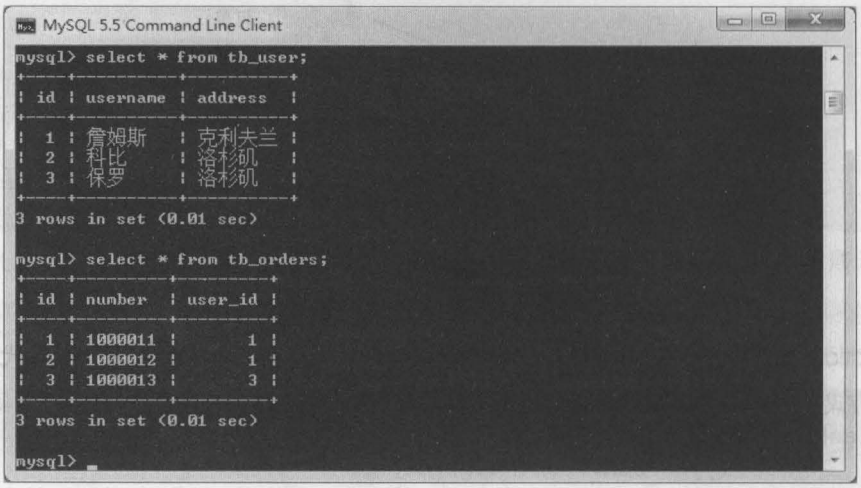


图9-9　tb_user和tb_orders表

（2）在 com.itheima.po 包中，创建持久化类 Orders 和 User，并在两个类中定义相关属性和方法，如文件 9-7 和文件 9-8 所示。

文件 9-7　Orders.java

```
1  package com.itheima.po;
2  /**
3   * 订单持久化类
4   */
5  public class Orders {
6      private Integer id;    //订单 id
7      private String number;//订单编号
8      public Integer getId() {
9          return id;
10     }
11     public void setId(Integer id) {
12         this.id = id;
13     }
14     public String getNumber() {
15         return number;
16     }
17     public void setNumber(String number) {
18         this.number = number;
19     }
20     @Override
21     public String toString() {
22         return "Orders [id=" + id + ", number=" + number + "]";
23     }
24 }
```

文件 9-8 User.java

```
package com.itheima.po;
import java.util.List;
/**
 * 用户持久化类
 */
public class User {
    private Integer id;                    // 用户编号
    private String username;               // 用户姓名
    private String address;                // 用户地址
    private List<Orders> ordersList; //用户关联的订单
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getAddress() {
        return address;
    }
    public void setAddress(String address) {
        this.address = address;
    }
    public List<Orders> getOrdersList() {
        return ordersList;
    }
    public void setOrdersList(List<Orders> ordersList) {
        this.ordersList = ordersList;
    }
    @Override
    public String toString() {
        return "User [id=" + id + ", username=" + username + ", address="
                + address + ", ordersList=" + ordersList + "]";
    }
}
```

在上述两个文件中，分别定义了各自的属性以及对应的 getter/setter 方法，同时为了方便查看输出结果，还重写了 toString()方法。

（3）在 com.itheima.mapper 包中，创建用户实体映射文件 UserMapper.xml，并在文件中编写一对多关联映射查询的配置，如文件 9-9 所示。

文件 9-9 UserMapper.xml

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

```
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<!-- namespace 表示命名空间 -->
<mapper namespace="com.itheima.mapper.UserMapper">
    <!-- 一对多：查看某一用户及其关联的订单信息
         注意：当关联查询出的列名相同，则需要使用别名区分 -->
    <select id="findUserWithOrders" parameterType="Integer"
                          resultMap="UserWithOrdersResult">
        SELECT u.*,o.id as orders_id,o.number
        from tb_user u,tb_orders o
        WHERE u.id=o.user_id
        and u.id=#{id}
    </select>
    <resultMap type="User" id="UserWithOrdersResult">
        <id property="id" column="id"/>
        <result property="username" column="username"/>
        <result property="address" column="address"/>
        <!-- 一对多关联映射：collection
            ofType 表示属性集合中元素的类型，List<Orders>属性即 Orders 类 -->
        <collection property="ordersList" ofType="Orders">
            <id property="id" column="orders_id"/>
            <result property="number" column="number"/>
        </collection>
    </resultMap>
</mapper>
```

在文件 9-9 中，使用了 MyBatis 嵌套结果的方式定义了一个根据用户 id 查询用户及其关联的订单信息的 select 语句。因为返回的用户对象中，包含 Orders 集合对象属性，所以需要手动编写结果映射信息。

（4）将映射文件 UserMapper.xml 的路径配置到核心配置文件 mybatis-config.xml 中，其代码如下所示。

```
<mapper resource="com/itheima/mapper/UserMapper.xml" />
```

（5）在测试类 MybatisAssociatedTest 中，编写测试方法 findUserTest()，其代码如下所示。

```
/**
 * 一对多
 */
@Test
public void findUserTest() {
    // 1. 通过工具类生成 SqlSession 对象
    SqlSession session = MybatisUtils.getSession();
    // 2. 查询 id 为 1 的用户信息
    User user = session.selectOne("com.itheima.mapper."
                        + "UserMapper.findUserWithOrders", 1);
    // 3. 输出查询结果信息
    System.out.println(user);
    // 4. 关闭 SqlSession
    session.close();
}
```

使用 JUnit4 执行 findUserTest()方法后，控制台输出结果如图 9-10 所示。

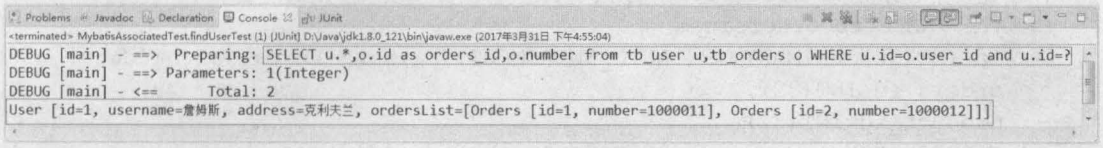


图9-10　运行结果

从图 9-10 可以看出，使用 MyBatis 嵌套结果的方式查询出了用户及其关联的订单集合信息。这就是 MyBatis 一对多的关联查询。

需要注意的是，上述案例从用户的角度出发，用户与订单之间是一对多的关联关系，但如果从单个订单的角度出发，一个订单只能属于一个用户，即一对一的关联关系。读者可根据 9.2 小节内容实现单个订单与用户之间的一对一关联关系，由于篇幅有限，这里不再重复赘述。

9.4　多对多

在实际项目开发中，多对多的关联关系也是非常常见的。以订单和商品为例，一个订单可以包含多种商品，而一种商品又可以属于多个订单，订单和商品就属于多对多的关联关系，如图 9-11 所示。

在数据库中，多对多的关联关系通常使用一个中间表来维护，中间表中的订单 id 作为外键参照订单表的 id，商品 id 作为外键参照商品表的 id。这三个表之间的关系如图 9-12 所示。

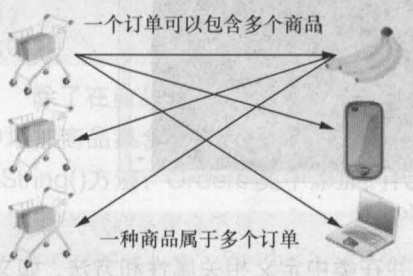


图9-11　订单和商品之间的关联关系

图9-12　数据库中订单表、中间表与商品表之间的关联

了解了数据库中订单表与商品表之间的多对多关联关系后，下面我们就通过具体的案例来讲解如何使用 MyBatis 来处理这种多对多的关系，具体实现步骤如下。

（1）创建数据表。在 mybatis 数据库中新建名称为 tb_product 和 tb_ordersitem 的两个数据表，同时在表中预先插入几条数据。其执行的 SQL 语句如下所示。

```
# 创建一个名称为 tb_product 的表
CREATE TABLE tb_product (
  id INT(32) PRIMARY KEY AUTO_INCREMENT,
  NAME VARCHAR(32),
  price DOUBLE
 );
# 插入 3 条数据
INSERT INTO tb_product VALUES ('1', 'Java 基础入门', '44.5');
INSERT INTO tb_product VALUES ('2', 'Java Web 程序开发入门', '38.5');
```

```
INSERT INTO tb_product VALUES ('3', 'SSM框架整合实战', '50');
# 创建一个名称为 tb_ordersitem 的中间表
CREATE TABLE tb_ordersitem  (
    id INT(32) PRIMARY KEY AUTO_INCREMENT,
    orders_id INT(32),
    product_id INT(32),
    FOREIGN KEY(orders_id) REFERENCES tb_orders(id),
    FOREIGN KEY(product_id) REFERENCES tb_product(id)
);
# 插入 3 条数据
INSERT INTO tb_ordersitem  VALUES ('1', '1', '1');
INSERT INTO tb_ordersitem  VALUES ('2', '1', '3');
INSERT INTO tb_ordersitem  VALUES ('3', '3', '3');
```

由于订单表在 9.3 小节中已经创建，所以这里只创建了商品表和中间表。完成上述操作后，tb_product 表和 tb_ordersitem 表中的数据如图 9-13 所示。

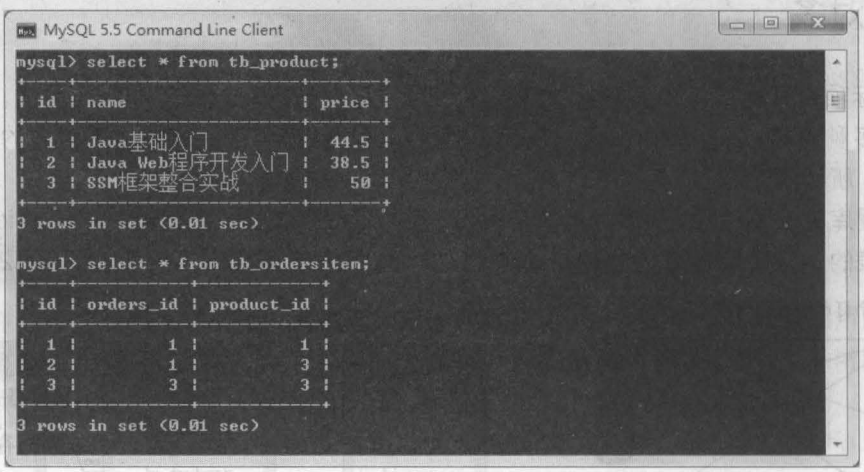


图9-13　tb_product和tb_ordersitem表

（2）在 com.itheima.po 包中，创建持久化类 Product，并在类中定义相关属性和方法，如文件 9-10 所示。

文件 9-10　Product.java

```
1  package com.itheima.po;
2  import java.util.List;
3  /**
4   * 商品持久化类
5   */
6  public class Product {
7      private Integer id;  //商品 id
8      private String name; //商品名称
9      private Double price;//商品单价
10     private List<Orders> orders; //与订单的关联属性
11     public Integer getId() {
12         return id;
13     }
```

```
    public void setId(Integer id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Double getPrice() {
        return price;
    }
    public void setPrice(Double price) {
        this.price = price;
    }
    public List<Orders> getOrders() {
        return orders;
    }
    public void setOrders(List<Orders> orders) {
        this.orders = orders;
    }
    @Override
    public String toString() {
        return "Product [id=" + id + ", name=" + name
                        + ", price=" + price + "]";
    }
}
```

除了在商品持久化类中需要添加订单的集合属性外，还需要在订单持久化类（Orders.java）中增加商品集合的属性及其对应的 getter/setter 方法，同时为了方便查看输出结果，需要重写 toString()方法，Orders 类中添加的代码如下所示。

```
//关联商品集合信息
private List<Product> productList;
//省略 getter/setter 方法，以及重写的 toString()方法
```

（3）在 com.itheima.mapper 包中，创建订单实体映射文件 OrdersMapper.xml 和商品实体映射文件 ProductMapper.xml，对两个映射文件进行编辑后，如文件 9-11 和文件 9-12 所示。

文件 9-11　OrdersMapper.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.itheima.mapper.OrdersMapper">
    <!-- 多对多嵌套查询：通过执行另外一条 SQL 映射语句来返回预期的特殊类型 -->
    <select id="findOrdersWithPorduct" parameterType="Integer"
            resultMap="OrdersWithProductResult">
        select * from tb_orders WHERE id=#{id}
    </select>
    <resultMap type="Orders" id="OrdersWithProductResult">
        <id property="id" column="id" />
```

```
        <result property="number" column="number" />
        <collection property="productList" column="id" ofType="Product"
            select="com.itheima.mapper.ProductMapper.findProductById">
        </collection>
    </resultMap>
</mapper>
```

在文件 9-11 中，使用嵌套查询的方式定义了一个 id 为 findOrdersWithPorduct 的 select 语句来查询订单及其关联的商品信息。在<resultMap>元素中使用了<collection>元素来映射多对多的关联关系，其中 property 属性表示订单持久化类中的商品属性，ofType 属性表示集合中的数据为 Product 类型，而 column 的属性值会作为参数执行 ProductMapper 中定义的 id 为 findProductById 的执行语句来查询订单中的商品信息。

文件 9-12　ProductMapper.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.itheima.mapper.ProductMapper">
    <select id="findProductById" parameterType="Integer"
                                    resultType="Product">
        SELECT * from tb_product where id IN(
          SELECT product_id FROM tb_ordersitem  WHERE orders_id = #{id}
        )
    </select>
</mapper>
```

在文件 9-12 中，定义了一个 id 为 findProductById 的执行语句，该执行语句中的 SQL 会根据订单 id 查询与该订单所关联的商品信息。由于订单和商品是多对多的关联关系，所以需要通过中间表来查询商品信息。

（4）将新创建的映射文件 OrdersMapper.xml 和 ProductMapper.xml 的文件路径配置到核心配置文件 mybatis-config.xml 中，代码如下所示。

```
<mapper resource="com/itheima/mapper/OrdersMapper.xml" />
<mapper resource="com/itheima/mapper/ProductMapper.xml" />
```

（5）在测试类 MybatisAssociatedTest 中，编写多对多关联查询的测试方法 findOrdersTest()，其代码如下所示。

```
/**
 * 多对多
 */
@Test
public void findOrdersTest(){
    // 1. 通过工具类生成 SqlSession 对象
    SqlSession session = MybatisUtils.getSession();
    // 2. 查询 id 为 1 的订单中的商品信息
    Orders orders = session.selectOne("com.itheima.mapper."
                    + "OrdersMapper.findOrdersWithPorduct", 1);
    // 3. 输出查询结果信息
    System.out.println(orders);
```

```
    // 4. 关闭 SqlSession
    session.close();
}
```

使用 JUnit4 执行 findOrdersTest()方法后，控制台的输出结果如图 9-14 所示。

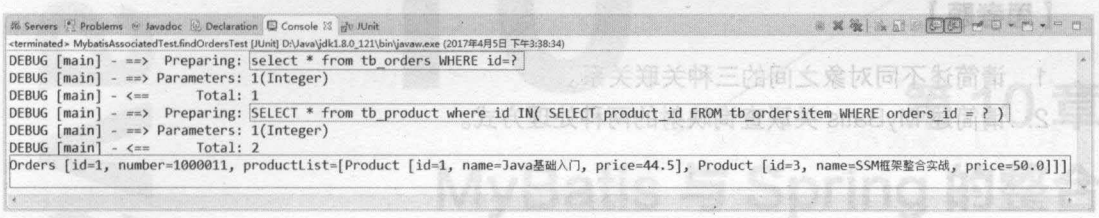


图9-14　运行结果

从图 9-14 可以看出，使用 MyBatis 嵌套查询的方式执行了两条 SQL 语句，并查询出了订单及其关联的商品信息，这就是 MyBatis 多对多的关联查询。

如果读者对多表关联查询的 SQL 语句比较熟，也可以在 OrdersMapper.xml 中使用嵌套结果的方式，其代码如下所示。

```
<!-- 多对多嵌套结果查询：查询某订单及其关联的商品详情 -->
<select id="findOrdersWithPorduct2" parameterType="Integer"
        resultMap="OrdersWithPorductResult2">
    select o.*,p.id as pid,p.name,p.price
    from tb_orders o,tb_product p,tb_ordersitem  oi
    WHERE oi.orders_id=o.id
    and oi.product_id=p.id
    and o.id=#{id}
</select>
<!-- 自定义手动映射类型 -->
<resultMap type="Orders" id="OrdersWithPorductResult2">
    <id property="id" column="id" />
    <result property="number" column="number" />
    <!-- 多对多关联映射：collection -->
    <collection property="productList" ofType="Product">
        <id property="id" column="pid" />
        <result property="name" column="name" />
        <result property="price" column="price" />
    </collection>
</resultMap>
```

在上述执行代码中，只定义了一条查询 SQL，通过该 SQL 即可查询出订单及其关联的商品信息。

9.5 本章小结

本章首先对开发中涉及的数据表之间以及对象之间的关联关系做了简要介绍，并由此引出了 MyBatis 框架中对关联关系的处理；然后通过案例对 MyBatis 框架处理实体对象之间的三种关联关系进行了详细讲解。通过本章的学习，读者可以了解数据表以及对象中所涉及的三种关联关系，

并能够使用 MyBatis 框架对三种关联关系的查询进行处理。MyBatis 中的关联查询操作在实际开发中非常普遍，熟练掌握这三种关联查询方式有助于提高项目的开发效率，因此读者一定要多加练习。

【思考题】

1. 请简述不同对象之间的三种关联关系。
2. 请简述 MyBatis 关联查询映射的两种处理方式。

Java EE

10 Chapter

第 10 章
MyBatis 与 Spring 的整合

学习目标

- 掌握传统 DAO 方式的开发整合
- 掌握 Mapper 接口方式的开发整合

在前面章节中，分别讲解了 Spring 和 MyBatis 的相关知识，然而在实际的项目开发中，Spring 与 MyBatis 都是整合在一起使用的。在掌握了 MyBatis 的使用后，本章将对 MyBatis 与 Spring 的整合内容进行详细讲解。

10.1 整合环境搭建

10.1.1 准备所需 JAR 包

要实现 MyBatis 与 Spring 的整合，很明显需要这两个框架的 JAR 包，但是只使用这两个框架中所提供的 JAR 包是不够的，还需要其他的 JAR 包来配合使用，整合时所需准备的 JAR 包具体如下。

1. Spring 框架所需的 JAR 包

Spring 框架所需要准备的 JAR 包共 10 个，其中包括：4 个核心模块 JAR，AOP 开发使用的 JAR，JDBC 和事务的 JAR（其中核心容器依赖的 commons-logging 的 JAR 在 MyBatis 框架的 lib 包中已经包含，所以这里不必再加入），具体如下所示。

- aopalliance-1.0.jar
- aspectjweaver-1.8.10.jar
- spring-aop-4.3.6.RELEASE.jar
- spring-aspects-4.3.6.RELEASE.jar
- spring-beans-4.3.6.RELEASE.jar
- spring-context-4.3.6.RELEASE.jar
- spring-core-4.3.6.RELEASE.jar
- spring-expression-4.3.6.RELEASE.jar
- spring-jdbc-4.3.6.RELEASE.jar
- spring-tx-4.3.6.RELEASE.jar

2. MyBatis 框架所需的 JAR 包

MyBatis 框架所需要准备的 JAR 包共 13 个，其中包括：核心包 mybatis-3.4.2.jar 以及其解压文件夹中 lib 目录中的所有 JAR，具体如下所示。

- ant-1.9.6.jar
- ant-launcher-1.9.6.jar
- asm-5.1.jar
- cglib-3.2.4.jar
- commons-logging-1.2.jar
- javassist-3.21.0-GA.jar
- log4j-1.2.17.jar
- log4j-api-2.3.jar
- log4j-core-2.3.jar
- mybatis-3.4.2.jar
- ognl-3.1.12.jar

- slf4j-api-1.7.22.jar
- slf4j-log4j12-1.7.22.jar

3. MyBatis 与 Spring 整合的中间 JAR

由于 MyBatis 3 在发布之前，Spring 3 就已经开发完成，而 Spring 团队既不想发布基于 MyBatis 3 的非发布版本的代码，也不想长时间的等待，所以 Spring 3 以后，就没有对 MyBatis 3 进行支持。为了满足 MyBatis 用户对 Spring 框架的需求，MyBatis 社区自己开发了一个用于整合这两个框架的中间件——MyBatis-Spring。

本书编写时，该中间件的最新版本为 mybatis-spring-1.3.1.jar，本书所使用的就是该版本，希望读者也同样下载该版本。此版本的 JAR 包可以通过如下链接获取“http://mvnrepository.com/artifact/org.mybatis/mybatis-spring/1.3.1”。

4. 数据库驱动 JAR 包

本书所使用的数据库驱动包为 mysql-connector-java-5.1.40-bin.jar。

5. 数据源所需 JAR 包

整合时所使用的是 DBCP 数据源，所以需要准备 DBCP 和连接池的 JAR 包，具体如下所示。

- commons-dbcp2-2.1.1.jar
- commons-pool2-2.4.2.jar

10.1.2　编写配置文件

在 Eclipse 中，创建一个名称为 chapter10 的 Web 项目，将上一小节中所准备的全部 JAR 包添加到项目的 lib 目录中，并发布到类路径下。

在项目的 src 目录下，分别创建 db.properties 文件、Spring 的配置文件以及 MyBatis 的配置文件，如文件 10-1、文件 10-2 和文件 10-3 所示。

文件 10-1　db.properties

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/mybatis
jdbc.username=root
jdbc.password=root
jdbc.maxTotal=30
jdbc.maxIdle=10
jdbc.initialSize=5
```

在文件 10-1 中，除配置了连接数据库的基本 4 项外，还配置了数据库连接池的最大连接数（maxTotal）、最大空闲连接数（maxIdle）以及初始化连接数（initialSize）。

文件 10-2　applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
```

```
9       http://www.springframework.org/schema/tx
10      http://www.springframework.org/schema/tx/spring-tx-4.3.xsd
11      http://www.springframework.org/schema/context
12      http://www.springframework.org/schema/context/spring-context-4.3.xsd
13      http://www.springframework.org/schema/aop
14      http://www.springframework.org/schema/aop/spring-aop-4.3.xsd">
15      <!--读取 db.properties -->
16      <context:property-placeholder location="classpath:db.properties"/>
17      <!-- 配置数据源 -->
18      <bean id="dataSource"
19            class="org.apache.commons.dbcp2.BasicDataSource">
20          <!--数据库驱动 -->
21          <property name="driverClassName" value="${jdbc.driver}" />
22          <!--连接数据库的 url -->
23          <property name="url" value="${jdbc.url}" />
24          <!--连接数据库的用户名 -->
25          <property name="username" value="${jdbc.username}" />
26          <!--连接数据库的密码 -->
27          <property name="password" value="${jdbc.password}" />
28          <!--最大连接数 -->
29          <property name="maxTotal" value="${jdbc.maxTotal}" />
30          <!--最大空闲连接 -->
31          <property name="maxIdle" value="${jdbc.maxIdle}" />
32          <!--初始化连接数 -->
33          <property name="initialSize" value="${jdbc.initialSize}" />
34      </bean>
35      <!-- 事务管理器，依赖于数据源 -->
36      <bean id="transactionManager" class=
37       "org.springframework.jdbc.datasource.DataSourceTransactionManager">
38          <property name="dataSource" ref="dataSource" />
39      </bean>
40      <!--开启事务注解 -->
41      <tx:annotation-driven transaction-manager="transactionManager"/>
42      <!--配置 MyBatis 工厂 -->
43      <bean id="sqlSessionFactory"
44            class="org.mybatis.spring.SqlSessionFactoryBean">
45          <!--注入数据源 -->
46          <property name="dataSource" ref="dataSource" />
47          <!--指定核心配置文件位置 -->
48      <property name="configLocation" value="classpath:mybatis-config.xml"/>
49      </bean>
50  </beans>
```

在文件 10-2 中，首先定义了读取 properties 文件的配置，然后配置了数据源，接下来配置了事务管理器并开启了事务注解，最后配置了 MyBatis 工厂来与 Spring 整合。其中，MyBatis 工厂的作用就是构建 SqlSessionFactory，它是通过 mybatis-spring 包中提供的 org.mybatis.spring.SqlSessionFactoryBean 类来配置的。通常，在配置时需要提供两个参数：一个是数据源，另一个是 MyBatis 的配置文件路径。这样 Spring 的 IoC 容器就会在初始化 id 为 sqlSessionFactory 的 Bean 时解析 MyBatis 的配置文件，并与数据源一同保存到 Spring 的 Bean 中。

文件 10-3 mybatis-config.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
   "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
   <!--配置别名 -->
   <typeAliases>
      <package name="com.itheima.po" />
   </typeAliases>
   <!--配置 Mapper 的位置 -->
    <mappers>
      ...
    </mappers>
</configuration>
```

由于在 Spring 中已经配置了数据源信息，所以在 MyBatis 的配置文件中就不再需要配置数据源信息。这里只需要使用<typeAliases>和<mappers>元素来配置文件别名以及指定 Mapper 文件位置即可。

此外，还需在项目的 src 目录下创建 log4j.properties 文件，该文件的编写可参考第 6 章的入门案例，也可将前面章节所创建的该文件复制到此项目中使用。

10.2 传统 DAO 方式的开发整合

上一小节已经完成了对 MyBatis 与 Spring 整合环境的搭建工作，可以说完成了这些配置后，就已经完成了这两个框架大部分的整合工作。接下来，本小节将通过传统 DAO 层开发的方式，来演示这两个框架实际的整合使用。

采用传统 DAO 开发方式进行 MyBatis 与 Spring 框架的整合时，我们需要编写 DAO 接口以及接口的实现类，并且需要向 DAO 实现类中注入 SqlSessionFactory，然后在方法体内通过 SqlSessionFactory 创建 SqlSession。为此，我们可以使用 mybatis-spring 包中所提供的 SqlSessionTemplate 类或 SqlSessionDaoSupport 类来实现此功能。这两个类的描述如下。

- SqlSessionTemplate：是 mybatis-spring 的核心类，它负责管理 MyBatis 的 SqlSession，调用 MyBatis 的 SQL 方法。当调用 SQL 方法时，SqlSessionTemplate 将会保证使用的 SqlSession 和当前 Spring 的事务是相关的。它还管理 SqlSession 的生命周期，包含必要的关闭、提交和回滚操作。
- SqlSessionDaoSupport：是一个抽象支持类，它继承了 DaoSupport 类，主要是作为 DAO 的基类来使用。可以通过 SqlSessionDaoSupport 类的 getSqlSession()方法来获取所需的 SqlSession。

了解了传统 DAO 开发方式整合可以使用的两个类后，下面以 SqlSessionDaoSupport 类的使用为例，讲解传统的 DAO 开发方式整合的实现，其具体步骤如下。

1. 实现持久层

（1）在 src 目录下，创建一个 com.itheima.po 包，并在包中创建持久化类 Customer，在

Customer 类中定义相关属性和方法后，如文件 10-4 所示。

文件 10-4　Customer.java

```
1  package com.itheima.po;
2  /**
3   * 客户持久化类
4   */
5  public class Customer {
6      private Integer id;          // 主键 id
7      private String username;     // 客户名称
8      private String jobs;         // 职业
9      private String phone;        // 电话
10     public Integer getId() {
11         return id;
12     }
13     public void setId(Integer id) {
14         this.id = id;
15     }
16     public String getUsername() {
17         return username;
18     }
19     public void setUsername(String username) {
20         this.username = username;
21     }
22     public String getJobs() {
23         return jobs;
24     }
25     public void setJobs(String jobs) {
26         this.jobs = jobs;
27     }
28     public String getPhone() {
29         return phone;
30     }
31     public void setPhone(String phone) {
32         this.phone = phone;
33     }
34     @Override
35     public String toString() {
36         return "Customer [id=" + id + ", username=" + username +
37                         ", jobs=" + jobs + ", phone=" + phone + "]";
38     }
39 }
```

（2）在 com.itheima.po 包中，创建映射文件 CustomerMapper.xml，在该文件中编写根据 id 查询客户信息的映射语句，如文件 10-5 所示。

文件 10-5　CustomerMapper.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
```

```
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.itheima.po.CustomerMapper">
    <!--根据 id 查询客户信息 -->
    <select id="findCustomerById" parameterType="Integer"
            resultType="customer">
        select * from t_customer where id = #{id}
    </select>
</mapper>
```

（3）在 MyBatis 的配置文件 mybatis-config.xml 中，配置映射文件 CustomerMapper.xml 的位置，具体如下。

```
<mapper resource="com/itheima/po/CustomerMapper.xml" />
```

2. 实现 DAO 层

（1）在 src 目录下，创建一个 com.itheima.dao 包，并在包中创建接口 CustomerDao，在接口中编写一个通过 id 查询客户的方法 findCustomerById()，如文件 10-6 所示。

文件 10-6　CustomerDao.java

```
package com.itheima.dao;
import com.itheima.po.Customer;
public interface CustomerDao {
    // 通过 id 查询客户
    public Customer findCustomerById(Integer id);
}
```

（2）在 src 目录下，创建一个 com.itheima.dao.impl 包，并在包中创建 CustomerDao 接口的实现类 CustomerDaoImpl，编辑后如文件 10-7 所示。

文件 10-7　CustomerDaoImpl.java

```
package com.itheima.dao.impl;
import org.mybatis.spring.support.SqlSessionDaoSupport;
import com.itheima.dao.CustomerDao;
import com.itheima.po.Customer;
public class CustomerDaoImpl
                    extends SqlSessionDaoSupport implements CustomerDao {
    // 通过 id 查询客户
    public Customer findCustomerById(Integer id) {
        return this.getSqlSession().selectOne("com.itheima.po"
                    + ".CustomerMapper.findCustomerById", id);
    }
}
```

在文件 10-7 中，CustomerDaoImpl 类继承了 SqlSessionDaoSupport 类，并实现了 CustomerDao 接口。其中，SqlSessionDaoSupport 类在使用时需要一个 SqlSessionFactory 或一个 SqlSessionTemplate 对象，所以需要通过 Spring 给 SqlSessionDaoSupport 类的子类对象注入一个 SqlSessionFactory 或 SqlSessionTemplate。这样，在子类中就能通过调用 SqlSessionDaoSupport 类的 getSqlSession()方法来获取 SqlSession 对象，并使用 SqlSession 对象中的方法了。

（3）在 Spring 的配置文件 applicationContext.xml 中，编写实例化 CustomerDaoImpl 的配

置，代码如下所示。

```
<!--实例化 Dao -->
<bean id="customerDao" class="com.itheima.dao.impl.CustomerDaoImpl">
    <!-- 注入 SqlSessionFactory 对象实例-->
    <property name="sqlSessionFactory" ref="sqlSessionFactory" />
</bean>
```

上述代码创建了一个 id 为 customerDao 的 Bean，并将 SqlSessionFactory 对象注入到了该 Bean 的实例化对象中。

3. 整合测试

在 src 目录下，创建一个 com.itheima.test 包，在包中创建测试类 DaoTest，并在类中编写测试方法 findCustomerByIdDaoTest()，如文件 10-8 所示。

文件 10-8　DaoTest.java

```
1  package com.itheima.test;
2  import org.junit.Test;
3  import org.springframework.context.ApplicationContext;
4  import
5      org.springframework.context.support.ClassPathXmlApplicationContext;
6  import com.itheima.dao.CustomerDao;
7  import com.itheima.po.Customer;
8  /**
9   * DAO 测试类
10  */
11 public class DaoTest {
12     @Test
13     public void findCustomerByIdDaoTest(){
14         ApplicationContext act =
15             new ClassPathXmlApplicationContext("applicationContext.xml");
16         // 根据容器中 Bean 的 id 来获取指定的 Bean
17         CustomerDao customerDao =
18                             (CustomerDao) act.getBean("customerDao");
19         Customer customer = customerDao.findCustomerById(1);
20         System.out.println(customer);
21     }
22 }
```

在上述方法中，我们采用的是根据容器中 Bean 的 id 来获取指定 Bean 的方式。有些读者在通过其他一些参考资料学习时，可能还会看到另一种获取 Bean 的方式，即根据类的类型来获取 Bean 的实例，此种方式就是第 1 章中讲解的 Spring 另一种获取 Bean 的方式。如果要采用这种方式，只需将第 17~18 行代码替换成如下。

```
CustomerDao customerDao = act.getBean(CustomerDao.class);
```

这样在获取 Bean 的实例时，就不再需要进行强制类型转换了。

使用 JUnit4 执行上述方法后，控制台的输出结果如图 10-1 所示。

从图 10-1 可以看出，通过 CustomerDao 实例的 findCustomerById()方法已经查询出了 id 为 1 的客户信息，这也就说明了 MyBatis 与 Spring 整合成功。

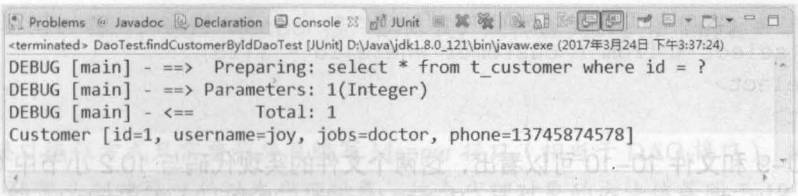


图10-1 运行结果

10.3 Mapper 接口方式的开发整合

在 MyBatis+Spring 的项目中，虽然使用传统的 DAO 开发方式可以实现所需功能，但是采用这种方式在实现类中会出现大量的重复代码，在方法中也需要指定映射文件中执行语句的 id，并且不能保证编写时 id 的正确性（运行时才能知道）。为此，我们可以使用 MyBatis 提供的另外一种编程方式，即使用 Mapper 接口编程。接下来的两个小节中，将讲解如何使用 Mapper 接口方式来实现 MyBatis 与 Spring 的整合。

10.3.1 基于 MapperFactoryBean 的整合

MapperFactoryBean 是 MyBatis-Spring 团队提供的一个用于根据 Mapper 接口生成 Mapper 对象的类，该类在 Spring 配置文件中使用时可以配置以下参数。

- mapperInterface：用于指定接口。
- SqlSessionFactory：用于指定 SqlSessionFactory。
- SqlSessionTemplate：用于指定 SqlSessionTemplate。如果与 SqlSessionFactory 同时设定，则只会启用 SqlSessionTemplate。

了解了 MapperFactoryBean 类后，接下来通过一个具体的案例来演示如何通过 MapperFactoryBean 来实现 MyBatis 与 Spring 的整合，具体步骤如下。

（1）在 src 目录下，创建一个 com.itheima.mapper 包，然后在该包中创建 CustomerMapper 接口以及对应的映射文件，编辑后如文件 10-9 和文件 10-10 所示。

文件 10-9 CustomerMapper.java

```
1  package com.itheima.mapper;
2  import com.itheima.po.Customer;
3  public interface CustomerMapper {
4      // 通过 id 查询客户
5      public Customer findCustomerById(Integer id);
6  }
```

文件 10-10 CustomerMapper.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
3      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
4  <mapper namespace="com.itheima.mapper.CustomerMapper">
5      <!--根据 id 查询客户信息 -->
6      <select id="findCustomerById" parameterType="Integer"
```

```
7              resultType="customer">
8          select * from t_customer where id = #{id}
9      </select>
10 </mapper>
```

从文件 10-9 和文件 10-10 可以看出，这两个文件的实现代码与 10.2 小节中 CustomerDao 接口以及映射文件 CustomerMapper.xml 的实现代码基本相同，只是本案例与 10.2 小节案例的区别在于，本案例将接口文件改名并与映射文件一起放在了 com.itheima.mapper 包中。

（2）在 MyBatis 的配置文件中，引入新的映射文件，代码如下所示。

```
<!-- Mapper 接口开发方式 -->
<mapper resource="com/itheima/mapper/CustomerMapper.xml" />
```

小提示

使用 Mapper 接口动态代理开发方式时，如果完全遵循了编写规范（请参见本节后面的脚下留心），那么在配置文件中可以不引入映射文件。

（3）在 Spring 的配置文件中，创建一个 id 为 customerMapper 的 Bean，代码如下所示。

```
<!-- Mapper 代理开发（基于 MapperFactoryBean） -->
<bean id="customerMapper"
      class="org.mybatis.spring.mapper.MapperFactoryBean">
   <property name="mapperInterface"
             value="com.itheima.mapper.CustomerMapper" />
   <property name="sqlSessionFactory" ref="sqlSessionFactory" />
</bean>
```

上述配置代码为 MapperFactoryBean 指定了接口以及 SqlSessionFactory。

（4）在测试类 DaoTest 中，编写测试方法 findCustomerByIdMapperTest()，其代码如下所示。

```
@Test
public void findCustomerByIdMapperTest(){
   ApplicationContext act =
         new ClassPathXmlApplicationContext("applicationContext.xml");
   CustomerMapper customerMapper = act.getBean(CustomerMapper.class);
   Customer customer = customerMapper.findCustomerById(1);
   System.out.println(customer);
}
```

上述方法中，通过 Spring 容器获取了 CustomerMapper 实例，并调用了实例中的 findCustomerById()方法来查询 id 为 1 的客户信息。

使用 JUnit4 执行上述方法后，控制台的输出结果如图 10-2 所示。

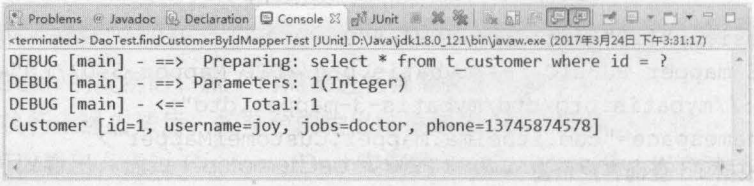


图10-2　运行结果

从图 10-2 可以看出，通过 Mapper 接口的方式，同样查询到了相应的客户信息。


脚下留心：

Mapper 接口编程方式只需要程序员编写 Mapper 接口（相当于 DAO 接口），然后由 MyBatis 框架根据接口的定义创建接口的动态代理对象，这个代理对象的方法体等同于 10.2 小节中 DAO 接口的实现类方法。

虽然使用 Mapper 接口编程的方式很简单，但是在具体使用时还是需要遵循以下规范。

（1）Mapper 接口的名称和对应的 Mapper.xml 映射文件的名称必须一致。

（2）Mapper.xml 文件中的 namespace 与 Mapper 接口的类路径相同（即接口文件和映射文件需要放在同一个包中）。

（3）Mapper 接口中的方法名和 Mapper.xml 中定义的每个执行语句的 id 相同。

（4）Mapper 接口中方法的输入参数类型要和 Mapper.xml 中定义的每个 sql 的 parameterType 的类型相同。

（5）Mapper 接口方法的输出参数类型要和 Mapper.xml 中定义的每个 sql 的 resultType 的类型相同。

只要遵循了这些开发规范，MyBatis 就可以自动生成 Mapper 接口实现类的代理对象，从而简化我们的开发。

10.3.2　基于 MapperScannerConfigurer 的整合

在实际的项目中，DAO 层会包含很多接口，如果每一个接口都像 10.2 小节中那样在 Spring 配置文件中配置，那么不但会增加工作量，还会使得 Spring 配置文件非常臃肿。为此，MyBatis-Spring 团队提供了一种自动扫描的形式来配置 MyBatis 中的映射器——采用 MapperScannerConfigurer 类。

MapperScannerConfigurer 类在 Spring 配置文件中使用时可以配置以下几个属性。

- basePackage：指定映射接口文件所在的包路径，当需要扫描多个包时可以使用分号或逗号作为分隔符。指定包路径后，会扫描该包及其子包中的所有文件。
- annotationClass：指定了要扫描的注解名称，只有被注解标识的类才会被配置为映射器。
- sqlSessionFactoryBeanName：指定在 Spring 中定义的 SqlSessionFactory 的 Bean 名称。
- sqlSessionTemplateBeanName：指定在 Spring 中定义的 SqlSessionTemplate 的 Bean 名称。如果定义此属性，则 sqlSessionFactoryBeanName 将不起作用。
- markerInterface：指定创建映射器的接口。

MapperScannerConfigurer 的使用非常简单，只需要在 Spring 的配置文件中编写如下代码：

