

Java Web 轻量级开发 面试教程

孟宪福 胡书敏 金华 编著



赢在起点，短时间帮你尽快升级到高级程序员

实用性案例+视频讲解+面试题解析



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Java Web 轻量级开发 面试教程

孟宪福 胡书敏 金华 编著

电子工业出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书围绕着软件公司对高级程序员的平均要求,根据笔者多年的面试和培训经验,构建了 Java Web 方面的高级程序员的进阶体系,包括初级程序员与高级程序员的差别对比、数据库实用知识点、JSP+Servlet+JavaBean+DB 开发框架、Struts 框架、Spring IoC、Spring AOP、拦截器、Hibernate 和 Spring 的整合方式等,最后两章讲述了基于 SSH 和基于 Spring MVC 的两个案例,以及在面试时如何高效地介绍自己项目经验的方法。

本书的阅读人群是,想从事软件行业的在校学生、正在找工作的大学毕业生、想转行做 Java 开发但缺乏经验和已经工作的初级程序员。

本书附带教学视频,视频里会讲到所有案例的配置和运行方式,建议先观看视频运行的实例代码,然后再来阅读本书。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

图书在版编目(CIP)数据

Java Web 轻量级开发面试教程 / 孟宪福, 胡书敏, 金华编著. —北京: 电子工业出版社, 2017.8

ISBN 978-7-121-32145-0

I. ①J… II. ①孟… ②胡… ③金… 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 161178 号

责任编辑: 安 娜

印 刷:

装 订:

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编: 100036

开 本: 787×980 1/16 印张: 20 字数: 375 千字

版 次: 2017 年 8 月第 1 版

印 次: 2017 年 8 月第 1 次印刷

印 数: 2500 册 定价: 69.00 元

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

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

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

序

已有的统计表明,Java 是目前世界上应用最为广泛的程序设计语言,而 Java Web 则是 Java 语言的重要组成部分,也是基于 Web 程序设计最重要的语言工具,这是因为 Java 语言本身就是因互联网的出现而产生的。随着网络应用的普及,越来越多的程序设计工作都与网络有关。在此背景下,一个缺少 Web 程序设计能力、不了解 Java Web 知识及其发展历程的计算机及其相关专业的毕业生是难以找到理想工作的。同时,一个有志于在 Web 领域大显身手的程序员,如果不能熟练应用 Java Web 语言工具进行程序设计也是难以满足 Web 领域的工作需求的。本书的作者既有 985 高校的计算机专业教师,也有 Java Web 程序设计第一线的网络程序设计人员和系统工程师。作为教师,作者承担着为计算机专业的本科生和研究生主讲面向对象程序设计和分布式对象技术等课程的教学任务,其中 Java 语言是课程的主线内容。同时,承担着大量与国内外合作的基于 Web 的研发课题,所培养的学生在阿里巴巴、腾讯和百度等公司就业的不占少数。作为一线的网络程序设计人员和系统工程师,作者既肩负着繁重的 Java Web 程序设计任务,同时又担当着对应聘者的面试工作和对新员工的培训任务。因此,熟知这一领域对相关人才的专业需求和能力要求。在此专业背景下,我们相信本书的出版不但能为读者提供扎实的 Java Web 领域的理论教学内容和实践应用范本,同时,也能够为应聘此领域工

作的同学们和寻求进入此领域的社会人员提供有价值的面试导引。

本书共 11 章，按照循序渐进的原则逐步介绍 Java Web 程序设计的基本概念、基本内容和基本过程。由于 Web 程序设计离不开数据库的支持，因此本书还专门利用一章的篇幅来详细介绍数据库的设计原则，相信本章的内容对于那些仅对数据库感兴趣的读者也会受益匪浅。书中所有完整的程序都是在 MyEclipse 环境下调试通过的。要学好 Java Web 程序设计方法，除了需要掌握其基本理论，还必须要加强实践环节。读者可以边学习边上机，刚开始时可以调试本书中的程序，待学习一段时间之后，就可以调试自己编写的程序了，只有这样，才能加快学习进度，提高学习效率。

孟宪福

2017 年 6 月于大连理工大学

前言

为什么要从诸多的 Java 书籍里选择这本？为什么在当前网络信息量如此大的情况下还要买这本书，而不是自己通过查阅网络资料学习？我已经会开发 Java Web 程序了，有没有必要买这本书？

笔者有 12 年的 Java 经验，目前是某大型公司的架构师，知道软件公司对高级程序员的实际需求；而且笔者在大公司里有过 5 年的技术面试经验，面试过的人数上百，所以知道毕业生和初级程序员的普遍情况。笔者更有过 5 年多的 Java 培训经验，知道如何在短时间内把有毅力、有决心，但暂时缺乏技能经验的人培养成高级程序员的方法。

我在工作和培训中接触过不少刚走出校园的大学生和工作经验少于 3 年的初级程序员，发现他们虽然很上进，平时都会上网查找资料学习，但普遍会走弯路，比如学了目前用不到的知识点，或者对重要的知识点了解不深，或者干脆不知道该学哪些以及学习的进阶路线，这就导致他们掌握的技能 and 公司的需求（主要体现在面试上）不匹配。我也见过不少人项目经验足够，能力也不差，但他们就是通不过高级程序员的面试。

在 Java Web 方面，笔者从事了十多年的开发和培训工作，所以对于这方面的技术点，我大致知道哪些是不学就找不到工作，也知道哪些可以推迟到成功进阶后再学，更知道哪些可

以不用了解。此外，我还从培训和带领毕业生和初级程序员的实践效果中总结出了一套适用于大多数初级程序员的能少走弯路的进阶路线，由此由浅入深地构建了本书的知识体系。

大多数初级程序员都能升级到高级，只是时间问题，而本书的目的就是让大家缩短升级的时间。

第一，本书列出了在 Web 方面初级程序员升级到高级的必备知识点，以多数人升级时遇到的陷阱为警示，确保大家不在众多的知识点里迷失，而最终导致方向性错误。

第二，以公司对高级程序员的实际需求告诉大家必备的知识点应该怎么学。

第三，本书更从语法之外，告诉读者应该怎么从“性能调优”和“框架”的角度往更高的目标发展。

本书还从面试官的角度，在准备简历和面试方面写了一些能帮助读者的攻略，从而让读者能真正地把技能和经验转化成为金钱（升级进入好公司后钱就多了），这套攻略的实践效果是，每次我的培训班结束，总会有一批学生成功跳槽。

话说回来，“时间”还真是个大问题，我知道大家工作后一般都很忙，能给学习挤出的时间并不像上学时那样多，针对这个现状，本书只给出了常用的必需的知识点，能让大家用较短的时间代价完成到高级程序员的升级。

从内容和叙述方式来看，本书的案例和文字都是根据初级程序员的现状而原创的，尤其提到了初级程序员会忽视的技术要点。本书摒弃了大段华而不实的理论描述，这样能让大家不为无用的篇幅买单。而且，本书给出的技术描述和实践建议对于初级程序员来说，不是高深的，而是确保在当前知识储备下能看懂的。对于那些能帮助到高级程序员和架构师但现阶段帮不到初级程序员的高深知识点，本书不讲。

从案例角度来看，本书一个知识点会配置一个案例，并且每个案例都有视频教学，保证大家能通过观看视频调试出来，而且保证能从案例中学到关联的知识点。

在面试过程中，本人一定“错杀”过一些有技能但表述能力不强的候选人，不过也错误地招过一些能力偏差但会面试的候选人。本人也约谈过这些人，从而了解到一些怎样“假装自己是高级程序员”的方式，以此来提升本人的甄别水平。技术上不能弄虚作假，为了帮助技术好但不擅长面试的踏实程序员，在描述各技术点时，我参考了诸多程序员的面试经验，尽可能多地加上了“能证明自己懂”的叙述。

这本书的价值不仅在于提供的若干代码案例和若干视频，更体现在能帮助大家在进阶过程中少走弯路，体现在能切实有效地帮助大家面试，在讲述知识点时，总是尽可能地告诉大

家如何有效展示自己了解这部分知识点的方法，而且在本书的最后一章——第 11 章中，以技术面试官的直接经验，不仅讲述了在简历中如何展示自己能力的方法，而且还通过分析面试流程，给出了如何准备面试的攻略，以保证大家能在掌握技能的前提下有效地证明自己行，从而让你的学习得到应得的回报。

本书没有展示 Java Web 方面的所有知识点，而是选择性地讲了“足够能证明自己能力”的知识点，从而避免大家把时间用在“现阶段用不到的知识点”的学习上。而且，这些知识点的选择以及讲述方式是根据多年的培训经验精炼出来的，从而保证大家花较少的时间和精力就能掌握 Java Web 方面高级程序员所必备的知识点。所以对于在校大学生、毕业生和工作经验少于 3 年的初级程序员而言，本书是个不错的选择。

笔者的邮箱是 hsm_computer@163.com，如果大家在下载视频和案例代码时有问题，请及时联系我，另外，如果大家在学习过程中有任何的问题，也请及时告诉我。

最后特别说明，对于高级程序员以及架构师而言，虽然从这本书中你们能看到当年自己升级的路线和知识体系，但其中的知识点你们已经掌握了，建议不要去买！但如果你们愿意分享自己的升级经验，帮助我们进一步完善这本书的文字代码和视频，那么我们将感激不尽。

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

- **下载资源**：本书如提供示例代码及资源文件，均可在 [下载资源](#) 处下载。
- **提交勘误**：您对书中内容的修改意见可在 [提交勘误](#) 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 [读者评论](#) 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/32145>



目录

第 1 章 高级程序员的 Web 知识体系	1
1.1 合格 Java 程序员和高级 Java 程序员的技能比较	1
1.1.1 Java Core 方面	2
1.1.2 Java Web 方面	3
1.1.3 数据库层面	4
1.1.4 项目管理方面	5
1.1.5 能帮助到你的加分项	5
1.2 你可以少走的弯路	6
1.2.1 哪些知识点可以延后了解	6
1.2.2 大学阶段的实习经验能帮到你	7
1.2.3 刚开始的 1 到 3 年，找个专注的方向	8
1.2.4 常学习、多实践的人工资涨得快	8
1.3 上手 Web 方面的开发工具	9

1.3.1	在 MyEclipse 上开发和部署 Web 项目	9
1.3.2	更换端口号	14
1.3.3	引入外部的 jar 包	15
1.3.4	支持中文	16
1.4	推荐一些经过实践检验的学习方法	19
第 2 章	需要了解的数据库知识	21
2.1	合理地使用各种连接	21
2.1.1	内连接和左连接	21
2.1.2	范式和连接的代价	24
2.1.3	表的设计和冗余	25
2.2	不复杂但容易忽视的 SQL 用法	25
2.2.1	group by 和 Having	26
2.2.2	Having 的另一个常用用途——查看重复记录	27
2.2.3	通过一些案例来看一下常用的 Select 定式	28
2.3	索引的用途和代价	29
2.3.1	从结构上来分析索引的好处和代价	29
2.3.2	建索引时我们需要权衡的因素	30
2.3.3	索引的正确用法	31
2.4	让你的 JDBC 代码更专业	32
2.4.1	用 try...catch...finally 从句	32
2.4.2	预处理和批处理	34
2.4.3	事务的提交与回滚	36
2.4.4	事务隔离级别	38
2.5	总结	39
第 3 章	JSP+Servlet+JavaBean 框架	41
3.1	只应负责界面显示的 JSP	41
3.1.1	从一个大而全的例子中分析 JSP 的语法	42
3.1.2	“大而全”和“小而精”	45

3.2	让 Servlet 承担控制器的角色	46
3.2.1	基本知识点	46
3.2.2	生命周期与多线程运行方式	47
3.2.3	JSP+Servlet 的开发模式	49
3.2.4	运行结果和 JSP+Servlet 模式的说明	53
3.3	JSP+Servlet+JavaBean+JDBC 的开发模式	54
3.3.1	通过一个简单的例子来了解 JavaBean	54
3.3.2	在 JavaBean 里编写业务逻辑	57
3.3.3	对 MVC 的总结	64
3.4	总结	64
第 4 章	通过 Struts 进一步了解 MVC	66
4.1	在 MyEclipse 里开发一个基本的 Struts 程序	66
4.1.1	创建 Web 项目，并导入必要的 jar 包	67
4.1.2	开发前端的 JSP 代码	67
4.1.3	在 web.xml 里声明使用 Struts	68
4.1.4	配置 struts.xml 文件	69
4.1.5	开发 Action 类	69
4.1.6	开发两个跳转结果页面	71
4.2	通过运行，了解 Struts 的工作流程	71
4.2.1	Struts 的跳转流程分析	72
4.2.2	Struts MVC 框架和 JSP+Servlet+JavaBean 框架的比较	73
4.3	通过 Struts 的验证机制校验输入内容	74
4.3.1	通过 Validate 方法来验证	74
4.3.2	在配置文件里定义验证方式	77
4.4	Struts 拦截器	81
4.4.1	拦截器与职责链设计模式	81
4.4.2	通过登录案例详解拦截器的用法	82
4.4.3	拦截器的使用要点归纳	87
4.5	为了尽快进阶，你必须知道如下知识	87

4.5.1 Action 里的跳转	88
4.5.2 可以指定 Action 里的处理方法	89
4.5.3 对 Struts 框架的进一步了解	89
4.6 关于 Struts 框架的常见面试题	90
第 5 章 Spring 的基本知识点分析	92
5.1 依赖注入的好处	92
5.1.1 一个基本的依赖注入的程序	93
5.1.2 IoC 的特点，不用 New 就可以初始化类	95
5.1.3 控制翻转和依赖注入	95
5.1.4 面向接口编程的本质是缩小修改的影响范围	96
5.2 依赖注入的常用知识点说明	100
5.2.1 读取配置文件的各种方式	100
5.2.2 单例多例，有状态无状态 Bean	100
5.2.3 Spring 的注入方式与 AutoWire	103
5.2.4 @Autowired 注解	105
5.2.5 Bean 的生命周期	106
5.3 AOP，面向切面编程	108
5.3.1 面向切面编程的使用场景	108
5.3.2 面向切面编程的案例演示	108
5.3.3 深入了解面向切面编程的各种概念	114
5.4 如何证明自己了解 Spring 的基本技能	114
第 6 章 Spring 的 MVC 框架	117
6.1 一个只包含 MVC 的案例	117
6.1.1 开发 Spring MVC 的代码	117
6.1.2 Spring MVC 的运行流程	120
6.1.3 与 Struts MVC 的区别	122
6.2 Spring MVC 的关键类说明	122
6.2.1 通过 HandlerMapping 来指定处理的控制器类	123

6.2.2	通过视图解析器处理响应结果	127
6.2.3	通过 ModelAndView 返回视图结果	129
6.2.4	Spring 的拦截器	131
6.3	在 Spring MVC 方面你如何准备自己	135
第 7 章	通过 ORM 专注业务（基础篇）	138
7.1	让你尽快掌握 Hibernate	138
7.1.1	通过 Hibernate 完成 ORM 的具体步骤	139
7.1.2	通过 Hibernate 的注解方式实现 ORM	144
7.1.3	Hibernate 里生成主键的方式	146
7.2	Session 对象在项目里的用法	150
7.2.1	Session 对象中的重要方法	150
7.2.2	Session 对象中的 load 和 get 方法的差别	152
7.2.3	Session 缓存与三种对象状态	154
7.2.4	FlushMode 与清空缓存的时间点	157
7.3	在 Hibernate 里执行复杂的查询	158
7.3.1	where、groupby 和 having	158
7.3.2	表关联查询和子查询	160
7.3.3	通过 SQLQuery 对象执行 SQL 语句	162
7.3.4	通过 Criteria 设置查询条件	163
7.4	针对 Hibernate 基础知识部分的小结	166
第 8 章	通过 ORM 专注业务（高级篇）	168
8.1	通过 Hibernate 关联多张表	168
8.1.1	通过配置文件实现一对一关联	169
8.1.2	一对一关联的注解实现方式	173
8.1.3	一对多关联（配置文件，返回 List）	177
8.1.4	一对多关联（注解，返回 Set）	182
8.1.5	用 Map 来管理一对多关联	186
8.1.6	通过配置文件实现多对多关联	189
8.1.7	多对多关联的注解实现方式	194

8.1.8	关联性操作里的 cascade 取值	196
8.1.9	通过 inverse 设置外键控制权	198
8.2	Hibernate 里的事件机制	201
8.2.1	在拦截器里放一些通用性的代码	201
8.2.2	事件系统和监听器	207
8.3	Hibernate 中的优化	212
8.3.1	结合数据库大背景	212
8.3.2	使用 SessionFactory 二级缓存	213
8.3.3	项目中常用的优化策略	218
8.4	在 Hibernate 方面我们面试的方式	218
第 9 章	Spring 整合数据库层面的应用	221
9.1	Spring 与 Hibernate 的整合	221
9.1.1	Spring 整合 Hibernate 注解的例子	222
9.1.2	配置数据池来提升效率	225
9.2	通过 Spring 管理事务	227
9.2.1	编程式事务管理方式	227
9.2.2	声明式事务管理方式	232
9.2.3	事务传播机制要解决的问题（适用范围）	235
9.3	针对 Spring 整合数据库的总结	237
第 10 章	Web 框架案例分析	238
10.1	Struts、Spring 与 Hibernate 的整合	238
10.1.1	SSH 整合案例的说明	239
10.1.2	编写登录页面和 Web.xml 配置文件	240
10.1.3	编写 Struts 的配置文件 struts.xml	242
10.1.4	编写 Spring 的配置文件	243
10.1.5	编写 Struts 的 Action 类	245
10.1.6	编写 Service 和 DAO 类	246
10.1.7	编写 Model 类和映射文件	248

10.1.8	编写显示返回结果的 index.jsp	249
10.1.9	对 SSH 框架的分析	250
10.2	基于 Spring MVC 的 Web 框架分析	251
10.2.1	Spring MVC 案例的说明	251
10.2.2	在 web.xml 里定义使用 Spring MVC	253
10.2.3	编写整合 Hibernate 的 xml 文件和 Model 类	254
10.2.4	配置 Spring MVC 的 xml 文件	257
10.2.5	编写前端的增改查 JSP 文件	258
10.2.6	编写拦截器类和控制器类	261
10.2.7	编写 Service 层的代码	264
10.2.8	编写 DAO 层的代码	265
10.3	描述商业项目案例经验	267
第 11 章	简历面试那些事	271
11.1	不要让你的简历进回收站	271
11.1.1	面试的基本流程	272
11.1.2	根据职务介绍再针对性地准备简历	272
11.1.3	哪类简历比较难获面试机会	273
11.1.4	准备简历时该注意哪些	273
11.2	面试之前，你要做哪些准备	274
11.2.1	准备项目经验描述，别害怕，因为面试官什么都不知道	274
11.2.2	面试官的策略——如何通过提问，找出你回答中的矛盾	275
11.3	面试陈述篇：充满自信地描述你的项目经验	275
11.3.1	准备项目的各种细节，一旦被问倒了，就说明你没做过	276
11.3.2	充分准备你要说的项目的框架、数据库	277
11.3.3	不露痕迹地说出面试官爱听的话	279
11.3.4	一定要主动，面试官没有义务挖掘你的亮点	280
11.3.5	一旦有低级错误，可能会直接出局	281
11.4	面试引导篇：把问题引入准备好的范围	282
11.4.1	项目要素、框架设计和数据库，这些是必须要准备的	282

11.4.2	准备些加分点，在介绍项目时有意提到，但别说全	284
11.4.3	对于面试官的圈套，别顺口回答	284
11.4.4	别自作聪明地回答面试官没问到的但你很想说的亮点	285
11.5	必问的问题：这些非技术问题你逃不掉的	286
11.5.1	如何描述你在项目里的作用？别单说你仅仅 coding	286
11.5.2	一定要找机会说出你的团队合作能力	287
11.5.3	当问项目周期规模和技术时，是在考查你值多少钱	288
11.5.4	想尽办法展示你的责任心和学习能力	289
11.6	开放性问题篇：面试官想摆脱你的准备，别慌，有技巧	290
11.7	当你有权提问时，别客气，这是个逆转的好机会	290
11.7.1	通过提问，进一步展示你和职位的匹配度	291
11.7.2	通过提问，说出你没被问到的亮点	291
11.7.3	可以展示一些非技术的特长	291
11.8	亡羊补牢：万一你回答不好，该怎么办	292
11.8.1	坦诚相对，说明你的擅长点，让面试官给次机会	292
11.8.2	展示你以前的亮点，让面试官相信你的潜力和能力	293
11.8.3	记下所有的面试题，迎接下次面试	295
11.9	基础差，不知道怎么应对面试时的对策	295
11.9.1	有计划的学习和实践	295
11.9.2	多挖掘你之前的项目经验和技能点	296
11.9.3	及时提升你项目里用过的知识	297
11.10	背景调查的一般流程	297
11.10.1	技术面试阶段，着重甄别是否是商业项目	298
11.10.2	关键因素一旦不对，立即出局	298
11.11	面试评分的一般依据	299
11.11.1	技术面试的考查要点	299
11.11.2	综合能力面试的考查方式	301

视频目录

第 1 章

安装本书运行环境 MyEclipse 和 MySQL 的步骤

运行第一个 Java Web 程序

更换端口号，引入 jar 包和让开发环境支持中文

分享 Java Web 方面的学习经验

第 2 章

关于连接的讲解

在项目里建表的注意点

如何正确地创建和使用索引

JDBC 代码的注意点

事务隔离级别的讲解

如何准备面试，数据库篇

第 3 章

简单 JSP 案例的讲解

JSP+Servlet 的案例分折

简单 JavaBean 案例的讲解

基于 JSP+Servlet+JavaBean+DB 案例的讲解

第 4 章

开发一个简单的 Struts 案例

在 Struts 里加入验证器

通过配置文件实现 Struts 验证器

Struts 拦截器的讲解

如何准备面试，Struts 篇

第 5 章

Spring IoC 的讲解

Spring 面向接口编程

用单例和多例创建 Spring Bean

SpringAutoWire By Name 的讲解

通过注解实现 Spring 的 IOC

Spring AOP 的讲解

第 6 章

Spring MVC 的讲解

通过 HandlerMapping 指定控制器

BeanNameUrlHandlerMapping 用法说明

在 Spring MVC 里配置多个视图解析器

Spring 拦截器的讲解

如何准备面试，Spring 篇

第 7 章

Hibernate 的简单案例说明

通过注解实现 Hibernate

Hibernate 里生成主键的方式

Hibernate 里 Session 对象的讲解

Hibernate 三种 Session 对象

HQL 的用法

通过 HQL 实现关联等复杂查询

在 Hibernate 里运行 SQL 语句

Hibernate 里 Criteria 对象的用法

第 8 章

Hibernate 里的单向一对一关联

通过注解实现双向一对一关联

通过配置文件实现一对多的关联

通过注解实现一对多关联

通过 Map 实现一对多关联

通过配置文件实现多对多关联

通过注解实现多对多关联

通过 inverse 设置外键控制权

在 Hibernate 里实现拦截器

Hibernate 监听器的讲解

在 Hibernate 里实现二级缓存

如何准备面试，Hibernate 篇

第 9 章

Spring 和 Hibernate 的整合

整合时 Hibernate 里配置连接池

编程式事务的讲解

声明式事务的讲解

第 10 章

SSH 整合案例的讲解

Spring MVC+Hibernate 整理案例的讲解

在面试里，如何准备你的项目描述

第 11 章

面试流程的讲解

如何准备你的简历

面试前的准备

面试技巧的综合说明

如何在短时间内提升你自己的能力

第 1 章

高级程序员的 Web 知识体系

衡量高级程序员的标准既简单又复杂，简单的标准是年限和工资。高级程序员需要的工作年限一般是本科生两年半左右（好学校的学生年限能适当降低到两年，硕士一般是一年），能给到的工资一般能达到当年毕业生平均工资的 1.5 倍左右（能力强的入门就能给到 2 倍以上）。

但年限只是一个基本的指标，比较复杂的标准是综合能力（似乎是废话）。作者以多年的高校教学经历及高级程序员+架构师+面试官的经验，在本书的诸多章节里列出了在 Java Web 方面高级程序员必须要掌握的知识点。本章没有用很大篇幅讲述环境的安装步骤（因为已经录制在视频里了），而是从总体上列出从新人到高级程序员升级的诀窍。

1.1 合格 Java 程序员和高级 Java 程序员的技能比较

合格程序员的最低标准是没有任何商业项目经验，但有能力从事 Java 方面软件开发（至少能成功通过标准最低的面试）的毕业生（或者想转行做 Java 的人）。初级程序员的标准是已经在干活了，至少有商业项目经验。所以说，合格<初级<高级。

本书阅读的起点其实是低于合格程序员的，但却按照大多数企业对高级程序员的最低标准，整理出的重要知识点。而在开篇，就从各方面对比了合格程序员和高级程序员的不同标准，这其实是大家的前进方向。

1.1.1 Java Core 方面

Java Core 开发也叫核心开发，不涉及 Web 方面的知识。虽然本书是针对 Java Web 方面的，但 Java Core 方面的能力却是 Web 开发的基础，所以还是会用少量的篇幅大致列出针对 Core 方面程序员的标准。如表 1.1 所示。

表 1.1 C ore 方面合格与高级程序员的差别

方面	合格程序员的标准	高级程序员的标准
集合	1.对于各线性表类对象（比如 Array、LinkedList、ArrayList、Stack、Set 等）和键值对类对象（比如 HashMap 等），会基本的遍历和增、删、改操作 2.会使用 Iterator、泛型、比较器等常用对象	1. 根据业务需求，合理地选用对象 2. 掌握一些高级对象的用法，比如 WeakHashMap 和 ConcurrentHashMap 等 3. 在使用集合对象时，能保证内存的性能不恶化
异常处理	1.会用 try...catch...finally 框架 2.了解各种异常的类型，比如运行期异常，数据库或者 I/O 等异常	1.合理地编写异常部分代码，出现问题，通过输出能找到问题点和解决方法 2.针对项目，能合理地设计异常处理流程
I/O	能完成基本的 I/O 操作，比如读写文件、读写内存、读写压缩包等	会用 NIO，而且在读写操作时能保证内存性能不恶化
JDBC	会基本的连接、增、删、改、查、预处理、批处理等操作	1.能够熟练地使用事务，掌握事务隔离级别等重要概念 2.编写出来的 JDBC 部分代码既能保证内存性能，又能保证数据库的性能
多线程	1.会创建多线程，而且能通过 notify、wait、sleep 等关键字，让多线程协调地完成项目里的任务 2.会用 synchronized 等常用关键字，在多线程读写情况下不会产生冲突问题 3.最好会用线程池	1.知道 Lock、Condition 等比较高级的关键字 2.根据项目的需求，能合适地选用各种线程池 3.能保证多线程在协同工作时尽可能少地产生死锁，即使出现死锁，也能快速地找到原因 4.在编写多线程场景的代码时，能事先考虑到线程安全性方面的问题
面向对象和设计模式	知道基本的概念，会用诸如继承、抽象类和接口等的语法	能根据项目的需求，合理地设计基类和接口，从而搭建合适项目的基本框架
垃圾回收和内存性能管理	知道基本的概念，知道基本的 System.gc 等的语法，知道通过 java -xms 等基本的配置内存的操作	1.了解内存回收的流程 2.在写代码的过程中，能保证最优化地使用内存 3.一旦出现内存溢出等异常，能根据监控日志找到问题点

对于合格程序员而言，大多数情况下，只要会干活就行，所以标准大多是：会某某技术，在代码里会某某操作。

对于高级程序员而言，除了需要了解更高级的知识点外，还要能根据项目的需求，自主地选用合适的对象，而且需要考虑性能优化等问题。

1.1.2 Java Web 方面

在 Web 方面，合格程序员的标准也是“能干活”，我们用表 1.2 列出了合格程序员与高级程序员的标准对比。

表 1.2 Web 方面合格程序员与高级程序员的标准

方面	合格程序员的标准	高级程序员的标准
JSP+Servlet+JavaBean (简单的 MVC 框架)	1. 会用 JSP+Servlet+JavaBean 这套框架编程，知道基本的 MVC 流程 2. 最好了解些简单的 JS、DIV、CSS 等前端技术 3. 知道怎样把 Web 程序发布到服务器上	1. 了解 JSP 和 Servlet 在多线程下编程的注意点 2. 知道在哪些场景下可以用这套框架 3. 了解适当的前端技术，比如 JS、DIV 和 CSS 4. 能独立地把 Web 程序发布到不同的服务器，比如 Tomcat 上
Struts 方面	可以不了解，因为用得比较少	最好能了解，因为这是一个比较成熟的基于 MVC 的框架
Spring 方面	1. 知道 IoC 和 AOP 的概念，知道如何使用这些技术 2. 知道 Spring MVC 的开发流程，能在项目经理带领下开发基于 Spring 的 Web 项目 3. 最好能了解 Spring MVC 方面的一些组件	1. 能体会包含在 IoC 和 AOP 技术里的面向接口和自由组装的思想，并能适当地用到项目里 2. 知道 Spring Bean 的生命周期，必要时，重载其中的一些方法 3. 了解 Spring 基于 MVC 的各种组件，比如 HandlerMapping 等，针对小型项目，能独立地设计框架，能在框架里编写一些通用性的配置文件和代码
Hibernate (或者 ORM)	1. 可以只掌握一种 ORM 技术 2. 能用 Hibernate 做些基本的增、删、改、查等操作 3. 熟悉一些基本组件，比如 SessionFactory、Criteria 和 Session 等的用法 4. 知道一对一、一对多、多对多的基本用法 5. 知道缓存概念，最好能了解性能优化等技能	1. 最好掌握多种 ORM 技术（不过不是绝对的） 2. 能根据场合，适当地选用 HQL 和 SQL 语言 3. 能根据项目的实际需求，设计一对一、一对多和多对多的模型，并能适当地设置 cascade 和 inverse 这两个关键字 4. 知道拦截器和监听器等事件相关的技术，在遇到项目需求时，能自主地选用这些技术 5. 编写代码时有性能优化意识，出现性能问题时能提出解决方案 6. 能配置声明式事务
Spring 和 Hibernate 整合	这点上合格程序员和高级程序员的差别不大，要求是，能整合，能配置连接池	能整合，能配置连接池

Web 方面其实要解决的问题相对简单，开发出来的网站只要能适应高并发大数据的需求即可，比如需要达到双 11 时的访问量。

上述这个需求点的技能其实是深不见底的，表 1.2 中列出的技术点其实是针对高级程序员的（最低标准）。相比合格程序员，对高级程序员的技能要求主要体现在以下三个方面。

- 首先知识面要全，能适当地选用技术来满足项目的实际需求，这种需求不仅仅是业务需求，还包括性能方面的需求。
- 其次是要有调优意识，在测试阶段和项目上线后，能通过打印日志或配置现成的监控程序来监控哪块性能不好，并给出解决方法。
- 关键点，要有框架方面的能力，合格程序员可能仅仅知道基本的 Spring MVC 框架怎么搭建，而对于高级程序员，遇到小项目（页面在 10 个左右，没有太大的并发和数据量的需求），需要能自主地搭建一整套框架，然后需要带领一帮程序员完成实际的项目。

1.1.3 数据库层面

对于合格的程序员，需要有基本的数据库操作技能，具体体现在以下三个方面。

- 第一，针对一类数据库（比如 MySQL、Oracle、SQL Server 等），会基本的增删改查操作，会用一些基本的函数，会编写存储过程触发器索引等工具。
- 第二，知道一些基本的对项目开发有帮助的概念，比如范式、索引、分区等。
- 第三，能编写一些相对复杂的 SQL 语句，比如带连接、带子查询、嵌套查询等。

对于高级程序员，用过的数据库种类当然是越多越好，此外，还要掌握如下三大方面的能力。

- 第一，能设计出各类复杂的 SQL 语句来满足项目中的各类需求。
- 第二，能根据项目情况，自主地设计数据表结构，并能合理地配置外键和主键。
- 第三，也是最重要的，要有一定的数据库调优能力，比如能合理地创建表结构，能正确地创建索引，而且能通过使用索引合理地优化数据库性能，能通过执行计划分析并优化 SQL。

由于各类数据表的调优技术不一定一致，所以高级程序员最好能掌握针对多种数据库的调优技术。

在项目开发过程中，合格程序员的职责一般是编写（增、删、改、查存储结构的）SQL 语句，而高级程序员的职责还包括设计和调优。

1.1.4 项目管理方面

达到高级程序员的标准后，今后不论是向技术层面的架构师方向发展，还是向管理方向的项目经理发展，都需要和项目打交道。

对于大多没有商业项目经验的合格程序员而言，可以只了解项目开发的基本流程或者常见管理方式，不了解关系也不大，毕竟这些要靠做项目来逐渐积累经验。

对于高级程序员而言，项目管理方面的能力可以不达到项目经理的高度，但至少需要具备如表 1.3 所示方面的能力。

表 1.3 项目管理方面合格与高级程序员的差别

方面	合格程序员的标准	高级程序员的标准
项目开发流程	可以只了解理论知识	至少知道一种项目开发的方式（比如敏捷开发），而且知道项目开发各阶段、各时间节点要做的事情，因为有可能要协助项目经理管理项目
项目开发和管理的工具	可以不用了解	知道各种常用的项目开发和管理工具，最好有过使用和配置的经验。比如，会用 Maven 或 GIT（或者其他工具）来管理项目，会用 Jenkins 等工具 build 或发布程序，会用 Jira 等工具来管理开发任务和记录 Bug，会用 Sonar 等工具来监测项目开发的质量
项目文档设计	最好会用 UML 绘制过设计图，知道项目开发需要哪些文档，其中具体包含哪些要素	会用 UML 等工具描述你对于项目设计的想法，比如通过流程图和时序图来说明某个功能模块的流程，通过类图来描述项目里静态的类结构，通过部署图来描述项目的发布结构等
综合能力	可以在项目经理的带领下完成开发工作	可以协助项目经理管理项目，比如可以切分功能模块，然后带领初级程序员一起做，而不是被动地等待分配任务 一旦遇到技术难题，能主动地通过查阅资料提出解决方案，有自己的设计和开发方面的想法，实在无法解决时再请别人帮忙，而不是等待别人给出指导

1.1.5 能帮助到你的加分项

对于即将工作的合格程序员而言，如果有如下的经验或技能，则在一定程度上能帮助你成功地找到工作。

第一，可以在英文环境下独立地工作，包括可以读懂和编写英文邮件，能独立和老外畅快地交流，如果听和说两方面达不到这个程度，至少能读写英文邮件。

第二，有相关实习经验，这里特别说明要有商业项目的实习经验，而不是毕业设计项目或者学习时自己搭建的项目又或者是培训学校里做的项目。实习时间越长越好，大公司的实

习经验比一般公司要好，商业公司的实习经验比在校帮老师干活的实习经验要好（因为在校项目一般规模不大，而且有些是为了应付毕业设计立项的，并不是真正的商业项目）。

第三，有一定的人际交往和沟通能力，或者说是亲和力，至少在面试的时候让人感觉你很好相处，不内向，能清晰地表达自己的想法。甚至这种交流沟通能力要比专业技能重要，因为多数刚毕业的大学生（或者工作经验很少的人）彼此的技能相差不多，发展空间也很大，所以一般公司更愿意找个表达、交流、沟通没问题的人。

第四，因为软件公司难免要加班，所以你最好能证明你是个非常能吃苦耐劳的人，千万别直接说不能接受加班。

比如应聘时有人在简历上写道:大三期间帮老师干活，由于工期紧，所以通宵了几天然后把项目做出来了。还有人这样写，在上家公司，我连续出差到 xx 城市 3 个月，期间在现场帮客户搭建了 xx 系统，而且解决了 xx 重大问题。这些不外乎是为了证明自己的能力与吃苦精神。

1.2 你可以少走的弯路

在和不少比较上进的初级程序员打交道的过程中，我们总结出了一些能帮到合格程序员尽快进阶的经验，从总体上来讲，多学、多实践不吃亏。

1.2.1 哪些知识点可以延后了解

在 Java Core 方面，表 1.4 中的知识点你可以不学习或者到用的时候再学习。

表 1.4 Jav a Core 方面可以延后学习的知识点

知识点	学习的时机
界面开发方面的知识，比如 Swing 等	Java 主要用在 Web 方面，很少有项目会用到这些 UI 部分的知识点。大家可以等实际用到时再学习
Socket 编程方面	可以先了解概念，等有项目需求时再学习
Applet 方面	很少用，等有项目需求时再学习
虚拟机方面	虚拟机很重要，因为能对性能调优产生立竿见影的效果。不过这得靠技术积累，所以建议有至少 2 年相关工作经验后再学习，刚开始时，可以先了解概念和相关的基本内存管理知识点

在 Web 方面，建议大家先了解一整套框架，别过早地钻入某个方面的知识点，比如大家可以先通过 Spring MVC+Hibernate（甚至是 JSP+Servlet+JavaBean）搭建一个包括基本的前端

页面+MVC 架构+后台代码+数据的 Web 小项目（比如学生管理系统），先了解基于 Web 框架开发的一整套知识体系，随后再不断深入了解各 Web 组件的 API 等细节。

一般来说，在开始阶段，表 1.5 所示的 Web 知识点可以延后学习。

表 1.5 Jav a Web 方面可以延后学习的知识点

知识点	学习的时机
JSP 内嵌对象	可以先大致了解概念和基本的用法，没必要刚开始就深入了解具体内嵌对象的 API
Java Scrip、CSS、DIV 等前端知识	如果不是走前端路线，在刚开始接触 Web 开发时，这些前端技术可以不必过多关注

1.2.2 大学阶段的实习经验能帮到你

一般公司在筛选候选人的简历时，一个非常重要的考查要点是相关经验的工作年限。说一个典型案例，某公司要招高级程序员，需要有 3 年左右经验，三个候选人都是毕业 1 年半的程序员，但其中一位在大三开始有 1 年实习经验，结果就他获得了技术面试的资格。

这种工作经验外加实习经验凑满年限要求的案例不在少数，对于刚毕业的多数大学生来说，一般商业项目经验的工作年限都比较少，这个时候，是否有实习经验直接决定了是否有面试机会甚至关系到是否能找到合适的工作。

目前大多数的软件公司在筛选简历时，对于工作年限小于 1 年或者没经验的简历，如果没有其他额外的加分项（比如海归或拿过编程大奖或英语非常好），一般会直接过滤，更何况候选人的学校还不是 985 或 211 学校。

毕竟，一个好学校的招牌确实能证明毕业生的平均能力很强，但这种证明力未必要比直接的工作经验来得更有效。

我们见过一些本科毕业生的简历，他们天真地把毕业设计项目（诸如图书馆管理系统或者学籍管理系统等）和一些实训项目（培训学校的教学项目）写到简历中。要知道商业项目（需要靠这个挣钱）和学习项目是两回事，侧重点也不同，所以公司只看重商业项目，一些学习项目的作用充其量只能证明你在这方面有过了解。

所以给大家的建议是，在不影响学习和毕业的前提下，尽早到一些公司去实习，哪怕钱很少加班多也要去。一方面可以通过实战提升自己的专业能力，另一方面，能给自己提升用钱买不到的商业项目工作年限。

1.2.3 刚开始的 1 到 3 年，找个专注的方向

这里的专注有两个含义：

第一，得专注地从事软件相关的行业。我们见过不少简历，工作经验确实有 3 年（甚至更多），但是在简历中堂而皇之地写明，有 1 年多是从从事非软件行业的，比如做硬件、做维护，甚至做和计算机专业无关的销售等，而编程相关的经验相对来说很少，那么这些简历中的相关工作年限只能扣除非编程经验的年限。

第二，需要在工作的前 3 年内，给自己制定一个大发展方向，是数据库管理和优化方向，或前端，或后端，或测试，或大数据等，换工作也以这个方向为前提。

比如某公司要找个做 Java 后端工作年限 5 年的人，但某份简历中，虽然也有 5 年经验，但前 2 年是偏重前端，第三年做测试，后 2 年才是 Java，那么相关后端经验也只能按 2 年算。

这里想请大家注意的是，你可以多充实自己，多学习各方向的知识点，但一定要在一个方面（比如最近比较流行的是大数据）钻进去，让你在这方面拥有较多的项目实践经验。否则，假设你工作年限也达到资深的标准了（3 年以上），但你在换工作的时候，会发现虽然你能去参加很多类型的面试，但每个方面你都没法证明自己是资深者（面试官自有一套甄别资深者的方式，而且有些问题一定是得做了很多项目才能回答出来，所以千万别有任何蒙混过关的想法）。

1.2.4 常学习、多实践的人工资涨得快

虽然在前文里给大家列了些别人的经验，但对于任何程序员来说，弯路一定会走，本书的宗旨是让大家少走点弯路。

怎么才能少走弯路（或者说早些从弯路上回头）？答案只有学习和实践。

比如某好学者一个月额外学习时间是 40 小时（平均每工作日学习 1 小时，每周末学习 5 小时，一个月算 4 周），目前一般的学习效率是 70% 左右，毕竟你不能保证你看的知识点一定有用，而且也不能保证你一定能准确地理解学到的知识点，那么算下来一个月的有效学习时间只有 28 小时，这和在线游戏里的练级时间一样，时间越长，你的等级也就越高，工资也就越高。

给大家讲些真金白银的例子，按 2017 年的行情，工作经验满 3 年的平均工资在 12000 元左右，上不封顶，看能力，我们见过较高的有 15000 元左右。问下来这批人大概每周的学习时间在 30 小时（也不算多）。我们也了解过不少工资在平均线（12000 元）以下的原因，虽

然和公司种类（比如外企创业公司或者互联网公司）行业（比如证券金融）等其他因素有关，但学习时间少也是相对重要的因素，毕竟如果你能力强，找到一个工资高的工作的机会就大。

1.3 上手 Web 方面的开发工具

在 Web 开发方面，一般是用 MyEclipse 开发，有些公司可能用 Eclipse 加其他插件的开发方式，但基本的界面和开发部署流程和 MyEclipse 很相似。

安装步骤请参照视频，下面我们来演示一下常用的操作。

1.3.1 在 MyEclipse 上开发和部署 Web 项目

我们看到的所有网站，比如淘宝、京东等，都是运行在 Web 服务器上的，常见的 Web 服务器有 Tomcat、Apache、Websphere 和 Weblogic 等。

本书用到的案例都是运行在 MyEclipse 自带的 Tomcat 服务器上，这里我们看下开发和运行一个基本的 Web 程序的步骤。

代码位置	视频位置
code/第 1 章/MyFirstWeb	视频/第 1 章/第一个 Web 程序的讲解

步骤一 新建一个 Web 项目，如图 1.1 所示。

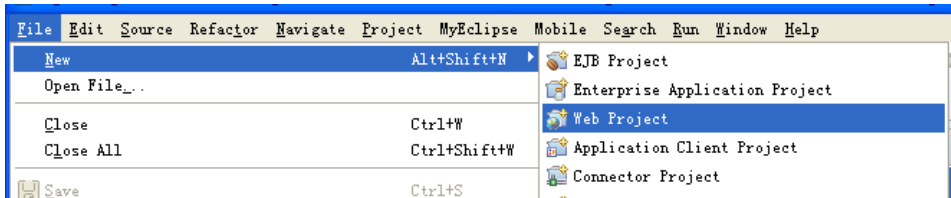


图 1.1 新建一个 Web 项目

在弹出的窗口里，输入项目“MyFirstWeb”，随后可以选择适当的 Java 版本，这里是 1.7，如图 1.2 所示。

完成创建后，能看到如图 1.3 所示的目录结构，在 Web 项目里，一般我们把 Java 程序放到 src 目录下，把 Web 相关的程序，比如 JSP,JS,xml 配置文件放到 WebRoot 目录下。

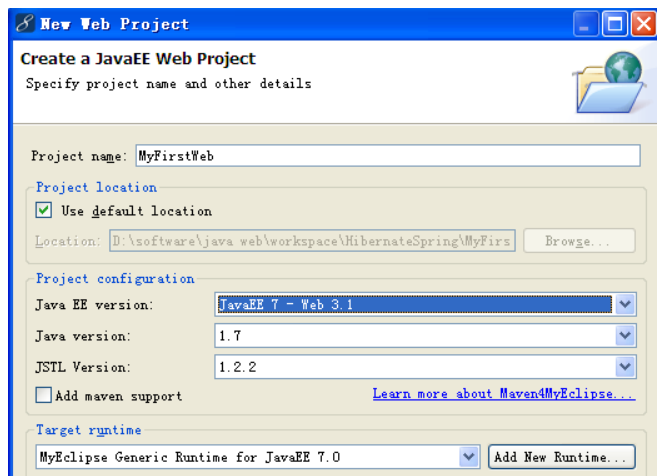


图 1.2 输入项目名，并选择 Java 版本

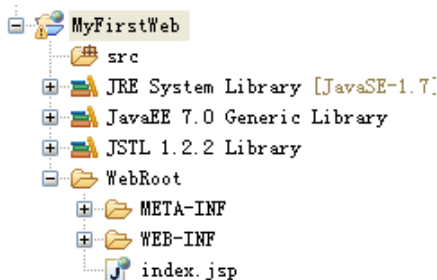


图 1.3 Web 项目的一般结构

步骤二 编写基本的 Web 程序。

在创建完 Web 项目后，会生成一个默认的 index.jsp 文件：

```
1 <%@ page language="java" import="java.util.*" pageEncoding="ISO-8859-1"%>
2 <%
3 String path = request.getContextPath();
4 String basePath =
5     request.getScheme()+ "://" + request.getServerName() + ":" + request.getServerPort() +
6     path + "/" ;
7 %>
8 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
9 <html>
10 <head>
11 <base href="<%=basePath%>">
12 <title>My JSP 'index.jsp' starting page</title>
13 <meta http-equiv="pragma" content="no-cache">
```

```

12 <meta http-equiv="cache-control" content="no-cache">
13 <meta http-equiv="expires" content="0">
14 <meta http-equiv="keywords" content="keyword1,keyword2,keyword3">
15 <meta http-equiv="description" content="This is my page">
16 </head>
17 <body>
18     This is my JSP page. <br>
19 </body>
20 </html>

```

这是个典型的 JSP 文件，通过第 18 行，能在页面上显示一段话。

步骤三 发布并运行这个 Web 程序。

我们可以在 Servers 这个控制标签里，看到 MyEclipse 自带的 Tomcat 服务器，这里是 Tomcat7，如图 1.4 所示。

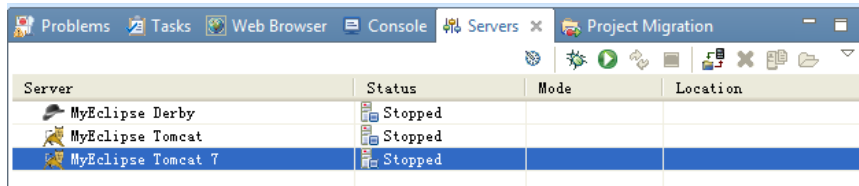


图 1.4 Servers 管理标签页

如果在下方看不到 Servers 标签，则可以通过如图 1.5 所示的方法，在 Windows→Show View 的菜单里打开 Servers，如果在 Show View 菜单里还是看不到 Servers，则可以从最下方的 Others 里打开。

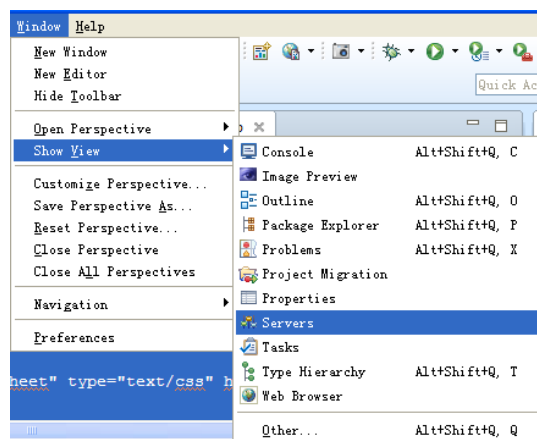


图 1.5 从菜单里打开 Servers 标签的示意图

用鼠标右击 Servers 管理标签里的 Tomcat7，能看到“Add Deployment”这个选项，如图 1.6 所示。

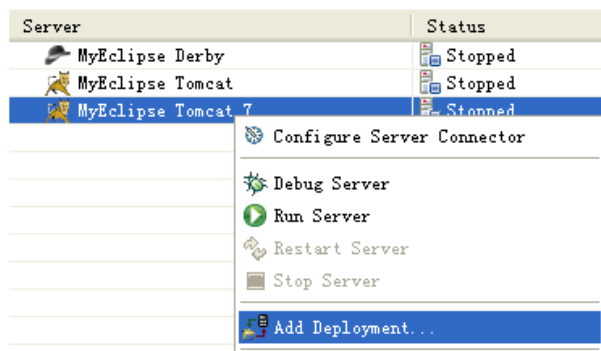


图 1.6 在 Tomcat7 上发布项目

在随后弹出的窗口里，选中刚才创建的 MyFirstWeb 这个 Web 项目，如图 1.7 所示，随后单击“Finish”按钮，即可完成发布工作。

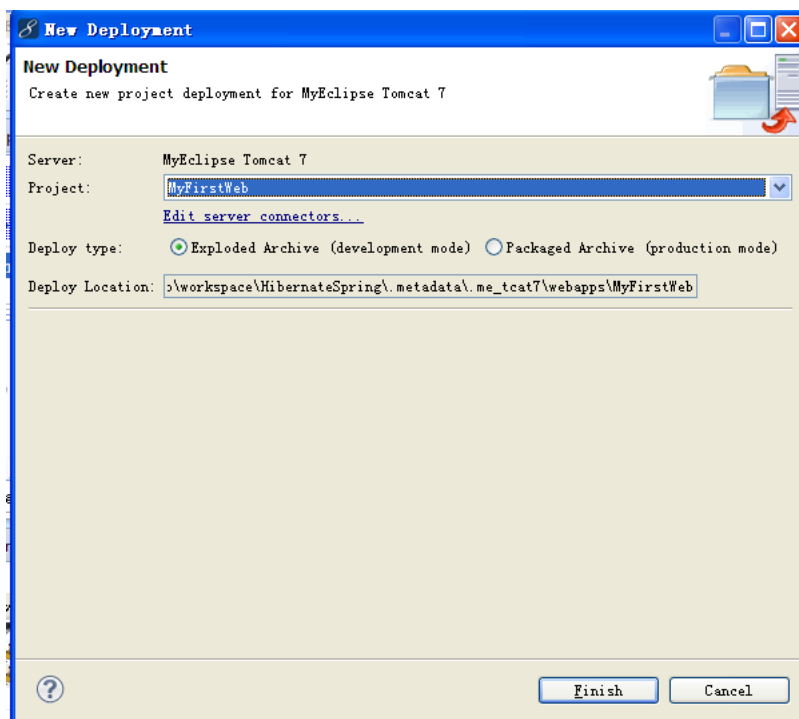


图 1.7 在 Tomcat7 上选择要发布的项目

步骤四 开启 Tomcat，运行 Web 页面。

完成发布 Web 程序后，能在 Tomcat7 里看到 MyFirstWeb 这个项目。用鼠标右击 Tomcat，在弹出的菜单里选择“Run Server”，即可开启 Tomcat 服务器，如图 1.8 所示。

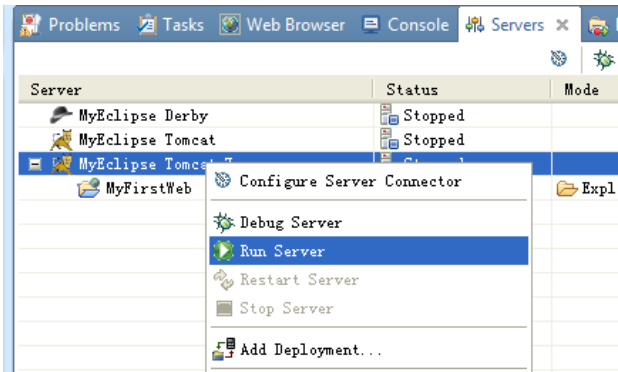


图 1.8 打开 Tomcat 服务器

启动后，Tomcat 的状态就会变成 Running，如图 1.9 所示。

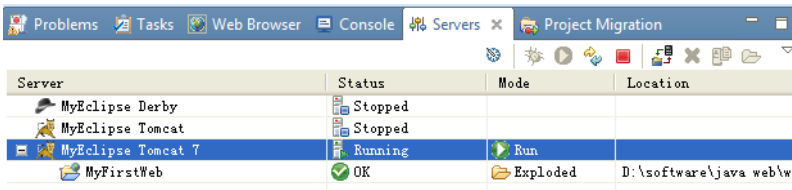


图 1.9 T omcat 成功启动后的示意图

当我们切换到 Console 这个标签后，能看到成功启动的打印信息，如图 1.10 所示。如果有问题，同样会在这里看到出错信息。

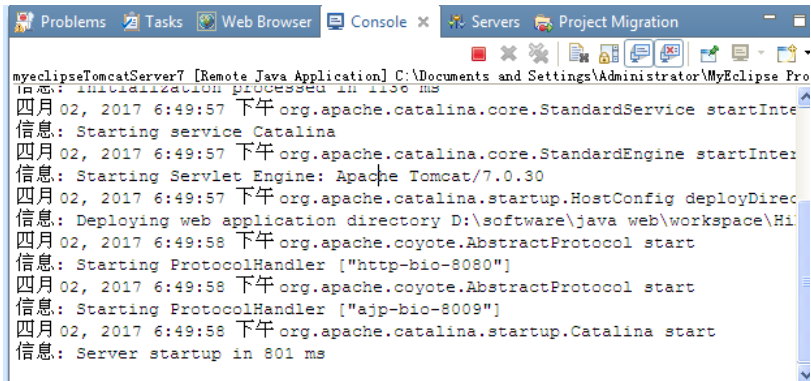


图 1.10 T omcat 成功启动后输出的打印信息

当我们成功地发布 MyFirstWeb 这个 Web 程序，而且成功地启动 Tomcat 服务器后，就能在浏览器里看到效果。

在浏览器里输入 `http://localhost:8080/MyFirstWeb/index.jsp`，其中，localhost 是 IP 地址，8080 是端口号，MyFirstWeb 是项目名，最后的 index.jsp 是文件名。这样，即可看到如图 1.11 所示的效果图。



图 1.11 在页面里看到的效果图

1.3.2 更换端口号

Tomcat 服务器一般用的是 8080 端口，如果一台服务器上运行了多个 Tomcat 服务器实例，则我们必须更改一个服务器的端口号。更改端口号是一个比较常见的应用，步骤如下。

步骤一 在 Tomcat7 这个服务器上，用鼠标单击右键，选中 `Configure Server Connector` 菜单，如图 1.12 所示。

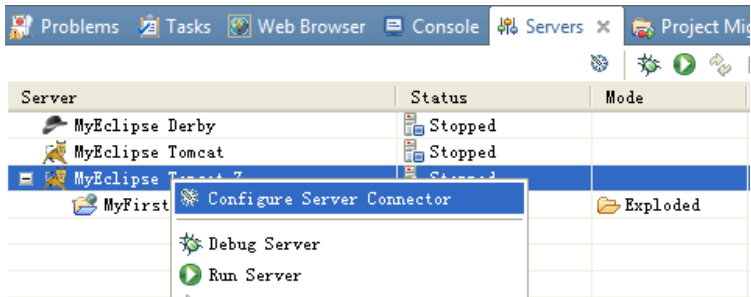


图 1.12 更改 Tomcat 服务器端口号的示意图

步骤二 在随后看到的窗口上的 `Port Number` 一栏里，我们就可以更改端口号了，如图 1.13 所示。

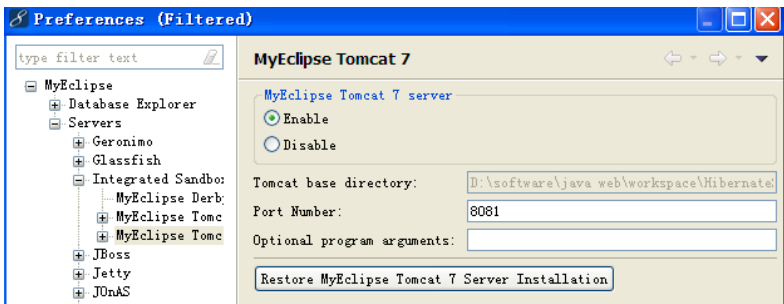


图 1.13 更改 Tomcat 服务器端口号的实际操作

1.3.3 引入外部的 jar 包

在开发项目过程中，我们可以通过引入不同的 jar 包来引入不同的支持文件，比如在后面的章节里，我们需要引入支持 MySQL 驱动等 jar 包，引入外部包的步骤如下。