```
<!-- Mapper 代理开发（基于 MapperScannerConfigurer） -->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="basePackage" value="com.itheima.mapper" />
</bean>
```

在通常情况下，MapperScannerConfigurer 在使用时只需通过 basePackage 属性指定需要扫描的包即可，Spring 会自动地通过包中的接口来生成映射器。这使得开发人员可以在编写很少代码的情况下，完成对映射器的配置，从而提高开发效率。

要验证上面的配置也很容易，只需将上述配置代码写入 Spring 的配置文件，并将 10.2 小节中的第（2）步和第（3）步的代码注释掉，再次执行 findCustomerByIdMapperTest()方法进行测试即可。方法执行后，会发现显示结果与图 10-2 中的结果一致。由于篇幅有限，这里将不再展示，读者可自行测试。

10.4 测试事务

在 MyBatis+Spring 的项目中，事务是由 Spring 来管理的。在 10.1.2 小节中，我们已经配置了事务管理器，并开启了事务注解，但是现在还不能够确定事务的配置是否正确，以及事务管理能否生效。接下来，本节将对如何测试项目中配置的事务进行详细讲解。

在项目中，业务层（Service 层）既是处理业务的地方，又是管理数据库事务的地方。要对事务进行测试，首先需要创建业务层，并在业务层编写添加客户操作的代码；然后在添加操作的代码后，有意地添加一段异常代码（如 int i = 1/0;）来模拟现实中的意外情况；最后编写测试方法，调用业务层的添加方法。这样，程序在执行到错误代码时就会出现异常。在没有事务管理的情况下，即使出现了异常，数据也会被存储到数据表中；如果添加了事务管理，并且事务管理的配置正确，那么在执行上述操作时，所添加的数据将不能够插入到数据表中。

下面对上述分析进行实际的编写测试，其具体的实现步骤如下。

（1）在 CustomerMapper 接口中，编写测试方法 addCustomer()，代码如下所示。

```
// 添加客户
public void addCustomer(Customer customer);
```

编写完接口中的方法后，在映射文件 CustomerMapper.xml 中编写执行插入操作的 SQL 配置，代码如下所示。

```
<!--添加客户信息 -->
<insert id="addCustomer" parameterType="customer">
    insert into t_customer(username,jobs,phone)
    values(#{username},#{jobs},#{phone})
</insert>
```

（2）在 src 目录下，创建一个 com.itheima.service 包，并在包中创建接口 CustomerService，在接口中编写一个添加客户的方法 addCustomer()，如文件 10-11 所示。

文件 10-11　CustomerService.java

```
1  package com.itheima.service;
2  import com.itheima.po.Customer;
3  public interface CustomerService {
4      public void addCustomer(Customer customer);
5  }
```

（3）在 src 目录下，创建一个 com.itheima.service.impl 包，并在包中创建 CustomerService 接口的实现类 CustomerServiceImpl，来实现接口中的方法，编辑后如文件 10-12 所示。

文件 10-12　CustomerServiceImpl.java

```
1  package com.itheima.service.impl;
2  import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import com.itheima.mapper.CustomerMapper;
import com.itheima.po.Customer;
import com.itheima.service.CustomerService;
@Service
//@Transactional
public class CustomerServiceImpl implements CustomerService {
    //注解注入 CustomerMapper
    @Autowired
    private CustomerMapper customerMapper;
    //添加客户
    public void addCustomer(Customer customer) {
        this.customerMapper.addCustomer(customer);
        int i=1/0; //模拟添加操作后系统突然出现的异常问题
    }
}
```

在文件 10-12 中，使用了 Spring 的注解@Service 来标识业务层的类，使用了@Transactional 注解来标识事务处理的类，并通过@Autowired 注解将 CustomerMapper 接口注入到本类中。

小提示

这里先将@Transactional 注解进行了注释，是为了先执行此类没有事务管理的情况。之后再删除注释，执行包含事务管理的情况，即可通过结果来验证事务是否配置成功。

（4）在 Spring 的配置文件中，编写开启注解扫描的配置代码，代码如下所示。

```
<!-- 开启扫描 -->
<context:component-scan base-package="com.itheima.service" />
```

（5）在 com.itheima.test 包中，创建测试类 TransactionTest，在测试类的 main()方法中编写测试事务执行的代码，如文件 10-13 所示。

文件 10-13 TransactionTest.java

```
package com.itheima.test;
import org.springframework.context.ApplicationContext;
import
    org.springframework.context.support.ClassPathXmlApplicationContext;
import com.itheima.po.Customer;
import com.itheima.service.CustomerService;
/**
 * 测试事务
 */
public class TransactionTest {
    public static void main(String[] args) {
        ApplicationContext act =
            new ClassPathXmlApplicationContext("applicationContext.xml");
        CustomerService customerService =
                                    act.getBean(CustomerService.class);
```

```
        Customer customer = new Customer();
        customer.setUsername("zhangsan");
        customer.setJobs("manager");
        customer.setPhone("13233334444");
        customerService.addCustomer(customer);
    }
}
```

在文件 10-13 中，首先获取了 CustomerService 的实例，然后创建了 Customer 对象，并向对象中添加属性值，最后调用了实例的 addCustomer()方法执行添加客户操作。

在运行测试方法之前，先来查看一下数据库中的已有数据，如图 10-3 所示。

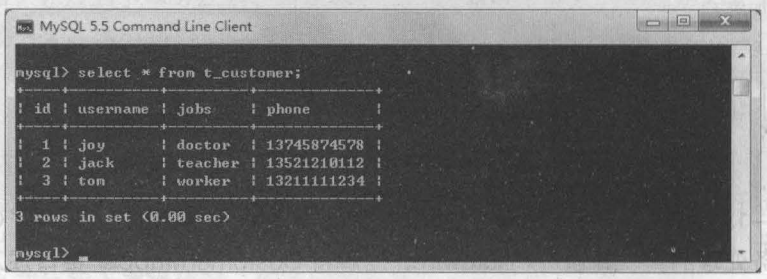


图10-3　t_customer表

从图 10-3 可以看到，此时数据表中只有 3 条数据。执行测试类中的 main()方法后，控制台的输出结果如图 10-4 所示。

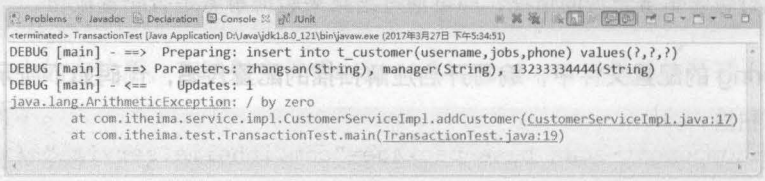


图10-4　运行结果

从图 10-4 可以看到，程序已经执行了插入操作，并且在执行到错误代码时抛出了异常信息。再次查询 t_customer 表，其表中的数据如图 10-5 所示。

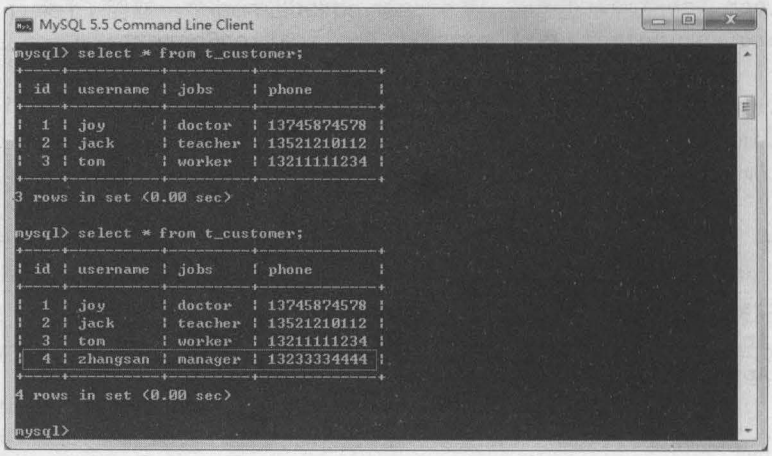


图10-5　t_customer表

从图 10-5 可以看到，新添加的数据已经存储在了 t_customer 表中，这说明项目中的事务管理没有起作用。此时将文件 10-12 中@Transactional 前面的注释删除，再次执行测试类中的 main()方法后，虽然 Eclipse 的控制台也会显示抛出的异常信息，但是此时 t_customer 表中依然只有 4 条数据，如图 10-6 所示。

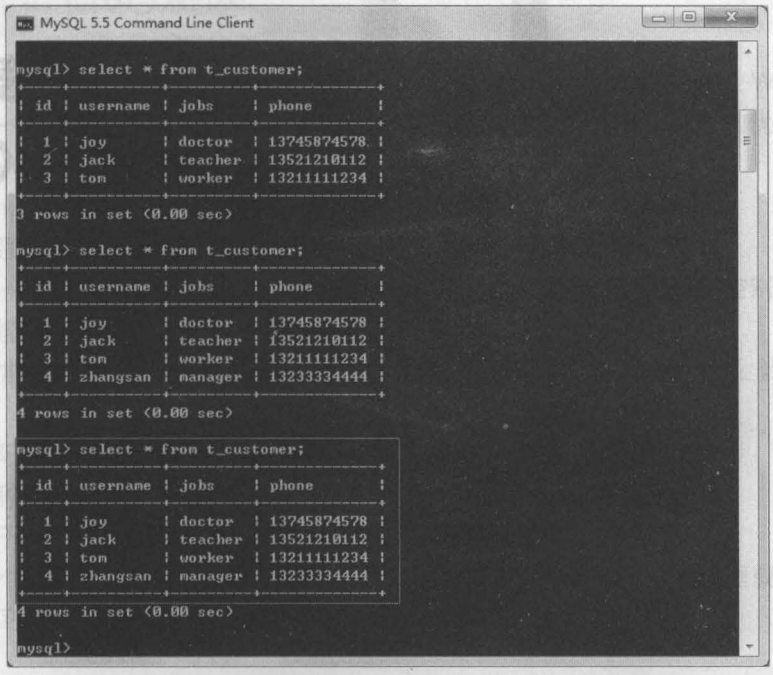


图10-6　t_customer表

这也就说明项目中的事务配置是正确的，至此，对于整合时事务功能的测试就已经完成。

10.5 本章小结

本章首先对 MyBatis 与 Spring 框架整合的环境搭建进行了讲解，然后讲解了使用传统 DAO 方式的开发整合，以及基于 Mapper 接口方式的开发整合。通过本章的学习，读者能够熟练地掌握 MyBatis 与 Spring 框架的几种整合方式，这将为后面项目的学习打下坚实的基础。

【思考题】

1. 请简述 MyBatis 与 Spring 整合所需 JAR 包的种类。
2. 请简述 MapperFactoryBean 和 MapperScannerConfigurer 的作用。

Java EE

11 Chapter

第11章 Spring MVC入门

学习目标

- 了解 Spring MVC 的特点
- 掌握 Spring MVC 入门程序的编写
- 熟悉 Spring MVC 的工作流程

通过前面章节的学习，读者已经可以掌握SSM框架中Spring框架和MyBatis框架的使用，并学会了如何将两个框架进行整合。从本章开始，将讲解SSM中的最后一个框架——Spring MVC。

11.1 Spring MVC概述

Spring MVC是Spring提供的一个实现了Web MVC设计模式的轻量级Web框架。它与Struts 2框架一样，都属于MVC框架，但其使用和性能等方面比Struts 2更加优异。

Spring MVC具有如下特点。

- 是Spring框架的一部分，可以方便地利用Spring所提供的其他功能。
- 灵活性强，易于与其他框架集成。
- 提供了一个前端控制器DispatcherServlet，使开发人员无须额外开发控制器对象。
- 可自动绑定用户输入，并能正确的转换数据类型。
- 内置了常见的校验器，可以校验用户输入。如果校验不能通过，那么就会重定向到输入表单。
- 支持国际化。可以根据用户区域显示多国语言。
- 支持多种视图技术。它支持JSP、Velocity和FreeMarker等视图技术。
- 使用基于XML的配置文件，在编辑后，不需要重新编译应用程序。

除上述几个优点外，Spring MVC还有很多其他优点，由于篇幅有限，这里就不一一列举了。在接下来的学习中，读者会逐渐地体会到Spring MVC的这些优点。

11.2 第一个Spring MVC应用

了解了什么是Spring MVC，以及它的一些优点后，接下来本小节将通过一个简单的入门案例，来演示Spring MVC的使用，具体实现步骤如下。

1. 创建项目，引入JAR包

在Eclipse中，创建一个名称为chapter11的Web项目，在项目的lib目录中添加运行Spring MVC程序所需要的JAR包，并发布到类路径下，添加后的项目结构如图11-1所示。

从图11-1可以看到，项目中添加了Spring的4个核心JAR包、commons-logging的JAR以及两个Web相关的JAR(可以在Spring解压文件夹的libs目录中找到)，这两个Web相关的JAR包就是Spring MVC框架所需的JAR包。由于本书中所使用的Spring版本是4.3.6，所以Spring MVC也是基于该版本的。

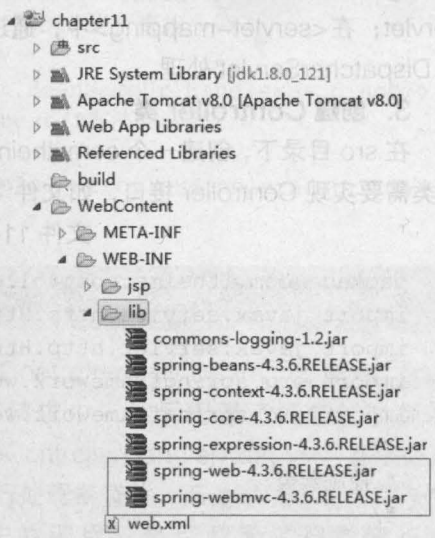


图11-1 添加JAR包后的项目结构

2. 配置前端控制器

在web.xml中，配置Spring MVC的前端控制器DispatcherServlet，如文件11-1所示。

文件 11-1　web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    id="WebApp_ID" version="3.0">
    <servlet>
        <!-- 配置前端过滤器 -->
        <servlet-name>springmvc</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <!-- 初始化时加载配置文件 -->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:springmvc-config.xml</param-value>
        </init-param>
        <!-- 表示容器在启动时立即加载 Servlet -->
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>springmvc</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

在文件 11-1 中，主要对<servlet>和<servlet-mapping>元素进行了配置。在<servlet>中，配置了 Spring MVC 的前端控制器 DispatcherServlet，并通过其子元素<init-param>配置了 Spring MVC 配置文件的位置，<load-on-startup>元素中的 1 表示容器在启动时立即加载这个 Servlet；在<servlet-mapping>中，通过<url-pattern>元素的“/”，会将所有 URL 拦截，并交由 DispatcherServlet 处理。

3. 创建 Controller 类

在 src 目录下，创建一个 com.itheima.controller 包，并在包中创建控制器类 FirstController，该类需要实现 Controller 接口，如文件 11-2 所示。

文件 11-2　FirstController.java

```
package com.itheima.controller;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;
/**
 * 控制器类
 */
public class FirstController implements Controller{
    @Override
```

```
    public ModelAndView handleRequest(HttpServletRequest request,
                        HttpServletResponse response) {
        // 创建ModelAndView对象
        ModelAndView mav = new ModelAndView();
        // 向模型对象中添加数据
        mav.addObject("msg", "这是我的第一个Spring MVC程序");
        // 设置逻辑视图名
        mav.setViewName("/WEB-INF/jsp/first.jsp");
        // 返回ModelAndView对象
        return mav;
    }
}
```

在文件 11-2 中，handleRequest()是 Controller 接口的实现方法，FirstController 类会调用该方法来处理请求，并返回一个包含视图名或包含视图名和模型的 ModelAndView 对象。本案例中，向模型对象中添加了一个名称为 msg 的字符串对象，并设置返回的视图路径为“/WEB-INF/jsp/first.jsp”，这样，请求就会被转发到 first.jsp 页面。

4. 创建 Spring MVC 的配置文件，配置控制器映射信息

在 src 目录下，创建配置文件 springmvc-config.xml，并在文件中配置控制器信息，如文件 11-3 所示。

文件 11-3 springmvc-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">
    <!-- 配置处理器Handle，映射“/firstController”请求 -->
    <bean name="/firstController"
            class="com.itheima.controller.FirstController" />
    <!-- 处理器映射器，将处理器Handle的name作为url进行查找 -->
    <bean class=
   "org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>
    <!-- 处理器适配器，配置对处理器中handleRequest()方法的调用-->
    <bean class=
  "org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"/>
    <!-- 视图解析器 -->
    <bean class=
  "org.springframework.web.servlet.view.InternalResourceViewResolver" />
</beans>
```

在文件 11-3 中，首先定义了一个名称为“/firstController”的 Bean，该 Bean 会将控制器类 FirstController 映射到“/firstController”请求中；然后配置了处理器映射器 BeanNameUrlHandlerMapping 和处理器适配器 SimpleControllerHandlerAdapter，其中处理器映射器用于将处理器 Bean 中的的 name（即 url）进行处理器查找，而处理器适配器用于完成对 FirstController 处理器中 handleRequest()方法的调用。最后配置了视图解析器 InternalResourceViewResolver 来解析结果视图，并将结果呈现给用户。

小提示

在老版本的 Spring 中，配置文件内必须要配置处理器映射器、处理器适配器和视图解析器，但在 Spring 4.0 以后，如果不配置处理器映射器、处理器适配器和视图解析器，也会使用 Spring 内部默认的配置来完成相应的工作，这里显示的配置处理器映射器、处理器适配器和视图解析器是为了让读者能够更加清晰地了解 Spring MVC 的工作流程。

5. 创建视图（View）页面

在 WEB-INF 目录下，创建一个 jsp 文件夹，并在文件夹中创建一个页面文件 first.jsp，在该页面中使用 EL 表达式获取 msg 中的信息，如文件 11-4 所示。

文件 11-4　first.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>入门程序</title>
</head>
<body>
    ${msg}
</body>
</html>
```

6. 启动项目，测试应用

全部文件创建完成后，项目的文件结构如图 11-2 所示。

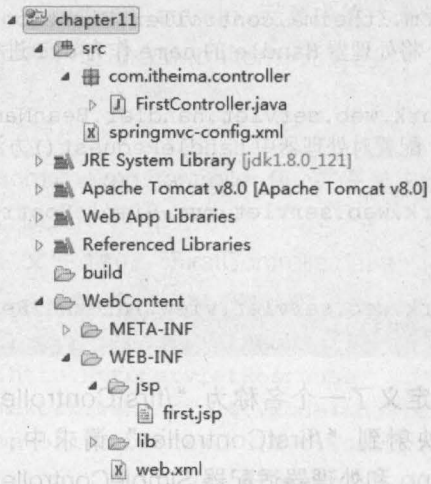


图11-2　案例完成时的项目目录结构

将 chapter11 项目发布到 Tomcat 中，并启动 Tomcat 服务器。在浏览器中访问地址 http://localhost:8080/chapter11/firstController，其显示效果如图 11-3 所示。

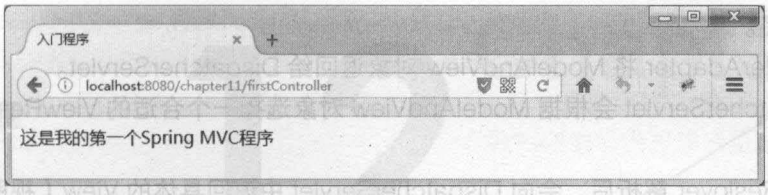


图11-3　访问结果

从图 11-3 可以看到，浏览器中已经显示出了模型对象中的字符串信息，这也就说明第一个 Spring MVC 程序执行成功。

11.3 Spring MVC 的工作流程

通过 11.2 节入门案例的学习，相信读者对 Spring MVC 的使用已经有了一个初步的了解，但是程序在项目中具体是怎么执行的呢？下面通过一张图来展示 Spring MVC 程序的执行情况，如图 11-4 所示。

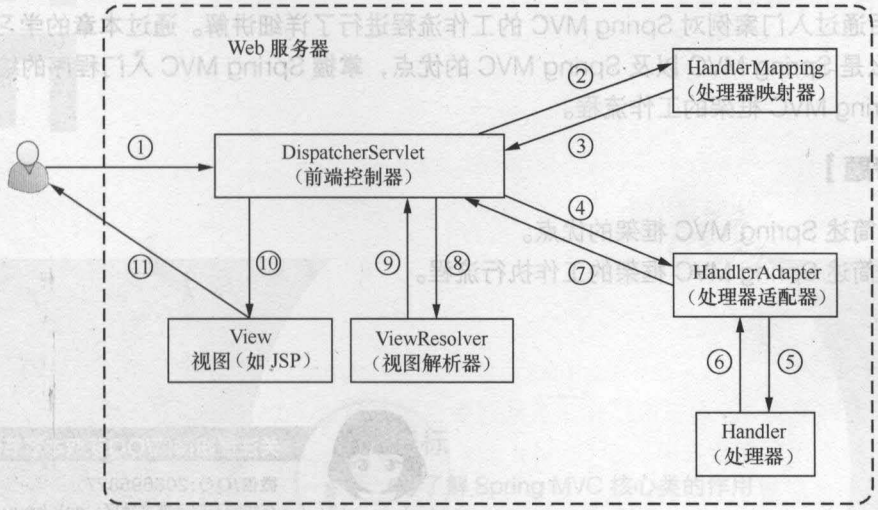


图11-4　Spring MVC的工作原理

按照图 11-4 中所标注的序号，Spring MVC 程序的完整执行流程如下。

（1）用户通过浏览器向服务器发送请求，请求会被 Spring MVC 的前端控制器 DispatcherServlet 所拦截。

（2）DispatcherServlet 拦截到请求后，会调用 HandlerMapping 处理器映射器。

（3）处理器映射器根据请求 URL 找到具体的处理器，生成处理器对象及处理器拦截器（如果有则生成）一并返回给 DispatcherServlet。

（4）DispatcherServlet 会通过返回信息选择合适的 HandlerAdapter（处理器适配器）。

（5）HandlerAdapter 会调用并执行 Handler（处理器），这里的处理器指的就是程序中编写的 Controller 类，也被称之为后端控制器。

（6）Controller 执行完成后，会返回一个 ModelAndView 对象，该对象中会包含视图名或包

含模型和视图名。

（7）HandlerAdapter 将 ModelAndView 对象返回给 DispatcherServlet。

（8）DispatcherServlet 会根据 ModelAndView 对象选择一个合适的 ViewReslover（视图解析器）。

（9）ViewReslover 解析后，会向 DispatcherServlet 中返回具体的 View（视图）。

（10）DispatcherServlet 对 View 进行渲染（即将模型数据填充至视图中）。

（11）视图渲染结果会返回给客户端浏览器显示。

在上述执行过程中，DispatcherServlet、HandlerMapping、HandlerAdapter 和 ViewResolver 对象的工作是在框架内部执行的，开发人员并不需要关心这些对象内部的实现过程，只需要配置前端控制器（DispatcherServlet），完成 Controller 中的业务处理，并在视图中（View）中展示相应信息即可。

11.4 本章小结

本章首先对 Spring MVC 框架进行了简单的介绍，然后讲解了一个 Spring MVC 入门程序的编写，最后通过入门案例对 Spring MVC 的工作流程进行了详细讲解。通过本章的学习，读者能够了解什么是 Spring MVC 以及 Spring MVC 的优点，掌握 Spring MVC 入门程序的编写，并能够熟悉 Spring MVC 框架的工作流程。

【思考题】

1. 请简述 Spring MVC 框架的优点。
2. 请简述 Spring MVC 框架的工作执行流程。

Java EE

12 Chapter

第12章 Spring MVC的核心类和注解

学习目标

- 了解 Spring MVC 核心类的作用
- 掌握 Spring MVC 常用注解的使用

在 Spring 2.5 之前，只能使用实现 Controller 接口的方式来开发一个控制器，第 11 章的入门案例就是使用的此种方式。在 Spring 2.5 之后，新增加了基于注解的控制器以及其他一些常用注解，这些注解的使用极大地减少了程序员的开发工作。接下来，本章将对 Spring MVC 中的常用核心类及其常用注解进行详细的讲解。

12.1 DispatcherServlet

DispatcherServlet 的全名是 org.springframework.web.servlet.DispatcherServlet，它在程序中充当着前端控制器的角色。在使用时，只需将其配置在项目的 web.xml 文件中，其配置代码如下。

```
<servlet>
    <!-- 配置前端过滤器 -->
    <servlet-name>springmvc</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <!-- 初始化时加载配置文件 -->
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:springmvc-config.xml</param-value>
    </init-param>
    <!-- 表示容器在启动时立即加载 Servlet -->
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>springmvc</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

在上述代码中，<load-on-startup>元素和<init-param>元素都是可选的。如果<load-on-startup>元素的值为 1，则在应用程序启动时会立即加载该 Servlet；如果<load-on-startup>元素不存在，则应用程序会在第一个 Servlet 请求时加载该 Servlet。如果<init-param>元素存在并且通过其子元素配置了 Spring MVC 配置文件的路径，则应用程序在启动时会加载配置路径下的配置文件；如果没有通过<init-param>元素配置，则应用程序会默认到 WEB-INF 目录下寻找如下方式命名的配置文件。

```
servletName-servlet.xml
```

其中，servletName 指的是部署在 web.xml 中的 DispatcherServlet 的名称，在上面 web.xml 中的配置代码中即为 springmvc，而-servlet.xml 是配置文件名的固定写法，所以应用程序会在 WEB-INF 下寻找 springmvc-servlet.xml。

12.2 Controller 注解类型

org.springframework.stereotype.Controller 注解类型用于指示 Spring 类的实例是一个控制器，其注解形式为@Controller。该注解在使用时不需要再实现 Controller 接口，只需要将@Controller

注解加入到控制器类上，然后通过 Spring 的扫描机制找到标注了该注解的控制器即可。

@Controller 注解在控制器类中的使用示例如下。

```
package com.itheima.controller;
import org.springframework.stereotype.Controller;
...
@Controller
public class FirstController{
    ...
}
```

为了保证 Spring 能够找到控制器类，还需要在 Spring MVC 的配置文件中添加相应的扫描配置信息，具体如下。

（1）在配置文件的声明中引入 spring-context。

（2）使用<context:component-scan>元素指定需要扫描的类包。

一个完整的配置文件示例如文件 12-1 所示。

文件 12-1 springmvc-config.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xmlns:context="http://www.springframework.org/schema/context"
5   xsi:schemaLocation="http://www.springframework.org/schema/beans
6   http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
7   http://www.springframework.org/schema/context
8   http://www.springframework.org/schema/context/spring-context-4.3.xsd">
9      <!-- 指定需要扫描的包 -->
10     <context:component-scan base-package="com.itheima.controller" />
11 </beans>
```

在文件 12-1 中，<context:component-scan>元素的属性 base-package 指定了需要扫描的类包为 com.itheima.controller。在运行时，该类包及其子包下所有标注了注解的类都会被 Spring 所处理。

与实现了 Controller 接口的方式相比，使用注解的方式显然更加简单。同时，Controller 接口的实现类只能处理一个单一的请求动作，而基于注解的控制器可以同时处理多个请求动作，在使用上更加的灵活。因此，在实际开发中通常都会使用基于注解的形式。

注意

使用注解方式时，程序的运行需要依赖 Spring 的 AOP 包，因此需要向 lib 目录中添加 spring-aop-4.3.6.RELEASE.jar，否则程序运行时会报错。

12.3 RequestMapping 注解类型

12.3.1 @RequestMapping 注解的使用

Spring 通过@Controller 注解找到相应的控制器类后，还需要知道控制器内部对每一个请求是如何处理的，这就需要使用 org.springframework.web.bind.annotation.RequestMapping

注解类型。RequestMapping 注解类型用于映射一个请求或一个方法，其注解形式为@RequestMapping，可以使用该注解标注在一个方法或一个类上。

1. 标注在方法上

当标注在一个方法上时，该方法将成为一个请求处理方法，它会在程序接收到对应的 URL 请求时被调用。使用@RequestMapping 注解标注在方法上的示例如下。

```
package com.itheima.controller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
...
@Controller
public class FirstController{
    @RequestMapping(value="/firstController")
    public ModelAndView handleRequest(HttpServletRequest request,
            HttpServletResponse response) {
        ...
        return mav;
    }
}
```

使用@RequestMapping 注解后，上述代码中的 handleRequest()方法就可以通过地址：http://localhost:8080/chapter12/**firstController** 进行访问。

2. 标注在类上

当标注在一个类上时，该类中的所有方法都将映射为相对于类级别的请求，表示该控制器所处理的所有请求都被映射到 value 属性值所指定的路径下。使用@RequestMapping 注解标注在类上的示例如下。

```
package com.itheima.controller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
...
@Controller
@RequestMapping(value="/hello")
public class FirstController{
    @RequestMapping(value="/firstController")
    public ModelAndView handleRequest(HttpServletRequest request,
            HttpServletResponse response){
        ...
        return mav;
    }
}
```

由于在类上添加了@RequestMapping 注解，并且其 value 属性值为“/hello”，所以上述代码方法的请求路径将变为：http://localhost:8080/chapter12**/hello/firstController**。如果该类中还包含其他方法，那么在其他方法的请求路径中也需要加入“/hello”。

12.3.2 @RequestMapping 注解的属性

@RequestMapping 注解除了可以指定 value 属性外，还可以指定其他一些属性，这些属性

如表 12-1 所示。

表 12-1 @RequestMapping 注解的属性

属性名	类型	描述
name	String	可选属性，用于为映射地址指定别名
value	String[]	可选属性，同时也是默认属性，用于映射一个请求和一种方法，可以标注在一个方法或一个类上
method	RequestMethod[]	可选属性，用于指定该方法用于处理哪种类型的请求方式，其请求方式包括 GET、POST、HEAD、OPTIONS、 PUT、PATCH、DELETE 和 TRACE， 例如 method=RequestMethod.GET 表示只支持 GET 请求，如果需要支持多个请求方式则需要通过{}写成数组的形式，并且多个请求方式之间是有英文逗号分隔
params	String[]	可选属性，用于指定 Request 中必须包含某些参数的值，才可以通过其标注的方法处理
headers	String[]	可选属性，用于指定 Request 中必须包含某些指定的 header 的值，才可以通过其标注的方法处理
consumes	String[]	可选属性，用于指定处理请求的提交内容类型（Content-type），比如 application/json、text/html 等
produces	String[]	可选属性，用于指定返回的内容类型，返回的内容类型必须是 request 请求头（Accept）中所包含的类型

在表 12-1 中，所有属性都是可选的，但其默认属性是 value。当 value 是其唯一属性时，可以省略属性名，例如下面两种标注的含义相同。

```
@RequestMapping(value="/firstController")
@RequestMapping("/firstController")
```

12.3.3 组合注解

前面两个小节已经对@RequestMapping 注解及其属性进行了详细讲解，而在 Spring 框架的 4.3 版本中，引入了组合注解，来帮助简化常用的 HTTP 方法的映射，并更好地表达被注解方法的语义。其组合注解如下所示。

- @GetMapping：匹配 GET 方式的请求。
- @PostMapping：匹配 POST 方式的请求。
- @PutMapping：匹配 PUT 方式的请求。
- @DeleteMapping：匹配 DELETE 方式的请求。
- @PatchMapping：匹配 PATCH 方式的请求。

以@GetMapping 为例，该组合注解是@RequestMapping(method = RequestMethod.GET)的缩写，它会将 HTTP GET 映射到特定的处理方法上。在实际开发中，传统的@RequestMapping 注解使用方式如下。

```
@RequestMapping(value="/user/{id}",method=RequestMethod.GET)
public String selectUserById(String id){
    ...
}
```

而使用新注解@GetMapping 后，可以省略 method 属性，从而简化代码，其使用方式如下。

```
@GetMapping(value="/user/{id}")
public String selectUserById(String id){
    ...
}
```

12.3.4 请求处理方法的参数类型和返回类型

在控制器类中，每一个请求处理方法都可以有多个不同类型的参数，以及一个多种类型的返回结果。例如在入门案例中，handleRequest()方法的参数就是对应请求的 HttpServletRequest 和 HttpServletResponse 两种参数类型。除此之外，还可以使用其他的参数类型，例如在请求处理方法中需要访问 HttpSession 对象，则可以添加 HttpSession 作为参数，Spring 会将对象正确地传递给方法，其使用示例如下。

```
@RequestMapping(value="/firstController")
public ModelAndView (HttpSession session){
    ...
    return mav;
}
```

在请求处理方法中，可以出现的参数类型如下。

- javax.servlet.ServletRequest / javax.servlet.http.HttpServletRequest
- javax.servlet.ServletResponse / javax.servlet.http.HttpServletResponse
- javax.servlet.http.HttpSession
- org.springframework.web.context.request.WebRequest 或 org.springframework.web.context.request.NativeWebRequest
- java.util.Locale
- java.util.TimeZone (Java 6+) / java.time.ZoneId (on Java 8)
- java.io.InputStream / java.io.Reader
- java.io.OutputStream / java.io.Writer
- org.springframework.http.HttpMethod
- java.security.Principal
- @PathVariable、@MatrixVariable、@RequestParam、@RequestHeader、@RequestBody、@RequestPart、@SessionAttribute、@RequestAttribute 注解
- HttpEntity<?>
- java.util.Map / org.springframework.ui.Model /org.springframework.ui.ModelMap
- org.springframework.web.servlet.mvc.support.RedirectAttributes
- org.springframework.validation.Errors /org.springframework.validation.BindingResult
- org.springframework.web.bind.support.SessionStatus
- org.springframework.web.util.UriComponentsBuilder

需要注意的是，org.springframework.ui.Model 类型不是一个 Servlet API 类型，而是一个包含了 Map 对象的 Spring MVC 类型。如果方法中添加了 Model 参数，则每次调用该请求处理方法时，Spring MVC 都会创建 Model 对象，并将其作为参数传递给方法。

在入门案例中，请求处理方法返回的是一个 ModelAndView 类型的数据。除了此种类型外，请求处理方法还可以返回其他类型的数据。Spring MVC 所支持的常见方法返回类型如下。

- ModelAndView
- Model
- Map
- View
- String
- void
- HttpEntity<?>或 ResponseEntity<?>
- Callable<?>
- DeferredResult<?>

在上述所列举的返回类型中，常见的返回类型是 ModelAndView、String 和 void。其中 ModelAndView 类型中可以添加 Model 数据，并指定视图；String 类型的返回值可以跳转视图，但不能携带数据；而 void 类型主要在异步请求时使用，它只返回数据，而不会跳转视图。

由于 ModelAndView 类型未能实现数据与视图之间的解耦，所以在企业开发时，方法的返回类型通常都会使用 String。既然 String 类型的返回值不能携带数据，那么在方法中是如何将数据带入视图页面的呢？这就用到了上面所讲解的 Model 参数类型，通过该参数类型，即可添加需要在视图中显示的属性。

返回 String 类型方法的示例代码如下。

```
@RequestMapping(value="/firstController")
public String handleRequest(HttpServletRequest request,
                            HttpServletResponse response, Model model) {
    // 向模型对象中添加数据
    model.addAttribute("msg", "这是我的第一个 Spring MVC 程序");
    // 返回视图页面
    return "/WEB-INF/jsp/first.jsp";
}
```

在上述方法代码中，增加了一个 Model 类型的参数，通过该参数实例的 addAttribute()方法即可添加所需数据。

String 类型除了可以返回上述代码中的视图页面外，还可以进行重定向与请求转发，具体方式如下。

1. redirect 重定向

例如，在修改用户信息操作后，将请求重定向到用户查询方法的实现代码如下。

```
@RequestMapping(value="/update")
public String update(HttpServletRequest request,
                            HttpServletResponse response, Model model){
    ...
    // 重定向请求路径
    return "redirect:queryUser";
}
```

2. forward 请求转发

例如，用户执行修改操作时，转发到用户修改页面的实现代码如下。

```
@RequestMapping(value="/toEdit")
public String update(HttpServletRequest request,
                          HttpServletResponse response, Model model){
    ...
    // 请求转发
   return "forward:editUser";
}
```

关于重定向和转发的具体使用，在本书后面章节中会有具体的应用案例，由于篇幅有限，这里就不再过多介绍。

12.4 ViewResolver（视图解析器）

Spring MVC 中的视图解析器负责解析视图，可以通过在配置文件中定义一个 ViewResolver 来配置视图解析器，其配置示例如下。

```
<!-- 定义视图解析器 -->
<bean id="viewResolver" class=
    "org.springframework.web.servlet.view.InternalResourceViewResolver">
    <!-- 设置前缀 -->
    <property name="prefix" value="/WEB-INF/jsp/" />
    <!-- 设置后缀 -->
    <property name="suffix" value=".jsp" />
</bean>
```

在上述代码中，定义了一个 id 为 viewResolver 的视图解析器，并设置了视图的前缀和后缀属性。这样设置后，方法中所定义的 view 路径将可以简化。例如，入门案例中的逻辑视图名只需设置为“first”，而不再需要设置为“/WEB-INF/jsp/first.jsp”，在访问时视图解析器会自动地增加前缀和后缀。

12.5 应用案例——基于注解的 Spring MVC 应用

通过前几个小节的学习，相信读者对 Spring MVC 的核心类和注解的使用已经有了一个初步的了解。为了帮助读者掌握这些知识，本节将通过前面所学内容，以注解的方式对入门案例进行改写，具体实现步骤如下。

1. 搭建项目环境

在 Eclipse 中，创建一个名为 chapter12 的 Web 项目，将 chapter11 项目中的所有 JAR 包以及编写的所有文件复制到 chapter12 项目，并向 lib 目录添加 Spring AOP 所需的 JAR（spring-aop-4.3.6.RELEASE.jar）。搭建后的项目结构如图 12-1 所示。

2. 修改配置文件

在 springmvc-config.xml 中添加注解扫描配置，并定义视图解析器，如文件 12-2 所示。

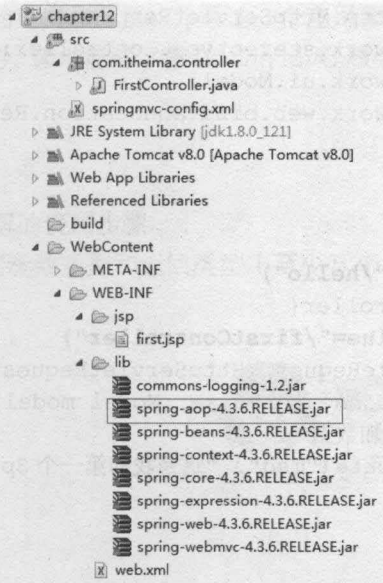


图12-1 chapter12的项目目录结构

文件 12-2 springmvc-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
       http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context-4.3.xsd">
    <!-- 指定需要扫描的包 -->
    <context:component-scan base-package="com.itheima.controller" />
    <!-- 定义视图解析器 -->
    <bean id="viewResolver" class=
   "org.springframework.web.servlet.view.InternalResourceViewResolver">
        <!-- 设置前缀 -->
        <property name="prefix" value="/WEB-INF/jsp/" />
        <!-- 设置后缀 -->
        <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

在文件 12-2 中，首先通过组件扫描器指定了需要扫描的包，然后定义了视图解析器，并在视图解析器中设置了视图文件的路径前缀和文件后缀名。

3. 修改 Controller 类

修改 FirstController 类，在类和方法上添加相应注解，如文件 12-3 所示。

文件 12-3 FirstController.java

```
package com.itheima.controller;
import javax.servlet.http.HttpServletRequest;
```