步骤一 右击项目名，在弹出的菜单里选择 Properties，如图 1.14 所示。

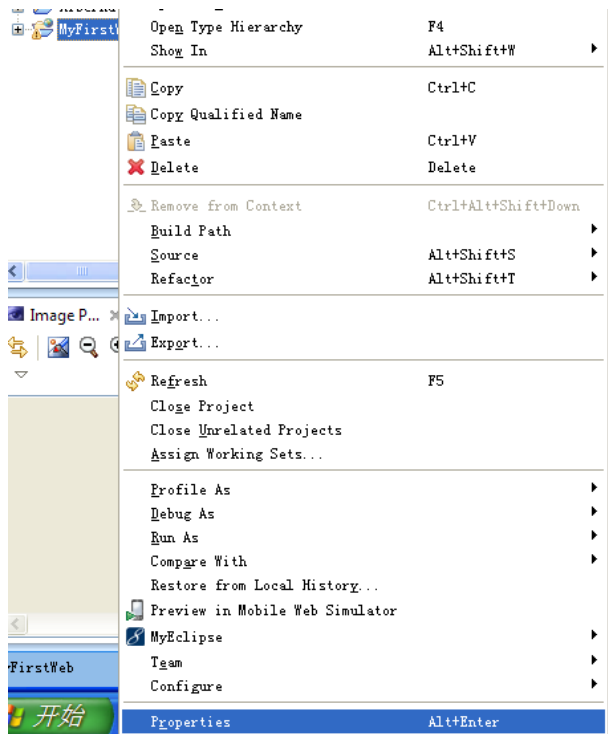


图 1.14 引入外部的 jar 包流程 1

步骤二 在弹出的窗口里的左边选择 Java Build Path，在中间的标签页里选择 Libraries，随后单击右边的 Add External JARs 按钮，如图 1.15 所示。

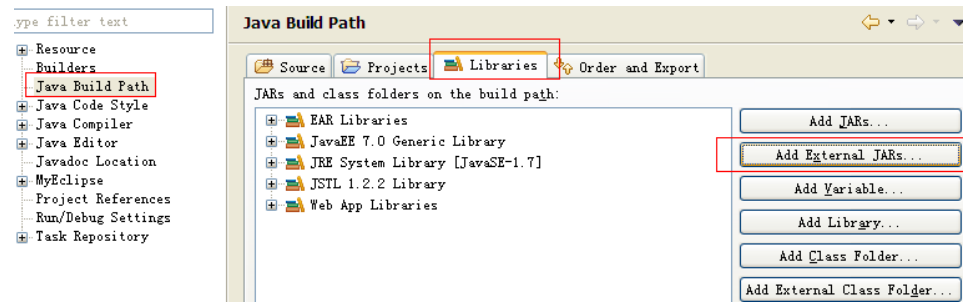


图 1.15 引入外部的 jar 包流程 2

步骤三 在弹出的对话框里选择要添加的 jar 文件即可。

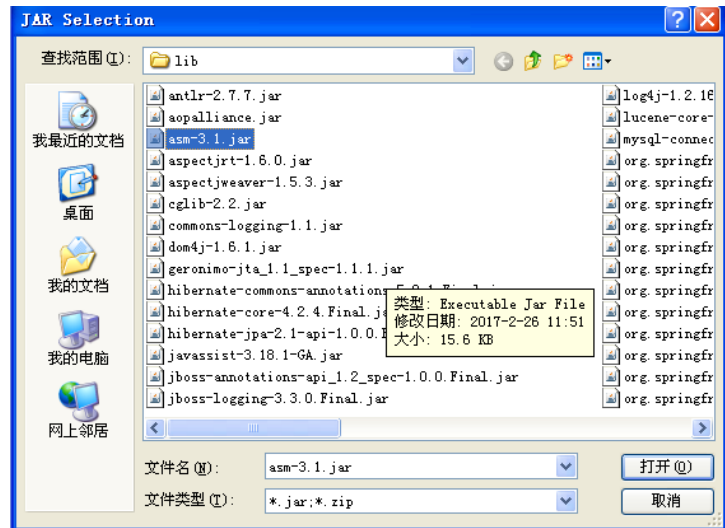


图 1.16 引入外部的 jar 包流程 3

1.3.4 支持中文

在默认的 MyEclipse 设置下，未必能支持中文，这样，一旦在 Java 或者 JSP 等文件里输入中文，可能就没法保存了。我们可以通过如下步骤，让 MyEclipse 支持中文输入。

选中 Window→Preference 菜单，如图 1.17 所示。

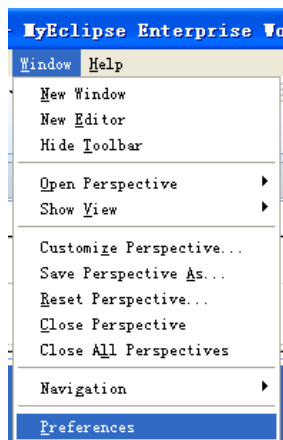


图 1.17 打开 Preferences 菜单

在随后弹出的窗口里，打开 General→Workspace，在 Text file encoding 里设置支持的字符集为 ISO-8859-1（或 UTF-8），这样，MyEclipse 里的所有项目就都能支持中文了。

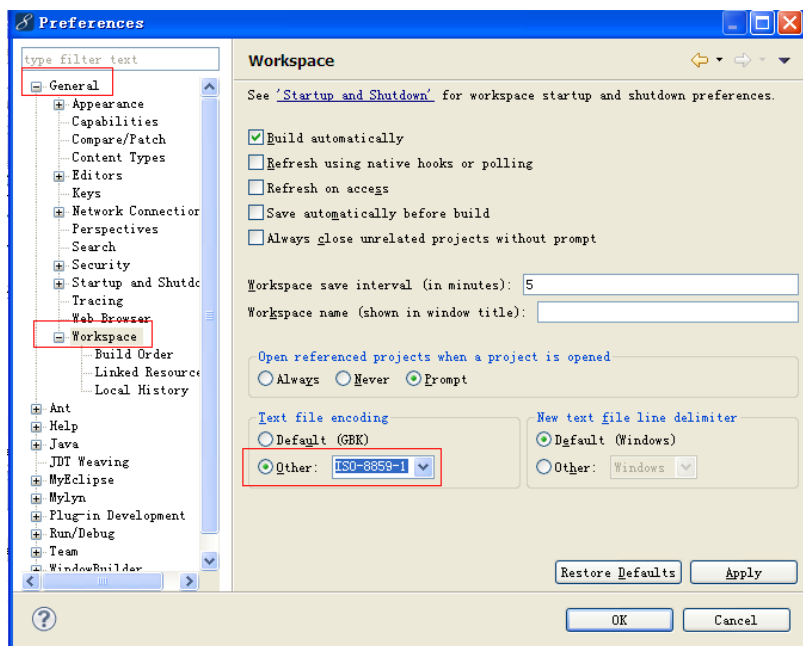


图 1.18 设置中文字符集

这样做的影响范围可能有些大，我们也可以只设置某类文件（比如 JSP 文件或者 Java 文件）支持中文，做法如下。

选中指定类型的文件,比如是 JSP 文件,右击该文件,在随后弹出的菜单里单击 Properties,如图 1.19 所示。

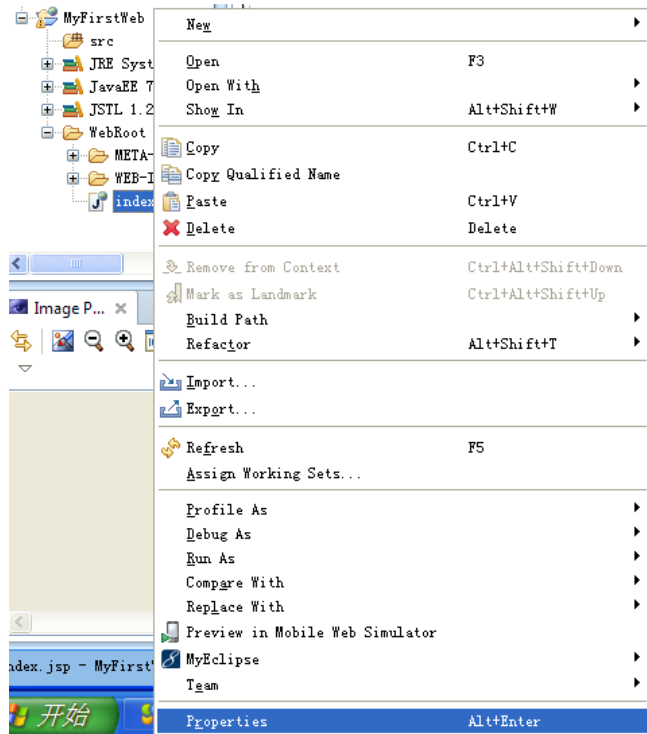


图 1.19 选中指定类型的文件

在随后弹出的窗口里,设置指定的字符集即可,如图 1.20 所示。

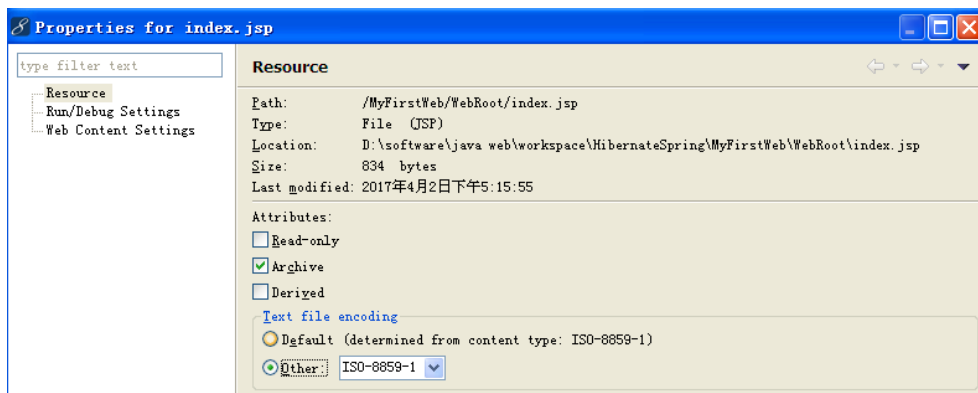


图 1.20 为指定类型的文件设置字符集

1.4 推荐一些经过实践检验的学习方法

作者做了多年的 Java 培训教师，也接触过不少初学者，根据多年的教学互动经验，总结了一些能少走弯路的学习方法，供大家参考。

第一，是要多学多练，这似乎是废话，但真正能非常上心学习的人还真是少数，大多数人下班之后，明明有足够多的时间，但宁可逛街、打游戏而不学习。

在我们所带的班级里，刚开始大家的能力其实差不了多少，但到结业后，常复习、常练习的同学要比不学不练好很多，往往是好的学生能直接跳槽，工资涨至少 20%，不学不练的同学还是老样子。

第二，别单看书或网站上的技术资料，一定得边看边练习。如果光看，知识点一定会在一个月后被忘干净。

培训班里，有些同学确实学习很认真，上课记笔记，相信下课也一定会多复习，但就是不肯多上机练习。到最后，这些同学确实很可惜，虽然用功，但方法不对（不练习），学习效果其实并不好。

第三，关于学习的次序，建议大家遵循技术到框架到细节的路线，比如先大致了解 Spring、Hibernate 等技术，再了解 Spring MVC 等 Web 框架，最后再深入学习 Spring 和 Hibernate 里的各种细节的 API。如果过早地接触各种细节，就可能无法从总体上把握。如果了解框架后不去深入了解细节，则很可能基础知识不扎实，对后继学习非常不好。

第四，在没有了解各种框架（比如 Spring MVC 框架）前，可以先借鉴别人的代码，通过适当地修改先让代码跑通，随后通过阅读代码来理解整体流程和关键代码。

比如我们在给学生讲述 Spring 控制反转时，一般会给学生一个可以运行的程序，让学生先运行通，然后会针对代码讲述装载配置文件和控制反转部分的关键代码，最后会请学生通过改写来实现类似的效果。根据实践情况，学生能很快地掌握 IoC 的流程以及一些关键点，从开始学习到最终掌握大概也就 1 个小时，当然这仅仅是入门级别的学习，深入学习其实也可以采用这套流程。

不过总有些学生在刚开始不熟悉控制反转细节的技术，就想要自己编写各部分的代码（比如配置文件和 Java 代码），美其名曰自己动手能对各部分印象更深。

这样做的大多数后果是，由于不熟悉，自己编写出来的代码往往会错误百出，而且自己还未必知道怎么才能把代码调通，导致最后对这个知识点的学习不了了之。这种情况一旦长久，一定会影响到学习的自信心。

所以千万记得，先“山寨”别人代码来学习，等你熟悉了，有足够多的能力时再亲自动手。

第 2 章

需要了解的数据库知识

对于一个初级程序员来说，对数据表的“增删改查”操作是一定要掌握的，但如果仅仅止步于此，是达不到公司的基本需求的。

具有 3 年相关经验，是初级程序员和高级程序员的分界点，本章将列出大多数公司对具有 3 年经验程序员在数据库方面的要求。通过本章的引导，大家可以找到合适的学习方向，甚至可以通过学习，直接提升自己的能力。

2.1 合理地使用各种连接

本部分将从最简单的“连接”入手，一直讲到“如何建表”。所包含的技术虽然简单，但容易被大多数人忽视，阅读之后，相信你一定能得到实惠。

2.1.1 内连接和左连接

假设我们有两张表，名为 `Student` 的学生表和名为 `studentcourse` 的学生成绩表，它们的结构如表 2.1 和 2.2 所示。

表 2.1 Stu dent（学生表）的结构

字段名	属性	说明
ID In	teger	学号
Name var	char	姓名
Sex var	char	性别
Age Integer		年龄

表 2.2 stu dentcourse（学生成绩表）的结构

字段名	属性	说明
SID Integer		学号
CID Integer		课程编号
Score I	nteger	成绩

在 Student 的学生表里，有如表 2.3 所示的数据。

表 2.3 Stu dent（学生表）的数据

ID Name		Sex	Age
1 M	ike	Male	15
2 T	om	Male	16
3 M	ary	Female	15

而在 studentcourse 的学生成绩表里，有如表 2.4 所示的数据。

表 2.4 stu dentcourse（学生成绩表）的数据

SID CID		Score
1 1		95
1 2		96
2 1		90

下面先给出一个大家一定见过的 SQL 语句：

```
Select * from Student s, studentcoure sc where s.ID = sc.SID
```

这叫隐式的内连接，运行结果如表 2.5 所示。

表 2.5 内连接的运行结果

ID Name		Sex	Age	SID	CID	Score
1	Mike	Male	15 1		1 95	
1	Mike	Male	15 1		2 96	
2	Tom	Male	16 2		1 90	

我们可以看到，前 4 列是属于 Student 表的，而后 3 列是属于 studentcourse 表的。

这句 SQL 语句只返回两张表里学号相同的记录。由于 Mary 没有选修课程，所以没有关于她的记录。

但在实际的项目里，我们的需求是，打印所有学生的成绩，即使 Mary 没有选修，我们也要从报表里反映出这点。

这里我们就需要用到左连接。左连接的含义是：

- ①以左表为主表，返回左表里所有的行，并以此和右表连接。
- ②即使右表没有和左表相对应的行，也需要返回 null 值。

根据这个规则，我们改写一下 SQL。

```
Select * from Student s left join studentcourse sc on s.ID = sc.SID
```

请大家注意，左连接的关键字是 left join，在其左右两边的分别是左表和右表。而且，是用 on 作为关联条件。返回的结果如表 2.6 所示。

表 2.6 左连接的运行结果

ID	Name	Sex	Age	SID	CID	Score
1	Mike	Male	15 1		1 95	
1	Mike	Male	15 1		2 96	
2	Tom	Male	16 2		1 90	
3	Mary	Female	15	null null	null	

大家可以看到，本次结果以 Student 所有记录为基准，其中 Mary 没有选修课，在 studentcourse 表里没有对应的基础，但还是出现在结果里。由于她没有选修课，所以对应的 studentcourse 部分的记录是 null。

在左连接里，我们同样可以加 where 条件，比如上述的语句我们修改一下：

```
Select * from Student s left join studentcourse sc on s.ID = sc.SID where Name = 'Mike'
```

就可以返回所有 Mike 选修的课程的信息。

讲到这里，我们再来介绍两个很相似的连接，**右连接**和**全连接**。

和左连接一样，右连接是以右表为基准，但在实际项目中即使我们要用右连接，我们可以通过交换连接次序把右连接改成左连接，比如：

A right join B，我们一般会改写成 B left join A。

全连接等于左连接的所有行加上右连接的所有行。

在平时的项目里，我们一般是用一个表为基准来关联其他信息的，比如拿一张银行账户表来关联账户流水明细表，可查询一个账户在上个月的所有存款和取款的明细流水账。但很少见到同时用左右两个表作为基准来关联。所以大家只要知道有全连接这个概念即可。

2.1.2 范式和连接的代价

计算机专业的学生一定学过三范式，其他从事软件开发的人应该也知道这个概念。我们在面试应聘者的时候也会随口问一下：“你平时是怎么设计数据表的？”

如果听到的回答是：“我会按照三范式的原则设计表”，那么就会追加一个问题：“你既然按三范式的原则设计表，那么你是怎么考虑表之间的连接代价？”

如果此人对这个问题说不上来，那么我们基本可以确认，他没有参与过数据表的设计。

为什么平时设计中不能完全用到三范式？下面我们来分析一个基本的场景。

有个在线购物网站，根据三范式的要求，为了描述订单信息，我们用至少三张表。

表 2.7 在线购物网站的订单表描述

表名	包含的字段	记录条数
订单流水表	至少包含订单编号、商品编号和下订单的会员编号	假设过去 1 个月有 100 万条
商品表	至少包含商品编号和商品名	假设过去一个月有 50 万条商品信息
会员表	至少包含会员编号和会员邮箱	假设过去一个月里有 10 万名会员下过订单

我们有个简单的需求，列出过去一个月里所有买过 Java 书籍的会员的邮箱，以便我们发些广告。

这条 SQL 语句不复杂，但关键是得“关联”，我们可以用订单流水表 left join 商品表 on 订单流水表的商品编号 = 商品表的商品编号，在 left join 会员表 on 订单流水表的会员编号 = 会员表的会员编号。

关联是要代价的，尤其是当表的数据量比较多的情况下，我们必须在“三范式”和“连接代价”之间权衡一下。

回到一开始的面试题，如果你没有商业项目的经验，你或许就会按教科书上的做法来设

计表，根本不会考虑到这种“代价的权衡”。

2.1.3 表的设计和冗余

“设计过表”则标志着一个初级程序员开始成熟，这里姑且不论数据表的业务逻辑，一般的公司是怎么“合理地设计数据表”？其实在设计一个商业项目的数据表时，离不开“业务需求”。

第一，如果在设计的时候，已经明确地知道这个系统的数据量不会太大，比如一个中学的图书管理系统，最多有 10 万条书本的数据，过去一个月里借阅记录不会超过 1 万条。也就是说，表之间的关联代价不会太高，那么用“三范式”的原则是必需的。毕竟三范式能避免数据冗余带来的更新插入上的“需要同时多表里相同字段”的麻烦。

第二，如果表的数据量很大，如前面举的在线购物网站的例子，我们可能就需要冗余数据。在订单流水表里，同时放入用户邮件地址和商品名的字段。

在得到“免去连接操作”的好处同时，也得付出相应的代价，比如用户一旦更新了邮件地址，那么我们就需要同时在会员表和订单流水表里修改该字段，这就是冗余带来的后果。

除去一些技术点的描述，本篇更想告诉大家的是，我们不仅需要掌握诸如“连接”和“范式”之类的技术，更应该从业务角度，权衡各种“建表代价”，从而挑选一种最符合本项目的解决方案。

在以往的面试官的经验里，有过两个案例，第一位他简历上项目很多，也具有 1 年相关经验，但他说建数据表要符合“三范式”，同时不清楚左连接、内连接等的具体做法，经过细问知道，原来他在简历上的项目是学习项目，而非商业项目。

第二位，虽然工作经验只有半年，但能把“建表需要权衡数据冗余和连接代价”的意思描述得很清楚。这样至少在数据库层面，第二位的得分就比第一位要高。

2.2 不复杂但容易忽视的 SQL 用法

不同的数据库，比如 DB2、MySQL 和 Oracle，它们增删改查的基本 SQL 语句在语法上有细微的差别，但这不是本章要讨论的内容。

我们在面试的时候，一般不会直接问这些基本的语法，而是说出些场景来让面试者写 SQL 语句。所以本节将直接讲述在 SQL 里初级程序员最容易忽视的知识点。

2.2.1 group by 和 Having

groupby 的原始含义是分组，我们来看如表 2.8 所示的一张学生兴趣小组记录表，其中有三个字段：年级、人数和参加的兴趣小组名称。

表 2.8 学生兴趣小组记录表

年级	人数	兴趣小组
3 5		手风琴
3 4		电子琴
4 6		钢琴
4 7		电子琴
5 4		钢琴
5 4		手风琴

从这张表里，我们可以看到，该学校开了 3 个兴趣小组，每个小组都有 3、4、5 年级的同学参加。

第一个需求里需要分析每个年级里参加兴趣小组的同学数量，SQL 语句是：

```
Select 年级 , sum(人数) as 总人数 from 学生兴趣小组记录表 group by 年级
```

返回结果是针对年级的分类统计汇总，如表 2.9 所示。

表 2.9 每个年级参加兴趣小组同学的数量

年级	总人数	说明
3 9		手风琴 5 人加电子琴 4 人
4 13		钢琴 6 人加电子琴 7 人
5 8		钢琴和手风琴各 4 人

在具体的执行过程中，会在内存里分成 3 组（有多少个年级分多少组），然后针对每块计算 sum（人数）。

第二个需求和第一个很相似，是分析每个兴趣小组里的人数，SQL 语句是：

```
Select 兴趣小组 , sun(人数) as 总人数 from 学生兴趣小组记录表 group by 兴趣小组
```

和第一个 SQL 不同的是，这里是根据兴趣小组来分组，返回结果如表 2.10 所示。

表 2.10 每个兴趣小组的同学的数量

兴趣小组	总人数	说明
手风琴 9		3 年级 5 人加 5 年级 4 人
电子琴 11		3 年级 4 人加 4 年级 7 人
钢琴 10		4 年级 6 人加 5 年级 4 人

下面来看一个出错的 SQL 语句，它非常具有典型性：

```
Select 年级, sum(人数) as 总人数, 兴趣小组 from 学生兴趣小组记录表 group by 年级
```

这个错误的原因是违反了如下原则：**select** 指定的字段要么包含在 **group by** 语句里，要么被包含在聚合函数中，在上文里，“兴趣小组”这个字段违反了这个原则。

抛开枯燥抽象的语法，我们从实际角度来分析一下这句 SQL 的错误。

在执行 SQL 语句里，由于 **group by** 年级，所以在内存里分了 3 个组，其中“3 年级”这个组里，“兴趣小组”的值不唯一，那么在结果里就无法返回唯一的值了，这就是错误原因。

再来看第三个需求，这个学校里开了 3 个兴趣小组，领导了解一下哪个组的人数低于 10 人，从而加强一下对这个组的宣传力度。对此，我们可以用 **having** 来实现：

```
Select 兴趣小组, sum(人数) as 总人数 from 学生兴趣小组记录表 group by 兴趣小组
Having sum(人数) < 10
```

having 子句的作用是筛选满足条件的组，即在分组之后过滤数据。

由于是对兴趣小组分组的，所以从表 2.10 中我们能看到，有 3 个组，总人数分别是 9、11 和 10。这里我们通过 **having**，只要求返回 **sum(人数)** 小于 10 的组，那么结果里只包含了手风琴。

兴趣小组	总人数
手风琴 9	

2.2.2 Having 的另一个常用用途——查看重复记录

在表 2.3 的基础上，再简化一下学生信息表，如表 2.11 所示。

表 2.11 简化后的学生信息表的数据

ID Name	
1 M	ike
1 M	ike
2 M	ary
3 T	om

在输入的过程中，老师不小心把 Mike 输入了两次（大家不要感觉奇怪，虽然我们可以通过建立主键约束来避免这类问题，但在项目里经常会遇到这类“重复数据”的问题）。

怎么来看哪些同学被重复输入了呢？

```
Select ID from 学生表 group by ID having count(*)>1
```

这里将根据 ID 分组，如果有重复，该 ID（具体到本案例，是 1）一定会有超过 1 条记录，由此可以查出重复记录。

2.2.3 通过一些案例来看一下常用的 Select 定式

SQL 语句里最常用的是 Select，本章我们将通过一些案例来描述一些常规的写法。常用的 select 语句，无外乎是子查询、in、group by 和 having 的组合。

先建 4 张表。

- ① 学生表：Student(SID,Sname,Sage)，字段的含义分别是学号、姓名和年龄。
- ② 课程表：Course(CID,Cname,TID)，字段的含义分别是课程编号、课程名和老师编号。
- ③ 成绩表：SC(SID,CID,score)，字段的含义分别是学号、课程号和成绩。
- ④ 老师表：Teacher(TID,Tname)，字段的含义分别是老师编号和老师姓名。

第一个案例需要查询每门功课都及格（分数大于 60 分）的学生信息，请注意，是所有功课都及格，如果某学生语文及格但数学没及格，就不符合要求。

SQL 语句如下：

```
select SID,Sname,Sage from student where sid not in  
(select sid from sc where score <60)
```

首先通过第 2 行的语句，查询出所有不及格学生的学号，然后用一个 not in，把这些学号排除在外。这里请大家注意 in 和子查询的写法。

第二个案例需要查询“语文”成绩比“数学”成绩差的学生学号和姓名。

这里需要用到多重嵌套，而且 select 出来的结果也可以作为字段或者是目标表。

SQL 语句如下。

```
1 select sid, sname from  
2 (select student.`sid`, student.sname, score,  
3     (select score from sc sc_2 where student.`sid`=sc_2.`sid` and  
4     sc_2.`CName`='语文') score2 from student , sc where sc.`sid`=student.`sid`  
5 and sc.`CName`='数学') s_2 where score2 < score
```

这个 SQL 语句比较复杂，在第 3 行和第 4 行的一个括号里，返回所有“语文”课的成绩 score2，而在第 2 行到第 5 行的括号范围里，用 score 返回数学成绩，最终用一个 score2<score 来筛选满足条件的记录。

第三个案例需要查询都学过 2 号同学学习过的课程的同学的学号。

```
select `Sid` from SC where `CID` in (select `Cid` from SC where `Sid`=2)
      group by `Sid` having count(*)=(select count(*) from SC where `Sid`=2)
```

这里用两个条件来保证符合要求：

- ① 在第 1 行里，将找到所有学过 2 号同学课程的同学。
- ② 在第 2 行里，用 `count(*)` 来保证这些同学学过的课程数量和 2 号同学课程数量一样多。

2.3 索引的用途和代价

索引是数据库优化所必需的工具，我们在面试的时候一般不会问概念性的问题，因为大家都能从教科书上找到答案，所以一般会问以下两方面的问题：

① 索引有什么代价？哪些场景下你需要建索引？或者有时候反过来问，哪些场景下不推荐建索引。

② 建好索引之后，怎么才能最高效地利用索引？或者反过来问，请说出一个无法有效利用已建索引的案例。

这些问题虽然不难回答，但对初学者（尤其是刚出校门的大学生）而言基本都回答不出来，因为他们根本没往这方面考虑过。

2.3.1 从结构上来分析索引的好处和代价

索引好比是一棵 B 树，假设学生表里只有学生 ID 和姓名两列，该学生表里有 1000 个学生，学号分别从 1 到 1000，如果针对 ID 建立索引，大致的结构如图 2.1 所示。

当然，在实际的数据库系统中，索引要比这个复杂得多，但从这个图里，我们能大致看出索引的工作原理。

索引建好后，如果我们要查找 ID 为 111 的学生，则数据库系统就会走索引，从图 2.1 中我们可以看到，根据根节点的指引，会找到第二层从左往右第二个数据块，依此类推，会在第四层里得到 ID 为 111 的物理地址，然后直接从硬盘里找数据。

反过来，如果没有建索引，数据库系统可能就要从一个大的范围里逐一定位查找，效率就没这么高了。

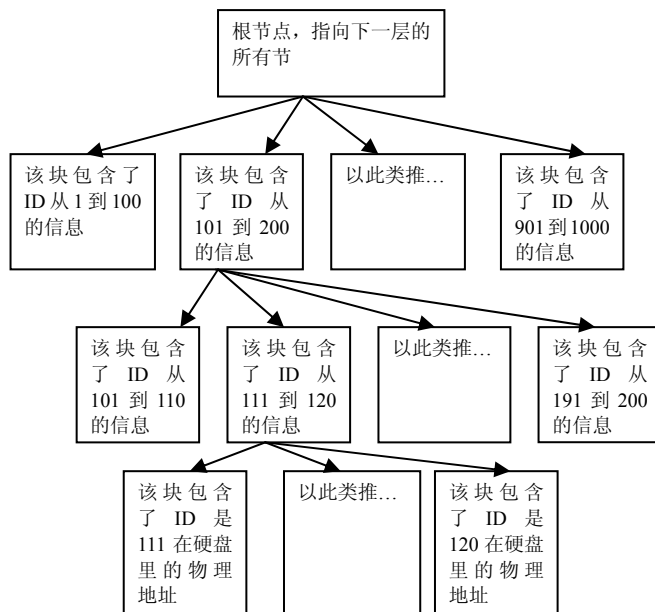


图 2.1 索引的结构图

索引的好处大家已经看到了，那么为了得到这个“查询效率高”的好处，我们要付出了什么样的代价呢？

- 索引需要占硬盘空间，这是空间方面的代价。
- 一旦插入新的数据，就需要重新建索引，这是时间上的代价。

2.3.2 建索引时我们需要权衡的因素

场景一，数据表规模不大，就几千行，即使不建索引，查询语句的返回时间也不长，这时建索引的意义就不大。当然，若就几千行，索引所占的空间也不多，所以这种情况下，顶多属于“性价比”不高。

场景二，某个商品表里有几百万条商品信息，同时每天会在一个时间点，往其中更新大概十万条左右的商品信息，现在用 `where` 语句查询特定商品时（比如 `where name = 'XXX'`）速度很慢。为了提升查询效率可以建索引，但当每天更新数据时，又会重建索引，这是要耗费时间的。

这时就需要综合考虑，甚至可以在更新前删除索引，更新后再重建。

场景三，从图 2.1 中可以看到，因为在数据表里 ID 值都不相同，所以索引能发挥出比较

大的作用。相反，如果某个字段重复率很高，如性别字段，或者某个字段大多数值是空（null），那么不建议对该字段建索引。

请大家记住，一定是有业务需求了才会建索引。比如在一个商品表里，我们经常要根据 name 做查询，如果没有索引，查询速度会很慢，这时就需要建索引。但在项目开发中，如果不经常根据商品编号查询，那么就没必要对编号建索引。

最后强调一点，建索引是要付出代价的，没事别乱建，同时在一个表上也不能建太多的索引。

2.3.3 索引的正确用法

还是用商品表来讨论，如果针对商品名（Name）这个字段建了索引，那么有些 SQL 语句是无法发挥出索引优势的，换句话说，如果出现一些不好的 SQL 语句，那么索引就白建了。下面通过一些具体的例子来看索引的正确用法。

① 语句一：select name from 商品表。

不会用到索引，因为没有 where 语句。

② 语句二：select * from 商品表 where name = 'Java 书'

会用到索引，如果项目里经常用到 name 来查询，且商品表的数据量很大，而 name 值的重复率又不高，那么建议建索引。

③ 语句三：select * from 商品表 where name like 'Java%'

这是个模糊查询，会用到索引，请大家记住，用 like 进行模糊查询时，如果第一个就是模糊的匹配符，比如 where name like '%java'，那么在查询时不会走索引。在其他情况下，不论用了多少个%，也不论%的位置，只要不出现在第一个位置，那么都能用到索引。

下面再用学生成绩表来分析另外三种错误的索引用法。

学生成绩表里有两个字段：姓名和成绩。现在对成绩这个整数类型的字段建索引。

① 第一种情况，当数字型字段遇到非等值操作符时，无法用到索引。比如：

```
select name from 学生成绩表 where 成绩>95
```

一旦出现大于符号，就不能用到索引，为了用到索引，我们应该改一下 SQL 语句里的 where 从句：

```
where 成绩 in (96,97,98,99,100)
```

② 第二种情况，如果对索引字段进行了某种左值操作，那么无法用到索引。

能用到索引的写法：`select name from 学生成绩表 where 成绩 = 60`

不能用到索引的写法：`select name from 学生成绩表 where 成绩+40 = 100`

③ 第三种情况，如果对索引字段进行了函数操作，那么无法用到索引。

比如 SQL 语句：`select * from 商品表 where substr(name)='J'`，我们希望查询商品名首字母是 J 的记录，可一旦针对 `name` 使用函数，即使 `name` 字段上有索引，也无法用到。

2.4 让你的 JDBC 代码更专业

虽然我们在本书的后面部分会用 Hibernate 之类的 ORM 来访问操作数据库，但从效率上来看，用 JDBC 操作数据库会更加高效。

作为一个面试官，我们也会问一些 JDBC 的问题。原因是这些基本的知识点，第一得了解，第二得深入了解。

2.4.1 用 try...catch...finally 从句

我们知道，在这个从句的结构里，不论是否发生异常，不论发生何种异常，`finally` 从句一定会执行。

根据这个特性，需要把关闭 JDBC 对象部分的代码写到 `finally` 从句里。

同时，在 `try...catch` 里应该注意如下三点：

- 第一，不能直接用 `Exception` 来接收所有异常，应当先用专业的异常处理类，比如 `SQLException` 来接收，最后再用 `Exception` 来做最后的防守。
- 第二，在 `catch` 从句里，别什么都不做，也别直接抛出异常了事，应该返回一些有可操作性的语句，提示用户在遇到异常时该怎么办，比如给出联系人的电话。
- 第三，因为监听和检测异常是需要代价的，所以应当尽量缩小 `try...catch` 的范围，只包括必要的代码即可，而不是把整个函数都包含在 `try...catch` 从句里。

大家别小看上述非常通俗易懂的原则。作者在某个大公司工作时，曾加过一个项目的评审团队，该公司所有的项目在上线前都要经过这个团队评审打分，如果分数达不到标准，就需要整改代码。

刚提到的这些原则都是具体的打分项，这个大公司是著名外企，其中的员工都很优秀，

但在异常处理点上失分是普遍现象。

下面通过一个简单的查询例子 `ResultDemo.java` 来观察下标准写法。

有个 `Student` 表，其中有两个字段，均为字符类型的学号 `ID` 和姓名 `Name`：

```

1  import java.sql.Connection;
2  import java.sql.DriverManager;
3  import java.sql.ResultSet;
4  import java.sql.SQLException;
5  import java.sql.Statement;
6  public class ResultDemo {
7      public static void main(String[] args) {
8          try {
9              //需要确保支持MySQL的jar能被读到
10         Class.forName("com.mysql.jdbc.Driver");
11         } catch (ClassNotFoundException e) {
12             System.out.println("Where is your MySQL JDBC Driver?");
13             e.printStackTrace();
14             return;
15         }
16         Connection connection = null;
17         Statement stmt = null;
18         try {
19             //这里是连接字符串
20             connection = DriverManager.getConnection(
21                 "jdbc:mysql://localhost:3306/class3", "root", "123456");
22             if (connection != null) {
23                 stmt = connection.createStatement();
24                 String query = "select ID,Name from student";
25                 ResultSet rs=stmt.executeQuery(query);
26                 while (rs.next())
27                 {
28                     System.out.println(rs.getString("ID"));
29                     System.out.println(rs.getString("Name"));
30                 }
31             } else {
32                 System.out.println("Failed to make connection!");
33             }
34         } catch (SQLException e) {
35             System.out.println("Check the JDBC Driver or Connection!");
36             e.printStackTrace();
37         } finally {
38             //close the connection
39             try {

```

```
40     stmt                .close();
41     conn                ection.close();
42     } catch (SQLException e) {
43         e.pr              intStackTrace();
44     }
45 }
46 }
47 }
```

上述代码的业务非常简单，连接数据库后依次打印 **Student** 表里的 **ID** 和 **Name** 信息。但请大家关注一下这段代码带给我们的启示。

第一，在短短的业务逻辑里，我们分别在第 8 到第 15 行，第 18 到第 44 行，用了两块 **try...catch**，而没有图省事用一块 **try...catch** 代码包含所有的业务方法，这遵循了“尽量缩小检测范围”的原则。


第二，在第 35 和 36 行的 **catch** 从句里，没有简单地抛出异常了事，而是输出了一些信息，根据这些信息，调试程序的开发人员能很快从中得到提示，从而很快地找到原因。

此外，在 **catch** 从句里，也可以输出一些面向使用者的提示信息，比如让使用者重启程序，总之一句话，需要把面向 **Java** 的异常翻译成让程序员或使用者能理解的提示信息。

第三，在第 37 到第 44 行的 **finally** 从句里，关闭了连接，因为不论发生了什么异常，或者是否发生异常，**finally** 从句一定会被执行到，所以可以把关闭连接的代码放入其中。如果不关闭连接，这个数据库连接对象是无法被回收的（**Java** 的垃圾回收机制也无法回收）。

2.4.2 预处理和批处理

预处理除了可以提升效率，还能避免 **SQL** 注入，从而保证系统的安全。例如有如图 2.2 所示的登录界面。



The image shows a simple login interface. It consists of two text input fields. The first field is labeled 'User Name' and the second is labeled 'User Password'. Below these fields is a button labeled 'submit'.

图 2.2 登录界面

我们一般用如下 **SQL** 来验证身份：

```
Select userName from users where username = '输入的用户名' and pwd = '输入的密码'
```

一般来说，如果用户名和密码不匹配，是无法通过验证的，但有人可以在 **User Name** 里

输入 1，在 User Password 部分输入：

```
1' and pwd = '1' or '1'='1
```

那么整个 SQL 语句就会变成：

```
Select userName from users where username = '1' and pwd = '1' or '1'='1'
```

这样就能绕过验证。

而处理对象 **PreparedStatement** 能有效防止这个现象的发生，因为一个“?”就是一个占位符，无法扩展。下面来看一个预处理和批处理结合的例子，同样用到 2.4.1 节里提到的 **Student** 表，这次来批量插入数据。

```
1  import java.sql.Connection;
2  import java.sql.DriverManager;
3  import java.sql.PreparedStatement;
4  import java.sql.SQLException;
5  import java.sql.Statement;
6  public class JDBCBatch {
7      public static void main(String[] args) {
8          try {
9              Class.forName("com.mysql.jdbc.Driver");
10             } catch (ClassNotFoundException e) {
11                 System.out.println("Where is your MySQL JDBC Driver?");
12                 e.printStackTrace();
13             }
14             return;
15             Connection connection = null;
16             PreparedStatement pstmt;
17             try {
18                 //这里是连接字符串
19                 connection = DriverManager.getConnection(
20                     "jdbc:mysql://localhost:3306/class3", "root", "123456");
21                 if (connection != null) {
22                     String query = "insert into student values (?,?)";
23                     pstmt = connection.prepareStatement(query); //开始设置参数
24                     pstmt.setString(1, "1");
25                     pstmt.setString(2, "Peter");
26                     pstmt.addBatch();
27                     //设置第二个参数
28                     pstmt.setString(1, "2");
29                     pstmt.setString(2, "Mike");
30                     pstmt.addBatch();
31                     //执行批处理
```

```
32     pstmt        t.executeBatch();
33   } else {
34     Syst          em.out.println("Failed to make connection!");
35   }
36   } catch (SQLException e) {
37     Syst          em.out.println("Some of Students were not inserted correctly, please
check the student table and insert manually.");
38     e.pr          intStackTrace();
39   } finally {
40     try           {
41       conn        ection.close();
42     } catch (SQLException e) {
43       e.pr         intStackTrace();
44     }
45   }
46 }
47 }
```

这部分的逻辑也比较简单，用 `insert` 语句批量插入数据。但请大家注意如下两点：

① 在 `PreparedStatement` 里，占位符的编号是从 1 开始，而不是从 0 开始。

② 批量操作能提升效率，但一次性操作多少，才能让效率提升到最高？这在不同的数据库里是不同的，一般是每批操作 500 到 1000 条语句。但切记，别一次性把所有的 `insert` 语句都用 `addBatch` 放入，因为如果 SQL 语句过多，会撑爆缓存，从而出错。

`PreparedStatement` 是个比较重要的 JDBC 对象。我们在面试的时候，有时会问这个问题：`Statement` 和 `PreparedStatement` 有什么差别？答案要点是 `PreparedStatement` 能预处理，如果能展开一下，说明能防止 SQL 注入就更好了。

2.4.3 事务的提交与回滚

事务是 JDBC 乃至数据库操作的一个重要概念，在 JDBC 里，一般采用如下方法使用事务。

① 通过 `setAutoCommit`，设置非自动提交。在 JDBC 里，一般默认是自动提交，即有任何增删改的 SQL 语句都会当场执行。如果大家设置了非自动提交，记得在用好事务后设置回“自动提交”。

② 在合适的地方用 `connection.commit()` 来提交事务。一般是在执行结束时提交。

③ 可以通过 `connection.rollback()` 来回滚事务，回滚语句应放在 `catch` 里。一般是有异常

了，再回滚事务。

④ `Connection` 提供了 `setSavepoint` 和 `releaseSavepoint` 两个方法，用它们可以设置和释放保存点，这样 `rollback` 就会到指定的位置，而不是事务的开始位置，但不推荐这种做法。

下面改写一下前面插入 `Student` 表的代码，引入事务。

```

1  try {
2      //这里是连接字符串
3      conn     ction = DriverManager.getConnection(
4          "jdbc      c:mysql://localhost:3306/class3", "root", "123456");
5          if (connection != null) {
6              //设置非自动提交，开始用事务的方式提交
7              connection.setAutoCommit(false);
8              String query = "insert into student values (?,?)";
9              pstmt      t = connection.prepareStatement(query);
10             pstmt      t.setString(1,"1");
11             pstmt      t.setString(2,"Peter");
12             pstmt      t.addBatch();
13
14             pstmt      t.setString(1,"2");
15             pstmt      t.setString(2,"Mike");
16             pstmt      t.addBatch();
17             pstmt      t.executeBatch();
18             //提交事务
19             connection.commit();
20         } else {
21             Syst      em.out.println("Failed to make connection!");
22         }
23     } catch (SQLException e) {
24         //在 catch 里，一旦出现异常，需要回滚事务
25         conn     ction.rollback();
26         Syst      em.out.println("Some of Students were not inserted correctly, please
check the student table and insert manually.");
27         e.pr      intStackTrace();
28     } finally {
29         try {
30             conn     ction.close();
31         } catch (SQLException e) {
32             e.pr      intStackTrace();
33         }
34     }

```

其中第 7 行里，通过 `setAutoCommit` 为 `false` 的语句，把事务设置成“手动提交”，在第

17 行的代码里，虽然进行了批量提交，但不会执行，直到第 19 行，运行了 `commit` 后才会把批量提交的数据插入到数据表里。一旦出现异常，会在第 25 行的 `catch` 从句里，通过 `rollback` 回滚事务。

2.4.4 事务隔离级别

在数据库操作中，为了有效保证并发读取数据的正确性，引入事务隔离级别。

在实际开发过程中，需要非常谨慎地使用事务隔离级别，因为一旦使用不当，导致数据库性能低下是小事，还有可能会引起频繁的死锁，所以使用的场合也不多。

但它却是一个重要面试点，理由是对于一个程序员，有必要了解数据库的并发知识。这里我们来总结一下。

在 JDBC 里，有 5 个常量可以用来描述事务隔离级别，级别从低到高，叙述如下。

① 读取未提交：TRANSACTION_READ_UNCOMMITTED，允许脏读、不可重复读和幻读。

② 读取提交：TRANSACTION_READ_COMMITTED，禁止脏读，但允许不可重复读和幻读。

③ 可重读：TRANSACTION_REPEATABLE_READ，禁止脏读和不可重复读，但允许幻读。

④ 可串行化：TRANSACTION_SERIALIZABLE，禁止脏读、不可重复读和幻读。

另外还有一个不支持事务的常量，TRANSACTION_NONE JDBC。

具体概念解释如下：

① **脏读**（dirty read）是指一个事务读取了另一个事务尚未提交的数据。

我们来看一个脏读的例子。假设张三的工资原本是 1000 元，财务人员在某一时刻将他的工资改成 5000 元（但未提交这个修改事务），此时张三读取自己的工资，发现工资变成了 5000 元，非常高兴！但随后，财务人员发现操作有误，回滚了事务，这时 Mary 的工资又变成了 1000 元。

像这样，张三记取的工资数 5000 就是一个脏数据。如果在第一个事务提交前，任何其他事务不可读取其修改过的值，则可以避免该问题。

② **不可重复读**（non-repeatable read），是指一个事务的操作导致另一个事务前后两次读

取到不同的数据。比如，同一查询在同一事务中多次进行，由于其他事务提交了所做的修改（或添加或删除等操作），所以每次查询都会返回不同的结果集，这就是不可重复读。我们也来举一个具体的例子来了解不可重复读。

在事务1中，张三读取了自己的工资为1000元，但针对工资的操作并没有完成。在另外一个事务中，财务人员修改了张三的工资为2000元，并提交了事务。这时在事务1中，张三再次读取自己的工资时，工资就变为了2000元，这就造成两次读的数据不一致了。具体的解决办法是，只有在修改事务完全提交之后，才允许读取数据。

③ 幻读（phantom read），是指一个事务的操作会导致另一个事务前后两次查询的结果不同。比如在事务1里，我们读取到了10条工资是1000元的员工记录，这时事务2又插入了一条员工记录，工资也是1000元，那么事务1再次以“工资是1000元的员工”作为查询条件读取时，就会返回11条数据。解决办法是，在操作事务完成数据处理之前，任何其他事务都不可以添加新数据，则可避免该问题。

从上文的描述中可以看到，一旦设置高级别的事务隔离级别，那么数据库系统就需要采取额外的措施来保证这个设置。

比如设置成禁止脏读的 `TRANSACTION_READ_COMMITTED`（读取提交），那么在数据库系统中，如果有执行修改功能的事务未被提交，那么读数据的操作一直会延后，直至这些事务提交。

再来看一个具体的例子，如果执行修改功能的事务运行时间很长（一个小时以上），假设此时有人想要看网站的数据（比如某货物的价格），那么这个请求就会处于等待状态，相应的这个请求所对应的连接也会一直持续着。以此类推，如果在事务运行的这段时间里来了足够多的请求，那么这些请求的连接同样也不会被释放，当这样的连接请求积累到一定数量时，则足以导致数据库崩溃。

所以对于事务隔离级别，如果你是工作经验低于3年的程序员，则可以停留在理论阶段，了解概念即可。事实上，一个工作了5年的程序员也只是偶尔会用到，因为一旦设置出错，后果就十分严重。

2.5 总结

如果在面试或与资深人事交流的过程中，你能有效合理地展示出本章所给出的一些知识点，那么对你的评价就会是“对数据库有深入了解”，甚至能加上“有设计数据表的经验”，即便你说有过3年商业项目数据库操作的经验，那么别人也能相信。

相反，如果一个工作经验满 3 年的程序员或许动手编程能力不差，但无法在交流沟通过程中证明这点，或者干脆不知道怎么证明，那么对他的评价往往可能是“数据库层面，有过商业项目的经验，但只会些基本的增删改查（顶多再加上会视图存储过程等技术），无法独立担当数据库方面的工作”。

下面列些数据库方面的常见面试问题，大家可以以此来衡量一下对本章知识点的掌握程度。

问题 1，你有没有建表的经验？建表时你是否会遵循三范式？

设计数据表时，需要权衡数据冗余和连接代价，详细内容请参考 2.1.3 节你的描述。

问题 2，你有过哪些数据库的优化经验？

大家可以说用过索引，具体的知识点可以参考 2.3 节。

问题 3，请你叙述一下不应该建索引的场景。

问题 4，like 语句会不会走索引？

问题 5，索引的结构是什么？建索引会有什么代价？我们应该如何权衡要不要建索引？

关于上述三个问题，代价可以参考 2.3 节的描述。

问题 6，事务隔离级别有哪些级别？具体的脏读，幻读和不可重复读的含义是什么？

参考 2.4.4 节的描述。

问题 7，Statement 和 PreparedStatement 对象有什么差别？

大家可以围绕预处理和批处理这两个角度来回答。

以下我们再列些数据库方面比较高级的知识点，这些知识点需要靠项目经验来沉淀，一般高级程序员都能知道，而初学者或初级程序员未必知道，大家可以在面试时，找合适的机会说出来，这样面试官一定会对你刮目相看。

知识点 1，在数据库编程时，尽量用 try...catch...finally 的代码结构，同时在 finally 里放置释放数据库连接等资源的代码，因为如果我们不主动地关闭数据库连接，这部分所占用的内存是无法被垃圾回收器（GC）主动回收的。

知识点 2，为了提升数据库操作的性能，会用到 PreparedStatement 来进行批处理操作，但每批执行的 SQL 语句的数量不能太多，否则会把缓冲区撑爆，一般每批是 500 条左右。

知识点 3，通过 PreparedStatement 的预处理机制，我们可以有效地防止 SQL 注入。

第 3 章

JSP+Servlet+JavaBean 框架

说到 Java Web 开发，JSP、Servlet 和 JavaBean 是绕不开的知识点，确实，其中每个技术都包含了许多语法，学起来很不轻松。

要告诉大家的是，现在用这套框架的项目都是些中小项目，大型项目很少用到它，这套框架的作用是程序员进阶的“敲门砖”，是大家了解后继 Struts、Spring 等框架的基础。

本章不会罗列很多不常用但很难记的语法点，而是只列出平时经常会用到的，并且会通过这个框架让你知道 Web 框架的大致模样。

3.1 只应负责界面显示的 JSP

与单个技术（比如 JSP 或 Servlet）相比，框架（或者说是架构）更在乎“分离”和“重用”，对初学者来说，往往是比较熟悉 JSP（或者 Servlet 或者 JavaBean）的用法（比如具体的 API），而忽视框架带来的好处。

3.1.1 从一个大而全的例子中分析 JSP 的语法

现在需要做一个简单的登录页面，如果用户通过验证，会显示 Welcome 用户名的欢迎词，反之则返回登录页面让用户再次输入，页面效果如图 3.1 所示。

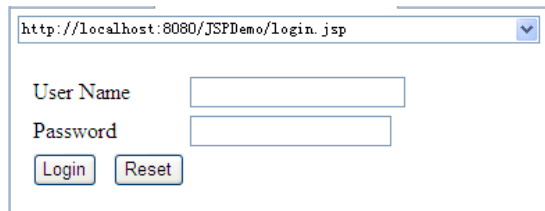


图 3.1 登录界面的效果图

这部分的完整代码是 JSPDemo 项目里的 login.jsp，下面来分析一下关键代码。

代码位置	视频位置
code/第 3 章/JSPDemo	视频/第 3 章/JSP 案例的讲解

根据 JSP 的语法，可以通过@import 来导入需要的 jar 包。这里需要导入支持 MySQL 的库文件，这部分的代码如下所示。

```
1  <%@ page language="java" import="java.util.*" pageEncoding="ISO-8859-1"%>
2  <%@ page import="java.sql.*"%>
3  <%@ page import="com.mysql.jdbc.Driver" %>
```

JSP 相当于在 HTML 页面加上 Java 代码，一般在<body>标签里放入主要代码。

首先来分析一下业务，当进入这个页面的时候，需要判断是否有用户名和密码信息输入，以此来不同的动作，所以在开始部分，需要用一个内嵌对象 request 来完成这个功能，主要代码如下。

```
4  <body>
5  <%
6      if(request.getParameter("username") == null)
7      {
8      %>
9      <form method="post" action="login.jsp" name="userInfo" id="userInfo">
10     <table>
11         <tr>
12             <td>        User Name</td>
13             <td><input type="text" name="username" id="username" /></td>
14         </tr>
15         <tr>
16             <td>        Password </td>
```



```
47 //can login
48 if(rs.next())
49 {
50 %>
51     Welcome<%= rs.getString("username") %>
```

从上述代码中的第 31 和 35 行得到了用户名和密码之后，用第 38 行的 `Class.forName` 载入了 JDBC 的驱动程序，然后在第 43 行定义了一个 `PreparedStatement` 来执行 SQL 语句，并在第 46 行用一个 `ResultSet` 来接收运行结果。

请大家注意，这里用到的 `PreparedStatement` 是为了避免 SQL 注入，这部分的知识大家
可以看本书的数据库相关的内容。

一旦通过第 48 行的 if 语句判断 rs.next() 有返回对象，那么就需要先在第 50 行用 %> 结束 Java 部分的 JDBC 访问数据库的代码，随后在第 51 行显示 Welcome 的字样。这里能看到 JSP 的另外一个语法，用 <%= %> 来显示变量的结果。

如果第 48 行里 `rs.next` 没有返回，也就是说，在数据库里找不到匹配的用户信息，那么就需要返回到登录页面，让用户再次输入登录信息，这部分的代码如下。

[illegible]

```
76     }  
77     }  
78 %>  
79 </body>  
80 </html>
```

这里其实是代码复制，又写了一遍提供用户名和密码输入的 form。

JSP 的语法不算简单，但大家可以有选择性地了解，从而把学习时间用到更重要的知识点（比如框架思想、优化思想）的学习上。

一般来说，Java 高级人才（比如高级开发者或者架构师）的发展方向分为前端和后端，前端主要负责 CSS+DIV、jQuery、Ajax 等和页面设计有关的工作，而后端一般负责数据库连接、业务实现等工作，本书主要集中在后端开发上。

对于一个后端开发人员来说，对 JSP 的要求并不高，需要了解的知识包括：

① 基本语法和 JSP 的代码结构，比如知道在<%%>中包含 Java 代码，用<%= %>显示变量名，再就是一些关于 Page 等的用法。

② 了解常用的一些内嵌对象的用法。必要时，了解一些诸如 Struts 或 JSTL 标签的用法。我们在面试的时候，除了核实一下面试者有没有在项目里用过 JSP，还会了解“如何使用”JSP 的问题：是在 MVC 架构里让 JSP 只负责显示，还是会写出和本部分案例一样“大而全”的代码？

3.1.2 “大而全”和“小而精”