```
import javax.servlet.http.HttpServletResponse;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
/**
 * 控制器类
 */
@Controller
@RequestMapping(value="/hello")
public class FirstController{
    @RequestMapping(value="/firstController")
    public String handleRequest(HttpServletRequest request,
        HttpServletResponse response, Model model) throws Exception {
        // 向模型对象中添加数据
        model.addAttribute("msg", "这是我的第一个 Spring MVC 程序");
        // 返回视图页面
        return "first";
    }
}
```

在文件 12-3 中，使用了@Controller 注解来标注控制器类，并使用了@RequestMapping 注解标注在类名和方法名上来映射请求方法。在项目启动时，Spring 就会扫描到此类，以及此类中标注了@RequestMapping 注解的方法。由于标注在类上的@RequestMapping 注解的 value 值为“/hello”，因此类中所有请求方法的路径都需要加上“/hello”。由于类中的 handlerRequest() 方法的返回类型为 String，而 String 类型的返回值又无法携带数据，所以需要通过参数 Model 对象的 addAttribute()方法来添加数据信息。因为在配置文件的视图解析器中定义了视图文件的前缀和后缀名，所以 handleRequest()方法只需返回视图名“first”即可，在访问此方法时，系统会自动访问“/WEB-INF/jsp/”路径下名称为 first 的 jsp 文件。

4. 启动项目，测试应用

将项目发布到 Tomcat 服务器并启动，在浏览器中访问地址 http://localhost:8080/chapter12/hello/firstController，其显示效果如图 12-2 所示。

图12-2 显示效果

从图 12-2 可以看出，通过注解的方式，同样实现了第一个 Spring MVC 程序的运行。

12.6 本章小结

本章主要对 Spring MVC 的核心类及其相关注解的使用进行了详细的讲解。首先介绍了 DispatcherServlet 的作用和配置；然后介绍了 Controller 和 RequestMapping 注解类型的相关

知识；最后讲解了视图解析器的定义和配置，并通过一个应用案例，将本章所讲解的内容进行了一个全面总结。通过本章的学习，读者能够了解 Spring MVC 核心类的作用，并掌握 Spring MVC 常用注解的使用。

【思考题】

1. 请简述@Controller 注解的使用步骤。
2. 请列举请求处理方法的参数类型和返回类型（至少 5 个）。

Java EE

13 Chapter

第13章 数据绑定

学习目标

- 了解 Spring MVC 中的数据绑定的概念
- 熟悉 Spring MVC 中的几种数据绑定类型
- 掌握 Spring MVC 数据绑定的使用

通过第 12 章的学习，读者已经知道后台的请求处理方法可以包含多种参数类型。在实际的项目开发中，多数情况下客户端会传递带有不同参数的请求，那么后台是如何绑定并获取这些请求参数的呢？接下来，本章将对 Spring MVC 框架中的数据绑定进行详细讲解。

13.1 数据绑定介绍

在执行程序时，Spring MVC 会根据客户端请求参数的不同，将请求消息中的信息以一定的方式转换并绑定到控制器类的方法参数中。这种将请求消息数据与后台方法参数建立连接的过程就是 Spring MVC 中的数据绑定。

在数据绑定过程中，Spring MVC 框架会通过数据绑定组件（DataBinder）将请求参数串的内容进行类型转换，然后将转换后的值赋给控制器类中方法的形参，这样后台方法就可以正确绑定并获取客户端请求携带的参数了。整个数据绑定的过程如图 13-1 所示。

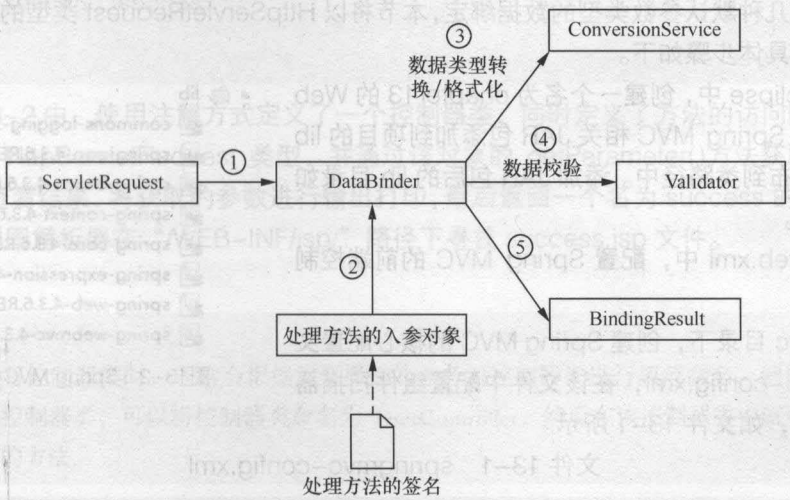


图13-1 Spring MVC数据绑定的过程

关于图 13-1 中信息处理过程的步骤描述如下。

（1）Spring MVC 将 ServletRequest 对象传递给 DataBinder。

（2）将处理方法的入参对象传递给 DataBinder。

（3）DataBinder 调用 ConversionService 组件进行数据类型转换、数据格式化等工作，并将 ServletRequest 对象中的消息填充到参数对象中。

（4）调用 Validator 组件对已经绑定了请求消息数据的参数对象进行数据合法性校验。

（5）校验完成后会生成数据绑定结果 BindingResult 对象，Spring MVC 会将 BindingResult 对象中的内容赋给处理方法的相应参数。

根据客户端请求参数类型和个数的不同，我们将 Spring MVC 中的数据绑定主要分为简单数据绑定和复杂数据绑定，接下来的几个小节中，就对这两种类型数据绑定进行详细讲解。

13.2 简单数据绑定

13.2.1 绑定默认数据类型

当前端请求的参数比较简单时，可以在后台方法的形参中直接使用 Spring MVC 提供的默认参数类型进行数据绑定。

常用的默认参数类型如下。

- HttpServletRequest：通过 request 对象获取请求信息。
- HttpServletResponse：通过 response 处理响应信息。
- HttpSession：通过 session 对象得到 session 中存储的对象。
- Model/ModelMap：Model 是一个接口，ModelMap 是一个接口实现，作用是将 model 数据填充到 request 域。

针对以上几种默认参数类型的数据绑定，本节将以 HttpServletRequest 类型的使用为例进行演示说明，其具体步骤如下。

（1）在 Eclipse 中，创建一个名为 chapter13 的 Web 项目，然后将 Spring MVC 相关 JAR 包添加到项目的 lib 目录下，并发布到类路径中。添加 JAR 包后的 lib 目录如图 13-2 所示。

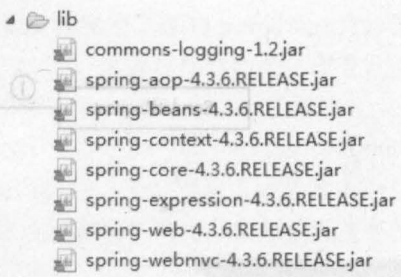


图13-2 Spring MVC相关JAR包

（2）在 web.xml 中，配置 Spring MVC 的前端控制器等信息。

（3）在 src 目录下，创建 Spring MVC 的核心配置文件 springmvc-config.xml，在该文件中配置组件扫描器和视图解析器，如文件 13-1 所示。

文件 13-1 springmvc-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context-4.3.xsd">
    <!-- 定义组件扫描器，指定需要扫描的包 -->
    <context:component-scan base-package="com.itheima.controller" />
    <!-- 定义视图解析器 -->
    <bean id="viewResolver" class=
    "org.springframework.web.servlet.view.InternalResourceViewResolver">
        <!-- 设置前缀 -->
        <property name="prefix" value="/WEB-INF/jsp/" />
        <!-- 设置后缀 -->
        <property name="suffix" value=".jsp" />
```

```
18      </bean>
19  </beans>
```

（4）在 src 目录下，创建一个 com.itheima.controller 包，在该包下创建一个用于用户操作的控制器类 UserController，编写后的代码如文件 13-2 所示。

文件 13-2　UserController.java

```
1  package com.itheima.controller;
2  import javax.servlet.http.HttpServletRequest;
3  import org.springframework.stereotype.Controller;
4  import org.springframework.web.bind.annotation.RequestMapping;
5  @Controller
6  public class UserController {
7      @RequestMapping("/selectUser")
8      public String selectUser(HttpServletRequest request) {
9          String id = request.getParameter("id");
10         System.out.println("id="+id);
11         return "success";
12     }
13 }
```

在文件 13-2 中，使用注解方式定义了一个控制器类，同时定义了方法的访问路径。在方法参数中使用了 HttpServletRequest 类型，并通过该对象的 getParameter()方法获取了指定的参数。为了方便查看结果，将获取的参数进行输出打印，最后返回一个名为 success 的视图，Spring MVC 会通过视图解析器在“/WEB-INF/jsp/”路径下寻找 success.jsp 文件。

小提示

后台在编写控制器类时，通常会根据需要操作的业务对控制器类进行规范命名。例如要编写一个对用户操作的控制器类，可以将控制器类命名为 UserController，然后在该控制器类中就可以编写任何有关用户操作的方法。

（5）在 WEB-INF 目录下，创建一个名为 jsp 的文件夹，然后在该文件夹中创建页面文件 success.jsp，该界面只作为正确执行操作后的响应页面，没有其他业务逻辑，如文件 13-3 所示。

文件 13-3　success.jsp

```
1  <%@ page language="java" contentType="text/html; charset=UTF-8"
2      pageEncoding="UTF-8"%>
3  <html>
4  <head>
5  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
6  <title>结果页面</title>
7  </head>
8  <body>
9      ok
10 </body>
11 </html>
```

（6）将 chapter13 项目发布到 Tomcat 服务器并启动，在浏览器中访问地址 http://localhost:

8080/chapter13/selectUser?id=1，其显示效果如图 13-3 所示。

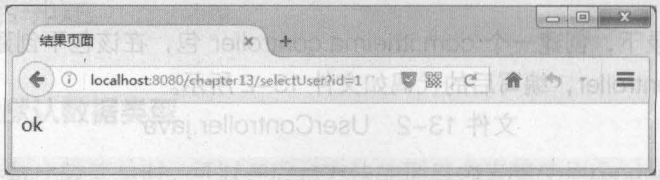


图13-3　success.jsp页面

此时的控制台打印信息如图 13-4 所示。

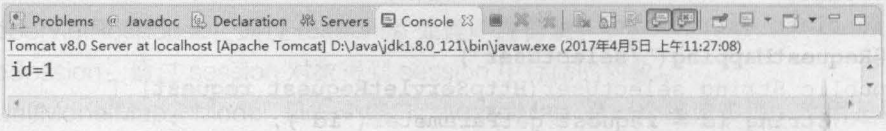


图13-4　运行结果

从图 13-4 可以看出，后台方法已从请求中正确地获取到了 id 的参数信息，这说明使用默认的 HttpServletRequest 参数类型已经完成了数据绑定。

13.2.2　绑定简单数据类型

简单数据类型的绑定，就是指 Java 中几种基本数据类型的绑定，如 int、String、Double 等类型。这里仍然以 13.2.1 小节中的参数 id 为 1 的请求为例，来讲解简单数据类型的绑定。

首先修改控制器类，将控制器类 UserController 中的 selectUser()方法的参数修改为使用简单数据类型的形式，修改后的代码如下。

```
@RequestMapping("/selectUser")
public String selectUser(Integer id) {
    System.out.println("id="+id);
    return "success";
}
```

与默认数据类型案例中的 selectUser()方法相比，此方法中只是将 HttpServletRequest 参数类型替换为了 Integer 类型。

启动项目，并在浏览器中访问地址 http://localhost:8080/chapter13/selectUser?id=1，会发现浏览器同样正确跳转到了 success.jsp 页面，此时再查看控制台打印信息，如图 13-5 所示。

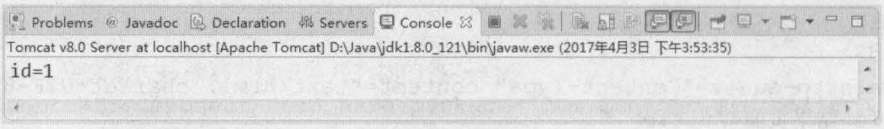


图13-5　运行结果

从图 13-5 可以看出，使用简单的数据类型同样完成了数据绑定。

需要注意的是，有时候前端请求中参数名和后台控制器类方法中的形参名不一样，这就会导致后台无法正确绑定并接收到前端请求的参数。为此，Spring MVC 提供了@RequestParam 注解来进行间接数据绑定。

@RequestParam 注解主要用于对请求中的参数进行定义，在使用时可以指定它的 4 个属性，具体如表 13-1 所示。

表 13-1 @RequestParam 注解的属性及说明

属性	说明
value	name 属性的别名，这里指参数的名字，即入参的请求参数名字，如 value="item_id"表示请求的参数中名字为 item_id 的参数的值将传入。如果只使用 vaule 属性，则可以省略 value 属性名
name	指定请求头绑定的名称
required	用于指定参数是否必须，默认是 true，表示请求中一定要有相应的参数
defaultValue	默认值，表示如果请求中没有同名参数时的默认值

@RequestParam 注解的使用非常简单，假设浏览器中的请求地址为 http://localhost:8080/chapter13/selectUser?user_id=1，那么在后台 selectUser()方法中的使用方式如下。

```
@RequestMapping("/selectUser")
public String selectUser(@RequestParam(value="user_id")Integer id) {
    System.out.println("id="+id);
    return "success";
}
```

上述代码会将请求中 user_id 参数的值 1 赋给方法中的 id 形参。这样通过输出语句就可以输出 id 形参中的值。

13.2.3 绑定 POJO 类型

在使用简单数据类型绑定时，可以很容易地根据具体需求来定义方法中的形参类型和个数，然而在实际应用中，客户端请求可能会传递多个不同类型的参数数据，如果还使用简单数据类型进行绑定，那么就需要手动编写多个不同类型的参数，这种操作显然比较烦琐。此时就可以使用 POJO 类型进行数据绑定。

POJO 类型的数据绑定就是将所有关联的请求参数封装在一个 POJO 中，然后在方法中直接使用该 POJO 作为形参来完成数据绑定。

接下来通过一个用户注册案例，来演示 POJO 类型数据的绑定，具体实现步骤如下。

（1）在 src 目录下，创建一个 com.itheima.po 包，在该包下创建一个 User 类来封装用户注册的信息参数，编辑后如文件 13-4 所示。

文件 13-4 User.java

```
1  package com.itheima.po;
2  /**
3   * 用户 POJO 类
4   */
5  public class User {
6      private Integer id;       //用户 id
7      private String username; //用户
8      private Integer password;//用户密码
9      public Integer getId() {
10         return id;
```

```
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public Integer getPassword() {
        return password;
    }
    public void setPassword(Integer password) {
        this.password = password;
    }
}
```

（2）在控制器 UserController 类中，编写接收用户注册信息和向注册页面跳转的方法，代码如下所示。

```
/**
 * 向用户注册页面跳转
 */
@RequestMapping("/toRegister")
public String toRegister( ) {
    return "register";
}
/**
 * 接收用户注册信息
 */
@RequestMapping("/registerUser")
public String registerUser(User user) {
    String username = user.getUsername();
    Integer password = user.getPassword();
    System.out.println("username="+username);
    System.out.println("password="+password);
    return "success";
}
```

（3）在/WEB-INF/jsp 目录下，创建一个用户注册页面 register.jsp，在该界面中编写用户注册表单，表单需要以 POST 方式提交，并且在提交时会发送一条以“/registerUser”结尾的请求消息，如文件 13-5 所示。

文件 13-5　register.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>注册</title>
```

```
7  </head>
8  <body>
9      <form action="${pageContext.request.contextPath }/registerUser"
10                                                      method="post">
11         用户名: <input type="text" name="username" /><br />
12         密   码: <input type="text" name="password" /><br />
13         <input type="submit" value="注册"/>
14     </form>
15 </body>
16 </html>
```

注意

在使用 POJO 类型数据绑定时，前端请求的参数名（本例中指 form 表单内各元素的 name 属性值）必须与要绑定的 POJO 类中的属性名一样，这样才会自动将请求数据绑定到 POJO 对象中，否则后台接收的参数值为 null。

（4）将项目发布到 Tomcat 服务器并启动，在浏览器中访问地址 http://localhost:8080/chapter13/toRegister，就会跳转到用户注册页面 register.jsp，如图 13-6 所示。

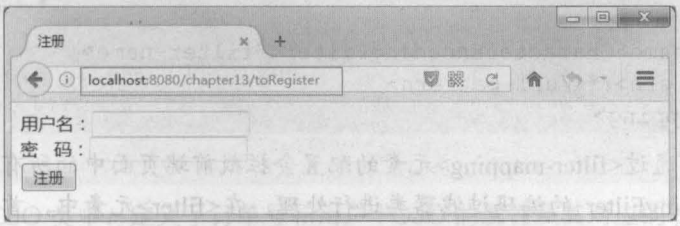


图13-6 register.jsp注册页面

在图 13-6 中，填写对应的用户名和密码，然后单击“注册”按钮即可完成模拟注册功能。这里假设用户注册的用户名和密码分别为“tom”和“123”，当单击“注册”按钮后，浏览器会跳转到结果页面，此时控制台的输出结果如图 13-7 所示。

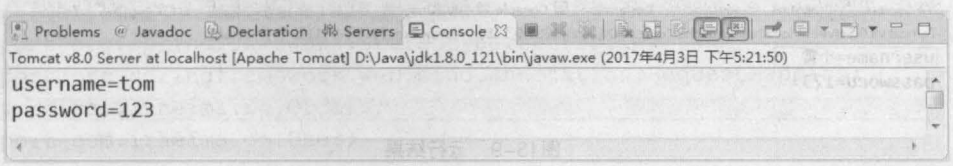


图13-7 运行结果

从图 13-7 可以看出，使用 POJO 类型同样可以获取前端请求传递过来的数据信息，这就是 POJO 类型的数据绑定。

多学一招：解决请求参数中的中文乱码问题

在前端请求中，难免会有中文信息传递，例如在图 13-6 中的用户名和密码输入框中输入用户名“小雪”和密码“123”时，虽然浏览器可以正确跳转到结果页面，但是在控制台中输出的中文信息却出现了乱码，如图 13-8 所示。

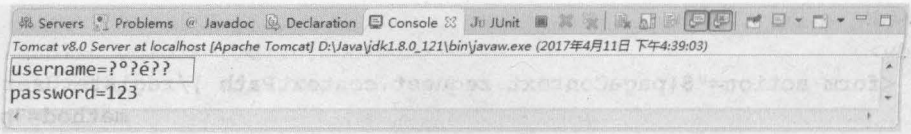


图13-8 运行结果

从图 13-8 可以看到，密码信息已正确显示，但用户名却显示为“?°?é??”。

为了防止前端传入的中文数据出现乱码问题，我们可以使用 Spring 提供的编码过滤器来统一编码。要使用编码过滤器，只需要在 web.xml 中添加如下代码。

```
<!-- 配置编码过滤器 -->
<filter>
  <filter-name>CharacterEncodingFilter</filter-name>
  <filter-class>
     org.springframework.web.filter.CharacterEncodingFilter
  </filter-class>
  <init-param>
     <param-name>encoding</param-name>
     <param-value>UTF-8</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>CharacterEncodingFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

上述代码中，通过<filter-mapping>元素的配置会拦截前端页面中的所有请求，并交由名称为 CharacterEncodingFilter 的编码过滤器类进行处理。在<filter>元素中，首先配置了编码过滤器类 org.springframework.web.filter.CharacterEncodingFilter，然后通过初始化参数设置统一的编码为 UTF-8。这样所有的请求信息内容都会以 UTF-8 的编码格式进行解析。

配置完成后，再次在页面中输入中文用户名“小雪”以及密码“123”，此时控制台的打印信息如图 13-9 所示。

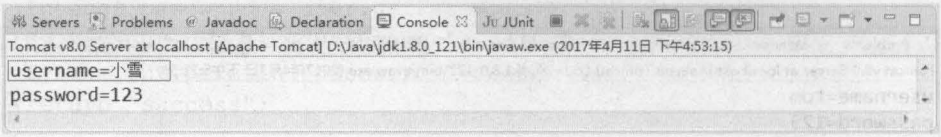


图13-9 运行结果

从图 13-9 可以看出，控制台中已经正确显示出了中文数据，这说明编码过滤器配置成功。

13.2.4 绑定包装 POJO

使用简单 POJO 类型已经可以完成多数的数据绑定，但有时客户端请求中传递的参数会比较复杂。例如，在用户查询订单时，页面传递的参数可能包括订单编号、用户名称等信息，这就包含了订单和用户两个对象的信息。如果将订单和用户的所有查询条件都封装在一个简单 POJO 中，显然会比较混乱，这时就可以考虑使用包装 POJO 类型的数据绑定。

所谓的包装 POJO，就是在一个 POJO 中包含另一个简单 POJO。例如，在订单对象中包含

用户对象。这样在使用时，就可以通过订单查询到用户信息。

下面通过一个订单查询的案例，来演示包装 POJO 数据绑定的使用，具体步骤如下。

（1）在 chapter13 项目的 com.itheima.po 包中，创建一个订单类 Orders，该类用于封装订单和用户信息，编辑后如文件 13-6 所示。

文件 13-6　Orders.java

```
1  package com.itheima.po;
2  /**
3   * 订单 POJO
4   */
5  public class Orders {
6      private Integer ordersId; // 订单编号
7      private User user;        // 用户 POJO，所属用户
8      public Integer getOrdersId() {
9          return ordersId;
10     }
11     public void setOrdersId(Integer ordersId) {
12         this.ordersId = ordersId;
13     }
14     public User getUser() {
15         return user;
16     }
17     public void setUser(User user) {
18         this.user = user;
19     }
20 }
```

在上述包装 POJO 类中，定义了订单号和用户 POJO 的属性及其对应的 getter/setter 方法。这样订单类中就不仅可以封装订单的基本属性参数，还可以封装 User 类型的属性参数。

（2）在 com.itheima.controller 包中，创建一个订单控制器类 OrdersController，在该类中编写一个跳转到订单查询页面的方法和一个查询订单及用户信息的方法，如文件 13-7 所示。

文件 13-7　OrdersController.java

```
1  package com.itheima.controller;
2  import org.springframework.stereotype.Controller;
3  import org.springframework.web.bind.annotation.RequestMapping;
4  import com.itheima.po.Orders;
5  import com.itheima.po.User;
6  @Controller
7  public class OrdersController {
8      /**
9       * 向订单查询页面跳转
10      */
11     @RequestMapping("/tofindOrdersWithUser")
12     public String tofindOrdersWithUser() {
13         return "orders";
14     }
15     /**
16      * 查询订单和用户信息
```

```
17      */
18     @RequestMapping("/findOrdersWithUser")
19     public String findOrdersWithUser(Orders orders) {
20         Integer orderId = orders.getOrdersId();
21         User user = orders.getUser();
22         String username = user.getUsername();
23         System.out.println("orderId="+orderId);
24         System.out.println("username="+username);
25         return "success";
26     }
27 }
```

在文件 13-7 中，通过访问页面跳转方法即可跳转到 orders.jsp 中，而通过查询订单和用户信息方法，即可通过传递的参数条件去调用 Service 中的相应方法来查询数据。这里只是为了讲解包装 POJO 的使用，所以只需将传递过来的参数进行输出。

（3）在/WEB-INF/jsp 目录下，创建一个用户订单查询页面 orders.jsp，在页面中编写通过订单编号和所属用户作为查询条件来查询订单信息的代码，如文件 13-8 所示。

文件 13-8　orders.jsp

```
1  <%@ page language="java" contentType="text/html; charset=UTF-8"
2      pageEncoding="UTF-8"%>
3  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
4      "http://www.w3.org/TR/html4/loose.dtd">
5  <html>
6  <head>
7  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
8  <title>订单查询</title>
9  </head>
10 <body>
11     <form action="${pageContext.request.contextPath }/findOrdersWithUser"
12                                                          method="post">
13         订单编号：<input type="text" name="ordersId" /><br />
14         所属用户：<input type="text" name="user.username" /><br />
15                 <input type="submit" value="查询" />
16     </form>
17 </body>
18 </html>
```

注意

在使用包装 POJO 类型数据绑定时，前端请求的参数名编写必须符合以下两种情况。

① 如果查询条件参数是包装类的直接基本属性，则参数名直接用对应的属性名，如上面代码中的 ordersId。

② 如果查询条件参数是包装类中 POJO 的子属性，则参数名必须为【对象.属性】，其中【对象】要和包装 POJO 中的对象属性名称一致，【属性】要和包装 POJO 中的对象子属性一致，如上述代码中的 user.username。

（4）将 chapter13 项目发布到 Tomcat 服务器并启动，在浏览器中访问地址 http://localhost:

8080/chapter13/tofindOrdersWithUser，其显示效果如图 13-10 所示。

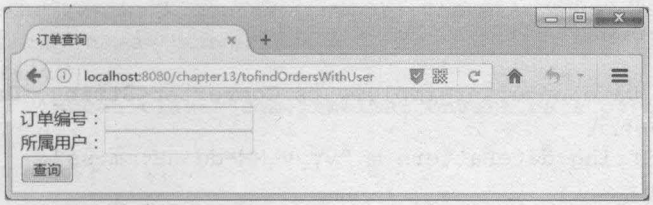


图13-10 orders.jsp订单页面

在图 13-10 中，填写订单编号为“321654”，所属用户为“小韩”，单击“查询”按钮后，浏览器会跳转到 success.jsp 页面，此时控制台中的打印信息如图 13-11 所示。

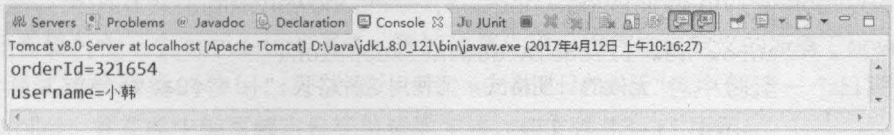


图13-11 运行结果

从图 13-11 可以看出，使用包装 POJO 同样完成了数据绑定。

13.2.5 自定义数据绑定

在一般情况下，使用基本数据类型和 POJO 类型的参数数据已经能够满足需求，然而有些特殊类型的参数是无法在后台进行直接转换的，例如日期数据就需要开发者自定义转换器（Converter）或格式化（Formatter）来进行数据绑定。

1. Converter

Spring 框架提供了一个 Converter 用于将一种类型的对象转换为另一种类型的对象。例如，用户输入的日期形式可能是“2017-04-08”或“2017/04/08”的字符串，而要 Spring 将输入的日期与后台的 Date 进行绑定，则需要将字符串转换为日期，此时就可以自定义一个 Converter 类来进行日期转换。

自定义 Converter 类需要实现 org.springframework.core.convert.converter.Converter 接口，该接口的代码如下所示。

```
public interface Converter<S, T> {
    T convert(S source);
}
```

在上述接口代码中，泛型中的 S 表示源类型，T 表示目标类型，而 convert(S source)表示接口中的方法。

在 src 目录下，创建一个 com.itheima.convert 包，在该包下创建日期转换类 DateConverter，并在该类中编写将 String 类型转换成 Date 类型的代码，如文件 13-9 所示。

文件 13-9 DateConverter.java

```
1  package com.itheima.convert;
2  import java.text.ParseException;
3  import java.text.SimpleDateFormat;
4  import java.util.Date;
```

```
import org.springframework.core.convert.converter.Converter;
/**
 * 自定义日期转换器
 */
public class DateConverter implements Converter<String, Date> {
    // 定义日期格式
    private String datePattern = "yyyy-MM-dd HH:mm:ss";
    @Override
    public Date convert(String source) {
        // 格式化日期
        SimpleDateFormat sdf = new SimpleDateFormat(datePattern);
        try {
            return sdf.parse(source);
        } catch (ParseException e) {
            throw new IllegalArgumentException(
                    "无效的日期格式，请使用这种格式:"+datePattern);
        }
    }
}
```

在上述代码中，DateConverter 类实现了 Converter 接口，该接口中第一个类型 String 表示需要被转换的数据类型，第二个类型 Date 表示需要转换成的目标类型。

为了让 Spring MVC 知道并使用这个转换器类，还需要在其配置文件中编写一个 id 为 conversionService 的 Bean，如文件 13-10 所示。

文件 13-10　springmvc-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:mvc="http://www.springframework.org/schema/mvc"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
  http://www.springframework.org/schema/mvc
  http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context-4.3.xsd">
    <!-- 定义组件扫描器，指定需要扫描的包 -->
    <context:component-scan base-package="com.itheima.controller" />
    <!-- 定义视图解析器 -->
    <bean id="viewResolver" class=
    "org.springframework.web.servlet.view.InternalResourceViewResolver">
        <!-- 设置前缀 -->
        <property name="prefix" value="/WEB-INF/jsp/" />
        <!-- 设置后缀 -->
        <property name="suffix" value=".jsp" />
    </bean>
    <!-- 显示的装配自定义类型转换器 -->
    <mvc:annotation-driven conversion-service="conversionService" />
    <!-- 自定义类型转换器配置 -->
```

```
    <bean id="conversionService" class=
    "org.springframework.context.support.ConversionServiceFactoryBean">
        <property name="converters">
            <set>
                <bean class="com.itheima.convert.DateConverter" />
            </set>
        </property>
    </bean>
</beans>
```

在文件 13-10 中，首先添加了 3 个 mvc 的 schema 信息；然后定义了组件扫描器和视图解析器；接下来显示装配了自定义的类型转换器；最后编写了自定义类型转换器的配置，其中 Bean 的类名称必须为 org.springframework.context.support.ConversionServiceFactoryBean，并且 Bean 中还需要包含一个 converters 属性，通过该属性列出程序中自定义的所有 Converter。

为了测试转换器类的使用，可以在 com.itheima.controller 包中创建一个日期控制器类 DateController，并在类中编写绑定日期数据的方法，如文件 13-11 所示。

文件 13-11 DateController.java

```
package com.itheima.controller;
import java.util.Date;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
/**
 * 日期控制器类
 */
@Controller
public class DateController {
    /**
     * 使用自定义类型数据绑定日期数据
     */
    @RequestMapping("/customDate")
    public String CustomDate(Date date) {
        System.out.println("date="+date);
        return "success";
    }
}
```

此时，如果发布项目并启动 Tomcat 服务器，在浏览器中访问地址 http://localhost:8080/chapter13/customDate?date= 2017-04-12 15:55:55 (注意日期数据中的空格)，控制台的打印信息如图 13-12 所示。

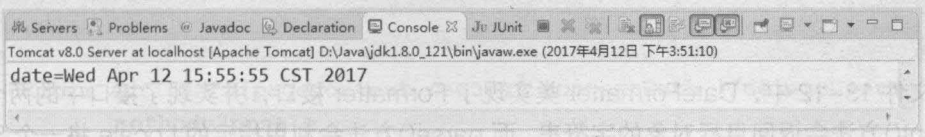


图13-12 运行结果

从图 13-12 可以看出，使用自定义类型转换器已从请求中正确获取到了日期信息，这就是自定义数据绑定。

2. Formatter

除了使用 Converter 进行转换外，我们还可以使用 Formatter 来进行类型转换。Formatter 与 Converter 的作用相同，只是 Formatter 的源类型必须是一个 String 类型，而 Converter 可以是任意类型。

使用 Formatter 自定义转换器类需要实现 org.springframework.format.Formatter 接口，该接口的代码如下所示。

```
public interface Formatter<T> extends Printer<T>, Parser<T> {}
```

在上述代码中，Formatter 接口继承了 Printer 和 Parser 接口，其泛型 T 表示输入字符串要转换的目标类型。在 Printer 和 Parser 接口中，分别包含一个 print()和 parse()方法，所有的实现类必须覆盖这两个方法。

在 com.itheima.convert 包中，创建日期转换类 DateFormatter，在该类中使用 Formatter 自定义日期转换器，如文件 13-12 所示。

文件 13-12　DateFormatter.java

```
1  package com.itheima.convert;
2  import java.text.ParseException;
3  import java.text.SimpleDateFormat;
4  import java.util.Date;
5  import java.util.Locale;
6  import org.springframework.format.Formatter;
7  /**
8   * 使用 Formatter 自定义日期转换器
9   */
10 public class DateFormatter implements Formatter<Date>{
11     // 定义日期格式
12     String datePattern = "yyyy-MM-dd HH:mm:ss";
13     // 声明 SimpleDateFormat 对象
14     private SimpleDateFormat simpleDateFormat;
15     @Override
16     public String print(Date date, Locale locale) {
17         return new SimpleDateFormat().format(date);
18     }
19     @Override
20     public Date parse(String source, Locale locale) throws ParseException
21     {
22         simpleDateFormat = new SimpleDateFormat(datePattern);
23         return simpleDateFormat.parse(source);
24     }
25 }
```

在文件 13-12 中，DateFormatter 类实现了 Formatter 接口，并实现了接口中的两个方法。其中 print()方法会返回目标对象的字符串，而 parse()方法会利用指定的 Locale 将一个 String 解析成目标类型。

要使用 Formatter 自定义的日期转换器，同样需要在 Spring MVC 的配置文件中进行注册，其配置代码如下所示。

```
<!-- 自定义类型格式化转换器配置 -->
<bean id="conversionService" class="org.springframework.format.support.
      FormattingConversionServiceFactoryBean">
   <property name="formatters">
      <set>
         <bean class="com.itheima.convert.DateFormatter" />
      </set>
   </property>
</bean>
```

与注册 Converter 类有所不同的是，注册自定义的 Formatter 转换器类时，Bean 的类名必须是 org.springframework.format.support.FormattingConversionServiceFactoryBean，并且其属性为 formatters。

完成后，通过地址 http://localhost:8080/chapter13/customDate?date= 2017-04-12 15:55:55 即可查看实现的效果。由于与图 13-12 的显示结果相同，这里不再做演示，读者可自行测试。

13.3 复杂数据绑定

在学习完前面小节讲解的简单数据绑定后，读者已经能够完成实际开发中多数的数据绑定问题，但仍可能遇到一些比较复杂的数据绑定问题，比如数组的绑定、集合的绑定，这在实际开发中也是十分常见的。接下来的两个小节中，将具体讲解一下数组绑定和集合绑定的使用。

13.3.1 绑定数组

在实际开发时，可能会遇到前端请求需要传递到后台一个或多个相同名称参数的情况（如批量删除），此种情况采用前面讲解的简单数据绑定的方式显然是不合适的。此时，就可以使用绑定数组的方式，来完成实际需求。

下面通过一个批量删除用户的例子来详细讲解绑定数组的操作，其具体实现步骤如下。

（1）在 chapter13 项目的/WEB-INF/jsp 目录下，创建一个展示用户信息的列表页面 user.jsp，编辑后的代码如文件 13-13 所示。

文件 13-13 user.jsp

```
1  <%@ page language="java" contentType="text/html; charset=UTF-8"
2      pageEncoding="UTF-8"%>
3  <html>
4  <head>
5  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
6  <title>用户列表</title>
7  </head>
8  <body>
9      <form action="${pageContext.request.contextPath }/deleteUsers"
10         method="post">
11        <table width="20%" border=1>
12           <tr>
13              <td>选择</td>
14              <td>用户名</td>
```

```
            </tr>
            <tr>
                <td><input name="ids" value="1" type="checkbox"></td>
                <td>tom</td>
            </tr>
            <tr>
                <td><input name="ids" value="2" type="checkbox"></td>
                <td>jack</td>
            </tr>
            <tr>
                <td><input name="ids" value="3" type="checkbox"></td>
                <td>lucy</td>
            </tr>
        </table>
        <input type="submit" value="删除"/>
    </form>
</body>
</html>
```

在上述页面代码中，定义了 3 个 name 属性相同而 value 属性值不同的复选框控件，并在每一个复选框对应的行中编写了一个对应用户。在单击“删除”按钮执行删除操作时，表单会提交到一个以“/deleteUsers”结尾的请求中。

（2）在控制器类 UserController 中，编写接收批量删除用户的方法（同时为了方便向用户列表页面跳转，还需增加一个向 user.jsp 页面跳转的方法），其代码如下所示。

```
/**
 * 向用户列表页面跳转
 */
@RequestMapping("/toUser")
public String selectUsers( ) {
    return "user";
}
/**
 * 接收批量删除用户的方法
 */
@RequestMapping("/deleteUsers")
public String deleteUsers(Integer[] ids) {
    if(ids !=null){
        for (Integer id : ids) {
            // 使用输出语句模拟已经删除的用户
            System.out.println("删除了 id 为"+id+"的用户！");
        }
    }else{
        System.out.println("ids=null");
    }
    return "success";
}
```

在上述代码中，先定义了一个向用户列表页面 user.jsp 跳转的方法，然后定义了一个接收前端批量删除用户的方法。在删除方法中，使用了 Integer 类型的数组进行数据绑定，并通过 for

循环执行具体数据的删除操作。

（3）发布项目到 Tomcat 服务器并启动后，在浏览器中访问地址 http://localhost:8080/chapter13/toUser，其显示效果如图 13-13 所示。

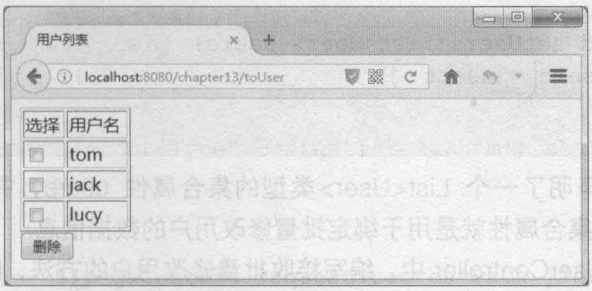


图13-13　user.jsp用户列表页面

勾选图 13-13 中的全部复选框，然后单击“删除”按钮，这样程序在正确执行后会跳转到 success.jsp 页面。此时控制台的打印信息，如图 13-14 所示。

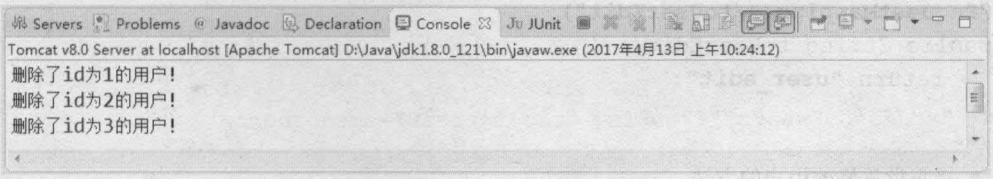


图13-14　运行结果

从图 13-14 可以看出，已成功执行了批量删除操作，这就说明已成功实现了数组类型的数据绑定。

13.3.2　绑定集合

在批量删除用户的操作中，前端请求传递的都是同名参数的用户 id，只要在后台使用同一种数组类型的参数绑定接收，就可以在方法中通过循环数组参数的方式来完成删除操作。但如果是批量修改用户操作，前端请求传递过来的数据可能就会批量包含各种类型的数据，如 Integer、String 等。这种情况使用数组绑定是无法实现的，那么我们应该怎么做呢?

针对这种情况，我们可以使用集合数据绑定。即在包装类中定义一个包含用户信息类的集合，然后在接收方法中将参数类型定义为该包装类的集合。

下面就以批量修改用户为例，来讲解一下集合数据绑定的使用，具体实现步骤如下。

（1）在 src 目录下，创建一个 com.itheima.vo 包，并在包中创建包装类 UserVO 来封装用户集合属性，编辑后的代码如文件 13-14 所示。

文件 13-14　UserVO.java

```
1  package com.itheima.vo;
2  import java.util.List;
3  import com.itheima.po.User;
4  /**
5   * 用户包装类
6   */
```

```
7  public class UserVO {
8      private List<User> users;
9      public List<User> getUsers() {
10         return users;
11     }
12     public void setUsers(List<User> users) {
13         this.users = users;
14     }
15 }
```

在上述代码中，声明了一个 List<User>类型的集合属性 users，并编写了该属性对应的 getter/setter 方法。该集合属性就是用于绑定批量修改用户的数据信息。

（2）在控制器类 UserController 中，编写接收批量修改用户的方法，以及向用户修改页面跳转的方法，其代码如下所示。

```
/**
 * 向用户批量修改页面跳转
 */
@RequestMapping("/toUserEdit")
public String toUserEdit() {
    return "user_edit";
}
/**
 * 接收批量修改用户的方法
 */
@RequestMapping("/editUsers")
public String editUsers(UserVO userList) {
    // 将所有用户数据封装到集合中
    List<User> users = userList.getUsers();
    // 循环输出所有用户信息
    for (User user : users) {
        // 如果接收的用户 id 不为空，则表示对该用户进行了修改
        if(user.getId() !=null){
            System.out.println("修改了 id 为"+user.getId()+
                    "的用户名为: "+user.getUsername());
        }
    }
    return "success";
}
```

在上述代码的两个方法中，通过 toUserEdit()方法将跳转到 user_edit.jsp 页面，通过 editUsers()方法将执行用户批量更新操作，其中 editUsers()方法的 UserVO 类型参数用于绑定并获取页面传递过来的用户数据。

注意

在使用集合数据绑定时，后台方法中不支持直接使用集合形参进行数据绑定，所以需要使用包装 POJO 作为形参，然后在包装 POJO 中包装一个集合属性。

（3）在项目的/WEB-INF/jsp 目录下，创建页面文件 user_edit.jsp，并编写页面信息，如文件 13-15 所示。

文件 13-15 user_edit.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>修改用户</title>
</head>
<body>
    <form action="${pageContext.request.contextPath }/editUsers"
          method="post" id='formid'>
        <table width="30%" border=1>
            <tr>
                <td>选择</td>
                <td>用户名</td>
            </tr>
            <tr>
              <td>
                <input name="users[0].id" value="1" type="checkbox" />
              </td>
              <td>
              <input name="users[0].username" value="tome" type="text" />
              </td>
            </tr>
            <tr>
              <td>
                <input name="users[1].id" value="2" type="checkbox" />
              </td>
              <td>
              <input name="users[1].username" value="jack" type="text" />
              </td>
            </tr>
        </table>
        <input type="submit" value="修改" />
    </form>
</body>
</html>
```

在上述页面代码中，模拟展示了 id 为 1、用户名为 tome 和 id 为 2、用户名为 jack 的两个用户。当单击“修改”按钮后，会将表单提交到一个以“/editUsers”结尾的请求中。

（4）发布项目到 Tomcat 服务器并启动后，在浏览器中访问地址 http://localhost:8080/chapter13/toUserEdit，其显示效果如图 13-15 所示。

将图 13-15 中的用户名 tome 改为 tom，jack 改为 jacks，并勾选两个数据前面的复选框，然后单击“修改”按钮后，浏览器会跳转到 success.jsp 页面中。此时控制台的打印信息如图 13-16 所示。

图13-15　user_edit.jsp页面

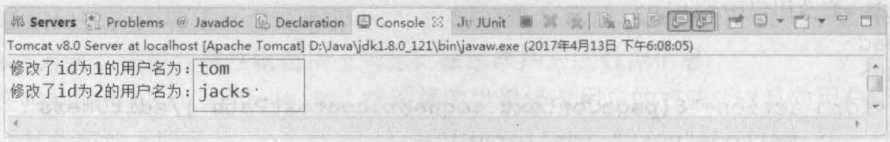


图13-16　运行结果

从图 13-16 可以看出，已经成功输出了请求中批量修改的用户信息，这就是集合类型的数据绑定。

13.4 本章小结

本章主要对 Spring MVC 中的数据绑定进行了详细讲解。首先讲解了简单的数据绑定，包括默认数据类型、简单数据类型、POJO 类型、包装 POJO 类型以及自定义参数类型绑定；然后讲解了复杂数据绑定，包括数组类型、集合类型绑定。通过本章的学习，读者能够熟练地掌握 Spring MVC 中几种数据类型的绑定使用，这将为后续的学习打下坚实的基础。

【思考题】

1. 请简述简单数据类型中的@RequestParam 注解及其属性作用。
2. 请简述包装 POJO 类型绑定时的注意事项。

Java EE

Chapter 14

第 14 章 JSON 数据交互和 RESTful 支持

学习目标

- 了解 JSON 的数据结构
- 掌握 Spring MVC 中 JSON 数据交互的使用
- 熟悉 RESTful 风格的请求样式
- 掌握 Spring MVC 中 RESTful 风格请求的使用

Spring MVC 在数据绑定的过程中，需要对传递数据的格式和类型进行转换，它既可以转换 String 类型的数据，也能够转换 JSON 等其他类型的数据。通过前面章节学习，读者已经掌握 String 等数据类型的转换和绑定，本章将针对 Spring MVC 中 JSON 类型的数据交互和 RESTful 支持进行详细讲解。

14.1 JSON 数据交互

JSON 是近几年才流行的一种新的数据格式，它与 XML 非常相似，都是用于存储数据的；但 JSON 相对于 XML 来说，解析速度更快，占用空间更小。因此在实际开发中，使用 JSON 格式的数据进行前后台的数据交互是很常见的。下面将对 Spring MVC 中 JSON 数据的交互内容进行详细的讲解。

14.1.1 JSON 概述

JSON（JavaScript Object Notation，JS 对象标记）是一种轻量级的数据交换格式。它是基于 JavaScript 的一个子集，使用了 C、C++、C#、Java、JavaScript、Perl、Python 等其他语言的约定，采用完全独立于编程语言的文本格式来存储和表示数据。这些特性使 JSON 成为理想的数据交互语言，它易于阅读和编写，同时也易于机器解析和生成。

与 XML 一样，JSON 也是基于纯文本的数据格式。初学者可以使用 JSON 传输一个简单的 String、Number、Boolean，也可以传输一个数组或者一个复杂的 Object 对象。

JSON 有如下两种数据结构。

1. 对象结构

对象结构以“{”开始，以“}”结束。中间部分由 0 个或多个以英文“,”分隔的 name/value 对构成（注意 name 和 value 之间以英文“:”分隔），其存储形式如图 14-1 所示。

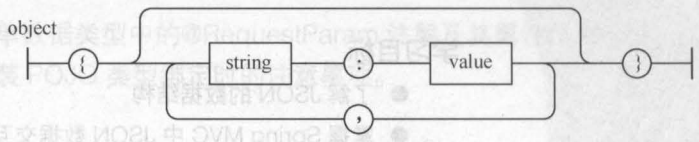


图14-1　存储对象

对象结构的语法结构代码如下。

```
{
    key1:value1,
    key2:value2,
    ...
}
```

其中关键字（key）必须为 String 类型，值（value）可以是 String、Number、Object、Array 等数据类型。例如，一个 address 对象包含城市、街道、邮编等信息，使用 JSON 的表示形式如下。

```
{"city":"Beijing","street":"Xisanqi","postcode":100096}
```

2. 数组结构

数组结构以“[”开始，以“]”结束。中间部分由 0 个或多个以英文“,”分隔的值的列表

组成，其存储形式如图 14-2 所示。

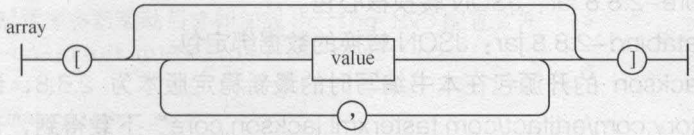


图14-2　存储数组

数组结构的语法结构代码如下。

```
[
    value1,
    value2,
    ...
]
```

例如，一个数组包含了 String、Number、Boolean、null 类型数据，使用 JSON 的表示形式如下。

```
["abc",12345,false,null]
```

上述两种（对象、数组）数据结构也可以分别组合构成更为复杂的数据结构。例如：一个 person 对象包含 name、hobby 和 address 对象，其代码表现形式如下。

```
{
    "name": "zhangsan"
    "hobby":["篮球","羽毛球","游泳"]
    "address":{
        "city":"Beijing"
        "street":"Xisanqi"
        "postcode":100096
    }
}
```

需要注意的是，如果使用 JSON 存储单个数据（如“abc”），一定要使用数组的形式，不要使用 Object 形式，因为 Object 形式必须是“名称：值”的形式。

14.1.2　JSON 数据转换

为了实现浏览器与控制器类（Controller）之间的数据交互，Spring 提供了一个 HttpMessageConverter<T>接口来完成此项工作。该接口主要用于将请求信息中的数据转换为一个类型为 T 的对象，并将类型为 T 的对象绑定到请求方法的参数中，或者将对象转换为响应信息传递给浏览器显示。

Spring 为 HttpMessageConverter<T>接口提供了很多实现类，这些实现类可以对不同类型的数据进行信息转换。其中 MappingJackson2HttpMessageConverter 是 Spring MVC 默认处理 JSON 格式请求响应的实现类。该实现类利用 Jackson 开源包读写 JSON 数据，将 Java 对象转换为 JSON 对象和 XML 文档，同时也可以将 JSON 对象和 XML 文档转换为 Java 对象。

要使用 MappingJackson2HttpMessageConverter 对数据进行转换，就需要使用 Jackson 的开源包，开发时所需的开源包及其描述如下所示。

- jackson-annoations-2.8.8.jar：JSON 转换注解包。
- jackson-core-2.8.8.jar：JSON 转换核心包。
- jackson-databind-2.8.8.jar：JSON 转换的数据绑定包。

以上 3 个 Jackson 的开源包在本书编写时的最新稳定版本为 2.8.8，读者可以通过链接“http://mvnrepository.com/artifact/com.fasterxml.jackson.core”下载得到，也可以在配套资源的源代码中找到 Jackson 的开源包。

在使用注解式开发时，需要用到两个重要的 JSON 格式转换注解，分别为@RequestBody 和@ResponseBody，关于这两个注解的说明如表 14-1 所示。

表 14-1　JSON 数据交互注解及说明

注解	说明
@RequestBody	用于将请求体中的数据绑定到方法的形参中。该注解用在方法的形参上
@ResponseBody	用于直接返回 return 对象。该注解用在方法上

了解了 Spring MVC 中 JSON 数据交互需要使用的类和注解后，接下来通过一个案例来演示如何进行 JSON 数据的交互，具体实现步骤如下。

（1）创建项目并导入相关 JAR 包。使用 Eclipse 创建一个名为 chapter14 的 Web 项目，然后将 Spring MVC 相关 JAR 包、JSON 转换包添加到项目的 lib 目录中，并发布到类路径下。添加后的 lib 目录如图 14-3 所示。

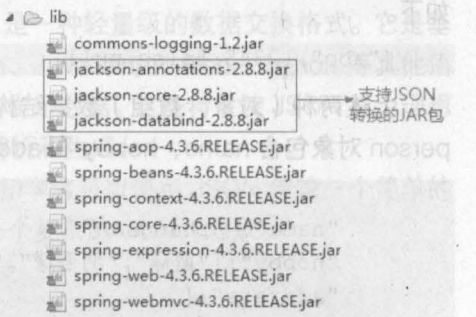


图14-3　项目相关JAR包

（2）在 web.xml 中，对 Spring MVC 的前端控制器等信息进行配置，如文件 14-1 所示。

文件 14-1　web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
         http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         id="WebApp_ID" version="3.1">
  <display-name>chapter14</display-name>
  <welcome-file-list>
      <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
  <!-- 配置 Spring MVC 前端控制器 DispatcherServlet -->
  <servlet>
      <servlet-name>springmvc</servlet-name>
      <servlet-class>
          org.springframework.web.servlet.DispatcherServlet
      </servlet-class>
      <!-- 配置 Spring MVC 加载配置文件路径 -->
      <init-param>
        <param-name>contextConfigLocation</param-name>
```

```
        <param-value>classpath:springmvc-config.xml</param-value>
      </init-param>
      <!-- 配置服务器启动后立即加载 Spring MVC 配置文件 -->
      <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
      <servlet-name>springmvc</servlet-name>
      <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

(3) 在 src 目录下，创建 Spring MVC 的核心配置文件 springmvc-config.xml，编辑后的代码如文件 14-2 所示。

文件 14-2 springmvc-config.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:mvc="http://www.springframework.org/schema/mvc"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
  http://www.springframework.org/schema/mvc
  http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context-4.3.xsd">
    <!-- 定义组件扫描器，指定需要扫描的包 -->
    <context:component-scan base-package="com.itheima.controller" />
   <!-- 配置注解驱动 -->
   <mvc:annotation-driven />
    <!--配置静态资源的访问映射，此配置中的文件，将不被前端控制器拦截 -->
    <mvc:resources location="/js/" mapping="/js/**" />
    <!-- 配置视图解析器 -->
    <bean class=
    "org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

在文件 14-2 中，不仅配置了组件扫描器和视图解析器，还配置了 Spring MVC 的注解驱动<mvc:annotation-drivern />和静态资源访问映射<mvc:resources .../>。其中<mvc:annotation-drivern/>配置会自动注册 RequestMappingHandlerMapping 和 RequestMappingHandlerAdapter 两个 Bean，并提供对读写 XML 和读写 JSON 等功能的支持。<mvc:resources... />元素用于配置静态资源的访问路径。由于在 web.xml 中配置的“/”会将页面中引入的静态文件也进行拦截，而拦截后页面中将找不到这些静态资源文件，这样就会引起页面报错。而增加了静态资源的访问映射配置后，程序会自动地去配置路径下找静态的内容。

<mvc:resources .../>中有两个重要属性 location 和 mapping，关于这两个属性的说明如

表 14-2 所示。

表 14-2 <mvc:resources>标签属性及说明

属性	说明
location	用于定位需要访问的本地静态资源文件路径，具体到某个文件夹
mapping	匹配静态资源全路径，其中“/**”表示文件夹及其子文件夹下的某个具体文件

（4）在 src 目录下，创建一个 com.itheima.po 包，并在包中创建一个 User 类，该类用于封装 User 类型的请求参数，编辑后如文件 14-3 所示。

文件 14-3 User.java

```
1  package com.itheima.po;
2  /**
3   * 用户 POJO
4   */
5  public class User {
6      private String username;
7      private String password;
8      public String getUsername() {
9          return username;
10     }
11     public void setUsername(String username) {
12         this.username = username;
13     }
14     public String getPassword() {
15         return password;
16     }
17     public void setPassword(String password) {
18         this.password = password;
19     }
20     @Override
21     public String toString() {
22      return "User [username=" + username + ", password=" + password + "]";
23     }
24 }
```

在文件 14-3 中，定义了 username 和 password 属性，及其对应的 getter/setter 方法，同时为了方便查询结果重写了 toString()方法。

（5）在 WebContent 目录下，创建页面文件 index.jsp 来测试 JSON 数据交互，编辑后如文件 14-4 所示。

文件 14-4 index.jsp

```
1  <%@ page language="java" contentType="text/html; charset=UTF-8"
2      pageEncoding="UTF-8"%>
3  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
4  "http://www.w3.org/TR/html4/loose.dtd">
5  <html>
6  <head>
7  <title>测试 JSON 交互</title>
```

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script type="text/javascript"
    src="${pageContext.request.contextPath }/js/jquery-1.11.3.min.js">
</script>
<script type="text/javascript">
function testJson(){
   // 获取输入的用户名和密码
    var username = $("#username").val();
    var password = $("#password").val();
    $.ajax({
        url : "${pageContext.request.contextPath }/testJson",
        type : "post",
        // data 表示发送的数据
        data : JSON.stringify({username:username,password:password}),
        // 定义发送请求的数据格式为 JSON 字符串
        contentType : "application/json;charset=UTF-8",
        //定义回调响应的数据格式为 JSON 字符串,该属性可以省略
        dataType : "json",
        //成功响应的结果
        success : function(data){
            if(data != null){
              alert("您输入的用户名为: "+data.username+
                          "密码为: "+data.password);
            }
        }
    });
}
</script>
</head>
<body>
   <form>
      用户名: <input type="text" name="username" id="username"><br />
      密   码:
        <input type="password" name="password" id="password"><br />
        <input type="button" value="测试 JSON 交互" onclick=" testJson()" />
   </form>
</body>
</html>
```

在文件 14-4 中，编写了一个测试 JSON 交互的表单，当单击“测试 JSON 交互”按钮时，会执行页面中的 testJson()函数。在函数中使用了 jQuery 的 AJAX 方式将 JSON 格式的用户名和密码传递到以“/testJson”结尾的请求中。

需要注意的是，在 AJAX 中包含了 3 个特别重要的属性，其说明如下。

- data：即请求时携带的数据，当使用 JSON 格式时，要注意编写规范。
- contentType：当请求数据为 JSON 格式时，值必须为 application/json。
- dataType：当响应数据为 JSON 时，可以定义 dataType 属性，并且值必须为 json。其中 dataType : "json"也可以省略不写，页面会自动识别响应的数据格式。

小提示

在上述测试页面 index.jsp 中使用的是 jQuery 的 AJAX 进行的 JSON 数据提交和响应，所以还需要引入 jquery.js 文件。本示例是引入了 WebContent 目录下 js 文件夹中的 jquery-1.11.3.min.js，读者可在所提供的源码中找到此文件。

（6）在 src 目录下，创建一个 com.itheima.controller 包，在该包下创建一个用于用户操作的控制器类 UserController，编辑后的代码如文件 14-5 所示。

文件 14-5　UserController.java

```
1  package com.itheima.controller;
2  import org.springframework.stereotype.Controller;
3  import org.springframework.web.bind.annotation.RequestBody;
4  import org.springframework.web.bind.annotation.RequestMapping;
5  import org.springframework.web.bind.annotation.ResponseBody;
6  import com.itheima.po.User;
7  @Controller
8  public class UserController {
9      /**
10      * 接收页面请求的 JSON 数据，并返回 JSON 格式结果
11      */
12     @RequestMapping("/testJson")
13     @ResponseBody
14     public User testJson(@RequestBody User user) {
15         // 打印接收的 JSON 格式数据
16         System.out.println(user);
17         // 返回 JSON 格式的响应
18         return user;
19     }
20 }
```

在文件 14-5 中，使用注解方式定义了一个控制器类，并编写了接收和响应 JSON 格式数据的 testJson()方法，在方法中接收并打印了接收到的 JSON 格式的用户数据，然后返回了 JSON 格式的用户对象。

方法中的@RequestBody 注解用于将前端请求体中的 JSON 格式数据绑定到形参 user 上，@ResponseBody 注解用于直接返回 User 对象（当返回 POJO 对象时，会默认转换为 JSON 格式数据进行响应）。

（7）将 chapter14 项目发布到 Tomcat 服务器并启动，在浏览器中访问地址 http://localhost:8080/chapter14/index.jsp，其显示效果如图 14-4 所示。

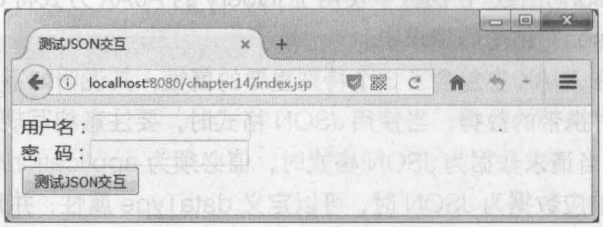


图14-4　index.jsp测试页面

在两个输入框中分别输入用户名“jack”和密码“123456”后，单击“测试 JSON 交互”按钮，当程序正确执行时，页面中会弹出显示用户名和密码的弹出框，如图 14-5 所示。

图14-5　index.jsp测试页面

与此同时，Eclipse 的控制台中也会显示相应数据，如图 14-6 所示。

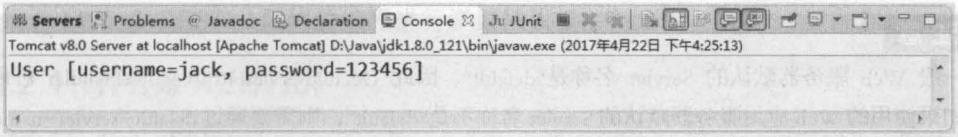


图14-6　运行结果

从图 14-5 和图 14-6 的显示结果可以看出，编写的代码已经正确实现了 JSON 数据交互，可以将 JSON 格式的请求数据转换为方法中的 Java 对象，也可以将 Java 对象转换为 JSON 格式的响应数据。

多学一招：

1. 使用<bean>标签方式的 JSON 转换器配置

在配置 JSON 转换器时，除了常用的<mvc:annotation-drivern />方式配置外，还可以使用<bean>标签的方式进行显示的配置。具体配置方式如下。

```
<!-- <bean>标签配置注解方式的处理器映射器和处理器适配器必须配对使用-->
<!-- 使用<bean>标签配置注解方式的处理器映射器 -->
<bean class="org.springframework.web.servlet.mvc.method
                        .annotation.RequestMappingHandlerMapping" />
<!-- 使用<bean>标签配置注解方式的处理器适配器 -->
<bean class="org.springframework.web.servlet.mvc.method
                        .annotation.RequestMappingHandlerAdapter">
   <property name="messageConverters">
      <list>
         <!-- 在注解适配器中配置 JSON 转换器 -->
         <bean  class="org.springframework.http.converter.json
                        .MappingJackson2HttpMessageConverter"/>
      </list>
   </property>
</bean>
```

从上述示例可以看出，使用<bean>标签配置方式配置 JSON 转换器时，需要同时配置处理

器映射器和处理器适配器，并且 JSON 转换器是配置在适配器中。

2. 配置静态资源访问的方式

除了使用<mvc:resources>元素可以实现对静态资源的访问外，还有另外两种静态资源访问的配置方式，具体分别如下。

（1）使用<mvc:default-servlet-handler>标签。

在 springmvc-config.xml 文件中，使用<mvc:default-servlet-handler>标签，具体如下。

```
<mvc:default-servlet-handler />
```

在 springmvc-config.xml 中配置<mvc:default-servlet-handler />后，会在 Spring MVC 上下文中定义一个 org.springframework.web.servlet.resource.DefaultServletHttpRequestHandler（即默认的 Servlet 请求处理器）。它会像一个检查员，对进入 DispatcherServlet 的 URL 进行筛查。如果发现是静态资源的请求，就将该请求转由 Web 服务器默认的 Servlet 处理，默认的 Servlet 就会对这些静态资源放行；如果不是静态资源的请求，才由 DispatcherServlet 继续处理。

注意

一般 Web 服务器默认的 Servlet 名称是"default"，因此 DefaultServletHttpRequestHandler 可以找到它。如果使用的 Web 应用服务器默认的 Servlet 名称不是"default"，则需要通过 default-servlet-name 属性显示指定，具体方式如下。

```
<mvc:default-servlet-handler  default-servlet-name="Servlet 名称" />
```

而 Web 服务器的 Servlet 名称是由使用的服务器确定的，常用服务器及其 Servlet 名称如下。

- Tomcat、Jetty、JBoss 和 and GlassFish 默认 Servlet 的名称—— "default"。
- Google App Engine 默认 Servlet 的名称——"_ah_default"。
- Resin 默认 Servlet 的名称——"resin-file"。
- WebLogic 默认 Servlet 的名称—— "FileServlet"。
- WebSphere 默认 Servlet 的名称——"SimpleFileServlet"。

（2）激活 Tomcat 默认的 Servlet 来处理静态文件访问。

激活 Tomcat 默认的 Servlet 时，需要在 web.xml 中添加以下内容。

```
<!--激活 tomcat 的静态资源拦截，需要哪些静态文件，再往下追加-->
<servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>*.js</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>*.css</url-pattern>
</servlet-mapping>
...
```

在上述代码中，配置了<servlet-mapping>元素来激活 Tomcat 默认的 Servlet 来处理静态文件，我们还可以根据需要继续追加<servlet-mapping>。此种配置方式和上一种方式本质上是一样的，都是使用 Web 服务器默认的 Servlet 来处理静态资源文件的访问。其中 Servelt 名称（即

<servlet-name>元素的值）也是由使用的服务器来确定的，不同的服务器需要使用不同的名称。

以上 3 种静态资源访问的配置方式不同，并且各有优缺点，具体如下。

- 第一和第三种配置方式可以选择性的释放静态资源。
- 第二种配置方式配置相对简单，只需要一行代码，就可以释放所有静态资源。
- 第二和第三种配置方式会导致项目移植性较差，需要根据具体的 Web 服务器来更改 Servlet 名称。
- 第三种配置方式运行效率更高，因为服务器启动时已经加载了 web.xml 中的静态资源。

在实际开发中，更为常用的配置还是第一种（即案例中的）配置方式，这样就不需要考虑 Web 服务器的问题了。

14.2 RESTful 支持

Spring MVC 除了支持 JSON 数据交互外，还支持 RESTful 风格的编程。接下来的两个小节就对 Spring MVC 中 RESTful 风格的编程进行详细的讲解。

14.2.1 什么是 RESTful

RESTful 也称之为 REST (Representational State Transfer)，可以将它理解为一种软件架构风格或设计风格，而不是一个标准。

简单来说，RESTful 风格就是把请求参数变成请求路径的一种风格。例如，传统的 URL 请求格式为：

```
http://.../queryItems?id=1
```

而采用 RESTful 风格后，其 URL 请求为：

```
http://.../items/1
```

从上述两个请求中可以看出，RESTful 风格中的 URL 将请求参数 id=1 变成了请求路径的一部分，并且 URL 中的 queryItems 也变成了 items (RESTful 风格中的 URL 不存在动词形式的路径，如 queryItems 表示查询订单，是一个动词，而 items 表示订单，为名词)。

RESTful 风格在 HTTP 请求中，使用 put、delete、post 和 get 方式分别对应添加、删除、修改和查询的操作。不过目前国内开发，还是只使用 post 和 get 方式来进行增删改查操作。

14.2.2 应用案例——用户信息查询

本案例将采用 RESTful 风格的请求实现对用户信息的查询，同时返回 JSON 格式的数据。其具体实现步骤如下。

(1) 在控制器类 UserController 中，编写用户查询方法 selectUser()，代码如下所示。

```
/**
 *接收 RESTful 风格的请求,其接收方式为 GET
 */
@RequestMapping(value="/user/{id}",method=RequestMethod.GET)
@ResponseBody
public User selectUser(@PathVariable("id") String id){
```

```
        //查看数据接收
        System.out.println("id="+id);
        User user=new User();
        //模拟根据 id 查询出到用户对象数据
        if(id.equals("1234")){
            user.setUsername("tom");
        }
        //返回 JSON 格式的数据
        return user;
    }
```

在上述代码中，@RequestMapping(value="/user/{id}",method=RequestMethod.GET)注解用于匹配请求路径（包括参数）和方式。其中 value="/user/{id}"表示可以匹配以“/user/{id}”结尾的请求，id 为请求中的动态参数；method=RequestMethod.GET 表示只接收 GET 方式的请求。方法中的@PathVariable("id")注解则用于接收并绑定请求参数，它可以将请求 URL 中的变量映射到方法的形参上，如果请求路径为“/user/{id}”，即请求参数中的 id 和方法形参名称 id 一样，则 @PathVariable 后面的“("id")”可以省略。

（2）在 WebContent 目录下，编写页面文件 restful.jsp，在页面中使用 AJAX 方式通过输入的用户编号来查询用户信息，如文件 14-6 所示。

文件 14-6　restful.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>RESTful 测试</title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script type="text/javascript"
   src="${pageContext.request.contextPath }/js/jquery-1.11.3.min.js">
</script>
<script type="text/javascript">
function search(){
    // 获取输入的查询编号
    var id = $("#number").val();
    $.ajax({
        url : "${pageContext.request.contextPath }/user/"+id,
        type : "GET",
        //定义回调响应的数据格式为 JSON 字符串，该属性可以省略
        dataType : "json",
        //成功响应的结果
        success : function(data){
            if(data.username != null){
                alert("您查询的用户是：  "+data.username);
            }else{
                alert("没有找到 id 为:"+id+"的用户！ ");
            }
```

```
28          }
29      });
30  }
31  </script>
32  </head>
33  <body>
34      <form>
35          编号: <input type="text" name="number" id="number">
36          <input type="button" value="搜索" onclick="search()" />
37      </form>
38  </body>
39  </html>
```

在文件 14-6 中，在请求路径中使用了 RESTful 风格的 URL，并且定义了请求方式为 GET。

（3）将项目发布到 Tomcat 服务器并启动，在浏览器中访问地址 http://localhost:8080/chapter14/restful.jsp，其显示效果如图 14-7 所示。

图14-7　restful.jsp测试页面

在输入框中输入编号“1234”后，单击“查询”按钮，程序正确执行后，浏览器会弹出用户信息窗口，如图 14-8 所示。

图14-8　restful.jsp测试页面

如果客户端使用的是火狐浏览器，使用 Firebug 查看请求地址，会发现请求的 URL 就是我们所需要的 RESTful 风格的路径，如图 14-9 所示。

与此同时，Eclipse 的控制台中，也打印出了请求的参数信息，如图 14-10 所示。

从图 14-9 和图 14-10 的显示结果可以看出，我们已经成功地使用 RESTful 风格的请求查询出了用户信息。

图14-9　restful.jsp测试页面

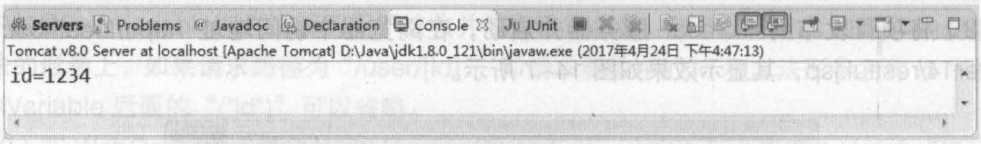


图14-10　运行结果

14.3 本章小结

本章主要对 Spring MVC 中的 JSON 数据交互和 RESTful 风格的请求进行了详细的讲解。首先简单介绍了 JSON 的概念、作用和结构，然后通过案例讲解了 Spring MVC 中如何实现 JSON 数据的交互。接下来讲解了什么是 RESTful，最后通过用户信息查询案例来演示 RESTful 的实际使用。通过本章的学习，读者可以掌握 Spring MVC 中的 JSON 数据交互和对 RESTful 风格支持，这对今后实际开发有极大的帮助。

【思考题】

1. 请简述 JSON 数据交互两个注解的作用。
2. 请简述静态资源访问的几种配置方式。

Java EE

15 Chapter

第15章 拦截器

学习目标

- 了解拦截器定义和配置方式
- 熟悉拦截器的执行流程
- 掌握拦截器的使用

在实际项目中，拦截器的使用是非常普遍的，例如在购物网站中通过拦截器可以拦截未登录的用户，禁止其购买商品，或者使用它来验证已登录用户是否有相应的操作权限等。在 Struts 2 框架中，拦截器是其重要的组成部分，而 Spring MVC 中也提供了拦截器功能，通过配置即可对请求进行拦截处理。本章将针对 Spring MVC 中拦截器的使用进行详细讲解。

15.1 拦截器概述

Spring MVC 中的拦截器（Interceptor）类似于 Servlet 中的过滤器（Filter），它主要用于拦截用户请求并做相应的处理。例如通过拦截器可以进行权限验证、记录请求信息的日志、判断用户是否登录等。

15.1.1 拦截器的定义

要使用 Spring MVC 中的拦截器，就需要对拦截器类进行定义和配置。通常拦截器类可以通过两种方式来定义。一种是通过实现 HandlerInterceptor 接口，或继承 HandlerInterceptor 接口的实现类（如 HandlerInterceptorAdapter）来定义；另一种是通过实现 WebRequestInterceptor 接口，或继承 WebRequestInterceptor 接口的实现类来定义。

以实现 HandlerInterceptor 接口的定义方式为例，自定义拦截器类的代码如下所示。

```
public class CustomInterceptor implements HandlerInterceptor{
    @Override
    public boolean preHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler)throws Exception {
        return false;
    }
    @Override
    public void postHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler,
        ModelAndView modelAndView) throws Exception {
    }
    @Override
    public void afterCompletion(HttpServletRequest request,
            HttpServletResponse response, Object handler,
            Exception ex) throws Exception {
    }
}
```

从上述代码可以看出，自定义的拦截器类实现了 HandlerInterceptor 接口，并实现了接口中的三个方法。关于这三个方法的具体描述如下。

- preHandler()方法：该方法会在控制器方法前执行，其返回值表示是否中断后续操作。当其返回值为 true 时，表示继续向下执行；当其返回值为 false 时，会中断后续的所有操作（包括调用下一个拦截器和控制器类中的方法执行等）。
- postHandle()方法：该方法会在控制器方法调用之后，且解析视图之前执行。可以通过此方法对请求域中的模型和视图做出进一步的修改。
- afterCompletion()方法：该方法会在整个请求完成，即视图渲染结束之后执行。可以通过

此方法实现一些资源清理、记录日志信息等工作。

15.1.2 拦截器的配置

要使自定义的拦截器类生效，还需要在 Spring MVC 的配置文件中进行配置，配置代码如下所示。

```
<!-- 配置拦截器 -->
<mvc:interceptors>
   <!--使用 bean 直接定义在<mvc:interceptors>下面的 Interceptor 将拦截所有请求-->
   <bean class="com.itheima.interceptor.CustomInterceptor"/>
   <!-- 拦截器 1 -->
   <mvc:interceptor>
      <!-- 配置拦截器作用的路径 -->
      <mvc:mapping path="/**"/>
      <!-- 配置不需要拦截器作用的路径 -->
      <mvc:exclude-mapping path=""/>
      <!-- 定义在<mvc:interceptor>下面的，表示对匹配路径的请求才进行拦截-->
      <bean class="com.itheima.interceptor.Interceptor1" />
   </mvc:interceptor>
   <!-- 拦截器 2 -->
   <mvc:interceptor>
      <mvc:mapping path="/hello"/>
      <bean class="com.itheima.interceptor.Interceptor2" />
   </mvc:interceptor>
   ...
</mvc:interceptors>
```

在上述代码中，<mvc:interceptors>元素用于配置一组拦截器，其子元素<bean>中定义的是全局拦截器，它会拦截所有的请求；而<mvc:interceptor>元素中定义的是指定路径的拦截器，它会对指定路径下的请求生效。<mvc:interceptor>元素的子元素<mvc:mapping>用于配置拦截器作用的路径，该路径在其属性 path 中定义。如上述代码中 path 的属性值“/**”表示拦截所有路径，“/hello”表示拦截所有以“/hello”结尾的路径。如果在请求路径中包含不需要拦截的内容，还可以通过<mvc:exclude-mapping>元素进行配置。

需要注意的是，<mvc:interceptor>中的子元素必须按照上述代码的配置顺序进行编写，即<mvc:mapping ... />→<mvc:exclude-mapping ... />→<bean ... />的顺序，否则文件会报错。

15.2 拦截器的执行流程

15.2.1 单个拦截器的执行流程

在运行程序时，拦截器的执行是有一定顺序的，该顺序与配置文件中所定义的拦截器的顺序相关。如果在项目中只定义了一个拦截器，那么该拦截器在程序中的执行流程如图 15-1 所示。

从图 15-1 可以看出，程序首先会执行拦截器类中的 preHandle()方法，如果该方法的返回

值为 true，则程序会继续向下执行处理器中的方法，否则将不再向下执行；在业务处理器（即控制器 Controller 类）处理完请求后，会执行 postHandle()方法，然后会通过 DispatcherServlet 向客户端返回响应；在 DispatcherServlet 处理完请求后，才会执行 afterCompletion()方法。

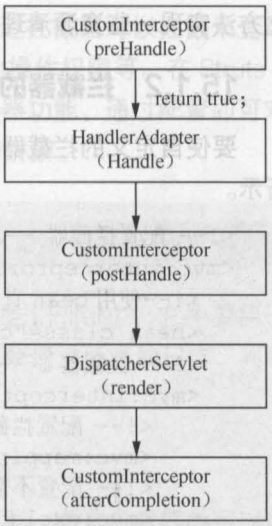


图15-1　单个拦截器的执行流程

为了验证上面所讲解的拦截器执行流程，下面通过一个案例来演示其使用，具体步骤如下。

（1）在 Eclipse 中，创建一个名为 chapter15 的 Web 项目，将 Spring MVC 程序运行所需 JAR 包复制到项目的 lib 目录中，并发布到类路径下。

（2）在 web.xml 中，配置 Spring MVC 的前端过滤器和初始化加载配置文件等信息。

（3）在 src 目录下，创建一个 com.itheima.controller 包，并在包中创建控制器类 HelloController，编辑后的代码如文件 15-1 所示。

文件 15-1　HelloController.java

```
1  package com.itheima.controller;
2  import org.springframework.stereotype.Controller;
3  import org.springframework.web.bind.annotation.RequestMapping;
4  @Controller
5  public class HelloController {
6      /**
7       * 页面跳转
8       */
9      @RequestMapping("/hello")
10     public String Hello() {
11         System.out.println("Hello!");
12         return "success";
13     }
14 }
```

（4）在 src 目录下，创建一个 com.itheima.interceptor 包，并在包中创建拦截器类 CustomInterceptor。该类需要实现 HandlerInterceptor 接口，并且在实现方法中需要编写输出语句来输出信息，如文件 15-2 所示。

文件 15-2　CustomInterceptor.java

```
1  package com.itheima.interceptor;
2  import javax.servlet.http.HttpServletRequest;
3  import javax.servlet.http.HttpServletResponse;
4  import org.springframework.web.servlet.HandlerInterceptor;
5  import org.springframework.web.servlet.ModelAndView;
6  /**
7   * 实现了 HandlerInterceptor 接口的自定义拦截器类
8   */
9  public class CustomInterceptor implements HandlerInterceptor{
10     @Override
```

```
    public boolean preHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler)throws Exception {
        System.out.println("CustomInterceptor...preHandle");
        //对拦截的请求进行放行处理
        return true;
    }
    @Override
    public void postHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler,
        ModelAndView modelAndView) throws Exception {
        System.out.println("CustomInterceptor...postHandle");
    }
    @Override
    public void afterCompletion(HttpServletRequest request,
        HttpServletResponse response, Object handler,
        Exception ex) throws Exception {
        System.out.println("CustomInterceptor...afterCompletion");
    }
}
```

（5）在 src 目录下，创建并配置 Spring MVC 的配置文件，如文件 15-3 所示。

文件 15-3 springmvc-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:mvc="http://www.springframework.org/schema/mvc"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
  http://www.springframework.org/schema/mvc
  http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context-4.3.xsd">
    <!-- 定义组件扫描器，指定需要扫描的包 -->
    <context:component-scan base-package="com.itheima.controller" />
    <!-- 定义视图解析器 -->
    <bean id="viewResolver" class=
    "org.springframework.web.servlet.view.InternalResourceViewResolver">
        <!-- 设置前缀 -->
        <property name="prefix" value="/WEB-INF/jsp/" />
        <!-- 设置后缀 -->
        <property name="suffix" value=".jsp" />
    </bean>
    <!-- 配置拦截器 -->
    <mvc:interceptors>
      <!--使用 bean 直接定义在<mvc:interceptors>下面的拦截器将拦截所有请求-->
      <bean class="com.itheima.interceptor.CustomInterceptor"/>
    </mvc:interceptors>
</beans>
```

由于配置拦截器使用的是<mvc:interceptors>元素，所以需要配置 mvc 的 schema 信息。本案例演示的是单个拦截器的执行顺序，所以这里只配置了一个全局的拦截器。

（6）在 WEB-INF 目录下，创建一个 jsp 文件夹，并在该文件夹中创建一个页面文件 success.jsp，然后在页面文件的<body>元素内编写任意显示信息，如“ok”。

（7）将项目发布到 Tomcat 服务器并启动，在浏览器中访问地址 http://localhost:8080/chapter15/hello，程序正确执行后，浏览器会跳转到 success.jsp 页面，此时控制台的输出结果如图 15-2 所示。

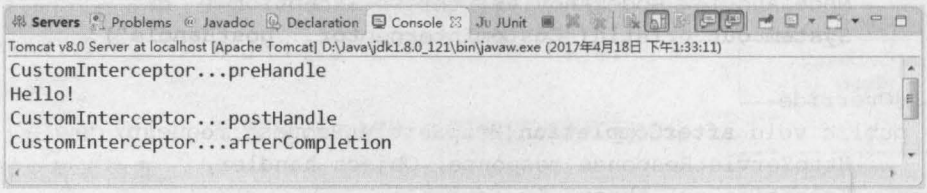


图15-2　运行结果

从图 15-2 可以看出，程序先执行了拦截器类中的 preHandle()方法，然后执行了控制器中的 Hello()方法，最后分别执行了拦截器类中的 postHandle()方法和 afterCompletion()方法。这与上文所描述的单个拦截器的执行顺序是一致的。

15.2.2　多个拦截器的执行流程

在大型的企业级项目中，通常不会只有一个拦截器，开发人员可能会定义很多拦截器来实现不同的功能。那么多个拦截器的执行顺序又是怎样的呢？下面通过一张图来描述多个拦截器的执行流程（假设有两个拦截器 Interceptor1 和 Interceptor2，并且在配置文件中，Interceptor1 拦截器配置在前），如图 15-3 所示。

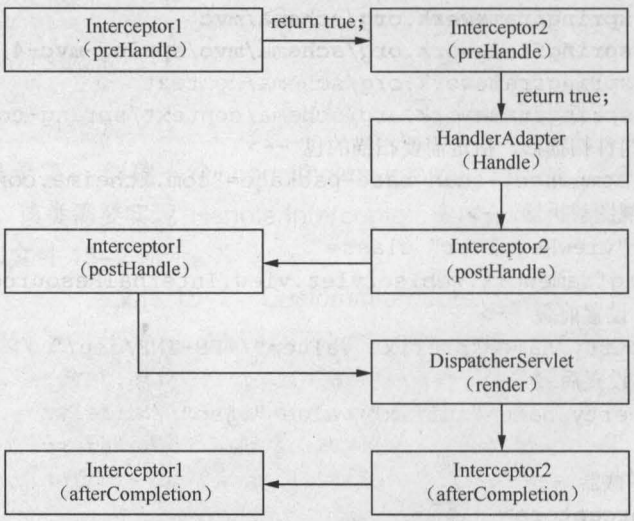


图15-3　多个拦截器的执行流程

从图 15-3 可以看出，当有多个拦截器同时工作时，它们的 preHandle()方法会按照配置文件中拦截器的配置顺序执行，而它们的 postHandle()方法和 afterCompletion()方法则会按照配置

顺序的反序执行。

为了验证上述描述，下面通过修改 15.2.1 小节的案例来演示多个拦截器的执行，具体步骤如下。

（1）在 com.itheima.interceptor 包中，创建两个拦截器类 Interceptor1 和 Interceptor2，这两个拦截器类均实现了 HandlerInterceptor 接口，并重写其中的方法，如文件 15-4 和文件 15-5 所示。

文件 15-4 Interceptor1.java