在上述代码给出的例子中，能看到如下的缺陷：

① 出现了代码复制，把供用户输入的 form 复制了两次。

② 频繁地切换 JSP 和 HTML 的逻辑，导致阅读上和开发上的困扰，日后如果别人来维护这部分代码，会很困难。

刚开始开发的时候这种缺陷还不明显，当深入开发的时候，这种缺陷就会爆发。

① 修改点一：我们需要调整供用户验证身份的 Form 代码，加入验证码的输入框。

我们不得不修改两个重复且相同的代码，这样不仅会增加工作量，而且一个疏忽就会导致忘记修改其中一个，增加代码维护的难度。

② 修改点二：要改成 Oracle 数据库，而且数据表名和字段都变了。

我们不仅需要修改 JDBC 的代码，而且还要修改 Welcome xxx 这部分的显示代码，也就是说，数据库方面的修改会直接牵涉其他类型的业务代码。

③ 修改点三：需要实现“三次验证不过就要锁用户”功能。

我们需要在 JDBC 部分的代码计数，如果登录次数小于 3，那么就需要重复性地复制 Form 的代码。如果超过三次，则还需要在 Java 代码里夹杂一个“提示锁屏”的显示页面，这会导致 JSP 更加混乱。

不是危言耸听，我们见过不少页面数量小于 10 的小项目，开发人员为了省事，直接在 JSP 里放入所有的功能，就像前面给出的例子一样，交货两三个月后，当完成用户提出若干改进意见后，这些个 JSP 代码就变得像天书一样，阅读性很差。

在提面试问题的时候，如果发现候选人使用“大而全”的开发方式，我们会让面试者说出理由，如果理由类似“项目紧，缺人手，权衡下来宁可牺牲代码的质量”，那么说明面试者至少和 MVC 开发模式比较过，可以接受。如果面试者直接不知道 MVC 的开发模式，甚至不知道“大而全”的缺点，那么我们的评价至少是“没架构意识”。

如何改进？分解业务，用分层的方式来分解不同类型的业务。

具体来说，在 JSP 页面里，剥离与显示无关的代码，一个好的 JSP 页面里，应该少见甚至不用`<%%>`包含起来的 Java 代码。

3.2 让 Servlet 承担控制器的角色

在平时的项目开发过程中，Servlet 一般不单独使用，而是会和 JSP 以及 JavaBean 等组件配合使用。Servlet 里的语法不少，大家应重点掌握负责“跳转”的功能。

3.2.1 基本知识点

Servlet 是运行在服务器端的 Java 应用程序，由 Web 服务器（比如 Tomcat）加载。Servlet 通过 Web 服务器接收来自客户端（一般来自 JSP 或者 html 页面）的请求，然后把请求转发到业务层，一般是 JavaBean 或者是包含业务逻辑的 Service 层，当接收到运行结果后，把结果返回给客户端。

Servlet 的基本运行流程如下：

① 客户端通过 HTTP 协议向 Web 服务器发送请求。Web 服务器接收该请求并将其发给

Servlet。如果这个 Servlet 尚未被加载，则 Web 服务器将把它加载到 Java 虚拟机并执行它。

② Servlet 接收该 HTTP 请求并执行相应的处理。

③ Servlet 向 Web 服务器返回应答。Web 服务器把从 Servlet 收到的应答发送给客户端。

3.2.2 生命周期与多线程运行方式

Servlet 的生命周期有三个，当被服务器实例化后，容器运行它的 `init` 方法；当请求（Request）到达时，运行其 `service` 方法，`service` 方法会运行与请求对应的 `doXXX` 方法（`doGet`、`doPost`）等；当服务器决定将实例销毁时，调用其 `destroy` 方法。

由于并发处理是 Web 开发的一个重要因素，即一个资深的程序员必须要考虑到这个因素，所以我们在面试的时候会问这样一个问题，当多个请求同时到达时，会启动一个还是多个 Servlet 来接收请求？请大家记得，Servlet 是“单实例多线程”。

- 这是个包含 Servlet 生命周期和线程安全的综合性问题，下面总结的知识点对工作满 5 年的资深程序员都不算过时：第一，可以有多个 Servlet 来处理业务请求，比如针对登录，或者针对订单提交，可以定义两个 Servlet。
- 第二，Servlet 是单实例的，对于同一种业务请求只有一个是实例。当 Web 服务器运行时，会读取 `web.xml` 的内容，加载所定义的 Servlet，当然，加载时会调用各自的 `init` 方法。
- 第三，同一个 Servlet 可以同时处理多个客户端的请求，比如同时有两个用户 Mike 和 Jack 登录时，会启动两个负责登录的 Servlet 线程，并通过触发 `Service` 方法来处理请求。在两个不同的线程里，接收到的用户名是不同的，也就是说，多个线程里的 Servlet，有可能其中的变量不相同。

虽说上述第三点描述的操作是符合业务逻辑的，因为在很多场合下，不同请求里包含的信息是不同的（大家很少看到有两个商品，其内容、价格、购买时间等因素都一样的订单），但这样会产生线程安全性方面的问题。

如果面试者有 3 年工作经验，我们还会追问一个问题，Servlet 是线程安全的还是不安全的？如果要保证线程安全，该怎么做？

在 Servlet 里，负责保存上下文的 `ServletContext` 和负责处理 Session 对象的 `HttpSession` 是线程不安全的，而处理请求的 `ServletRequest` 是线程安全的。也就是说，大家不用担心不同的 Servlet 接收到的请求（比如用户名）会因为多线程因素而被篡改。但如果多个线程企图同时往内存里（比如 Session 和 `ServletContext`）存放相同的数据时，会导致内容冲突。

所以刚才问题的结论是，Servlet 是线程不安全的，在并发请求数量多的情况下（比如同时有很多人登录），某个 Servlet 内的实例变量可能被其他 Servlet 线程错误修改。这是大家享受多线程带来高效的同时所无法避免的问题。下面说一些保证线程安全的做法。

① 有一个 `SingleThreadModel` 接口，如果一个 Servlet 实现了这个接口，那么 Servlet 容器将保证在一个时刻仅有一个线程可以在给定的 Servlet 实例的 `service` 方法中执行。但由于效率低下，这个接口现在已经被 Servlet 规范抛弃了。

② 由于 `ServletRequest` 是线程安全的，每个 Servlet 线程会创建自己的 `ServletRequest` 对象，所以尽量在 Servlet 中使用局部变量，即有单属于本 Servlet 的对象，比如：

```
String user=request.getParameter("user");
```

③ 在多个 Servlet 中，如果要同时对外部对象（比如文件）进行修改，则一定要加锁，做到互斥的访问。但由于 Servlet 一般只负责跳转，不大负责业务处理，所以这点在实际开发过程中很少用到。

④ 对于需要保护的实例，用 `synchronized` 加以保护，但需要尽量缩小保护的范围。对于这个描述可用如下的代码进行说明。

```
1  public class ConcurrentServlet Test extends HttpServlet {
2      PrintWriter output;
3      public void service (HttpServletRequest request,
4                          HttpServletResponse response) throws ServletException, IOException
5      {
6          //username 是有局部变量 request 产生的，所以是线程安全的
7          username = request.getParameter ("username");
8          synchronized (this){
9              output = response.getWriter ();
10             try {
11                 Thread. Sleep (5000);
12             } Catch (InterruptedException e){}
13             output.println("用户名:"+Username+" ");
14         } //end of catch
15     } //end of service function
16 } //end of ConcurrentServlet
```

其中第 2 行定义的 `output` 是负责输出的变量，在多个 Servlet 里可能被共享。

为了避免这共享，我们用 `synchronized` 来保护，但大家可以看到，保护范围仅限于 `output` 输出本身，没有牵涉其他无关的代码。

息，用户可以通过这个表单里第 19 行类型为“submit”类型的登录按钮，提交登录信息。

第 7 行里，通过 form 元素的 action 属性，设定了表单要跳转到的页面，此处提交到的页面是 /servlet/ 目录下的 Validator，它是一个 Servlet 程序，这里请大家注意提交的路径。

而 form 元素的 method 属性用于设定提交页面的方式：一种是 POST 方式，另一种是 GET 方式。前者将表单中的值包含在 form 表单中提交到另一页面，后者将表单值作为 URL 的一部分传递给另一个页面。

纵观上述的 JSP 代码，是用 JSP 文件来显示登录所用的一些显示界面，并用 form 里 action 的方式，指明了本页面在被提交后所要跳转到的 Servlet 页面，而并没有出现验证身份的业务逻辑代码，这种“只让 JSP 负责显示逻辑”的做法是符合“逻辑分离”原则的。

请大家把这个程序和之前的“大而全”程序对比一下，这个程序由于只包含了显示逻辑，所以一旦要修改界面，将不会牵涉无关的（比如数据库连接和验证身份方面）逻辑。

步骤二 编写一个 Servlet 的代码，Validator.java，具体做法是，在刚才创建的项目里创建一个 Servlet 程序，其实也是个 Java 程序，代码如下。

```
1  省略必要的 import 代码
2  public class Validator extends HttpServlet {
3      private static final long serialVersionUID = 1L;
4      public Validator() {
5          super();
6      }
7      protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
8          // TODO Auto-generated method stub
9      }
10
11     protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
12         //      setContentType() 方法用与设定字符编码，如果要显示中文，可以将 charset 设定为
13         //          GB18030
14         response.setContentType("text/html;charset= GB18030");
15         PrintWriter out = response.getWriter();
16         //使用 request.getParameter() 方法获取表单中的值
17         String username = request.getParameter("username");
18         String password = request.getParameter("password");
19         out.println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01
Transitional//EN\">");
20         out.println("<HTML>");
21         out.println("  <HEAD><TITLE>验证页面</TITLE></HEAD>");
```

```

22     out.        println(" <BODY>");
23     //判断用户是否填写用户名和密码
24     if(u        sername.length()<1||password.length()<1)
25     {
26         out.        println("请输入您的用户名和密码,<a href='./login.jsp'>点击此处返回
</a>");
27         //判断用户是否是 hacker, 如果是, 则显示禁止访问页面的信息
28     }
29     else        if(username.equalsIgnoreCase("hacker"))
30     {
31         out.        println("很抱歉, 您禁止登录此页面<a href='./login.jsp'>点击此处返回
</a>");
32         //通过验证, 则表示正常登录, 页面跳转至 welcome.jsp 页面
33     }
34     else
35     {
36     resp        onse.sendRedirect("../welcome.jsp?username="+username);
37     }
38     out.        println(" </BODY>");
39     out.        println("</HTML>");
40     out.        flush();
41     out.        close();
42 }
43 }

```

前文里提到, 根据 Servlet 的生命周期, 应该在其中实现 init、service 和 destroy 三个方法。

但在传统的项目开发里, 我们一般让 Servlet 程序继承 HttpServlet, 就像第 2 行定义的那样: Validator extends HttpServlet

这样就可以只需实现 doPost 或 doGet 方法, servlet 会根据接收到的请求的种类调用其中的一个。

第 11 行的 doPost()方法里实现了身份验证的逻辑代码, 这个 doPost 可以说是 Servlet 里的 main 方法, 当有其他代码跳转到这个 Servlet 里请求时, 就从这个方法开始执行。如果在登录页面 login.jsp 中, 将 form 的表单属性 method 设定为 GET, 则 Servlet 将自动执行定义在第 7 行的 doGet()方法, 而不会执行 doPost()方法。

上述的 doPost 方法有 HttpServletRequest 和 HttpServletResponse 两个类型的参数, 用来存储根据 HTTP 协议接收和发送的信息。

在这个方法里, 首先是通过 request.getParameter()方法, 在第 17 和 18 行里获得了从 login.jsp 里提交过来的用户名和密码信息, 其次从第 19 行开始, 通过 out.println 的方式, 动

态地向浏览器输出返回界面。

这里如果用户名或密码为空，则通过第 26 行的如下代码，向用户提供一个用于返回的超链接。

```
out.println("请输入您的用户名和密码 , <a href='../logon.jsp'>点击此处返回</a>");
```

如果输入的用户名是“hacker”，则会执行第 29 行的 else 流程，通过输出类似的超链接，让用户返回到 login.jsp 页面。

在其他场景下，通过第 36 行的如下语句，跳转到欢迎页面。

```
response.sendRedirect("../welcome.jsp?username="+username);
```

步骤三 开发欢迎界面 Welcome.jsp，代码相对简单，为了支持中文显示，在第一行同样指定了 pageEncoding。

```
1  <%@ page language="java" pageEncoding="GB18030"%>
2  <body>
3  欢迎<%=request.getParameter("username") %> : <br>
4  <h1> 您已成功登录! </h1>
5  </body>
```

步骤四 需要在 Web 服务器里配置 Servlet，具体而言，需要在 web.xml 里做如下的定义，关键代码如下。

```
1  <servlet>
2  <description>This is the description of my J2EE component</description>
3  <display-name>This is the display name of my J2EE component</display-name>
4  <servlet-name>Validator</servlet-name>
5  <servlet-class>Validator</servlet-class>
6  </servlet>
7
8  <servlet-mapping>
9  <servlet-name>Validator</servlet-name>
10 <url-pattern>/servlet/Validator</url-pattern>
11 </servlet-mapping>
```

在 login.jsp 里调用 servlet 的代码如下：

```
<form name="form1" action="../servlet/Validator" method="POST">
```

此时，Web 服务器通过阅读 web.xml 里第 8 行的 servlet-mapping 标签，知道如果调用路径是/servlet/Validator，那么对应的 Servlet 的 name 是 Validator。并且从第 1 行到第 6 行的 <servlet>标签里，知道 servlet-name 是 Validator 的 Servlet 所对应的 servlet-class 是 Validator，

所以 login.jsp 的请求会由 Validator.java 这个类来处理。

3.2.4 运行结果和 JSP+Servlet 模式的说明

运行这个程序的结果，我们打开 login.jsp 后，能看到如图 3.2 所示的页面。

用户名:

密码:

图 3.2 登录页面

只要输入的用户名不是 hacker，那么就能进入到欢迎页面，如图 3.3 所示。

欢迎java:

您已成功登录！

图 3.3 登录成功后的欢迎页面

通过对比，能看到 JSP +Servlet 的好处，那就是“分离了显示和跳转”，但在实际开发过程中，这种程度的分离还不够。

在上述代码里，Servlet 里还负责了身份验证的逻辑，如果用户名是 hacker，那么需要跳转回登录页面，如果不是，才能进入欢迎页面。

还需要专门一层来存放业务相关的代码，如果在 Servlet 里放入大量的业务相关代码，则不仅会再次遭遇到前面“大而全”JSP 代码类似的问题，而且如果一旦访问量增大，还会遇到并发方面的“对象混乱”问题。

在讲述 Servlet 多线程相关的知识点时，我们知道在 Servlet 里需要尽量少地放实例对象，因为 Servlet 不是线程安全的，在高并发的情况下，某个 Servlet 的对象会被其他同类错误地修改掉。

出于上述原因，单纯 JSP+Servlet 的开发模式也不是一个好的选项。

我们在某大公司里做项目评审时，“不单独用 JSP”或者“不单独用 JSP+Servlet”并不在检查项之列，不是因为不用检查，而是因为在这个公司里，根本不存在用这两种模式开发的项目。在面试别人的时候，面试者也基本没遇到过用这两种模式开发项目。由此可见，大家在学到这个程度后，得赶紧趁热打铁，了解后续一个基于 MVC 的模式：JSP+Servlet+JavaBean

的开发模式。

3.3 JS P+Servlet+JavaBean+JDBC 的开发模式

引入 Servlet，能比较好地解决跳转的问题，至少不会在同一个 JSP 里自行跳转。但 Servlet 作为控制器，本身不该包含“业务相关的逻辑”，即不该在 Servlet 里实现业务相关的代码。

这里我们将引入 JavaBean，把业务逻辑定义在 JavaBean 里。当再次分离业务逻辑后，就能看到 一个 MVC 的雏形，并能了解到基本的 Web 开发模式。

3.3.1 通过一个简单的例子来了解 JavaBean

对于 JavaBean 的常规定义如下：JavaBean 是使用一种符合某些命名方法和设计规范的 Java 类。一个 JavaBean 需要具备如下特征：

第一，JavaBean 类必须具备一个没有参数的构造函数，这个构造函数会被 JSP 的标签 <jsp:useBean>实例化该 JavaBean 对象时调用。如果实现 JavaBean 类时没有定义任何构造函数，系统将自动生成一个没有参数的构造函数。

第二，JavaBean 内的属性都应定义为私有的，这些属性应该通过 get 和 set 方法被外部来使用。

下面通过一个例子来认识一下 JavaBean。这部分的代码包含在 SimpleJavaBean 项目里。该例子纯粹是为了展示 JavaBean，并没有考虑业务分离等其他因素。

代码位置	视频位置
code/第 3 章/SimpleJavaBean	视频/第 3 章/JavaBean 的讲解

步骤一 创建一个 javabean，它其实是个 Java 程序，需要注意的是要把它放到一个 package 里，代码如下。

```
1 package JavaBeanPackage;
2 public class NameHandler{
3     private String userName;
4     public NameHandler(){
5         user      Name = null;
6     }
7     public void setUserName(String name){
8         userName = name;
9     }
```

```

10     public String getUsername(){
11         return userName;
12     }
13 }

```

这段代码符合 **JavaBean** 的定义。第一，在第 4 行里，我们定义了一个不带参数的构造函数；第二，在第 3 行里定义了一个 **userName** 变量，并在第 7 行和第 10 行定义了对应的 **Set** 和 **get** 方法，所以，它是个标准的 **Javabean**。

在开发过程中，请大家注意如下事项：

- ① 尽量把这个 **Javabean** 放到一个 **package** 里，否则后面在 **JSP** 里可能会找不到。
- ② 请注意大小写，比如属性名定义成 **userName**（**n** 大写，**u** 小写），而 **get** 和 **set** 方法里，**U** 大写，**N** 也大写。

步骤二 开发一个实现登录功能的 **hellouser.jsp**，代码如下。

```

1  <%@ page contentType="text/html; charset=gb2312" language="java" errorPage="" %>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3  <jsp:useBean id="mybean" scope="session" class="JavaBeanPackage.NameHandler"/>
4      <jsp:setProperty name="mybean" property="userName" value = "*" />
5  <%
6      if(request.getParameter("username")!=null)
7      {
8          response.sendRedirect("responseuser.jsp");
9      }
10 %>
11 <html xmlns="http://www.w3.org/1999/xhtml">
12 <head>
13 <meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
14 <title>Hello,User</title>
15 </head>
16 <body>
17     <h1>Hello,my name is 张三.What's your name?</h1>
18     <form method = "get" action = "responseuser.jsp">
19         <input type = "text" name = "username" size = "25"><br>
20         <input type = "submit" value = "提交">
21         <input type = "reset" value = "重置">
22     </form>
23 </body>
24 </html>

```

在第 18 行的 **form** 标签里，通过 **action** 定义了跳转的目标，是 **responseuser.jsp**。

此外，请大家请注意在代码开始部分有如下定义：

```
<jsp:useBean id="mybean" scope="session" class="JavaBeanPackage.NameHandler"/>
<jsp:setProperty name="mybean" property="userName" value = "*" />
```

这里是引入了一个 Javabean，在 class 位置，需要指定它的路径和文件名。在 setProperty 部分，是往 mybean 这个 id 的 Javabean（也就是 NameHandler）的 userName 属性（注意大小写）里设置一个值，这里的*是默认空值的意思。

步骤三 开发 responseuser.jsp，代码如下。

```
1  <%@ page contentType="text/html; charset=gb2312" language="java" errorPage="" %>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3  <jsp:useBean id="mybean" scope="session" class="JavaBeanPackage.NameHandler"/>
4  <jsp:setProperty name="mybean" property="userName" value
   ='<%=request.getParameter("username")%>' />
5  <html xmlns="http://www.w3.org/1999/xhtml">
6  <head>
7  <meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
8  <title>hello</title>
9  </head>
10     <h1>Hello,<jsp:getProperty name = "mybean" property = "userName" /></h1>
11 <body>
12 </body>
13 </html>
```

在第 10 行里，我们在如下的代码里，通过 getProperty 来获取并展示 userName 这个属性。

```
<h1>Hello,<jsp:getProperty name = "mybean" property = "userName" /></h1>
```

需要注意的是：

① NameHandler.class 需要放到 web-inf 的 JavaBeanPackage 目录下，这样的话，JSP 能自动找到这个 Bean。

② 请注意大小写一致，在 JSP 页面里“用户”这个标签是 username，n 是小写的，但在 Javabean 里，userName 的 N 是大写。这个不是绝对应该这样设置，而一旦在 Javabean 和 jsp 里设置好的对应的属性，引用的时候就该正确写好大小写。

当运行 hellouser.jsp 时，能看到如图 3.4 所示的效果。

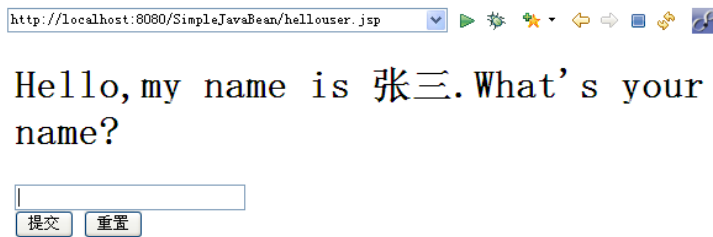


图 3.4 hellouser.jsp 的运行效果

输入用户名并单击“提交”按钮后，能进入到 responderuser.jsp 页面，效果如图 3.5 所示。

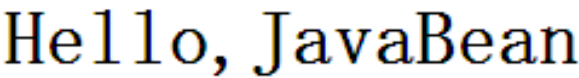


图 3.5 登录后的效果

3.3.2 在 JavaBean 里编写业务逻辑

在刚才的例子中，定义的 JavaBean 只承担了“传递变量”的作用。在一般的项目中，JavaBean 的作用不仅如此，广义来讲，在 JSP+Servlet+JavaBean 的项目中，一般是把 .java 的文件都叫作 JavaBean。

下面用 JSP+Servlet+JavaBean+JDBC 的模式，开发一个“展示所有留言”的项目，从中大家可以了解一下基本 MVC 项目的开发方式。

代码位置	视频位置
code/第 3 章/ForumDemo	视频/第 3 章/JavaBean 的案例分析

1. 准备工作，创建数据库

在 MySQL 中创建一个名为 ForumDemo 的 Schema，并在其中创建一个 notes 表，用于存放论坛留言。notes 表的结构如表 3.1 所示。

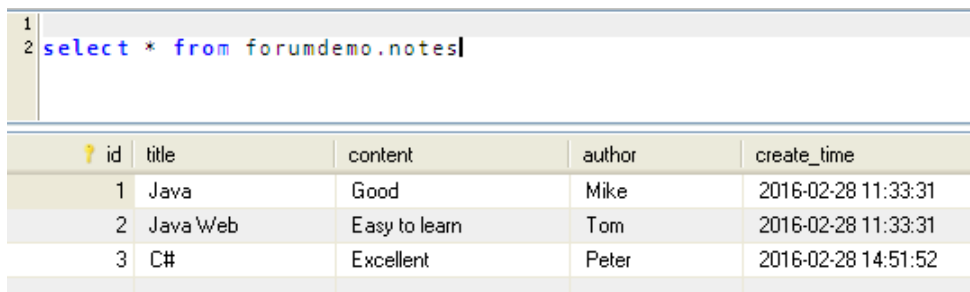
表 3.1 notes（留言表）的结构

字段名	属性	说明
ID In	integer	编号，自增长
Title varchar		标题
content var	char	内容
author var	char	作者
create_time Dateti	me	发布时间

其中，ID 是自增长的主键，create_time 是 Datetime 类型，表示本留言发布的时间。创建完成后，可通过类似如下语句，向表里插入若干记录。

```
insert into forumdemo.notes (title, content,author,create_time)
values('C#','Excellent','Peter',now())
```

完成插入后，可通过 `select * from forumdemo.notes` 来查看结果，我们插入了 3 条记录，如图 3.6 所示。



id	title	content	author	create_time
1	Java	Good	Mike	2016-02-28 11:33:31
2	Java Web	Easy to learn	Tom	2016-02-28 11:33:31
3	C#	Excellent	Peter	2016-02-28 14:51:52

图 3.6 notes 表里的记录

2. 准备工作，创建 Web 项目

这里创建一个名为 myBBS 的 Web 项目，由于要连接 MySQL 数据库，所以在项目里需要引入 `mysql-connector-java-5.1.19-bin.jar` 这个包。如果用的是 Tomcat 作为服务器，则一般把这个包放在 `WEB-INF/lib` 目录里，如图 3.7 所示。

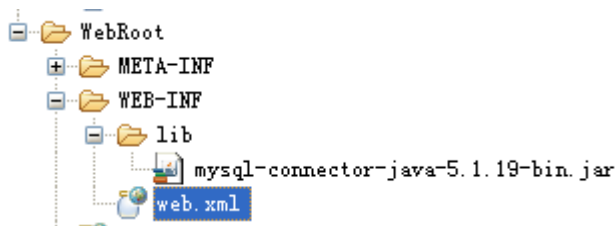


图 3.7 把 `mysql-connector-java-5.1.19-bin.jar` 放入 `WEB-INF/lib` 目录

3. 在项目里编写 `welcome.jsp`

我们简化这个页面功能，在其中只设计一个按钮，单击后能跳转到展示所有留言的 `notelist.jsp` 页面，代码如下。

```
1 <%@ page language="java" import="java.util.*" pageEncoding="GB2312"%>
2 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
3 <html>
4 <head>
```

```

5  <title>欢迎进入论坛</title>
6  </head>
7  <body>
8  <form name="form1" action="./servlet/Enter" method="POST">
9    <table>
10     <tr>
11       <td align="center">
12 <input type="submit" name="logon" value="进入论坛" />
13       </td>
14     </tr>
15   </table>
16 </form>
17 </body>
18 </html>

```

在第 8 行的 form 里定义了一个 submit 类型的按钮，一旦点击，会跳转到./servlet/Enter 这个 Servlet 上。

4. 在 web.xml 里定义 Servlet

关键代码如下。

```

1  <servlet>
2  <servlet-name>Enter</servlet-name>
3  <servlet-class>servlet.Enter</servlet-class>
4  </servlet>
5  <servlet-mapping>
6  <servlet-name>Enter</servlet-name>
7  <url-pattern>/servlet/Enter</url-pattern>
8  </servlet-mapping>

```

定义之后，一旦在 welcome.jsp 里发起“/servlet/Enter”的请求，Web 服务器就会根据第 7 行<servlet-mapping>里的<url-pattern>，交由名为 Enter 的这个 Servlet 来处理这个请求。

根据定义在<Servlet>里的<servlet-name>和<servlet-class>的对应关系，能看出这个请求最终将由 Servlet 这个 package 里的 Enter.java 来处理。

5. 定义具体的 Servlet 类

根据 web.xml 里的定义，来看一下 Enter.java 这个 Servlet 的具体实现。

```

1  省略必要的 package 和 import 方法
2  public class Enter extends HttpServlet {
3      private static final long serialVersionUID = 1L;
4      public Enter() {
5          super();

```

```

6      }
7      protected void doPost(HttpServletRequest request, HttpServletResponse
      response) throws ServletException, IOException {
8          resp      onse.sendRedirect("../notelist.jsp");
9      }
10 }

```

由于从 `welcome.jsp` 过来的请求是 `post` 类型的，所以这个请求将由 `doPost` 方法来实现。其中的逻辑非常简单，调用 `response` 的 `sendRedirect` 方法，跳转到 `notelist.jsp` 上。

6. 编写 `notelist.jsp` 页面

在这个页面里将展示所有的留言信息，代码如下。

```

1  <%@ page language="java" import="java.util.*" pageEncoding="GB2312"%>
2  <%@ page import="com.business.NoteBean, com.domain.Note, java.util.*" %>
3  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
4  <html>
5  <head>
6  <title>留言信息列表</title>
7  </head>
8  <body>
9  <h2>    所有留言信息如下 :</h2><br />
10 <tab    le border="1" width="700">
11 <tr>
12 <th          width="100">留言标题</th>
13 <th          width="200">内容</th>
14 <th          width="50">作者姓名</th>
15 <th          width="200">发布时间</th>
16 </tr>    >
17 <%
18     NoteBean noteBean = new NoteBean();
19     List      <Note> noteList = noteBean.findAllNotes();
20     Iter      ator<Note> noteIt = noteList.iterator();
21     while      e(noteIt.hasNext()) {
22         Note          tn = (Note)noteIt.next();
23         %>
24 <tr>
25 <td>          <%= tn.getTitle() %></td>
26 <td>          <%= tn.getContent() %></td>
27 <td>          <%= tn.getAuthor() %></td>
28 <td>          <%= tn.getCreateTime() %></td>
29 </tr>    >
30 <%
31 }

```

```

32      %>
33    </table>
34  </body>
35 </html>

```

在第 19 行里，通过调用 `JavaBean` 里的 `findAllNotes` 方法，返回所有的留言。注意，这里仅仅是调用，而没有亲自到 `MySQL` 里查找所有的留言并返回。

随后，通过第 21 行到第 31 行的 `while` 循环，依次输出所有的信息。

7. 编写 `NoteBean` 和 `Note` 类

在 `NoteBean` 的第 3 行里，定义了获取所有留言的 `findAllNotes` 方法，这个方法的返回类型是 `List<Note>`。

```

1  省略必要的 package 和 import 代码
2  public class NoteBean {
3      public List<Note> findAllNotes() {
4          try {
5              NoteDAO noteDAO = new NoteDAO();
6              return noteDAO.findAll();
7          }
8          catch (Exception e) {
9              //print the error msg
10             System.out.println("could not find all notes.");
11             e.printStackTrace();
12         }
13         return new ArrayList<Note>();
14     }
15 }

```

注意这里再次分离了“业务逻辑”（得到所有留言）和“数据访问逻辑”（怎么从数据库里得到留言），把“怎么从数据库得到留言”这个动作定义到 `DAO` 层里。这个分离给我们带来的好处是，如果更改了数据库的存储方式，比如要从 `MySQL` 迁移到 `Oracle`，那么只需修改 `DAO` 部分，而不用修改和业务有关的 `NoteBean` 部分。

这里定义的 `Note.java` 所包含的数据属性和数据表 `notes` 里的基本一致，代码如下：

```

1  省略必要的 package 和 import 代码
2  public class Note {
3      private int id;
4      private String title;
5      private String content;
6      private String author;
7      private Timestamp createTime;

```



```
8
9     public String getAuthor() {
10     return author;
11     }
12     public void setAuthor(String author) {
13     this.author = author;
14     }
15     //省略对其他属性的 get 和 set 方法
16     ...
17 }
```

我们看到，第 3 行到第 7 行的 id、title 等属性和 notes 表里的列属性能一一对应上。这种做法叫作“映射”，用代码层面的 List<Note>来映射数据库里 note 表的记录。这里的映射做得很粗糙，更好的做法可以参考后文讲到的 ORM 技术。

在这里仅仅是从数据库里拿出所有的 note 记录并存放在 List<Note>里，在实际的项目里，一般是在业务层对保存 note 记录的 List<Note>进行操作，（而不是直接对数据库的 note 表进行操作），在 List<Note>里增加、删除或者修改记录后，再把这部分的数据写回到数据库。

在平时的项目里，会大量出现这种“业务和数据库分离”的做法，因为这非常符合逻辑。比如要订火车票，我们只管向业务实现层发起请求，比如向车站售票员发起请求，而如何在数据库里生成这张车票，是单独层面的逻辑，售票员不用关心，我们更不用关心。

8. 编写 NoteDAO 类

虽然分离了业务和数据库实现的逻辑，但最终的实现还是要落实到数据库层。针对数据库的实现放在 NoteDAO 里，代码如下。

```
1  省略必要的 package 和 import 方法
2  public class NoteDAO {
3      public Connection getConnection() throws Exception {
4      Class.forName("com.mysql.jdbc.Driver");
5      Connection connection =
6      DriverManager.getConnection("jdbc:mysql://localhost:3306/forumDemo", "root",
7      "123456");
8      return connection;
9      }
10     public List<Note> findAll() throws Exception {
11     Connection con = getConnection();
12     String sql = "select id,title,content,author,create_time from notes";
13     Statement sta = con.createStatement();
```

```

13     Resu         ltSet rs = sta.executeQuery(sql);
14         List<Note> notes = new ArrayList<Note>();
15     while         e(rs.next()) {
16         Note             tn = new Note();
17         tn.s              etId(rs.getInt("id"));
18         tn.s              etTitle(rs.getString("title"));
19         tn.s              etContent(rs.getString("content"));
20         tn.s              etAuthor(rs.getString("author"));
21         tn.s              etCreateTime(rs.getTimestamp("create_time"));
22         note              s.add(tn);
23     }
24     con.            close();
25     retu            rn notes;
26 }
27 }

```

代码中的第3行，定义了获取连接的 `getConnection` 方法，第9行定义了返回所有 `note` 信息的 `findAll` 方法。

在 `getConnection` 方法里，可根据一系列的数据库连接信息，比如连接字符串、用户名和密码，返回一个 `Connection` 对象，而在 `findAll` 方法里，从数据库里通过 `select` 语句，返回所有的 `note`，并通过 `List<Note>` 对象返回。

9. 运行结果

当发布好这个项目并启动 Tomcat 服务器后，在浏览器里输入 `http://localhost:8080/myBBS/welcome.jsp`，就能看到如图 3.8 所示的页面。

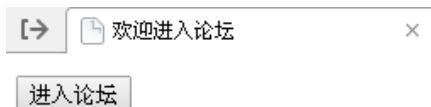
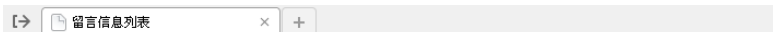


图 3.8 欢迎页面

单击“进入论坛”按钮后，能看到所有的留言信息，如图 3.9 所示。



所有留言信息如下：

留言标题	内容	作者姓名	发布时间
Java	Good	Mike	2016-02-28 11:33:31.0
Java Web	Easy to learn	Tom	2016-02-28 11:33:31.0
C#	Excellent	Peter	2016-02-28 14:51:52.0

图 3.9 展示所有留言信息的页面

3.3.3 对 MVC 的总结

在前面的留言系统里，已经用到了 MVC 的架构。其中 JSP 承担了 V 即显示层的功能，Servlet 承担了 C 即控制跳转的功能，而在 JavaBean 里定义的实现业务逻辑的代码则承担了 M 即模型层的功能。

通过前面留言系统来归纳一下 MVC 架构的工作流程。

① 使用者从显示层（welcome.jsp）发起请求，这个请求将会被控制器层（Servlet）转发到合适的处理程序上。

② 当 Model 层接收到请求后，一般会调用 DAO 层，得到结果并返回给显示层。比如在这个的项目里，是 NoteBean 调用 NoteDAO 里的代码，从数据库得到结果后，封装在一个 List<Note>对象里返回给 notelist.jsp，并由显示层展示结果。

一般我们在面试过程中，不会要求面试者画出流程（因为抽象的流程大家都会），而是会问：“你对 MVC 模式有什么认识”，或者“你在项目里一般怎么使用 MVC”。在回答里，我们更关心面试者是否有“业务分离”的意识，比如数据库逻辑和业务逻辑分离，MVC 各层是怎么定义的，它们怎么交互。

这个问题一般大家都能很好地回答，所以一般用来“过滤差者”，如果有人对 MVC 根本不了解，或者对 MVC 有哪些模块都不知道，那么这些人可能真就没做过 Web 方面的开发。

3.4 总结

MVC 模式不是某些技术大牛拍脑袋想出来然后向大家推广的，而是通过无数次的吃亏总结出来的，所以本章的叙述方式不是先讲理论，而是通过例子来向大家展示 MVC 的好处。

这里给出的基于 MVC 的 JSP+Servlet+JavaBean+JDBC 的开发模式是非常基本的，从我们的面试经验来看，一般计算机专业的学生一毕业就能掌握到这种程度。

但是，我们也发现了一个现象，那就是工作经验在 1 年左右的人，不少人也开发过 MVC 方面的代码，但他们对 MVC 的认识只停留在理论层面，能向考官泛泛而谈 MVC 的结构以及优势，但当考官让他们结合具体项目实例来论述 MVC 的好处时，这些人无法具体说出“分层”以及“分层”带来的好处。

在实际工作中，经验低于 3 年的程序员一般是无法独立设计架构的，一般是在项目经理的带领下从事 Web 架构的开发工作。

从这个意义上来讲，工作经验低于 3 年的程序员，只要是在交流沟通的过程中，能有效地结合项目表述出“分离带来的好处”，并且在开发过程中主动地划分 M、V、C、业务层和数据访问层，那么他给别人的感觉就是“超出平均水平”。

读完本章之后，希望大家能对 Web 架构有个基本的认识，这是后继学习 SSH 和 SpringMVC 架构的基础。

同样，在这里我们总结些 JSP 和 Servlet 相关的面试题。

问题 1，Java Servlet 中 forward 和 redirect 有何区别？

forward 是服务器请求资源，服务器直接访问目标地址的 URL，把那个 URL 的响应内容读取过来，然后把这些内容再发给浏览器，浏览器根本不知道服务器发送的内容是从哪儿来的，所以它的地址栏中还是原来的地址。

而 redirect 是服务端根据逻辑，发送一个状态码，告诉浏览器重新去请求那个地址，一般来说浏览器会用刚才请求的所有参数重新请求，所以 session,request 参数都可以获取。

问题 2，Servlet 里 doPost 和 doGet 的区别？和这个问题相关的另外一个常见问题是，Get 和 Post 的两种请求方式有什么差别？

www.abc.com?username=xxx，这是通过 get 来传递参数的方式，这种方式的好处是简单，但一般不会用来传输敏感数据。

在 Post 的方式里，我们一般是会把整个 form 表单传送到服务器端，post 适合发送大量的数据。

问题 3，有哪些方法可以实现页面的跳转？或者可以这样问，forward 和 sendRedirect 这两个方法有什么差别？

我们可以通过 jsp:forward 操作把当前的 JSP 页面转发到另一个页面上，具体的语法是 <jsp:forward page="test2.jsp"/>，使用该方法时，浏览器的地址栏中地址不会发生任何变化。

我们也可以使用 response.sendRedirect("URL 地址");的方法来重定向请求，在执行完这个方法跳转到新页面时，地址栏里显示的是新的 url。

问题 4，说一下 Servlet 的生命周期。

Servlet 的生命周期主要由 init，service 和 destroy 这 3 个过程组成。

如果我们是通过 HttpServlet 类来编写 Servlet 时，则是需要实现其中的 doGet 或 doPost 方法，当发出客户端请求时，Servlet 首先判断该请求是 GET 还是 POST，然后会调用适当的方法。

第 4 章

通过 Struts 进一步了解 MVC

和前文里介绍的 JSP+Servlet+JavaBean 框架相比，Struts 是一个实现 MVC 各部分之间跳转的模板，程序员只要通过编写一些代码和配置文件，就能很方便地实现“从前端数据请求→请求跳转→处理请求”等跳转动作。

如果没有 JSP+Servlet+JavaBean 这种基于 MVC 的实现，程序员就不得不在“剪不断理还乱”的 JSP 里大量嵌入 Java 代码。但如果单纯用 JSP+Servlet+JavaBean，而不用 Struts，那么程序员可能需要把一部分精力用到“不能实际产生价值”的 MVC 之间的交互上。

Struts 框架能根据程序员的配置文件，自动地实现 MVC 之间的跳转，从而让程序员可以更多地关注项目开发的灵魂——业务实现。

4.1 在 MyEclipse 里开发一个基本的 Struts 程序

下面通过一个 Struts 的程序，了解一下 Struts 的基本工作流程，从中，请大家体会一下开发 Struts 程序的基本步骤。

代码位置	视频位置
code/第 4 章/StrutsDemo	视频/第 4 章/开发一个简单的 Struts 程序

4.1.1 创建 Web 项目，并导入必要的 jar 包

步骤一 创建一个 Web project，命名为 strutsDemo。

步骤二 导入如下的包。

这里的包在 code/第 4 章/strutsDemo/WebRoot/WEB-INF/lib 下能找到，这些是 struts2.3.4 的支持包。

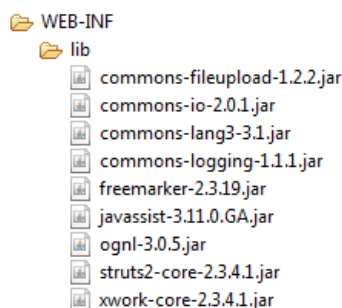


图 4.1 S truts 的支持包

4.1.2 开发前端的 JSP 代码

随后，在项目的 WebRoot 目录下，创建前端的 JSP 代码 calSum.jsp。

```

1  <%@ page language="java" pageEncoding="GBK" %>
2  <%@ taglib prefix="s" uri="/struts-tags"%>
3  <html>
4  <head>
5  <title>输入操作数</title>
6  </head>
7  <body>
8  求和<br/>
9  <s:form action="mystruts/calSum" >
10 <s:textfield name="num1" label="数 1"/>
11 <s:textfield name="num2" label="数 2" />
12 <s:submit value="求和" />
13 </s:form>
14 </body>
15 </html>

```

这个页面的效果如图 4.2 所示。



图 4.2 sum.jsp 的效果图

在第 2 行里引入了 Struts 的标签，前缀是 s，所以可以用 s:form 和 s:textfield 来定义 form 和文本框。

在第 9 行到第 13 行的 form 里，定义了两个输入框和一个提交按钮。当用户输入两个数字后，单击“求和”按钮后，本页面将根据定义在第 9 行的定义，跳转到 mystruts/calSum.action。

4.1.3 在 web.xml 里声明使用 Struts

如果要使用基于 Struts 的 MVC，则必须要在 web.xml 里声明，否则系统服务器是不会知道在项目里用到了基于 Struts 的 MVC 处理器。web.xml 的代码如下。

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:s="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
   version="2.5">
3    <filter>
4      <filter-name>struts2</filter-name>
5      <filter-class>
6        org.apache.struts2.dispatcher.FilterDispatcher
7      </filter-class>
8    </filter>
9    <filter-mapping>
10     <filter-name>struts2</filter-name>
11     <url-pattern>/*</url-pattern>
12   </filter-mapping>
13 </web-app>
```

从第 2 行到第 13 行之间的 web-app 元素里，我们声明了两件事：

(1) 通过第 11 行的 url-pattern，说明/*，也就是任何请求，都将由名为 struts2 的过滤器来处理。

(2) 在第3行到第8行之间, 指定了 `struts2` 这个过滤器它的后台处理类。

综合上述两点可知, 本项目的任何请求, 都将由 `Struts` 的后台处理类来处理。

4.1.4 配置 struts.xml 文件

在 `calSum.jsp` 里指定了 `form` 跳转的目的地。

```
<s:form action="mystruts/calSum" >
```

那么这个目的地究竟是哪? 一起来看一下 `struts.xml` 这个配置文件。

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">
3  <struts>
4  <package name="struts2" namespace="/mystruts"
5      extend="struts-default">
6  <action name="calSum" class="action.myAction">
7  <result name="Positive">/positive.jsp</result>
8  <result name="Negative">/negative.jsp</result>
9  </action>
10 </package>
11 </struts>
```

在第4和第5行里, 能看到名为 `calSum` 的 `action` 处于 `/mystruts` 这个命名空间里, 它所对应的是 `action.myAction` 这个处理类。也就是说, `calSum.jsp` 的请求最终是由 `action.myAction` 接收和处理。

4.1.5 开发 Action 类

随后开发 `myAction.java`, 它是放在 `action` 这个 `package` 下的。

```
1  package action;
2  import com.opensymphony.xwork2.ActionSupport;
3  public class myAction extends ActionSupport
4  {
5      private int num1;
6      private int num2;
7
8      public String execute() throws Exception
9      {
10         // 如果和大于等于0, 则跳到 positive.jsp 页面
```



```
11     if (getSum() >= 0)
12     {
13         return "Positive";
14     }
15     // 否则跳到 negative.jsp 页面
16     else
17     {
18         return "Negative";
19     }
20 }
21 public int getNum1() {
22     return num1;
23 }
24 public void setNum1(int num1) {
25     this.num1 = num1;
26 }
27 public int getNum2() {
28     return num2;
29 }
30 public void setNum2(int num2) {
31     this.num2 = num2;
32 }
33 public int getSum()
34 {
35     return num1 + num2; // 计算两个整数的代码数和
36 }
37 }
```

作为一个 Struts 的 Action，本类继承了 ActionSupport 类。当一个请求最终到达本 Action 时，如果没有额外的配置，就会如同本类一样，由 execute 方法来处理请求。

在第 8 行的 execute 方法里可能看到，如果两个数的和大于 0，则返回 Positive 字符串，反之则返回 Negative。这两个返回是和 struts.xml 里的配置相对应的。

在 struts.xml 的第 6 行和第 7 行里，看到有如下的配置。

```
6 <result name="Positive">/positive.jsp</result>
7<result name="Negative">/negative.jsp</result>
```

这说明根据不同的字符串，Struts 处理容器将会跳转到两个不同的 jsp 里。

在这个 Action 代码里，并没有给 num1 和 num2 赋值，这是因为它们和 calSum.jsp 里 form 中的两个输入框同名，所以会自动拿到我们输入的值。

4.1.6 开发两个跳转结果页面

上文提到过，Action 的 execute 方法里返回的不同字符串后，Struts MVC 处理器会读取 struts.xml 里的配置，并相应地，跳转到不同的页面。下面就来编写这两个页面的代码。先来看一下 positive.jsp。

```

1  <%@ page language="java" pageEncoding="GBK"%>
2  <%@ taglib prefix="s" uri="/struts-tags" %>
3  <html>
4  <head>
5  <title>显示和</title>
6  </head>
7  <body>
8  结果大于或等于 0 , <h1><s:property value="sum" /></h1>
9  </body>
10 </html>

```

这里的关键代码是在第 8 行，通过一个 Struts 的标签，来获取在 Action 里名为“sum”的属性对象。在 myAction.java 里，定义了一个 getSum 的方法，所以就会自动生成一个 sum 的属性。结果是两个操作数之和。

Negative.jsp 代码和 positive.jsp 很相似。

```

1  <%@ page language="java" pageEncoding="GBK"%>
2  <%@ taglib prefix="s" uri="/struts-tags" %>
3  <html>
4  <head>
5  <title>显示和</title>
6  </head>
7  <body>
8  结果小于 0 , <h1><s:property value="sum" /></h1>
9  </body>
10 </html>

```

唯一的差别在第 8 行，这里的叙述文字是“结果小于 0”。

4.2 通过运行，了解 Struts 的工作流程

Struts 的框架是基于 MVC 的，它提供了一套标准化的跳转流程，这里通过运行代码，来分析 Struts 的大致工作流程。

4.2.1 Struts 的跳转流程分析

现在来总结一下整个程序的运行和跳转流程。

第一，打开 Tomcat 服务器，并在浏览器里输入 `http://localhost:8080/strutsDemo/calSum.jsp` 后，能看到如图 4.3 所示页面。

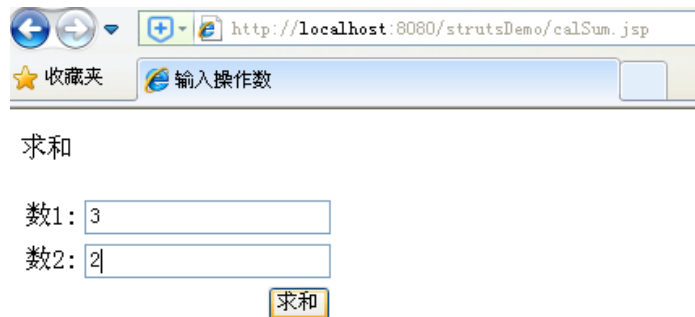


图 4.3 运行 `calSum.jsp` 的效果图

当输入两个数字，并单击“求和”按钮后，根据如下 form 里的定义，会跳转到 `mystruts/calSum` 里。

```
1 <s:form action="mystruts/calSum" >
2 <s:textfield name="num1" label="数 1"/>
3 <s:textfield name="num2" label="数 2" />
4 <s:submit value="求和" />
5 </s:form>
```

第二，根据 `web.xml` 的定义，可知这个跳转请求将由 `struts` 来处理。再根据 `struts.xml` 的定义，得知这个请求最终将由 `myAction.java` 来处理。

```
1 <package name="struts2" namespace="/mystruts"
   extend="struts-default">
2 <action name="calSum" class="action.myAction">
3 <result name="Positive">/positive.jsp</result>
4 <result name="Negative">/negative.jsp</result>
5 </action>
6 </package>
```

第三，在 `myAction.java` 的 `execute` 方法里，根据最终 `sum` 的值，分别返回两个不同的字符串，从上文 `struts.xml` 片段的第 3 行和第 4 行得知，根据两个不同的返回值，会跳转到两个不同的页面里。

4.2.2 Struts MVC 框架和 JSP+Servlet+JavaBean 框架的比较

在一个项目里，我们应更关注“业务该怎么处理”这个问题，而不应把大多数精力放在调试 JSP 到 Servlet 之类的跳转上。

Struts 给我们提供了一套跳转机制，我们可以简单地通过编写 struts.xml 和 web.xml，就能比较省心地实现从前端 JSP 跳转到具体业务处理类的功能。而且，也只需在 Action 类里编写返回字符串，同时在 struts.xml 里编写返回字符串和跳转页面的对应关系，就能根据业务执行结果方便地跳转回前端页面。

Struts 框架是基于 MVC 的，它的 View 部分主要由 JSP 页面实现。Controler 部分主要由 Action 类来承担，而 Model 部分主要由 ActionForm 组成。需要说明的是，在 Struts2.0 以上的版本里，开发者不需要额外定义 ActionForm，由用户在前端 Form 里传入的数据将被自动封装成 ActionFrom 对象，并被最终转发给 Action。

通过表 4.1 来对比一下 Struts 和前文提到的 JSP+Servlet+JavaBean 框架，综合各项对比的指标，Struts 略优于 JSP+Servlet+JavaBean 框架。

表 4.1 两种框架的比较

比较项 S	truts	JSP+Servlet+JavaBean	结论
如何在后端接收前端传来的参数	参数组装成 ActionForm，并自动发送到 Action 里	需要在 Servlet 里编写接收参数的代码	Struts 比较省心
如何把前端的请求发送到合适的处理页面	可以在 struts.xml 里统一定义请求和处理类的对应关系	可以在 web.xml 里统一定义请求和处理类的对应关系	基本持平
后端处理请求的方式	定义在 Action 里	大多定义在 Servlet类的 doPost 或 doGet 方法里	基本持平
如何把后端的处理结果再回传到前端	可以在 struts.xml 里统一地定义处理结果和返回页面的对应关系	需要在 Servlet 里手动地跳转 St	ruts 略优
项目的开发方式	程序员的工作量比较少，在必要的地方（比如 Action 类和 Struts.xml）里填写必要的代码即可，Struts 处理器能方便地实现 MVC 之间的跳转	程序员可能得操心必要的细节，比如 Servlet 里如何接收参数，如何跳转到前端，等等	Struts 的开发流程比较省心

为了让别人知道你确实用 Struts 的 MVC 做过项目，仅仅了解 Struts 运行流程是不够的，因为在项目里，还会经常用到一些诸如验证器和过滤器等技术，下面就依次来讲解常用的技术。

4.3 通过 Struts 的验证机制校验输入内容

在实际项目里，经常需要验证输入的内容。比如刚才的例子，要求输入数字，假设项目的要求是，如果输入非数字，就需要提示错误。

在前端，也可以通过 JS 实现验证，而且这类验证的好处是请求无须提交到后台 Struts 类，从而能减少后端 Web 服务的压力。

我们在面试一些初级程序员时，有时会听到他们仅仅用验证器验证诸如是否为空或者是否是数字这类的简单需求，但当我们进一步提问为什么不用 JS 而要用 Struts 时，他们往往就不知道了。有人甚至不知道 JS 验证和 Struts 验证的区别。

给大家的建议是，Struts 验证器往往是处理些需要前后端结合验证的，比如验证某用户是否存在于数据库里。并且，你在准备面试时，最好结合项目准备一个使用场景来证明你用 Struts 验证器。从这个意思上来讲，本部分给出的其实都是“学习用例”，而不是“商业应用”。

4.3.1 通过 Validate 方法来验证

我们可以在 Action 里添加一个 Validate 方法来实现验证。和 4.1 节的项目相比，它做了些细微的改变。

代码位置	视频位置
code/第 4 章/StrutsValidate	视频/第 4 章/Struts 验证器的讲解

改变 1，改写 myAction.java 类，具体代码如下。

```
1 package action;
2 import com.opensymphony.xwork2.ActionSupport;
3 public class myAction extends ActionSupport
4 {
5     //num1 和 num2 修改成 String 类型，原来是 int 类型
6     private String num1;
7     private String num2;
8     //用了 n1 和 n2 来接收转换后的 num1 和 num2 这两个输入参数
9     private int n1;
10    private int n2;
11    //execute 方法没修改
12    public String execute() throws Exception
13    {
14        //    如果和大于或等于 0，则跳到 positive.jsp 页面
15        if (getSum() >= 0)
16        {
```

```

17         return "Positive";
18     }
19     // 否则跳到 negative.jsp 页面
20     else
21     {
22         return "Negative";
23     }
24     }
25
26     public String getNum1() {
27         return num1;
28     }
29     public void setNum1(String num1) {
30         this.num1 = num1;
31     }
32     public String getNum2() {
33         return num2;
34     }
35     public void setNum2(String num2) {
36         this.num2 = num2;
37     }
38     // 请注意这里用 n1 和 n2 计算和, 而不是 num1 和 num2
39     public int getSum()
40     {
41         return num1 + num2; // 计算两个整数的代码数和
42     }
43     // 在这个方法里执行验证
44     public void validate()
45     {
46         // 如果无法转换成 int, 则直接抛出异常
47         try{
48             num1 = Integer.valueOf(num1);
49         }
50         catch (Exception e)
51         {
52             // 第二个参数表示出错以后该提示什么, 第一参数表示往哪里输出提示
53             addFieldError("num1", "num1 is not Number.");
54         }
55         try{
56             num2 = Integer.valueOf(num2);
57         }
58         catch (Exception e)
59         {
60             addFieldError("num2", "num2 is not Number.");

```

```
61     }  
62 }  
63 }
```

在代码的第 6 行和第 7 行里，把 `num1` 和 `num2` 修改成了 `String` 类型，这样就能接收从前端传输过来的任何类型的参数。

而在执行验证的第 44 行的 `Validate` 方法里，把 `num1` 和 `num2` 转换成 `int` 类型，如果出错，则会通过第 53 和第 60 行的 `addFieldError` 来抛出错误提示。

请大家注意 `addFieldError` 方法的第一个参数，它表示往哪里输出提示，这个需要和 JSP 页面里的元素保持一致。

改变 2，修改 `struts.xml`。

```
1 <struts>  
2 <package name="struts2" namespace="/mystruts"  
3     extends="struts-default">  
4 <action name="calSum" class="action.myAction">  
5 <result name="Positive">/positive.jsp</result>  
6 <result name="Negative">/negative.jsp</result>  
7 <result name="input">/calSum.jsp</result>  
8 </action>  
9 </package>  
10 </struts>
```

与 `strutsDemo` 项目相比，多了一行 `result name="input"`。如果在 `myAction` 这个类的 `Validate` 方法确实遇到异常，比如输入的 `num1` 不是数字，那么 `Validate` 在检验出问题后，会往 “input” 上跳转，根据配置文件里的定义，会跳转到 `calSum.jsp` 上。

来看一下结果，发现一旦输入了非数字的格式，就会在页面上看到相应的提示信息。

求和

num1 is not Number.

数1:

num2 is not Number.