```
1  package com.itheima.interceptor;
2  import javax.servlet.http.HttpServletRequest;
3  import javax.servlet.http.HttpServletResponse;
4  import org.springframework.web.servlet.HandlerInterceptor;
5  import org.springframework.web.servlet.ModelAndView;
6  /**
7   * 以实现接口的方式定义拦截器
8   */
9  public class Interceptor1 implements HandlerInterceptor {
10     @Override
11     public boolean preHandle(HttpServletRequest request,
12         HttpServletResponse response, Object handler) throws Exception {
13         System.out.println("Intercepter1...preHandle");
14         return true;
15     }
16     @Override
17     public void postHandle(HttpServletRequest request,
18         HttpServletResponse response, Object handler,
19         ModelAndView modelAndView) throws Exception {
20         System.out.println("Intercepter1...postHandle");
21     }
22     @Override
23     public void afterCompletion(HttpServletRequest request,
24         HttpServletResponse response, Object handler,
25         Exception ex) throws Exception {
26         System.out.println("Intercepter1...afterCompletion");
27     }
28 }
```

文件 15-5 Interceptor2.java

```
1  package com.itheima.interceptor;
2  import javax.servlet.http.HttpServletRequest;
3  import javax.servlet.http.HttpServletResponse;
4  import org.springframework.web.servlet.HandlerInterceptor;
5  import org.springframework.web.servlet.ModelAndView;
6  /**
7   * 以实现接口的方式定义拦截器
8   */
9  public class Interceptor2 implements HandlerInterceptor{
10     @Override
```

```
    public boolean preHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler) throws Exception {
        System.out.println("Interceptor2...preHandle");
        return true;
    }
    @Override
    public void postHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler,
        ModelAndView modelAndView) throws Exception {
        System.out.println("intercepter2...postHandle");
    }
    @Override
    public void afterCompletion(HttpServletRequest request,
        HttpServletResponse response, Object handler, Exception ex)
        throws Exception {
        System.out.println("Intercepter2...afterCompletion");
    }
}
```

（2）在配置文件 springmvc-config.xml 中的<mvc:interceptors>元素内配置上面所定义的两个拦截器，配置代码如下所示。

```
<!-- 拦截器 1 -->
<mvc:interceptor>
    <!-- 配置拦截器作用的路径 -->
    <mvc:mapping path="/**" />
    <!-- 定义在<mvc:interceptor>下面的表示匹配指定路径的请求才进行拦截的 -->
    <bean class="com.itheima.interceptor.Interceptor1" />
</mvc:interceptor>
<!-- 拦截器 2 -->
<mvc:interceptor>
    <mvc:mapping path="/hello" />
    <bean class="com.itheima.interceptor.Interceptor2" />
</mvc:interceptor>
```

在上述拦截器的配置代码中，第一个拦截器会作用于所有路径下的请求，而第二个拦截器会作用于以“/hello”结尾的请求。

小提示

为了不影响程序的输出结果，可将上一小节案例中所配置的 CustomInterceptor 的拦截器配置注释掉。

（3）发布项目到 Tomcat 服务器并启动，在浏览器中访问地址 http://localhost:8080/chapter15/hello，控制台中输出的信息如图 15-4 所示。

从图 15-4 可以看出，程序先执行了前两个拦截器类中的 preHandle()方法，这两个方法的执行顺序与配置文件中定义的顺序相同；然后执行了控制器类中的 Hello()方法；最后执行了两个拦截器类中的 postHandle()方法和 afterCompletion()方法，且这两个方法的执行顺序与配置文件中所定义的拦截器顺序相反。

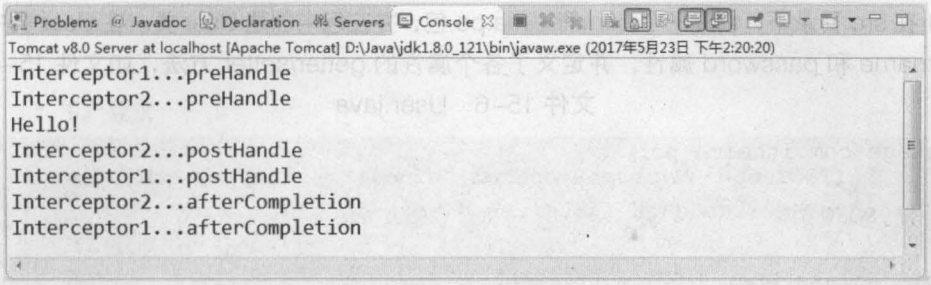


图15-4　运行结果

15.3 应用案例——实现用户登录权限验证

本小节将通过拦截器来完成一个用户登录权限验证的案例。本案例中，只有登录后的用户才能访问系统中的主页面，如果没有登录系统而直接访问主页面，则拦截器会将请求拦截，并转发到登录页面，同时在登录页面中给出提示信息。如果用户名或密码错误，也会在登录页面给出相应的提示信息。当已登录的用户在系统主页中单击“退出”链接时，系统同样会回到登录页面。该案例的整个执行流程如图 15-5 所示。

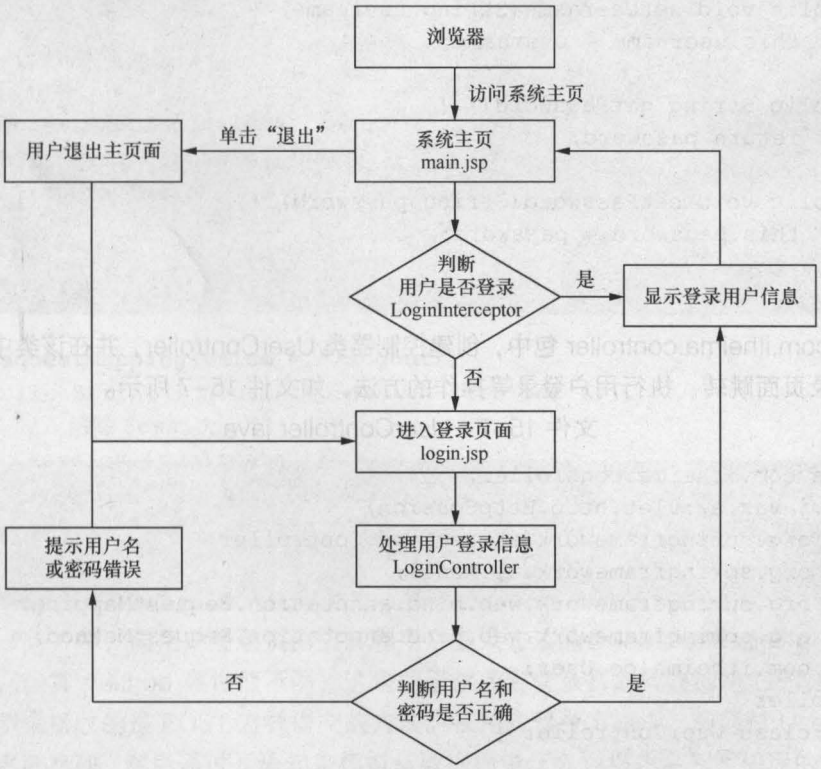


图15-5　用户权限验证的执行流程图

了解了案例的整个执行流程后，接下来讲解如何在项目中实现用户登录权限验证，具体步骤如下。

（1）在 src 目录下，创建一个 com.itheima.po 包，并在包中创建 User 类。在 User 类中，声明 id、username 和 password 属性，并定义了各个属性的 getter/setter 方法，如文件 15-6 所示。

文件 15-6　User.java

```
1  package com.itheima.po;
2  /**
3   * 用户 POJO 类
4   */
5  public class User {
6      private Integer id;       //id
7      private String username;  //用户名
8      private String password;  //密码
9      public Integer getId() {
10         return id;
11     }
12     public void setId(Integer id) {
13         this.id = id;
14     }
15     public String getUsername() {
16         return username;
17     }
18     public void setUsername(String username) {
19         this.username = username;
20     }
21     public String getPassword() {
22         return password;
23     }
24     public void setPassword(String password) {
25         this.password = password;
26     }
27 }
```

（2）在 com.itheima.controller 包中，创建控制器类 UserController，并在该类中定义向主页跳转、向登录页面跳转、执行用户登录等操作的方法，如文件 15-7 所示。

文件 15-7　UserController.java

```
1  package com.itheima.controller;
2  import javax.servlet.http.HttpSession;
3  import org.springframework.stereotype.Controller;
4  import org.springframework.ui.Model;
5  import org.springframework.web.bind.annotation.RequestMapping;
6  import org.springframework.web.bind.annotation.RequestMethod;
7  import com.itheima.po.User;
8  @Controller
9  public class UserController {
10     /**
11      * 向用户登录页面跳转
12      */
13     @RequestMapping(value="/login",method=RequestMethod.GET)
14     public String toLogin() {
```

```
15          return "login";
16      }
17      /**
18       * 用户登录
19       */
20      @RequestMapping(value="/login",method=RequestMethod.POST)
21      public String login(User user,Model model,HttpSession session) {
22          // 获取用户名和密码
23          String username = user.getUsername();
24          String password = user.getPassword();
25          // 此处模拟从数据库中获取用户名和密码后进行判断
26          if(username != null && username.equals("xiaoxue")
27                  && password != null && password.equals("123456")){
28              // 将用户对象添加到 Session
29              session.setAttribute("USER_SESSION", user);
30              // 重定向到主页面的跳转方法
31              return "redirect:main";
32          }
33          model.addAttribute("msg", "用户名或密码错误，请重新登录！");
34          return "login";
35      }
36      /**
37       * 向用户主页面跳转
38       */
39      @RequestMapping(value="/main")
40      public String toMain() {
41          return "main";
42      }
43      /**
44       * 退出登录
45       */
46      @RequestMapping(value = "/logout")
47      public String logout(HttpSession session) {
48          // 清除 Session
49          session.invalidate();
50          // 重定向到登录页面的跳转方法
51          return "redirect:login";
52      }
53  }
```

在文件 15-7 中，向用户登录页面跳转和用户登录方法的@RequestMapping 注解的 value 属性值相同，但其 method 属性值不同，这是由于跳转到登录页面接收的是 GET 方式提交的方法，而用户登录接收的是 POST 方式提交的方法。在用户登录方法中，先通过 User 类型的参数获取了用户名和密码，然后通过 if 语句来模拟从数据库中获取到用户名和密码后的判断。如果存在此用户，就将用户信息保存到 Session 中，并重定向到主页，否则跳转到登录页面。

（3）在 com.itheima.interceptor 包中，创建拦截器类 LoginInterceptor，编辑后的代码如文件 15-8 所示。

文件 15-8　LoginInterceptor.java

```
1  package com.itheima.interceptor;
2  import javax.servlet.http.HttpServletRequest;
3  import javax.servlet.http.HttpServletResponse;
4  import javax.servlet.http.HttpSession;
5  import org.springframework.web.servlet.HandlerInterceptor;
6  import org.springframework.web.servlet.ModelAndView;
7  import com.itheima.po.User;
8  /**
9   * 登录拦截器
10  */
11 public class LoginInterceptor implements HandlerInterceptor{
12     @Override
13     public boolean preHandle(HttpServletRequest request,
14         HttpServletResponse response, Object handler) throws Exception {
15         // 获取请求的 URL
16         String url = request.getRequestURI();
17         // URL:除了 login.jsp 是可以公开访问的，其他的 URL 都进行拦截控制
18         if(url.indexOf("/login")>=0){
19             return true;
20         }
21         // 获取 Session
22         HttpSession session = request.getSession();
23         User user = (User) session.getAttribute("USER_SESSION");
24         // 判断 Session 中是否有用户数据，如果有，则返回 true,继续向下执行
25         if(user != null){
26             return true;
27         }
28          // 不符合条件的给出提示信息，并转发到登录页面
29          request.setAttribute("msg", "您还没有登录，请先登录！");
30          request.getRequestDispatcher("/WEB-INF/jsp/login.jsp")
31                                                 .forward(request, response);
32          return false;
33     }
34     @Override
35     public void postHandle(HttpServletRequest request,
36             HttpServletResponse response, Object handler,
37             ModelAndView modelAndView) throws Exception {
38     }
39     @Override
40     public void afterCompletion(HttpServletRequest request,
41             HttpServletResponse response, Object handler, Exception ex)
42             throws Exception {
43     }
44 }
```

在文件 15-8 的 preHandle()方法中，先获取了请求的 URL，然后通过 indexOf()方法判断 URL 中是否有“/login”字符串。如果有，则返回 true，即直接放行；如果没有，则继续向下执行拦截处理。接下来获取了 Session 中的用户信息，如果 Session 中包含用户信息，即表示用户

已登录，也直接放行；否则会转发到登录页面，不再执行后续程序。

（4）在配置文件的<mvc:interceptors>元素中，配置自定义的登录拦截器信息，代码如下所示。

```
<mvc:interceptor>
    <mvc:mapping path="/**" />
    <bean class="com.itheima.interceptor.LoginInterceptor" />
</mvc:interceptor>
```

（5）在 WEB-INF 目录下的 jsp 文件夹中，创建一个系统主页面 main.jsp。在该页面中，使用 EL 表达式获取用户信息，并且通过一个超链接来实现“退出”功能，如文件 15-9 所示。

文件 15-9　main.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>系统主页</title>
</head>
<body>
    当前用户：${USER_SESSION.username}
    <a href="${pageContext.request.contextPath }/logout">退出</a>
</body>
</html>
```

（6）在 WEB-INF 目录下的 jsp 文件夹中，创建一个登录页面 login.jsp，在页面中编写一个用于实现登录操作的 form 表单，如文件 15-10 所示。

文件 15-10　login.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>用户登录</title>
</head>
<body>
    ${msg}
    <form action="${pageContext.request.contextPath }/login"
          method="POST">
        用户名：<input type="text" name="username"/><br />
        密   码：
              <input type="password" name="password"/><br />
        <input type="submit" value="登录" />
    </form>
</body>
</html>
```

（7）将项目发布到 Tomcat 服务器并启动，在浏览器中访问地址 http://localhost:8080/

chapter15/main，其显示效果如图 15-6 所示。

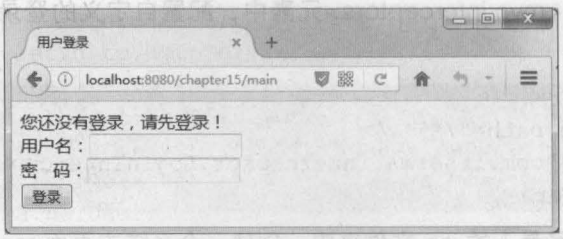


图15-6 login.jsp登录页面

从图 15-6 可以看出，当用户未登录而直接访问主页面时，访问请求会被登录拦截器拦截，从而跳转到登录页面，并提示用户未登录信息。如果在用户名输入框中输入“jack”，密码框中输入“123456”，当单击“登录”按钮后，浏览器的显示结果如图 15-7 所示。

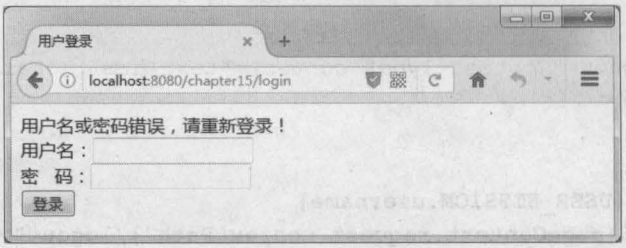


图15-7 login.jsp登录页面

当输入正确的用户名“xiaoxue”和密码“123456”，并单击“登录”按钮后，浏览器会跳转到系统主页面，如图 15-8 所示。

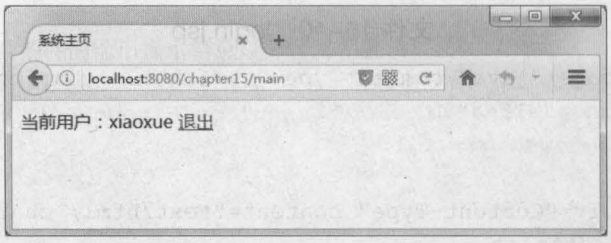


图15-8 main.jsp系统主页面

当单击图 15-8 中的“退出”链接后，用户即可退出当前系统，系统会从主页面重定向到登录页面。

15.4 本章小结

本章主要对 Spring MVC 中的拦截器使用进行了详细讲解。首先介绍了如何在 Spring MVC 项目中定义和配置拦截器，然后详细讲解了单个拦截器和多个拦截器的执行流程，最后通过一个用户登录权限验证的应用案例演示了拦截器的实际应用。通过本章的学习，读者可以对 Spring MVC 中拦截器的定义和配置方式有一定的了解，能够熟悉拦截器的执行流程，并能够掌握拦截器的使用。

【思考题】

1. 请简述 Spring MVC 拦截器的定义方式。
2. 请简述单个拦截器和多个拦截器的执行流程。

Java EE

Chapter 16

第 16 章
文件上传和下载

学习目标

- 熟悉 Spring MVC 中文件上传的实现步骤
- 掌握文件上传案例的编写
- 掌握中英文名称文件下载程序的编写

文件的上传和下载是项目开发中最常用的功能，例如图片的上传与下载、邮件附件的上传与下载等。接下来，本章将对 Spring MVC 环境中文件的上传和下载进行详细的讲解。

16.1 文件上传

16.1.1 文件上传概述

多数文件上传都是通过表单形式提交给后台服务器的，因此，要实现文件上传功能，就需要提供一个文件上传的表单，而该表单必须满足以下 3 个条件。

- form 表单的 method 属性设置为 post。
- form 表单的 enctype 属性设置为 multipart/form-data。
- 提供<input type="file" name="filename" />的文件上传输入框。

文件上传表单的示例代码如下。

```
<form action="uploadUrl" method="post" enctype="multipart/form-data">
    <input type="file" name="filename" multiple="multiple" />
    <input type="submit" value="文件上传" />
</form>
```

上述代码中，除了满足上传表单所必须的 3 个条件外，在<input>元素中还增加了一个 multiple 属性。该属性是 HTML5 中的新属性，如果使用了该属性，则可以同时选择多个文件进行上传，即可实现多文件上传。

当客户端 form 表单的 enctype 属性为 multipart/form-data 时，浏览器就会采用二进制流的方式来处理表单数据，服务器端就会对文件上传的请求进行解析处理。Spring MVC 为文件上传提供了直接的支持，这种支持是通过 MultipartResolver（多部件解析器）对象实现的。MultipartResolver 是一个接口对象，需要通过它的实现类 CommonsMultipartResolver 来完成文件上传工作。在 Spring MVC 中使用 MultipartResolver 对象非常简单，只需要在配置文件中定义 MultipartResolver 接口的 Bean 即可，其具体配置方式如下。

```
<bean id="multipartResolver" class=
  "org.springframework.web.multipart.commons.CommonsMultipartResolver">
  <!-- 设置请求编码格式，必须与 JSP 中的 pageEncoding 属性一致，默认为 ISO-8859-1 -->
  <property name="defaultEncoding" value="UTF-8" />
  <!-- 设置允许上传文件的最大值（2MB），单位为字节 -->
  <property name="maxUploadSize" value="2097152" />
  ...
</bean>
```

在上述配置代码中，除配置了 CommonsMultipartResolver 类外，还通过<property>元素配置了编码格式以及允许上传文件的大小。

通过<property>元素可以对文件解析器类 CommonsMultipartResolver 的如下属性进行配置。

- maxUploadSize：上传文件最大长度（以字节为单位）。
- maxInMemorySize：缓存中的最大尺寸。

- defaultEncoding：默认编码格式。
- resolveLazily：推迟文件解析，以便在 Controller 中捕获文件大小异常。

注意

因为 MultipartResolver 接口的实现类 CommonsMultipartResolver 内部是引用 multipartResolver 字符串获取该实现类对象并完成文件解析的，所以在配置 CommonsMultipartResolver 时必须指定该 Bean 的 id 为 multipartResolver。

由于 CommonsMultipartResolver 是 Spring MVC 内部通过 Apache Commons FileUpload 技术实现的，所以 Spring MVC 的文件上传还需要依赖 Apache Commons FileUpload 的组件，即需要导入支持文件上传的相关 JAR 包，具体如下。

- commons-fileupload-1.3.2.jar
- commons-io-2.5.jar

以上两个 JAR 包的版本是本书编写时的最新版本，读者可以通过 Apache 官网地址“http://commons.apache.org/”下载（进入该网址后，在 Apache Commons Proper 下方列表的 Components 列中找到 FileUplod 和 IO，单击链接后，即可在打开页面找到下载链接），也可以直接使用本书源码中的 JAR 包。

当完成页面表单和文件上传解析器的配置后，在 Controller 中编写文件上传的方法即可实现文件上传。在 Spring MVC 中，文件上传的方法编写十分简单，其代码如下所示。

```
@Controller
public class FileUploadController {
    @RequestMapping("/fileUpload ")
    public String handleFormUpload(@RequestParam("name") String name,
                    @RequestParam("filename") MultipartFile file,...) {
        if (!file.isEmpty()) {
            // 具体的执行方法
            ...
            return "uploadSuccess";
        }
        return "uploadFailure";
    }
}
```

在上述代码中，包含一个 MultipartFile 接口类型的参数 file，上传到程序中的文件就是被封装在该参数中的。org.springframework.web.multipart.MultipartFile 接口中提供了获取上传文件、文件名称等方法，这些方法及其说明如表 16-1 所示。

表 16-1　MultipartFile 接口中的主要方法

方法	说明
byte[] getBytes()	以字节数组的形式返回文件的内容
String getContentType()	返回文件的内容类型
InputStream getInputStream()	读取文件内容，返回一个 InputStream 流
String getName()	获取多部件 form 表单的参数名称

续表

方法	说明
String getOriginalFilename()	获取上传文件的初始化名
long getSize()	获取上传文件的大小，单位是字节
boolean isEmpty()	判断上传的文件是否为空
void transferTo(File file)	将上传文件保存到目标目录下

16.1.2　应用案例——文件上传

通过 16.1.1 节的学习，相信读者对 Spring MVC 中实现文件上传的步骤和配置已经有了一个大致的了解。接下来，本小节就通过一个具体的案例来演示文件上传功能的实现，其具体步骤如下。

（1）在 Eclipse 中创建一个名为 chapter16 的 Web 项目，将 Spring MVC 相关 JAR 包以及支持文件上传下载的 JAR 包添加到项目的 lib 目录中，并发布到类路径下。添加后的 lib 目录如图 16-1 所示。

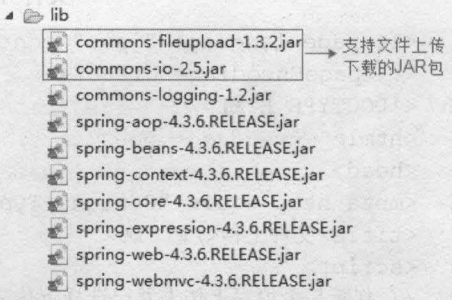


图16-1　Spring MVC环境下文件上传下载的JAR包

（2）在 web.xml 文件中，配置 Spring MVC 的前端控制器等信息。

（3）在 src 目录下，创建并编写 Spring MVC 的核心配置文件 springmvc-config.xml，如文件 16-1 所示。

文件 16-1　springmvc-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:mvc="http://www.springframework.org/schema/mvc"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
  http://www.springframework.org/schema/mvc
  http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context-4.3.xsd">
    <!-- 定义组件扫描器，指定需要扫描的包 -->
    <context:component-scan base-package="com.itheima.controller" />
    <!--配置注解驱动  -->
    <mvc:annotation-driven />
    <!-- 定义视图解析器 -->
    <bean id="viewResolver" class=
    "org.springframework.web.servlet.view.InternalResourceViewResolver">
        <!-- 设置前缀 -->
        <property name="prefix" value="/WEB-INF/jsp/" />
        <!-- 设置后缀 -->
```

```
        <property name="suffix" value=".jsp" />
    </bean>
    <!-- 配置文件上传解析器 MultipartResolver -->
    <bean id="multipartResolver" class=
   "org.springframework.web.multipart.commons.CommonsMultipartResolver">
        <!-- 设置请求编码格式-->
        <property name="defaultEncoding" value="UTF-8" />
    </bean>
</beans>
```

在文件 16-1 中，除配置了 Spring MVC 环境需要的组件扫描器、注解驱动和视图解析器外，还增加了支持文件上传的解析器 CommonsMultipartResolver 的配置。

（4）在 WebContent 目录下，创建一个用于上传文件的页面 fileUpload.jsp，编辑后如文件 16-2 所示。

文件 16-2　fileUpload.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE HTML>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>文件上传</title>
<script>
// 判断是否填写上传人并已选择上传文件
function check(){
    var name = document.getElementById("name").value;
    var file = document.getElementById("file").value;
    if(name==""){
        alert("填写上传人");
        return false;
    }
    if(file.length==0||file==""){
        alert("请选择上传文件");
        return false;
    }
    return true;
}
</script>
</head>
<body>
    <form action="${pageContext.request.contextPath }/fileUpload"
    method="post" enctype="multipart/form-data" onsubmit="return check()">
     上传人：<input id="name" type="text" name="name" /><br />
     请选择文件：<input id="file" type="file" name="uploadfile"
                                        multiple="multiple" /><br />
                <input type="submit" value="上传" />
     </form>
</body>
</html>
```

在文件 16-2 中，编写了一个用于文件上传的 form 表单，该表单可以填写上传人并上传文件。当单击“上传”按钮时，会先执行 check()方法来检查上传人文本框和文件选择框中的内容是否为空。只有填写了上传人并选择了需要上传的文件后，才能正常提交表单；否则表单将不会提交，并给出相应提示信息。提交表单后，会以 POST 方式提交到一个以“/fileUpload”结尾的请求中。

（5）在 WEB-INF 目录下，创建 jsp 文件夹，并在文件夹中创建 success.jsp 和 error.jsp 文件，分别在两个文件的<body>元素内编写显示上传成功的信息（如“文件上传成功!”）和显示上传失败的信息（如“文件上传失败，请重新上传!”）。

（6）在 src 目录下，创建一个 com.itheima.controller 包，在该包下创建一个用于文件上传的控制器类 FileUploadController，编辑后如文件 16-3 所示。

文件 16-3　FileUploadController.java

```
1  package com.itheima.controller;
2  import java.io.File;
3  import java.util.List;
4  import java.util.UUID;
5  import javax.servlet.http.HttpServletRequest;
6  import org.springframework.stereotype.Controller;
7  import org.springframework.web.bind.annotation.RequestMapping;
8  import org.springframework.web.bind.annotation.RequestParam;
9  import org.springframework.web.multipart.MultipartFile;
10 /**
11  * 文件上传
12  */
13 @Controller
14 public class FileUploadController {
15     /**
16      * 执行文件上传
17      */
18     @RequestMapping("/fileUpload")
19     public String handleFormUpload(@RequestParam("name") String name,
20             @RequestParam("uploadfile") List<MultipartFile> uploadfile,
21             HttpServletRequest request) {
22         // 判断所上传文件是否存在
23         if (!uploadfile.isEmpty() && uploadfile.size() > 0) {
24             //循环输出上传的文件
25             for (MultipartFile file : uploadfile) {
26                 // 获取上传文件的原始名称
27                 String originalFilename = file.getOriginalFilename();
28                 // 设置上传文件的保存地址目录
29                 String dirPath =
30                     request.getServletContext().getRealPath("/upload/");
31                 File filePath = new File(dirPath);
32                 // 如果保存文件的地址不存在，就先创建目录
33                 if (!filePath.exists()) {
34                     filePath.mkdirs();
35                 }
36                 // 使用 UUID 重新命名上传的文件名称(上传人_uuid_原始文件名称)
```

```
37                  String newFilename = name+ "_"+UUID.randomUUID() +
38                                                  "_"+originalFilename;
39                  try {
40                      // 使用MultipartFile接口的方法完成文件上传到指定位置
41                      file.transferTo(new File(dirPath + newFilename));
42                  } catch (Exception e) {
43                      e.printStackTrace();
44                      return"error";
45                  }
46              }
47              // 跳转到成功页面
48              return "success";
49          }else{
50              return"error";
51          }
52      }
53  }
```

在文件 16-3 中，使用注解方式定义了一个控制器类，并在类中定义了执行文件上传的方法 handleFormUpload()。在 handleFormUpload()方法参数中使用了 List<MultipartFile>集合类型来接收用户上传的文件，然后判断所上传的文件是否存在。如果存在，则继续执行上传操作，在通过 MultipartFile 接口的 transferTo()方法将上传文件保存到用户指定的目录位置后，会跳转到 success.jsp 页面；如果文件不存在或者上传失败，则跳转到 error.jsp 页面。

（7）将项目发布到 Tomcat 服务器中并启动，在浏览器中访问地址 http://localhost:8080/chapter16/fileUpload.jsp，其显示效果如图 16-2 所示。

图16-2　fileUpload.jsp文件上传页面

在图 16-2 的文件上传页面中，填写上传人并选择所要上传的文件，单击“上传”按钮后就可向后台发送上传请求信息。这里填写上传人为“小韩”，然后选择两张图片后，浏览器的显示效果如图 16-3 所示。

图16-3　fileUpload.jsp文件上传页面

单击“上传”按钮，程序在正确执行后浏览器就会跳转到 success.jsp 页面，此时查看项目的发

布目录，即可发现在 chapter16 项目中多了一个 upload 文件夹，该文件夹中的内容如图 16-4 所示。

图16-4 upload文件夹

从图 16-4 可以看出，已经成功上传了两张图片，并且图片的名称为“上传人_uuid_原始文件名称”的形式。

需要注意的是，upload 文件夹是在项目的发布路径中，而不是创建的项目所在目录。如果未更改项目发布路径，则要去工作空间的 metadata 目录中寻找项目发布目录（路径为：workspace\.metadata\.plugins\org.eclipse.wst.server.core\tmp1\wtpwebapps\chapter16\upload）；如果将项目的发布路径已更改到 Tomcat 中，则需要在 Tomcat 的 webapps 目录中寻找项目。

16.2 文件下载

16.2.1 实现文件下载

文件下载就是将文件服务器中的文件下载到本机上。在 Spring MVC 环境中，实现文件下载大致可分为如下两个步骤。

（1）在客户端页面使用一个文件下载的超链接，该链接的 href 属性要指定后台文件下载的方法以及文件名（需要先在文件下载目录中添加了一个名称为“1.jpg”的文件），具体代码示例如下。

```
<a href="${pageContext.request.contextPath }/download?filename=1.jpg">
    文件下载
</a>
```

（2）在后台 Controller 类中，使用 Spring MVC 提供的文件下载方法进行文件下载。Spring MVC 提供了一个 ResponseEntity 类型的对象，使用它可以很方便地定义返回的 HttpHeaders 对象和 HttpStatus 对象，通过对这两个对象的设置，即可完成下载文件时所需的配置信息。文件下载的示例代码如下所示。

```
@RequestMapping("/download")
public ResponseEntity<byte[]> fileDownload(HttpServletRequest request,
                                String filename) throws Exception{
    // 指定要下载的文件所在路径
    String path = request.getServletContext().getRealPath("/upload/");
    // 创建该文件对象
    File file = new File(path+File.separator+filename);
    // 设置响应头
    HttpHeaders headers = new HttpHeaders();
    // 通知浏览器以下载的方式打开文件
```

```
        headers.setContentDispositionFormData("attachment", filename);
        // 定义以流的形式下载返回文件数据
        headers.setContentType(MediaType.APPLICATION_OCTET_STREAM);
        // 使用 Sring MVC 框架的 ResponseEntity 对象封装返回下载数据
        return new ResponseEntity<byte[]>(FileUtils.readFileToByteArray(file),
                                          headers,HttpStatus.OK);
    }
```

在 fileDownload()方法中，首先根据文件路径和需要下载的文件名来创建文件对象，然后对响应头中文件下载时的打开方式以及下载方式进行了设置，最后返回 ResponseEntity 封装的下载结果对象。

ResponseEntity 对象有些类似前面章节介绍的@ResponseBody 注解，它用于直接返回结果对象。上面示例中，设置响应头信息中的 MediaType 代表的是 Interner Media Type（即互联网媒体类型），也叫作 MIME 类型，MediaType.APPLICATION_OCTET_STREAM 的值为 application/octet-stream，即表示以二进制流的形式下载数据；HttpStatus 类型代表的是 Http 协议中的状态，示例中的 HttpStatus.OK 表示 200，即服务器已成功处理了请求。

在 Eclipse 的 WebContent 目录下，创建一个页面文件 download.jsp，将上述第（1）步的页面代码编写到 download.jsp 中，然后将第（2）步的 fileDownload()方法编写在 FileUploadController 类中。发布项目并启动 Tomcat 服务器，在浏览器中访问地址 http://localhost:8080/chapter16/download.jsp，其显示效果如图 16-5 所示。

图16-5　fileDownload.jsp文件下载页面

单击图 16-5 中的“文件下载”链接后，会出现下载提示弹窗，如图 16-6 所示（这里以火狐浏览器为例进行演示）。

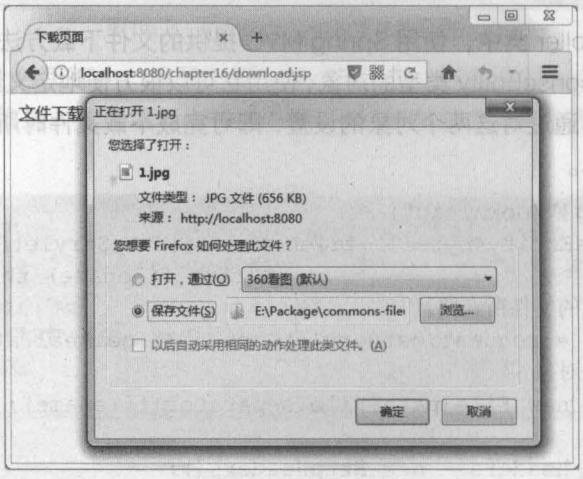


图16-6　文件下载弹窗

选择图 16-6 中的“保存文件”并单击“确定”按钮后，即可下载该文件。

16.2.2　中文名称的文件下载

虽然在 16.2.1 小节中，通过 Spring MVC 实现了文件下载功能，但此案例代码只适用于非中文名称的文件下载。当对中文名称的文件进行下载时，因为各个浏览器内部转码机制的不同，就会出现不同的乱码以及解析异常问题。例如在文件下载目录中添加一个名称为“壁纸.jpg”的文件，当通过浏览器下载该文件时，下载弹出窗口的显示如图 16-7 所示。

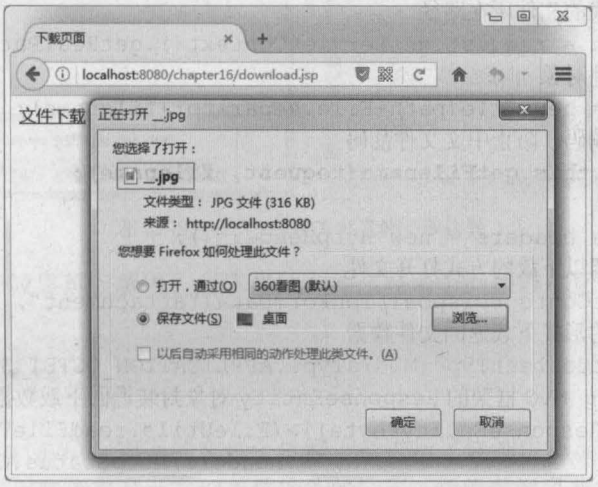


图16-7　火狐浏览器中文名文件下载信息

从图 16-7 可以看出，所要下载的文件名称并不是“壁纸.jpg”，而是“_.jpg”，这就表示中文文件名称出现了乱码。那么我们要如何解决这种乱码问题呢?

为了解决浏览器中文件下载时中文名称的乱码问题，可以在前端页面发送请求前先对中文名进行统一编码，然后在后台控制器类中对文件名称进行相应的转码，其具体实现步骤如下。

（1）在下载页面中对中文文件名编码。可以使用 Servlet API 中提供的 URLEncoder 类中的 encoder(String s, String enc)方法将中文转为 UTF-8 编码。该方法中第一个参数表示需要转码的字符串，第二个参数表示编码格式，其具体实现方式如文件 16-4 所示。

文件 16-4　download.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@page import="java.net.URLEncoder"%>
<!DOCTYPE HTML>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>下载页面</title>
</head>
<body>
    <a href="${pageContext.request.contextPath }/download?filename=<%=
                                URLEncoder.encode("壁纸.jpg", "UTF-8")%>">
        中文名称文件下载
```

```
    </a>
</body>
</html>
```

（2）修改控制器类 FileUploadController 中的 fileDownload()方法，并增加对文件名进行编码的方法，其代码如下所示。

```
@RequestMapping("/download")
public ResponseEntity<byte[]> fileDownload(HttpServletRequest request,
                                    String filename) throws Exception{
    // 指定要下载的文件所在路径
    String path = request.getServletContext().getRealPath("/upload/");
    // 创建该文件对象
    File file = new File(path+File.separator+filename);
    // 对文件名编码，防止中文文件乱码
    filename = this.getFilename(request, filename);
    // 设置响应头
    HttpHeaders headers = new HttpHeaders();
    // 通知浏览器以下载的方式打开文件
    headers.setContentDispositionFormData("attachment", filename);
    // 定义以流的形式下载返回文件数据
    headers.setContentType(MediaType.APPLICATION_OCTET_STREAM);
    // 使用 Sring MVC 框架的 ResponseEntity 对象封装返回下载数据
   return new ResponseEntity<byte[]>(FileUtils.readFileToByteArray(file),
                                    headers,HttpStatus.OK);
}
/**
 * 根据浏览器的不同进行编码设置，返回编码后的文件名
 */
public String getFilename(HttpServletRequest request,
                                    String filename) throws Exception {
    // IE 不同版本 User-Agent 中出现的关键词
    String[] IEBrowserKeyWords = {"MSIE", "Trident", "Edge"};
    // 获取请求头代理信息
    String userAgent = request.getHeader("User-Agent");
    for (String keyWord : IEBrowserKeyWords) {
        if (userAgent.contains(keyWord)) {
            //IE 内核浏览器，统一为 UTF-8 编码显示
            return URLEncoder.encode(filename, "UTF-8");
        }
    }
    //火狐等其他浏览器统一为 ISO-8859-1 编码显示
    return new String(filename.getBytes("UTF-8"), "ISO-8859-1");
}
```

在方法 getFilename()中，由于 IE 浏览器在文件编码上与其他浏览器的方式不同，所以在中文编码设置上 IE 浏览器设置为 UTF-8 编码，而火狐等其他浏览器设置为 ISO-8859-1 编码。另外由于不同版本的 IE 浏览器，请求代理 User-Agent 中的关键字也略有不同，所以在判断 IE 浏览器时，需要特别注意 User-Agent 中的关键字。

再次进行中文名的文件下载测试，并在 IE 和火狐浏览器中分别单击文件下载链接后，两个

浏览器的显示效果如图 16-8 所示。

图16-8　IE和火狐中文名文件下载效果

从图 16-8 的显示效果可以看出，所下载的文件已在两个浏览器中正确显示出了中文名称。

16.3 本章小结

本章主要对 Spring MVC 环境下的文件上传和下载进行了详细讲解。首先讲解了如何实现文件上传，并通过一个应用案例来演示文件上传功能的实现；然后讲解了非中文名称文件下载的实现过程，以及中文名称文件下载的实现过程。通过本章的学习，读者可以学会如何在 Spring MVC 环境下进行文件上传和下载，并能够掌握中文名称文件下载时乱码的解决方案。

【思考题】

1. 请简述上传表单需要满足的 3 个条件。
2. 请简述如何解决中文文件名称下载时的乱码问题。

Java EE

17 Chapter

第 17 章 SSM 框架整合

学习目标

- 了解 SSM 框架的整合思路
- 熟悉 SSM 框架整合时的配置文件内容
- 掌握 SSM 框架整合应用程序的编写

通过前面章节的学习，相信读者已经掌握了 Spring、MyBatis 以及 Spring MVC 框架的使用，并且已经掌握了 Spring 与 MyBatis 的整合使用。在实际项目开发中，这三大框架通常都会整合在一起使用。接下来，本章就对 SSM（Spring、Spring MVC 以及 MyBatis）框架的整合使用进行详细的讲解。

17.1 整合环境搭建

17.1.1 整合思路

由于 Spring MVC 是 Spring 框架中的一个模块，所以 Spring MVC 与 Spring 之间不存在整合的问题，只要引入相应 JAR 包就可以直接使用。因此 SSM 框架的整合就只涉及 Spring 与 MyBatis 的整合，以及 Spring MVC 与 MyBatis 的整合，如图 17-1 所示。