数2:

图 4.4 验证出错时的效果图

4.3.2 在配置文件里定义验证方式

我们还可以通过在 Struts 提供的配置文件里定义验证的方式，这部分的代码在 code/第4章/strutsValidatewithXML 目录下，下面我们来详细说明具体的编写方式。

代码位置	视频位置
code/第4章/strutsValidatewithXML	视频/第4章/通过配置文件实现 Struts 验证器

步骤一 创建 Web 项目，并导入 Struts 的包。

步骤二 编写前端的 register.jsp 代码，这里我们是模拟用户注册的需求。

```

1  <%@ page language="java" pageEncoding="GBK" %>
2  <%@ taglib uri="/struts-tags" prefix="s"%>
3  <html>
4  <body>
5  <s:fielderror/>
6  <form action="registerAction" validate="true" method="post">
7  登录用户:<input type="text" name="username"><br>
8  登录密码:<input type="text" name="password"><br>
9  确认密码:<input type="text" name="confirm"><br>
10  邮件地址:<input type="text" name="email"><br>
11  手机号码:<input type="text" name="handphone"><br>
12  <input type="submit" value="提交">
13 </form>
14 </body>
15 </html>

```

第7到第11行的代码提供了能让用户输入信息的若干文本框，从第6行的 form 的定义里可看到，当单击“提交”按钮后，这个 form 将会提交到 registerAction 这个 Struts 的 Action 里。

请大家尤其注意，在第5行我们加上了<s:fielderror/>标签，否则一旦出现验证错误的信息，将无法回写到这个页面。

步骤三 编写 web.xml 和 struts.xml。

在 web.xml 里，同样定义将使用 Struts，代码和 StrutsDemo 项目里一致，所以不再给出。在 struts.xml 里，定义了 Action 类以及一些跳转信息。

```

1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE struts PUBLIC
3  "http://struts.apache.org/dtds/struts-2.0.dtd">

```



```
4  <struts>
5  <package name="struts2" namespace="/"
6      extends="struts-default">
7  <action name="registerAction" class="action.registerAction">
8  <result name="success">
9  /registerSuccess.jsp
10 </result>
11 <result name="input">/register.jsp</result>
12 </action>
13 </package>
14 </struts>
```

第 8 行和第 11 行为 `registerAction` 定义了两个返回。第 8 行中，定义了一旦验证通过，将跳转到 `registerSuccess.jsp` 页面；在第 11 行中，通过 `input` 这个返回值，定义了一旦验证失败，将跳转回 `register.jsp`。

步骤四 编写 `registerAction.java` 这个 Action 类。

```
1  package action;
2  import com.opensymphony.xwork2.ActionSupport;
3  public class registerAction extends ActionSupport
4  {
5      private String username;
6      private String password;
7      private String confirm;
8      private String email;
9      private String handphone;
10
11     public String execute() throws Exception
12     {
13         return "success";
14     }
15     public String getUsername() {
16         return username;
17     }
18     public void setUsername(String username) {
19         this.username = username;
20     }
21     //略去针对其他属性的 get 和 set 方法
22 }
```

从第 3 行可看到，`registerAction` 同样继承了 `ActionSupport`，所以它承担了 `Action` 的角色。从第 5 行到第 9 行，定义了接收注册页面的一些属性，请注意，同样需要和在页面里的属性名保持一致。在第 11 行的 `execute` 方法中，是直接跳转。结合 `struts.xml` 可知如果输入的属性

值通过验证，则将直接跳转到 registerSuccess.jsp 页面。

步骤五 也是重头戏，编写定义验证方式的 validation.xml 文件。

需要说明的是，这个文件名的格式是 Action 名+"-validation.xml"，所以在这个项目中，文件名是 registerAction-validation.xml。另外，这个文件需要和 Action 处在同一个 package 里。

```

1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE validators PUBLIC
3    "-//OpenSymphony Group//XWork Validator 1.0.2//EN"
4    "http://struts.apache.org/dtds/xwork-validator-1.0.3.dtd">
5  <validators>
6    <field name="username">
7      <field-validator type="requiredstring">
8        <param name="trim">true</param>
9        <message>必须输入用户名。</message>
10     </field-validator>
11   </field>
12   <field name="password">
13     <field-validator type="requiredstring">
14       <param name="trim">true</param>
15       <message>必须输入密码。</message>
16     </field-validator>
17   </field>
18   <field name="confirm">
19     <field-validator type="requiredstring">
20       <param name="trim">true</param>
21       <message>必须输入确认密码。</message>
22     </field-validator>
23   </field>
24   <field name="confirm">
25     <field-validator type="fieldexpression">
26       <param name="expression"> (confirm eq password)</param>
27       <message>密码必须和确认密码一致</message>
28     </field-validator>
29   </field>
30   <field name="email">
31     <field-validator type="email">
32       <message>电子邮件地址必须有效。</message>
33     </field-validator>
34   </field>
35   <field name="handphone">
36     <field-validator type="requiredstring">
37       <message>手机号不能是空。</message>
38     </field-validator>

```

```

39     <field-validator type="regex">
40         <param name="regexExpression"><![CDATA[^\1[358]\d{9}$]]></param>
41         <message>请填写正确的手机号.</message>
42     </field-validator>
43 </field>
44 </validators>

```

在第 4 行需要引入一个 `xwork-validator-1.0.3.dtd` 文件，否则验证流程将无法继续。

从第 6 行到第 11 行，针对“用户名”这个输入参数，定义了一个“非空”验证，一旦为空，将返回定义在第 9 行里的 `<message>` 部分的信息。

从第 24 行到第 29 行，通过了 `expression` 这个表达式，定义了“密码必须和确认密码一致”这个验证规则。

从第 30 行到第 34 行，通过 `<field-validator type="email">`，定义了针对 Email 格式的验证规则。如果这里用户输入了非 Email 的字符，比如不包含 `@`，将提示出错。

在第 39 行和第 40 行里，通过正则表达式，定义了针对手机的验证规则。

步骤六 编写通过验证后的跳转目标 `registerSuccess.jsp` 页面，这个页面比较简单。

```

1  <%@ page language="java" pageEncoding="GBK"%>
2  <%@ taglib prefix="s" uri="/struts-tags" %>
3  <html>
4  <body>
5  <s:property value="username" />
6  </body>
7  </html>

```

这里请大家注意第 5 行，一旦通过验证，`username` 这个属性将会传递到这个页面。

只有输入正确的格式后，才能通过验证。

登录用户:	<input type="text" value="username"/>	
登录密码:	<input type="text" value="123456"/>	真实的项目里，密码需要用*来替代，这里仅仅是演示效果
确认密码:	<input type="text" value="123456"/>	
邮件地址:	<input type="text" value="hsm_computer@163.com"/>	只有当输入正确的邮件地址和手机的格式后，才能通过验证
手机号码:	<input type="text" value="13"/>	
<input type="button" value="提交"/>		

图 4.5 输入信息正确的效果图

一旦出错，将出现提示信息如图 4.6 所示。

- 密码必须和确认密码一致
- 手机号不能是空。

登录用户:	<input type="text" value="us"/>
登录密码:	<input type="text" value="123456"/>
确认密码:	<input type="text" value="1234"/>
邮件地址:	<input type="text" value="hsm_computer@163.com"/>
手机号码:	<input type="text"/>
<input type="button" value="提交"/>	

图 4.6 输入出错后看到的效果图

4.4 S truts 拦截器

从前端 JSP 发出的请求，在被 Action 处理之前，可以先被拦截器处理。通过拦截器，能保证只有合法的请求才能进到 Action 里。

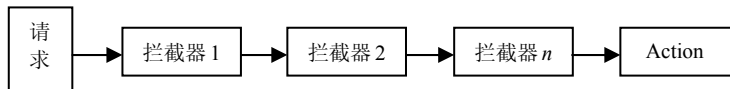
在项目开发中，可以用拦截器来拦截一些非法请求，也可以拦截一些非法参数。确实，这些拦截功能可以在 Action 里实现，但我们还是会把它们封装到拦截器部分，原因是 Action 里应该更多地关注业务，就像公司的业务人员不应当过多关注非业务部分一样。

一旦在项目里用到了 Struts，那么拦截器出现的概率是相当大的，所以，我们经常用拦截器这方面的知识点来确认面试者是否真的用过 Struts。

4.4.1 拦截器与职责链设计模式

在一个项目里，可以定义不同的拦截器，比如第一个拦截器过滤掉记录在黑名单里的 IP 地址，第二个可以用来过滤不当参数。

如果定义了多个拦截器，这就和“职责链”设计模式的思想相匹配。在一般的应用里，每个拦截器只承担拦截特定请求的功能，这和面向对象思想里的“一个方法最好只实现一种功能”的原则相一致。



4.4.2 通过登录案例详解拦截器的用法

通过这个接近真实项目的例子，大家可以很快了解拦截器的用法。

代码位置	视频位置
code/第 4 章/StrutsInterceptLogin	视频/第 4 章/Struts 拦截器的讲解

步骤一 我们建立一个名为 StrutsInterceptLogin 的 Web 项目，同样在 web.xml 里定义使用 Struts。

步骤二 开发 login.jsp，代码如下。

```
1  <%@ page language="java" pageEncoding="UTF-8"%>
2  <%@ taglib prefix="s" uri="/struts-tags"%>
3
4  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
5  <html>
6  <head>
7  <title>登录界面</title>
8  </head>
9  <body>
10  用户登录
11  <br>
12  <s:form action="/login" method="post">
13  <s:textfield name="username" label="用户名"/>
14  <s:textfield name="password" label="密码"/>
15  <s:submit value="登录"/>
16  </s:form>
17  </body>
18  </html>
```

这段代码大家应该很熟悉了，当填写好第 13 行和第 14 行里的用户名和密码后，单击第 15 行的“登录”按钮后，会跳转到第 12 行定义的 action 里。

步骤三 拦截器代码 UserLoginInterceptor.java。

前文里提到过，任何提交到 Action 的请求都可以被拦截器处理。在这个拦截器里，可判断请求的来源，如果不是来自登录页面，则说明这个请求来源不正，随后会跳转到登录页面，代码如下。

```
1  package action;
2  import java.util.Map;
3  import com.opensymphony.xwork2.Action;
4  import com.opensymphony.xwork2.ActionInvocation;
```

```

5  import com.opensymphony.xwork2.interceptor.AbstractInterceptor;
6  import action.LoginAction;
7
8  public class UserLoginInterceptor extends AbstractInterceptor {
9  public String intercept(ActionInvocation arg0) throws Exception {
10 //    判断请求是否来源于登录界面(login),如果是则不拦截
11     if(LoginAction.class == arg0.getAction().getClass())
12     {
13         return arg0.invoke();
14     }
15
16     Map sessionMap = arg0.getInvocationContext().getSession();
17     if(null == sessionMap.get("username"))
18     {
19 System.out.println("in LoginInterceptor, the Username is null.");
20     return Action.LOGIN;
21     }
22     return arg0.invoke();
23 }
24 }

```

为了开发一个拦截器我们需要：

- ① 如第 8 行所示，继承 `AbstractInterceptor` 类。
- ② 如第 9 行所示，在 `intercept` 方法里，编写这个拦截器里需要实现的功能。

在本拦截器里，通过第 11 行的 `if` 来判断，如果是发往 `LoginAction` 的，则调用 `arg0.invoke` 方法，不做任何事情，把请求继续发向下一个处理环节。而在第 17 行的 `if` 判断里，如果发现 `Session` 里的 `username` 为空，则返回 `Action.LOGIN`，跳转回登录页面。

步骤四 编写处理登录请求的 `LoginAction.java` 类。

```

1  package action;
2  import java.util.Map;
3  import com.opensymphony.xwork2.ActionContext;
4  import com.opensymphony.xwork2.ActionSupport;
5  import service.CheckUser;
6
7  public class LoginAction extends ActionSupport {
8      private String username;
9      private String password;
10
11     public String execute() throws Exception {
12         // 设置 Session

```

```
13      Map sessionMap = ActionContext.getContext().getSession();
14      sessionMap.put("username", username);
15      // 验证用户
16      if (CheckUser.isUser(username, password) ) {
17          return SUCCESS;
18      }
19      return LOGIN;
20  }
21
22      //省略 get 和 set username 和 password 的方法
23      ...
24  }
```

如果从 login.jsp 来的请求没有被拦截掉，那么在本 Action 的 execute 方法里，会通过第 16 行的 CheckUser.isUser 方法，判断登录的用户名和密码是否是合法的。如果是，则在第 17 行里，返回 SUCCESS；如果不是，则在第 19 行里返回 LOGIN。下面来看一下定义 isUser 方法的 Checkuser 类。

```
1  package service;
2  public class CheckUser {
3      //判断用户是否存在的方法
4      public static boolean isUser(String username, String password)
5      {
6          String name = username.trim();
7          String pwd = password.trim();
8          if (name.equals("Java") && pwd.equals("StrutsIntercept"))
9          {
10             return true;
11          }
12          return false;
13      }
14  }
```

在这个方法的第 8 行里，可判断用户名和密码是否是给定的值，从而分别返回 true 和 false。

步骤五 在 struts.xml 里定义诸多 Action 的属性，并部署拦截器，代码如下。

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts Configuration
2.3//EN"
3      "http://struts.apache.org/dtds/struts-2.3.dtd">
4  <struts>
5  <package name="interceptLogin" extends="struts-default" namespace="/">
6  <interceptors>
7  <interceptor name="loginCheck"
```

```

8         class="action.UserLoginInterceptor">
9     </interceptor>
10    <!-- 自定义拦截器栈 -->
11    <interceptor-stack name="myDefaultStack">
12        <interceptor-ref name="loginCheck"/>
13        <interceptor-ref name="defaultStack"/>
14    </interceptor-stack>
15 </interceptors>
16 <default-interceptor-ref name="myDefaultStack"/>
17
18 <global-results>
19 <result name="login" type="redirect">/login.jsp</result>
20 </global-results>
21
22 <action name="login" class="action.LoginAction">
23 <result name="success" type="redirectAction">
24 <param name="actionName">HelloUser</param>
25 <param name="namespace"></param>
26 <param name="username">${username}</param>
27 </result>
28 </action>
29
30 <action name="HelloUser" class="action.HelloUserAction">
31 <result name="success">/welcome.jsp</result>
32 <result name="login">/login.jsp</result>
33 </action>
34 </package>
35 </struts>

```

从第 5 行到第 16 行里，可以看到配置拦截器的一般步骤。

① 从第 7 行到第 9 行里，配置定义的拦截器类。

② 从第 11 行到第 14 行里，编写拦截器栈 myDefaultStack，并把定义好的拦截器 loginCheck 放入其中。

③ 在第 16 行里，通过<default-interceptor-ref name="myDefaultStack"/>，把已经定义好的拦截器栈 myDefaultStack 设置成系统默认的拦截器栈。

当服务器启动后，系统会默认拦截器栈是 myDefaultStack，并且还能知道在这个拦截器栈里，有 loginCheck 这样一个包含拦截器处理功能的类，所以一旦有请求想要发送到 Action 类，这个请求就会自动被 loginCheck 这个拦截器类来处理。

此外，从第 22 行到第 28 行可看到，如果在 LoginAction 里返回的是 SUCCESS，那么它

的 type 是 redirectAction，这里和以往的代码不同，以前是往 JSP 页面上跳转，而这里是往另外一个 Action 上跳转。并且从第 24 行的参数里，就能看出是往 HelloUser 跳转。

第 30 行到第 33 行的代码，是对 HelloUser 这个 Action 的定义，从中可看到，根据不同的返回值，能跳转到不同的 JSP 页面。

步骤六 编写 HelloUser 这个 Action，从配置文件里我们能看出，它对应于 HelloUserAction 这个类。

```
1 package action;
2 import com.opensymphony.xwork2.ActionSupport;
3 public class HelloUserAction extends ActionSupport {
4     String username;
5     public String getUsername() {
6         return username;
7     }
8     public void setUsername(String username) {
9         this.username = username;
10    }
11
12    public String execute() throws Exception {
13        return SUCCESS;
14    }
15 }
```

这个类非常简单，纯粹起到了传递参数的作用，在第 12 行的 execute 方法里，是直接返回 SUCCESS。从 struts.xml 这个配置文件可以看出，这里是把 username 这个参数直接传递到 welcome.jsp 这个页面里。

最后，编写 welcome.jsp。

```
1 <%@ page language="java" contentType="text/html; charset=UTF-8"
2   pageEncoding="UTF-8"%>
3 <%@ taglib prefix="s" uri="/struts-tags"%>
4 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
5   "http://www.w3.org/TR/html4/loose.dtd">
6 <html>
7 <head>
8 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
9 <title>登陆成功</title>
10 </head>
11 <body>
12 <center>
13 <s:property value="username"/>, 欢迎光临
```

```

12 </center>
13 </body>
14 </html>

```

在第 11 行里，会读取传递来的 `username` 参数，并展示 xxx，欢迎光临的效果。

4.4.3 拦截器的使用要点归纳

上文讲述了开发一个拦截器的必要步骤，包括如何写拦截器的处理类和配置文件，下面来总结一下拦截器的关键用法。

- ① 为了实现一个拦截器，可以继承 `AbstractInterceptor` 类。
- ② 可以在 `intercept` 方法里，定义拦截器要实现的功能。
- ③ 如果本拦截器发现无须拦截请求，则可以通过 `arg0.invoke()`，把请求传递给下个拦截器或者是 `Action`。
- ④ 在配置文件里，定义拦截器的一般代码如下所示，我们可以通过第 2 行到第 4 行的代码定义一个拦截器，可以通过第 6 行到第 9 行的代码定义一个拦截器栈，并放入已定义好的拦截器，最后通过第 11 行的代码，让 Struts 处理程序使用我们已定义好的拦截器栈。这样定义好的拦截器就能起作用了。

```

1 <interceptors>
2 <interceptor name="loginCheck"
3     class="action.UserLoginInterceptor">
4 </interceptor>
5 <!-- 自定义拦截器栈 -->
6 <interceptor-stack name="myDefaultStack">
7 <interceptor-ref name="loginCheck"/>
8 <interceptor-ref name="defaultStack"/>
9 </interceptor-stack>
10 </interceptors>
11 <default-interceptor-ref name="myDefaultStack"/>

```

4.5 为了尽快进阶，你必须知道如下知识

目前来看，使用 Struts 做项目的公司似乎不多，不过它确实是帮助大家了解 Web 框架开发的很好的敲门砖。如果面试者没做过 Struts，我们不会对他减分，不过一旦通过提问确认他确实有过 Struts 开发的经验，这也算是个加分项。

一个程序员做过多个 Struts 项目后，一定会使用过或了解一些技能点，这往往是我们面试的必考点。当然不仅是为了通过面试，掌握如下知识点后，对你 Web 框架技能的提升，多少是有些帮助的。

4.5.1 Action 里的跳转

在 Action 的 execute 方法里，会返回一个 String 类型的变量，在 struts.xml 配置文件里，会在如下格式的代码里，指定跳转的 type 类型：

```
<result name="login" type="redirect">/login.jsp</result>
```

这个 type 类型默认是 dispatcher，在大多数情况下会使用这个。也就是说，默认是跳转到 JSP 页面的。比如返回 Login，会跳转到 login.jsp，等等。

在刚才拦截器的例子里，可看到了 type 是 redirect 的情况，下面通过表 4.2 一起分析下各种 type 所定义的 Action 的跳转类型。

表 4.2 Action 的各种跳转方式汇总表

参数值	含义
dispatcher	默认值，用来转向页面，通常是跳转到 JSP 等页面
freemaker	处理 FreeMarker 模板，这个不常用
httpheader	控制特殊 HTTP 行为的结果类型，不常用
stream	通常用来处理文件下载，还可用于返回 AJAX 数据
velocity	处理 Velocity 模板,当前模板技术还算流行，Struts2 同样为模板作为 Result 做出了支持
xslt	处理 XML/XLST 模板
plainText	显示原始文件内容
chain	在一个 action 执行完毕之后，forward 到另外一个 action，所以他们之间是共享 HttpServletRequest 的
redirect	重定向到一个 URL，源地址与目标地址之间是 2 个不同的 HttpServletRequest。所以目标地址将无法通过 ValueStack 等 Struts2 的特性来获取源 Action 中的数据
redirectAction	重定向到另一个 Action

这里的 chain、redirect 和 redirectAction 都涉及跳转，所以我们面试的时候一般会问它们之间有什么差别？

① 如果用 redirect，Action 处理完后跳转到另一个资源（比如 JSP 页面）时，参数会全部丢失，而 action 处理结果也会全部丢失。

② redirectAction 一般是跳转到另一个 Action 里，同样，跳转后请求参数和 action 处理结果也会丢失。

③ 如果 type 是 chain，也是跳转到另一个 Action，这种情况下，参数会丢失，但 action 处理结果不会丢失。

4.5.2 可以指定 Action 里的处理方法

在上文的例子中，Action 的处理方法都是 execute，其实还可以指定由另外的方法来处理请求。

方法 1 在 JSP 页面的 Form 代码部分，指定处理方法，代码如下。

```
1 <form method="post" action="loginAction!login ">
```

这样我们就可以指定由 login 而不是 execute 方法来处理请求。

方法 2 配置 Action 的 method 属性，由此来指定处理方法，代码如下。

```
1 <action name="userLogin" class="xxx"method="login">
2     <result name="success">/success.jsp</result>
3     <result name="error">/error.jsp</result>
4 </action>
```

通过第 1 行的 method 方法属性，就能指定这个 Action 的处理方法是 login，而不是传统的 execute。

4.5.3 对 Struts 框架的进一步了解

Struts 作为一个基于 MVC 的框架，能很好地处理跳转业务，能让程序员把精力更多地集中到业务开发上，通过它，程序员能很好地了解框架编程的思路和一般方法。

不过任何框架都不是十全十美的，当你比较熟悉 Struts 框架后，一定能感受到它在项目开发里的一些缺陷。对于 Struts 的局限性，不同的人有不同的观点。但是请大家记住，在面试时，当你结合你的项目自信地说出 Struts 框架的局限时，即使面试官和你的观点不一致，他也一定会认为你对 Struts 有足够的认识。这里罗列出一些局限性供大家参考，如表 4.3 所示。

表 4.3 Struts 2 框架的局限性

局限性	结合项目说明
为每个请求创建一个 Action	Struts2 里，会为每个 http 请求实例化一个 Action 对象，这对处理高并发会有些难度
对 Action 执行前和后的处理支持不大好	比如每次进到 Action 前我们需要打印内存使用量，执行后需要记录跳转目标 URL，这个当然可以直接写到 Action 里，但大家比较下 Spring 的 AOP 处理方式，就会发现单纯把前后处理写到 Action 里有什么不足

续表

局限性	结合项目说明
对跳转的支持	如果在一个 Action 里，根据处理结果可能会跳转到 10 个页面，那么代码可能会比较烦琐。而且一旦跳转目标页面出现变更，比如换了目录，那么在修改配置文件后，可能要求重新部署和重启 Web 服务器，无法实现轻量级修改
安全性上的瑕疵	我们可以通过类似方式直接执行 Action 里的方法： <code>http://url:8080/proName/userLogin!list</code> 但事实上，该系统需要用户先登录才能开放用户列表。虽然我们可以通过定义拦截器来弥补这个缺陷，但毕竟属于额外的工作量
属于侵入式	在使用 Struts 时，需要继承 struts 的类，而且要编写 execute 方法。这样 Struts 框架就侵入到项目里了。比如做的是银行业务，如果哪天要把这套业务移植到其他 Spring 等框架项目里，可能工作量就比较大了

4.6 关于 Struts 框架的常见面试题

Struts 的地位似乎有些尴尬，一些小型（比如 10 个页面左右）的商业项目可能直接会用 JSP+Servlet+JavaBean+DB 的开发模式，一些大型基于企业级的项目往往采用 Spring+MyBatis 的框架。

所以本章没有过多地深入讲述诸如标签或内置拦截器等已经不怎么常用的技术，通过运行和理解本章的案例，大家可以很好地了解 Struts 框架的工作流程，从而进一步深入理解 MVC 的开发模式。

我们在面试初级程序员时，如果候选人没有在商业项目里用过 Struts 框架，这很正常，如果他在学校或者培训机构学过 Struts，这或许也可以成为一个加分项。但如果我们看到某人在最近的商业项目里用过，那么就会详细问原因，为什么这个公司现在还要用 Struts 框架？反而有不少人，在这个问题上会弄巧成拙。我们听到的最多的合理回答是，基于某种原因，比如历史原因或者客户要求，这个项目还是得用 Struts 框架。

具体到面试题，大家可以从网上找，在必要的时候，大家也可以多多益善地突击准备。我们给大家推荐回答问题的方式是“结合项目”，请大家看如下的问题点。

第一，说明在这个项目里，都用到了 Struts 的哪些组件，比如拦截器或者验证器，你是怎么用的？

第二，结合项目，说明自己做了哪些工作，比如在 Action 里，你怎么和业务以及数据库代码耦合？

第三，能结合项目，告诉考官一些技术细节，比如拦截器是怎么用的，在该项目里结合一个需求，告诉考官拦截器的工作流程和开发方式。或者结合项目说明下验证器的用法。

也就是说，任何技术都别停留在纸上，需要结合项目说明。

第四，说明使用 Struts 框架给这个项目带来哪些痛点？

第五，和 Spring 框架相比，你感觉各自有什么优缺点，请结合项目说明，不要空谈。

当你学好了 Spring 框架后就知道该怎么说？

第六，你的项目经常会扩展，业务实现方式也经常会变更，结合 Struts 框架说明一旦出现变更了你需要做哪些事？

比如需要更改业务，你该更改哪些文件？一旦更改了代码，如何部署到服务器上？

第七，这个项目的访问量是多少？最高的并发访问量能达到多少？Struts 框架能否很好地处理高并发的情况？

关于常用的技术问题，在本章里都已经提到，我们通过如下问题点来对本章做个总结。

- ① 在 Struts2 里，如何实现一个 Action？
- ② 怎么指定进入 Action 后该调用哪个方法？
- ③ 定义验证器的步骤是什么？
- ④ 定义拦截器的步骤是什么？
- ⑤ Struts2 中的 type 类型有哪些？如果不写 type，默认是什么？
- ⑥ 如何通过配置 type 类型，实现一个 Action 往另外一个 Action 的跳转？
- ⑦ 描述下 Struts MVC 的工作流程和开发模式。

⑧ 和 JSP+Servlet+JavaBean 的开发模式相比，Struts MVC 有哪些好处，同时，说明下 Struts 框架有哪些不足。

第 5 章

Spring 的基本知识点分析

可以这样说，Spring 颠覆了我们对编程的一些传统观念，所以要完全掌握 Spring 的精髓并不简单，不过大家一旦在大脑里固化了 Spring 的一些思维方式，就可以立竿见影地提升自己的能力。

Spring 的技术点主要分为四大块。第一是常规知识点，比如 SpringIoC 和 AOP 等；第二是 Web 应用方面的 Spring 的 MVC 框架；第三是和其他框架的整合技术，比如和 Hibernate 整合；第四是数据方面的应用，比如事务等。

本章主要讲述 Spring 的常规知识点，通过一些案例向大家展示一些不可思议的“编程方式”，通过学习，大家不仅能体会到 Spring 给我们项目开发带来的切实的好处，更能为了解 Spring 的其他知识点打好坚实的基础。

5.1 依赖注入的好处

不讲概念，先通过一个简单的程序，来看依赖注入的具体做法。

代码位置	视频位置
code/第 5 章/SpringIoCDemo	视频/第 5 章/Spring IoC 的讲解

5.1.1 一个基本的依赖注入的程序

这次我们是需要创建 Java 项目，在导入必要的包以后，从开发服务提供程序做起。

步骤一 开发提供服务的 SayHello.java 程序。

```
1 package com;
2 public class SayHello {
3     private HelloWorldSpring helloWorldSpring;
4     public HelloWorldSpring getHelloWorldSpring() {
5         return helloWorldSpring;
6     }
7     public void setHelloWorldSpring(HelloWorldSpring helloWorldSpring) {
8         this.helloWorldSpring = helloWorldSpring;
9     }
10    public void sayHello(){
11        System.out.println("Say Hello:" + helloWorldSpring.sayHello());
12    }
13 }
```

第 10 行定义了一个 sayHello 的方法，在这方法里，调用了在第 3 行定义的 helloWorldSpring 对象，输出一串文字。

这里有一个比较有意思的现象，虽然在第 4 行和第 7 行针对 helloWorldSpring 对象定义了 get 和 set 的方法，但在第 11 行使用 helloWorldSpring 对象之前，始终没有用 new 关键字初始化这个对象，那么按照以往的经验，会不会出现空指针异常呢？

别着急，先看下 HelloWorldSpring 这个类里有没有特殊的动作。

步骤二 定义 HelloWorldSpring.java 这个类。

```
1 package com;
2 public class HelloWorldSpring {
3     private String sayContent;
4     public String sayHello() {
5         System.out.println("HelloWorld Spring!");
6         return "Hello World Spring";
7     }
8 }
```

这里直接在第 4 行定义了 sayHello 的方法，也没看到特殊的代码。接下来看一下在

SpringMain 这个类里是如何调用的。

步骤三 开发调用者 SpringMain.java，请大家注意一下调用 sayHello 方法的方式。

```
1 package com;
2
3 import org.springframework.context.ApplicationContext;
4 import org.springframework.context.support.ClassPathXmlApplicationContext;
5 public class SpringMain {
6     public static void main(String[] args) {
7         ApplicationContext context = new ClassPathXmlApplicationContext(
8             "conf/applicationContext-*.xml");
9
10    SayHello sayHello = (SayHello) context.getBean("SayHello");
11    sayHello.sayHello();
12    }
13 }
```

关键代码在第 10 行和第 11 行，是通过一个 `getBean` 方法，获得了 SayHello 对象的一个实例，随后调用了其中的 `sayHello` 方法。在这之前，通过第 7 行的代码，装载了一个配置文件。

戏法变到这里，大家看到了两个不可思议的地方。第一，在 SayHello 类里，始终没有初始化 HelloWorldSpring 对象，就直接用了。第二，在 SpringMain 类里，没有像往常那样用

```
SayHello sayHello = new SayHello();
```

的方法初始化对象，而是通过 `getBean` 的方式来获得类的实例。其实，这里大家已经能看到“低耦合”的写法了。让我们最后看完 Spring 的配置文件再来综合体验 IoC 的好处。

步骤四 编写 Spring 的配置文件 applicationContext-service-api.xml，关键代码如下：

```
1 <bean id="HelloWorldSpring" class="com.HelloWorldSpring">
2 </bean>
3 <bean id="SayHello" class="com.SayHello" >
4 <property name="helloWorldSpring" ref="HelloWorldSpring" />
5 </bean>
```

在第 1 行和第 2 行里，定义了一个 Bean。在 Spring 里，一个 Bean 往往和一个类所对应，这里 id 是 HelloWorldSpring 的这个 Bean 是和 com.HelloWorldSpring 这个类所对应。

而在第 3 行到第 5 行里，用 id 是 SayHello 这个 Bean 对应 com.SayHello 这个类，请大家注意第 4 行，用内置 `property` 这个方式，把 HelloWorldSpring 这个类内嵌到 SayHello 类里。

5.1.2 IoC 的特点，不用 New 就可以初始化类

SpringMain.java 的主要代码如下。

```
1  ApplicationContext context = new ClassPathXmlApplicationContext(
    "conf/applicationContext-*.xml");
2  SayHello sayHello = (SayHello) context.getBean("SayHello");
3  sayHello.sayHello();
```

运行 SpringMain.java 时，首先是把配置文件里定义的信息装载到 context 类里；接着在第 2 行里，通过 context.getBean 方法，根据配置文件的定义，获取 ID 为 SayHello（即 class 是 com.SayHello）这个类；随后在第 3 行里使用这个类的 sayHello 方法。从代码中大家可以看出，这里同样没有用到 new，而是根据配置文件来初始化类。

没有使用 new，就意味着低耦合，具体而言，就是 SayHello、HelloWorldSpring 和 SpringMain 这三个类之间的耦合度很低。

假设有三个团队在开发维护这三个类，如果用常规 new 方法来创建类，比如在 SayHello 类里用 HelloWorldSpring helloWorldSpring = new HelloWorldSpring();，那么一旦管理 HelloWorldSpring 类的团队要修改调用的接口，比如 new 的构造函数需要带参数，那么 SayHello 类的管理者就不得不受无妄之灾，也需要修改本身的代码。

要知道在公司里，修改代码并且发布到生产环境，要经过很烦琐且很严格的审批流程，必须要经历代码审查、代码提交、测试人员测试、领导审批、最终发布以及发布后检查这些步骤。如果用刚才看到的通过配置文件装载类，在本地代码里没有 new 的这套开发方式，那么如果一个团队修改了代码，其他团队就有可能不必改动代码，这样就可以很大程度上避免不必要的工作。

5.1.3 控制翻转和依赖注入

控制翻转的英文名字叫 IoC（Inversion of Control），依赖注入英文名叫 DI（Dependency Injection），下面通过表 5.1 来看一下它们的概念。

表 5.1 相关概念的描述

概念名	含义	表现形式
控制翻转（IoC，控制反转）	类之间的关系，不用代码控制，而是由 Spring 容器（也就是 Spring 的 jar 包）来控制。控制权由代码翻转到容器里，这叫控制翻转	在初始化对象时，在代码里无须 new，而是把类之间的关系写到配置文件里

续表

概念名	含义	表现形式
依赖注入（DI）	在代码运行时，如果我们要在一个类里使用（也叫注入）另一个类，比如在上述的 SayHello 类里要初始化另外一个 HelloWorldSpring 类，那么这种注入就是依赖于配置文件的	同样是把类之间的调用关系写到配置文件里，在运行时，会根据配置文件，把 HelloWorldSpring 这个类注入 SayHello 里

通过上面的描述，能看到它们其实是从不同的角度讲述的同一件事情。依赖注入强调类的注入是由 Spring 容器在运行时完成，而控制反转强调类之间的关系是由 Spring 容器控制。

从这两个名词可知，Spring 给我们带来了一种全新的编程理念，即不用 new 也可以创建和使用对象。这种开发方式让我们能像搭积木一样组装不同的类，组装后的类之间的耦合度很低，一个类的修改可以不影响（或者影响度很小）其他的类，这样就可以避免一个小修改带来的一大串连锁反应。

大家在了解 Spring 的时候，一定请理解“低耦合”这个好处，这本来是面向对象思想带给我们的好处，在 Spring 开发的过程中我们确实能感受到。

5.1.4 面向接口编程的本质是缩小修改的影响范围

我们知道，建造房子的时候地基必须牢固，否则房子就不稳固。在程序里，服务提供商对外的调用方法也应该是相对稳固，假如某个银行的支付方法被 100 个外部程序调用，假设某天这个方式的参数由 2 个变成了 3 个，那么所有调用的 100 个外部程序都得修改。

我们期望的情况是，这个银行的支付方法在内部可以不断改进，比如增加性能等，但提供的方法，具体而言是方法名、参数的个数以及参数的类型等要尽量少改，否则影响面太大。

为了达到这个效果，可采用“面向接口编程”方法。我们知道，在 Java 里，接口是没有方法体的，因为没有，所以简单稳定。

从图 5.1 中可能看到面向接口的调用方式。其中，服务调用者是通过接口（而不是直接使用服务具体实现类）来使用服务，这样，服务的具体实现代码一旦改变，服务的调用者甚至感觉不到，所以也就无须跟着修改了。

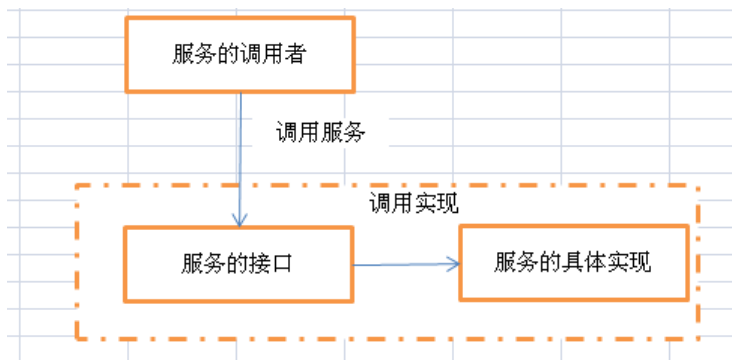


图 5.1 面向接口编程的调用方式

在使用集合时其实已经无意中用到这种编程方式了，比如：

```

1 List list = new ArrayList();
2 //非面向接口的写法是 ArrayList list = new ArrayList();
3 list.add("abc");

```

第 1 行，是用 List 这个接口，而不是 ArrayList 这个 List 接口的实现类来创建对象。第 3 行是通过接口提供的 add 方法来添加对象。

下面来看一下通过依赖注入实现面向接口的编程方式，这里给大家变的戏法是，抽象类可以不用实现，就能直接使用。

代码位置	视频位置
code/第 5 章/SpringIoCAbstractClass	视频/第 5 章/Spring 面向接口编程

步骤一 编写一个返回 Random 对象的接口 HelloInterface.java。

```

1 package BaseOnInterface;
2 public interface HelloInterface {
3     public abstract Random createRandom();
4 }

```

这是个纯粹的接口，在第 3 行里，定义了一个抽象方法 createRandom，它会返回一个 Random 对象。大家可以从随后的代码里看到，这个抽象方法始终没有实现。

步骤二 开发实现接口的类 HelloAbstract.java。

```

1 package BaseOnInterface;
2 public abstract class HelloAbstract implements HelloInterface
3 {
4     private Random random;
5     public void setRandom(Random random) {

```

```
6         thisandom = random;
7     }
8     publicabstract Random createRandom();
9 }
```

第 2 行可以看到，它 implements 了 HelloInterface 这个接口，第 4 行，定义了 Random 对象。第 8 行，还是没有实现 createRandom 方法，它依然是个抽象方法。

步骤三 在接口和实现里我们都用到了 Random 这个类，接下来，将开发这个类。

```
1 package BaseOnInterface;
2 public class Random {
3     private int randomNum = (int) (50 * Math.random());
4     public void printRandom() {
5         Syst          em.out.println(randomNum);
6     }
7 }
```

在这个类里，通过第 4 行的 printRandom 方法，返回了一个 50 以内的随机整数，这是一个真正服务的提供者，而在 HelloAbstract 里，是调用这个第三方的服务，向服务使用者返回随机数。

上述三段代码，可以通过表 5.2 来归纳三个类之间的关系。

表 5.2 三个相关类的关系

类名	作用	和其他类的关系
HelloInterface	是个接口，它向外部调用者提供了用于返回 Random 对象的 createRandom 方法	1 返回 Random 类 2 被 HelloAbstract 实现 3 Rando m 类如果更改了提供随机数的代码，我们无须修改这个接口的代码
HelloAbstract	实现了 HelloInterface 这个接口，外部调用者通过 createRandom 得到 Random 对象时，是和 HelloInterface 交互，根本不知道有这个类的存在	1 返回 Random 类 2 实现 HelloInterface 接口 3 Rando m 类即便更改了提供随机数的代码，我们也无须修改这个类的代码
Random	提供了返回随机数的 printRandom 方法	HelloInterface 和 HelloAbstract 提供的 createRandom 方法里，返回这个对象

步骤四 编写实际调用随机数服务的 SpringMain 类。

```
1 package BaseOnInterface;
2 import org.springframework.beans.factory.BeanFactory;
3 import org.springframework.beans.factory.xml.XmlBeanFactory;
4 import org.springframework.core.io.ClassPathResource;
```

```

5  import org.springframework.core.io.Resource;
6
7  public class SpringMain {
8      public static void main(String[] args) {
9          Resource res = new ClassPathResource("BaseOnInterface/bean.xml");
10         BeanFactory ft = new XmlBeanFactory(res);
11         HelloInterface h = (HelloInterface) ft.getBean("hello");
12         Random random = h.createRandom();
13         random.printRandom();
14     }
15 }

```

第9行，是通过 `Resource` 对象来读取配置文件的，第10行是通过 `BeanFactory` 来装载包含在配置文件里的类以及类之间的关系。

第11行是通过 `getBean` 的方式获得了一个 `HelloInterface` 的接口，请注意，是接口而不是 `HelloAbstract` 这个实现类，随后在第12行是通过 `h` 这个接口对象得到一个 `random` 对象，并在第13行调用其中的 `printRandom` 方法来打印随机数。

到此为止确实没有看到 `createRandom` 方法的实现，它始终是个抽象方法。这样的写法有什么好处呢？在接口和实现类里，无须关注 `Random` 对象是怎么创建的，即使提供随机数服务的 `Random` 对象出现变动，也不会影响到其他类。

最后来看一下提供实现面向接口编程和无须实现抽象方法的坚强保障，`bean.xml` 这个配置文件。

```

1  <?xml version="1.0" encoding="GBK"?>
2  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
   "http://www.springframework.org/dtd/spring-beans.dtd">
3  <beans>
4  <bean id="ran" class="BaseOnInterface.Random" />
5  <bean id="hello" class="BaseOnInterface.HelloAbstract">
6  <lookup-method name="createRandom" bean="ran"/>
7  </bean>
8  </beans>

```

在第4行可看到 `id` 是 `ran` 的 `bean` 其实是对应于 `Random` 这个对象。而在第5行到第7行声明 `hello` 这个 `bean` 里，是通过第6行的 `lookup` 方法，说明了 `createRandom` 方法返回的 `id` 是 `ran`，也就是 `Random` 这个对象，这也是为什么不用实现 `createRandom` 方法的原因。

在这个例子里，`SpringMain` 是最终调用随机数服务的客户，`HelloInterface` 就像搜索引擎一样，能让客户找到 `Random` 这个随机数服务，而 `Random` 是最终的服务提供者。通过面向接口的编程方式，各类之间的耦合度相当低，除了不用 `new` 外，甚至不用实现抽象方法，这

种做法会把修改范围降低到一个很低的程度。

场景 1，即使服务的供应商 `Random` 类变更了提供随机数的算法，由于 `HelloInterface` 接口和 `HelloAbstract` 类对外界提供的仅仅是 `createRandom` 这个抽象方法，所以无须改变。而对于 `SpringMain` 这个使用随机数的客户来说，先是从搜索引擎这里找到服务提供者，随后调用服务，所以服务内部细节的变动不会影响到 `SpringMain` 这个客户。

场景 2，搜索引擎在 `HelloAbstract` 里变更了搜索 `Random` 类的实现细节，比如本来是找个价格最便宜的服务，现在要找个性能最好的服务，那么服务提供者 `Random` 类和这个变更无关，无须改动，而最终用户用的是接口，所以甚至不用在意搜索细节的变动，所以也无须更改代码。

5.2 依赖注入的常用知识点说明

前面的两个例子能保证零基础的初学者快速地掌握 `Spring` 里依赖注入和面向接口的编程模式，下面将讲述一些常用的知识点。

5.2.1 读取配置文件的各种方式

在 `Spring` 里，通常在配置文件中描述各类以及类之间的包含关系，在使用的时候，会先加载配置文件，`Spring` 的内核会读取配置文件，随后动态地组装各类。

通过表 5.3 来总结一下读取配置文件的各种方式，它们之间没有优劣之分，大家可以挑选个最适用的。

表 5.3 `Spring` 读取配置文件的方式归纳表

类名	例子
<code>XmlBeanFactory</code>	<code>Resource resource = new ClassPathResource("bean.xml");</code> <code>BeanFactory factory = new XmlBeanFactory(resource);</code>
<code>ClassPathXmlApplicationContext</code>	<code>ApplicationContext factory=new</code> <code>ClassPathXmlApplicationContext("conf/appcontext.xml");</code>
用文件系统类来读取 <code>FileSystemXmlApplicationContext</code>	<code>ApplicationContext factory=new</code> <code>FileSystemXmlApplicationContext("classpath:appcontext.xml");</code>

5.2.2 单例多例，有状态无状态 Bean

我们知道，`Spring` 的容器会在程序运行时，根据配置文件自动地创建（或者叫实例化）

具体的 Java 类（也叫 class，或叫 Bean）。在配置文件里，可以设置创建文件时是否用单例的方式，如果没有设置，则会自动用默认的单例的方式来创建文件。

如果不想用单例，则可以通过如下两种语法来修改，它们是等价的。

```
<bean id="SayHello" class="com.SayHello" singleton="false">
```

或者

```
<bean id="SayHello" class="com.SayHello" scope="prototype">
```

在实际项目中，一般用单例模式来创建无状态的 Bean，而对于有状态的 Bean，一般不用这种模式。所谓无状态的 Bean，是指没有能够标识它目前状态属性的 Bean，比如共享单车，A 用好以后，可以放入共享池（即放到马路边上），B 可以继续使用。由于没有供某个特定的用户使用，所以也就不能保持某一用户的状态，所以叫无状态 Bean。相反，如果针对个人的自行车，那么会有个状态来表明是个人的。

讲到这里，请大家确认如下概念，并不是我们首先设置了 singleton 是 false，所以 Spring 容器才用单例的方式，恰恰相反，根据实际的需求，待创建的类可以被其他多个类共享，因此我们才设置 singleton 是 false。是先有需求再有具体的实现。

这个知识点可以说是 Spring 面试的必考点，下面通过表 5.4 来对比一下两者的差别

表 5.4 有状态、无状态 Bean 的使用情况

列别	实际用例	特点
有状态 Bean	我们访问网站登录后都有自己的用户名和密码，系统可以用一个有状态的 Bean 来记录我们的访问信息，比如来源 IP 访问页面列表和访问时间等	会为每次调用创建一个实例，一旦调用结束，比如用户离开了网站，则该 Bean 就会被销毁
无状态 Bean	数据库连接的通用类，其他类可以用它来获取数据库连接并进行操作	可以在缓冲池里只维护一个实例，无须创建和销毁操作，性能高，但是线程不安全

下面通过一个具体的代码来观察下单例和多例的用法。

代码位置	视频位置
code/第 5 章/SpringSingleton	视频/第 5 章/用单例和多例创建 Spring Bean

步骤一 创建一个提供随机数的类 Random.java。

```
1 package Demo;
2 public class Random {
3     private int i = (int) (100 * Math.random());
4     public void printRandom() {
5         Syst      em.out.println("输出随机整数:  " + i);
```



```
6    }  
7    }
```

在这段代码的第 4 行里，提供了 `printRandom` 的方法来返回一个 100 以内的随机整数。

步骤二 编写它的实际 Main 类。

```
1  package Demo;  
2  import org.springframework.beans.factory.BeanFactory;  
3  import org.springframework.beans.factory.xml.XmlBeanFactory;  
4  import org.springframework.core.io.ClassPathResource;  
5  import org.springframework.core.io.Resource;  
6  
7  public class Main {  
8      public static void main(String[] args) {  
9          Resource res = new ClassPathResource("Demo/bean.xml");  
10         BeanFactory ft = new XmlBeanFactory(res);  
11         Random r1 = (Random) ft.getBean("ran");  
12         Random r2 = (Random) ft.getBean("ran");  
13         System.out.println("Random 的两个实例是否指向同一个引用：" + (r1 == r2));  
14         r1.printRandom();  
15         r2.printRandom();  
16     }  
17 }
```

在第 8 行的 `main` 方法里，从第 9 行和第 10 行读取并装载了 `bean.xml` 这个配置文件。随后在第 11 行和第 12 行用同样的 `getBean` 方法，创建了两个不同的对象。在 13 行里，通过比较地址来分析它们是否是同一个。在第 14 行和第 15 行里，用 `r1` 和 `r2` 分别产生随机数。

步骤三 编写 `bean.xml` 这个配置文件。

```
1  <?xml version="1.0" encoding="GBK"?>  
2  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"  
    "http://www.springframework.org/dtd/spring-beans.dtd">  
3  <beans>  
4  <bean id="ran" class="Demo.Random" singleton="true"/>  
5  </beans>
```

请看一下第 4 行的代码，通过 `singleton` 属性来确立是用单例模式，随后大家可以运行步骤二里编写的 `Main.java`，结果如下：

```
1  Random 的两个实例是否指向同一个引用：true  
2  输出随机整数： 52  
3  输出随机整数： 52
```

从结果能看到，由于是单例，所以在 `Main` 类里有创建 `r1` 和 `r2` 的需求时，`Spring` 容器其

实只创建了一个对象。若把 bean.xml 里的第 4 行修改成 singleton 是 false，或者把 scope 设置成 "prototype"，再运行一下看结果，r1 和 r2 就是两个不同的对象了。

```
1 Random 的两个实例是否指向同一个引用：false
2 输出随机整数： 8
3 输出随机整数： 33
```

5.2.3 Spring 的注入方式与 AutoWire

在 Spring 里，经常会在一个类中注入另外一个类。比如在上面的例子里，我们的做法是：通过 property 和 ref，把 HelloWorldSpring 这个类注入 SayHello 里，这样在 SayHello 里无须 new 就可以用它了，这种做法能降低类之间的耦合度。

```
1 <bean id="SayHello" class="com.SayHello" >
2 <property name="helloWorldSpring" ref="HelloWorldSpring" />
3 </bean>
```

此外，还可以用 autowire 来组装不同类，这个关键字的字面含义是自动装配，可以用它来指定注入的方式。在 Spring 里，它有 6 种方式，但常用的是 byName 和 byType。下面通过表 5.5 来归纳一下 autowire 的各种取值。

表 5.5 autowire 各种取值的含义

取值	说明
no	不使用自动装配，必须显式地使用<ref>标签明确地指定 bean
byName	根据属性名自动装配。此选项将检查容器，根据名字查找与属性完全一致的 bean，并将其与属性自动装配
byType	如果容器中存在一个与指定属性类型相同的 bean，那么将与该属性自动装配。如果存在多个该类型的 bean，那么将会抛出异常，并指出不能使用 byType 方式进行自动装配。若没有找到相匹配的 bean，则什么事都不发生，属性也不会被设置
constructor	与 byType 的方式类似，不同之处在于它应用于构造器参数。如果在容器中没有找到与构造器参数类型一致的 bean，那么将会抛出异常
autodetect	通过 bean 类的自省机制（introspection）来决定是使用 constructor 还是 byType 方式进行自动装配。如果发现默认有构造器，那么将使用 byType 方式
default	由上级标签<beans>的 default-autowire 属性确定

接下来我们来看下 byName 的使用要点。

代码位置	视频位置
code/第 5 章/SpringAutoWireDemo	视频/第 5 章/SpringAutoWire By Name 的讲解

第一，由于需要在使用 SayHello 时自动装配 HelloWorldSpring，因此需要在第 2 行定义

这个对象。请大家注意，这里的变量名是 `helloWorldSpring`。并且需要针对这个变量编写 `get` 和 `set` 方法。

```
1 public class SayHello {
2     private HelloWorldSpring helloWorldSpring;
3     public HelloWorldSpring getHelloWorldSpring() {
4         return helloWorldSpring;
5     }
6     public void setHelloWorldSpring(HelloWorldSpring helloWorldSpring) {
7         this.helloWorldSpring = helloWorldSpring;
8     }
9     public void sayHello(){
10        System.out.println("Say Hello:" + helloWorldSpring.sayHello());
11    }
12 }
```

第二，修改配置文件，请大家对比一下两个不同的版本。

ByName 的版本。

```
1 <bean id="helloWorldSpring" class="com.HelloWorldSpring"/>
2 <bean id="SayHello" class="com.SayHello" autowire="byName">
3 </bean>
```

原始版本。

```
1 <bean id="HelloWorldSpring" class="com.HelloWorldSpring">
2 </bean>
3 <bean id="SayHello" class="com.SayHello" >
4 <property name="helloWorldSpring" ref="HelloWorldSpring" />
5 </bean>
```

通过对比发现，在 ByName 的第 1 行，它的 id 是 `helloWorldSpring`，是小写，这个和 `SayHello.java` 第 2 行的变量是一致的。而且，在 ByName 版本的第 2 行里，不需要为 `SayHello` 加入 `property` 这种引入 `HelloWorldSpring` 的方式，而是通过 `autowire` 这种方式。

在运行过程中，`SayHello` 这个类一旦发现出现 `helloWorldSpring` 这个变量，就会根据 `autowire="byName"` 这个提示，到配置文件里找 id 是 `helloWorldSpring` 所对应的 bean，随后根据 `class="com.SayHello"` 这个设置，初始化 `helloWorldSpring` 这个对象。

再来看下 `byType` 的使用要点。

它和 `byName` 的主体部分代码完全一致，差别主要在配置文件里，下面比较一下两者的配置文件。

byType 的代码。

```
1 <bean id="meanlessName" class="com.HelloWorldSpring"/>
2 <bean id="SayHello" class="com.SayHello" autowire="byType">
3 </bean>
```

byName 的代码。

```
1 <bean id="helloWorldSpring" class="com.HelloWorldSpring"/>
2 <bean id="SayHello" class="com.SayHello" autowire="byName">
3 </bean>
```

请大家阅读 byType 的第 1 行代码，这里的 id 不需要和 SayHello 里的变量名 helloWorldSpring 一致。当 SayHello 遇到这个变量时，由于 autowire="byType"，它会到配置文件里找 type 是 HelloWorldSpring 的类，随后装配。如果发现配置文件里有两个不同的 bean，且它们都是 com.HelloWorldSpring，则会提示异常。

5.2.4 @AutoWired 注解

在 Spring 里，还会遇到一种新的编程方式，那就是通过注解来定义某些属性。

目前这种做法还有人用，不过请大家同时了解下注解方式的缺点，这些经验是资深程序员总结出来的，如果大家在面试的时候能说出如下观点，那么面试官就会认为你比较熟悉 Spring。

第一，代码可读性不好，不容易维护，因为我们不得不在代码里找到依赖关系。

第二，通用性不好，比如哪天我们的代码不用 Spring 了，那么我们就得一个一个删除。

在上个部分，我们是通过配置文件设置 autowire 的方式，下面我们演示通过 @AutoWired 的方式来注入类，@Autowired 等同 autowire="byType"。

代码位置	视频位置
code/第 5 章/SpringAnnotationDemo	视频/第 5 章/通过注解实现 Spring 的 IoC

下面重用 5.2.3 节里的例子，不过要做些修改。

修改点 1，在配置文件里添加<context:annotation-config />。

修改前做 byType 的代码。

```
1 <bean id="meanlessName" class="com.HelloWorldSpring"/>
2 <bean id="SayHello" class="com.SayHello" autowire="byType">
3 </bean>
```

修改后：

```
1 <bean id="meaninglessName" class="com.HelloWorldSpring"/>
2 <bean id="SayHello" class="com.SayHello"/>
3 <context:annotation-config />
```

区别在于，在第 2 行，没有了 `autowire` 的设置，而且在第 3 行加了一段话，如果要用注解，就要添加这段话。

修改点 2，在原来的 `SayHello` 这个类的 `helloWorldSpring` 变量前，多加个 `@Autowired`，就像代码第 3 行那样。

```
1 public class SayHello {
2     // 这里多加个@Autowired
3     @Autowired
4     private HelloWorldSpring helloWorldSpring;
5     public HelloWorldSpring getHelloWorldSpring() {
6         return helloWorldSpring;
7     }
8     public void setHelloWorldSpring(HelloWorldSpring helloWorldSpring) {
9         this.helloWorldSpring = helloWorldSpring;
10    }
11    public void sayHello(){
12        System.out.println("Say Hello:" + helloWorldSpring.sayHello());
13    }
14 }
```

这样一来，即使没有在配置文件里加 `autowire="byType"`，但在变量 `helloWorldSpring` 前加了 `@Autowired`，就也能像 `byType` 那样自动装载这个类了。

也可以通过 `@Resource` 来自动装载，只不过它等同于 `autowire="name"`。

5.2.5 Bean 的生命周期

在使用 Spring 带给我们便利时，会发现有些动作是不可见的，比如我们只需在配置文件里编写类之间的耦合关系，Spring 容器就会在运行时自动装配。

在某些重要项目里，有必要监控 Bean 从创建到销毁的全过程。比如在某银行项目里，创建了一个 Bean 用来管理用户存钱的操作。如果不监控，一旦这个 Bean 在 Spring 容器处理过程中出了问题（Spring 容器本身出错概率不大，但程序员某些错误的做法可能会导致问题），那么就无法定位到错误的具体位置，这是非常可怕的。错误本身不可怕，是可以用 `try...catch` 之类的代码来监控和处理异常的，但如果我们不知道错误的来源，那么就根本谈不上处理了。

目前有个比较有意思的现象,有些公司在做项目时其实不用关心 Bean 从创建到销毁的流程,他们的项目里用 IoC 或者其他一些基本技能就足够了,但他们在招聘面试时还会具体询问 Bean 生命周期的知识点,美其名曰考查程序员的基本功。不过这种做法也有一定的道理,掌握 Spring 生命周期的各知识点能帮助我们更好、更高效地开发项目。下面通过表 5.6 来看一下 Bean 生命周期里的各重要时间点。

表 5.6 Bean 生命周期一览表

步骤	说明
容器启动, 读取配置文件实例化所有实现了 BeanFactoryPostProcessor 接口的 Bean, 如果在配置文件里有定义, 注入 Bean 的属性	BeanFactoryPostProcessor 接口里定义了 postProcessBeanFactory 方法, 如果有必要, 我们会实现这个方法。在 Bean 加载前, 会执行实现后的该方法
读取配置文件, 实例化剩下的 Bean, 如果在配置文件里有定义, 注入 Bean 的属性	在配置文件里, 我们可以定义 Bean 的初始化属性, 比如一个账户 Bean, 我们在配置文件里设置了初始化金额是 0 元, 那么在创建 Bean 时, 会自动初始化该属性
如果有 Bean 类实现了 BeanNameAware 接口, 则执行其 setBeanName 方法	Bean 被创建后, 一般不会知道它在容器里的名字, 但在某些场景下, 比如我们需要对零件编号, 用有编号的零件搭建产品, 那么我们可以通过 setBeanName 方法, 给某个零件 Bean 赋个特定的编号
如果有 Bean 类实现了 BeanFactoryAware 接口, 则执行其 setBeanFactory 方法。	用这个方法可以设置创建 Bean 的工厂名, 一般不怎么用。比较可能的用途是, 在根据某个配置文件创建了一个新工厂之后, 重写这个接口设置工厂名
如果有 Bean 类实现了 ApplicationContextAware 接口, 则执行其中的 setApplicationContext 方法	上述的三个 aware 方法, 一般不怎么调用, 除非有比较特殊的需求
如果有实现 BeanPostProcessors 接口的实例, 则任何 Bean 在初始化之前都会执行这个实例的 processBeforeInitialization()方法	一般不怎么调用, 除非有比较特殊的需求
如果有 Bean 类实现了 InitializingBean 接口, 则执行其 afterPropertiesSet 方法	一般不怎么调用, 除非有比较特殊的需求
如果有在 Bean 在配置文件中“init-method”属性设定方法初始化方法, 则调用该方法	如果在初始化 Bean 时有些特殊的动作, 那么我们可以通过配置文件设定 init-method 属性, 从而把特殊的动作放到初始化方法里
如果有实现 BeanPostProcessor 接口的 Bean, 那么调用 postProcessAfterInitialization 方法	一般不怎么调用, 除非有比较特殊的需求
Bean 被使用	
在容器关闭时, 如果 Bean 类实现了 DisposableBean 接口, 则执行它的 destroy()方法	比如在打开容器时, 初始化了某些资源, 那么可以在 destroy 方法里统一关闭
在容器关闭时, 可以在 Bean 定义文件中使用“destroy-method”定义的方法	

5.3 AOP，面向切面编程

面向切面编程也叫 AOP（Aspect Oriented Programming），它是 Spring 的重要特性，通过它，能很大程度上降低各业务类之间的耦合度。下面将结合项目需求来讲述 AOP 的知识点。

5.3.1 面向切面编程的使用场景

我们可以把一些通用的方法集成到基类里，比如中国人、美国人等各国人都会学习，那么就可以定义人类这个基类，把学习方法放在这个基类里各国人的类只要继承这个基类，然后重写这个方法即可，这种面向对象的编程方式能提升代码的可读性和可维护性。

现在有如下需求，在调用项目里的很多方法之后，需要调用内存回收的方法，用如下方式可以实现这个需求。

```
1 void f1() {  
2     f1    的正常业务  
3     //    回收内存  
4     clearMem();  
5 }  
6 void f2() {  
7     f2    的正常业务  
8     //    回收内存  
9     clearMem();  
10 }
```

这种实现方法的问题在于，我们不得不在所有必要的方法里写上调用回收内存的代码，这样会造成代码重复。如果有修改，还不得不修改很多处重复的地方。

更糟糕的是，假如回收内存的方法由内存管理团队来维护，而正常的 f1 和 f2 之类的业务由业务团队来维护。假如哪天内存管理团队升级代码，比如修改了 clearMem 方法名，或者变更了参数，那么会连带着业务团队也要变动代码。之前说过，项目代码如果要部署到生产环境上，一般需要比较大的代价，内存管理团队部署代码，这是情理中的事，但就没有必要牵连到业务团队了。

在这种场景下，面向对象的编程方式就帮不到我们了，就需要用面向切面的编程方式了。

5.3.2 面向切面编程的案例演示

有这样的需求，在银行项目里，当我们调用给账户加钱和扣钱的方法前后，需要完成一些固定的动作，比如记流水或者安全性检查，可以通过如下步骤用 AOP 的方式实现。

代码位置	视频位置
code/第5章/SpringAOPDemo	视频/第5章/Spring AOP 的讲解

步骤一 编写账户的接口和实现类。这里依然遵循着面向接口编程的思路。

接口代码是 `Account.java`，其中在第 2 行和第 3 行定义了两个方法。

```
1 public interface Account{
2     void add(int money);
3     void minus(int money);
4 }
```

实现类的代码是 `AccountImpl.java`，其中在第 2 行定义了账户名这个属性，并在第 3 行和第 6 行定义了它的 `get` 和 `set` 方法。

```
1 public class AccountImpl implements Account {
2     private String name;
3     public String getName() {
4         return name;
5     }
6     public void setName(String name) {
7         this.name = name;
8     }
9     public void add(int money) {
10        System.out.println("给" + name + "账户加钱 " + money + "元");
11    }
12    public void minus(int money) {
13        System.out.println("从" + name + "账户扣钱: " + money + "元");
14    }
15 }
```

步骤二 编写前置处理类 `BeforeAdvice.java`，在项目里需要在调用加钱和扣钱方法之前，调用封装在其中的 `before` 方法。

```
1 import java.lang.reflect.Method;
2 import org.springframework.aop.MethodBeforeAdvice;
3 public class BeforeAdvice implements MethodBeforeAdvice
4 {
5     public void before(Method m, Object[] args, Object target) throws Throwable
6     {
7         System.out.println("在方法调用之前");
8         System.out.println("执行的方法是:" + m);
9         System.out.println("方法的参数是:" + args[0]);
10        System.out.println("目标对象是:" + target);
11    }
12 }
```



```
12 }
```

请大家注意第 3 行，需要实现 `MethodBeforeAdvice` 接口，第 5 行，可以在 `before` 方法里定义前置操作的业务动作。这里我们是打印了一些信息。

同样，在 `AfterAdvice.java` 里，定义了后置操作。它和前置操作很像，这里是在第 5 行实现了 `AfterReturningAdvice` 接口里的 `afterReturning` 方法，同样是打印了一些信息。

```
1  import org.springframework.aop.AfterReturningAdvice;
2  import java.lang.reflect.Method;
3  public class AfterAdvice implements AfterReturningAdvice
4  {
5      public void afterReturning(Object returnValue, Method m, Object[] args, Object
        target) throws Throwable
6      {
7          System.out.println("在方法调用之后");
8          System.out.println("执行的方法是：" + m);
9          System.out.println("方法的参数是：" + args[0]);
10         System.out.println("目标对象是：" + target);
11     }
12 }
```

此外，还可以定义环绕操作，代码如下。

```
1  import org.aopalliance.intercept.MethodInterceptor;
2  import org.aopalliance.intercept.MethodInvocation;
3  public class AroundInterceptor implements MethodInterceptor
4  {
5      public Object invoke(MethodInvocation invocation) throws Throwable
6      {
7          System.out.println("调用方法之前：invocation 对象：" + invocation + "");
8          Object val = invocation.proceed();
9          System.out.println("调用结束...");
10         return val;
11     }
12 }
```

这里是在第 3 行实现了 `MethodInterceptor` 接口，一旦调用到第 5 行定义的 `invoke` 方法时，Spring 容器就会把本尊方法（比如加钱或扣钱方法）的控制权交给 `invoke` 方法，而一般在本方法里，会加些前置（比如第 7 行的打印）和后置（比如第 9 行的打印）操作，而后通过第 8 行的动作执行本尊方法。

到目前为止，共定义了两套方法：第一套是本尊的加钱和扣钱方法，第二套是前置、后置以及环绕的动作。大家可以看到在本尊方法里，并没有任何前后环绕处理的业务，它们两

者是完全分离的，用专业的话来说，它们之间的耦合度很低。

步骤三 如何在 Main 类里使用，代码如下。

```
1  import org.springframework.context.ApplicationContext;
2  import org.springframework.context.support.FileSystemXmlApplicationContext;
3  public class Main
4  {
5      public static void main(String[] args) throws Exception
6      {
7          ApplicationContext ctx = new
8          FileSystemXmlApplicationContext("src/ApplicationContext.xml");
9          Account account = (Account)ctx.getBean("account");
10         System.out.println("第一个 add 方法");
11         account.add(100);
12         System.out.println("第二个 minus 方法");
13         account.minus(100);
14     }
15 }
```

在第 7 行和第 8 行里，装载并获取了 Account 的实例，并在第 10 行和第 12 行调用了加钱和扣钱的方法。

从调用的代码来看，依然看不出任何前置环绕处理的痕迹，那么它们是怎么装配起来的呢？

最后来看下配置文件，给出主体部分的代码。

```
1  <beans>
2  <!-- 配置目标对象-->
3  <bean id="accountTarget" class="AccountImpl">
4  <!-- 为目标对象注入 name 属性值-->
5  <property name="name">
6  <value>Java</value>
7  </property>
8  </bean>
```

在第 3 行里，定义了 accountTarget 这个 Bean，并在第 5 行到第 7 行，设置了 name 的初始值是 Java。

```
9  <!-- 第一个处理类-->
10 <bean id="myAdvice" class="BeforeAdvice"/>
11 <!-- 第二个处理类-->
12 <bean id="myAroundInterceptor" class="AroundInterceptor"/>
13 <!--指定了对哪些方法增加处理-->
```

```
14 <bean id="addAdvisor"
    class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
15 <!-- advice 属性确定处理 bean-->
16 <property name="advice">
17 <!-- 此处的处理 bean 定义采用嵌套 bean, 也可引用容器的另一个 bean-->
18 <bean class="AfterAdvice"/>
19 </property>
20 <!-- 确定正则表达式模式-->
21 <property name="patterns">
22 <list>
23 <!-- 确定正则表达式列表-->
24 <value>.*add*.</value>
25 </list>
26 </property>
27 </bean>
```

从第 9 行到第 27 行, 定义了 3 个处理类的 Bean, 其中前两个比较简单, 只是使用了前置处理类和环绕处理类。对于第三个后置处理类, 我们把第 18 行定义好的 AfterAdvice 这个 Bean 包装到第 14 行到第 27 行的 addAdvisor 这个 Bean 里, 同时, 通过第 21 行和第 24 行指定了该拦截器只能作用在 add 方法上, 对其它方法是没有效果的。

```
28 <!-- 使用 ProxyFactoryBean 产生代理对象-->
29 <bean id="account" class="org.springframework.aop.framework.ProxyFactoryBean">
30 <!-- 代理对象所实现的接口-->
31 <property name="proxyInterfaces">
32 <value>Account</value>
33 </property>
34 <!-- 设置目标对象-->
35 <property name="target">
36 <ref local="accountTarget"/>
37 </property>
38 <!-- 代理对象所使用的拦截器-->
39 <property name="interceptorNames">
40 <list>
41 <value>runAdvisor</value>
42 <value>myAdvice</value>
43 <value>myAroundInterceptor</value>
44 </list>
45 </property>
46 </bean>
47 </beans>
```

定义的前后环绕处理是作用在 Account 类的 add 和 minus 方法上的, 体现在第 29 行到第 46 行的 Account 这个 Bean 里, 而第 41 行到第 43 行里指定了要使用上文里定义好的三个处理

类。

当运行 Main 这个主类时，会看到如下输出，其中，有行号的是正式的输出，没有行号的是说明。

```

1  第一个 add 方法
2  这个是在 Main 类里调用 add 方法之前的输出
3  在方法调用之前
4  在 add 方法调用之前，调用定义在 BeforeAdvice 里的 before 方法
5  执行的方法是:public abstract void Account.add(int)
6  方法的参数是:100
7  目标对象是:AccountImpl@11d5b59
8  从第 3 行到第 5 行是 before 方法的输出，从中能看到执行的方法是 add，参数是 100，目标对象是
   AccountImpl
9  调用方法之前。 invocation 对象:[ReflectiveMethodInvocation: public abstract void
   Account.add(int); target is of class [AccountImpl]]
10 这个是针对 add 方法的环境操作的 invocation.proceed() 之前的输出
11 给 Java 账户加钱 100 元
12 这个是正常的加钱操作
13 调用结束...
14 这个是针对 add 方法的环境操作的 invocation.proceed() 之后的输出
15 在方法调用之后
16 这个是调用 AfterAdvice 里的 afterReturning 方法。
17 执行的方法是:public abstract void Account.add(int)
18 方法的参数是:100
19 目标对象是:AccountImpl@11d5b59
20 从第 10 行到第 12 行是 afterReturning 的输出
21 第二个 minus 方法
22 这个是在 Main 类里调用 minus 方法前的输出
23 在方法调用之前
24 同样是在 minus 方法调用之前，调用定义在 BeforeAdvice 里的 before 方法
25 执行的方法是:public abstract void Account.minus(int)
26 方法的参数是:100
27 目标对象是:AccountImpl@11d5b59
28 从第 15 行到第 17 行，同样是看到 before 方法里的输出。
29 调用方法之前。 invocation 对象:[ReflectiveMethodInvocation: public abstract void
   Account.minus(int); target is of class [AccountImpl]]
30 这个是针对 minus 方法环境操作的 invocation.proceed() 之前的输出
31 从 Java 账户扣钱: 100 元
32 调用 minus 方法时的输出
33 调用结束...
34 这个是针对 add 方法的环境操作的 invocation.proceed() 之前的输出
35 在配置文件里，由于 AfterAdvice 这个类是仅仅作用在 add 方法上，所以在 minus 方法后不会执行这个

```

5.3.3 深入了解面向切面编程的各种概念

从刚才的例子中能看到用 AOP 组合各种相关业务的好处，这个好处主要体现在耦合度低这个方面。从项目管理角度来看，如果要更改加钱和扣钱的前置操作，本来是要做安全验证，现在还要添加反洗钱管理，那么加钱和扣钱这两个方法本身不要做改动。

表 5.7 深入描述了一些重要的面向切面编程的概念。

表 5.7 面向切面编程的概念

概念名	说明
通知 (Advice)	有 5 种类型的通知，除了上文提到的 Before、After 和 Around 外，还有 After-returning（在方法成功执行之后调用）和 After-throwing（抛出异常后调用的通知）。一般用在通知里，定义前置、后置、环绕等操作中
连接点和切入点	任何可能触发通知的都是连接点，只想在特定的方法上执行通知。比如上文里只想在 add 里执行后置操作。那么执行通知的位置就叫切入点
切面 (Aspect)	通知和切入点的结合，具体而言，在调用 add 方法之前（切入点）想打印日志（前置通知），这种需求就是切面
目标 (target)	比如上文里的 Account 类，就可以是目标

在实际使用过程中，大家可以不用太多地了解概念，而是应该掌握如下知识点：

- ① 有哪些通知的类型？
- ② 结合项目说明你是如何使用 AOP 的？
- ③ 如果前置通知只想作用在某些方法上，不想作用在全部方法上，该怎么做？

同样可以通过注解（而不是通过配置文件）来实现面向切面编程，这种做法的局限性在上文已经提到过，所以这里就不再一一赘述了。

5.4 如何证明自己了解 Spring 的基本技能

我们会经常用到 Spring 的 Web 开发技能、其他组件（比如 Hibernate）整合的技能以及数据库事务等技能，而 IoC 和 AOP 作为 Spring 的基础，更是会被频繁用到。

针对这些基础技能，我们一般不会问概念，因为这没法区分稍懂（刚学但没商业项目经验）和精通（工作 3 年左右正在往高级程序员升）的差别。

首先，我们会查看候选人对 Spring 的综合理解，一般来说，希望候选人能够结合项目实际说出 Spring 的优势，比如能告诉我们面向接口自由组装降低耦合之类特性在他的项目里是

怎么实现的，或者给他们的的项目带来了哪些具体的好处。这里我们很关注结合项目，因为这是判断候选人是否用过 Spring 以及是否精通 Spring 的必要条件。

其次，候选人既然做了 Spring 的不少项目，那么可以讲讲项目开发过程中用 Spring 时走的一些弯路，有哪些经验体会。我们听到的回答有大量用到了注解会对项目维护带来问题，或者是用单例创建 Bean 的时候对多线程之间的调用带来了一定的问题。

这方面我们一般不拘泥于答案本身，甚至只要别错得离谱就行，我们关注的是候选人对 Spring 的熟悉程度，以此来衡量他的 Spring 方面的经验。

最后，我们会从 Spring 里 Bean 的生命周期和源代码这两个方面来提问。这部分确实是比较资深的内容，对工作经验 3 年以内正要升级的程序员来说，我们的期望要求是能知道 Bean 的生命周期，遇到一些需求时，知道该调用哪些节点方法来实现。至于 Spring 内核实现 IoC 和 AOP 的源代码，这是对资深程序员（工作经验 5 年左右）的要求，初级程序员如果能回答上，就是个加分项。

下面总结下我们经常提问的问题点，请大家结合这些问题来检查本章的学习情况。

第一，阅读简历发现候选人做过 Spring 项目后，会请他结合实际项目说明是如何使用 IoC 的。

IoC 本身的技术点并不难描述，但千万记得要结合具体的项目，比如在银行项目里一个管理账户的类可以通过 IoC 的特性动态地加载另外一个管理安全风险的类，这样能减少耦合度。

第二，请结合项目说明你用过哪些 autowired 的种类。

第三，请说明你用过哪些注解。

@Autowired 之类的注解本身并不难描述，但这里请大家同样说下用注解的坏处，这样就能说明你做过不少 Spring 的项目。

第四，先问如何实现单例，如何实现多例，然后请候选人结合项目说下在哪种情况下该用单例模式（或多例）创建 Bean。

这里首先请了解设置单例的语法，然后请了解有状态和无状态 Bean 的概念和用途，最后再说明如何创建。这里的回答要点是，根据你的需求确认该用单例创建就行了。

第五，具体说下 Bean 的生命周期，说下你重写过其中的哪些方法。或者这样问，如果要给 Bean 编号，该怎么做？或者该如何设置 Spring 的初始化方法。总之，当你了解了 Spring 生命周期里各节点，遇到实际需求后，你就能清楚地了解该重写哪些方法来解决。

这里请大家熟悉整个 Spring Bean 的生命周期以及其中各节点的方法。

第六，说下你在项目里如何使用 AOP，大多候选人都说用 AOP 来实现日志打印，这确实是非常合适的用途。

这里同样请大家结合项目实际来说明。

第七，说下你在项目里具体用过哪些 AOP，比如前置、后置、环绕等。

这其实是个通俗的说法，更专业的问法是你用过哪些 AOP 的通知类型。我们知道 AOP 的通知类型一共有 5 种，在项目里不是用得越多越好，而是应该用在合适的场合，哪怕你们项目就用了前置，但用得很恰当，那就行。我们还真见过有人说 5 种都用过但用的场合不对，这可能就属于画蛇添足了。

第 6 章

Spring 的 MVC 框架

用 Struts 的 MVC 开发项目时，在业务代码里可能会较多地看到 Struts 的痕迹，比如业务和 Action 类关联度很大，用专业的话来讲就是 Struts 的 MVC 和业务代码的耦合度较高。

通过了解 Spring 的 IoC 和 AOP 等特性，可发现 Spring 比较擅长解耦合，所以 Spring 的 MVC 能做到和业务耦合度很低。

6.1 一个只包含 MVC 的案例

大家刚学，所以这里例子没有任何业务，纯粹是讲 MVC 的开发流程和运行流程。

代码位置	视频位置
code/第 6 章/SpringMVCDemo	视频/第 6 章/Spring MVC 的讲解

6.1.1 开发 Spring MVC 的代码

创建一个 Java Web 项目。

步骤一 编写 web.xml，在其中指定使用 Spring 的 MVC，主要的代码如下。

```
1 <servlet>
2 <servlet-name>spring</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
  -class>
3 <load-on-startup>1</load-on-startup>
4 </servlet>
5 <servlet-mapping>
6 <servlet-name>spring</servlet-name>
7 <url-pattern>/</url-pattern>
8 </servlet-mapping>
```

这里定义了一个 Servlet 和 servlet-mapping，它们一般是成对出现的。在第 1 行到第 4 行的 servlet 元素里，还定义了一个名为 Spring 的 Servlet，并在第 2 行指定了它的处理类是 Spring 的 DispatcherServlet。也就是说，在这个项目里是用到了 Spring 的 MVC 处理类，在第 3 行里通过 load-on-startup 来指定这个 Servlet 在 Spring 容器加载时就会被加载。

而在第 5 行到第 8 行的 servlet-mapping 里，通过第 7 行的 url-pattern 来指定任何请求都将被 Spring 这个 Servlet 来处理。

通常用 localhost:8080/项目名/index.jsp（IP 地址:端口号/项目名/目录名/JSP 文件名）这种方式来运行 Web 程序，这里的 url-pattern 是/。也就是说，IP 地址:端口号/项目名里的任何 URL 请求都将被 Spring 这个 Servlet 来处理。

步骤二 开发承担控制器角色的 RestController 类：

```
1 package com.mvc.rest;
2 import javax.servlet.http.HttpServletRequest;
3 import javax.servlet.http.HttpServletResponse;
4 import org.springframework.stereotype.Controller;
5 import org.springframework.ui.ModelMap;
6 import org.springframework.web.bind.annotation.PathVariable;
7 import org.springframework.web.bind.annotation.RequestMapping;
8 import org.springframework.web.bind.annotation.RequestMethod;
9 import org.springframework.web.servlet.ModelAndView;
10
11 @Controller
12 public class RestController {
13     public RestController() { }
14
15     @RequestMapping(value = "/login/{userName}", method = RequestMethod.GET)
16     public ModelAndView myMethod(HttpServletRequest request, HttpServletResponse
        response, @PathVariable("userName") String userName, ModelMap modelMap)
```

```

        throws Exception {
17         modelMap.put("loginUser", userName);
18         return new ModelAndView("/login/hello", modelMap);
19     }
20     //通过注解说明该方法用 Get 的方式来处理/welcome 的请求
21     @RequestMapping(value = "/welcome", method = RequestMethod.GET)
22     public String registPost() {
23         return "/welcome";
24     }
25 }

```

在第 11 行，通过@Controller 这个注解来说明 RestController 类承担了控制器的角色。其中在第 16 行和第 21 行分别定义了两个处理函数。先来看个比较简单的 registPost 方法。

第 21 行的@RequestMapping 这个注解，通过 method 来说明用 Get 方法来处理请求，一旦遇到了比如 localhost:8080/SpringMVCDemo/welcome 的请求时，将用这个 registPost 方法来处理。

另外的 myMethod 方法就比较复杂了。

```

1  @RequestMapping(value = "/login/{userName}", method = RequestMethod.GET)
2  public ModelAndView myMethod(HttpServletRequest request, HttpServletResponse
    response,
        @PathVariable("userName") String userName, ModelMap modelMap)
    throws Exception {
3      modelMap.put("loginUser", userName);
4      return new ModelAndView("/login/hello", modelMap);
5  }

```

一旦用户输入 localhost:8080/SpringMVCDemo/login/登录名时，该方法将用 Get 的方式来处理。这里 value 的写法是 "/login/{userName}"，也就是说，假如输入了 localhost:8080/SpringMVCDemo/login/Java，那么对应的 userName 值是 Java。

在第 3 行通过 modelMap 的 put 方法，将 userName 放到 loginUser 这个属性里，modelMap 对象一般用来传递数据到其它页面。在第 4 行，通过 ModelAndView 对象，携带包含 loginUser 参数的 modelMap 对象跳转到/login/hello 页面。

步骤三 编写针对 Spring MVC 处理的配置文件。这个配置文件名字可以随便起，一般是放在和 web.xml 相同的目录里，该配置文件的关键代码如下。

```

1  <!-- 自动扫描 bean，把作了注解的类转换为 bean -->
2  <context:component-scan base-package="com.mvc.rest" />
3  <!--打开 Spring MVC 的注解功能-->
4  <bean
    class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandler

```

```
Adapter" />
5  <!-- 对模型视图名称的解析，在其中为模型视图的名称添加前后缀 -->
6  <bean
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
7  <property name = "prefix" value="/"></property>
8  <property name = "suffix" value = ".jsp"></property>
9  </bean>
```

其中，第 4 行的代码可以开启注解模式，这样诸如@Controller 之类的注解就能生效了。第 6 行到第 9 行的代码给请求加上前缀和后缀。

步骤四 编写两个 jsp 文件，请注意 hello.jsp 是放在 Webroot/login 目录下。先来看 hello.jsp 代码。

```
1  <html>
2  <head>
3  <meta http-equiv="Content-Type" content="text/html; charset=GBK">
4  <title></title>
5  </head>
6  <body>
7  hello:<%=request.getAttribute("loginUser") %>
8  </body>
9  </html>
```

在第 7 行能显示 loginUser 的值。

welcome.jsp 代码就比较简单，就是在第 7 行固定显示 welcome 文字。

```
1  <html>
2  <head>
3  <meta http-equiv="Content-Type" content="text/html; charset=GBK">
4  <title></title>
5  </head>
6  <body>
7  welcome
8  </body>
9  </html>
```

6.1.2 Spring MVC 的运行流程

把 Spring MVC 和 Struts 部分的代码对比一下，发现 Spring 里不需要专门的配置文件来定义页面的跳转，而且控制器类的代码也比较随意，从中看不出明显的 Spring 痕迹。下面通过运行来说明它的执行流程。

方式一，启动 Tomcat 服务器后，通过 `http://localhost:8080/SpringMVCDemo/welcome` 的方式来访问。

① 向服务器发送 URL 请求后，根据 `web.xml` 的配置，该请求会由 Spring 的 `DispatcherServlet` 来处理。

② 根据 `@Controller` 注解，可知 `RestController` 类承担着控制器的角色。然后逐一去匹配各方法前的 `@RequestMapping`。

③ `/welcome` 能和 `registPost` 方法对上，所以执行其中的 `return "/welcome";` 代码，根据配置文件，会在其后加上 `.jsp` 的后缀，也就是说，会返回到 `welcome.jsp` 页面上。

④ 展示 `welcome.jsp` 页面，显示 `welcome` 的文字。

方式二，通过 `http://localhost:8080/SpringMVCDemo/login/java` 的方式来访问。

这里同样需要在 `RestController` 类里去逐一匹配 `@RequestMapping`，匹配后知道应该由 `myMethod` 方法处理。

```

1  @RequestMapping(value = "/login/{userName}", method = RequestMethod.GET)
2      public ModelAndView myMethod(HttpServletRequest request, HttpServletResponse
      response,
3          @PathVariable("userName") String userName, ModelMap modelMap) throws
      Exception {
4          modelMap.put("loginUser", userName);
5          return new ModelAndView("/login/hello", modelMap);
6      }

```

根据 `@RequestMapping` 的 `value` 值，可知 `userName` 的值是 `java`，在第 4 行把 `java` 这个值设置到 `loginUser` 里，然后跳转到 `/login/hello` 这个页面上。

同样这里需加上后缀 `.jsp`，跳转到 `/login/hello.jsp` 页面上，最后将显示 `hello:java` 的字样。

刚才针对具体项目描述 Spring MVC 的流程，现在抽象地看一下它的运行流程。

第一，URL 请求首先会被 `DispatcherServlet` 处理。

第二，请求会通过 `DispatcherServlet` 来进行转发，然后通过 `HandlerMapping` 接口的映射来访问具体的 `Controller`。

这里提到的 `HandlerMapping` 接口是用于处理请求的映射，它有两个实现类。

对于 `SimpleUrlHandlerMapping`，可以通过编写配置文件，把 URL 映射到 `Controller` 上。

对于 `DefaultAnnotationHandlerMapping`，可以通过编写注解，把一个 URL 映射到具体的

Controller 类上。

第三，不同的 URL 请求到达 HandlerMapping 时，HandlerMapping 会根据实际情况访问不同 Controller 类来进行进行处理。

第四，Controller 处理完成后，可以通过 ModelAndView 类来封装要返回的数据以及要跳转的目标页面。

第五，ModelAndView 封装完之后再通过一个视图解析器 ViewResolver 来进行解析，最后跳转到相应的页面上，从而完成本次请求响应。

6.1.3 与 Struts MVC 的区别

Spring MVC 框架目前要比 Struts MVC 流行，下面通过表 6.1 来对比一下两者的差别。

表 6.1 Spring MVC 和 Struts MVC 对比

对比项目	Spring MVC	Struts MVC
针对 URL 请求的处理方式	是针对方法级别,可以通过@RequestMapping 等注解用一个方法处理一类 URL 请求,这样有多少类请求,都可以通过扩展方法来实现	是类级别的拦截, Action 的一个方法(比如 execute)可以对应一个 URL,但 URL 的属性(比如参数)却被所有方法共享,这导致了有多少个可能的 URL 请求,我们就得写多少个 Action 类
拦截器的实现机制	用独立的 AOP 方式	Struts2 有自己的 interceptor 机制,会导致配置文件比 Spring MVC 要多
性能	一般来说, Spring MVC 要比 Struts MVC 好	
开发方式	Spring 的 MVC 可以通过注解来开发,比开发 Struts MVC 要灵活	

6.2 Spring MVC 的关键类说明

通过刚才的项目就能很快地了解 Spring MVC 的开发方式和运行流程,但这远远不够。下面将介绍一些 Spring MVC 的重要接口和类。

只要用 Spring 做过商业项目,那么你或多或少会对这些类有所了解,我们也经常会通过提问其中的一些细节来核实候选人是否真的做过 Spring 的项目,并且还真抓查出不少只具有纸面经验(没有真的做过但在简历上写熟悉 Spring)和学习经验(仅在学习的时候了解过 Spring,但没真的做过商业项目)的人。

6.2.1 通过 HandlerMapping 来指定处理的控制器类

在讲述 Spring MVC 的时候，大家了解过 HandlerMapping 这个接口，但在实际项目中，要根据实际情况创建多个控制器处理类（而不是仅仅就一个控制器处理类），这时就需要依靠这个接口来把不同类型的 URL 请求发送到对应的控制器类上。

这个接口的实现类主要有基于配置文件的 SimpleUrlHandlerMapping 和 BeanNameUrlHandlerMapping，以及基于注解的 AnnotationMethodHandlerAdapter。

下面就来依次看下这些 HandlerMapping 实现类的用法。

1. SimpleUrlHandlerMapping 实现类

代码位置	视频位置
code/第6章/SpringSimpleUrlHandleMappingDemo	视频/第6章/通过 HandlerMapping 指定控制器

创建好 Java Web 程序，并导入必要的 Spring 4.3 的 jar 后，就可以按如下步骤开发使用该实现类的 MVC 代码。

步骤一 开发 login.jsp，在这个页面中，用户可以输入用户名和密码，然后通过单击“提交”按钮来登录。

```
1 <%@ page language="java" pageEncoding="GBK" %>
2 <html>
3 <body>
4 <form action="loginAction" method="post">
5 登录用户:<input type="text" name="username"><br>
6 登录密码:<input type="text" name="password"><br>
7 <input type="submit" value="提交">
8 </form>
9 </body>
10 </html>
```

当用户单击“提交”按钮后，从第 4 行的代码里能看到，这个请求将以 post 方式，提交到 loginAction 上。

步骤二 在 web.xml 里指定使用 Spring 的 MVC，关键代码如下。

```
1 <servlet>
2 <servlet-name>spring</servlet-name>
   <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
   -class>
3 <load-on-startup>1</load-on-startup>
4 </servlet>
```

```
5 <servlet-mapping>
6 <servlet-name>spring</servlet-name>
7 <url-pattern>/</url-pattern>
8 </servlet-mapping>
```

在第 1 行到第 4 行里，定义了名为 **Spring** 的 **Servlet**，由此指定将由 **Spring** 的 **DispatcherServlet** 来处理请求。在第 5 行到第 8 行的 **servlet-mapping** 设定了我们定义的任何请求都将被 **Spring** 这个 **Servlet** 来处理。

步骤三 在 **spring-servlet.xml** 这个配置文件里，配置了用于分发请求到具体控制器处理类的 **SimpleUrlHandlerMapping**，代码如下。

```
1 <bean id="loginController" class="Controller.LoginController"/>
2
3 <bean id="handlerMapping"
4   class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
5   <property name="mappings">
6     <props>
7     <prop key="/loginAction">loginController</prop>
8     </props>
9   </property>
10 </bean>
11 <bean
12   class="org.springframework.web.servlet.view.InternalResourceViewResolver">
13   <property name="prefix" value="/"></property>
14   <property name="suffix" value=".jsp"></property>
15 </bean>
```

在第 1 行，定义了 **id** 是 **loginController** 的控制器类，从第 3 行到第 9 行，具体配置了 **SimpleUrlHandlerMapping**，其中通过第 7 行的如下代码，指定了 **loginAction** 这样的 URL 将由 **loginController** 这个控制器类来处理：

```
<prop key="/loginAction">loginController</prop>
```

这里只指定了一类 URL 的处理方式，事实上可以通过多个 **prop** 来配置多种映射方式。

在第 11 行到第 14 行里，定义了请求的前缀和后缀。

步骤四 编写控制器的处理类，从步骤三的代码里，已经知道该控制器处理类的名字是 **LoginController.java**。

```
1 package Controller;
2 import javax.servlet.http.HttpServletRequest;
3 import javax.servlet.http.HttpServletResponse;
```

```

4  import org.springframework.web.servlet.ModelAndView;
5  import org.springframework.web.servlet.mvc.AbstractController;
6
7  public class LoginController extends AbstractController{
8  public ModelAndView handleRequestInternal(HttpServletRequest request,
9      HttpServletResponse response)throws Exception{
10
11      String userName=request.getParameter("username");
12      //省略身份验证的动作
13      ModelAndView mav=new ModelAndView("welcome");
14      mav.addObject("userName",userName);
15      return mav;
16  }
17  }

```

这里展示了另一种通过配置文件实现控制器类的做法，但并没有用@Controller来指定控制器类，也没有在方法前加上注解。

在第8行的ModelAndView方法里，首先通过第11行的request.getParameter方法得到从login.jsp页面用post方式传递来的参数，随后在第13行定义了用户名页面跳转的ModelAndView对象，第14行在该对象里放入用户名，随后返回。

前面的配置文件中曾提到，在这个项目里针对每个请求都将加.jsp的后缀，所以当用户在login.jsp页面输入用户并单击“提交”按钮后，会经由该控制器处理类跳转到welcome.jsp页面。

而welcome.jsp页面比较简单，在第7行展示了welcome:用户名的文字。

```

1  <html>
2  <head>
3  <meta http-equiv="Content-Type" content="text/html; charset=GBK">
4  <title>welcome</title>
5  </head>
6  <body>
7  welcome:<%=request.getAttribute("userName") %>
8  </body>
9  </html>

```

2. BeanNameUrlHandlerMapping 实现类

代码位置	视频位置
code/第6章/SpringBeanNameUrlHandleMappingDemo	视频/第6章/BeanNameUrlHandlerMapping 用法说明

这里将在上个项目的基础上添加 BeanNameUrlHandlerMapping，也就是说，将用两个

HandlerMapping 把不同类型的 URL 请求映射到不同的处理类里，这也是项目中通常的做法。

添加点 1，在 Controller 这个包下添加 RegisterController.java。

```
1 package Controller;
2 import javax.servlet.http.HttpServletRequest;
3 import javax.servlet.http.HttpServletResponse;
4 import org.springframework.web.servlet.ModelAndView;
5 import org.springframework.web.servlet.mvc.AbstractController;
6
7 public class RegisterController extends AbstractController{
8     public ModelAndView handleRequestInternal(HttpServletRequest request,
9         HttpServletResponse response)throws Exception{
10
11         String userName=request.getParameter("username");
12         //省略注册的动作
13         ModelAndView model=new ModelAndView("welcome");
14         model.addObject("userName",userName);
15         return model;
16     }
17
18 }
```

这里的做法和刚才看到的 LoginController 类的做法很相似，都是继承了 AbstractController 类，而且获取了 username 的参数后跳转到 welcome.jsp 页面上。

添加点 2，添加用于注册的页面 register.jsp。

```
1 <%@ page language="java" pageEncoding="GBK" %>
2 <html>
3 <body>
4 <form action="registerAction" method="post">
5 注册用户:<input type="text" name="username"><br>
6 注册密码:<input type="password" name="password"><br>
7 <input type="submit" value="完成注册">
8 </form>
9 </body>
10 </html>
```

和刚才的登录页面的差别是，在第 4 行是往 registerAction 上跳转。

同时，在原来的 spring-servlet.xml 文件里，添加针对 BeanNameUrlHandlerMapping 的配置。

```
1 <bean
    class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>
```

```

2
3 <bean name="/registerAction" class="Controller.RegisterController">
4 </bean>

```

其中，在第 1 行引入 `BeanNameUrlHandlerMapping` 这个处理类，在第 3 行和第 4 行，通过 `name` 指定了 `/registerAction` 这样的 URL 将由 `registerController` 来处理。从这里可以看到，该映射类是通过 `name` 来指定 URL 样式的。

3. AnnotationMethodHandlerAdapter

我们之前用过这个，它是通过注解的方式来实现控制器类，现在来回顾下它的用法。

① 需要在配置文件里编写如下代码以开启注解。

```

1 <bean
  class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandler
  Adapter" />

```

② 开启后，用 `@Controller` 注解来标示控制器类。

6.2.2 通过视图解析器处理响应结果

在分析 Spring MVC 的流程时，可以看到，一旦在控制器类里完成处理后，就会返回一个 `ModelAndView` 对象，之后，该对象会被视图解析器（`ViewResolver`）解析，随后跳转到对应的页面上。

之前看到的 `InternalResourceViewResolver` 就属于视图解析器，一般会用它给请求加上前缀和后缀，下面列出了常规的用法。

```

1 <bean
  class="org.springframework.web.servlet.view.InternalResourceViewResolver">
2 <property name = "prefix" value="/"></property>
3 <property name = "suffix" value = ".jsp"></property>
4 </bean>

```

其中，`InternalResourceViewResolver` 比较常见，不过还可以使用其它的视图解析器。下面通过表 6.2 来看些常见的解析器类，如果没有其它需求，一般就用 `InternalResourceViewResolver` 即可。

表 6.2 解析器类一览表

类名	特点
AbstractCachingViewResolver	是个抽象类，会把它曾解析过的视图保存起来。先从缓存里找，如果找到了就直接返回。如果没有找到就创建一个新的视图对象，然后把它放到一个用于缓存的 map 中，接着再把新建的视图返回。使用这种视图缓存的方式可以把解析视图的性能问题降到最低
UrlBasedViewResolver	它是对 ViewResolver 的一种简单实现，而且继承了 AbstractCachingViewResolver，主要是用提供的一种拼接 URL 的方式来解析视图
InternalResourceViewResolver	是 UrlBasedViewResolver 的子类，所以 UrlBasedViewResolver 支持的特性它都支持。在实际应用中，InternalResourceViewResolver 也是使用最广泛的一个视图解析器
XmlViewResolver	继承自 AbstractCachingViewResolver，所以它也支持视图缓存。XmlViewResolver 需要给定一个 xml 配置文件。在该文件中定义的每一个视图的 bean 对象都给定一个名字，然后，XmlViewResolver 将根据 Controller 处理器方法返回的逻辑视图名称，到 XmlViewResolver 指定的配置文件中寻找对应名称的视图 bean 用于处理视图

此外，在 SpringMVC 中，还可以同时定义多个 ViewResolver 视图解析器，从而组成一个 ViewResolver 链。由此会引入一个问题，在遇到多个 ViewResolver 时，解析次序是怎样的？请大家按照如下思路来回答。

第一，当 Controller 处理器里的方法返回一个逻辑视图名称后，ViewResolver 链将根据其中 ViewResolver 的优先级来进行处理。具体到配置文件里的 ViewResolver，则是通过 order 属性来指定顺序的。order 属性是整数类型，数值越小，对应的 ViewResolver 就有越高的解析视图的权利。

第二，当一个 ViewResolver 在进行解析后返回的对象如果是 null 的话，就表示该 ViewResolver 无法解析该视图，这时如果还存在其他 order 值比它大的 ViewResolver，则会调用剩余的 ViewResolver 中的 order 值最小的那个来解析该视图，依此类推。

反之，当 ViewResolver 返回的是一个非空对象的时候，就表明该 ViewResolver 能够解析，那么解析视图这一步就完成了，后续的 ViewResolver 将不会再解析该视图。

第三，当定义的所有 ViewResolver 都不能解析该视图的时候，Spring 就会抛出一个异常。这里给出一个多个 ViewResolver 视图解析器的代码示例。

代码位置	视频位置
code/第 6 章/SpringMutilResolverDemo	视频第 6 章/在 Spring MVC 里配置多个视图解析器

在上述 SpringBeanNameUrlHandleMappingDemo 案例的基础上修改 spring-servlet.xml 文件。在其中配置多个视图解析器，相关代码如下。

```

1  <bean
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
2  <property name="viewNames" value="welcome*" />
3  <property name = "prefix" value="/" /></property>
4  <property name = "suffix" value = ".jsp" /></property>
5  <property name="order" value="1" />
6  </bean>
7
8  <bean
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
9  <property name="viewNames" value="account*" />
10 <property name = "prefix" value="/Account" /></property>
11 <property name = "suffix" value = ".jsp" /></property>
12 <property name="order" value="0" />
13 </bean>

```

在第 1 行和第 8 行分别配置了两个解析器，在第 5 行和第 12 行分别定义了它们的优先级。

第一个解析器里，在第 2 行通过 `viewNames` 定义了它该解析 `welcome*` 的视图，同时第 3 行和第 4 行指定了前后缀。第二个解析器里，指定了它该解析 `account*` 的视图，请注意它的前缀是 `/Account`。

情况 1，控制器处理类返回 `welcome`，根据 `viewNames` 的规则，该由第一个解析器处理，加上前后缀，该调用 `/welcome.jsp` 页面。

情况 2，当返回 `account` 时（当然这里没实现），该由第二个解析器处理，同样加上前后缀，该调用 `/Account/account.jsp` 页面。

刚才提到，如果有多个解析器，而且当 `order` 是 0 的解析器返回是空时，会交给第二个解析器处理。

但是由于 `InternalResourceViewResolver` 解析器内部有个 `buildView` 方法，该方法的返回一定不是空，所以这里如果单单写 `order` 而不配置 `viewNames` 属性，则 Spring 容器是只会调用 `order` 为 0 的这个解析器，所以针对该种解析器，应当把 `order` 结合 `viewNames` 使用。

6.2.3 通过 ModelAndView 返回视图结果

上述代码里是通过 `ModelAndView` 返回视图，该视图由视图解析器处理后跳转到具体的页面，它的一般用法如下所示。

```

1  ModelAndView model=new ModelAndView("welcome");
2  model.addObject("userName",userName);

```

```
3    return model;
```

下面来讲些它的常见用法。

1. 用来重定向

重定向有两种方式：一种是 `forward`，另一种是 `redirect`。

`forward` 是服务器直接访问目标地址的 URL，把那个 URL 的返回内容读取过来然后放置到浏览器上，用这种方式地址栏的 URL 保持不变。而 `redirect` 是在客户端重定向，当前请求返回后，服务器将给个状态标志，通过该标志就知道应该重新请求另一个 URL，具体表现就是地址栏的 URL 变成了新的 URL。

`ModelAndView` 支持这两种重定向的方式，它默认使用 `forward` 的方式。比如可以通过如下代码重定向到 404 页面，也可以把 404 页面改成其他需要的页面。如果要使用 `forward` 的话只需把 `redirect` 换成 `forward` 即可。

```
return new ModelAndView("redirect:/404.htm");
```

2. 在不同控制器之间跳转

在讲 `Struts` 时曾提到过一个 `Action` 可以跳转到另一个 `Action`，通过 `ModelAndView` 对象也可以实现类似的功能。

比如在一个控制器的方法里完成了添加用户这个操作，此时需要跳转到另外一个控制器方法里实现展示所有用户（包括新增的用户）的操作。用如下代码实现，其中 `displayAddUsers` 是目标控制器的方法：

```
return new ModelAndView("redirect:/displayAllUsers");
```

3. 在不同控制器之间带参数跳转

这里的需求比刚才更进一步，比如当前页面展示的用户是 Tom 的购物车页面，带有“购物车用户是 Tom”这个查询条件，当 Tom 再添加一个商品到购物车时，会有一个控制器方法执行添加购物车的操作，该操作完成后，应当带着“购物车用户是 Tom”这个参数返回到展示购物车的页面。

第一种实现方法是在 URL 里手动拼装参数，这类似于 `get` 的做法，比如：

```
new ModelAndView("redirect:/displayAllUsers?owner="+Tom+"&type="+1);
```

第二种做法是用 `RedirectAttributes` 的 `addAttribute` 方法，比如在控制器方法里我们写如下代码，在第 2 行通过 `RedirectAttributes` 来传递参数。

```

1 public String displayCart(RedirectAttributes attr){
2     attr.addAttribute("owner", "Tom");
3     return "redirect:/displayAll";
4 }

```

这样在 `displayAll` 这个目标控制器方法中,就可以通过如下获得参数的方式获得这个参数了。

```
request.getParameter("owner ");
```

第二种做法有些小问题,有可能会造成多次提交,比如用户按 F5 刷新页面,此时同样的数据会再提交一次。

当然,可以通过额外的代码来规避这个问题,也可以通过 `RedirectAttributes` 类的 `addFlashAttribute` 方法来实现。该方法的做法是把参数放在 Session 中,跳转后再移除。

具体的做法是,在 `displayCart` 方法里的第 2 行改用 `addFlashAttribute`,其余代码不变。而在目标方法 `displayAll` 里,通过 `ModelAttribute` 来获取,代码如下,在方法里,就可以用到 `owner` 这个变量了。

```

public String ModelAttribute (@ModelAttribute("owner") String owner)
{...}

```

6.2.4 Spring 的拦截器

类似于 Servlet 开发中的过滤器 Filter,在 Spring Web 里,也可以通过拦截器对请求进行预处理和后处理。具体而言,可以通过拦截器定义请求进入到控制器处理类之前(或之后)的动作。

在项目里,拦截器一般有如下的应用场景:

- ① 记录请求信息,以便进行信息监控和信息统计。
- ② 检查权限,比如检测请求进入之前是否登录,如果没有可以返回到登录页面。
- ③ 监控性能,比如可以通过拦截器记录请求进入处理器的开始时间,在处理后再记录结束时间,由此可以统计出该请求的处理时间。

在讲述 AOP 时其实就已经用过拦截器,当时实现环绕操作的关键代码如下。

```

1 public class AroundInterceptor implements MethodInterceptor
2 {
3     public Object invoke(MethodInvocation invocation) throws Throwable
4     {

```

```

5         System.out.println("调用方法之前");
6         Object val = invocation.proceed();
7         System.out.println("调用结束...");
8         return val;
9     }
10 }

```

第一行的 `MethodInterceptor` 其实就是一个拦截器对象，通过重写第 3 行的 `invoke` 方法，可以实现针对方法的拦截。当时这个拦截器是作用在加钱和扣钱方法上的，第 6 行的 `invocation.proceed` 方法是调用目标方法本身，通过配置文件，可以在第 6 行之前或之后定义前操作和后操作。

通过这个例子可以看到拦截器的一般作用：截获发往控制器方法的请求，并直接拦截掉，也可以在完成前置操作后把请求传递给控制器方法，当然还可以配置后置操作。

这个例子是用 `MethodInterceptor` 这个接口针对方法调用进行拦截，在 Spring MVC 应用里，一般是通过继承 `HandlerInterceptorAdapter` 来实现针对 URL 请求的拦截处理。

下面通过一个案例来具体了解下 Spring 拦截器在 Web 应用里的用法。

代码位置	视频位置
code/第 6 章/SpringInterceptorDemo	视频/第 6 章/Spring 拦截器的讲解

在 `SpringSimpleUrlHandleMappingDemo` 这个项目里，如果用户在浏览器里输入 `http://localhost:8080/SpringInterceptorDemo/loginAction`，那么会直接跳转到 `welcome.jsp` 页面，在其中会显示 `welcome:null` 字样，因为用户根本跳过了 `login.jsp` 这个登录页面。

这会造成安全问题，对此需要在这个项目的基础上添加拦截器来解决这个问题。

添加点 1，添加实现拦截器的类 `LoginInterceptor.java`，关键代码如下。

```

1 //通过继承 HandlerInterceptorAdapter 类来实现拦截器
2 public class LoginInterceptor extends HandlerInterceptorAdapter {
3     @Override
4     public boolean preHandle(HttpServletRequest request, HttpServletResponse
5         response, Object handler) throws Exception {
6         //获取 URL 地址
7         String reqUrl = request.getRequestURI();
8         if (reqUrl.contains("/loginAction")) {
9             String username = request.getParameter("username");
10            if (username == null ||
11                username.trim().equals("")) {
12                response.sendRedirect("login.jsp");

```

```

13         return false;
14     }
15     else
16     {
17         return true;
18     }
19     }
20     else
21     {
22         response.sendRedirect("login.jsp");
23         return false;
24     }
25     }
26
27     @Override
28     public void postHandle(HttpServletRequest request, HttpServletResponse
        response, Object handler, ModelAndView modelAndView) throws Exception {
29         super.postHandle(request, response, handler, modelAndView);
30     }
31 }

```

这里的关键代码在第 4 行的 `preHandle` 方法里，这个用于针对请求的前处理，当请求发送到 `LoginController` 这个控制器之前，会被这个方法拦截处理。同样，还可以在第 28 行的 `postHandle` 方法里实现针对请求的后处理，只不过这里没有添加后处理罢了。

我们知道，发往登录控制器处理类的请求一定包含 `loginAction` 字符串，所以需要在第 7 行判断请求中是否有这个字符串，如果没有，那么一定是非法请求，将在第 22 行和第 23 行的代码里跳转回登录页面。即使包含了 `loginAction`，如果在 `request` 里没有用户名，那么也可能没有通过登录页面，而是通过如下类似的 URL 来访问，`http://localhost:8080/SpringInterceptorDemo/loginAction`，因此也需要跳转回登录页面。只有当请求既包含 `loginAction`，又能看到用户名时，我们才在第 17 行返回 `true`，将请求交给后继的 `loginController` 这个控制器处理类继续执行。

添加点 2，在 `spring-servlet.xml`，我们需要添加针对拦截器的配置。

```

1 <bean id="loginController" class="Controller.LoginController"/>
2 <bean id="handlerMapping"
3     class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
4     <property name="mappings">
5     <prop key="/loginAction">loginController</prop>
6     </props>
7 </property>

```



```
8    </bean>
9
10   <bean
11       class="org.springframework.web.servlet.view.InternalResourceViewResolver">
12       <property name = "prefix" value="/"></property>
13       <property name = "suffix" value = ".jsp"></property>
14   </bean>
15
16   <context:component-scan base-package="Inteceptor"/>
17   <!--设置登录拦截器 -->
18   <mvc:interceptors>
19   <mvc:interceptor>
20   <mvc:mapping path="/*" />
21   <bean class="Inteceptor.LoginInterceptor"/>
22   </mvc:interceptor>
23 </mvc:interceptors>
```

新添加的代码在 15 行之后。在第 15 行我们指定了待扫描的包是 `Inteceptor`，因为我们把 `LoginInterceptors` 这个拦截器类放在了在这个包里面。

在第 17 行到第 22 行的 `mvc:interceptors` 这个配置里的第 19 行和第 20 行指定了任何请求（也就是 `/*`）都需要经由 `LoginInterceptors` 这个拦截器处理。

需要额外说明的是，由于添加了 `mvc:interceptors` 这个属性，所以需要在这个 `xml` 的定义里添加如下第 7 行所示的代码，否则会出错。用过拦截器的程序员一般都知道这点，所以我们有时也会用它来确认候选人是否真的用过 `Spring` 的拦截器。