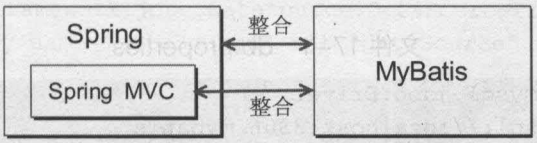


图17-1　SSM整合

在第 10 章讲解 Spring 与 MyBatis 框架的整合时，我们是通过 Spring 实例化 Bean，然后调用实例对象中的查询方法来执行 MyBatis 映射文件中的 SQL 语句的，如果能够正确查询出数据库中的数据，那么我们就认为 Spring 与 MyBatis 框架整合成功。同样，在学习完 Spring MVC 后，如果我们可以通过前台页面来执行查询方法，并且查询出的数据能够在页面中正确显示，那么我们也可以认为三大框架整合成功。

17.1.2 准备所需 JAR 包

要实现 SSM 框架的整合，首先要准备这三个框架的 JAR 包，以及其他整合所需的 JAR 包。在第 10 章讲解 Spring 与 MyBatis 框架整合时，已经介绍了 Spring 与 MyBatis 整合所需要的 JAR 包，这里只需要再加入 Spring MVC 的相关 JAR 包即可，具体如下。

- spring-web-4.3.6.RELEASE.jar
- spring-webmvc-4.3.6.RELEASE.jar

因此，SSM 整合时所需的全部 JAR 包如图 17-2 所示。

图17-2　SSM整合JAR包

小提示

SSM 框架整合时所需的 Spring 与 MyBatis 的相关 JAR 包可以参考第 10 章中 Spring 与 MyBatis 框架整合介绍的 JAR 包，也可以直接使用本书源码中的 JAR 包。

17.1.3 编写配置文件

（1）在 Eclipse 中，创建一个名为 chapter17 的 Web 项目，将整合所需的 JAR 包添加到项目的 lib 目录中，并发布到类路径下。

（2）在 chapter17 项目下，创建一个名为 config 的源文件夹（Source Folder），在该文件夹中分别创建数据库常量配置文件 db.properties、Spring 配置文件 applicationContext.xml 以及 MyBatis 的配置文件 mybatis-config.xml。这三个配置文件的实现代码如文件 17-1、文件 17-2 和文件 17-3 所示。

文件 17-1　db.properties

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/mybatis
jdbc.username=root
jdbc.password=root
jdbc.maxTotal=30
jdbc.maxIdle=10
jdbc.initialSize=5
```

文件 17-2　applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-4.3.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.3.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-4.3.xsd">
    <!-- 读取 db.properties -->
    <context:property-placeholder location="classpath:db.properties"/>
    <!-- 配置数据源 -->
     <bean id="dataSource"
                        class="org.apache.commons.dbcp2.BasicDataSource">
        <!--数据库驱动 -->
        <property name="driverClassName" value="${jdbc.driver}" />
```

```
22          <!--连接数据库的 url -->
23          <property name="url" value="${jdbc.url}" />
24          <!--连接数据库的用户名 -->
25          <property name="username" value="${jdbc.username}" />
26          <!--连接数据库的密码 -->
27          <property name="password" value="${jdbc.password}" />
28          <!--最大连接数 -->
29          <property name="maxTotal" value="${jdbc.maxTotal}" />
30          <!--最大空闲连接  -->
31          <property name="maxIdle" value="${jdbc.maxIdle}" />
32          <!--初始化连接数  -->
33          <property name="initialSize" value="${jdbc.initialSize}" />
34      </bean>
35      <!-- 事务管理器，依赖于数据源 -->
36      <bean id="transactionManager" class=
37      "org.springframework.jdbc.datasource.DataSourceTransactionManager">
38          <property name="dataSource" ref="dataSource" />
39      </bean>
40      <!-- 开启事务注解 -->
41      <tx:annotation-driven transaction-manager="transactionManager"/>
42      <!-- 配置 MyBatis 工厂 SqlSessionFactory -->
43      <bean id="sqlSessionFactory"
44                          class="org.mybatis.spring.SqlSessionFactoryBean">
45          <!--注入数据源 -->
46          <property name="dataSource" ref="dataSource" />
47          <!--指定核 MyBatis 心配置文件位置 -->
48    <property name="configLocation" value="classpath:mybatis-config.xml" />
49      </bean>
50      <!-- 配置 mapper 扫描器 -->
51      <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
52          <property name="basePackage" value="com.itheima.dao"/>
53      </bean>
54      <!-- 扫描 Service -->
55      <context:component-scan base-package="com.itheima.service" />
56 </beans>
```

在文件 17-2 中，首先定义了读取 db.properties 文件的配置和数据源配置，然后配置了事务管理器并开启了事务注解。接下来配置了用于整合 MyBatis 框架的 MyBatis 工厂信息，最后定义了 mapper 扫描器来扫描 DAO 层以及扫描 Service 层的配置。

小提示

在实际开发时，为了避免 Spring 配置文件中的信息过于臃肿，通常会将 Spring 配置文件中的信息按照不同的功能分散在多个配置文件中。例如可以将事务配置放置在名称为 applicationContext-transaction.xml 的文件中，将数据源等信息放置在名称为 applicationContext-db.xml 的文件中等。这样，在 web.xml 中配置加载 Spring 文件信息时，只需通过 applicationContext-*.xml 的方式即可自动加载全部配置文件。

文件 17-3　mybatis-config.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <!-- 别名定义 -->
    <typeAliases>
        <package name="com.itheima.po" />
    </typeAliases>
</configuration>
```

由于在 Spring 中已经配置了数据源信息以及 mapper 接口文件扫描器，所以在 MyBatis 的配置文件中只需要根据 POJO 类路径进行别名配置即可。

（3）在 config 文件夹中，创建 Spring MVC 的配置文件 springmvc-config.xml，编辑后如文件 17-4 所示。

文件 17-4　springmvc-config.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:mvc="http://www.springframework.org/schema/mvc"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
  http://www.springframework.org/schema/mvc
  http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context-4.3.xsd">
    <!-- 配置包扫描器，扫描@Controller 注解的类 -->
    <context:component-scan base-package="com.itheima.controller" />
    <!-- 加载注解驱动 -->
    <mvc:annotation-driven />
    <!-- 配置视图解析器 -->
    <bean class=
   "org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

在文件 17-4 中，主要配置了用于扫描@Controller 注解的包扫描器、注解驱动器以及视图解析器。

（4）在 web.xml 中，配置 Spring 的文件监听器、编码过滤器以及 Spring MVC 的前端控制器等信息，如文件 17-5 所示。

文件 17-5　web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://xmlns.jcp.org/xml/ns/javaee"
```

```
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
    id="WebApp_ID" version="3.1">
    <!-- 配置加载 Spring 文件的监听器-->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:applicationContext.xml</param-value>
    </context-param>
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    <!-- 编码过滤器 -->
    <filter>
        <filter-name>encoding</filter-name>
        <filter-class>
            org.springframework.web.filter.CharacterEncodingFilter
        </filter-class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>encoding</filter-name>
        <url-pattern>*.action</url-pattern>
    </filter-mapping>
    <!-- 配置 Spring MVC 前端核心控制器 -->
    <servlet>
        <servlet-name>springmvc</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:springmvc-config.xml</param-value>
        </init-param>
        <!-- 配置服务器启动后立即加载 Spring MVC 配置文件 -->
    <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>springmvc</servlet-name>
        <!--/:拦截所有请求（除了 jsp）-->
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

17.2 整合应用测试

17.1 节已经完成了 SSM 框架整合环境的搭建工作，可以说完成了这些配置后，就已经完成了这三个框架大部分的整合工作。接下来，同样以查询客户信息为例，来讲解 SSM 框架的整合开发，其具体实现步骤如下。

（1）在 src 目录下，创建一个 com.itheima.po 包，并在包中创建持久化类 Customer，编辑后如文件 17-6 所示。

文件 17-6　Customer.java

```
1  package com.itheima.po;
2  /**
3   * 客户持久化类
4   */
5  public class Customer {
6      private Integer id;          // 主键 id
7      private String username;     // 客户名称
8      private String jobs;         // 职业
9      private String phone;        // 电话
10     public Integer getId() {
11         return id;
12     }
13     public void setId(Integer id) {
14         this.id = id;
15     }
16     public String getUsername() {
17         return username;
18     }
19     public void setUsername(String username) {
20         this.username = username;
21     }
22     public String getJobs() {
23         return jobs;
24     }
25     public void setJobs(String jobs) {
26         this.jobs = jobs;
27     }
28     public String getPhone() {
29         return phone;
30     }
31     public void setPhone(String phone) {
32         this.phone = phone;
33     }
34 }
```

在文件 17-6 中，编写了一个用于映射数据库表 t_customer 的客户持久化类，在类中分别定义了 id、username、jobs 和 phone 属性，以及其对应的 getter/setter 方法。

（2）在 src 目录下，创建一个 com.itheima.dao 包，并在包中创建接口文件 CustomerDao 以及对应的映射文件 CustomerDao.xml，编辑后分别如文件 17-7 和文件 17-8 所示。

文件 17-7　CustomerDao.java

```
1  package com.itheima.dao;
2  import com.itheima.po.Customer;
3  /**
4   * Customer 接口文件
5   */
6  public interface CustomerDao {
7      /**
8       * 根据 id 查询客户信息
9       */
10     public Customer findCustomerById(Integer id);
11 }
```

从上述代码可以看出，CustomerDao 中只定义了一个根据 id 查询客户信息的方法。

文件 17-8　CustomerDao.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
3     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
4  <mapper namespace="com.itheima.dao.CustomerDao">
5      <!--根据 id 查询客户信息 -->
6      <select id="findCustomerById" parameterType="Integer"
7                                    resultType="Customer">
8          select * from t_customer where id = #{id}
9      </select>
10 </mapper>
```

在文件 17-8 中，根据文件 17-7 中接口文件的方法编写了对应的执行语句信息。

小提示

在前面小节整合环境搭建时，已经在配置文件 applicationContext.xml 中使用包扫描的形式加入了扫描包 com.itheima.dao 下的所有接口及映射文件，所以在这里完成 DAO 层接口及映射文件开发后，就不必再进行映射文件的扫描配置了。

（3）在 src 目录下，创建一个 com.itheima.service 包，然后在包中创建接口文件 CustomerService，并在 CustomerService 中定义通过 id 查询客户的方法，如文件 17-9 所示。

文件 17-9　CustomerService.java

```
1  package com.itheima.service;
2  import com.itheima.po.Customer;
3  public interface CustomerService {
4      public Customer findCustomerById(Integer id);
5  }
```

（4）在 src 目录下，创建一个 com.itheima.service.impl 包，并在包中创建 CustomerService 接口的实现类 CustomerServiceImpl，编辑后如文件 17-10 所示。

文件 17-10　CustomerServiceImpl.java

```
1  package com.itheima.service.impl;
2  import org.springframework.beans.factory.annotation.Autowired;
3  import org.springframework.stereotype.Service;
4  import org.springframework.transaction.annotation.Transactional;
5  import com.itheima.dao.CustomerDao;
6  import com.itheima.po.Customer;
7  import com.itheima.service.CustomerService;
8  @Service
9  @Transactional
10 public class CustomerServiceImpl implements CustomerService {
11     //注解注入 CustomerDao
12     @Autowired
13     private CustomerDao customerDao;
14     //查询客户
15     public Customer findCustomerById(Integer id) {
16         return this.customerDao.findCustomerById(id);
17     }
18 }
```

在文件 17-10 中，使用了@Service 注解来标识业务层的实现类，使用了@Transactional 注解来标识类中的所有方法都纳入 Spring 的事务管理，并使用@Autowired 注解将 CustomerDao 接口对象注入到本类中，然后在本类的查询方法中调用了 CustomerDao 对象的查询客户方法。

小提示

在上述代码中，@Transactional 注解主要是针对数据的增加、修改、删除进行事务管理，上示例中的查询方法并不需要使用该注解，此处的作用就是告知读者该注解在实际开发中应该如何使用。

（5）在 src 目录下，创建一个 com.itheima.controller 包，并在包中创建用于处理页面请求的控制器类 CustomerController，编辑后如文件 17-11 所示。

文件 17-11　CustomerController.java

```
1  package com.itheima.controller;
2  import org.springframework.beans.factory.annotation.Autowired;
3  import org.springframework.stereotype.Controller;
4  import org.springframework.ui.Model;
5  import org.springframework.web.bind.annotation.RequestMapping;
6  import com.itheima.po.Customer;
7  import com.itheima.service.CustomerService;
8  @Controller
9  public class CustomerController {
10     @Autowired
11     private CustomerService customerService;
12     /**
13      * 根据 id 查询客户详情
14      */
15     @RequestMapping("/findCustomerById")
```

```
16      public String findCustomerById(Integer id,Model model) {
17          Customer customer = customerService.findCustomerById(id);
18          model.addAttribute("customer", customer);
19          //返回客户信息展示页面
20          return "customer";
21      }
22  }
```

在文件17-11中，先使用了Spring的注解@Controller来标识控制器类，然后通过@Autowired注解将CustomerService接口对象注入到本类中，最后编写了一个根据id查询客户详情的方法findCustomerById()，该方法会将获取的客户详情返回到视图名为customer的jsp页面中。

（6）在WEB-INF目录下，创建一个jsp文件夹，在该文件夹下创建一个用于展示客户详情的页面文件customer.jsp，编辑后如文件17-12所示。

文件17-12 customer.jsp

```
1  <%@ page language="java" contentType="text/html; charset=UTF-8"
2      pageEncoding="UTF-8"%>
3  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
4      "http://www.w3.org/TR/html4/loose.dtd">
5  <html>
6  <head>
7  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
8  <title>客户信息</title>
9  </head>
10 <body>
11     <table border=1>
12         <tr>
13             <td>编号</td>
14             <td>名称</td>
15             <td>职业</td>
16             <td>电话</td>
17         </tr>
18         <tr>
19             <td>${customer.id}</td>
20             <td>${customer.username}</td>
21             <td>${customer.jobs}</td>
22             <td>${customer.phone}</td>
23         </tr>
24     </table>
25 </body>
26 </html>
```

在文件17-12中，编写了一个用于展示客户信息的表格，表格会通过EL表达式来获取后台控制层返回的客户信息。

（7）将项目发布到Tomcat服务器并启动，在浏览器中访问地址http://localhost:8080/chapter17/findCustomerById?id=1，其显示效果如图17-3所示。

从图17-3可以看出，通过浏览器已经成功查询出了t_customer表中id为1的客户信息，这也就说明SSM框架整合成功。

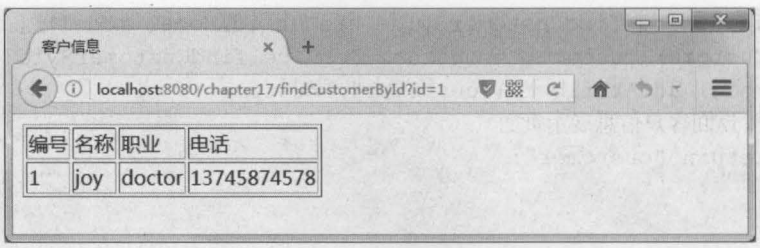


图17-3 查询结果

17.3 本章小结

本章主要讲解了 SSM 框架的整合知识。首先对 SSM 框架整合的环境搭建进行了讲解，然后通过一个查询客户信息的案例讲解了具体的整合过程。通过本章的学习，读者将能够了解 SSM 框架的整合思路，掌握 SSM 框架的整合过程以及如何使用。框架的整合是 SSM 框架使用的基础，读者一定要多加练习，并熟练掌握。

【思考题】

1. 请简述 SSM 框架整合思路。
2. 请简述 SSM 框架整合时，Spring 配置文件中的配置信息（无须写代码，只需简单描述所要配置的内容即可）。

Java EE

18 Chapter

第 18 章
BOOT 客户管理系统

学习目标

- 了解系统架构和文件组织结构
- 熟悉系统环境搭建的步骤
- 掌握登录模块和客户管理模块功能代码的编写

本章将通过前面章节学习的 SSM（Spring+Spring MVC+MyBatis）框架知识来实现一个简易的客户管理系统。该系统在开发过程中，整合了三大框架，并在整合的基础上实现了系统的登录模块以及客户管理模块。

18.1 系统概述

18.1.1 系统功能介绍

本系统后台使用 SSM 框架编写，前台页面使用当前主流的 Bootstrap 和 jQuery 框架完成页面信息展示功能（关于 Bootstrap 的知识，有兴趣的读者可参考黑马程序员编著的《响应式 Web 开发项目教程》）。

系统中主要实现了两大功能模块：用户登录模块和客户管理模块，这两个模块的主要功能如图 18-1 所示。

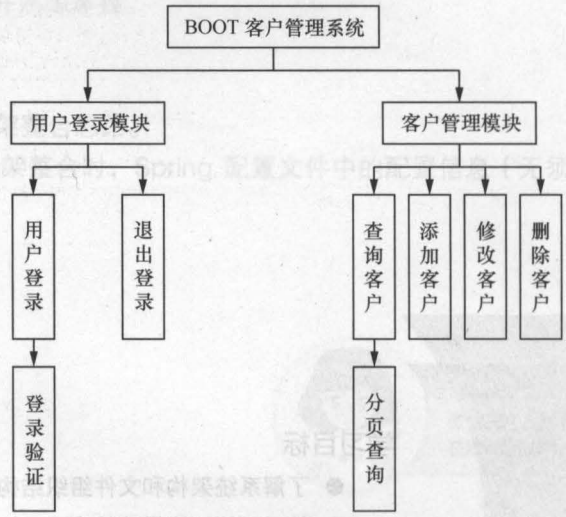


图18-1 系统功能结构

18.1.2 系统架构设计

本系统根据功能的不同，项目结构可以划分为以下几个层次。

- 持久对象层（也称持久层或持久化层）：该层由若干持久化类（实体类）组成。
- 数据访问层（DAO 层）：该层由若干 DAO 接口和 MyBatis 映射文件组成。接口的名称统一以 Dao 结尾，且 MyBatis 的映射文件名称要与接口的名称相同。
- 业务逻辑层（Service 层）：该层由若干 Service 接口和实现类组成。在本系统中，业务逻辑层的接口统一使用 Service 结尾，其实现类名称统一在接口名后加 Impl。该层主要用于实现系统的业务逻辑。
- Web 表现层：该层主要包括 Spring MVC 中的 Controller 类和 JSP 页面。Controller 类主要负责拦截用户请求，并调用业务逻辑层中相应组件的业务逻辑方法来处理用户请求，然后将相

应的结果返回给 JSP 页面。

为了让读者更清晰地了解各个层次之间的关系，下面通过一张图来描述各个层次的关系和作用，如图 18-2 所示。

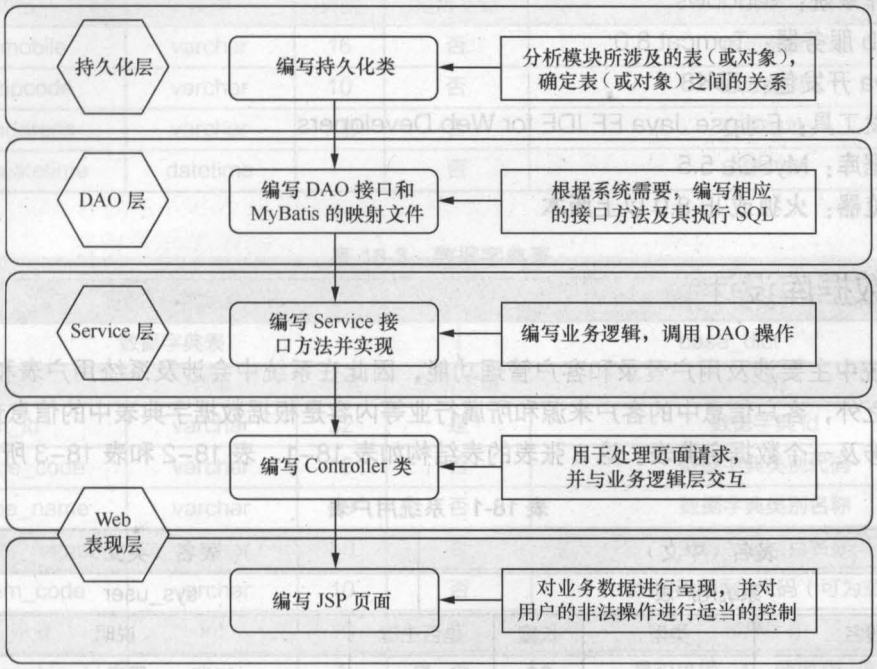


图18-2　系统层次结构

18.1.3　文件组织结构

在正式讲解项目的编写之前，先来了解项目中所涉及的包文件、配置文件以及页面文件等在项目中的组织结构，如图 18-3 所示。

图18-3　项目文件组织结构

18.1.4 系统开发及运行环境

BOOT 客户管理系统开发环境如下。

- 操作系统：Windows
- Web 服务器：Tomcat 8.0
- Java 开发包：JDK8
- 开发工具：Eclipse Java EE IDE for Web Developers
- 数据库：MySQL 5.5
- 浏览器：火狐或 IE 8.0 以上版本

18.2 数据库设计

本系统中主要涉及用户登录和客户管理功能，因此在系统中会涉及系统用户表和客户信息表。除此之外，客户信息中的客户来源和所属行业等内容是根据数据字典表中的信息查询出的，所以还会涉及一个数据字典表。这 3 张表的表结构如表 18-1、表 18-2 和表 18-3 所示。

表 18-1 系统用户表

表名（中文）			表名（英文）	
系统用户表			sys_user	
字段名	类型	长度	是否主键	说明
user_id	int	32	是	用户 id
user_code	varchar	32	否	用户账号
user_name	varchar	50	否	用户名称
user_password	varchar	32	否	用户密码
user_state	varchar	1	否	用户状态（1：正常，0：暂停）

表 18-2 客户信息表

表名（中文）			表名（英文）	
客户信息表			customer	
字段名	类型	长度	是否主键	说明
cust_id	int	32	是	客户编号
cust_name	varchar	50	否	客户名称
cust_user_id	int	32	否	负责人 id
cust_create_id	int	32	否	创建人 id
cust_source	varchar	50	否	客户信息来源
cust_industry	varchar	50	否	客户所属行业
cust_level	varchar	32	否	客户级别
cust_linkman	varchar	50	否	联系人
cust_phone	varchar	64	否	固定电话

续表

表名（中文）			表名（英文）	
客户信息表			customer	
字段名	类型	长度	是否主键	说明
cust_mobile	varchar	16	否	移动电话
cust_zipcode	varchar	10	否	邮政编码
cust_address	varchar	100	否	联系地址
cust_createtime	datetime		否	创建时间

表 18-3　数据字典表

表名（中文）			表名（英文）	
数据字典表			base_dict	
字段名	类型	长度	是否主键	说明
dict_id	varchar	32	是	数据字典 id
dict_type_code	varchar	10	否	数据字典类别代码
dict_type_name	varchar	50	否	数据字典类别名称
dict_item_name	varchar	50	否	数据字典项目名称
dict_item_code	varchar	10	否	数据字典项目代码（可为空）
dict_sort	int	10	否	排序字段
dict_enable	char	1	否	是否可用：1：使用 0：停用
dict_memo	varchar	100	否	备注

18.3　系统环境搭建

18.3.1　准备所需 JAR 包

由于本系统使用的是 SSM 框架开发，因此需要准备这三大框架的 JAR 包。除此之外，项目中还涉及数据库连接、JSTL 标签等，所以还要准备其他 JAR 包。整个系统所需要准备的 JAR 共计 35 个，具体如下所示。

1. Spring 框架所需的 JAR 包（10 个）

主要包括 4 个核心模块 JAR，AOP 开发使用的 JAR，JDBC 和事务的 JAR。

- aopalliance-1.0.jar
- aspectjweaver-1.8.10.jar
- spring-aop-4.3.6.RELEASE.jar
- spring-aspects-4.3.6.RELEASE.jar
- spring-beans-4.3.6.RELEASE.jar
- spring-context-4.3.6.RELEASE.jar

- spring-core-4.3.6.RELEASE.jar
- spring-expression-4.3.6.RELEASE.jar
- spring-jdbc-4.3.6.RELEASE.jar
- spring-tx-4.3.6.RELEASE.jar

2. Spring MVC 框架所需要的 JAR 包（2 个）

- spring-web-4.3.6.RELEASE.jar
- spring-webmvc-4.3.6.RELEASE.jar

3. MyBatis 框架所需的 JAR 包（13 个）

主要包括核心包 mybatis-3.4.2.jar，以及其解压文件夹中 lib 目录下的所有 JAR。

- ant-1.9.6.jar
- ant-launcher-1.9.6.jar
- asm-5.1.jar
- cglib-3.2.4.jar
- commons-logging-1.2.jar
- javassist-3.21.0-GA.jar
- log4j-1.2.17.jar
- log4j-api-2.3.jar
- log4j-core-2.3.jar
- mybatis-3.4.2.jar
- ognl-3.1.12.jar
- slf4j-api-1.7.22.jar
- slf4j-log4j12-1.7.22.jar

4. MyBatis 与 Spring 整合的中间 JAR（1 个）

- mybatis-spring-1.3.1.jar

5. 数据库驱动 JAR 包（1 个）

- mysql-connector-java-5.1.40-bin.jar

6. 数据源 dbcp 所需 JAR 包（2 个）

- commons-dbcp2-2.1.1.jar
- commons-pool2-2.4.2.jar

7. JSTL 标签库 JAR 包（2 个）

- taglibs-standard-impl-1.2.5.jar
- taglibs-standard-spec-1.2.5.jar

8. Jackson 框架所需 JAR 包（3 个）

- jackson-annotations-2.8.6.jar
- jackson-core-2.8.6.jar
- jackson-databind-2.8.6.jar

9. Java 工具类 JAR（1 个）

- commons-lang3-3.4.jar

上面所需要准备的 JAR 包（除 JSTL 的 2 个 JAR 包和 commons-lang3-3.4.jar 外），都

是本书前面章节所使用过的。读者在学习本章时，可以直接下载项目源码，并使用源码中的JAR包。

小提示

本书中使用的JSTL版本是1.2.5，此版本中需要引入的JAR包为taglibs-standard-spec-1.2.5.jar（相当于之前的jstl.jar，属于接口定义类）和taglibs-standard-impl-1.2.5.jar jar（相当于之前的standard.jar，属于实现类）。这两个JAR包可以通过网址“http://tomcat.apache.org/download-taglibs.cgi#Standard-1.2.5”下载得到。

18.3.2 准备数据库资源

通过MySQL 5.5 Command Line Client登录数据库后，创建一个名称为boot_crm的数据库，并选择该数据库。通过SQL命令将本书资源中所提供的boot_crm.sql文件导入到boot_crm数据库中，即可导入本系统所使用的全部数据，其具体实现SQL命令如下。

（1）创建数据库。

```
CREATE DATABASE boot_crm;
```

（2）选择所创建的数据库。

```
USE boot_crm;
```

（3）导入数据库文件，这里假设该文件在F盘的根目录下，其导入命令如下。

```
source F:\boot_crm.sql;
```

除此之外，还可以通过其他客户端软件导入sql文件，如SQLyog等。

18.3.3 准备项目环境

1. 创建项目，引入JAR包

在Eclipse中，创建一个名称为boot-crm的Web项目，将系统所准备的全部JAR包复制到项目的lib目录中，并发布到类路径下。

2. 编写配置文件

（1）在项目目录下创建一个源文件夹config，并在config文件夹下分别创建数据库常量配置文件、Spring配置文件、MyBatis配置文件、log4j配置文件、资源配置文件以及Spring MVC配置文件。其中log4j配置文件log4j.properties、数据库常量配置文件db.properties与MyBatis配置文件mybatis-config.xml的配置与第17章讲解整合时的配置代码基本相同（注意修改数据库名称与包名），这里将不再重复讲解。其他3个配置文件的代码分别如文件18-1、文件18-2和文件18-3所示。

文件18-1 applicationContext.xml

```
1  <beans xmlns="http://www.springframework.org/schema/beans"
2      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3      xmlns:mvc="http://www.springframework.org/schema/mvc"
4      xmlns:context="http://www.springframework.org/schema/context"
5      xmlns:aop="http://www.springframework.org/schema/aop"
```

```
6     xmlns:tx="http://www.springframework.org/schema/tx"
7     xsi:schemaLocation="http://www.springframework.org/schema/beans
8     http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
9     http://www.springframework.org/schema/mvc
10    http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd
11    http://www.springframework.org/schema/context
12    http://www.springframework.org/schema/context/spring-context-4.3.xsd
13    http://www.springframework.org/schema/aop
14    http://www.springframework.org/schema/aop/spring-aop-4.3.xsd
15    http://www.springframework.org/schema/tx
16    http://www.springframework.org/schema/tx/spring-tx-4.3.xsd">
17    <!--读取 db.properties -->
18    <context:property-placeholder location="classpath:db.properties"/>
19    <!-- 配置数据源 -->
20    <bean id="dataSource"
21        class="org.apache.commons.dbcp2.BasicDataSource">
22        <!--数据库驱动 -->
23        <property name="driverClassName" value="${jdbc.driver}" />
24        <!--连接数据库的 url -->
25        <property name="url" value="${jdbc.url}" />
26        <!--连接数据库的用户名 -->
27        <property name="username" value="${jdbc.username}" />
28        <!--连接数据库的密码 -->
29        <property name="password" value="${jdbc.password}" />
30        <!--最大连接数 -->
31        <property name="maxTotal" value="${jdbc.maxTotal}" />
32        <!--最大空闲连接  -->
33        <property name="maxIdle" value="${jdbc.maxIdle}" />
34        <!--初始化连接数  -->
35        <property name="initialSize" value="${jdbc.initialSize}" />
36    </bean>
37    <!-- 事务管理器 -->
38    <bean id="transactionManager" class=
39    "org.springframework.jdbc.datasource.DataSourceTransactionManager">
40        <!-- 数据源 -->
41        <property name="dataSource" ref="dataSource" />
42    </bean>
43    <!-- 通知 -->
44    <tx:advice id="txAdvice" transaction-manager="transactionManager">
45        <tx:attributes>
46            <!-- 传播行为 -->
47            <tx:method name="save*" propagation="REQUIRED" />
48            <tx:method name="insert*" propagation="REQUIRED" />
49            <tx:method name="add*" propagation="REQUIRED" />
50            <tx:method name="create*" propagation="REQUIRED" />
51            <tx:method name="delete*" propagation="REQUIRED" />
52            <tx:method name="update*" propagation="REQUIRED" />
53            <tx:method name="find*" propagation="SUPPORTS"
```

```
                                    read-only="true" />
            <tx:method name="select*" propagation="SUPPORTS"
                                    read-only="true" />
            <tx:method name="get*" propagation="SUPPORTS"
                                    read-only="true" />
        </tx:attributes>
    </tx:advice>
    <!-- 切面 -->
    <aop:config>
        <aop:advisor advice-ref="txAdvice"
            pointcut="execution(* com.itheima.core.service.*.*(..))" />
    </aop:config>
    <!-- 配置 MyBatis 的工厂 -->
    <bean class="org.mybatis.spring.SqlSessionFactoryBean">
        <!-- 数据源 -->
        <property name="dataSource" ref="dataSource" />
        <!-- 配置 MyBatis 的核心配置文件所在位置 -->
        <property name="configLocation"
                value="classpath:mybatis-config.xml" />
    </bean>
    <!-- 接口开发,扫描 com.itheima.core.dao 包 , 写在此包下的接口即可被扫描到 -->
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <property name="basePackage" value="com.itheima.core.dao" />
    </bean>
    <!-- 配置扫描@Service 注解 -->
    <context:component-scan base-package="com.itheima.core.service"/>
</beans>
```

上述代码与上一章整合时的配置文件代码有所不同的是增加了事务传播行为以及切面的配置。在事务的传播行为中，只有查询方法的事务为只读，添加、修改和删除的操作必须纳入事务管理。

文件 18-2 resource.properties

```
customer.from.type=002
customer.industry.type=001
customer.level.type=006
```

上述配置代码分别表示客户来源、所属行业和客户级别，其值对应的是数据字典表中 dict_type_code 字段的值。

文件 18-3 springmvc-config.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
```

```
    http://www.springframework.org/schema/mvc
    http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.3.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-4.3.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-4.3.xsd">
    <!-- 加载属性文件 -->
    <context:property-placeholder
            location="classpath:resource.properties" />
    <!-- 配置扫描器 -->
    <context:component-scan
            base-package="com.itheima.core.web.controller" />
    <!-- 注解驱动：配置处理器映射器和适配器 -->
    <mvc:annotation-driven />
    <!--配置静态资源的访问映射，此配置中的文件，将不被前端控制器拦截 -->
    <mvc:resources location="/js/" mapping="/js/**" />
    <mvc:resources location="/css/" mapping="/css/**" />
    <mvc:resources location="/fonts/" mapping="/fonts/**" />
    <mvc:resources location="/images/" mapping="/images/**" />
    <!-- 配置视图解释器-->
    <bean id="jspViewResolver" class=
    "org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

上述代码除配置了需要扫描的包、注解驱动和视图解析器外，还增加了加载属性文件和访问静态资源的配置。

（2）在 web.xml 中，配置 Spring 的监听器、编码过滤器和 Spring MVC 的前端控制器等信息，如文件 18-4 所示。

文件 18-4　web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                    http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
    id="WebApp_ID" version="3.1">
    <!-- 配置加载 Spring 文件的监听器-->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:applicationContext.xml</param-value>
    </context-param>
    <listener>
        <listener-class>
```

```
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    <!-- 编码过滤器 -->
    <filter>
        <filter-name>encoding</filter-name>
        <filter-class>
            org.springframework.web.filter.CharacterEncodingFilter
        </filter-class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>encoding</filter-name>
        <url-pattern>*.action</url-pattern>
    </filter-mapping>
    <!-- 配置 Spring MVC 前端核心控制器 -->
    <servlet>
        <servlet-name>crm</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:springmvc-config.xml</param-value>
        </init-param>
        <!-- 配置服务器启动后立即加载 Spring MVC 配置文件 -->
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>crm</servlet-name>
        <url-pattern>*.action</url-pattern>
    </servlet-mapping>
    <!-- 系统默认页面 -->
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

3. 引入页面资源

将项目运行所需要的 CSS 文件、字体、图片、JS、自定义标签文件和 JSP 文件按照图 18-3 中的结构引入到项目中。

至此，开发系统前的环境准备工作就已经完成。此时如果将项目发布到 Tomcat 服务器并访问项目首页地址 http://localhost:8080/boot-crm/index.jsp，如图 18-4 所示。

图18-4　登录页面

从图 18-4 可以看出，访问系统首页时，页面所展示的是系统登录页面。在下一节中，我们将对系统的登录功能编写进行详细讲解。

18.4 用户登录模块

18.4.1 用户登录

BOOT 客户管理系统用户登录功能的实现流程如图 18-5 所示。

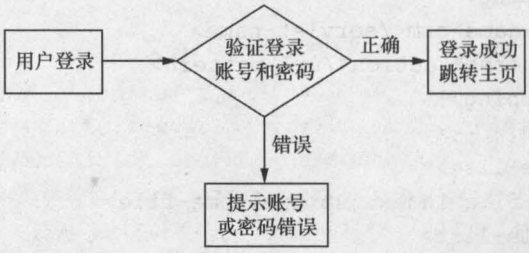


图18-5　登录流程

从图 18-5 可以看出，用户登录过程中首先要验证用户名和密码是否正确，如果正确，可以成功登录系统，系统会自动跳转到主页；如果错误，则在登录页面给出错误提示信息。

下面就依照图 18-5 中的流程，来实现系统登录功能，具体步骤如下。

1. 创建持久化类

在 src 目录下，创建一个 com.itheima.core. po 包，在包中创建用户持久化类 User，并在

User 类中定义用户相关属性以及相应的 getter/setter 方法，如文件 18-5 所示。

文件 18-5 User.java

```
package com.itheima.core.po;
import java.io.Serializable;
/**
 * 用户持久化类
 */
public class User implements Serializable{
    private static final long serialVersionUID = 1L;
    private Integer user_id;      //用户 id
    private String user_code;     //用户账号
    private String user_name;     //用户名称
    private String user_password; //用户密码
    private Integer user_state;   //用户状态
    public Integer getUser_id() {
        return user_id;
    }
    public void setUser_id(Integer user_id) {
        this.user_id = user_id;
    }
    public String getUser_code() {
        return user_code;
    }
    public void setUser_code(String user_code) {
        this.user_code = user_code;
    }
    public String getUser_name() {
        return user_name;
    }
    public void setUser_name(String user_name) {
        this.user_name = user_name;
    }
    public String getUser_password() {
        return user_password;
    }
    public void setUser_password(String user_password) {
        this.user_password = user_password;
    }
    public Integer getUser_state() {
        return user_state;
    }
    public void setUser_state(Integer user_state) {
        this.user_state = user_state;
    }
}
```

2. 实现 DAO

（1）创建用户 DAO 层接口。在 src 目录下，创建一个 com.itheima.core.dao 包，在包中创建一个用户接口 UserDao，并在接口中编写通过账号和密码查询用户的方法，如文件 18-6

所示。

文件 18-6　UserDao.java

```
1  package com.itheima.core.dao;
2  import org.apache.ibatis.annotations.Param;
3  import com.itheima.core.po.User;
4  /**
5   * 用户 DAO 层接口
6   */
7  public interface UserDao {
8      /**
9       * 通过账号和密码查询用户
10      */
11     public User findUser(@Param("usercode") String usercode,
12                          @Param("password") String password);
13 }
```

在上述方法代码的参数中，@Param("usercode")表示为参数 usercode 命名，命名后，在映射文件的 SQL 中，使用#{usercode}就可以获取 usercode 的参数值。

（2）创建映射文件。在 com.itheima.core.dao 包中，创建一个 MyBatis 映射文件 UserDao.xml，并在映射文件中编写查询用户信息的执行语句，如文件 18-7 所示。

文件 18-7　UserDao.xml

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
3     "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
4  <mapper namespace="com.itheima.core.dao.UserDao" >
5     <!-- 查询用户 -->
6     <select id="findUser" parameterType="String" resultType="user">
7         select *
8         from sys_user
9         where user_code = #{usercode}
10        and user_password =#{password}
11        and user_state = '1'
12    </select>
13 </mapper>
```

上述代码通过映射查询语句来查询系统用户表中的可用用户。

3. 实现 Service

（1）创建用户 Service 层接口。在 src 目录下，创建一个 com.itheima.core.service 包，在包中创建 UserService 接口，并在该接口中编写一个通过账号和密码查询用户的方法，如文件 18-8 所示。

文件 18-8　UserService.java

```
1  package com.itheima.core.service;
2  import com.itheima.core.po.User;
3  /**
4   * 用户 Service 层接口
5   */
```

```
6  public interface UserService {
7      // 通过账号和密码查询用户
8      public User findUser(String usercode,String password);
9  }
```

（2）创建用户 Service 层接口的实现类。在 src 目录下，创建一个 com.itheima.core.service.impl 包，并在包中创建 UserService 接口的实现类 UserServiceImpl，在类中编辑并实现接口中的方法，如文件 18-9 所示。

文件 18-9　UserServiceImpl.java

```
1  package com.itheima.core.service.impl;
2  import org.springframework.beans.factory.annotation.Autowired;
3  import org.springframework.stereotype.Service;
4  import org.springframework.transaction.annotation.Transactional;
5  import com.itheima.core.dao.UserDao;
6  import com.itheima.core.po.User;
7  import com.itheima.core.service.UserService;
8  /**
9   * 用户 Service 接口实现类
10  */
11 @Service("userService")
12 @Transactional
13 public class UserServiceImpl implements UserService {
14     // 注入 UserDao
15     @Autowired
16     private UserDao userDao;
17     // 通过账号和密码查询用户
18     @Override
19     public User findUser(String usercode, String password) {
20         User user = this.userDao.findUser(usercode, password);
21         return user;
22     }
23 }
```

在上述代码的 findUser()方法中，调用了 UserDao 对象中的 findUser()方法来查询用户信息，并将查询到的信息返回。

4. 实现 Controller

在 src 目录下，创建一个 com.itheima.core.web.controller 包，在包中创建用户控制器类 UserController，编辑后的代码如文件 18-10 所示。

文件 18-10　UserController.java

```
1  package com.itheima.core.web.controller;
2  import javax.servlet.http.HttpSession;
3  import org.springframework.beans.factory.annotation.Autowired;
4  import org.springframework.stereotype.Controller;
5  import org.springframework.ui.Model;
6  import org.springframework.web.bind.annotation.RequestMapping;
7  import org.springframework.web.bind.annotation.RequestMethod;
```