```
1 <beans xmlns="http://www.springframework.org/schema/beans"
2       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3       xmlns:tx="http://www.springframework.org/schema/tx"
4       xmlns:context="http://www.springframework.org/schema/context"
5       xmlns:mvc="http://www.springframework.org/schema/mvc"
6       xsi:schemaLocation="http://www.springframework.org/schema/beans
7       http://www.springframework.org/schema/beans/spring-beans.xsd
8       http://www.springframework.org/schema/context
9       http://www.springframework.org/schema/context/spring-context-4.0.xsd
10      http://www.springframework.org/schema/tx
11      http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
12      http://www.springframework.org/schema/mvc
13      http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd ">
```

6.3 在 Spring MVC 方面你如何准备自己

Java 的优势主要集中在 Web 层面,如果你通过看 JD(职务介绍)发现面试的公司对 Spring 这块有要求,那么你就得好好准备 Spring 的 Web 方面的经验了,而我们面试的职位大多是 Web 方面的,所以也见过不少用过 Spring 的程序员。

假如目前某公司需要一个 Java 初级程序员(3 年左右工作经验),而且需要有 Spring 方面的经验,以这种需求我们面试过不少人,下面就通过表 6.3 来归纳下我们见过的情况。

表 6.3 Spring 方面我们面试过的各种情况归纳表

简历描述	面试表现	可能的后果
工作经验小于 1 年半(工作经验可以包括毕业前的实习经验),或者简历上没写 Spring 方面的经验,或者没有类似 Web 方面的经验		如果简历上没有其他出彩的地方(比如名校,有海外工作经验或者有其他我们各项目关注的条件等)无法通过简历初选的概率是 7 成
工作经验 2 到 3 年,但最近半年的项目和 Spring 或者 Java Web 开发项目无关		如果简历上没有数据库优化或,Java 性能调优之类的加分项,也有可能通不过初选,即使通过考官的技术面试,也会加上“最近没有相关项目经验”的评语
工作经验 2 到 3 年,最近在做 Spring 相关项目,如果简历上的项目和考官的项目的技术匹配度很高,工作年限可以降低到 1 年半左右	无法结合项目说出是怎么用 Spring 技术的,或者通过考官的询问说出以前只在学习上用过,或者 Spring 项目是培训机构给的	如果 Spring 技术确实可以,而且其他综合能力也不差,考官会让他通过,然后加上“没有 Spring 商业项目的经验”,如果有其他更好的候选人,考官不会录用没有商业项目的候选人
	能结合项目告诉考官 IoC 和 AOP 的用途,能结合项目叙述 Spring MVC 的流程,而且技术问题也能大致上说,但无法说出对技术的观点,比如用注解有什么优缺点。	至少 Spring 方面过关了,但考官会加上如下评语:能用 Spring 做项目,但对相关技术把握程度不深 这种候选人纯粹是仅仅能跟在项目经理后面做项目,一般很少有自己的发挥,离升级到高级程序员还有一定的距离
	不仅能说出一些只有用过才知道的技术或者调试方式,而且能说出自己的观点,比如我仅仅在 xxx 情况下 URL 里带参数的方式传值,原因是 xxx,或者我们项目综合考虑还是用到注解,原因是 xxx 对框架有自己的观点。能结合自己的项目说出具体框架的局限性,或者能说出用 Spring 框架	由于考官的要求是初级程序员,所以观点只要别太离谱就行 是否有自己的想法是一个重要的衡量标准。项目经理有时候没法顾及到细节,这时候就需要程序员自己发挥,所以考官遇到这类的候选人,一般是优先考虑的 很少有人能达到这种水准,这种候选人基本上可以升级到高级程序员了

续表

简历描述	面试表现	可能的后果
	解决哪些项目里的重大问题,或者架构师在搭建项目架构时,你参与了	
	针对某个项目的具体问题,能自己搭建基于 Spring 的框架	没见过,这属于高级程序员的能力范围了,写在这里是给大家一个努力的方向

这里的商业项目经验是说你参与的项目是否是用来挣钱的,所以兼职项目也算商业项目。如果没有毕业后的商业项目经验,那么读书阶段的实习项目也算聊胜于无,我们也录用过工作经验才 1 年但读书阶段在外面打工 2 年的候选人。

从上面的归纳情况来看,大家除了要关注“结合项目说明技术点”外,还得培养对 Spring 的综合意识,说具体点就是对某个技术的认识,它有哪些优缺点,适用于哪些场景,哪些场景下一定不能用 xxx 技术。

下面列出 Spring Web 方面的常见问题,除此之外,大家也可以自己不断收集,不断提升。

问题 1, 你们的项目是如何搭建 Spring Web 框架的, 具体而言, 如何定义控制器类, 视图解析器有几个, 一般是怎么定义的?

请结合项目的具体需求说下整个 Web 的处理流程, 别泛泛而言。比如是银行项目, 就拿一个具体的存钱请求, 怎么发 URL, URL 会经什么样的拦截器处理, 然后怎么被控制器类接收处理, 最后怎么经过视图解析器, 从而把结果返回到前端页面。

问题 2, 在项目里, 你们是否用到拦截器? 拦截器起了什么作用?

问题 3, 你们项目里有多少个控制器处理类? 你们怎么把不同的请求定位到具体的控制器处理类上的?

上述两个问题都是问技术, 不难回答。

问题 4, Spring 的拦截器和 Struts 的过滤器(Filter)有什么差别?

- ① 拦截器是基于 Java 反射机制的, 而过滤器是基于函数回调。
- ② 过滤器依赖于 Servlet 容器, 而拦截器不依赖于 Servlet 容器。
- ③ 拦截器只能对 action 请求起作用, 而过滤器则可以对几乎所有的请求起作用。
- ④ 拦截器可以访问 action 上下文、值栈里的对象, 而过滤器不能。

⑤ 在 action 的生命周期中，拦截器可以多次被调用，而过滤器只能在容器初始化时被调用一次。

问题 5，在你们的项目里，你们用到了 Spring Web 技术？解决了哪些比较麻烦的问题？

这个是开放性的问题，一般属于加分项，可以结合项目的实际情况来说。

问题 6，你感觉 Spring Web 框架有哪些可以改进的地方？或者你们项目在使用 Spring Web 框架时，有哪些问题是框架本身无法解决的？

这个考查对 Spring 框架的熟练程度，对初级程序员来说，他们未必能有体会，如果有人能回答上，而且大致不差，这也是个加分项。

问题 7，你们使用 Spring Web 框架做项目时，除了做开发以外，有没有参与框架搭建？对搭建 Spring Web 框架这方面，你有什么体会？

这个问题也属于开放性问题，对初级程序员来说难度比较大，我们一般是问高级程序员的，不过如果大家要走架构师这条路的话，这些针对框架的问题是大家提升自己能力的方向。

第 7 章

通过 ORM 专注业务（基础篇）

ORM 是对象关系映射（Object Relation Mapping）的缩写，用这种技术可以把数据库里的数据映射成 Java 对象（比如链表 Map 或 Set 等），从而能让我们在业务代码里通过操作 Java 对象而不是数据库来实现业务，使得我们更专注于业务本身，而不是更多地关注业务背后所对应的数据库操作。

专业地讲，通过 ORM 技术能做到“实现业务”和“操作数据库”的解耦合，通过这种解耦合的编程方式，能让项目里最有价值的核心部分——业务代码——保持稳定，这就是 ORM 技术的核心价值所在。

7.1 让你尽快掌握 Hibernate

目前能用 Hibernate 或 MyBatis 等技术来实现数据库对象到 Java 类的映射，其中比较主流的工具是 Hibernate，目前已经升级到 5.0 版本，下面将针对 Hibernate 5.0 来讲解 ORM 技术。为了演示方便，数据库我们用 MySQL。

7.1.1 通过 Hibernate 完成 ORM 的具体步骤

和 Spring 一样，可以通过配置文件和注解这两种方式来用 Hibernate 实现 ORM。下面将逐一讲述，在讲述的过程中，我们将抛出不少“海底针”，也就是只有做过 Hibernate 才知道的知识点，大家能以此来提升自己的能力。

代码位置	视频位置
code/第 7 章/HibernateDemo	视频/第 7 章/Hibernate 的简单案例说明

这里请大家注意，Hibernate 的配置文件是放在本项目的 lib 包里的，同样，这里将导入 MySQL 的驱动支持包。

步骤一 在 MySQL 的 hibernatchart 这个数据库里，创建 UserInfo 表，字段如表 7.1 所示。

表 7.1 UserInfo 表结构

字段名	类型	含义
UserID int		主键，用户的 ID
UserName varchar		用户名
Pwd var	char	密码
UserType var	char	用户类型

步骤二 创建名为 HibernateDemo 的 Java 项目，并导入必要的 jar 包。

步骤三 编写配置文件 hibernate.cfg.xml，一般这个文件放在 src 目录下，其中包含了数据库连接等配置信息。

```
1 <!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration
  DTD 3.0//EN" "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
2 <hibernate-configuration>
3 <session-factory>
4 <property name="hibernate.connection.driver_class">
5     com.mysql.jdbc.Driver
6 </property>
7 <!-- 数据库连接设置 -->
8 <property name="hibernate.connection.url">
9     jdbc:mysql://localhost:3306/hibernatchart
10 </property>
11 <property name="hibernate.connection.username">root</property>
12 <property name="hibernate.connection.password">123456</property>
13 <!-- 打印 SQL 语句-->
14 <property name="hibernate.show_sql">true</property>
```

```
15 <!-- SQL dialect 方言 -->
16 <property name="hibernate.dialect">
17     org.hibernate.dialect.MySQLDialect
18 </property>
19 <property name="hibernate.hbm2ddl.auto">
20     Validate
21 </property>
22 <!-- 添加实体类的映射文件-->
23 <mapping resource="Model/UserInfo.hbm.xml" />
24 </session-factory>
25 </hibernate-configuration>
```

在 Hibernate 里，Session 是一个比较重要的对象。和 Web 应用一样，它表示“会话”；和 Web 应用不同的是，Hibernate 中会话的主要任务是操作数据库，包括对表的增删改查和事务操作等。

在 Hibernate ORM 的流程中，一般是通过 Session Factory 来设计模式里的工厂模式创建 Session 会话的，在这个会话中，可以通过 SQL 或者 HQL（SQL 语句变种）把数据库里的表数据装载到 Java 的 Model 类（也就是映射类）里，也可以把 JavaModel 类中更改后的数据更新到数据表里。

在上述配置文件里，主要是从第 3 行到第 21 行定义了 SessionFactory 的配置，下面就通过表 7.2 来看一下其中的重要属性。SessionFactory 可以用这些属性连接到 MySQL 数据库，并创建 Session 对象。

表 7.2 配置文件里的重要属性

行号	属性名	含义
第 4 行	hibernate.connection.driver_class	连接数据库的配驱动包，这里是连接 MySQL 的，所以是用 MySQL 的驱动包
第 4 行	hibernate.connection.url	连接数据库的 URL。请注意 MySQL 的基本格式是： jdbc:mysql://ip:端口号/数据库名
第 11 行	hibernate.connection.username	连接 MySQL 数据库的用户名
第 12 行	hibernate.connection.password	连接 MySQL 数据库的密码
第 14 行	hibernate.show_sql	是否在控制台输出 SQL 语句，一般在调试的时候会写 true，上线后就会改成 false
第 16 行	hibernate.dialect	dialect 是“方言”，Hibernate 为了更好地对应各种关系数据库，因而对每种数据库都指定了一个方言 dialect

请重点关注第 19 行的 hibernate.hbm2ddl.auto 属性，该属性用来表明 Hibernate 在处理数据表时的做法，常用的取值如下。

① **create**: 表示启动时先 **drop** 数据表，再根据 `UserInfo.hbm.xml` 配置文件里的数据表定义 **create** 数据表，原有表的数据会全部丢失。

② **create-drop**: 表示创建，只不过在系统关闭前执行一下 **drop**。

③ **update**: 这个操作启动的时候会去检查 **schema** 是否一致，如果不一致会做 **schema** 更新。

上述三种做法会影响到原有的数据库，所以请谨慎使用。

④ **validate**: 这也是本案例中的取值，启动时验证现有 **schema** 与你配置的 **Hibernate** 是否一致，如果不一致就抛出异常，并不做更新。

此外，在第 20 行里，通过 **mapping** 来引入数据表和类文件的映射文件 `UserInfo.hbm.xml`。

```
<mapping resource="Model/UserInfo.hbm.xml" />
```

步骤四 编写对应于 `UserInfo` 数据表的 `java Model` 文件 `UserInfo.java`。

```
1 package Model;
2 public class UserInfo {
3     //这些属性是和数据表里的字段是一一对应的
4     private int userID;
5     private String userName;
6     private String pwd;
7     private String userType;
8     public int getUserID() {
9         return userID;
10    }
11    public void setUserID(int userID) {
12        this.userID = userID;
13    }
14    //...省略针对其他属性的getter和setter方法
15 }
```

在这个文件的第 4 行到第 7 行里，包含了一些和数据库字段对应的属性，并包含着针对这些属性的 **getter** 和 **setter** 方法。一般把这些和数据表对应的类叫做 **Model** 类，或者叫实体类。

但是在运行过程中，系统并不知道 `UserInfo` 数据表是和 `UserInfo.java` 相对应的，更不知道数据字段和 `UserInfo.java` 类中字段的对应关系，所以还需配置映射（**mapping**）文件来向系统说明，这个映射文件就是刚才大家所看到的 `UserInfo.hbm.xml`。

步骤五 编写数据表和 `Java Model` 类之间的对应关系文件 `userInfo.hbm.xml`。

```
1 <?xml version="1.0"?>
```



```

2  <!DOCTYPE hibernate-mapping PUBLIC
3  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
5  <hibernate-mapping package="Model">
6  <class name="UserInfo" table="UserInfo">
7    <id name="userID" type="int">
8      <column name="UserID" />
9    </id>
10   <property name="userName" type="java.lang.String">
11     <column name="username" length="20" not-null="true" unique="true" />
12   </property>
13   <property name="pwd" type="java.lang.String">
14     <column name="pwd" length="20"/>
15   </property>
16   <property name="userType" type="java.lang.String">
17     <column name="usertype" length="20"/>
18   </property>
19 </class>
20 </hibernate-mapping>

```

在第 6 行中，定义了 `UserInfo` 这个 Java 类来对应数据库里的 `UserInfo` 这个表。随后在第 7 行到第 9 行里，通过 `id` 来说明 `UserInfo` 这个 Java 类的 `int` 类型的 `userID` 字段对应于 `UserInfo` 表里的 `UserID` 列（有些像绕口令，但请大家一定要，搞清楚这种对应关系），因为 `UserID` 在表里是主键，所以这里用到的是 `id` 这个元素。

再来一遍绕口令，让大家加深印象。在第 10 行到第 12 行里，是通过 `property` 这个元素，来说明 `UserInfo` 类里 `String` 类型的 `userName` 这个属性对应于 `UserInfo` 表里的长度是 20 的非空的 `username` 字段。同样从第 13 行到第 18 行指定了 `UserInfo` 类里的另外两个属性和数据表里另外两个字段的对应关系。

定义了 `Model` 类 `UserInfo`，而且也写明了数据表和 `Model` 之间的对应关系。在实际应用中，操作的是 `UserInfo` 类，而不是 `MySQL` 数据库里的 `UserInfo` 数据表。这也是 `ORM` 里常规的做法，在 `HibernateMain.java` 这个主程序里，用 `ORM` 的编程模式，首先向 `UserInfo` 表里插入了一条记录，随后打印所有的记录。由于代码比较长，因此这里省略了 `package` 和 `import` 语句。

```

1  public class HibernateMain {
2  public static void main(String[] args) {
3    // 通过第 3 行到第 12 行代码获得 sessionFactory 对象
4    StandardServiceRegistry registry = null;
5    SessionFactory sessionFactory = null;
6    try{

```

```

7         registry = new
StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
8         sessionFactory = new
MetadataSources(registry).buildMetadata().buildSessionFactory();
9         }catch(Exception ex){
10             ex.printStackTrace();
11             StandardServiceRegistryBuilder.destroy(registry);
12         }
13 //创建 session，并向 UserInfo 表里插入一条数据
14         Session session = sessionFactory.openSession();
15         Transaction tx = session.beginTransaction();
16         UserInfo user = new UserInfo();
17         user .setUserID(3);
18         user .setUserName("Peter");
19         user .setPwd("456");
20         user .setUserType("Vip");
21         try{
22 session.save(user);
23         tx.commit();
24         }
25         catch(Exception e)
26         {
27 e.printStackTrace();
28 tx.rollback();
29         }
30         //通过 HQL 语句，遍历 UserInfo 表的所有记录
31         List<UserInfo> users = new ArrayList<UserInfo>();
32         String hqlstr = "from UserInfo";
33         Query query = session.createQuery(hqlstr);
34         users = query.list();
35         System.out.println("users size is:" + users.size());
36         System.out.println("UserName" + "\t" + "UserType");
37         for(UserInfo u : users){
38             System.out.println(u.getUserName() + "\t" + u.getUserType());
39         }
40         //最后需要关闭 session 和 sessionFactory 对象
41         session.close();
42         sessionFactory.close();
43     }
44 }

```

从第 3 行到第 12 行里，通过一些例行公事的代码，根据第 7 行里指定的配置文件，创建一个 SessionFactory 对象，并在第 14 行通过 SessionFactory 创建一个 Session 对象。

在前文里已经提到，在 Hibernate 里，Session 对象里包含着一些操作数据库的方法。从第 15 行到 29 行里，首先是在第 15 行创建了一个 UserInfo 类，并通过一些 setter 方法设置了该类的一些属性值，随后通过第 22 行和第 23 行的 session.save 和 tx.commit 来完成数据的插入动作。

这里是通过事务来完成插入的，具体而言，在第 22 行完成操作后（save）执行 commit 动作进行提交，一旦出现问题，会在第 28 行的 catch 异常处理中通过 rollback 操作进行数据的回滚。

从上述代码里能看到，操作的是 UserInfo 这个 Model 类，比如这里是创建 UserInfo 后并设置属性值，随后在必要的时候通过 save 把 Model 类更新到数据库里。在这种基于 ORM 的做法里，程序员接触到的是 Model 类而不是数据表，从而可以更关注与 UserInfo 类相关的业务操作（比如新增了一个用户），而没必要关心这些操作是如何落实到数据库里的。

在第 30 行到第 39 行里，通过 HQL 语句拿到 UserInfo 表里的所有记录并打印。具体的做法是，在第 31 行里定义了一个 UserInfo 类的 ArrayList 用来接收所有的记录，随后通过第 33 行的 session.createQuery 来执行 from UserInfo 这个 HQL 语句。HQL 和 SQL 很相似，这里是拿所有记录的意思，随后在第 34 行里，通过 Query 对象，把记录映射到 UserInfo 这个 list 里。

这里其实有个隐藏的操作，通过 Session 对象执行 HQL 语句并得到数据库结果后，系统会根据 UserInfo.hbm.xml 这个映射文件里的定义（比如哪个表对应哪个 Java Model 文件，表里哪个字段对应 Model 文件里哪个属性），把数据库中 UserInfo 这个表里的数据映射到 UserInfo 这个列表里。

之后的第 35 行到第 39 行的打印语句没什么可说，但请大家务必注意，用好之后一定请关闭 Session 和 SessionFactory，如第 41 行和第 42 行所示，否则这个基于 Hibernate 的进程有可能无法终止。

7.1.2 通过 Hibernate 的注解方式实现 ORM

在 Hibernate 的高级版本里，同样可以通过注解来实现 ORM。下面演示一下通过注解实现 Hibernate 的做法。

代码位置	视频位置
code/第 7 章/HibernateAnnotationDemo	视频/第 7 章/通过注解实现 Hibernate

这个案例和 HibernateDemo 的代码很相似，但有如下修改点。

修改点 1，需额外多导入一些 jar 包，其中，jboss-annotations-api_1.2_spec-1.0.0.Final.jar

用于支持注解，hibernate-entitymanager-5.0.9.Final.jar 用来支持持久化映射。

修改点 2，在 hibernate.cfg.xml 里，连接数据库的配置不变，但需要修改描述 Model 类位置的 mapping 元素。

用配置文件时的写法如下，其中指定了用于描述映射关系的 hbm.xml 文件。

```
<mapping resource="Model/UserInfo.hbm.xml" />
```

用注解时，写法如下，直接指向具体的 UserInfo 这个类，注意，这里需要写包名。

```
<mapping class="Model.UserInfo" />
```

修改点 3，将数据库表和 Java Model 类的映射关系写到了 UserInfo.java 里，所以这个文件需要修改，代码如下。

```
1  package Model;
2  import javax.persistence.Column;
3  import javax.persistence.Entity;
4  import javax.persistence.Id;
5  import javax.persistence.Table;
6  import javax.persistence.Transient;
7  @Entity
8  @Table(name="userinfo")
9  public class UserInfo {
10     @Id
11     @Column(name = "UserID")
12     private int userID;
13     @Column(name = "username")
14     private String userName;
15     @Column(name = "pwd")
16     private String pwd;
17     @Column(name = "usertype")
18     private String userType;
19     @Transient
20     private int age;
21     public int getUserID() {
22         return userID;
23     }
24     public void setUserID(int userID) {
25         this.userID = userID;
26     }
27     //省略针对其他属性的 getter 和 setter 方法
28 }
```

这里给出了注解的一般用法，请参考表 7.3 来了解一下各注解的含义。

表 7.3 常用注解的含义

行号	注解名	含义
第 7 行和第 8 行	@Entity 和 @Table	描述这个类是和哪个数据表关联，在这个文件里，是说明 UserInfo 这个 Java Model 类是和 MySQL 里 UserInfo 表关联
第 10 行	@Id	描述主键，这里是作用在 userID 这个属性上，所以是说明该属性所对应的字段是主键
第 13 行	@Column	这里的用法是@Column(name = "username")，是作用在第 14 行的 userName 属性上，这是说明数据库里@Column 所指定的 username 这一列和第 14 行的 userName 属性相关联
第 19 行	@Transient	这个属性是作用在第 20 行的 age 属性上，表明这个属性和数据库里字段无关，该属性不会映射到数据表里。 我们也发现，在 MySQL 的 UserInfo 表里没这个字段，所以一定要在 age 属性前加上@Transient，否则 Hibernate 默认是要映射的，于是会出现映射错误的异常

同时，无须修改 HibernateMain.java 里的代码。

一般来说，通过配置文件的开发方式比较利于维护，但配置文件可能会比较多，比如一个项目里要操作多个表就需要有多个 mapping 文件。

相比之下，注解的方式比较灵活，写起来很方便，而且不需要编写额外的配置文件。

但如果数据库部分有改动（比如数据库字段变动了或者甚至从 Hibernate 切换到 MyBatis），那么可能就需要直接更改 Java 代码，毕竟更改配置文件的代价要小于更改 Java 代码。所以这两种方式各有适用的场合，没有优劣之分，大家可以根据实际需求，选用合适的方式。

7.1.3 Hibernate 里生成主键的方式

在上文中创建的 UserInfo 表虽然有主键，但这个主键不是自动生成的。实际开发中，一般会在数据库里创建一个整型自增长的主键，以便提升数据库的性能，做过 Hibernate 的程序员一般都了解这个技术，所以我们经常会通过提问“主键生成方式”这方面的问题来核实面试者是否真的用过 Hibernate。下面来看下在 Hibernate 里如何管理这类自增长的主键。

代码位置	视频位置
code/第 7 章/HibernatePKDemo	视频/第 7 章/Hibernate 里生成主键的方式

步骤一 在数据库里创建 Account 表，表结构如表 7.4 所示。

表 7.4 Account 表结构

字段名	类型	含义
ID int		自增长的主键
Name var	char	账户拥有者的名字
Bank var	char	所属银行
balance float		余额

步骤二 创建名为 HibernatePKDemo 的 Java 项目，并导入必要的 jar 包。

步骤三 编写 hibernate.cfg.xml 这个配置文件，这里和 7.1.1 部分很相似，用相同的账户同样是连接 hibernatechart 数据库，只不过这里是通过 Account.hbm.xml 这个文件来指定数据表和 Java 类的对应关系。

```
1 <!-- 添加实体类的映射文件-->
2 <mapping resource="Model/Account.hbm.xml" />
```

步骤四 编写描述对应关系的 Account.java。

```
1 <?xml version="1.0"?>
2 <!DOCTYPE hibernate-mapping PUBLIC
3 "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
5 <hibernate-mapping package="Model">
6   <class name="Account" table="Account">
7     <id name="ID" column="ID" type="int">
8       <generator class="increment" />
9     </id>
10    <property name="name" type="java.lang.String">
11      <column name="Name" length="20" not-null="true"/>
12    </pr operty>
13    <property name="bank" type="java.lang.String">
14      <column name="Bank" length="20"></column>
15    </pr operty>
16    <pro perty name="balance" type="float">
17      <column name="Balance"></column>
18    </pr operty>
19  </class>
20 </hibernate-mapping>
```

从第 7 行到第 9 行，用了 id 属性来描述主键，在第 8 行里，通过 generator 来指定生成主键的方式是 increment。除了这个值以外，常用的生成主键的方式如表 7.5 所示。

表 7.5 H ibernate 里生成主键的常用方式

用法示例	说明
<generator class="assigned" />	主键由程序负责生成，在 save 方法前必须由其他程序指定主键，

	<p>Hibernate 不负责维护主键生成。可以跨数据库，在存储对象前，必须要主键赋值。这种做法一般不推荐</p>
<pre><generator class="increment" /></pre>	<p>由 Hibernate 从数据库中取出主键的最大值，每个 Session 只取 1 次，以该值为基础，每次增量为 1。可以跨数据库</p> <p>不过在多个进程同时并发访问数据库时，可能会发生插入主键冲突的错误</p> <p>该做法不适合多进程并发更新数据库的场景</p>
<pre><id name="id" column="id"> <generator class="hilo"> <param name="table">hibernate_hilo</param> <param name="column">next_hi</param> <param name="max_lo">100</param> </generator> </id></pre>	<p>hilo 是 Hibernate 中最常用的一种生成方式，生成主键的过程如下：</p> <ol style="list-style-type: none"> ① 读取 hibernate_unique_key 表中 next_hi 字段的值，以获得 hi 值 ② 从 0 到 max_lo 循环取值，差值为 1。当值为 max_lo 时，重新获取 hi 值，然后 lo 值继续从 0 到 max_lo 循环，依次获得 lo 值 ③ 根据公式 $hi * (max_lo + 1) + lo$ 计算生成主键值
<pre><generator class="sequence"> <param name="sequence">hibernate_id</param> </generator> <param name="sequence">hibernate_id</param></pre>	<p>采用数据库提供的 sequence 机制生成主键，不过需要数据库支持 sequence，比如 Oracle 等，但 MySQL 不支持</p> <p>Hibernate 生成主键时，查找 sequence 并赋给主键值，主键值由数据库生成，Hibernate 不负责维护，使用时必须先创建一个 sequence</p>
<pre><generator class="identity" /></pre>	<p>identity 是由数据库自己生成的，但这个主键必须设置为自增长，使用 identity 的前提条件是底层数据库支持自动增长字段类型，如 MySQL 等，Oracle 这类没有自增字段的则不支持</p>
<pre><generator class="native"/></pre>	<p>native 由 Hibernate 根据使用的数据库自行判断采用 identity、hilo、sequence 其中一种作为主键生成方式，灵活性很强。如果能支持 identity 则使用 identity，如果支持 sequence 则使用 sequence</p>
<pre><generator class="uuid"/></pre>	<p>Hibernate 在保存对象时，生成一个 UUID 字符串作为主键，保证了唯一性，但其并无任何业务逻辑意义，只能作为主键，缺点是长度较大，是 32 位</p>

最后，在 HibernateMain 里调用的代码如下。

```

1  public class HibernateMain {
2      public static void main(String[] args) {
3          StandardServiceRegistry registry = null;
4          SessionFactory sessionFactory = null;
5          try{
6              registry = new
StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
7              sessionFactory = new
MetadataSources(registry).buildMetadata().buildSessionFactory();
8          }catch(Exception ex){

```

```

9         ex.printStackTrace();
10        StandardServiceRegistryBuilder.destroy(registry);
11    }
12    Session session = sessionFactory.openSession();
13    Transaction tx = session.beginTransaction();
14    Account acc = new Account();
15    acc.setName("Tom");
16    acc.setBank("CitiBank");
17    acc.setBalance(100);
18    try{
19 session.save(acc);
20        tx.commit();
21    }
22    catch(Exception e)
23    {
24 e.printStackTrace();
25 tx.rollback();
26    }
27    session.close();
28 sessionFactory.close();
29 }
30 }

```

从第 13 行到第 26 行里通过一个事务插入 Account 对象，这里请注意，并没有设置主键。运行程序两次后，能看到插入了两条 ID 主键不同的数据。

导入向导 导出向导 筛选向导			
ID	Name	Bank	balance
1	Tom	CitiBank	100
2	Tom	CitiBank	100

如果用注解的编程方式，则可以通过在 Model 类里，用如下代码来设置主键的生成方式。

```

1    @Id //说明该属性是主键
2    //设置主键生成的方式，这里可以取适当的值
3    @GeneratedValue(strategy = IDENTITY)
4    @Column(name = "ID") //说明是和数据表里哪个字段对应
5    private int ID;

```

7.2 Session 对象在项目里的用法

通过上述案例可看到 Hibernate 能通过配置文件或注解映射的方式，把数据表里的数据映

射成 Java 类（一般是 Model 类），所以程序员能“忽略”数据表，可以直接操作 Java 对象，而不用像 JDBC 那样还要负责把业务结果设置到数据库里。

在这个流程中，Session 对象发挥了重要的作用，它不仅提供了针对数据库增删改查的操作接口，而且还承担着缓存的作用。通过 Session，Hibernate 能把从数据库里读到的数据缓存起来，从而提升效率。

7.2.1 Session 对象中的重要方法

Session 对象里包含了针对数据库操作的方法，在上文中，用 save 把一个 Java Model 对象插入数据库，但 Session 对象的方法不仅于此，下面将通过案例来讲述该对象在项目里的一般用法。

代码位置	视频位置
code/第 7 章/HibernateSessionDemo	视频/第 7 章/Hibernate 里 Session 对象的讲解

在 HibernatePKDemo 的基础上，同样是用 MySQL 里的 Account 表，而 Hibernate.cfg.xml 配置文件、UserInfo 类和 UserInfo.hbm.xml 这三个文件均维持原样。在 HibernateMain 类里，其中用到了 Session 类的诸多方法实现了针对 Account 表的增删改查操作。

```
1 //省略 package 和 import 的语句
2 public class HibernateMain {
3     public static void main(String[] args) {
4         StandardServiceRegistry registry = null;
5         SessionFactory sessionFactory = null;
6         try{
7             registry = new
StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
8             sessionFactory = new
MetadataSources(registry).buildMetadata().buildSessionFactory();
9         }catch(Exception ex){
10             ex.printStackTrace();
11             StandardServiceRegistryBuilder.destroy(registry);
12         }
13         Session session = sessionFactory.openSession();
14         Transaction tx = session.beginTransaction();
15         //得到 Account 表里所有记录
16         List<Account> accountList = new ArrayList<Account>();
17         String hqlstr = "from Account";
18         Query query = session.createQuery(hqlstr);
19         accountList = query.list();
20         //用 load 方式
```

```

21     Syst     em.out.println("load...");
22         for(int i = 0;i<accountList.size();i++)
23         {
24             Account one = (Account)session.load(Account.class, accountList.get(i).getID());
25             System.out.println(one.getName() + "\t" + one.getBank() + "\t" +
26                 one.getBalance());
27         }
28         //用 get 方式
29     Syst     em.out.println("get...");
30         for(int i = 0;i<accountList.size();i++)
31         {
32             Account one = (Account)session.get(Account.class, accountList.get(i).getID());
33             System.out.println(one.getName() + "\t" + one.getBank() + "\t" +
34                 one.getBalance());
35         }
36         //delete account one by one
37         for(int i = 0;i<accountList.size();i++)
38         {
39             int ID = accountList.get(i).getID();
40             System.out.println("delete the Account with ID=" + ID);
41             session.delete(accountList.get(i));
42         }
43         //插入两个 Account 对象
44         Account accForJava = new Account();
45         accForJava.setName("Java");
46         accForJava.setBank("JavaBank");
47         accForJava.setBalance(100);
48         session.save(accForJava);
49         Account accForORM = new Account();
50         accForORM.setName("ORM");
51         accForORM.setBank("ORMBank");
52         accForORM.setBalance(1000);
53         session.save(accForORM);
54         tx.commit();
55         //再开启另外一个事务更新数据
56         tx = session.beginTransaction();
57         accountList = query.list();
58         //用 load 方式拿到数据，并给每个账户加 200 元
59         for(int i = 0;i<accountList.size();i++)
60         {
61             Account one = (Account)session.load(Account.class, accountList.get(i).getID());
62             one.setBalance(one.getBalance() + 200);
63             session.update(one);
64         }

```

```
63         tx.commit();
64         //关闭 Session 和 SessionFactory 对象
65         session.close();
66         sessionFactory.close();
67     }
68 }
```

从第 4 行到第 13 行,用类似的代码获得了 Session 对象,在第 17 行,定义了 from Account 这个 HQL 语句。

从第 20 行到第 26 行里,演示了用 Session 的 load 方法获得对象的方式。请大家关注第 24 行代码,load 的具体用法是,用第 1 个参数指定 load 对象的类型,用第 2 个参数指定需要 load 对象的主键,这个方法将返回一个由第 1 个参数指定的 Account 对象。

```
Account one=(Account) session.load(Account.class,accountList.get(i).getID())
```

同样,在第 27 行到第 33 行里,演示了用 get 方法获得对象的方式,它的用法和 load 很相似,同样用第 1 个参数指定对象类型,用第 2 个参数指定需要 get 的主键值。

```
Account one=(Account) session.get(Account.class,accountList.get(i).getID())
```

从第 34 行到第 40 行,我们遍历并且删除了 Account 表里的所有记录,关键代码在第 39 行,通过 session.delete(accountList.get(i))来删除,其中 delete 方法的参数是待删除的 Account 对象。

完成删除后,从第 41 行到第 52 行,通过 save 方法插入两条 Account 记录。

插入后,从第 57 行到第 62 行之间的 for 循环里,通过 load 方法得到刚插入的两条记录,并给各账户的 balance 加 200 元,随后通过第 61 行的 session.update(one);把本地的 Account 类更新到数据表里。Update 方法的参数是待更新的 Account 类。

7.2.2 Session 对象中的 load 和 get 方法的差别

在上面的例子中演示了 Session 对象里常用的增删改查的方法。在查的时候,用到了 get 和 load 两种方式,它们的区别是,执行 load 方法时,Hibernate 认为该数据一定存在,可以使用代理来延迟加载,如果在使用过程中发现了问题,则只能抛异常。而对于 get 方法来说,Hibernate 一定要获取到真实的数据,否则返回 null。

在上述代码里的第 21 行和第 22 行之间加入如下的 load 测试代码。

```
20 //用 load 方式
21 System.out.println("load...");//之后加入 load 测试代码
22 Account loadNotExistAcc = (Account) session.load(Account.class, 100);
```

```

23  if(loadNotExistAcc == null)
24  {
25      System.out.println("loadNotExistAcc is null");
26  }
27  else
28  {
29      loadNotExistAcc.getBank();
30  }//之后是正常的 for 循环
31  for(int i = 0;i<accountList.size();i++)

```

在第22行里,我们试图load一个id为100的不存在的Account。根据load的特性,Hibernate认为该对象一定存在,所以不会走25行这个打印语句,而是到第29行,当要用到这个对象的getBank方法时,才会加载这个对象,由于ID为100的Account不存在,所以针对第29行会报异常。

在第28行和第29行之间加入针对get的测试代码。

```

27  //用get方式
28  System.out.println("get...");//如下是针对get的测试代码
29  Account getNotExistAcc = (Account)session.get(Account.class, 100);
30  if(getNotExistAcc == null)
31  {
32      System.out.println("getNotExistAcc is null");
33  }
34  else
35  {
36      getNotExistAcc.getBank();
37  }//以后是正常的 for 循环
38  for(int i = 0;i<accountList.size();i++)

```

针对get,Hibernate会立即加载(而不是像load一样会延迟加载),所以会打印第32行的语句。

在实际使用中,如果能确定对应的数据在表里一定存在,那么可以使用load,反之就用get。get比load安全,但是安全是要牺牲性能作为代价的。

7.2.3 Session 缓存与三种对象状态

调用Session对象提供的方法操作数据库,是为了提升访问数据库的效率,Session对象会开辟一块内存空间作为缓存,在这个缓存里存放着一些和数据表相关联的Java Model对象。

Session的缓存是内置的,也叫作Hibernate的第一级缓存。常规情况下,一级缓存是由Hibernate维护的,无须人工干预。

和缓存相对应，在 Java 端的 Model 对象有临时状态、持久化状态和游离状态这三类。

刚创建出来的对象，比如上文里刚创建出来的 Account 或 UserInfo 对象，此时还没更新到数据表里，也不处于 Session 缓存里，这种对象的状态叫作临时（transient）状态，处于临时状态的对象叫临时对象。

如果一个对象已经通过 save 方法被保存到数据库里，那么该对象在程序结束后也不会消失，所以该对象已经被持久化，或者说，该对象处于持久化状态(persistent)，一般来说，处于持久化状态的对象同时会被加入到 Session 缓存中。

创建出来一个临时对象 Account，并且通过 save 方法把它持久化到数据库后，此时的 Account 对象是处于持久化状态，但此时如果我们通过 close 方法关闭了 Session，虽然该对象和数据库里的一条记录有映射关系，但此时 Session 都关了，所以该对象和数据表里对应记录的联系已经中断，此时该对象处于“游离”（Detached）状态。

了解这些概念后，再来深入地了解 Session 对象的其他重要方法。

代码位置	视频位置
code/第 7 章/HibernateSessionStatusDemo	视频/第 7 章/Hibernate 三种 Session 对象

把 HibernateSessionDemo 项目的 HibernateMain 类改写一下，加入一些重要方法，这个项目还是用数据库的 Account 表。这个类的代码比较长，我们分段说明。

```
1 //省略必要的 package 和 import 代码
2 public class HibernateMain {
3     public static void main(String[] args) {
4         StandardServiceRegistry registry = null;
5         SessionFactory sessionFactory = null;
6         try{
7             registry = new
            StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
8             sessionFactory = new
            MetadataSources(registry).buildMetadata().buildSessionFactory();
9         }catch(Exception ex){
10             ex.printStackTrace();
11             StandardServiceRegistryBuilder.destroy(registry);
12         }
```

至此，我们创建了 SessionFactory，后面将继续创建 Session 对象，并调用其中的一些重要方法。

```
13         Session session = sessionFactory.openSession();
14         //演示 update 和 merge 以及 saveOrUpdate 的差别
```

```

15      Transaction tx = session.beginTransaction();
16      Account peterAcc = new Account();
17      peterAcc.setName("Peter");
18      peterAcc.setBank("PeterBank");
19      peterAcc.setBalance(200);
20      //session.update(peterAcc); error
21      session.merge(peterAcc); //ok
22      session.saveOrUpdate(peterAcc); //ok
23      tx.commit();

```

在第 13 行到第 23 行里，演示了 `merge`、`update` 和 `saveOrUpdate` 这三个方法。在执行前，清空了 MySQL 里的 `Account` 表，大家如果打开第 20 的注释并运行，就会发现有异常，原因是 `update` 只能作用在持久化对象上，而刚创建的 `peterAcc` 对象是临时对象。

但第 21 行的 `merge` 方法可以作用在临时对象上，它的作用是把这个 `peterAcc` 对象插入 `Account` 表里。

第 22 行的 `saveOrUpdate` 方法就比较灵活，如果该参数还没有被持久化（比如刚创建出来的对象），对其调用 `save` 方法，否则通过 `update` 方法执行更新操作。

这段代码执行后，大家在 `Account` 表里能看到有两条相似的记录，只不过它们的主键 ID 不一样。

```

24      //演示 save 和 persist 的差别
25      tx = session.beginTransaction();
26      Account mikeAcc = new Account();
27      mikeAcc.setName("Mike");
28      mikeAcc.setBank("MikeBank");
29      mikeAcc.setBalance(300.0f);
30      System.out.println("the new inserted id is:" + session.save(mikeAcc));
31      //persist 方法没有返回类型
32      session.persist(mikeAcc);
33      tx.commit();

```

在第 24 行到第 33 行里，演示了 `save` 和 `persist` 方法的差别。`save` 方法返回插入后的主键值，由于 `Account` 表里 ID 是自增长的主键，所以没有设置本地 `mikeAcc` 对象的 id，在第 30 行里，完成 `save` 后，该方法是会返回刚插入记录的 ID 值。

第 32 行里，调用了 `persist` 方法，它没有返回值，这点和 `save` 方法不同。而且，`persist` 方法并“不保证”标识符(这里是已经被插入的 `Account` 记录的主键 ID)被立刻加入到持久化实例中，标识符的加入动作可能被推迟到用 `flush` 方法的时候。

```

34      //演示 flush 的用法

```

```
35     Account flushAcc = new Account();
36     flushAcc.setName("Flush");
37     flushAcc.setBank("FlushBank");
38     flushAcc.setBalance(1000);
39     session.save(flushAcc);
40     session.flush();
```

和之前的代码不同，这里没有用到事务，没有 `tx.commit` 代码。第 40 行里 `flush` 方法的作用是清理缓存，在清理之前会强制把 `Session` 缓存里的修改同步到数据库里。

所以在代码里，如果没有必要用事务，则必须在代码的最后（或必要的位置）调用 `flush` 方法，否则在对本地的 `Model` 对象修改时无法更新到数据库里。

```
41     //演示 evict 和 clear
42     Account evictAcc = new Account();
43     evictAcc.setName("Evict");
44     evictAcc.setBank("EvictBank");
45     evictAcc.setBalance(1000);
46     session.save(evictAcc);
47     //session.evict(evictAcc); error
48     session.clear();
49     session.flush();
```

`evict` 方法的作用是在 `Session` 缓存里清除指定的对象。

在第 46 行通过 `save` 语句把本地的 `evictAcc` 对象变成“持久态”，在未执行第 49 行的 `flush` 同步之前，如果打开第 47 行的注释，用 `evict` 方法清除待 `save` 的 `evictAcc` 对象，同时注释掉第 48 行的 `clear`，那么运行的时候会报如下异常。

```
Exception in thread "main" org.hibernate.AssertionFailure: possible non-threadsafe
access to session
```

原因是 `Hibernate` 系统认为 `save` 后的结果一定会通过事务或者 `flush` 等方式同步到数据表里，所以不允许在 `save` 之后同步之前删除缓存中待 `save` 的对象。

相比之下，`clear` 方法的作用是清除 `Session` 缓存里的全部对象，但操作中的对象除外。这里已经被调用 `save` 方法的 `evictAcc` 对象被认定为“处于操作中”，所以不会把它从 `Session` 里删除。

```
50     //关闭必要的对象
51     session.close();
52     sessionFactory.close();
53 }
54 }
```

最后关闭两个必要的对象。

一般的初级程序员会有个错误的想法，认为 Session 的 save（或者 update 等）方法会立即执行，所以会在提交事务或者调用 flush 之前过早地把待 save（或 update）的对象从 Session 里删除，就如刚才调用 evict 方法一样。

我们经常见到程序员犯这种错误，这里报异常还算好，至少知道出错了。在有些错误的代码中，比如在第 100 行程序员就已经调用 save 了，但却在第 150 行左右才提交事务或者调用 flush，此时一旦在第 100 行到第 150 行之间对 save 的对象进行了修改，那么最终插入数据库的就不是我们期望的值了，这种错误由于不会报异常，隐蔽性很强，很难发现，所以请大家特别留意下 save 之后 flush（或者提交事务）之前的代码，别出现误修改的情况。

7.2.4 FlushMode 与清空缓存的时间点

从上文中能看到，Session 会把操作数据库的结果缓存起来以便提升效率，为了更好地管理缓存，可以通过 Session 对象的 setFlushMode 方法来设置清空缓存的时间点。它的用法是：session.setFlushMode(FlushMode.AUTO);。

其中，FlushMode 的取值有 5 种，通过表 7.6 来归纳下各种取值对清空缓存的影响。

表 7.6 H ibernate 里生成主键的常用方式

清理缓存的模式	调用查询方法	调用 commit 方法	调用 flush 方法
FlushMode.AUTO（默认值）	清理缓存	清理缓存	清理缓存
FlushMode.COMMIT	不清理	清理缓存	清理缓存
FlushMode.NEVER（已经废弃）	不清理	不清理	清理
FlushMode.MANUAL	不清理	不清理	清理
FlushMode.ALWAYS	清理缓存	清理缓存	清理缓存

针对上表做如下的说明。

第一，调用查询方法的含义是比如通过 query 执行 HQL 的查询语句。

第二，比如通过 Session 的 load 方法，从数据库得到 ID 为 5 的 Account 对象，该对象是存在缓存里的。假设当前的值是 AUTO，那么在执行查询方法时，Session 会把 ID 为 5 这个对象从缓存里清空，再到数据库里拿一次再放到缓存里。而在调用 commit 和 flush 方法时，也会清空缓存，然后把对应的 Java Account 这个 Model 类的值设置到缓存里。

第三，如果不指定，默认的值是 AUTO，大家可以通过如下语句来验证。

```
//之前不做任何设置，从如下语句我们可以看到默认将返回 AUTO
```



```
System.out.println("Flush Mode is: " + session.getFlushMode());
```

第四，从表 7.6 来看，AUTO 和 ALWAYS 清空缓存的时间点是一样的，它们的差别是，用 ALWAYS 的方式时，当 Hibernate 缓存中的对象发生改动后，会无条件地被标记为脏数据；而当设置为 AUTO 时，Hibernate 在查询时会判断缓存中的数据是否为脏数据，是则更新，否则不操作。但 ALWAYS 是无条件地直接刷新。所以 AUTO 比 ALWAYS 要高效。

给大家的建议是，如果没有特别的需求，可以用默认的 AUTO，当然用 MANUAL 的方式可能性能更好些，但在使用前请谨慎，确保不清理缓存对你的项目没有影响。

之所以讲述这个知识点，是因为它和性能调试有关，而性能调优的能力是高级程序员必备的。

当大家了解这个知识点后，如果被问到“如何提升系统性能”这方面的问题时，就可以胸有成竹地讲述“通过合理设置 FlushMode 从而提升性能”的方式。

7.3 在 Hibernate 里执行复杂的查询

之前的案例中，是通过简单的 HQL 语句，用 from xxx 的形式返回数据表里的所有数据，这种做法并不能满足实际项目的需求。实际项目的查询语句可能较为复杂，可能不仅包含 where 条件，而且可能还得包含关联、group by 和 having 等功能。

在本部分，将用 Hibernate 提供的 HQL 语句实现各种针对数据库查询的需求，还将结合使用 Query 和 Criteria 这两个对象，用一种比较直观的方式得到查询结果。

7.3.1 where、groupby 和 having

我们经常会在查询中用到这三个关键字，下面通过 HibernateHQLDemo 这个项目来演示它们在 HQL 中的用法。

代码位置	视频位置
code/第 7 章/HibernateHQLDemo	视频/第 7 章/HQL 的用法

该项目是从 HibernateSessionDemo 改写而来，这里只改动了 HibernateMain 这个类，代码如下。

```
1 //省略必要的package和import语句
2 public class HibernateMain {
```

```

3 public static void main(String[] args) {
4     // 依照老办法得到 SessionFactory 对象
5     StandardServiceRegistry registry = null;
6     SessionFactory sessionFactory = null;
7     try{
8         registry = new
9         StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
10        sessionFactory = new
11        MetadataSources(registry).buildMetadata().buildSessionFactory();
12    }catch(Exception ex){
13        ex.printStackTrace();
14        StandardServiceRegistryBuilder.destroy(registry);
15    }
16    //得到 Session
17    Session session = sessionFactory.openSession();
18    //以 Object 数组的形式返回字段
19    List<Object[]> accountList = (List<Object[]>)session.createQuery("select
20    name,bank,balance from Account where balance < :highBalance and
21    balance > :lowBalance").setParameter("highBalance",800.0f).setParameter("lowB
22    alance", 50.0f).list();
23    for(int i=0;i<accountList.size();i++)
24    {
25        Object[] one = accountList.get(i);
26        System.out.println(one[0] + "\t" + one[1] + "\t" + one[2]);
27    }
28 }

```

在第 17 行 `createQuery` 方法的参数里，传入了一个 HQL 语句，它和 SQL 的形式基本相似，在 `where` 条件里，用带冒号的查询参数，比如 `:highBalance`，通过 `setParameter` 方法，给指定的参数设置具体的值，最终，通过 `list` 返回查询结果。

`setParameter` 有两个参数，分别对应参数的名字和参数值。比如 HQL 里通过冒号指定了参数名是 `highBalance`，那么通过（“`highBalance`”，`500.0f`）这两个参数来指定将要在 HQL 里 `:highBalance` 这个位置传入 `500.0f` 这个参数。

请大家尤其注意，通过第 17 行里 `list` 方法返回的对象是 `List<Object>`，而不是 `List<Account>`，所以在 `for` 循环里，我们是通过 `one[0]` 这样的形式来访问的。

在 HQL 的 `where` 条件从句里，也可以使用类似 SQL 里的“`like`、`is null`”以及“`!=`”这种操作符号，这些用法和 SQL 里很相似，所以就不再举例了。

```

23 //2 带 group by 的形式
24 List<Object[]> groupbyDemoList = (List<Object[]>)session.createQuery("select
25 name,bank,count(*) from Account group by name,bank having count(*)>1").list();
26 for(int i=0;i<groupbyDemoList.size();i++)

```

```

26 {
27     Object[] one = groupbyDemoList.get(i);
28     System.out.println(one[0] + "\t" + one[1] + "\t" + one[2]);
29 }
30 //关闭 session 和 sessionFactory
31 session.close();
32 sessionFactory.close();
33 }
34 }

```

在 Account 表里，同一个人同一银行应该只有一个账户，通过 groupby 对 name 和 bank 分组，并通过 having 查询每个分组里个数大于 1 的值，以此来查询是否有重复数据。

在第 24 行里，演示了 groupby 和 having 的用法，大家可以发现 HQL 和 SQL 的用法也是非常相似的，同样这里返回的也是 List<Object>对象。

7.3.2 表关联查询和子查询

项目里经常会遇到表关联查询和通过 in 等关键字进行子查询的情况，这里用 HQL 语言来实现。

代码位置	视频位置
code/第 7 章/HibernateHQLJoinDemo	视频/第 7 章/通过 HQL 实现关联等复杂查询

这个项目是基于上面 HibernateHQLDemo 项目修改而来，这里将关联 Account 和 UserInfo 两张表，所以需要加入这两个 Model 文件，同时，还需要在 hibernate.cfg.xml 中一并引入这两个 Model 类。

```

<mapping resource="Model/Account.hbm.xml" />
<mapping resource="Model/UserInfo.hbm.xml" />

```

在 HibernateMain 这个类里，放入了关联查询和子查询的代码。

```

1 //省略 package 和 import 代码
2 public class HibernateMain {
3     public static void main(String[] args) {
4         // 依照原来的代码获得 SessionFactory 对象
5         StandardServiceRegistry registry = null;
6         SessionFactory sessionFactory = null;
7         try{
8             registry = new
9             StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
10            sessionFactory = new

```

```

        MetadataSources(registry).buildMetadata().buildSessionFactory();
10    } catch (Exception ex) {
11        ex.printStackTrace();
12        StandardServiceRegistryBuilder.destroy(registry);
13    }
14    //获得 Session 对象
15    Session session = sessionFactory.openSession();
16    //1 关联查询
17    List<Object[]> list = (List<Object[]>)session.createQuery("select
acc.name,acc.bank,acc.balance,u.userType from Account acc, UserInfo u where
acc.ID=u.userID").list();
18    for(int i=0;i<list.size();i++)
19    {
20    Object[] one = list.get(i);
21    System.out.println(one[0] + "\t" + one[1] + "\t" + one[2] + "\t" + one[3]);
22    }

```

在第 17 行中通过 HQL 语句关联了 Account 表和 UserInfo 表,关联条件写在 where 语句里。关联查询的结果还是 List<Object[]>类型,所以从第 18 行到第 22 行里,通过 for 循环用 one[0] 之类的形式输出结果。

```

23    //子查询
24    List<Object[]> groupbyDemoList =
(List<Object[]>)session.createQuery("select name,bank,balance from Account
where ID in (select userID from UserInfo)").list();
25    for(int i=0;i<groupbyDemoList.size();i++)
26    {
27    Object[] one = groupbyDemoList.get(i);
28    System.out.println(one[0] + "\t" + one[1] + "\t" + one[2]);
29    }
30    //关闭必要的对象
31    session.close();
32    sessionFactory.close();
33    }
34    }

```

在第 24 行中用 in 这个子查询语句,返回了 ID 只存在于 UserInfo 表里的数据,并在第 25 到第 29 行里通过 for 循环输出。

7.3.3 通过 SQLQuery 对象执行 SQL 语句

在很多情况下可以通过 HQL 来得到必要的数 据,但在项目的一些特殊需求里,有时还会使用一些比较复杂的 SQL 语句,比如多表之间关联外加 With、groupby having 等,这种情况

下如果我们还是用 HQL，代码会很长，可读性也会降低，有时候甚至根本无法用 HQL 实现，这时我们不得不使用 SQL。

大家知道，HQL 和 SQL 语句夹杂使用会极大地增加项目的维护难度，但完成需求是第一位的，所以接下来看下这种“应急”的情况的处理方式。

HibernateHQLDemo 代码位置	视频位置
code/第 7 章/HibernateHQLSQLDemo	视频/第 7 章/在 Hibernate 里运行 SQL 语句

这个项目是根据 HibernateHQLDemo 项目改编的，只修改其中的 HibernateMain 这个类。

```
1 //省略package和import代码
2 public class HibernateMain {
3     public static void main(String[] args) {
4         StandardServiceRegistry registry = null;
5         SessionFactory sessionFactory = null;
6         try{
7             registry = new
8             StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
9             sessionFactory = new
10             MetadataSources(registry).buildMetadata().buildSessionFactory();
11         }catch(Exception ex){
12             ex.printStackTrace();
13             StandardServiceRegistryBuilder.destroy(registry);
14         }
15         Session session = sessionFactory.openSession();
16         //1 执行 SQL
17         SQLQuery query = session.createSQLQuery("select id,name,bank,balance from
18         account where ID = ?");
19         //设置第一个参数的值为 1，即查询 ID=1 的 account
20         query.setParameter(0, 1);
21         List<Object> list = query.list();
22         for(int i=0;i<list.size();i++)
23         {
24             Object[] one = list.get(i);
25             System.out.println(one[0] + "\t" + one[1] + "\t" + one[2] + "\t" + one[3]);
26         }
27     }
28 }
```

在第 15 行中使用的是 SQLQuery 对象，而不是像之前那样的 Query 对象，该对象是由 session.createSQLQuery 方法创建的。

在 createSQLQuery 的参数中用到的是 SQL（而不是 HQL），这里还用到了带问号这种预处理的方式。在第 17 行中通过 setParameter 方法，给第 0 个（索引号从 0 而不是 1 开始）问号的位置设置了参数 1。

```

24         //实体查询
25         query = session.createSQLQuery("select id,name,bank,balance from account where
            id = ?").addEntity(Account.class);
26         query.setParameter(0, 1);
27         List<Account> accList = query.list();
28         for(int i=0;i<accList.size();i++)
29         {
30             Account one = accList.get(i);
31             System.out.println(one.getID() + "\t" + one.getName() + "\t" + one.getBank() +
                "\t" + one.getBalance());
32         }

```

演示的第二种方式是通过实体来返回查询结果，请大家注意第 25 行通过 `addEntity` 的方式设置了该 SQL 语句将返回 `Account` 类型，之后在第 27 行里，用了 `List<Account>` 而不是 `List<Object[]>` 来接收结果。

```

33         //执行更新
34         query = session.createSQLQuery("update balance = ? from Account where id = ?");
35         query.setFloat(0, 2000f);
36         query.setInteger(1, 1);
37         query.executeUpdate();
38         session.close();
39         sessionFactory.close();
40     }
41 }

```

最后演示了执行更新操作（而不是查询操作）的方式，这里还是用到了预处理的方式，是通过第 37 行的 `executeUpdate` 方式执行更新操作。

7.3.4 通过 Criteria 设置查询条件

`Criteria` 提供了一种更为直观的查询方式，可以通过设置一些查询条件，实现类似于 `where`、`group by` 和 `order by` 等功能的操作。

从功能上讲，通过 `Criteria` 及其相关类，能实现较为复杂的查询，比如多表关联外加子查询等，但在实际开发中，一般只用它进行单表操作，而不会用它添加过多的查询条件。

一方面，很难把一些带 `in`、`join`、`group by` 甚至 `with` 等复杂语句翻译成基于 `Criteria` 的语句，另一方面，即使勉强翻译成了，那么相关的 `Criteria` 代码也会很长、很复杂，别人不大容易读懂，使得维护性很差。在真的遇到需要有很复杂的 SQL 语句的情况下，一般是用前文提到的 `SQLQuery` 对象来实现。

代码位置	视频位置
code/第 7 章/HibernateQueryCrDemo	视频/第 7 章/Hibernate 里 Criteria 对象的用法

这个项目是根据 HibernateHQLDemo 项目改编的，只修改其中的 HibernateMain 这个类。

```

1  //省略 package 和 import
2  public class HibernateMain {
3      public static void main(String[] args) {
4          StandardServiceRegistry registry = null;
5          SessionFactory sessionFactory = null;
6          try{
7              registry = new
StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
8              sessionFactory = new
MetadataSources(registry).buildMetadata().buildSessionFactory();
9          }catch(Exception ex){
10              ex.printStackTrace();
11              StandardServiceRegistryBuilder.destroy(registry);
12          }
13          Session session = sessionFactory.openSession();
14          //到这里和往常一样得到 Session 对象
15          //通过 Criteria 添加条件
16          Criteria crit = session.createCriteria(Account.class);
17          crit.add(Restrictions.ge("ID", new Integer(5)));
18          crit.add(Restrictions.and( Restrictions.ge("balance", new
Float(300f)) ));
19          crit.addOrder(Order.asc("name"));
20          crit.setMaxResults(2);
21          List<Account> accountList = crit.list();
22          for(int i=0;i<accountList.size();i++)
23          {
24              Account one = accountList.get(i);
25              System.out.println(one.getID() + "\t" + one.getName() + "\t" + one.getBank() +
"\t" + one.getBalance());
26          }

```

在 16 行通过 Session 对象创建了一个 Criteria 对象，它将作用于 Account 这个 Model 类，也就是数据库里的 Account 表。

Restrictions 一般和 Criteria 对象配套使用，通过它可以设置查询条件，比如，在第 17 行中通过 Restrictions.ge 方法，添加了“ID 大于等于 5”这个条件；在第 18 行，不仅通过 ge 方法添加了“balance 大于等于 300”这个条件，而且还通过 Restrictions.and 方法，指定了这个条件和第 17 行中添加的条件之间的关系是“and”。

在第 19 行，通过 Criteria 提供的 addOrder 方法，指定了返回对象将按 name 升序排列，在第 20 行，通过 setMaxResults(2)，指定了返回前 2 条数据。

设置好必要的条件后，在第 21 行中，通过 list 方法返回了用 Criteria 条件得到的结果。由于在第 16 行已经指定了该 Criteria 对象作用在 Account 这个 Model 上，所以第 21 行的返回结果是 List<Account>类型，随后在第 22 行到第 26 行之间，用 for 循环输出结果。

```

27         //返回记录条数
28         List<results> = session.createCriteria(Account.class)
29         .setProjection(Projections.rowCount())
30         .list();
31         System.out.println(results.get(0));

```

Projections 类主要用于数据的分组查询和统计功能，在第 29 行中，通过该类提供的 rowCount 方法，指定了回 Account 的记录条数，并且在第 31 行中输出了这个结果。

```

32         ProjectionList proList = Projections.projectionList();
33         proList.add(Projections.rowCount());
34         proList.add(Projections.max("balance"));
35         proList.add(Projections.sum("balance"));
36         proList.add(Projections.groupProperty("name"));
37         Criteria crit = session.createCriteria(Account.class);
38         crit.setProjection(proList);
39         results = crit.list();
40         //输出结果
41         for(int i=0;i<results.size();i++)
42         {
43             Object[] arr = (Object[])results.get(i);
44             for(Object ob : arr){
45                 System.out.print(ob.toString() + "\t");
46             }
47             System.out.println();
48         }
49         //关闭必要的对象
50         session.close();
51         sessionFactory.close();
52     }
53 }

```

通过 Projections 对象可以进行分组操作，在第 36 行中，设置了在 Account 表里，对 name 进行分组，同时，在第 33 行到第 35 行之间，设置了针对分组后的计算，通过打印的输出结果可以看出，这里等价于如下的 SQL 语句。

```
Hibernate: select count(*) as y0_, max(this_.Balance) as y1_, sum(this_.Balance) as
```



```
y2_, this_.Name as y3_ from Account this_ group by this_.Name
```

第 17 行，通过 Restrictions.ge 设置了“大于等于”的效果，此外，Restrictions 还支持如表 7.7 所示的表达式。

表 7.7 R estrictions 支持的表达式

表达式	含义
Restrictions.eq =	
Restrictions.gt	>
Restrictions.ge	>=
Restrictions.lt	<
Restrictions.le	<=
Restrictions.between BET	WEEN
Restrictions.like LIK	E
Restrictions.in in	
Restrictions.and and	
Restrictions.or or	

从上述的代码中可知，通过 Criteria 对象能很直观地设置查询条件，或者设置分组以及排序情况，但是我们也看到，一般是要通过设置多个 add 等语句，才能完成一句 SQL 语句所对应的动作。

所以在使用时请大家控制好分寸，如果遇到 add 等语句过多的情况（正常情况下，这类设置条件分组排序等语句不宜超过 10 个，这个尺度各项目组会自己把握），那么你就得改写了，或者可以直接使用能运行 SQL 语句的 SQLQuery 对象。

7.4 针对 Hibernate 基础知识部分的小结

本章接触到了基于 Hibernate ORM 的编程方式，该方式是通过配置文件（或者注解）把数据表里的数据映射到 Java Model 里的，随后是操作 Model 对象（而不是数据表），完成操作后再把 Model 对象更新到数据库里。

这部分给出的都是非常基础的知识点，请大家对照如下问题点来回顾自己的学习情况。

问题 1，在项目里，你是怎么实现数据表和 Java Model 类之间的映射的？

可以通过配置文件，也可以通过注解。

问题 2，在 Hibernate 里，有哪些主键的生成方式？在项目里你用到的是哪些方式？

问题 3，在 Hibernate 里，有临时对象、持久化对象和游离对象这三类，你了解多少？通过项目里的例子举例说明这三种状态？

首先你需要对 Session 的方法有一定的了解，其次你需要知道这些方法对状态的影响，比如调用 save 后会变成什么状态。

问题 4，通过 HQL 语言，如何执行带 where、group by 的操作，如何通过 HQL 执行表之间的关联操作。

问题 5，如何在 Hibernate 里执行 SQL 语句。

问题 6，通过 Criteria 对象，如何设置查询条件，如何设置分组？

第 8 章

通过 ORM 专注业务（高级篇）

在第 7 章的基础上，本章将把 Hibernate 应用到更多的项目场景里。

通过本章讲的“映射”知识点，可把数据表里的数据映射成 List、Map 和 Set 等类型，以便能更好地在业务代码里使用。此外，还将通过 Hibernate 把多张表关联到一起，并实现级联操作的效果，这种操作会让我们管理业务对象变得更加方便。

类似于 Spring 拦截器，在 Hibernate 里也可以把一些通用的方法放到“拦截方法”里，以避免复制重复的代码。更重要的是，还将说明在项目里提升 Hibernate 性能的一些经验。

上一章是让你会用 Hibernate，本章则是让你了解作为有 3 年 Hibernate 经验的初级程序员应该知道的事情。

8.1 通过 Hibernate 关联多张表

我们经常关联查询多个数据表以获得业务结果，对应地，也可以通过 Hibernate 把多个表的关联结果映射到 Java Model 类里。数据表之间的关联关系可以是一对一、一对多或是多对

多，这些关联关系都可以通过 Hibernate 来实现。

8.1.1 通过配置文件实现一对一关联

这里将用到两张表，第一是描述账户信息的 Person 表，它的字段如表 8.1 所示。

表 8.1 Pers on表结构

字段名	类型	含义
ID V	archar	主键，用户的 ID
Name V	archar	用户名
Phone V	archar	电话

和这张表对应的是用于描述银行卡的 Card 表，结构如表 8.2 所示，其中的 PersonID 和 Person 表里的 ID 相对应，它们之间是外键关系。

表 8.2 C ard 表结构

字段名	类型	含义
CardID v	archar	主键，用户的 ID
PersonID var	char	用户 ID，和 Person 表里的 ID 字段相对应
Bank var	char	银行名
Balance f	loat	银行卡的余额

这个案例的项目名是 HibernateSenior1to1Demo，下面将通过这个 Java 项目来描述一个账户只能拥有一张银行卡这种一对一之间的对应关系。

请注意，这种对应关系是单向的：通过一个 Person 对象，能查找到对应的 Card 对象，但反过来就不行了。

代码位置	视频位置
code/第 8 章/HibernateSenior1to1Demo	视频/第 8 章/Hibernate 里的单向一对一关联

步骤一 在完成创建项目，导入必要的 jar 包后，编写配置文件 hibernate.cfg.xml，其中包含了数据库连接等配置信息。

```
1 <!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration
  DTD 3.0//EN" "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
2 <hibernate-configuration>
3 <session-factory>
4 <property name="hibernate.connection.driver_class">
5     com.mysql.jdbc.Driver
6 </property>
```

```

7  <!-- 数据库连接设置 -->
8  <property name="hibernate.connection.url">
9      jdbc:mysql://localhost:3306/hibernatechart
10 </property>
11 <property name="hibernate.connection.username">root</property>
12 <property name="hibernate.connection.password">123456</property>
13 <!-- 打印 SQL 语句-->
14 <property name="hibernate.show_sql">true</property>
15 <!-- SQL dialect 方言 -->
16 <property name="hibernate.dialect">
17     org.hibernate.dialect.MySQLDialect
18 </property>
19 <property name="hibernate.hbm2ddl.auto">
20     create
21 </property>
22 <!-- 添加实体类的映射文件-->
23 <mapping resource="Model/Card.hbm.xml" />
24 <mapping resource="Model/Person.hbm.xml" />
25 </session-factory>
26 </hibernate-configuration>

```

这里的一些属性在第 7 章都用到过，不同的是，在第 23 行和第 24 行通过 `mapping` 属性设置了针对两个表的映射文件。

请大家重点注意，在第 19 行到第 21 行之间，设置了 `hibernate.hbm2ddl.auto` 这个属性值是 `create`，这意味着，每次运行该项目时，都会先到数据库里删除 `Card` 和 `Person` 两张表，随后根据 `Card.hbm.xml` 和 `Person.hbm.xml` 里的描述再重建它们，原先表里的数据将全部丢失。这样做的原因是为了方便演示，大家在项目里，如果没有特殊的原因，不要使用这个值。

步骤二 编写针对 `Person` 表和 `Card` 表的两个映射文件，其中 `Person.hbm.xml` 的代码如下。

```

1  <?xml version="1.0"?>
2  <!DOCTYPE hibernate-mapping PUBLIC
3  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
5  <hibernate-mapping package="Model">
6  <class name="Person" table="Person">
7  <id name="ID" type="java.lang.String">
8  </id >
9  <property name="Name" type="java.lang.String">
10 <column name="Name" length="20" not-null="true" />
11 </pr operty>
12 <pro perty name="Phone" type="java.lang.String">
13 <column name="Phone" length="20"></column>

```

```

14     </property>
15     <one-to-one name="Card" class="Model.Card" cascade="all" ></one-to-one>

16 </class>
17 </hibernate-mapping>

```

在第6行中，设置了 Java Model 类和数据表之间的对应关系，具体是，Person.java 这 Model 将对应于数据表里 Person 这个表。

在第7行到第14行，设置了数据表里的诸多字段和 person.java 这个 Model 里各属性的对应关系。在第15行，通过 one-to-one 这个元素，设置了 Person 和 Card 这两个业务实体实现的对应关系，并通过 cascade 设置了级联操作关系。

通过 cascade 定义了针对 Person 的附属的 Card 对象的操作方式。这里的值是 all，说明插入父对象（Person）时，将自动插入子对象（Card），同样当我们删除父对象时，其附属的子对象也将被删除。

Card.hbm.xml 的代码如下，其中定义了 Card 表和 Card.java 类之间的对应关系。

```

1  <?xml version="1.0"?>
2  <!DOCTYPE hibernate-mapping PUBLIC
3  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
5  <hibernate-mapping package="Model">
6  <class name="Card" table="Card">
7  <id name="CardID" column="CardID" type="java.lang.String">
8  </id>
9
10 <property name="PersonID" type="java.lang.String">
11     <column name="PersonID" length="20" not-null="true"/>
12 </property>
13     <property name="Bank" type="java.lang.String">
14         <column name="Bank" length="20"></column>
15     </property>
16 <property name="Balance" type="float">
17     <column name="Balance"></column>
18 </property>
19 </class>
20 </hibernate-mapping>

```

通过上述的两个映射文件，知道了账户和银行卡之间是一对一的关系，随后，一起来看一下在 Model 文件里是如何定义这层对应关系的。

步骤三 编写 Person.java 和 Card.java 这两个 Model 类。

```
1 public class Person {
2     private String ID;
3     private String Name;
4     private String Phone;
5     //体现一对一的关系，保存映射类的实例对象
6     private Card card;
7     //省略针对各属性的 get 和 set 方法
8 }
```

在第 2 行到第 4 行中定义了和 Person 表里各字段相对应的各属性。

由于一个账户只能拥有一张银行卡，所以在第 6 行中定义了 Card（而不是 List<Card>）来存放该 Person 所拥有的卡。

而 Card.java 这个类更简单，其只包含了和 Card 表中各字段对应的属性。

```
1 public class Card {
2     private String CardID;
3     private String PersonID;
4     private String Bank;
5     private float Balance;
6     //省略 get 和 set 方法
7 }
```

最后，来看下在 HibernateMain.java 这个类里是如何实现这种一对一关系的。

```
1 public class HibernateMain {
2     public static void main(String[] args) {
3         StandardServiceRegistry registry = null;
4         SessionFactory sessionFactory = null;
5         try{
6             registry = new
7             StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
8             sessionFactory = new
9             MetadataSources(registry).buildMetadata().buildSessionFactory();
10        }catch(Exception ex){
11            ex.printStackTrace();
12            StandardServiceRegistryBuilder.destroy(registry);
13        }
14        Session session = sessionFactory.openSession();
15        //以上得到 sessionFactory 和 Session 对象
16        //产生一张卡
17        Card card = new Card();
18        card.setCardID("Card123");
19        card.setPersonID("Person123");
20        card.setBank("Citi");
```

```

19         card.setBalance(100f);
20 //生成一个 Person 对象
21         Person person = new Person();
22         person.setID("Person123");
23         person.setName("Peter");
24         person.setPhone("123456");
25 //给 Person 对象设置一个 Card 对象
26         person.setCard(card);
27 //这里只 save 了 Person 对象，没 save Card 对象
28         session.save(person);
29         //用 load 方法拿到刚插入的 Person 时，能同时拿到附属的 Card 对象
30         Person one = (Person)session.load(Person.class, "Person123");
31         Card oneCard = one.getCard();
32         //能看到该 Person 所对应的 Card 的各属性
33         System.out.println(oneCard.getCardID() + "\t" + oneCard.getPersonID() + "\t" +
            oneCard.getBank() + "\t" + oneCard.getBalance() );
34         //删除 Person 对象后，附属的 Card 对象也一起被删除了
35         session.delete(one);
36         session.flush();
37         session.close();
38         sessionFactory.close();
39     }
40 }

```

在第 14 行到第 19 行中，生成了一个 Card 对象，并设置了对应的值，在第 21 行到第 26 行中生成了 Person 对象，在第 28 行中通过 save 方法把 Person 对象持久化到数据库里。

这个运行结果是，虽然没有保存 Card，但由于保存了 Person，而且 Person 拥有一个 Card，所以在数据表里，能同时在 Person 和 Card 表里分别看到一条记录。

所以在第 30 行中，用主键“Person123”，通过 load 方法得到刚插入的一个 Person 对象后，在第 33 行中，能看到其附属的 Card 信息。

在第 35 行中，通过 delete 方法删除 Person 对象后，会发现它对应的 Card 对象也会被删除。其原因是，在 Person.hbm.xml 这个配置文件里，通过 cascade 这个属性定义了父子对象之间的级联关系是 all。

8.1.2 一对一关联的注解实现方式

在这个例子中，将通过注解的方式，实现“双向”的一对一对象关系，也就是说，不仅能通过一个 Person 对象定位到一个 Card，反过来通过一个 Card 也能找到其唯一对应的 Person。

代码位置	视频位置
code/第 8 章/HibernateSenior1to1AnnoDemo	视频/第 8 章/通过注解实现双向一对一关联

由于 Person 和 Card 表都需要关联到对方，所以表结构如表 8.3 和表 8.4 所示。

表 8.3 双向一对一的 Person 表结构

字段名	类型	含义
ID v	archar	主键，用户的 ID
Name var	char	用户名
Phone var	char	电话
CardID v	archarr	关联到 Card 表

表 8.4 双向一对一的 Card 表结构

字段名	类型	含义
CardID v	archar	主键，用户的 ID
PersonID var	char	用户 ID，和 Person 表里的 ID 字段相对应
Bank var	char	银行名
balance float		银行卡的余额

这个项目里的 hibernate.cfg.xml 和 HibernateSenior1to1Demo 项目里的很相似，不同的是，这里我们是通过 mapping class 来指定映射的 Model 类，而不是指定 hbm.xml 这样的映射文件。

```

1  <!-- 添加实体类的映射文件-->
2  <mapping class="Model.Person" />
3  <mapping class="Model.Card" />

```

这里是在 Model 文件里通过注解来配置映射关系，下面一起来看一下 Person.java。

```

1  //省略必要的 import 代码
2  @Entity
3  @Table(name="Person")
4  public class Person {
5      //设置主键
6      @Id
7      @Column(name = "ID")
8      private String ID;
9      //通过@Column 逐一设置数据表字段和各属性之间的对应关系
10     @Column(name = "Name")
11     private String Name;
12     @Column(name = "Phone")
13     private String Phone;
14     //定义一对一关系
15     @OneToOne(cascade=CascadeType.ALL)

```

```

16 @JoinColumn(name = "CardID")
17     private Card card;
18 //省略 get 和 set 方法
19 }

```

通过第 2 行和第 3 行说明该 Model 类和 Person 表相对应。

请注意在第 15 行和第 16 行,通过@OneToOne 这个注解,说明了 Person 表里是通过 CardID 这个字段和 Card 相对应,而且 Person 和 Card 之间是一一对应关系。此外,还通过 cascade 设置了两个业务对象之间的级联关系。

在 Card.java 里,通过@OneToOne 注解设置了 Card 和 Person 之间的对应关系,也就是说,Person 和 Card 之间可以相互关联,即所谓的“双向一对一关联”。

```

1 //省略必要的 import
2 @Entity
3 @Table(name="Card")
4 public class Card {
5     @Id
6     @Column(name = "CardID")
7     private String CardID;
8     @Column(name = "Bank")
9     private String Bank;
10    @Column(name = "Balance")
11    private float Balance;
12    //通过 PersonID, 和 Person 之间有一一对应关系
13    @OneToOne(cascade=CascadeType.ALL)
14    @JoinColumn(name = "PersonID")
15    private Person person;
16    //省略必要的 get 和 set 方法
17 }

```

由于 Person 和 Card 之间是双向一对一关系,所以在 HibernateMain.java 里,可以通过其中的一个对象访问到另外一个。

```

1 //省略必要的 import 代码
2 public class HibernateMain {
3     public static void main(String[] args) {
4         StandardServiceRegistry registry = null;
5         SessionFactory sessionFactory = null;
6         try{
7             registry = new
8             StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
9             sessionFactory = new
10             MetadataSources(registry).buildMetadata().buildSessionFactory();

```

```

9         }catch(Exception ex){
10             ex.printStackTrace();
11             StandardServiceRegistryBuilder.destroy(registry);
12         }
13         //得到 Session 对象
14         Session session = sessionFactory.openSession();
15         //生成 Card 业务实体对象
16         Card card = new Card();
17         card.setCardID("Card123");
18         card.setBank("Citi");
19         card.setBalance(100f);
20         //生成 Person 业务实体对象
21         Person person = new Person();
22         person.setID("Person123");
23         person.setName("Peter");
24         person.setPhone("123456");
25         card.setPerson(person);
26         person.setCard(card);
27         //save Person 对象后，对应的 Card 也能一起插入数据表
28         session.save(person);
29         session.flush();
30         Person onePerson = (Person)session.load(Person.class, "Person123");
31         Card oneCard = onePerson.getCard();
32         System.out.println(oneCard.getCardID() + "\t" + oneCard.getPerson().getName() +
            "\t" + oneCard.getBank() + "\t" + oneCard.getBalance() );

```

在第 28 行中，通过 `session.save` 方法，把 `Person` 对象持久化，也就是插入到数据表里，在第 31 行和第 32 行里，能够通过 `Person` 对象获得其附属的 `Card` 对象。

```

33         //删除 Person 对象后，对应的 Card 也会被一起删除
34         session.delete(onePerson);
35         session.flush();
36         //保存 Card 对象后，其附属的 Person 对象也能一起被持久化
37         session.save(card);
38         session.flush();
39         oneCard = (Card)session.load(Card.class, "Card123");
40         onePerson = oneCard.getPerson();
41         System.out.println(onePerson.getName() + "\t" + onePerson.getPhone());
42         //同样，删除 Card 对象后，其附属的 Person 对象也会被删除
43         session.delete(oneCard);
44         session.flush();
45         session.close();
46         sessionFactory.close();
47     }
48 }

```

如果想通过配置文件的方式实现双向的一对一关联，那么就需要在两边的 hbm.xml 文件里，通过如下形式的 one-to-one 代码来设置。

```
<one-to-onename="子类名" class="Model 类" cascade="all">
</one-to-one>
```

8.1.3 一对多关联（配置文件，返回 List）

这里的业务是，一个人（Person 对象）有多张银行卡（Card 对象），那么在一个 Person 对象里，就有必要用一个容器（一般是 List 或者 Set）来保存他拥有的多张银行卡。

代码位置	视频位置
code/第 8 章/HibernateSeniorOne2MoreDemo	视频/第 8 章/通过配置文件实现一对多的关联

该案例的表结构如下。由于 Person 表无须关联到 Card 表，所以不用 CardID 这个字段。

表 8.5 一对多的 Person 表结构

字段名	类型	含义
ID V	archar	主键，用户的 ID
Name var	char	用户名
Phone var	char	电话

表 8.6 一对多的 Card 表结构

字段名	类型	含义
CardID v	archar	主键，用户的 ID
PersonID var	char	用户 ID，和 Person 表里的 ID 字段相对应
Bank var	char	银行名
balance float		银行卡的余额
cardIndex int		用于排序的索引值

我们用 List 来管理一个人有多张银行卡的情况。在这种情况下，可以保存多个重复的对象，同时可以保证有序性。

假设这里是个扣钱业务，某人有多张银行卡，首先是到花旗卡上扣钱，如果花旗卡没钱，那么到建行卡，如果建行卡也没钱，再找某某其他卡。这里就涉及业务上的“有序性”，而 cardIndex 就用来保存这个有序性。

创建好项目后，首先编写 hibernate.cfg.xml 文件。这里重用 HibernateSenior1to1Demo 项目，请注意以下几个要点。

第一，设置 hbm2dll.auto 属性是 create，这样就没必要每次都创建表了。再次强调，在这

种方式下每次启动 Hibernate 后都会清空数据，请务必谨慎使用。

```
<property name="hibernate.hbm2ddl.auto">create</property>
```

第二，通过 `mapping resource` 来设置对应的映射文件。

```
<mapping resource="Model/Card.hbm.xml"/>
<mapping resource="Model/Person.hbm.xml" />
```

其他连接 MySQL 表的配置完全一致，这里不再重复。

第三，通过 `Person.hbm.xml` 这个映射文件来指定一对多关联，代码如下。

```
1  <?xml version="1.0"?>
2  <!DOCTYPE hibernate-mapping PUBLIC
3  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
5  <hibernate-mapping package="Model">
6  <class name="Person" table="Person">
7  <id name="ID" type="java.lang.String"></id>
8  <property name="Name" type="java.lang.String">
9  <column name="Name" length="20" not-null="true" />
10 </pr operty>
11 <pro perty name="Phone" type="java.lang.String">
12 <column name="Phone" length="20"></column>
13 </pr operty>
14 <list name="cardList" table="Person" cascade="all">
15 <key column="PersonID" />
16 <list-index column="cardIndex"/>
17 <one-to-many class="Model.Card"/>
18 </list>
19 </class>
20 </hibernate-mapping>
```

从第 7 行到第 13 行，指定了映射关系，从第 14 行到第 18 行，通过 `List` 元素指定了 `Person` 和 `Card` 之间的一对多关联。

第 14 行中，指定了 `name` 是 `cardList`，那么在 `Person.java` 这个类里，就必须要用 `List<Card>` `cardList` 这个属性来保存一个 `Person` 所对应的多张银行卡。用 `table` 来说明一对多里的“一”，也就是 `Person` 所对应的数据表，这里的级联操作依然是 `all`。

在表 8.6 中，已经看到了一对多案例中 `Card` 表，该表是用 `PersonID` 字段来和 `Person` 表关联。在第 15 行，通过 `key` 来说明在“多方”（也就是银行卡 `Card` 这端）和 `Person` 对象关联的字段，这个值是 `PersonID`。

在第 16 行中，指定了用于描述多方索引次序的 `cardIndex` 字段。在第 17 行中，通过 `one-to-many` 这个元素来说明多方的实体对象是 `Card`。

第四，我同样需要定义描述 `Card` 映射关系的 `card.hbm.xml` 文件，代码如下，这里纯粹是描述映射关系，所以不再重复讲解。

```

1  <?xml version="1.0"?>
2  <!DOCTYPE hibernate-mapping PUBLIC
3  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
5  <hibernate-mapping package="Model">
6  <class name="Card" table="Card">
7  <id name="CardID" column="CardID" type="java.lang.String">
8  </id>
9  <property name="PersonID" type="java.lang.String">
10 <column name="PersonID" length="20" not-null="true"/>
11 </property>
12 <property name="Bank" type="java.lang.String">
13 <column name="Bank" length="20"/></column>
14 </property>
15 <property name="Balance" type="float">
16 <column name="Balance"/></column>
17 </property>
18 </class>
19 </hibernate-mapping>

```

第五，定义 `Person` 和 `Card` 这两个实体类。`Person.java` 代码如下。

```

1  public class Person {
2      private String ID;
3      private String Name;
4      private String Phone;
5      //通过 List 来实现一对多
6      private List<Card> cardList;
7      //省略必要的 get 和 set 方法
8  }

```

这里除了在第 2 行到第 4 行中定义了和 `Person` 表里相映射的字段外，还在第 6 行用 `List<Card>` 的形式，定义了用来存储多个卡的 `cardList` 字段。注意，这个 `cardList` 属性名需要和 `person.hbm.xml` 里的 `list name` 相一致。

而 `Card.java` 的代码就相对简单了，就定义了一些能与 `Card` 表里相映射的字段。

```

1  public class Card {
2      private String CardID;

```

```
3 private String PersonID;
4 private String Bank;
5 private float Balance;
6 //省略 get 和 set 方法
7 }
```

最后，通过 `HibernateMain.java`，来观察下一对多关联的用法。

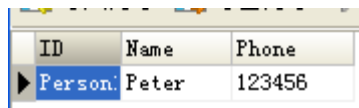
```
1 //省略必要的 import 代码
2 public class HibernateMain {
3     public static void main(String[] args) {
4         StandardServiceRegistry registry = null;
5         SessionFactory sessionFactory = null;
6         try{
7             registry = new
8             StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
9             sessionFactory = new
10             MetadataSources(registry).buildMetadata().buildSessionFactory();
11         }catch(Exception ex){
12             ex.printStackTrace();
13             StandardServiceRegistryBuilder.destroy(registry);
14         }
15         Session session = sessionFactory.openSession();
16         //这里我们生成了两个 Card 对象
17         Card card1 = new Card();
18         card1.setCardID("Card123");
19         card1.setPersonID("Person123");
20         card1.setBank("Citi");
21         card1.setBalance(100f);
22         Card card2 = new Card();
23         card2.setCardID("Card456");
24         card2.setPersonID("Person123");
25         card2.setBank("ICBC");
26         card2.setBalance(200f);
27         //生成一个 Person 对象
28         Person person = new Person();
29         person.setID("Person123");
30         person.setName("Peter");
31         person.setPhone("123456");
32         //把两个 Card 对象放到一个 list 里
33         List<Card> cards = new ArrayList<Card>();
34         cards.add(card1);
35         cards.add(card2);
36         //把这个 list 存入 Person 对象
37         person.setCardList(cards);
```

```

36 //save person
37     session.save(person);
38     session.flush();

```

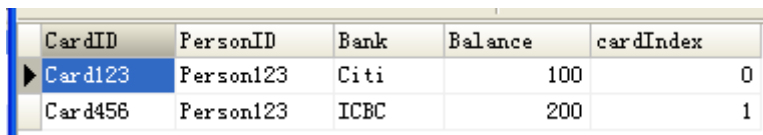
这里生成了两个 Card 对象，并把它们设置到 Person 对象的 cardList 里。运行到这里，能在 Person 表里看到如图 8.1 所示的一条记录。



ID	Name	Phone
Person123	Peter	123456

图 8.1 Person 表里的一条记录

由于设置了级联操作，所以在 Card 表里能同时看到两条 Card 记录，如图 8.2 所示。请注意它们的 cardIndex 的值，随着插入时间的推移，这个索引值是自动增长的。



CardID	PersonID	Bank	Balance	cardIndex
Card123	Person123	Citi	100	0
Card456	Person123	ICBC	200	1

图 8.2 Card 表里的两条 Card 记录

```

39 Person one = (Person)session.load(Person.class, "Person123");
40 List<Card> cardList = one.getCardList();
41 for(int i = 0; i < cardList.size(); i++)
42 {
43     Card oneCard = cardList.get(i);
44     System.out.println(oneCard.getCardID() + "\t" + oneCard.getPersonID() + "\t" +
45         oneCard.getBank() + "\t" + oneCard.getBalance() );
46 }
47 //以上是打印语句
48 session.delete(person);
49 session.flush();
50 session.close();
51 sessionFactory.close();
52 }

```

问题在第 47 行的 delete 上，这个动作会抛出异常。原因是，当删除 Person 对象时，Hibernate 系统并没有像之前那样级联删除它对应的两个 Card 对象，而是企图通过下面的 Query，想在设置 PersonID 和 cardIndex 为空的前提下，保留对应的 Card。

```
update Card set PersonID=null, cardIndex=null where PersonID=?
```

由于在 Card.hbm.xml 里，了解到 Card 的 PersonID 是不能为空的，所以这里会有异常。

为什么这里的 delete 没有级联操作而是仅仅设置对应的字段为空？原因出在 inverse 上。在 Person.hbm.xml 里，定义 list 时，没有加上 inverse 的设置，这里如果做如下的改进，加上 inverse 是 true 的设置，那么在删除 person 时，对应的 Card 也将被删除。

```
1 <list name="cardList" inverse="true" table="Person" cascade="all">
2 <key column="PersonID" />
3 <list-index column="cardIndex"/>
4 <one-to-many class="Model.Card"/>
5 </list>
```

这个现象的背后是 inverse 的含义，它的作用是指定由哪一方来维护关联关系。由于之前并没有设置 inverse，它的默认值是 false。也就是说，Person 端并不负责维护关联关系，所以即使在 Person 对象被删除后，它对应的 Card 对象也不会被删除，而是仅仅去掉和该 Person 对象有联系的痕迹（PersonID 和 cardIndex 设置为空）。

8.1.4 一对多关联（注解，返回 Set）

在上面的案例中，是用 List 来存储一对多关系中的“多方”对象，它的适用场景是多方可以重复，而且需要用 index 来描述多方的有序性。

根据定义，Set 对象不能存储重复值，而且不需要用额外的 index 字段来维护次序关系，所以同样可以通过 Set 对象来保存多方对象。

代码位置	视频位置
code/第 8 章/HibernateSeniorOne2MoreAnno	视频/第 8 章/通过注解实现一对多关联

在这个案例中，通过注解实现了双向的一对多关联。也就是说，一个 Person 类可以关联到多个 Card 类，而反过来，也可以通过 Card 类找到它的主人 Person 类。

该案例的表结构如下。由于 Person 表无须关联到 Card 表，所以不用 CardID 这个字段。

表 8.7 用注解实现可一对多的 Person 表结构

字段名	类型	含义
ID V	archar	主键，用户的 ID
Name V	archar	用户名
Phone V	archar	电话

表 8.8 用注解实现可一对多的 Card 表结构

字段名	类型	含义
CardID V	archar	主键，用户的 ID

续表

字段名	类型	含义
PersonID V	archar	用户 ID，和 Person 表里的 ID 字段相对应
Bank V	archar	银行名
balance Float		银行卡的余额

和刚才的案例相比，这里由于是用 Set，所以无须用 cardIndex 来标识多个 Card 的次序。

在这个项目的 hibernate.cfg.xml 里，用于配置连接数据库的属性和之前一致，所以不再额外给出。

步骤一 由于是注解，所以通过 mapping class 来说明对应的 Model 类。

```
1 <!-- 添加实体类的映射文件-->
2 <mapping class="Model.Person" />
3 <mapping class="Model.Card" />
```

步骤二 编写 Person.java 这个业务对象，代码如下。

```
1 //省略必要的 package 和 import 代码
2 //说明本业务类对应到 Person 这个数据表
3 @Entity
4 @Table(name="Person")
5 public class Person {
6     @Id
7     @Column(name = "ID")
8     private String ID;
9     @Column(name = "Name")
10     private String Name;
11     @Column(name = "Phone")
12     private String Phone;
13 //以上代码用于配置映射的属性
14 @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY, mappedBy = "person")
15     private Set<Card> cards;
16 //省略 get 和 set 方法的代码
17 }
```

在第 15 行中，通过 Set<Card>来存储一个 Person 对象可能包含的多个 Card。而第 14 行是作用在 Set<Card>上的，通过 mappedBy 属性，指定在 Card 对象里，是通过 person（注意大小写）这个属性来表明该 Card 所对应的 Person 类。

这里还用到了 fetch 元素，取值是 LAZY，它有两个取值，默认的是 LAZY。

FetchType.LAZY 是懒加载，这意味着当加载一个业务实体 Person 时，定义懒加载的属性

(对应的 `Set<Card>`) 不会马上从数据库中加载。另外一个取值是 `FetchType.EAGER`，这个是急加载，用到这个值时，定义成急加载的属性会立即从数据库中加载。

步骤三 编写描述卡业务模型的 `Card.java` 代码。

```
1 //描述该业务对象和 Card 这个数据表相关联
2 @Entity
3 @Table(name="Card")
4 public class Card {
5     @Id
6     @Column(name = "CardID")
7     private String CardID;
8     @Column(name = "Bank")
9     private String Bank;
10    @Column(name = "Balance")
11    private float Balance;
12    //之前的代码用来定义各业务属性和数据表里各字段的关联
13    @ManyToOne(cascade = CascadeType.ALL)
14        @JoinColumn(name="PersonID")
15    private Person person;
16    //省略 get 和 set 方法
17 }
18 }
```

通过第 15 行的 `Person` 对象，说明该 `Card`（多方）和 `Person`（一方）的对应关系，这里的属性名需要和 `Person.java` 里的 `mappedBy` 相对应。

```
1 @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY, mappedBy = "person")
```

此外，通过第 14 行的代码，用 `JoinColumn` 来说明 `Card` 和 `Person` 之间是用 `PersonID` 这个字段相关联的。

最后通过 `HibernateMain.java`，看一下如何调用一对多关联。

```
1 //省略 package 和 import 代码
2 public class HibernateMain {
3     public static void main(String[] args) {
4         StandardServiceRegistry registry = null;
5         SessionFactory sessionFactory = null;
6         try{
7             registry = new
8             StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
9             //不指定文件名时默认是找 hibernate.cfg.xml 文件
10            sessionFactory = new
11            MetadataSources(registry).buildMetadata().buildSessionFactory();
12        } catch (Exception e) {
13            e.printStackTrace();
14        }
15    }
16 }
```

```

10         }catch(Exception ex){
11             ex.printStackTrace();
12             StandardServiceRegistryBuilder.destroy(registry);
13         }
14         Session session = sessionFactory.openSession();
15         //创建两个不同的 Card 对象
16         Card card = new Card();
17         card.setCardID("Card123");
18         card.setBank("Citi");
19         card.setBalance(100f);
20         Card anotherCard = new Card();
21         anotherCard.setCardID("Card456");
22         anotherCard.setBank("Citi");
23         anotherCard.setBalance(100f);
24         //创建一个 Person 对象
25         Person person = new Person();
26         person.setID("Person123");
27         person.setName("Peter");
28         person.setPhone("123456");
29         //把两张不同的卡加入到 cards 这个 Set 对象里
30         Set<Card> cards = new HashSet<Card>();
31         card.setPerson(person);
32         anotherCard.setPerson(person);
33         cards.add(card);
34         cards.add(anotherCard);
35         //把 cards 设置到 person 对象里，实现一对多关联
36         person.setCards(cards);
37         session.save(person);
38         session.flush();

```

执行到这里，我们在 Person 表里能看到一条数据，请记住，它的 ID 是 Person123。

ID	Name	Phone
Person123	Peter	123456

在 Card 表里，能看到两条数据，和预期的一致，同时它们的 PersonID 都是 Person123，通过这个字段能关联到对应的 Person 字段。

CardID	Balance	Bank	PersonID
Card123	100	Citi	Person123
Card456	100	Citi	Person123

```

39 Person onePerson = (Person)session.load(Person.class, "Person123");
40 Set<Card> cardSet = onePerson.getCards();
41 //省略 System.out.println 打印语句
42 Card oneCard = (Card)session.load(Card.class, "Card123");

```

```
43     PersoncardOwner = oneCard.getPerson();
44     //省略 System.out.println 打印语句
```

由于在 Person 类里设置了 OneToMany，而在 Card 类里设置了 ManyToOne，所以能从其中的一个对象找到对应的另外一个对象。

```
45     session.delete(onePerson);
46     session.flush();
47     //由于我们通过 cascade 设置了级联关系，所以删除了 onePerson 后，
48     //对应的 cards 也会被从数据库里删除
49     session.close();
50     sessionFactory.close();
51 }
52 }
```

可以通过 List 或是 Set 来保存一对多关系里的“多方”，它们有各自的适用范围，如果多方的对象一定不会重复，那么可以用 Set；如果需要保留多方对象的次序，那么可以通过 List 以及其中的 index 属性来实现。

8.1.5 用 Map 来管理一对多关联

在前文，我们用 List 来管理实体之间的关联，比如在 Person 类里是用如下的代码来保存多个 Card 对象。

```
private List<Card> cardList;
```

我们也实现过用 Set 来保存多个 Card 对象的方式。

```
private Set<Card> cards;
```

此外，我们还可以通过 Map 的形式来保存一个 Person 里的多个 Card 对象。

代码位置	视频位置
code/第 8 章/HibernateSeniorMapDemo	视频/第 8 章/通过 Map 实现一对多关联

这个例子所用到的数据表是如 8.7 和 8.8 两个表所示的 Person 表和 Card 表，其中 Person 表里包含 ID、Name 和 Phone 字段，而 Card 表里包含了 CardID、PersonID、Bank 和 balance 这四个字段。

这些例子中的 hibernate.hbm.xml 文件是沿用 8.1.4 节里的，除了需要配置连接 MySQL 外，还可以通过 mapping resource 来设置对应的映射文件。

```
<mapping resource="Model/Card.hbm.xml"/>
<mapping resource="Model/Person.hbm.xml" />
```

第一，需要改写 Person.java 这个业务类。

```

1  public class Person {
2      //第 3 行到第 5 行的属性和数据表里的一致
3      private String ID;
4      private String Name;
5      private String Phone;
6      //通过 Map 来管理多个 Card，其中 key 是银行卡号，value 是 Card 对象
7      private Map <String,Card> cardMap;
8      //省略必要的 get 和 set 方法
9  }

```

第二，需要改写 person.hbm.xml 这个映射文件。其中，描述 Person.java 里的属性和 Person 表里映射关系的代码不变，在这个基础上，通过如下的 map 元素来定义映射关系。

```

1  <map name="cardMap" table="Card" inverse="true" cascade="all">
2  <key column="PersonID"></key>
3  <!-- 对应于 Map<key,value>中的 key -->
4  <index column="cardID" type="string"></index>
5  <!-- 对应于 Map<key,value>中的 value -->
6  <one-to-many class="Model.Card"/>
7  </map>

```

在第 1 行，name 为 cardMap 的 Map 需要和 Person 类里的 Map 类型的属性名一致。

在第 2 行，描述了在 Card 端，是通过 PersonID 这个字段和 Person 关联的。

在第 4 行，定义了 Map 里的 Key 属性，这里是用 Card 端的 cardID 来作为 Key，而且它是 string 类型的。同样在第 6 行，通过 one-to-many，不仅指定了 Person 和 Card 之间是一对多关系，而且指定了 Map 里的 value 是 Card 对象。

这里指定的 Key 和 Value 类型和 Person.java 里的如下 Map 定义是一致的。

```
private Map <String,Card> cardMap;
```

这里不需要改变 Card.java 和 card.hbm.xml 文件，在 HibernateMain 这个类里，来看下如何使用 Map 对象。

```

1  //省略必要的 package 和 import 代码
2  public class HibernateMain {
3  public static void main(String[] args) {
4      StandardServiceRegistry registry = null;
5      SessionFactory sessionFactory = null;
6      try{
7          registry = new
StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();

```

```

8         sessionFactory = new
    MetadataSources(registry).buildMetadata().buildSessionFactory();
9     } catch (Exception ex) {
10         ex.printStackTrace();
11         StandardServiceRegistryBuilder.destroy(registry);
12     }
13     Session session = sessionFactory.openSession();
14     //初始化两个 Card 对象
15     Card card1 = new Card();
16     card1.setCardID("Card123");
17     card1.setPersonID("Person123");
18     card1.setBank("Citi");
19     card1.setBalance(100f);
20     Card card2 = new Card();
21     card2.setCardID("Card456");
22     card2.setPersonID("Person123");
23     card2.setBank("ICBC");
24     card2.setBalance(200f);
25     //初始化一个 Person 对象
26     Person person = new Person();
27     person.setID("Person123");
28     person.setName("Peter");
29     person.setPhone("123456");
30     //把两个 Card 对象放入一个 Map 里
31     Map<String, Card> cards = new HashMap<String, Card>();
32     cards.put(card1.getCardID(), card1);
33     cards.put(card2.getCardID(), card2);
34     person.setCardMap(cards);
35     //保存 person 对象
36     session.save(person);
37     session.flush();
38     //之后通过 load 方法，得到一个 Person 对象
39     Person one = (Person) session.load(Person.class, "Person123");
40     //通过 Map 对象来获取该人所拥有的 Card
41     Map<String, Card> cardsForOnePerson = one.getCardMap();
42     // 遍历这个 Map，打印出卡号和余额
43     for (Map.Entry<String, Card> entry : cardsForOnePerson.entrySet()) {
44         System.out.println("key= " + entry.getKey() + " and value= " +
            entry.getValue().getBalance());
45     }
46     session.close();
47     sessionFactory.close();
48 }
49 }

```

8.1.6 通过配置文件实现多对多关联

在实际的项目里，还可能遇到多对多的关联，比如在某个选课系统里，学生和课程之间就是多对多的关联。

代码位置	视频位置
code/第 8 章/HibernateSenorM2MDemo	视频/第 8 章/通过配置文件实现多对多关联

在这个案例中，通过配置文件的方式实现了多对多关联。

这里将用到三张表：第一是描述学生信息的 Student 表，第二是描述课程信息的 Course 表，第三是描述学生选课信息的 Students_Courses 表。Student 表的字段如表 8.9 所示。

表 8.9 多对多关联中的 Student 表

字段名	类型	含义
StudentID V	archar	主键，学生 ID
Name V	archar	学生的姓名

描述课程信息的 Course 表的字段如表 8.10 所示。

表 8.10 Course 表

字段名	类型	含义
CourseID V	archar	主键，课程的 ID
CourseName V	archar	课程名

在 Students_Courses 表里描述学生和课程之间的对应关系，字段如表 8.11 所示。

表 8.11 Students_Courses 表

字段名	类型	含义
Student_id V	archar	学生 ID
Course_ID V	archar	课程

步骤一 在创建完 Java 项目并导入必要的 jar 包后，首先创建描述连接信息的 hibernate.cfg.xml。其中的连接属性和前文的一致，有区别之处是通过 mapping resource 指定两个对应的映射文件。

```
1 <!-- 添加实体类的映射文件-->
2 <mapping resource="Model/Student.hbm.xml" />
3 <mapping resource="Model/Course.hbm.xml" />
```

步骤二 编写描述学生业务实体的 Student 表和描述映射关系的 Student.hbm.xml。

Student.java 的代码如下。

```
1 //省略必要的 package 和 import 代码
2 public class Student {
3     private String studentID;
4     private String studentName;
5     private Set<Course> courses;
6     //省略 get 和 set 方法
7 }
```

在第 5 行，通过 Set<Course>来描述该学生所选的课程集合。

Student.hbm.xml 代码如下。

```
1 <?xml version="1.0"?>
2 <!DOCTYPE hibernate-mapping PUBLIC
3     "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4     "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
5 <hibernate-mapping package="Model">
6     <class name="Student" table="Student">
7         <id name="studentID" column="StudentID" type="java.lang.String">
8
9         </id>
10        <property name="studentName" type="java.lang.String">
11            <column name="Name" length="20" not-null="true" />
12        </property>
13        <set name="courses" table="students_courses" cascade="save-update">
14            <key column="student_id"></key>
15            <many-to-many class="Model.Course" column="course_id"></many-to-many>
16        </set>
17    </class>
18 </hibernate-mapping>
```

从第 12 行到第 15 行，通过 set 指定了和 Student 表相对应的 Course 表。建表时已经看到，这个案例中通过 Students_Courses 表来存放学生和课程的对应关系，这个表名需要和第 12 行的 table 属性相一致。

在第 13 行，指定了在 Students_Courses 表里，指向 Student 表的字段名是 student_id。在第 14 行，通过 many-to-many 属性和 class，指定了 Student 实体和 Course 实体相对应，而且通过 column 指定了在 Students_Courses 表里，指向 Course 表的字段名是 course_id。

步骤三 编写描述课程业务实体的 Course 和描述映射关系的 Course.hbm.xml。Course.java 代码如下，代码中第 5 行通过 Set<Student>定义了和课程相对应的学生集合。

```
1 //省略必要的 package 和 import
```

```

2 public class Course {
3     private String courseID;
4     private String courseName;
5     private Set<Student> students;
6     //省略必要的 get 和 set 方法
7 }

```

Course.hbm.xml 代码如下。

```

1 <?xml version="1.0"?>
2 <!DOCTYPE hibernate-mapping PUBLIC
3     "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4     "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
5 <hibernate-mapping package="Model">
6     <class name="Course" table="Course">
7         <id name="courseID" column="CourseID" type="java.lang.String">
8             </id>
9         <property name="courseName" type="java.lang.String">
10             <column name="CourseName" length="20" not-null="true"/>
11         </property>
12         <set name="students" table="students_courses" cascade="save-update"
13             inverse="true">
14             <key column="course id"></key>
15             <many-to-many class="Model.Student" column="student id"></many-to-many>
16         </set>
17     </class>
18 </hibernate-mapping>

```

和 Student.hbm.xml 相似，这里在第 12 行到第 15 行，同样是通过 Set 在 Course 端定义了和 Student 的对应关系，并在第 12 行通过 table，定义了通过 students_courses 表来存储学生和课程之间的多对多关系。在第 13 行，定义了学生在 students_courses 表里，用 course_id 来关联 Course 端。在第 14 行，用 many-to-many 以及 class 来说明 Course 端是和 Student 端相关联。

请注意，在 Student.hbm.xml 和这里，cascade 都是 save-update，而不是像之前那样设置成 delete。因为这里是多对多关联，比如一个学生转学了，我们需要删除这个学生，但该学生所选的课程依然有其他学生在上，所以我们就不能直接删除了。从业务上讲，多对多关联中的 cascade 一般不会轻易地设置成 delete。

再来看下在 HibernateMain 这个类使用学生和课程的情况。

```

1 //省略必要的 package 和 import 代码
2 public class HibernateMain {
3     public static void main(String[] args) {
4         StandardServiceRegistry registry = null;

```

```

5         SessionFactory sessionFactory = null;
6         try{
7             registry = new
StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
8             sessionFactory = new
MetadataSources(registry).buildMetadata().buildSessionFactory();
9         }catch(Exception ex){
10             ex.printStackTrace();
11             StandardServiceRegistryBuilder.destroy(registry);
12         }
13 //得到 Session 对象
14         Session session = sessionFactory.openSession();
15 //创建三个学生对象
16         Student s1 = new Student();
17         s1.setStudentID("1");
18         s1.setStudentName("Peter");
19         Student s2 = new Student();
20         s2.setStudentID("2");
21         s2.setStudentName("Mike");
22         Student s3 = new Student();
23         s3.setStudentID("3");
24         s3.setStudentName("John");
25 //创建 3 个课程
26         Course c1 = new Course();
27         c1.setCourseID("c1");
28         c1.setCourseName("Math");
29         Course c2 = new Course();
30         c2.setCourseID("c2");
31         c2.setCourseName("Java");
32         Course c3 = new Course();
33         c3.setCourseID("c3");
34         c3.setCourseName("C#");
35         Set<Course> computerCourses = new HashSet<Course>();
36         computerCourses.add(c2);
37         computerCourses.add(c3);
38         Set<Course> mathCourses = new HashSet<Course>();
39         mathCourses.add(c1);
40 //让学生选修课程
41         s1.setCourses(computerCourses);
42         s2.setCourses(computerCourses);
43         s3.setCourses(mathCourses);
44 //保存三个学生以及他们选修的课程
45         session.save(s1);
46         session.save(s2);

```

```

47     sessio.save(s3);
48     sessio.flush();

```

执行到这里，我们能在数据库的 Student 表里，看到有三个学生。

StudentID	Name
1	Peter
2	Mike
3	John

在 Course 表里，看到有三门课程。

CourseID	CourseName
c1	Math
c2	Java
c3	C#

同时能在描述关联关系的 Students_Courses 表里看到学生和课程的对应关系。

student_id	course_id
1	c2
1	c3
2	c2
2	c3
3	c1

```

49     sessio.delete(s3);
50     sessio.flush();

```

当我们在第 49 行删除 s3 这个学生后，会发现第一在 Student 表里，没有了 s3 这个学生。而在 Course 表里，课程依然是 3 门，因为没有设置级联删除，删除学生不会影响到课程。而在描述对应关系的表里，“s3”同学选课这层关系被删除了。

student_id	course_id
1	c2
1	c3
2	c2
2	c3

```

51     sessio.delete(c1);
52     sessio.flush();
53     sessio.close();
54     sessionFactory.close();
55 }

```

```
56 }
```

同样在第 51 行删除 c1 这门课程后，会发现在 Course 和 Students_Courses 表里，相关的记录被删除了。

这里请大家注意，如果在第 49 行删除的是 s1 这个学生，同时在第 51 行删除 c1 这门课程，则会出现如下错误。

```
deleted object would be re-saved by cascade (remove deleted object from associations):
[Model.Course#c1]
```

这种错误在一对多和多对多级联操作时很容易出现。原因是，当删除 s1 这个学生后，在关联表里能看到 c1 这门课程还在和 3 号同学（也就是 s3 对象）关联，在消除这层关联关系之前，是无法删除 c1 这个课程的。

student_id	course_id
2	c2
2	c3
3	c1

遇到这种情况时，我们见过有些程序员会直接删除 cascade 关系（或者是一对多、多对多关系）了事，但这样消除了保护机制后，会导致错误的数据。正确的做法是，先删除和 c1 关联的所有学生后，再删除这门课程。

8.1.7 多对多关联的注解实现方式

下面通过注解的方式来实现多对多关联。

代码位置	视频位置
code/第 8 章/HibernateSenorM2MAnnoDemo	视频/第 8 章/通过注解实现多对多关联

这里的表结构和用配置文件方式实现多对多关联的表结构一致。

修改点 1：在 Hibernate.cfg.xml 里，通过 mapping class 来指定映射文件。

```
1 <!-- 添加实体类的映射文件-->
2 <mapping class="Model.Student" />
3 <mapping class="Model.Course" />
```

修改点 2，没有了 Student.hbm.xml 和 Course.hbm.xml 这两个描述映射关系的文件，而是通过注解的方式在两个业务实体类对象里描述映射关系。

Student.java 的代码如下。

```

1  //省略 package 和 import 代码
2  //通过@Entity 和@Table 指定该类和学生表相对应
3  @Entity
4  @Table(name="Student")
5  public class Student {
6  //指定主键
7  @Id
8  @Column(name="StudentID")
9  private String studentID;
10 @Column(name="StudentName")
11     private String studentName;
12 //通过@ManyToOne 来描述多对多关联
13 @ManyToOne
14 @Cascade(value = {org.hibernate.annotations.CascadeType.SAVE_UPDATE })
15 @JoinTable(name="Students_Courses",joinColumns={@JoinColumn(name="StudentID")
16     },inverseJoinColumns={@JoinColumn(name="CourseID") })
17     private Set<Course> courses;
17 //省略必要的 get 和 set 方法
18 }

```

在第 16 行，通过 Set<Course>这个类型的 courses 来描述该学生所对应的课程集合。

第 13 行到第 15 行的注解会作用在第 16 行的对象上，其中第 13 行指明是多对多关联。在第 14 行，指定了在 Student 端的级联关系是 SAVE_UPDATE，这和用配置文件实现多对多关联一致。

关键是在第 15 行，是通过@JoinTable 里的 name 来指定存储多对多关系的表的名字，这里是 Students_Courses，通过 joinColumn 来指定在 Students_Courses 这个关联表里，通过 StudentID 字段来关联到本端（也就是 Student 端），通过 inverseJoinColumns 来指定在 Students_Courses 表里关联到另一端（和学生端对应的是 Course 端）的字段。

修改后的 Course.java 代码如下。

```

1  //省略必要的 package 和 import 代码
2  //本业务模型关联到的 Course 这个数据表
3  @Entity
4  @Table(name="Course")
5  public class Course {
6  //定义各属性和数据表里各字段的映射关系
7  @Id
8  @Column(name = "CourseID")
9  private String courseID;
10 @Column(name = "CourseName")
11     private String courseName;

```

```
12 //同样是通过 ManyToMany 定义多对多关系
13 @ManyToMany
14 @Cascade(value = {org.hibernate.annotations.CascadeType.SAVE_UPDATE })
15 @JoinTable(name="Students_Courses", joinColumns={@JoinColumn(name="CourseID") }
16     ,inverseJoinColumns={@JoinColumn(name="StudentID") })
17     private Set<Student> students;
17 //省略 get 和 set 方法
18 }
```

这里是通过第 16 行的 `Set<Student>` 类型的 `students` 对象来存储该门课程所选修的学生集。

同样,在第 13 行,通过 `ManyToMany` 来定义多对多关联,在第 14 行指定了 `SAVE_UPDATE` 这个级联关系。

在第 15 行,通过 `@JoinTable` 来指定保存 `Student` 表和 `Course` 表对应关系的表名,这需要和 `Student.java` 这个业务类里的表名一致。同样是通过 `JoinColumn` 和 `inverseJoinColumns` 来定义关联表里和本端（也就是 `Course` 端）以及另一端（和 `Course` 端对应的是 `Student` 端）的对应字段。

这里没有修改 `HibernateMain.java` 这个类,所以调用后的结果和 8.1.5 节的完全一致。

8.1.8 关联性操作里的 cascade 取值

前文在介绍关联性操作时,看到大量描述级联操作的 `cascade` 和描述谁来维护关联关系的 `inverse`。

是否使用这两个关键字来管理关联性,不同的项目有不同的做法,所以在看到不用级联操作的情况时不要感到不可思议。

比如一个项目比较简单,数据来源也比较简单,在插入数据时一般不会有重复数据（比如有多个 ID 相同的账号）,也不会有非法的冲突数据（比如一个人有两个相同 ID 的银行账号）,那么这时就可以让 `Hibernate` 来管理关联性操作,进行级联插入删除等。

但有些项目,他们的数据来源比较复杂,可以由其他渠道通过 `xls` 等方式导入数据,或者由多个团队导入数据,而各团队导入数据时的标准不同。总之,在输入数据时比较难以控制数据的准确性,如果用级联操作,就会产生大量的异常,而项目组就会两害相逢取其轻,宁可在表里出现些非法数据,宁可用定时跑的程序清理这些非法数据,也不会用这些级联操作。

在配置文件中,经常用到 `cascade` 的选项分别是: `all`、`delete`、`none`、`save-update` 和 `delete-orphan`。

下面通过表 8.12 来总结下它们的适用范围。

表 8.12 c ascade 选值的含义

所选值	含义	项目中的使用要点
none	不进行级联操作，这个是默认值	这时项目组就需要自己管理数据的关联性，自己清理不一致的数据
save-update	在执行 save、update 或 saveOrUpdate 时进行关联操作	在多对多的场景下，一般使用这个，而不会使用 delete
delete	在执行 delete 时进行关联操作	使用时需谨慎，要确保另外一方是应该被删除的，因为使用这个值时可能会有误删除的情况
delete-orphan	删除所有和当前对象解除关联关系的对象	这个比较少使用，因为在项目里，解除关联关系的场景比较少
all	包含 save-update 以及 delete 的行为	由于包含了 delete 的行为，所以要避免误删除的情况

在注解方式的实体类里，可以通过类似如下代码的方式，用 CascadeType 来定义实体类之间的级联方式。

请注意 CascadeType 是包含在两个 jar 包里的：第一是 org.hibernate.annotations.CascadeType，第二是 javax.persistence.CascadeType。