```
import com.itheima.core.po.User;
import com.itheima.core.service.UserService;
/**
 * 用户控制器类
 */
@Controller
public class UserController {
    // 依赖注入
    @Autowired
    private UserService userService;
    /**
     * 用户登录
     */
    @RequestMapping(value = "/login.action", method = RequestMethod.POST)
    public String login(String usercode,String password, Model model,
                                                HttpSession session) {
        // 通过账号和密码查询用户
        User user = userService.findUser(usercode, password);
        if(user != null){
            // 将用户对象添加到 Session
            session.setAttribute("USER_SESSION", user);
            // 跳转到主页面
            return "customer";
        }
        model.addAttribute("msg", "账号或密码错误，请重新输入！");
        // 返回到登录页面
        return "login";
    }
}
```

在文件 18-10 中，首先通过@Autowired 注解将 UserService 对象注入到了本类中，然后创建了一个用于用户登录的 login()方法。由于在用户登录时，表单都会以 POST 方式提交，所以将@RequestMapping 注解的 method 属性值设置为 RequestMethod.POST。在 login()方法中，首先通过页面中传递过来的账号和密码查询用户，然后通过 if 语句判断是否存在该用户。如果存在，就将用户信息存储到 Session 中，并跳转到系统主页面；如果不存在，则提示错误信息，并返回到登录页面。

5. 实现页面功能

（1）系统默认首页 index.jsp 主要实现了一个转发功能，在访问时会转发到登录页面，其实现代码如文件 18-11 所示。

文件 18-11　index.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!-- 转发到登录页面 -->
<jsp:forward page="/WEB-INF/jsp/login.jsp"/>
```

（2）登录页面中，主要包含一个登录表单，其页面实现代码如文件 18-12 所示。

文件 18-12 login.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE HTML>
<html>
<head>
<title>登录页面</title>
<meta http-equiv=Content-Type content="text/html; charset=utf-8">
<meta content="MSHTML 6.00.2600.0" name=GENERATOR>
<link href="${pageContext.request.contextPath}/css/style.css"
    type=text/css rel=stylesheet>
<link href="${pageContext.request.contextPath}/css/boot-crm.css"
    type=text/css rel=stylesheet>
<script
    src="${pageContext.request.contextPath}/js/jquery-1.11.3.min.js">
</script>
<script>
// 判断是登录账号和密码是否为空
function check(){
    var usercode = $("#usercode").val();
    var password = $("#password").val();
    if(usercode=="" || password==""){
    $("#message").text("账号或密码不能为空! ");
       return false;
    }
    return true;
}
</script>
</head>
<body leftMargin=0 topMargin=0 marginwidth="0" marginheight="0"
    background="${pageContext.request.contextPath}/images/rightbg.jpg">
<div ALIGN="center">
<table border="0" width="1140px" cellspacing="0" cellpadding="0"
                                                    id="table1">
    <tr>
        <td height="93"></td>
        <td></td>
    </tr>
    <tr>
  <td background="${pageContext.request.contextPath}/images/rights.jpg"
        width="740" height="412">
  </td>
  <td class="login_msg" width="400" align="center">
    <!-- margin:0px auto; 控制当前标签居中 -->
    <fieldset style="width: auto; margin: 0px auto;">
      <legend>
        <font style="font-size:15px" face="宋体">
            欢迎使用 BOOT 客户管理系统
        </font>
```

```
49          </legend>
50          <font color="red">
51              <%-- 提示信息--%>
52              <span id="message">${msg}</span>
53          </font>
54          <%-- 提交后的位置：/WEB-INF/jsp/customer.jsp--%>
55          <form action="${pageContext.request.contextPath }/login.action"
56                                  method="post" onsubmit="return check()">
57                           <br /><br />
58          账 号：<input id="usercode" type="text" name="usercode" />
59          <br /><br />
60          密 码：<input id="password" type="password" name="password" />
61          <br /><br />
62                 
63          <center><input type="submit" value="登录" /></center>
64          </form>
65      </fieldset>
66      </td>
67      </tr>
68  </table>
69  </div>
70  </body>
71  </html>
```

在文件 18-12 中，核心代码是用户登录操作的 form 表单，该表单在提交时会通过 check() 方法检查账户或密码是否为空，如果为空，则通过标签提示用户“账号或密码不能为空！”；如果账号和密码都已填写，则将表单提交到以“/login.action”结尾的请求中。

6. 启动项目，测试登录

将项目发布到 Tomcat 服务器并启动，成功访问登录页面后，即可输入账号和密码登录系统。在执行登录操作之前，先查看一下数据库中 sys_user 表中的数据，如图 18-6 所示。

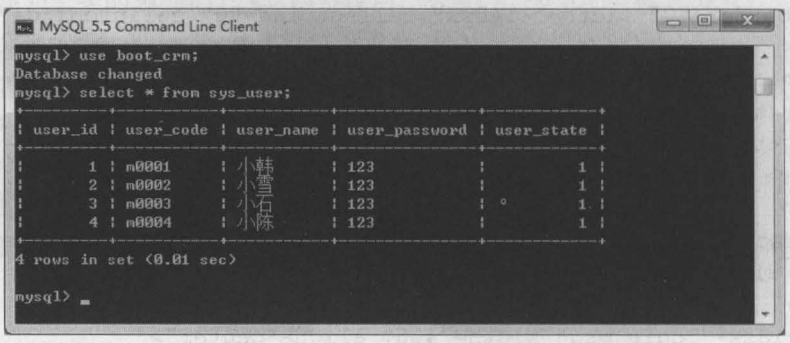


图18-6　sys_user表信息

从图 18-6 可以看出，表 sys_user 中包含 4 个账号（user_code）以 m 开头的用户信息。此时在登录页面中输入账号“m0001”和密码“123”，单击“登录”按钮后，浏览器的显示结果如图 18-7 所示。

图18–7　客户管理页面

从图 18–7 可以看出，系统已经成功进入客户管理页面，这说明系统登录成功。此时由于项目中并没有实现客户查询功能，所以客户列表中没有任何数据。

18.4.2　实现登录验证

虽然在 18.4.1 节中已经实现了用户登录功能，但是此功能还并不完善。假设在其他控制器类中也包含一个访问客户管理页面的方法，那么用户完全可以绕过登录步骤，而直接通过访问该方法的方式进入客户管理页面。

为了验证上述内容，我们可以在用户控制器类 UserController 中编写一个跳转到客户管理页面的方法，其代码如下所示。

```
/**
 * 模拟其他类中跳转到客户管理页面的方法
 */
@RequestMapping(value = "/toCustomer.action")
public String toCustomer() {
    return "customer";
}
```

此时，如果通过浏览器访问地址 http://localhost:8080/boot–crm/toCustomer.action，浏览器就会直接显示客户管理页面，如图 18–8 所示。

显然，让未登录的用户直接访问到客户管理页面，是十分不安全的。为了避免此种情况的发生，并提升系统的安全性，我们可以创建一个登录拦截器来拦截所有请求。只有已登录用户的请

求才能够通过，而对于未登录用户的请求，系统会将请求转发到登录页面，并提示用户登录，其执行流程如图 18-9 所示。

图18-8　客户管理页面

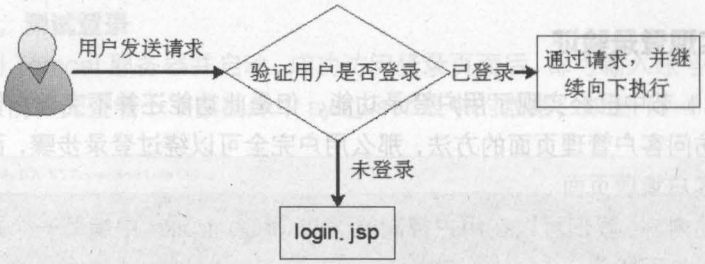


图18-9　登录验证

实现用户登录验证的具体过程如下。

1. 创建登录拦截器类

在 src 目录下，创建一个 com.itheima.core.interceptor 包，并在包中创建登录拦截器类 LoginInterceptor，来实现用户登录的拦截功能，编辑后如文件 18-13 所示。

文件 18-13　LoginInterceptor.java

```
1  package com.itheima.core.interceptor;
2  import javax.servlet.http.HttpServletRequest;
3  import javax.servlet.http.HttpServletResponse;
4  import javax.servlet.http.HttpSession;
5  import org.springframework.web.servlet.HandlerInterceptor;
```

```
import org.springframework.web.servlet.ModelAndView;
import com.itheima.core.po.User;
/**
 * 登录拦截器
 */
public class LoginInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request,
            HttpServletResponse response, Object handler)
            throws Exception {
        // 获取请求的 URL
        String url = request.getRequestURI();
        // URL:除了登录请求外，其他的 URL 都进行拦截控制
        if (url.indexOf("/login.action") >= 0) {
            return true;
        }
        // 获取 Session
        HttpSession session = request.getSession();
        User user = (User) session.getAttribute("USER_SESSION");
        // 判断 Session 中是否有用户数据，如果有，则返回 true,继续向下执行
        if (user != null) {
            return true;
        }
        // 不符合条件的给出提示信息，并转发到登录页面
        request.setAttribute("msg", "您还没有登录，请先登录！");
        request.getRequestDispatcher("/WEB-INF/jsp/login.jsp")
                                        .forward(request, response);
        return false;
    }
    @Override
    public void postHandle(HttpServletRequest request,
            HttpServletResponse response, Object handler,
            ModelAndView modelAndView) throws Exception {
    }
    @Override
    public void afterCompletion(HttpServletRequest request,
            HttpServletResponse response, Object handler, Exception ex)
            throws Exception {
    }
}
```

在文件 18-13 的 preHandle()方法中，首先获取了用户 URL 请求，然后通过请求来判断是否为用户登录操作，只有对用户登录的请求才不进行拦截。接下来获取了 Session 对象，并获取 Session 中的用户信息。如果 Session 中的用户信息不为空，则表示用户已经登录，拦截器将放行；如果 Session 中的用户信息为空，则表示用户未登录，系统会转发到登录页面，并提示用户登录。

2. 配置拦截器

在 springmvc-config.xml 文件中，配置登录拦截器信息，其配置代码如下。

```
<!-- 配置拦截器 -->
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/**" />
        <bean class="com.itheima.core.interceptor.LoginInterceptor" />
    </mvc:interceptor>
</mvc:interceptors>
```

上述配置代码会将所有的用户请求都交由登录拦截器来处理。至此，登录拦截器的实现工作就已经完成。

发布项目并启动 Tomcat 服务器后，再次通过浏览器访问地址 http://localhost:8080/boot-crm/toCustomer.action 时，浏览器的显示结果如图 18-10 所示。

图18-10　登录页面

从图 18-10 可以看出，未登录的用户在执行访问客户管理页面方法后，并没有成功跳转到客户管理页面，而是转发到了系统登录页面，同时在页面的登录窗口中也给出了提示信息。这也就说明用户登录验证功能已成功实现。

18.4.3　退出登录

用户登录模块中还包含一个功能——退出登录。成功登录后的用户会跳转到客户管理页面，

并且在页面中会显示已登录的用户名称，如图 18-11 所示。

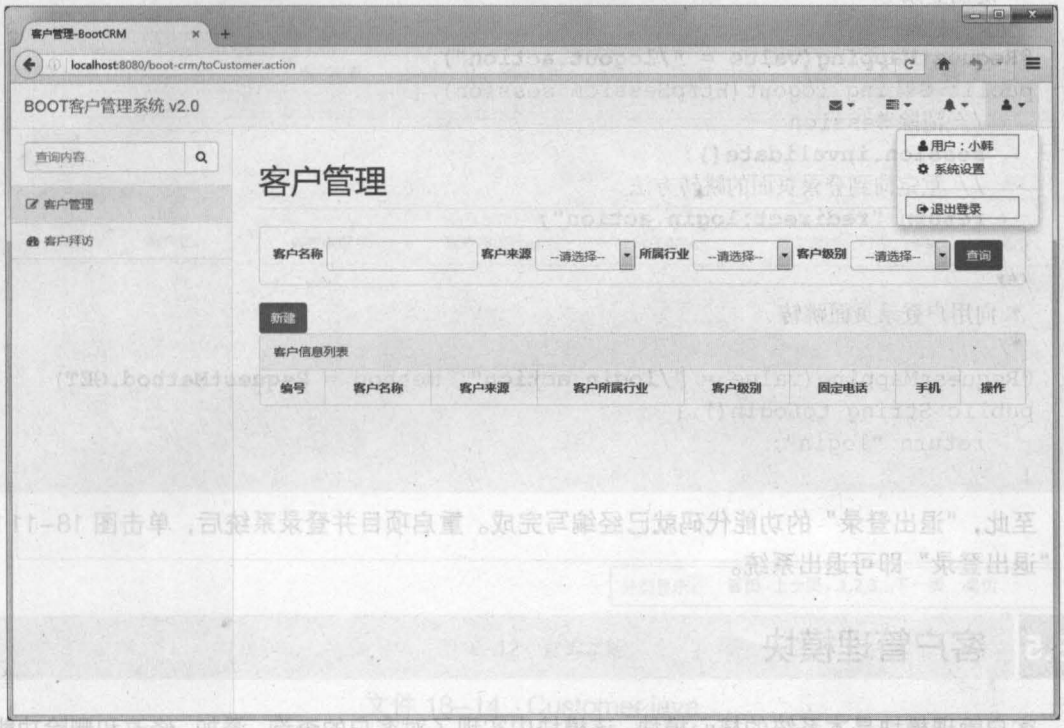


图18-11 客户管理页面

从图 18-11 可以看出，页面的右上角中已经显示了登录用户“小韩”，并且弹出列表框最下方为“退出登录”。那么要如何实现“退出登录”功能呢?

在 customer.jsp 页面中，图 18-11 中弹出列表框的实现代码如下。

```
<ul class="dropdown-menu dropdown-user">
    <li><a href="#"><i class="fa fa-user fa-fw"></i>
            用户：${USER_SESSION.user_name}
        </a>
    </li>
    <li><a href="#"><i class="fa fa-gear fa-fw"></i> 系统设置</a></li>
    <li class="divider"></li>
    <li>
        <a href="${pageContext.request.contextPath }/logout.action">
            <i class="fa fa-sign-out fa-fw"></i>退出登录
        </a>
    </li>
</ul>
```

从上述代码中可以看出，显示的登录用户名称是通过 EL 表达式从 Session 中获取的，而单击“退出登录”链接时，会提交一个以“/logout.action”结尾的请求。

为了完成退出登录功能，我们需要在用户控制器类中编写一个退出登录的方法。在方法执行时，需要清除 Session 中的用户信息，并且在退出登录后，系统要返回到登录页面。因此，需要在用户控制器类 UserController 中编写退出登录和返回到登录页面的方法，这两个方法的实现代码如下。

```
/**
 * 退出登录
 */
@RequestMapping(value = "/logout.action")
public String logout(HttpSession session) {
    // 清除 Session
    session.invalidate();
    // 重定向到登录页面的跳转方法
    return "redirect:login.action";
}
/**
 * 向用户登录页面跳转
 */
@RequestMapping(value = "/login.action", method = RequestMethod.GET)
public String toLogin() {
    return "login";
}
```

至此，“退出登录”的功能代码就已经编写完成。重启项目并登录系统后，单击图 18-11 中的“退出登录”即可退出系统。

18.5 客户管理模块

客户管理模块是本系统的核心模块，该模块中实现了对客户的查询、添加、修改和删除功能。在接下来的几个小节中，将对这几个功能的实现进行详细讲解。

18.5.1 查询客户

在实际应用中，无论是企业级项目，还是互联网项目，使用最多的一定是查询操作。不管是在列表中展示所有数据的操作，还是对单个数据的修改或者删除操作，都需要先查询并展示出数据库中的数据。

查询操作通常可以分为按条件查询和查询所有，但在实际使用时，我们可以将这两种查询编写在一个方法中使用，即当有条件时，就按照条件查询；当没有条件时，就查询所有。同时，由于数据库中的数据可能有很多，如果让这些数据在一个页面中全部显示出来，势必会使页面数据的可读性变得很差，所以我们还需要考虑将这些数据进行分页查询显示。

综合上述分析以及客户页面的显示功能，BOOT 客户管理系统的查询功能需要实现的功能如图 18-12 所示。

从图 18-12 可以看出，客户管理模块中的查询可分为按照条件查询和分页查询，这两种查询操作所查询出的数据都会显示在客户信息列表中。如果未选择任何条件，那么客户信息列表将分页查询显示出所有数据。我们要如何实现客户的条件查询和分页查询呢？下面将对客户管理中的查询功能实现进行详细讲解，具体步骤如下。

1. 创建持久化类

在 com.itheima.core.po 包中，创建客户持久化类和数据字典持久化类，编辑后如文件 18-14 和文件 18-15 所示。

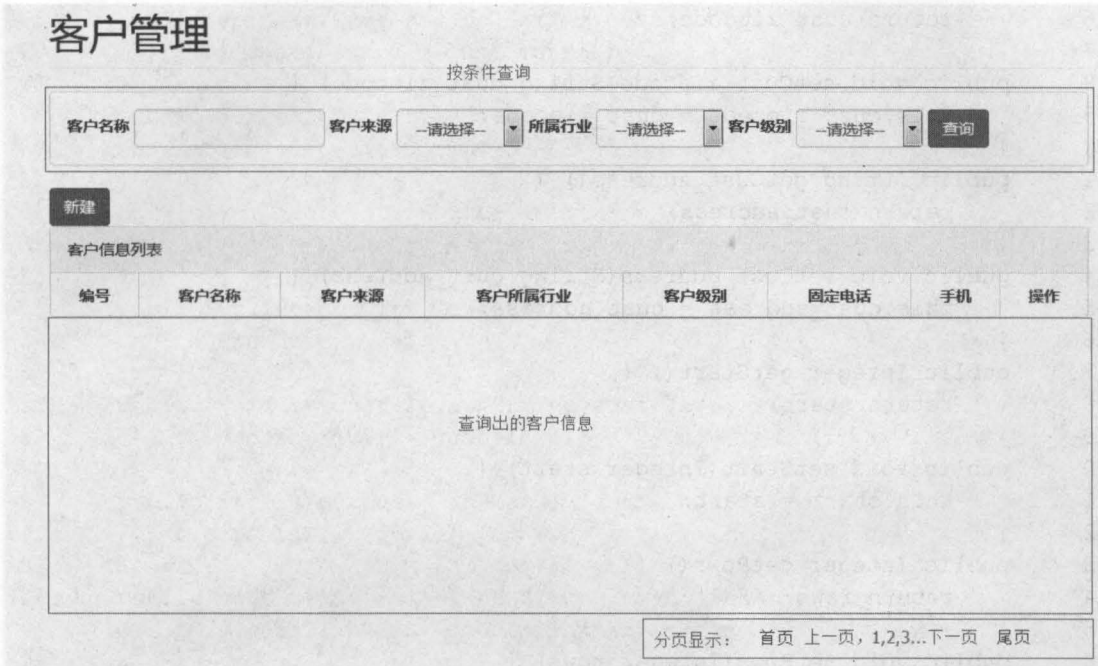


图18-12 查询功能

文件 18-14 Customer.java

```
1  package com.itheima.core.po;
2  import java.io.Serializable;
3  import java.util.Date;
4  /**
5   * 客户持久化类
6   */
7  public class Customer implements Serializable {
8      private static final long serialVersionUID = 1L;
9      private Integer cust_id;          // 客户编号
10     private String cust_name;         // 客户名称
11     private Integer cust_user_id;     // 负责人 id
12     private Integer cust_create_id;// 创建人 id
13     private String cust_source;       // 客户信息来源
14     private String cust_industry;     // 客户所属行业
15     private String cust_level;        // 客户级别
16     private String cust_linkman;      // 联系人
17     private String cust_phone;        // 固定电话
18     private String cust_mobile;       // 移动电话
19     private String cust_zipcode;      // 邮政编码
20     private String cust_address;      // 联系地址
21     private Date cust_createtime;     // 创建时间
22     private Integer start;            // 起始行
23     private Integer rows;             // 所取行数
24
25     public String getCust_zipcode() {
```

```
26          return cust_zipcode;
27      }
28      public void setCust_zipcode(String cust_zipcode) {
29          this.cust_zipcode = cust_zipcode;
30      }
31      public String getCust_address() {
32          return cust_address;
33      }
34      public void setCust_address(String cust_address) {
35          this.cust_address = cust_address;
36      }
37      public Integer getStart() {
38          return start;
39      }
40      public void setStart(Integer start) {
41          this.start = start;
42      }
43      public Integer getRows() {
44          return rows;
45      }
46      public void setRows(Integer rows) {
47          this.rows = rows;
48      }
49      public Integer getCust_id() {
50          return cust_id;
51      }
52      public void setCust_id(Integer cust_id) {
53          this.cust_id = cust_id;
54      }
55      public String getCust_name() {
56          return cust_name;
57      }
58      public void setCust_name(String cust_name) {
59          this.cust_name = cust_name;
60      }
61      public Integer getCust_user_id() {
62          return cust_user_id;
63      }
64      public void setCust_user_id(Integer cust_user_id) {
65          this.cust_user_id = cust_user_id;
66      }
67      public Integer getCust_create_id() {
68          return cust_create_id;
69      }
70      public void setCust_create_id(Integer cust_create_id) {
71          this.cust_create_id = cust_create_id;
72      }
73      public String getCust_source() {
74          return cust_source;
75      }
```

```
76      public void setCust_source(String cust_source) {
77          this.cust_source = cust_source;
78      }
79      public String getCust_industry() {
80          return cust_industry;
81      }
82      public void setCust_industry(String cust_industry) {
83          this.cust_industry = cust_industry;
84      }
85      public String getCust_level() {
86          return cust_level;
87      }
88      public void setCust_level(String cust_level) {
89          this.cust_level = cust_level;
90      }
91      public String getCust_linkman() {
92          return cust_linkman;
93      }
94      public void setCust_linkman(String cust_linkman) {
95          this.cust_linkman = cust_linkman;
96      }
97      public String getCust_phone() {
98          return cust_phone;
99      }
100     public void setCust_phone(String cust_phone) {
101         this.cust_phone = cust_phone;
102     }
103     public String getCust_mobile() {
104         return cust_mobile;
105     }
106     public void setCust_mobile(String cust_mobile) {
107         this.cust_mobile = cust_mobile;
108     }
109     public Date getCust_createtime() {
110         return cust_createtime;
111     }
112     public void setCust_createtime(Date cust_createtime) {
113         this.cust_createtime = cust_createtime;
114     }
115 }
```

在文件 18-14 中，声明了与客户数据表对应的属性并定义了各个属性的 getter/setter 方法。需要注意的是，属性中的 star 和 rows 用于执行分页操作，其中 star 表示分页操作中的起始行，而 rows 则表示分页中所选取的行数。

文件 18-15　BaseDict.java

```
1  package com.itheima.core.po;
2  import java.io.Serializable;
3  /**
4   * 数据字典持久化类
```

```
5  */
6  public class BaseDict implements Serializable {
7      private static final long serialVersionUID = 1L;
8      private String dict_id;          // 数据字典 id
9      private String dict_type_code; // 数据字典类别代码
10     private String dict_type_name; // 数据字典类别名称
11     private String dict_item_name; // 数据字典项目名称
12     private String dict_item_code; // 数据字典项目代码
13     private Integer dict_sort;       // 排序字段
14     private String dict_enable;      // 是否可用
15     private String dict_memo;        // 备注
16
17     public String getDict_id() {
18         return dict_id;
19     }
20     public void setDict_id(String dict_id) {
21         this.dict_id = dict_id;
22     }
23     public String getDict_type_code() {
24         return dict_type_code;
25     }
26     public void setDict_type_code(String dict_type_code) {
27         this.dict_type_code = dict_type_code;
28     }
29     public String getDict_type_name() {
30         return dict_type_name;
31     }
32     public void setDict_type_name(String dict_type_name) {
33         this.dict_type_name = dict_type_name;
34     }
35     public String getDict_item_name() {
36         return dict_item_name;
37     }
38     public void setDict_item_name(String dict_item_name) {
39         this.dict_item_name = dict_item_name;
40     }
41     public String getDict_item_code() {
42         return dict_item_code;
43     }
44     public void setDict_item_code(String dict_item_code) {
45         this.dict_item_code = dict_item_code;
46     }
47     public Integer getDict_sort() {
48         return dict_sort;
49     }
50     public void setDict_sort(Integer dict_sort) {
51         this.dict_sort = dict_sort;
52     }
53     public String getDict_enable() {
54         return dict_enable;
```

```
    }
    public void setDict_enable(String dict_enable) {
        this.dict_enable = dict_enable;
    }
    public String getDict_memo() {
        return dict_memo;
    }
    public void setDict_memo(String dict_memo) {
        this.dict_memo = dict_memo;
    }
}
```

在文件 18-15 中，声明了与数据字典表对应的属性并定义了各个属性的 getter/setter 方法。

2. 实现 DAO 层

（1）创建客户 DAO 层接口和映射文件。在 com.itheima.core.dao 包中，创建一个 CustomerDao 接口，并在接口中编写查询客户列表和客户总数的方法，然后创建一个与接口同名的映射文件，如文件 18-16 和文件 18-17 所示。

文件 18-16 CustomerDao.java

```
package com.itheima.core.dao;
import java.util.List;
import com.itheima.core.po.Customer;
/**
 * Customer 接口
 */
public interface CustomerDao {
    // 客户列表
    public List<Customer> selectCustomerList(Customer customer);
    // 客户数
    public Integer selectCustomerListCount(Customer customer);
}
```

文件 18-17 CustomerDao.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
          "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="com.itheima.core.dao.CustomerDao">
    <!--SQL 片段 -->
    <sql id="selectCustomerListWhere">
        <where>
            <if test="cust_name != null" >
                cust_name like "%"#{cust_name}"%"
            </if>
            <if test="cust_source != null" >
             and cust_source = #{cust_source}
            </if>
            <if test="cust_industry != null" >
             and cust_industry = #{cust_industry}
            </if>
            <if test="cust_level != null" >
```

```
18            and cust_level = #{cust_level}
19          </if>
20        </where>
21    </sql>
22    <!-- 查询客户列表 -->
23    <select id="selectCustomerList" parameterType="customer"
24                                    resultType="customer">
25        SELECT
26            cust_id,
27            cust_name,
28            cust_user_id,
29            cust_create_id,
30            b.dict_item_name cust_source,
31            c.dict_item_name cust_industry,
32            d.dict_item_name cust_level,
33            cust_linkman,
34            cust_phone,
35            cust_mobile,
36            cust_createtime
37        FROM
38            customer a
39        LEFT JOIN (
40            SELECT
41                dict_id,
42                dict_item_name
43            FROM
44                base_dict
45            WHERE
46                dict_type_code = '002'
47        ) b ON a.cust_source = b.dict_id
48        LEFT JOIN (
49            SELECT
50                dict_id,
51                dict_item_name
52            FROM
53                base_dict
54            WHERE
55                dict_type_code = '001'
56        ) c ON a.cust_industry = c.dict_id
57        LEFT JOIN (
58            SELECT
59                dict_id,
60                dict_item_name
61            FROM
62                base_dict
63            WHERE
64                dict_type_code = '006'
65        ) d ON a.cust_level = d.dict_id
66        <include refid="selectCustomerListWhere"/>
67        <!-- 执行分页查询 -->
```

```
68          <if test="start !=null and rows != null">
69              limit #{start},#{rows}
70          </if>
71      </select>
72      <!-- 查询客户总数 -->
73      <select id="selectCustomerListCount" parameterType="customer"
74                                           resultType="Integer">
75          select count(*) from customer
76          <include refid="selectCustomerListWhere"/>
77      </select>
78  </mapper>
```

在文件 18-17 中，首先编写了一个 SQL 片段来作为映射查询客户信息的条件，然后编写了查询所有客户的映射查询方法。在方法的 SQL 中，分别通过左外连接的方式从数据字典表 base_dict 中的类别代码字段查询出了相应的类别信息，同时通过 limit 来实现数据的分页查询。最后编写了一个查询客户总数的映射查询语句用于分页使用。

（2）创建数据字典 DAO 层接口和映射文件。在 com.itheima.core.dao 包中，创建一个 BaseDictDao 接口，并在接口中编写根据类别代码查询数据字典的方法，然后创建一个与接口同名的映射文件，如文件 18-18 和文件 18-19 所示

文件 18-18 BaseDictDao.java

```
1  package com.itheima.core.dao;
2  import java.util.List;
3  import com.itheima.core.po.BaseDict;
4  /**
5   * 数据字典
6   */
7  public interface BaseDictDao {
8      // 根据类别代码查询数据字典
9      public List<BaseDict> selectBaseDictByTypeCode(String typecode);
10 }
```

文件 18-19 BaseDictDao.xml

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
3                  "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
4  <mapper namespace="com.itheima.core.dao.BaseDictDao" >
5    <!-- 根据类别代码查询 -->
6    <select id="selectBaseDictByTypeCode" resultType="baseDict"
7                                          parameterType="String" >
8      select * from base_dict where dict_type_code = #{typecode}
9    </select>
10 </mapper>
```

3. 实现 Service 层

（1）引入分页标签类。在 src 目录下，创建一个 com.itheima.common.utils 包，在包中引入分页时使用的标签类文件 Page.java 和 NavigationTag.java，这两个文件可直接从源代码中获取，其具体实现代码如文件 18-20 和文件 18-21 所示。

文件 18-20　Page.java

```
1  package com.itheima.common.utils;
2  import java.util.List;
3  public class Page<T> {
4      private int total;    // 总条数
5      private int page;     // 当前页
6      private int size;     // 每页数
7     private List<T> rows; // 结果集
8      public int getTotal() {
9          return total;
10     }
11     public void setTotal(int total) {
12         this.total = total;
13     }
14     public int getPage() {
15         return page;
16     }
17     public void setPage(int page) {
18         this.page = page;
19     }
20     public int getSize() {
21         return size;
22     }
23     public void setSize(int size) {
24         this.size = size;
25     }
26     public List<T> getRows() {
27         return rows;
28     }
29     public void setRows(List<T> rows) {
30         this.rows = rows;
31     }
32 }
```

文件 18-21　NavigationTag.java

```
1  package com.itheima.common.utils;
2  package com.itheima.common.utils;
3  import java.io.IOException;
4  import java.util.Map;
5  import javax.servlet.http.HttpServletRequest;
6  import javax.servlet.jsp.JspException;
7  import javax.servlet.jsp.JspWriter;
8  import javax.servlet.jsp.tagext.TagSupport;
9  /**
10  * 显示格式：首页  上一页 1 2 3 4 5 ... 下一页 尾页
11  */
12 public class NavigationTag extends TagSupport {
13     static final long serialVersionUID = 2372405317744358833L;
14     /**
```

```
15      * request 中用于保存 Page<E> 对象的变量名,默认为 “page”
16      */
17     private String bean = "page";
18     /**
19      * 分页跳转的 url 地址,此属性必需
20      */
21     private String url = null;
22     /**
23      * 显示页码数量
24      */
25     private int number = 5;
26
27     @Override
28     public int doStartTag() throws JspException {
29         JspWriter writer = pageContext.getOut();
30         HttpServletRequest request =
31                 (HttpServletRequest) pageContext.getRequest();
32         Page page = (Page) request.getAttribute(bean);
33         if (page == null)
34             return SKIP_BODY;
35         url = resolveUrl(url, pageContext);
36         try {
37             // 计算总页数
38             int pageCount = page.getTotal() / page.getSize();
39             if (page.getTotal() % page.getSize() > 0) {
40                 pageCount++;
41             }
42             writer.print("<nav><ul class=\"pagination\">");
43             //首页链接路径
44             String homeUrl = append(url, "page", 1);
45             //末页链接路径
46             String backUrl = append(url, "page", pageCount);
47             // 显示“上一页”按钮
48             if (page.getPage() > 1) {
49                 String preUrl = append(url, "page", page.getPage() - 1);
50                 preUrl = append(preUrl, "rows", page.getSize());
51                 writer.print("<li><a href=\"" + homeUrl + "\">"
52                         + "首页</a></li>");
53                 writer.print("<li><a href=\"" + preUrl + "\">"
54                         + "上一页</a></li>");
55             } else {
56                 writer.print("<li class=\"disabled\"><a href=\"#\">"
57                         + "首页 </a></li>");
58                 writer.print("<li class=\"disabled\"><a href=\"#\">"
59                         + "上一页 </a></li>");
60             }
61             /*
62              *  显示当前页码的前两页码和后两页码
63              * 若1 则 1 2 3 4 5, 若2 则 1 2 3 4 5, 若3 则1 2 3 4 5,
64              * 若4 则 2 3 4 5 6 ,若10 则 8 9 10 11 12
```

```
65              *  int indexPage=(page.getPage()-2>0)?page.getPage()-2:1;
66              */
67             int indexPage =1;
68             if(page.getPage() - 2 <=0){
69                 indexPage=1;
70             }else if(pageCount-page.getPage() <=2){
71                 indexPage=pageCount-4;
72             }else{
73                 indexPage= page.getPage() - 2;
74             }
75  for (int i = 1; i <= number && indexPage <= pageCount;indexPage++,i++){
76                 if (indexPage == page.getPage()) {
77                     writer.print("<li class=\"active\"><a href=\"#\">"
78     + indexPage + "<span class=\"sr-only\">(current)</span></a></li>");
79                     continue;
80                 }
81                 String pageUrl = append(url, "page", indexPage);
82                 pageUrl = append(pageUrl, "rows", page.getSize());
83                 writer.print("<li><a href=\"" + pageUrl + "\">"
84                 + indexPage + "</a></li>");
85             }
86             // 显示“下一页”按钮
87             if (page.getPage() < pageCount) {
88                 String nextUrl = append(url, "page", page.getPage() + 1);
89                 nextUrl = append(nextUrl, "rows", page.getSize());
90                 writer.print("<li><a href=\"" + nextUrl + "\">"
91                             + "下一页</a></li>");
92                 writer.print("<li><a href=\"" + backUrl + "\">"
93                             + "尾页</a></li>");
94             } else {
95                 writer.print("<li class=\"disabled\"><a href=\"#\">"
96                             + "下一页</a></li>");
97                 writer.print("<li class=\"disabled\"><a href=\"#\">"
98                             + "尾页</a></li>");
99             }
100            writer.print("</nav>");
101        } catch (IOException e) {
102            e.printStackTrace();
103        }
104        return SKIP_BODY;
105    }
106    private String append(String url, String key, int value) {
107        return append(url, key, String.valueOf(value));
108    }
109    /**
110     * 为url 参加参数对儿
111     */
112    private String append(String url, String key, String value) {
113        if (url == null || url.trim().length() == 0) {
114            return "";
```

```
115         }
116         if (url.indexOf("?") == -1) {
117             url = url + "?" + key + "=" + value;
118         } else {
119             if (url.endsWith("?")) {
120                 url = url + key + "=" + value;
121             } else {
122                 url = url + "&amp;" + key + "=" + value;
123             }
124         }
125         return url;
126     }
127     /**
128      * 为 url 添加翻页请求参数
129      */
130     private String resolveUrl(String url,
131         javax.servlet.jsp.PageContext pageContext) throws JspException {
132         Map params = pageContext.getRequest().getParameterMap();
133         for (Object key : params.keySet()) {
134             if ("page".equals(key) || "rows".equals(key)){
135                 continue;
136             }
137             Object value = params.get(key);
138             if (value == null){
139                 continue;
140             }
141             if (value.getClass().isArray()) {
142                 url = append(url, key.toString(), ((String[]) value)[0]);
143             } else if (value instanceof String) {
144                 url = append(url, key.toString(), value.toString());
145             }
146         }
147         return url;
148     }
149
150     public String getBean() {
151         return bean;
152     }
153
154     public void setBean(String bean) {
155         this.bean = bean;
156     }
157
158     public String getUrl() {
159         return url;
160     }
161
162     public void setUrl(String url) {
163         this.url = url;
164     }
```

```
165
166     public void setNumber(int number) {
167         this.number = number;
168     }
169}
```

（2）创建数据字典及客户的 Service 层接口。在 com.itheima.core.service 包中创建一个名称为 BaseDictService 和 CustomerService 的接口，编辑后如文件 18-22 和文件 18-23 所示。

文件 18-22　BaseDictService.java

```
1  package com.itheima.core.service;
2  import java.util.List;
3  import com.itheima.core.po.BaseDict;
4  /**
5   * 数据字典 Service 接口
6   */
7  public interface BaseDictService {
8      //根据类别代码查询数据字典
9      public List<BaseDict> findBaseDictByTypeCode(String typecode);
10 }
```

文件 18-23　CustomerService.java

```
1  package com.itheima.core.service;
2  import com.itheima.common.utils.Page;
3  import com.itheima.core.po.Customer;
4  public interface CustomerService {
5      // 查询客户列表
6      public Page<Customer> findCustomerList(Integer page, Integer rows,
7                                  String custName,String custSource,
8                                  String custIndustry,String custLevel);
9  }
```

（3）创建数据字典及客户 Service 层接口的实现类。在 com.itheima.core.service.impl 包中分别创建数据字典和客户 Service 层接口的实现类 BaseDictServiceImpl 和 CustomerServiceImpl，编辑后的代码如文件 18-24 和文件 18-25 所示。

文件 18-24　BaseDictServiceImpl.java

```
1  package com.itheima.core.service.impl;
2  import java.util.List;
3  import org.springframework.beans.factory.annotation.Autowired;
4  import org.springframework.stereotype.Service;
5  import com.itheima.core.dao.BaseDictDao;
6  import com.itheima.core.po.BaseDict;
7  import com.itheima.core.service.BaseDictService;
8  /**
9   * 数据字典 Service 接口实现类
10  */
11 @Service("baseDictService")
12 public class BaseDictServiceImpl implements BaseDictService{
13     @Autowired
```

```
14      private BaseDictDao baseDictDao;
15      //根据类别代码查询数据字典
16      public List<BaseDict> findBaseDictByTypeCode(String typecode) {
17          return baseDictDao.selectBaseDictByTypeCode(typecode);
18      }
19  }
```

文件 18-25 CustomerServiceImpl.java

```
1   package com.itheima.core.service.impl;
2   import java.util.List;
3   import org.apache.commons.lang3.StringUtils;
4   import org.springframework.beans.factory.annotation.Autowired;
5   import org.springframework.stereotype.Service;
6   import org.springframework.transaction.annotation.Transactional;
7   import com.itheima.common.utils.Page;
8   import com.itheima.core.dao.CustomerDao;
9   import com.itheima.core.po.Customer;
10  import com.itheima.core.service.CustomerService;
11  /**
12   * 客户管理
13   */
14  @Service("customerService")
15  @Transactional
16  public class CustomerServiceImpl implements CustomerService {
17      // 声明 DAO 属性并注入
18      @Autowired
19      private CustomerDao customerDao;
20      // 客户列表
21      public Page<Customer> findCustomerList(Integer page, Integer rows,
22              String custName,  String custSource,String custIndustry,
23                                                  String custLevel) {
24          // 创建客户对象
25          Customer customer = new Customer();
26          // 判断客户名称
27          if(StringUtils.isNotBlank(custName)){
28              customer.setCust_name(custName);
29          }
30          // 判断客户信息来源
31          if(StringUtils.isNotBlank(custSource)){
32              customer.setCust_source(custSource);
33          }
34          // 判断客户所属行业
35          if(StringUtils.isNotBlank(custIndustry)){
36              customer.setCust_industry(custIndustry);
37          }
38          // 判断客户级别
39          if(StringUtils.isNotBlank(custLevel)){
40              customer.setCust_level(custLevel);
41          }
42          // 当前页
```

```
43          customer.setStart((page-1) * rows) ;
44          // 每页数
45          customer.setRows(rows);
46          // 查询客户列表
47          List<Customer> customers =
48                              customerDao.selectCustomerList(customer);
49          // 查询客户列表总记录数
50          Integer count = customerDao.selectCustomerListCount(customer);
51          // 创建 Page 返回对象
52          Page<Customer> result = new Page<>();
53          result.setPage(page);
54          result.setRows(customers);
55          result.setSize(rows);
56          result.setTotal(count);
57          return result;
58      }
59  }
```

在文件 18-25 的实现方法中，首先创建了客户对象，然后判断条件查询中的客户名称、信息来源、所属行业和客户级别是否为空，只有不为空时，才添加到客户对象中。接下来获取了页面传递过来的当前页和每页数信息，并查询所有的客户信息以及客户总数。最后将查询出的所有信息封装到 Page 对象中并返回。

4. 实现 Controller

在 com.itheima.core.web.controller 包中，创建客户控制器类 CustomerController，编辑后如文件 18-26 所示。

文件 18-26　CustomerController.java

```
1   package com.itheima.core.web.controller;
2   import java.util.List;
3   import org.springframework.beans.factory.annotation.Autowired;
4   import org.springframework.beans.factory.annotation.Value;
5   import org.springframework.stereotype.Controller;
6   import org.springframework.ui.Model;
7   import org.springframework.web.bind.annotation.RequestMapping;
8   import org.springframework.web.bind.annotation.RequestParam;
9   import com.itheima.common.utils.Page;
10  import com.itheima.core.po.BaseDict;
11  import com.itheima.core.po.Customer;
12  import com.itheima.core.service.BaseDictService;
13  import com.itheima.core.service.CustomerService;
14  /**
15   * 客户管理控制器类
16   */
17  @Controller
18  public class CustomerController {
19      // 依赖注入
20      @Autowired
21      private CustomerService customerService;
22      @Autowired
```

```
23      private BaseDictService baseDictService;
24      // 客户来源
25      @Value("${customer.from.type}")
26      private String FROM_TYPE;
27      // 客户所属行业
28      @Value("${customer.industry.type}")
29      private String INDUSTRY_TYPE;
30      // 客户级别
31      @Value("${customer.level.type}")
32      private String LEVEL_TYPE;
33      /**
34       * 客户列表
35       */
36      @RequestMapping(value = "/customer/list.action")
37      public String list(@RequestParam(defaultValue="1")Integer page,
38              @RequestParam(defaultValue="10")Integer rows,
39              String custName, String custSource, String custIndustry,
40              String custLevel, Model model) {
41          // 条件查询所有客户
42          Page<Customer> customers = customerService
43                  .findCustomerList(page, rows, custName,
44                                  custSource, custIndustry,custLevel);
45          model.addAttribute("page", customers);
46          // 客户来源
47          List<BaseDict> fromType = baseDictService
48                  .findBaseDictByTypeCode(FROM_TYPE);
49          // 客户所属行业
50          List<BaseDict> industryType = baseDictService
51                  .findBaseDictByTypeCode(INDUSTRY_TYPE);
52          // 客户级别
53          List<BaseDict> levelType = baseDictService
54                  .findBaseDictByTypeCode(LEVEL_TYPE);
55          // 添加参数
56          model.addAttribute("fromType", fromType);
57          model.addAttribute("industryType", industryType);
58          model.addAttribute("levelType", levelType);
59          model.addAttribute("custName", custName);
60          model.addAttribute("custSource", custSource);
61          model.addAttribute("custIndustry", custIndustry);
62          model.addAttribute("custLevel", custLevel);
63          return "customer";
64      }
65  }
```

在客户控制器类中，首先声明了 customerService 和 baseDictService 属性，并通过 @Autowired 注解将这两个对象注入到本类中；然后分别定义了客户来源、所属行业和客户级别属性，并通过@Value 注解将 resource.properties 文件中的属性值赋给这 3 个属性；最后编写了查询客户列表的方法来执行查询操作，其中第 1 个参数 page 的默认值为 1，表示从第 1 条开始，第 2 个参数的默认值为 10，表示每页显示 10 条数据。

5. 实现页面显示

（1）在 18.3.3 小节准备项目环境时，已经说明了需要引入自定义标签文件。在本项目中，自定义标签文件主要用于实现分页功能，其标签名称为 commons.tld，标签中的实现代码如文件 18-27 所示。

文件 18-27　commons.tld

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE taglib
  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
  "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
    <!-- 指定标签库的版本号 -->
    <tlib-version>2.0</tlib-version>
    <!-- 指定 JSP 的版本号 -->
    <jsp-version>1.2</jsp-version>
    <!-- 指定标签库的名称 -->
    <short-name>common</short-name>
    <!-- 指定标签库的 URI -->
    <uri>http://itheima.com/common/</uri>
    <!-- 指定标签库的显示名称 -->
    <display-name>Common Tag</display-name>
    <!-- 指定标签库的描述 -->
    <description>Common Tag library</description>
    <!-- 注册一个自定义标签 -->
    <tag>
        <!-- 指定注册的自定义标签名称 -->
        <name>page</name>
        <!-- 指定自定义标签的标签处理器类 -->
        <tag-class>com.itheima.common.utils.NavigationTag</tag-class>
        <!-- 指定标签体类型 -->
        <body-content>JSP</body-content>
        <!-- 描述 -->
        <description>create navigation for paging</description>
        <!-- 指定标签中的属性 -->
        <attribute>
            <!-- 指定属性名称 -->
            <name>url</name>
            <!-- 该属性为 true 时表示其指定是属性为必须属性 -->
            <required>true</required>
            <!-- 该属性用于指定能不能使用表达式来动态指定数据，为 true 时表示可以 -->
            <rtexprvalue>true</rtexprvalue>
        </attribute>
        <attribute>
            <name>bean</name>
            <rtexprvalue>true</rtexprvalue>
        </attribute>
        <attribute>
            <name>number</name>
            <rtexprvalue>true</rtexprvalue>
```

```
44          </attribute>
45      </tag>
46  </taglib>
```

在文件 18-27 中，第 13 行代码就是我们在使用自定义标签时引入的 URI，第 23 行代码指定了自定义标签的处理器类，其他内容参见代码注释信息。

小提示

在实际开发时，分页功能通常都会使用通用的工具类，或分页组件来实现，而这些工具类和组件一般不需要开发人员自己编写，只需学会使用即可。所以本书中的分页工具类和上面的分页标签文件读者只需直接引入，并且掌握如何使用，而不需要自己编写。

（2）在 customer.jsp 中，编写条件查询和显示客户列表以及分页查询的代码，具体如文件 18-28 所示。

文件 18-28　customer.jsp

```
1  <%@ page language="java" contentType="text/html; charset=UTF-8"
2      pageEncoding="UTF-8"%>
3  <%@ page trimDirectiveWhitespaces="true"%>
4  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
5  <%@ taglib prefix="itheima" uri="http://itheima.com/common/"%>
6  ...
7  <div id="page-wrapper">
8      <div class="row">
9          <div class="col-lg-12">
10             <h1 class="page-header">客户管理</h1>
11         </div>
12         <!-- /.col-lg-12 -->
13     </div>
14     <!-- /.row -->
15     <div class="panel panel-default">
16       <div class="panel-body">
17       <form class="form-inline" method="get"
18       action="${pageContext.request.contextPath }/customer/list.action">
19               <div class="form-group">
20                  <label for="customerName">客户名称</label>
21                  <input type="text" class="form-control" id="customerName"
22                                  value="${custName }" name="custName" />
23               </div>
24               <div class="form-group">
25                  <label for="customerFrom">客户来源</label>
26                  <select class="form-control" id="customerFrom"
27                          name="custSource">
28                     <option value="">--请选择--</option>
29                     <c:forEach items="${fromType}" var="item">
30                        <option value="${item.dict_id}"
31               <c:if test="${item.dict_id == custSource}"> selected</c:if>>
32                           ${item.dict_item_name }
```

```
                    </option>
                </c:forEach>
            </select>
        </div>
        <div class="form-group">
            <label for="custIndustry">所属行业</label>
<select class="form-control" id="custIndustry"  name="custIndustry">
                <option value="">--请选择--</option>
                <c:forEach items="${industryType}" var="item">
                    <option value="${item.dict_id}"
<c:if test="${item.dict_id == custIndustry}"> selected</c:if>>
                        ${item.dict_item_name }
                    </option>
                </c:forEach>
            </select>
        </div>
        <div class="form-group">
<label for="custLevel">客户级别</label>
<select class="form-control" id="custLevel" name="custLevel">
                <option value="">--请选择--</option>
                <c:forEach items="${levelType}" var="item">
                    <option value="${item.dict_id}"
<c:if test="${item.dict_id == custLevel}"> selected</c:if>>
                        ${item.dict_item_name }
                    </option>
                </c:forEach>
            </select>
        </div>
        <button type="submit" class="btn btn-primary">查询</button>
    </form>
</div>
</div>
<a href="#" class="btn btn-primary" data-toggle="modal"
  data-target="#newCustomerDialog" onclick="clearCustomer()">新建</a>
<div class="row">
    <div class="col-lg-12">
        <div class="panel panel-default">
            <div class="panel-heading">客户信息列表</div>
            <!-- /.panel-heading -->
            <table class="table table-bordered table-striped">
                <thead>
                    <tr>
                        <th>编号</th>
                        <th>客户名称</th>
                        <th>客户来源</th>
                        <th>客户所属行业</th>
                        <th>客户级别</th>
                        <th>固定电话</th>
                        <th>手机</th>
                        <th>操作</th>
```

```
83                  </tr>
84               </thead>
85               <tbody>
86                  <c:forEach items="${page.rows}" var="row">
87                     <tr>
88                        <td>${row.cust_id}</td>
89                        <td>${row.cust_name}</td>
90                        <td>${row.cust_source}</td>
91                        <td>${row.cust_industry}</td>
92                        <td>${row.cust_level}</td>
93                        <td>${row.cust_phone}</td>
94                        <td>${row.cust_mobile}</td>
95                        <td>
96       <a href="#" class="btn btn-primary btn-xs" data-toggle="modal"
97          data-target="#customerEditDialog"
98          onclick="editCustomer(${row.cust_id})">修改</a>
99       <a href="#" class="btn btn-danger btn-xs"
100        onclick="deleteCustomer(${row.cust_id})">删除</a>
101                       </td>
102                    </tr>
103                 </c:forEach>
104              </tbody>
105           </table>
106           <div class="col-md-12 text-right">
107        <itheima:page
108     url="${pageContext.request.contextPath }/customer/list.action" />
109           </div>
110           <!-- /.panel-body -->
111        </div>
112        <!-- /.panel -->
113     </div>
114     <!-- /.col-lg-12 -->
115  </div>
116</div>
117...
```

在上述页面代码中，第 4 行和第 5 行代码引入了 JSTL 标签和自定义的分页标签；第 17～62 行代码是一个条件查询的 form 表单，单击“查询”按钮后，会提交到一个“list.action”请求中；第 72～105 行代码是显示客户信息列表的表格，查询出的客户信息会在此表格中显示；第 107～108 行代码是自定义的分页标签，该标签会根据客户数以及设定的页数数据显示内容。

6. 测试条件查询和分页

发布项目并启动 Tomcat 服务器后，进入客户管理页面，然后单击“查询”按钮即可查询出所有客户信息，并且这些信息都已分页显示，如图 18-13 所示。

选择图 18-13 中的客户来源为“电话营销”后，再次单击“查询”按钮，其查询结果如图 18-14 所示。

图18-13　客户信息列表显示

图18-14　条件查询后的客户信息列表显示

当单击列表下方的页码链接时，列表也会显示出相应页面中的数据，如图 18-15 所示。

图18-15　分页查询后的客户信息列表显示

从图 18-13、图 18-14 和图 18-15 中可以看出，客户管理模块中的查询功能已经实现成功。

细心的读者一定会发现，在进入客户管理页面时，客户信息列表是没有任何显示的，只有单击“查询”按钮后，才会显示出数据。那么有什么办法能够让其进入该页面时就默认显示数据呢?

要实现登录后展示客户信息列表的操作很简单，只需将用户控制器类（UserController）中用户登录方法（login()）内跳转到主页面的语句修改为重定向到主页的跳转方法即可，修改后的语句如下。

```
return "redirect:customer/list.action";
```

这样，登录成功后，客户管理页面将会直接显示出分页后的客户列表信息。

18.5.2　添加客户

在本系统中，添加客户的操作是通过页面弹出窗口实现的，当单击“新建”按钮时，将弹出“新建客户信息”窗口，如图 18-16 所示。

图18-16　新建客户信息窗口

填写完图 18-16 中的所有客户信息后，单击“创建客户”按钮，将执行添加客户的操作。那么此操作具体是如何实现的呢？下面将对系统中的添加客户的功能实现进行详细讲解，具体步骤如下。

1. 实现页面功能代码

在页面中，“新建”按钮链接的实现代码如下。

```
<a href="#" class="btn btn-primary" data-toggle="modal"
   data-target="#newCustomerDialog" onclick="clearCustomer()">新建</a>
```

在上述代码中，data-toggle="modal" 和 data-target="#newCustomerDialog"是 Bootstrap 的模态框代码，当单击“新建”按钮后，会弹出 id 为 newCustomerDialog 的窗口，同时通过 onclick 属性执行 clearCustomer()方法来清除窗口中的所有数据。

在 customer.jsp 中，新建客户模态框的显示代码如文件 18-29 所示。

文件 18-29　customer.jsp

```
...//此处省略页面其他代码
<!-- 创建客户模态框 -->
<div class="modal fade" id="newCustomerDialog" tabindex="-1" role="dialog"
    aria-labelledby="myModalLabel">
<div class="modal-dialog" role="document">
<div class="modal-content">
    <div class="modal-header">
        <button type="button" class="close" data-dismiss="modal"
                aria-label="Close">
            <span aria-hidden="true">&times;</span>
        </button>
        <h4 class="modal-title" id="myModalLabel">新建客户信息</h4>
```

```
13    </div>
14    <div class="modal-body">
15       <form class="form-horizontal" id="new_customer_form">
16          <div class="form-group">
17          <label for="new_customerName" class="col-sm-2 control-label">
18                客户名称
19          </label>
20       <div class="col-sm-10">
21       <input type="text" class="form-control"
22        id="new_customerName" placeholder="客户名称" name="cust_name" />
23          </div>
24          </div>
25          <div class="form-group">
26             <label for="new_customerFrom"
27                    style="float:left;padding:7px 15px 0 27px;">
28                客户来源
29             </label>
30             <div class="col-sm-10">
31                <select class="form-control" id="new_customerFrom"
32                       name="cust_source">
33                   <option value="">--请选择--</option>
34                   <c:forEach items="${fromType}" var="item">
35                      <option value="${item.dict_id}"
36          <c:if test="${item.dict_id == custSource}">selected</c:if>>
37                      ${item.dict_item_name }
38                      </option>
39                   </c:forEach>
40                </select>
41             </div>
42          </div>
43          <div class="form-group">
44             <label for="new_custIndustry"
45          style="float:left;padding:7px 15px 0 27px;">所属行业</label>
46             <div class="col-sm-10">
47                <select    class="form-control" id="new_custIndustry"
48                                              name="cust_industry">
49                   <option value="">--请选择--</option>
50                   <c:forEach items="${industryType}" var="item">
51                      <option value="${item.dict_id}"
52           <c:if test="${item.dict_id == custIndustry}">selected</c:if>>
53                      ${item.dict_item_name }
54                      </option>
55                   </c:forEach>
56                </select>
57             </div>
58          </div>
59          <div class="form-group">
60             <label for="new_custLevel"
61                    style="float:left;padding:7px 15px 0 27px;">客户级别
62             </label>
```

```
63              <div class="col-sm-10">
64                 <select    class="form-control" id="new_custLevel"
65                                                  name="cust_level">
66                    <option value="">--请选择--</option>
67                    <c:forEach items="${levelType}" var="item">
68                     <option value="${item.dict_id}"
69                 <c:if test="${item.dict_id == custLevel}">selected</c:if>>
70                        ${item.dict_item_name }
71                     </option>
72                    </c:forEach>
73                 </select>
74              </div>
75           </div>
76           <div class="form-group">
77              <label for="new_linkMan" class="col-sm-2 control-label">
78                  联系人
79              </label>
80              <div class="col-sm-10">
81                 <input type="text" class="form-control"
82               id="new_linkMan" placeholder="联系人" name="cust_linkman" />
83              </div>
84           </div>
85           <div class="form-group">
86              <label for="new_phone" class="col-sm-2 control-label">
87                  固定电话
88              </label>
89              <div class="col-sm-10">
90                 <input type="text" class="form-control"
91                 id="new_phone" placeholder="固定电话" name="cust_phone" />
92              </div>
93           </div>
94           <div class="form-group">
95              <label for="new_mobile" class="col-sm-2 control-label">
96                  移动电话
97              </label>
98              <div class="col-sm-10">
99                 <input type="text" class="form-control"
100              id="new_mobile" placeholder="移动电话" name="cust_mobile" />
101             </div>
102          </div>
103          <div class="form-group">
104             <label for="new_zipcode" class="col-sm-2 control-label">
105                 邮政编码
106              </label>
107             <div class="col-sm-10">
108                <input type="text" class="form-control"
109             id="new_zipcode" placeholder="邮政编码" name="cust_zipcode" />
110             </div>
111          </div>
112          <div class="form-group">
```

```
113              <label for="new_address" class="col-sm-2 control-label">
114                  联系地址
115              </label>
116              <div class="col-sm-10">
117                 <input type="text" class="form-control"
118              id="new_address" placeholder="联系地址" name="cust_address" />
119              </div>
120           </div>
121       </form>
122    </div>
123    <div class="modal-footer">
124       <button type="button" class="btn btn-default"
125               data-dismiss="modal">关闭</button>
126       <button type="button" class="btn btn-primary"
127               onclick="createCustomer()">创建客户</button>
128    </div>
129</div>
130</div>
131</div>
132  ...
```

在上述代码中，第 15 ~ 121 行代码的 form 表单中的实现代码，即为用户所需要填写的客户信息。

为保证每次单击“新建”按钮后所弹出的模态框内没有任何数据，需要在页面中创建一个 clearCustomer()方法来清空模态框中的内容，clearCustomer()方法的实现代码如下所示。

```
// 清空新建客户窗口中的数据
function clearCustomer() {
    $("#new_customerName").val("");
    $("#new_customerFrom").val("")
    $("#new_custIndustry").val("")
    $("#new_custLevel").val("")
    $("#new_linkMan").val("");
    $("#new_phone").val("");
    $("#new_mobile").val("");
    $("#new_zipcode").val("");
    $("#new_address").val("");
}
```

填写完模态框中的信息后，单击“创建客户”按钮，会执行 createCustomer()方法，该方法的实现代码如下。

```
// 创建客户
function createCustomer() {
$.post("<%=basePath%>customer/create.action",
        $("#new_customer_form").serialize(),function(data){
    if(data =="OK"){
        alert("客户创建成功！");
        window.location.reload();
    }else{
        alert("客户创建失败！");
```

```
            window.location.reload();
        }
    });
}
```

createCustomer()方法会通过 jQuery Ajax 的 POST 请求将 id 为 new_customer_form 的表单序列化，然后提交到以“/create.action”结尾的请求中，如果其返回值为“OK”则表示客户创建成功，否则创建客户失败。

2. 实现 Controller 层方法

在客户控制器类 CustomerController 中编写创建客户的方法，其代码如下所示。

```
/**
 * 创建客户
 */
@RequestMapping("/customer/create.action")
@ResponseBody
public String customerCreate(Customer customer,HttpSession session) {
    // 获取 Session 中的当前用户信息
    User user = (User) session.getAttribute("USER_SESSION");
    // 将当前用户 id 存储在客户对象中
    customer.setCust_create_id(user.getUser_id());
    // 创建 Date 对象
    Date date = new Date();
    // 得到一个 Timestamp 格式的时间，存入 mysql 中的时间格式“yyyy/MM/dd HH:mm:ss”
    Timestamp timeStamp = new Timestamp(date.getTime());
    customer.setCust_createtime(timeStamp);
    // 执行 Service 层中的创建方法，返回的是受影响的行数
    int rows = customerService.createCustomer(customer);
    if(rows > 0){
        return "OK";
    }else{
        return "FAIL";
    }
}
```

在上述方法代码中，首先获取了 Session 中的当前用户信息，然后将当前用户 id 信息添加到 Customer 对象的创建人 id 属性中。接下来创建了 Date 对象，并将格式化的时间信息添加到 Customer 对象的 cust_createtime 属性中。最后执行 Service 层中的 createCustomer()方法，其返回值为数据库中受影响的行数，如果其值大于 0，则表示创建成功，返回“OK”字符串信息，否则返回“FAIL”字符串。

小提示

@Responsebody 注解一般在异步获取数据时使用。在使用@RequestMapping 注解后，方法的返回值通常会被解析为跳转路径（如某个页面或某个方法），而加上@Responsebody 注解后，其返回结果将不会被解析为跳转路径，而是将通过 HttpMessageConverter 转换为指定格式后的结果（如 json、xml 等）直接写入 HTTP Response 对象的 body 中，这样页面中的方法就可以获取其返回值。

3. 实现 Service 层方法

（1）创建接口方法。在 CustomerService 接口中，创建一个 createCustomer()方法，其代码如下所示。

```
public int createCustomer(Customer customer);
```

（2）创建实现类方法。在 CustomerServiceImpl 中，实现 createCustomer()方法，编辑后的代码如下所示。

```
/**
 * 创建客户
 */
@Override
public int createCustomer(Customer customer) {
    return customerDao.createCustomer(customer);
}
```

从上述代码可以看出，createCustomer()方法并没有实现其他功能，而是直接返回了 DAO 层中的 createCustomer()方法。

4. 实现 DAO 层方法

（1）创建接口方法。在 CustomerDao 中，编写创建客户的方法，其代码如下所示。

```
// 创建客户
public int createCustomer(Customer customer);
```

（2）创建映射插入语句。在 CustomerDao.xml 中，编写执行插入操作的映射插入语句，其代码如下所示。

```
<!-- 添加客户 -->
<insert id="createCustomer" parameterType="customer">
    insert into customer(
                cust_name,
                cust_user_id,
                cust_create_id,
                cust_source,
                cust_industry,
                cust_level,
                cust_linkman,
                cust_phone,
                cust_mobile,
                cust_zipcode,
                cust_address,
                cust_createtime
        )
        values(#{cust_name},
               #{cust_user_id},
               #{cust_create_id},
               #{cust_source},
               #{cust_industry},
               #{cust_level},
               #{cust_linkman},
```

```
                #{cust_phone},
                #{cust_mobile},
                #{cust_zipcode},
                #{cust_address},
                #{cust_createtime}
            )
    </insert>
```

5. 添加客户测试

至此，添加客户的实现代码就已经编写完成。发布并启动项目后，进入客户管理页面，单击“新建”按钮，并填写新建客户信息，如图 18-17 所示。

图18-17　添加客户信息

单击图 18-17 中的“创建客户”按钮后，如果程序正确执行，则会弹出“客户创建成功!”的弹出窗口，再次单击“确定”后，浏览器就会刷新当前页面。

要查询所创建的客户是否已创建成功非常简单，只需要在条件查询中查找客户名称为“小程”的客户，如图 18-18 所示。

图18-18　查询客户

从图 18-18 可以看出，新创建的客户“小程”的信息已被正确查询出。至此添加客户的功能就已经成功实现。

18.5.3 修改客户

修改操作与添加操作一样，也是通过页面弹出窗口实现的，当单击页面对应数据的“修改”按钮时，将弹出“修改客户信息”窗口，如图 18-19 所示。

图18-19 修改客户信息窗口

从图 18-19 可以看出，修改客户信息窗口与新建客户信息窗口的显示内容基本相同，但修改客户信息窗口中回显出了需要修改的客户信息。当修改客户信息后，单击“保存修改”按钮，即可执行修改操作。下面就对本系统中修改客户的功能实现进行详细讲解，具体步骤如下。

1. 实现页面功能代码

在页面中，“修改”按钮链接的实现代码如下。

```
<a href="#" class="btn btn-primary btn-xs" data-toggle="modal"
            data-target="#customerEditDialog"
            onclick="editCustomer(${row.cust_id})">修改
</a>
```

与新建方法一样，当单击“修改”按钮后，会弹出 id 为 customerEditDialog 的模态窗口，同时通过执行 onclick 属性的 editCustomer()方法来获取需要修改客户的所有数据。

在 customer.jsp 中，修改客户模态框的显示代码如文件 18-30 所示。

文件 18-30 customer.jsp

```
1  ... //此处省略页面其他代码
2  <!-- 修改客户模态框 -->
3  <div class="modal fade" id="customerEditDialog"
4       tabindex="-1" role="dialog" aria-labelledby="myModalLabel">
```

```
<div class="modal-dialog" role="document">
<div class="modal-content">
    <div class="modal-header">
        <button type="button" class="close"
                data-dismiss="modal" aria-label="Close">
            <span aria-hidden="true">&times;</span>
        </button>
        <h4 class="modal-title" id="myModalLabel">修改客户信息</h4>
    </div>
    <div class="modal-body">
        <form class="form-horizontal" id="edit_customer_form">
            <input type="hidden" id="edit_cust_id" name="cust_id"/>
            <div class="form-group">
                <label for="edit_customerName"
                       class="col-sm-2 control-label">客户名称</label>
                <div class="col-sm-10">
             <input type="text" class="form-control" id="edit_customerName"
                    placeholder="客户名称" name="cust_name" />
                </div>
            </div>
            <div class="form-group">
              <label for="edit_customerFrom"
                     style="float:left;padding:7px 15px 0 27px;">客户来源
              </label>
               <div class="col-sm-10">
                  <select class="form-control" id="edit_customerFrom"
                          name="cust_source">
                     <option value="">--请选择--</option>
                     <c:forEach items="${fromType}" var="item">
                        <option value="${item.dict_id}"
            <c:if test="${item.dict_id == custSource}"> selected</c:if>>
                           ${item.dict_item_name }
                        </option>
                     </c:forEach>
                  </select>
               </div>
            </div>
            <div class="form-group">
               <label for="edit_custIndustry"
                      style="float:left;padding:7px 15px 0 27px;">所属行业
               </label>
               <div class="col-sm-10">
                  <select class="form-control" id="edit_custIndustry"
                          name="cust_industry">
                     <option value="">--请选择--</option>
                     <c:forEach items="${industryType}" var="item">
                        <option value="${item.dict_id}"
                           <c:if test="${item.dict_id == custIndustry}">
                              Selected
                           </c:if>>
```

```
                    ${item.dict_item_name }
                </option>
                </c:forEach>
            </select>
        </div>
    </div>
    <div class="form-group">
        <label for="edit_custLevel"
               style="float:left;padding:7px 15px 0 27px;">客户级别
        </label>
        <div class="col-sm-10">
            <select class="form-control" id="edit_custLevel"
                    name="cust_level">
                <option value="">--请选择--</option>
                <c:forEach items="${levelType}" var="item">
                    <option value="${item.dict_id}"
                        <c:if test="${item.dict_id == custLevel}">
                            Selected
                        </c:if>>
                        ${item.dict_item_name }
                    </option>
                </c:forEach>
            </select>
        </div>
    </div>
    <div class="form-group">
        <label for="edit_linkMan" class="col-sm-2 control-label">
            联系人
        </label>
        <div class="col-sm-10">
          <input type="text" class="form-control" id="edit_linkMan"
                 placeholder="联系人" name="cust_linkman" />
        </div>
    </div>
    <div class="form-group">
        <label for="edit_phone" class="col-sm-2 control-label">
            固定电话
        </label>
        <div class="col-sm-10">
            <input type="text" class="form-control" id="edit_phone"
                   placeholder="固定电话" name="cust_phone" />
        </div>
    </div>
    <div class="form-group">
        <label for="edit_mobile" class="col-sm-2 control-label">
            移动电话
        </label>
        <div class="col-sm-10">
            <input type="text" class="form-control" id="edit_mobile"
                   placeholder="移动电话" name="cust_mobile" />
```

```
105             </div>
106          </div>
107          <div class="form-group">
108             <label for="edit_zipcode" class="col-sm-2 control-label">
109                 邮政编码
110             </label>
111             <div class="col-sm-10">
112               <input type="text" class="form-control" id="edit_zipcode"
113                     placeholder="邮政编码" name="cust_zipcode" />
114             </div>
115          </div>
116          <div class="form-group">
117             <label for="edit_address" class="col-sm-2 control-label">
118                 联系地址
119             </label>
120             <div class="col-sm-10">
121               <input type="text" class="form-control" id="edit_address"
122                     placeholder="联系地址" name="cust_address" />
123             </div>
124          </div>
125       </form>
126    </div>
127    <div class="modal-footer">
128       <button type="button" class="btn btn-default" data-dismiss="modal">
129           关闭
130       </button>
131       <button type="button" class="btn btn-primary"
132             onclick="updateCustomer()">保存修改</button>
133    </div>
134</div>
135</div>
136</div>
137  ...
```

在上述代码中，第 15 ~ 125 行代码中的 form 表单就是修改客户信息的实现代码。

由于在修改客户信息时，需要先获取到该客户的所有信息，并显示到修改信息的窗口内，所以需要在页面中编写一个获取客户信息的方法 editCustomer()，该方法的实现代码如下所示。

```
// 通过 id 获取修改的客户信息
function editCustomer(id) {
   $.ajax({
      type:"get",
      url:"<%=basePath%>customer/getCustomerById.action",
      data:{"id":id},
      success:function(data) {
         $("#edit_cust_id").val(data.cust_id);
         $("#edit_customerName").val(data.cust_name);
         $("#edit_customerFrom").val(data.cust_source)
         $("#edit_custIndustry").val(data.cust_industry)
         $("#edit_custLevel").val(data.cust_level)
```

```
            $("#edit_linkMan").val(data.cust_linkman);
            $("#edit_phone").val(data.cust_phone);
            $("#edit_mobile").val(data.cust_mobile);
            $("#edit_zipcode").val(data.cust_zipcode);
            $("#edit_address").val(data.cust_address);

        }
    });
}
```

上述方法代码使用了 jQuery Ajax 的方式来获取所需要修改的客户信息，获取成功后，会将该客户信息添加到修改客户模态框中的相应位置。

2. 实现 Controller 层方法

在 CustomerController 类中，编写通过 id 获取客户信息和更新客户的方法，其代码如下所示。

```
/**
 * 通过 id 获取客户信息
 */
@RequestMapping("/customer/getCustomerById.action")
@ResponseBody
public Customer getCustomerById(Integer id) {
    Customer customer = customerService.getCustomerById(id);
    return customer;
}
/**
 * 更新客户
 */
@RequestMapping("/customer/update.action")
@ResponseBody
public String customerUpdate(Customer customer) {
    int rows = customerService.updateCustomer(customer);
    if(rows > 0){
        return "OK";
    }else{
        return "FAIL";
    }
}
```

上述两个方法中，都只是调用了 Service 层中的相应方法，并将相应的执行结果返回。

3. 实现 Service 层方法

（1）创建接口方法。在 CustomerService 接口中，创建通过 id 获取客户信息和更新客户的方法，代码如下所示。

```
// 通过 id 查询客户
public Customer getCustomerById(Integer id);
// 更新客户
public int updateCustomer(Customer customer);
```

（2）创建实现类方法。在 CustomerServiceImpl 中，实现接口中的方法，编辑后的代码如下所示。

```
/**
 * 通过 id 查询客户
```

```
     */
    @Override
    public Customer getCustomerById(Integer id) {
        Customer customer = customerDao.getCustomerById(id);
        return customer;
    }
    /**
     * 更新客户
     */
    @Override
    public int updateCustomer(Customer customer) {
        return customerDao.updateCustomer(customer);
    }
```

在上述代码中，getCustomerById()方法返回的是所需要修改的客户对象，而updateCustomer()方法执行后，返回的是数据库中受影响的行数。

4. 实现 DAO 层方法

（1）创建接口方法。在 CustomerDao 接口中，编写通过 id 获取客户信息和更新客户的方法，其代码如下所示。

```
    // 通过 id 查询客户
    public Customer getCustomerById(Integer id);
    // 更新客户信息
    public int updateCustomer(Customer customer);
```

（2）创建映射语句。在 CustomerDao.xml 中，编写执行插入操作的映射插入语句，代码如下所示。

```
    <!-- 根据 id 获取客户信息 -->
    <select id="getCustomerById" parameterType="Integer"
                                 resultType="customer">
        select * from customer where cust_id = #{id}
    </select>
    <!-- 更新客户 -->
    <update id="updateCustomer" parameterType="customer">
        update customer
        <set>
            <if test="cust_name!=null">
                cust_name=#{cust_name},
            </if>
            <if test="cust_user_id!=null">
                cust_user_id=#{cust_user_id},
            </if>
            <if test="cust_create_id!=null">
                cust_create_id=#{cust_create_id},
            </if>
            <if test="cust_source!=null">
                cust_source=#{cust_source},
            </if>
            <if test="cust_industry!=null">
                cust_industry=#{cust_industry},
```

```
        </if>
        <if test="cust_level!=null">
            cust_level=#{cust_level},
        </if>
        <if test="cust_linkman!=null">
            cust_linkman=#{cust_linkman},
        </if>
        <if test="cust_phone!=null">
            cust_phone=#{cust_phone},
        </if>
        <if test="cust_mobile!=null">
            cust_mobile=#{cust_mobile},
        </if>
        <if test="cust_zipcode!=null">
            cust_zipcode=#{cust_zipcode},
        </if>
        <if test="cust_address!=null">
            cust_address=#{cust_address},
        </if>
        <if test="cust_createtime!=null">
            cust_createtime=#{cust_createtime},
        </if>
    </set>
    where cust_id=#{cust_id}
</update>
```

5. 修改客户测试

至此，修改客户的实现代码就已经编写完成。发布并启动项目后，进入客户管理页面，单击列表中编号为 14 客户后面的“修改”按钮，将所属行业修改为“对外贸易”，并将客户级别修改为“VIP 客户”，如图 18-20 所示。

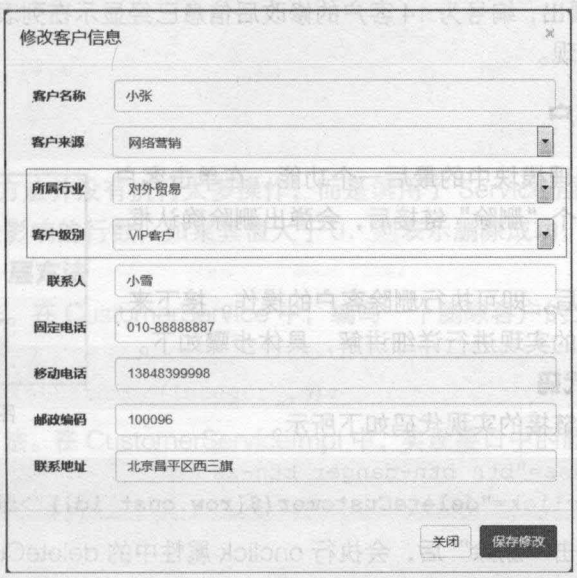


图18-20 修改客户信息窗口

单击图 18-20 中的“保存修改”按钮后，如果程序正确执行，将会出现“客户信息更新成功”提示框，确认后，将回到客户管理列表页面，此时页面中的信息如图 18-21 所示。

图18-21　客户信息列表

从图 18-21 可以看出，编号为 14 客户的修改后信息已经显示在列表中，这也就说明系统的修改客户功能已成功实现。

18.5.4　删除客户

删除客户是客户管理模块中的最后一个功能。在单击客户信息列表中操作列的某个“删除”链接后，会弹出删除确认框，如图 18-22 所示。

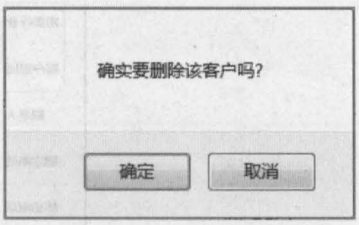


图18-22　删除确认框

单击“确定”按钮后，即可执行删除客户的操作。接下来，本节将对删除客户功能的实现进行详细讲解，具体步骤如下。

1. 实现页面功能代码

页面中，删除客户链接的实现代码如下所示。

```
<a href="#" class="btn btn-danger btn-xs"
            onclick="deleteCustomer(${row.cust_id})">删除</a>
```

上述代码中，当单击“删除”后，会执行 onclick 属性中的 deleteCustomer()方法，该方法中的参数${row.cust_id}会获取当前所在行的客户 id。

在页面中，编写删除客户的方法 deleteCustomer()，其方法代码如下所示。

```
// 删除客户
function deleteCustomer(id) {
    if(confirm('确实要删除该客户吗?')) {
$.post("<%=basePath%>customer/delete.action",{"id":id},function(data){
            if(data =="OK"){
                alert("客户删除成功！");
                window.location.reload();
            }else{
                alert("删除客户失败！");
                window.location.reload();
            }
        });
    }
}
```

执行上述方法时，会通过 jQuery Ajax 的方式发送一个以“/delete.action”结尾的请求，该请求会将所要删除的客户 id 传入后台处理方法中。

2. 实现 Controller 层方法

在 CustomerController 类中，创建一个删除客户的方法 customerDelete()，编辑后的实现代码如下所示。

```
/**
 * 删除客户
 */
@RequestMapping("/customer/delete.action")
@ResponseBody
public String customerDelete(Integer id) {
    int rows = customerService.deleteCustomer(id);
    if(rows > 0){
        return "OK";
    }else{
        return "FAIL";
    }
}
```

customerDelete()方法并没有执行太多操作，而是调用了 Service 层中的 deleteCustomer()方法来获取数据库中受影响的行数，如果其值大于 0，则表示删除成功，否则表示删除失败。

3. 实现 Service 层方法

（1）创建接口方法。在 CustomerService 中，编写一个删除客户的方法，代码如下所示。

```
// 删除客户
public int deleteCustomer(Integer id);
```

（2）创建实现类方法。在 CustomerServiceImpl 中，实现接口中的删除方法，编辑后的代码如下所示。

```
/**
 * 删除客户
 */
```

```
@Override
public int deleteCustomer(Integer id) {
    return customerDao.deleteCustomer(id);
}
```

4. 实现 DAO 层方法

（1）创建接口方法。在 CustomerDao 接口中，编写通过 id 删除客户的方法，代码如下所示。

```
// 删除客户
int deleteCustomer (Integer id);
```

（2）创建映射语句。在 CustomerDao.xml 中编写执行删除操作的映射语句，代码如下所示。

```
<!-- 删除客户 -->
<delete id="deleteCustomer" parameterType="Integer">
    delete from customer where cust_id=#{id}
</delete>
```

5. 删除客户测试

至此，删除客户功能的实现代码就已经编写完成。下面以删除编号为 14 的客户“小张”为例，来测试系统的删除功能。

单击编号为 14 客户所在行的“删除”链接后，会弹出删除确认框，单击“确定”按钮后，页面中会弹出客户删除成功的提示框，如图 18-23 所示。

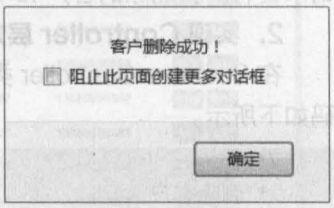


图18-23　删除提示框

确定后，系统会刷新当前页面。此时页面中客户信息列表所显示的数据如图 18-24 所示。

图18-24　客户管理页面

从图 18-24 可以看到，执行删除操作后，编号为 14 的客户“小张”并没有在客户信息列表中显示，这也就说明删除操作执行成功。

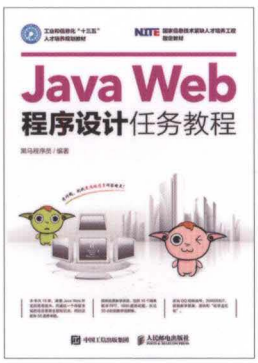
18.6 本章小结

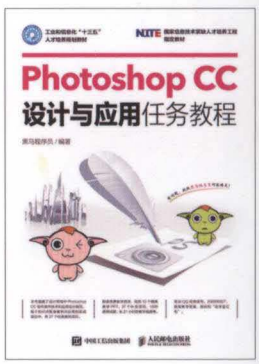
本章主要通过一个 BOOT 客户管理系统来讲解 SSM 框架的实际使用。首先对系统的功能、结构等进行了简单的介绍，然后讲解了系统所使用的数据库表。接下来，详细地讲解了系统的环境搭建工作。最后，讲解了系统用户登录模块和客户管理模块的实现。通过本章的学习，读者可以熟练地掌握 SSM 框架的整合使用，并能熟练地使用 SSM 框架实现系统功能模块的开发工作。本系统是 SSM 框架综合使用的案例，读者一定要多加练习，并熟练编写各个功能模块的实现代码，这样才能将前面所学知识融会贯通。

【思考题】

1. 请简述系统中各个层次的组成和作用。
2. 请简述引入 SQL 文件的过程。

应用型IT套系教材推荐 RECOMMEND APPLICATION IT SERIES TEXTBOOK





人邮教育
www.ryjiaoyu.com

教材服务热线：010-81055256
反馈/投稿/推荐信箱：315@ptpress.com.cn
人民邮电出版社教育服务与资源下载社区：www.ryjiaoyu.com

ISBN 978-7-115-46102-5

ISBN 978-7-115-46102-5
定价：49.80 元

封面设计：董志桢