```
@ManyToMany@Cascade (value = {org.hibernate.annotations.CascadeType.SAVE_UPDATE })
@JoinTable (name="Students_Courses", joinColumns={@JoinColumn (name="StudentID")
}, inverseJoinColumns={@JoinColumn (name="CourseID") })
```

表 8.13 注解方式中 CascadeType 取值的含义

包名	取值	含义
org.hibernate.annotations.CascadeType SAVE_UPDATE	SAVE_UPDATE	和表 8.12 里的一致
org.hibernate.annotations.CascadeType DELETE	DELETE	和表 8.12 里的一致
org.hibernate.annotations.CascadeType LOCK	LOCK	把 lock 操作级联到被关联的实例，不怎么用
org.hibernate.annotations.CascadeType REPLICATE	REPLICATE	把针对数据库的 replicate()操作级联到被关联的对象
org.hibernate.annotations.CascadeType EVICT	EVICT	当对象被 Hibernate Session 中的 evict()调用时，Hibernate 从持久化上下文清除被关联的对象
org.hibernate.annotations.CascadeType DELETE_ORPHAN	DELETE_ORPHAN	和表 8.12 里的一致
javax.persistence.CascadeType PERSIST	PERSIST	一方对象被持久化，另一方也会被持久化
javax.persistence.CascadeType MERGE	MERGE	一方新增或者变化，会级联到另一方对象
javax.persistence.CascadeType REMOVE	REMOVE	一方被删除，另一方会级联删除
javax.persistence.CascadeType REFRESH	REFRESH	当对象被 refresh()时，Hibernate 从数据库中重新读取被关联对象的状态

8.1.9 通过 inverse 设置外键控制权

Inverse 的英文含义是反转，在 Hibernate 中用来决定是由哪方来维护两个业务实体类之间的关联关系，具体而言，就是由哪方去设置这个被外键约束的字段值。

它的默认值是 false，也就是说，本端（比如 inverse=false 写在学生端，那么本端是学生，另外一方是课程）不“反转控制权”，这句别扭的话的另外一种说法是，本端维护关联关系。如果两边都不写，那么两端都维护。这样会造成问题，即新时因为两端都控制关系，因此可能会导致重复更新。

注意，inverse 仅仅是指定由谁来设置外键值，而不是用来设置级联操作，级联操作的方式由 cascade 来负责，很多人会混淆它们的含义和用法。

下面通过一个一对多的例子来看看 inverse 对数据操作的影响。

代码位置	视频位置
code/第 8 章/HibernateSeniorInverseDemo	视频/第 8 章/通过 inverse 设置外键控制权

这个例子所用到的数据表是如表 8.7 和 8.8 两个表所示的 Person 表和 Card 表，其中 Person 表里包含 ID、Name 和 Phone 字段，而 Card 表里包含了 CardID、PersonID、Bank 和 balance 四个字段。

在 hibernate.cfg.xml 文件里，通过 mapping resource 来指定对应的映射文件

```
1  <!-- 添加实体类的映射文件-->
2  <mapping resource="Model/Card.hbm.xml" />
3  <mapping resource="Model/Person.hbm.xml" />
```

在 Person.hbm.xml 里，用如下代码来指定人（一方）和卡（多方）的关系，其中在人这一端，inverse 是 true。

```
1  <set name="cards" cascade="save-update" inverse="true">
2  <key column="PersonID"/>
3  <one-to-many class="Card"/>
4  </set>
```

在 HibernateMain.java 里，通过如下关键代码插入了一个人的信息。

```
1  //创建一张卡
2  Card card1 = new Card();
3      card1.setCardID("Card123");
4      card1.setPersonID("Person123");
5      card1.setBank("Citi");
6      card1.setBalance(100f);
```

```

7 //初始化一个人
8     Person person = new Person();
9     person.setID("Person123");
10    person.setName("Peter");
11    person.setPhone("123456");
12 //在 cards 这个 set 里加入 card1
13    Set<Card> cards = new HashSet<Card>();
14    card.add(card1);
15    person.setCards(cards);
16 //保存人的信息
17    session.save(person);
18    session.flush();

```

运行后，在输出信息里能看到如下两条相关的插入语句：一条是插入 Person 信息，另一条是插入 card 信息。

```

1 Hibernate: insert into Person (Name, Phone, ID) values (?, ?, ?)
2 Hibernate: insert into Card (PersonID, Bank, Balance, Person, CardID) values
  (?, ?, ?, ?, ?)

```

当把 person.hbm.xml 里的 inverse 设置成 false 时，能看到相关的语句里会多出一句 update 语句。

```

1 Hibernate: insert into Person (Name, Phone, ID) values (?, ?, ?)
2 Hibernate: insert into Card (PersonID, Bank, Balance, Person, CardID) values
  (?, ?, ?, ?, ?)
3 Hibernate: update Card set PersonID=? where CardID=?

```

原因是，当设置 inverse 为 true 时，Person 这一端反转外键控制权，也就是由 Card 这端来管理外键，而在代码里我们仅仅是插入了 Person，没有插入 Card，所以就没有更新两个外键（PersonID 和 CardID）的操作。相反，当 inverse 为 false 时，管理外键控制权是在 Person 端，那么当插入 Person 时，Hibernate 就需要额外的一句 update 语句来管理外键了。

在一对多的例子里，inverse 不论取什么值，对结果都没有影响，所以很容易让人忽视它的作用。

在一对多的例子里，一般是让多方来管理外键控制权，比如一个人有 100 张卡，那么如果由 Person 方来管理的话，无形中可能会多出 100 个 update 操作，效率上就不太好了。

如果在一对多案例中，inverse 只是影响效率的话，那么在多对多的例子中，inverse 的设置就可能影响到数据。

在上文的 HibernateSenorM2MDemo 案例中，是用多对多来管理学生选课的情况。在 Student.hbm.xml 里，描述多对多关系的语句里可加上 inverse=“true”的语句。

```
1 <set name="courses" table="students_courses" inverse="true"
  cascade="save-update">
2 <key column="student_id"></key>
3 <many-to-many class="Model.Course" column="course_id"></many-to-many>
4 </set>
```

在 `Course.hbm.xml` 里，不加任何关于 `inverse` 的语句，也就是说，在 `Student` 端反转外键控制权，把控制权交到 `Course` 端。

在 `HibernateMain` 这个类里，通过如下代码让 `s1` 学生选修计算机课程。

```
1 //设置一个学生，学号是 1
2 Student s1 = new Student();
3 s1.setStudentID("1");
4 s1.setStudentName("Peter");
5 //设置多个课程
6 Course c1 = new Course();
7 c1.setCourseID("c1");
8 c1.setCourseName("Math");
9 Course c2 = new Course();
10 c2.setCourseID("c2");
11 c2.setCourseName("Java");
12 Course c3 = new Course();
13 c3.setCourseID("c3");
14 c3.setCourseName("C#");
15 //设置计算机课程这个 Set
16 Set<Course> computerCourses = new HashSet<Course>();
17 computerCourses.add(c2);
18 computerCourses.add(c3);
19 //让 s1 这个学生选修计算机课程（也就是 c2 和 c3 课程）
20 s1.setCourses(computerCourses);
21 //保存 s1
22 session.save(s1);
23 session.flush();
```

执行结果是，虽然能在 `Student` 和 `Course` 表里看到相关的学生和课程的记录，但在关键的描述学生选课关联表（`students_courses`）里，看不到任何关联记录。原因是已经通过设置 `inverse` 把外键管理权交给 `Course` 方了，这里仅仅是保存学生，并没有保存课程，所以没有插入外键的动作。如果要在 `students_courses` 表里插入外键关联，就需要在 `person.hbm.xml` 里设置 `inverse` 的值为 `false`。

所以，在多对多关联里，设置错了 `inverse` 值会导致结果出错，请大家根据具体项目的情况适当设值。

8.2 Hibernate 里的事件机制

通过 Hibernate 提供的 ORM 相关方法，确实只需要操作业务模型，比如当调用 save 和 delete 之类的方法后，Hibernate 会自动把这些操作对应地作用到数据表上。

此外，还可以通过 Hibernate 里的事件机制，监听或者拦截某些事情，并且在这些事件发生时执行已定义好的代码，由此来扩展 Hibernate 本身提供给我们的功能。

Hibernate 的事件机制框架由两部分组成：第一是拦截器机制，它能拦截特定的动作拦截，并能在这些动作发生时调用我们自己定义的方法；第二是事件系统，通过它我们能重写 Hibernate 的事件监听器。

8.2.1 在拦截器里放一些通用性的代码

在 Hibernate 里，也能像 Spring 那样，实现拦截器里已经封装好的回调函数。这样，在执行 Hibernate 的一些动作（比如执行事务或保存数据等）时，就能自动地执行定义的动作。

先通过如下例子来看一下如何使用拦截器。

代码位置	视频位置
code/第 8 章/HibernateSeniorInceptorDemo	视频/第 8 章/在 Hibernate 里实现拦截器

这里用到的数据表是 UserInfo，它的结构如表 8.14 所示。

表 8.14 UserInfo 表结构

字段名	类型	含义
UserID In	t	主键，用户的 ID
UserName varchar		用户名
Pwd var	char	密码
UserType V	archar	用户类型

首先来配置 hibernate.cfg.xml，其中和数据库连接的属性和前文一致，这里通过如下的代码来引入业务实体类。

```
<mapping class="Model.UserInfo" />
```

第二步，编写包含业务实体的 UserInfo 类，这里将通过注解的方式配置数据表和业务实体之间的映射关系。

```
1 //省略必要的 package 和 import 方法
2 //通过 Entity 和 Table 来定义映射到 UserInfo 这个数据表
```

```
3  @Entity
4  @Table(name="userinfo")
5  public class UserInfo {
6      //定义各属性和数据表各字段的映射关系
7      @Id
8      @Column(name = "UserID")
9      private int userID;
10     @Column(name = "username")
11     private String userName;
12     @Column(name = "pwd")
13     private String pwd;
14     @Column(name = "usertype")
15     private String userType;
16     //省略必要的get和set方法
17 }
```

第三步，在 `UserInfoInterceptor` 这个类里，编写与拦截器相关的方法。

```
1  //省略必要的package和import方法
2  //EmptyInterceptor里包含了针对各动作的回调函数
3  public class UserInfoInterceptor extends EmptyInterceptor {
4      //当事务开始执行后，会调用这个回调方法
5      @Override
6      public void afterTransactionBegin(Transaction tx){
7          //这里仅仅加了打印语句，在项目里可以加上监控事务避免死锁的代码
8          System.out.println("begin transaction");
9      }
10     //当事务执行结束后，调用这个方法
11     @Override
12     public void afterTransactionCompletion(Transaction tx){
13         //通过getStauts方法得到事物的状态
14         if ( tx.getStatus() == TransactionStatus.COMMITTED) {
15             //也可以加些打印事务执行时间等监控状态的语句
16             System.out.println("after transaction completion");
17         }
18     }
19     //当执行save方法的时候会调用这个方法
20     //entity参数表示被save的对象
21     //state表示被save对象的各属性值
22     //propertyNames表示被save对象的各属性名
23     //Type表示被save对象的各属性的类型
24     @Override
25     public boolean onSave(Object entity, Serializable id, Object[] state, String[]
propertyNames, Type[] types){
26         System.out.println("on save");
27     }
28 }
```

```

27     //判断被 save 的对象是否是 UserInfo 类型的
28     if (entity instanceof UserInfo)
29     {
30         for (int index=0;index<propertyNames.length;index++)
31         {
32             //通过 for 循环，遍历 porpertyNames 属性，找到名为“ userType ”的属性
33             if ("userType".equals(propertyNames[index]))
34             {
35                 //在对应的 state 里，把 Vip 这种类型修改成 VVIP
36                 if (state[index].toString().equals("Vip"))
37                 {
38                     state[index] = "VVIP";
39                     return true;
40                 }
41             }
42         }
43     }
44     return false;
45 }
46 //当执行 delete 时，会调用这个方法
47 public void onDelete(Object entity, Serializable id, Object[] state, String[]
propertyNames, Type[] types){
48     System.out.println("on delete");
49     return
50 }
51 }

```

Hibernate 拦截器相关的方法定义在 `Interceptor` 这个接口里，如果通过实现这个接口（implements `Interceptor`）来定义在项目里的拦截器，那么就不得不在实现类里重写它的所有方法，即便有些方法在项目里用不到，那么也不得不写多个空方法，这种做法很不方便。

为了避免这种不大好看的写法，通常在第 1 行里中通过实现 `EmptyInterceptor` 类来定义拦截器。

在第 6 行和第 12 行里，重写了两个回调函数，它们会在执行事务的前后被调用。

在第 25 行，重写了 `onSave` 这个方法，它将在 `session.save` 方法调用时被执行。其中，把 `UserType` 为 `Vip` 的值更新成 `VVIP`。请注意，`onSave` 方法是 `boolean` 类型，如果返回是 `true`，那么在拦截器里的修改会作用到数据表里；反之，如果返回是 `false`，就不会作用到数据表里了。

在第 47 行，重写了 `onDelete` 方法，这个方法将在 `session.delete` 时被调用。

最后来通过 HibernateMain 这个类，看下拦截器里各方法的调用时间点。

```
1 //省略必要的 package 和 import 方法
2 public class HibernateMain {
3     public static void main(String[] args) {
4         // 初始化自定义的拦截器类
5         UserInfoInterceptor interceptor = new UserInfoInterceptor();
6         Configuration cfg = null;
7         SessionFactory sessionFactory = null;
8         //这里我们通过 Configuration 来创建 sessionFactory
9         try
10        {
11            cfg = new Configuration().configure();
12            sessionFactory =
13            cfg.setInterceptor(interceptor).buildSessionFactory();
14        }
15        catch (Throwable e)
16        {
17            throw new ExceptionInInitializerError(e);
18        }
19        //我们需要在 withOption.interceptor 方法里，引入自定义拦截器
20        Session session =
21        sessionFactory.withOptions().interceptor(interceptor).openSession();
22        Transaction tx = session.beginTransaction();;
```

在第 19 行里，我们开启了一个事务，此时，定义在 UserInfoInterceptor 类里的相关的回调函数 afterTransactionBegin 会被执行。

```
20 //创建一个 UserInfo 业务实体，请注意它的 UserType 是 Vip
21 UserInfo user = new UserInfo();
22 user.setUserID(3);
23 user.setUserName("Peter");
24 user.setPwd("456");
25 user.setUserType("Vip");
26 try{
27 session.save(user);
28 tx.commit();
29 }
30 catch (Exception e)
31 {
32 e.printStackTrace();
33 tx.rollback();
34 }
```

在第 27 行通过 session.save 方法保存了一个 user 对象，此时会自动调用定义在

UserInfoInterceptor 类里的 onSave 方法，虽然在代码里设置的 UserType 是 Vip，但是在 onSave 方法里，会把它更新成 VVIP。

如果在 onSave 的如下方法里，修改更新部分的 UserType 代码，把返回值修改成 false，那么虽然也会调用 onSave 这个回调方法，但其中的修改就不会作用到数据表里了。

```
if (state[index].toString().equals("Vip"))
{
    state[index] = "VVIP";
    return false; //原来是 true
}
```

在第 28 行，我们通过 commit 方法提交了一个事务，此时会自动调用 UserInfoInterceptor 里的 afterTransactionCompletion 方法。

```
35 List<UserInfo> users = new ArrayList<UserInfo>();
36 String hqlstr = "from Model.UserInfo";
37 Query query = session.createQuery(hqlstr);
38 users = query.list();
39 System.out.println("users size is:" + users.size());
40 System.out.println("UserName" + "\t" + "UserType");
41 //通过打印语句，我们能确认 Vip 确实被更新成 VVIP
42 for(UserInfo u : users){
43     System.out.println(u.getUserName() + "\t" + u.getUserType());
44 }
45 //这里的 delete 操作会自动触发拦截器类里的 onDelete 方法
46 session.delete(user);
47 session.flush();
48 session.close();
49 sessionFactory.close();
50 }
51 }
```

下面来总结下使用拦截器的一般方法。

第一，自定义一个拦截器类，该类需要继承（extends）EmptyInterceptor 个类。

第二，在自定义拦截器里，通过重写方法来定义针对特定事件（比如 save 或者 delete）的操作方法。

第三，在使用时，通过如下的代码，在创建 Session 时指定需要引入的拦截器类。

```
Session session =
sessionFactory.withOptions().interceptor(interceptor).openSession();
```


第四，在上述代码里，是通过 session.withOptions 来引入拦截器，此时，拦截器只作用在当前 Session 上。此外，我们还可以用如下代码，通过 SessionFactory 设置拦截器，这样，拦截器就能作用在该 SessionFactory 所生成的所有 Session 上了。

```
try
{
    cfg = new Configuration().configure();
    //通过 setInterceptor 方法在 SessionFactory 上引入拦截器
    sessionFactory = cfg.setInterceptor(interceptor).buildSessionFactory();
}
catch (Throwable e)
{
    throw new ExceptionInInitializerError(e);
}
//通过 sessionFactory 创建的 session 都会拦截器影响到了
Session session = sessionFactory.openSession();
```

刚才只是实现了 EmptyInterceptor 里的一部分方法，下面通过表 8.15 来说明该类里其他的一些常用方法。

表 8.15 拦截器相关的方法归纳表

方法名	执行时间点
onLoad(Object entity, Serializable id, Object[] state, String[] propertyNames, Type[] types)	该方法在持久化对象初始化之前加载,这个持久化对象处于刚被创建的状态（对象的属性值都未赋值）
preFlush()	该方法在调用 Session 对象的 flush()方法之前被调用
postFlush(Iterator entities)	该方法在调用 Session 对象的 flush()方法之后被调用
onFlushDirty(Object entity, Serializable id, Object[] currentState, Object[] previousState, String[] propertyNames, Type[] types)	当调用 Session 对象的 flush()方法进行脏数据检查时，如果发现持久化对象的状态发生了改变，则会调用该方法

不是每个项目都会用到事件机制，所以我们在面试候选人时，这点属于加分项，毕竟在 Hibernate 里用到的技术点越多，他们对 ORM 的了解可能就越多。

另外，我们不会在意他们在各回调方法里放了什么业务，只要放的业务不太离谱就行。曾有人告诉我们，他们的项目里会在回调函数里放对事物的回滚操作，当我们细问的时候又说得自相矛盾，这种就还不如说没用过。关于事件机制，我们一般会核实如下知识点。

第一，你是用哪种方式？是用到拦截器还是后文提到的监听器。

第二，你是怎么用的？比如在项目里是怎么实现拦截器的？比如怎么继承什么类？重写了哪些回调函数等？这里我们会落实些代码细节。

第三，为什么要用？比如用了以后能给你的项目带来哪些好处？或者你项目里有哪些需求最好用拦截器实现？

第四，拦截器有基于 Session 和 SessionFactory 的，你的项目是用哪一种？为什么要用这种？

8.2.2 事件系统和监听器

通过监听器机制来监听 Hibernate 的各动作并执行对应的回调函数。监听器的实现原理是，为每一个事件（比如更新、添加、删除等）注册一个或多个监听器，一旦事件发生，则通知所有相关的监听器执行对应的回调方法。

和上文提到的拦截器相比，监听器可以获取被监听对象修改前和修改后的状态值，而且，监听器可以通过 Event 对象得到 Session 对象。

下面通过例子来了解监听器的用法。

代码位置	视频位置
code/第 8 章/HibernateSeniorEventDemo	视频/第 8 章/Hibernate 监听器的讲解

这里用到的数据表是表 8.14 所示的 UserInfo，它包含 UserID、UserName、Pwd 和 UserType 四个字段。

首先编写 UserInfo.java 这个业务实体类，它和上文拦截器部分的代码一致，所以就不再重复给出了。

其次，编写实现监听器的 UserInfoListener 类。

```
1 //省略必要的 package 和 import 代码
2 //这里实现了三个监听器类，用于监听 SaveOrUpdate, Update 和 Delete 事件
3 public class UserInfoListener implements SaveOrUpdateEventListener,
4     PostUpdateEventListener ,PostDeleteEventListener {
5     //一旦发生 session.SaveOrUpdate，就会触发这个方法
6     @Override
7     public void onSaveOrUpdate(SaveOrUpdateEvent event)
8     {
9         Systemout.println("Save Or Update");
10        Objectobj = event.getObject();
11        Systemout.println(event.getObject());
12        if(obj instanceof UserInfo)
13        {
14            ((UserInfo) obj).setUserType("VVIP");
15        }
16    }
17 }
```

```

14     }
15 }

```

这个方法是在 `SaveOrUpdateEventListener` 这个接口里的。在第 9 行，通过 `event.getObject` 方法得到了待 `SaveOrUpdate` 的对象，从第 10 行的输出中能看到它是 `UserInfo` 类型的。在第 13 行，通过 `obj`，把待更新对象的 `UserType` 设置成 `VVIP`，这样，最终到数据表里的用户就是 `VVIP` 了。

```

16 //这个方法将在 Session 的 update 方法后被触发
17 @Override
18 public void onPostUpdate(PostUpdateEvent event) {
19     System.out.println("after Update");
20     System.out.println(event.getEntity().getClass().getName());
21     //得到并输出待 update 的对象名
22     Object obj = event.getEntity();
23     System.out.println(obj); //输出还是 UserInfo 对象
24     if(obj instanceof UserInfo)
25     { //更新 UserType 属性
26         ((UserInfo) obj).setUserType("VVIP");
27     }
28     //通过 for 循环，输出更新前后的值
29     for (int i = 0; i < event.getState().length; i++) {
30         // 更新前的值
31         Object oldValue = event.getOldState()[i];
32         // 更新后的新值
33         Object newValue = event.getState()[i];
34         //属性名
35         String propertyName = event.getPersister().getPropertyNames()[i];
36         System.out.println("property" + propertyName + "change from " + oldValue + "
to " + newValue);
37     }
38 }

```

这个方法是在 `PostUpdateEventListener` 接口里的。

在第 29 行通过 `event.getState().length` 得到更新后的属性的数组的长度。在随后的 `for` 循环的第 31 行中，通过 `event.getOldState()[i]` 来得到更新前的值。在第 33 行，通过 `event.getState()[i]` 得到更新后的值。在第 35 行，通过 `event.getPersister().getPropertyNames()[i]` 得到该类 (`UserInfo`) 的各属性名，并在第 36 行里统一输出。

```

39 //在提交事务之后会被触发，这里并没有实现这个方法
40 @Override
41 public boolean requiresPostCommitHandling(EntityPersister arg0) {
42     return false;

```

```

43 }
44 //这个方法将在 Session 的 delete 方法后被触发
45 @Override
46 public void onPostDelete(PostDeleteEvent event) {
47     System.out.println("after delete");
48     System.out.println(event.getEntity().getClass().getName());
49     for (int i = 0; i < event.getDeletedState().length; i++)
50 {
51         // 删除前的值
52         Object deletedValue = event.getDeletedState()[i];
53         //被删除的属性名
54         String propertyName = event.getPersister().getPropertyNames()[i];
55         System.out.println("property" + propertyName + " value " + deletedValue);
56     }
57 }
58 }

```

onPostDelete 是封装在 PostDeleteEventListener 接口里的，在第 48 行，打印了被删除对象的类名，在第 49 行到第 56 行里，用 for 循环，通过遍历 event.getDeletedState 对象，依次输出被删除的属性名和属性值。

第三步，需要在 hibernate.cfg.xml 这个配置文件里，配置监听器，当发生 Hibernate 事件时，监听器里定义的回调方法就能自动执行了。下面列出关键的 session-factory 部分代码。

```

1  <session-factory>
2  <property name="hibernate.connection.driver_class">
3      com.mysql.jdbc.Driver
4  </property>
5  //省略其他针对 MySQL 连接的配置
6  <mapping class="Model.UserInfo" />
7  <event type="save-update">
8  <listener class="listener.UserInfoListener"/>
9  </event>
10 <listener class="listener.UserInfoListener" type="post-update"/>
11 <listener class="listener.UserInfoListener" type="post-delete"/>
12 </session-factory>

```

在第 6 行，通过 mapping class 来指定引入 UserInfo 这个和数据表的映射类。在第 7 行到第 11 行里，通过两种方式配置了监听器。

从第 7 行到第 9 行，通过 event 元素，定义了一旦发生 save-update 类型的事件，就调用 UserInfoListener 这个类里的方法。由于这个类实现了 SaveOrUpdateEventListener 接口，而且实现了该接口里的 onSaveOrUpdate 方法，因此一旦有该类型的事件，就会自动调用

onSaveOrUpdate 这个方法。

在第 10 行和第 11 行里，通过 listener 元素，定义了两类（post-update 和 post-delete）监听器，同样指定了这两类监听器的处理类是 UserInfoListener。由于该类同时又实现了这两类监听器的接口，所以发生这两类事件时，也会自动调用相应的方法。

最后通过 HibernateMain 这个类，看下发生事件时，监听器是如何工作的。

```
1  //省略必要的package和import代码
2  public class HibernateMain {
3      public static void main(String[] args) {
4          Configuration cfg = null;
5          SessionFactory sessionFactory = null;
6          // 这里我们通过 Configuration 来创建 sessionFactory
7          try
8          {
9              cfg = new Configuration().configure();
10             sessionFactory = cfg.buildSessionFactory();
11         }
12         catch (Throwable e)
13         {
14             throw new ExceptionInInitializerError(e);
15         }
16         //得到 session
17         Session session = sessionFactory.openSession();
18         Transaction tx = session.beginTransaction();
19         UserInfo user = new UserInfo();
20         user .setUserID(3);
21         user .setUserName("Peter");
22         user .setPwd("456");
23         user .setUserType("Vip");
24         try{
25             session.saveOrUpdate(user);
26             tx.commit();
27         }
28         catch(Exception e)
29         {
30             e.printStackTrace();
31             tx.rollback();
32         }
```

由于在第 25 行中发生了 session.saveOrUpdate 操作，所以会自动调用 UserInfoListener 类里的 onSaveOrUpdate 方法。

```
33     List<UserInfo> users = new ArrayList<UserInfo>();
```

```
34     String hqlstr = "from Model.UserInfo";
35     Query query = session.createQuery(hqlstr);
36     users = query.list();
37     System.out.println("users size is:" + users.size());
38     System.out.println("UserName" + "\t" + "UserType");
39     for(UserInfo u : users){
40         System.out.println(u.getUserName() + "\t" + u.getUserType());
41     }
```

在 UserInfoListener 类的 onSaveOrUpdate 方法里，把 userType 修改成 VVIP 值，通过第 40 行的输出语句，能看到这个修改。

```
42     user.setUserName("Mike");
43     session.update(user);
44     session.flush();
```

第 43 行的 update 操作会触发 UserInfoListener 类的 onPostUpdate 方法。

```
45     session.delete(user);
46     session.flush();
47     session.close();
48     sessionFactory.close();
49 }
50 }
```

第 45 行的 delete 操作会触发 UserInfoListener 里的 onPostDelete 方法。

这个例子实现了针对三种事件的监听器，如果有其它方面的需求，还可以通过如下方式监听其它事件。

第一，在对应的类里（比如 UserInfoListener）实现如表 8.16 所述的接口，代码如下所示。

```
public class UserInfoListener implements 实现对应的接口
```

第二，在 hibernate.cfg.xml 里，通过 event 或 listener 属性注册监听器。

```
<event type="事件类型"><listener class="对应的类"/></event>
```

或者

```
<listener class="对应的类" type="事件类型"/>
```

表 8.16 常用的监听器接口和事件类型对应表

在类里需要实现的接口	注册时用到的事件类型
MergeEventListener m	erge
DeleteEventListener delete	
PersistEventListener persist	

续表

在类里需要实现的接口	注册时用到的事件类型
EvictEventListener evict	
FlushEventListener flush	
LoadEventListener load	
RefreshEventListener ref	resh
ReplicateEventListener replicate	
SaveUpdateEventListener save-	update
PreLoadEventListener pr	e-load
PreUpdateEventListener pr	e-update
PreDeleteEventListener pre-delete	
PreInsertEventListener pre-insert	
PostLoadEventListener post-load	
PostUpdateEventListener post-	update
PostInsertEventListener post-insert	
PostDeleteEventListener post-delete	

8.3 Hibernate 中的优化

对于初级程序员，这些优化技术属于加分项，对于有两到三年经验准备向高级程序员晋升的候选人，这些属于必问项，毕竟高级程序员应该有优化方面的意识。

有时候我们在看候选人简历时，明明描述了一些调优技术，但提问时却一脸茫然，这种情况属于会用不会说，甚至比不会更吃亏。

Hibernate 优化技术深不见底，高级程序员也需要不断总结，本章讨论的不仅是具体技术，而是从项目开发过程中总结出来的能让大家现在就能吸收的一些方向性和策略性的东西。

8.3.1 结合数据库大背景

虽然 Hibernate 能屏蔽掉数据库操作的细节，但在调优时，还是应该结合具体数据库的种类进行。对于不同类型的数据库（比如 MySQL、Oracle 或 DB2），各种调优参数在设置上应该不同，或者至少应通过测试找到合适的参数值。

策略 1，调整批处理的条数。

比如下面的 batch_size 参数是设定每次从数据库中取出的记录条数，一般设置为 30、50、

100。一般的说法是，Oracle 数据库的 JDBC 驱动默认值是 15，当设置为 30 或 50 时，性能会有明显提升，如果继续增大，超出 100，则性能提升就不明显了。

```
<prop key="hibernate.jdbc.batch_size">50</prop>
```

关于这点，比较好的描述是，在做项目前，新建 3 个表，往表里插入了几万条数据，然后写测试程序，对 3 个表进行关联查询，随后不断调整这个数值，通过打印运行时间和内存使用量等参数，查看哪个值最合适。通过对删除、更新、插入等操作的测试，最终发现对 Oracle 来说比较适合的值是 100。

策略 2，数据库本身的优化不可少。

数据库性能调优是一个综合工程，代码、数据库本身、Hibernate、甚至网络配置等因素都会影响性能。

之前介绍过一些针对数据库的优化策略，比如适当地建索引，如果数据量很大就少用关联，或者适当地建分区，这些措施如果得当，Hibernate 的性能自然就会好。

策略 3，如果操作很复杂，可以使用 SQL，必要时甚至可以绕过 Hibernate 直接用 JDBC。

通过 HQL 也能进行关联、子查询、group by 等操作。不少人在遇到比较复杂的需求时，就会写很复杂的 HQL 语句，这不仅影响代码可读性，还导致性能无法衡量。

遇到这种情况时，建议使用 SQL，因为可以事先在数据库层面，通过执行计划等工具，查看并优化 SQL，随后把优化过的语句放入 Hibernate。

如果在项目里有大量的 insert/delete/update 操作，则可以绕过 Hibernate，直接使用 JDBC，这样性能会更好，示例代码如下。

```
//通过 session 得到 Connection 对象
Connection conn=session.connection();
//创建 PreparedStatement
PreparedStatement stmt=conn.prepareStatement("update ...");
//随后通过 stmt 对象进行预处理和批处理
```

8.3.2 使用 SessionFactory 二级缓存

Hibernate 里一级缓存是 Session 级别的，当程序调用 Session 的 save()、update()、saveOrUpdate()、get()或 load()，以及 ist()、iterate()或 filter()方法时，如果在一级缓存中还存在有相应的对象，则 Hibernate 就不会去读数据库，而是直接从一级缓存中拿数据。

对于一级缓存，可以通过 evict(Object obj)方法，从缓存中清除指定的持久化对象，也可

以通过 clear()方法，清空缓存中所有的持久化对象。

Hibernate 的二级缓存是 SessionFactory 级别的，默认情况下，SessionFactory 是不会启用这个插件的。

请大家注意，在项目里，并不是所有的对象都适合放在二级缓存中。适合存放二级缓存里的数据种类是：

- ① 很少被修改的数据，比如历史数据。
- ② 参考数据，比如项目里的常数对应表。
- ③ 不会被并发访问的数据。

不适合放在二级缓存里的数据种类是：

- ① 经常被修改的数据。
- ② 绝对不允许出现并发操作的数据。

默认情况下，Hibernate 会使用 EhCache 作为二级缓存组件，但是大家在项目里也可以通过设置 hibernate.cache.provider_class 属性，指定使用的缓存种类。Hibernate 内置支持的二级缓存组件如表 8.17 所示。

表 8.17 Hibernate 所支持的二级缓存组件

组件 Prov	ider 类	类型
Hashtable or	g.hibernate.cache.HashtableCacheProvider	内存
EHCache or	g.hibernate.cache.EhCacheProvider	内存，硬盘
OSCache or	g.hibernate.cache.OSCacheProvider	内存，硬盘
SwarmCache or	g.hibernate.cache.SwarmCacheProvider	集群
JBoss TreeCache	org.hibernate.cache.TreeCacheProvider	集群

接下来通过一个使用 EHCache 的例子来观察一下二级缓存的一般用法。

代码位置	视频位置
code/第 8 章/HibernateSeniorCacheDemo	视频/第 8 章/在 Hibernate 里实现二级缓存

这里用到的数据表是 Accout，它的结构如表 8.18 所示。

表 8.18 Acc ount 表结构

字段名	类型	含义
ID In	t	自增长的主键
Name V	archar	账户拥有者的名字

续表

字段名	类型	含义
Bank V	archar	所属银行
Balance Float		余额

第一步，创建 Java 项目，并导入必要的 jar 包，然后导入 EhCache 的相关包：ehcache-core-2.4.3.jar 和 hibernate-ehcache-4.2.4.Final.jar。

第二步，在 hibernate.cfg.xml 里的 session-factory 元素中，配置二级缓存，关键代码如下。

```
1  <session-factory>
2  <!--配置二级缓存-->
3  <property name="hibernate.cache.use_second_level_cache">
4      true
5  </property>
6  <property name="hibernate.cache.region.factory_class">
7      org.hibernate.cache.ehcache.EhCacheRegionFactory
8  </property>
9  <property name="hibernate.cache.use query cache">
10     true
11 </property>
12 ...
13 </session-factory>
```

其中，第 3 行到第 5 行，设置启用二级缓存。第 6 行到第 8 行，设置二级缓存的支持包是 EhCacheRegionFactory，这次使用的是基于 EhCache 的缓存。第 9 行到第 11 行，是针对查询启用了缓存。

第三步，需要额外地在 src 目录下编写 ehcache.xml，用以配置 EhCache 二级缓存的具体属性值，代码如下。

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3      xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd">
4      <defaultCache maxElementsInMemory="1000" eternal="false"
5          timeToIdleSeconds="120" timeToLiveSeconds="120" overflowToDisk="true"
6          diskPersistent="false" memoryStoreEvictionPolicy="LFU" />
7      <diskStore path="d:\\ehcache"/>
8  </ehcache>
```

在第 3 行的 defaultCache 元素里，通过下面诸多的属性值定义了本程序里二级缓存的各参数。

- ① 通过 `eternal` 来设置缓存是否永久有效。
- ② 通过 `maxElementsInMemory` 设置了最大允许缓存的对象数，这里是 1000。
- ③ 通过 `overflowToDisk` 来定义：缓存对象达到最大数后，是否把缓存写到硬盘里。
- ④ 通过 `diskPersistent` 来定义是否支持硬盘持久化。
- ⑤ 通过 `timeToIdleSeconds` 来设置最后一次访问缓存的时间和失效时的时间间隔。
- ⑥ 通过 `timeToLiveSeconds` 来设置缓存自创建日期起至失效时的间隔时间。

⑦ 通过 `memoryStoreEvictionPolicy` 来设置缓存清空策略。清空需有三个参数：第一是 FIFO，表示先来的缓存先被清除。第二是 LFU，表示优先清除一直以来最少被使用的。第三是 LRU，表示优先清除最近最少使用的。

在第 5 行，通过 `diskStore` 的 `path` 属性指定了二级缓存在磁盘上的存放路径。

第四步，在 `Account.hbm.xml` 这个映射文件里，通过 `cache` 元素，指定针对 `Account` 二级缓存的策略。

```
1  <?xml version="1.0"?>
2  <!DOCTYPE hibernate-mapping PUBLIC
3  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
5  <hibernate-mapping package="Model">
6  <class name="Account" table="Account">
7  <!-- 应用 EhCache 二级缓存的策略 -->
8  <cache usage="read-only"/>
9  <id name="ID" column="ID" type="int">
10 <generator class="increment" />
11 </id>
12 <property name="name" type="java.lang.String">
13 <column name="Name" length="20" not-null="true"/>
14 </property>
15 <property name="bank" type="java.lang.String">
16 <column name="Bank" length="20"/></column>
17 </property>
18 <property name="balance" type="float">
19 <column name="Balance"/></column>
20 </property>
21 </class>
22 </hibernate-mapping>
```

在第 8 行，可知 `usage` 取值为 `read-only`，表示是只读缓存，也可以用 `Read-write` 表示读

写缓存。

最后通过 HibernateMain，来看下缓存的作用。

```

1 //省略必要的package和import代码
2 public class HibernateMain {
3     public static void main(String[] args) {
4         Configuration configuration = new Configuration();
5         try {
6             configuration.configure();
7             catch (HibernateException e) {
8                 e.printStackTrace();
9             }
10            ServiceRegistry serviceRegistry = new
ServiceRegistryBuilder().applySettings(configuration.getProperties()).buildSe
rviceRegistry();
11            SessionFactory sessionFactory =
configuration.buildSessionFactory(serviceRegistry);
12 //得到 Session 对象
13            Session session = sessionFactory.openSession();
14 //开启一个事务，加载 ID 为 1 的 Account 对象
15            Transaction tr1 = session.beginTransaction();
16            Account acc1 = (Account) session.get(Account.class, 1);
17            System.out.println("In First Session, name is " + acc1.getName());
18            tr1.commit();
19            session.close();
20 //在关闭 session 后，再启动新的 session 加载 ID 同样是 1 的对象
21            Session session2 = sessionFactory.openSession();
22            Transaction tr2 = session2.beginTransaction();
23            Account acc2 = (Account) session2.get(Account.class, 1);
24            System.out.println("In Another Session, name is " + acc2.getName());
25            tr2.commit();
26            session2.close();
27            sessionFactory.close();
28        }
29    }

```

二级缓存不是基于 Session，而是基于 SessionFactory 的。从这段代码的输出里可以看到，虽然在第 16 行和第 23 行中，分别用两个不同的 Session 对象执行了 get 方法，但 Hibernate 只向数据库发起了一次 select 请求，第二次 get 实际上是从二级缓存里得到数据的。

```

1 Hibernate: select account0_.ID as ID1_0_0_, account0_.Name as Name2_0_0_,
account0_.Bank as Bank3_0_0_, account0_.Balance as Balance4_0_0_ from Account
account0_ where account0_.ID=?
2 In First Session, name is Java

```

8.3.3 项目中常用的优化策略

除了上述的优化方法外，在项目中经常用到的优化策略还包括如下几项。

第一，在合适的时间点清除缓存。

一般会有大量的数据保存在 Session 的一级缓存中，如果缓存太大时就会拖累性能，所以在必要时可以使用 `session.clear()` 或者 `session.Evict(Object)`，清除全部缓存或某个对象。

第二，在不知道返回结果是多少时，谨慎使用 `list`，因为一旦返回数目过大，就会超出内存，导致 OOM 异常。在数据返回量大的情况下，可以通过分页的形式，用多次请求分批返回结果。

第三，在一对多和多对多的关系中，在加载时，可以使用延迟加载机制，这样能使不少对象只有在使用时才会被初始化，从而可以节省内存空间。

第四，在编写读数据的代码之前，对 SQL 或 HQL 语句要操作的表做个评估。比如要关联多个大表时，建议不要使用 HQL 的关联代码，而是使用 SQL。

最后，在性能调优方面，告诉大家一个通用的回答方法！①在遇到性能瓶颈时，可以在一些疑似性能问题的代码前后输出当前时间和当前内存使用量，由此判断出这段代码的运行时长和内存占用量。②在找到问题症结后，可以通过直接使用 JDBC API，或者通过观察执行计划来优化 SQL 语句。

8.4 在 Hibernate 方面我们面试的方式

在 Hibernate（或者其他 ORM 框架）方面，我们一般会把候选人分为两个级别：

第一是会用 Hibernate 干活，能在项目经理带领下做项目，如果需要的是初级程序员，那么达到这个标准就可以了。

第二个级别是针对高级程序员的，要求是不仅要熟悉 Hibernate 的用法，更重要的是，能根据项目里数据表的需求，适当地选用技术种类，而且，一旦出现性能问题，能知道怎么排查和调试。

在面试之前，我们是通过看简历来初步了解候选人的工作经验，如果发现该候选人 Hibernate 经验不足 1 年，或者最近半年没有用到，那么会着重问些技术上的细节，由此来确

认他在 ORM 方面的能力，毕竟一些技能是要靠多使用来积累的。

首先可通过如下提问来了解候选人在项目里使用 Hibernate 的基本情况。

① 在项目里，你们用的是哪个版本？对应的数据库是什么？

② 你们是用注解还是配置文件的方式编写映射文件？

③ 在项目里，你们大多是用 HQL 还是 SQL 的方式获取数据？

④ 你们项目的数据规模是多大？一张表里最多有多少数据？你们项目里通过 Hibernate 装载的数据量一般是多少？

这些问题纯粹是确认候选人在简历上描述的信息，一般只要用 Hibernate 做过项目的，都能说上来。

① 然后会进行深入地提问：在项目里你们有没有用到过一对一、一对多或多对多关联？相关的配置文件该怎么写？

如果候选人在项目里确实没怎么用过，只要说出合适的理由即可。比如有人说，他的项目数据量比较少，业务比较简单，项目经理认为没有必要用，那么我们也会认可。

② 在配置一对一、一对多或者多对多关联时、cascade 和 inverse 该怎么配？结合项目需求阐述 inverse 该配在一方还是多方？

③ 在一对多或者多对多关联的情况下，在一方这端，你们是用 set、list 还是 map 或是其他什么类型来装载多方的数据？说明选用的理由。

④ 叙述 Session 缓存里的三种对象状态？Session 的一些重要方法（比如 flush、save、persist、clear、evict 的作用）？save、persist 和 saveOrUpdate 这三个方法的不同之处？

总之要想方设法确认候选人是否掌握了让对象在三种状态之间转换的对应的 Session 里的方法。

⑤ Session 里 load 和 get 方法有什么差别？比如在一对多情况下，如果在加载一方的时候，不想加载多方，那么该怎么办？通过这个问题，来确认候选人是否具备基本的调优技能。

⑥ 在项目里，你们是否用到了 Hibernate 的拦截器或者监听器？为什么要用？在拦截器和监听器里，你们实现了什么功能？

如果候选人在项目里没用过拦截器或监听器，这个不会成为扣分项。如果用过，而且在拦截器和监听器里加入的功能确实有必要，那么这个会成为加分项。

还可能会再问些基础问题，比如在 **Hibernate** 里你们怎么实现事务？你们一般用到了哪些注解？主键生成策略是什么？

上述问题主要用来考查候选人是否达到第一个级别（能否用 **Hibernate** 来干活），如果我们要招一个初级程序员，那么达到这个标准就可以了。

我们一般会用性能调优方面的问题来区分初级程序员和高级程序员，因为在操作数据时，性能是一个不可或缺的指标项。主要包括如下问题：

① 你们项目里用到的是一级还是二级缓存？如果是用到二级缓存，那么用到的是什么组件？一般你们是把项目里的什么数据放入二级缓存？

我们会确认候选人是否把适当的数据放入二级缓存，如果候选人没有用到二级缓存，那么也不要紧，我们会通过其他问题来考查性能优化的知识点。

② 这是个开放性的问题，在 **Hibernate** 里，你们在性能优化方面，做了哪些方面的事情？或者你们在写代码的时候，如何保证 **Hibernate** 操作数据库的性能？

③ 你们在项目里，一般怎么监控 **Hibernate** 操作数据库的性能？

这个问题也没有标准答案，但一般项目都会监控数据库。

具体的措施可以是输出各 **SQL** 的运行时间；也可以监控数据库本身，比如一旦连接数过多，或者出现死锁情况，就发报警邮件；也可以监控项目内存和数据库所在服务器的内存使用情况，如果使用量过高，发报警邮件。

④ 在你们项目里，一旦出现性能问题，你们怎么排查定位？

一般是会在各方法运行前后打印时间戳和内存使用情况，出现问题后通过查看日志可以定位到究竟是哪个方法、哪个 **SQL**（**HQL**）导致的问题。

第 9 章

Spring 整合数据库层面的应用

通过 Spring 的 IoC 编程方式，让业务模块以耦合度较低的方式整合到一起，从而让项目有较好的扩展性和可维护性。通过 Hibernate 的 ORM，可以把更多地精力用在业务对象层面，减少关注数据库层面的实现。

此外，Spring 的“解耦合性”也被广泛地用来管理事务，在项目里，操作数据库和事务本身是两件不同的事情，前者和业务相关，而后者仅仅是规定了一批操作，要么全做，要么全不做。

在传统的 JDBC 等方式里，事务提交、回退等操作是紧密地和操作代码耦合在一起的，而在 Spring 的声明式事务管理方式里，在数据库相关的操作代码里我们是看不到事务痕迹的，这种解耦合大大降低了相关代码的维护代价。

9.1 Spring 与 Hibernate 的整合

总体来说，Spring 和 Hibernate 整合有两大方面的好处：第一，由 IoC 容器来管理 Hibernate 的 SessionFactory，这样能做到业务代码和数据库管理代码的低耦合组装；第二，能在 Hibernate

上使用声明式事务。

正因如此，我们在面试过程中会默认地认为候选人掌握了整合相关的基础技能，这里先来看一下如何通过 IoC 的方式来管理 SessionFactory。

9.1.1 Spring 整合 Hibernate 注解的例子

数据表如表 9.1 所示，用的是 MySQL 里的 UserInfo 表。

表 9.1 UserInfo 表结构

字段名	类型	含义
UserID int		主键，用户的 ID
UserName varchar		用户名
Pwd var	char	密码
UserType var	char	用户类型

这里的项目是 Java 类型的 HibernateSpringDemo，请大家注意下，这类项目要导入的包在该项目的 lib 目录下。

代码位置	视频位置
code/第 9 章/HibernateSpringDemo	视频/第 9 章/Spring 和 Hibernate 的整合

在整合过程中，可以把和 Hibernate 相关的配置信息写入 Spring 的配置文件里，所以这个项目里就不用再创建 hibernate.cfg.xml 文件里，不过依然要通过一个带注解的实体类来映射到 UserInfo 数据表，这个代码前面用过，所以大家看一下代码里的注释说明即可，这里不再额外讲解。

```
1 //省略必要的package和import代码
2 //用Entity和Table来表示所关联的表
3 @Entity
4 @Table(name="userinfo")
5 public class UserInfo {
6     //用ID来标识主键
7     @Id
8     @Column(name = "UserID")
9     private int userID;
10    //如下表示字段和属性之间的关联
11    @Column(name = "username")
12    private String userName;
13    @Column(name = "pwd")
14    private String pwd;
```

```

15 @Column(name = "usertype")
16     private String userType;
17 @Transient
18 private int age;
19 //省略必要的 get 和 set 方法
20 ...
21 }

```

在 Spring 的配置文件 ApplicationContext.xml 中，放入了针对 SessionFactory 的配置，代码如下。

```

1  <?xml version="1.0" encoding="gb2312"?>
2  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
3      "http://www.springframework.org/dtd/spring-beans.dtd">
4  <beans>
5  <bean id="dataSource"
6      class="org.springframework.jdbc.datasource.DriverManagerDataSource" >
7      <property name="driverClassName" value="com.mysql.jdbc.Driver"></property>
8      <property name="url"
9          value="jdbc:mysql://localhost:3306/hibernatechart"></property>
10     <property name="username" value="root"></property>
11     <property name="password" value="123456"></property>
12 </bean>

```

从第 5 行到第 10 行，定义了一个 id 是 dataSource 的 bean，其中通过诸多 property 定义了连接数据库的各种属性。

```

11 <bean id="sessionFactory"
12     class="org.springframework.orm.hibernate5.LocalSessionFactoryBean"
13     lazy-init="false">
14     <property name="dataSource" ref="dataSource" />
15     <property name="hibernateProperties">
16     <props>
17     <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
18     <prop key="hibernate.show_sql">true</prop>
19     <prop key="hibernate.hbm2ddl.auto">create</prop>
20     </props>
21     </property>
22     <property name="annotatedClasses">
23     <list>
24     <value>Model.UserInfo</value>
25     </list>
26     </property>
27 </bean>
28 </beans>

```

从第 11 行到第 25 行，定义了一个 id 为 SessionFactory 的 bean。在第 12 行中，引入了刚才定义的 dataSource 这个 bean，通过 dataSource，连接到 MySQL 数据库上。

从第 13 行到第 19 行，定义了 name 为 hibernateProperties 的 property，其中通过多个 prop 来定义 Hibernate 的各种属性，比如通过 hibernate.dialect 来定义方言，通过 hibernate.show_sql 来定义是否要在代码里输出 SQL 语句，通过 hibernate.hbm2ddl.auto 来定义程序启动时针对数据表的动作。

在第 20 行中，通过 annotatedClasses 这个属性来引入包含和数据库映射关系的 UserInfo 实体类。

在 HibernateMain 主类里，可以用 spring 的方式获得 SessionFactory 对象，从而进行数据库操作，代码如下。

```
1 //省略 package 和 import 代码
2 public class HibernateMain {
3     public static void main(String[] args) {
4         SessionFactory sessionFactory = null;
5         //这里是从 spring 的配置文件里得到 sessionFactory
6         ApplicationContext context=new
7         ClassPathXmlApplicationContext("applicationContext.xml");
8         sessionFactory = (SessionFactory)context.getBean("sessionFactory");
9         //通过 SessionFactory 创建 Session
10        Session session = sessionFactory.openSession();
11        //通过 Session 操作数据库
12        Transaction tx = session.beginTransaction();
13        UserInfo user = new UserInfo();
14        user .setUserID(3);
15        user .setUserName("Peter");
16        user .setPwd("456");
17        user .setUserType("Vip");
18        try{
19            session.save(user);
20            tx.commit();
21        } catch (Exception e) {
22            {
23                e.printStackTrace();
24            }
25            tx.rollback();
26        }
27        List<UserInfo> users = new ArrayList<UserInfo>();
28        String hqlstr = "from Model.UserInfo";
29        Query query = session.createQuery(hqlstr);
```

```

29     users = query.list();
30     System.out.println("users size is:" + users.size());
31     System.out.println("UserName" + "\t" + "UserType");
32     for(UserInfo u : users){
33         System.out.println(u.getUserName() + "\t" + u.getUserType());
34     }
35     session.close();
36     sessionFactory.close();
37 }
38 }

```

在第7行中，通过 `getBean` 方法，得到在配置文件里定义好的 `SessionFactory` 对象，通过 `SessionFactory` 创建 `Session`，并进行之后的数据库操作。

9.1.2 配置数据池来提升效率

通过 Spring 的配置文件，可以很方便地在配置 Hibernate 数据源时引入连接池，从而提升数据库访问效率。

代码位置	视频位置
code/第9章/HibernateSpringXmlDemo	视频/第9章/整合时 Hibernate 里配置连接池

下面通过修改刚才的 `HibernateSpringDemo` 项目来使用 `c3p0` 连接池。

修改点 1，由于要用到 `c3p0` 连接池，因此除了要引入刚才 `HibernateSpringDemo` 的一些 jar 包外，还需要引入 `mchange-commons-java-0.2.12.jar` 和 `c3p0-0.9.2.1.jar` 这两个包。

修改点 2，在之前的 `UserInfo.java` 里，是通过注解的方式来描述映射，而这里是用配置文件的方式。相关的文件前面我们讲解过，所以这里不再重复列出。

修改点 3，在 `ApplicationContext.xml` 里，引入连接池，代码如下。

```

1  <?xml version="1.0" encoding="gb2312"?>
2  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
3      "http://www.springframework.org/dtd/spring-beans.dtd">
4  <beans>
5  <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
6      destroy-method="close">
7  <property name="driverClass" value="com.mysql.jdbc.Driver">
8  </property>
9  <property name="jdbcUrl"
10     value="jdbc:mysql://localhost:3306/hibernatechart"></property>
11 <property name="user" value="root" />

```

```
11 <property name="password" value="123456"/>
12 <property name="minPoolSize" value="10" />
13 <property name="maxPoolSize" value="100" />
14 <property name="maxIdleTime" value="1800" />
15 <property name="acquireIncrement" value="3" />
16 <property name="maxStatements" value="1000" />
17 <property name="initialPoolSize" value="10" />
18 <property name="idleConnectionTestPeriod" value="60" />
19 <property name="acquireRetryAttempts" value="30" />
20 <property name="breakAfterAcquireFailure" value="true" />
21 <property name="testConnectionOnCheckout" value="false" />
22 </bean>
```

和之前的不同，在第 5 行到第 22 行的 `dataSource` 这个 bean 里，没有用到 MySQL 的连接驱动程序，而是用了 `c3p0` 连接池。

```
23 <bean id="sessionFactory"
    class="org.springframework.orm.hibernate5.LocalSessionFactoryBean"
    lazy-init="false">
24 <property name="dataSource" ref="dataSource" />
25 <property name="hibernateProperties">
26 <props>
27 <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
28 <prop key="hibernate.show_sql">true</prop>
29 <prop key="hibernate.hbm2ddl.auto">create</prop>
30 </props>
31 </property>
32 <property name="mappingLocations"
33     value="file:/*.hbm.xml"></property>
34 </bean>
35 </beans>
```

在第 23 行到第 34 行定义 `SessionFactory` 的 Bean 里，依然引用的是 `dataSource` 这个 Bean，不过配置 `dataSource` 时，已经用到了 `c3p0` 连接池。

由于这里用的是配置文件的方式来描述数据表和 Java Model 类的对应关系，所以在第 33 行里，需要通过 `mappingLocations` 来引入 `hbm.xml` 文件。

在 `HibernateMain` 里，使用方式没有任何改变，依然是通过如下的关键代码来创建 `SessionFactory` 和 `Session`。

```
1 ApplicationContext context=new
  ClassPathXmlApplicationContext("applicationContext.xml");
2 sessionFactory = (SessionFactory) context.getBean("sessionFactory");
3 Session session = sessionFactory.openSession();
```

刚才在配置连接池里，我们用到了一些属性，下面通过表 9.2 来说明下各常用属性值的含义。

表 9.2 c3 p0 连接池常用属性归纳表

属性名	含义
acquireRetryAttempts	定义从数据库获取新连接失败后重复尝试的次数
acquireRetryDelay	两次连接的间隔时间，单位毫秒
autoCommitOnClose	连接关闭时默认将所有未提交的操作回滚
checkoutTimeout	当连接池用完时，客户端调用 <code>getConnection()</code> 后，等待获取新连接的时间，超时后将抛出 <code>SQLException</code> 。如设为 0 则无限期待。单位毫秒
idleConnectionTestPeriod	检查所有连接池中的空闲连接的时间间隔
initialPoolSize	初始化时创建的连接数
maxIdleTime	最大空闲时间，这段时间内未使用则连接被丢弃。若为 0 则永不丢弃
maxPoolSize	连接池中保留的最大连接数
minPoolSize	最小的连接数量
maxStatements	用以控制数据源内加载的 <code>PreparedStatements</code> 数量

9.2 通过 Spring 管理事务

Hibernate 和 Spring 整合后，可以由 Spring 来管理事务。一般可以用编程式和声明式这两种方式来管理事务。相比之下，声明式这种管理方式更为常见，因为事务管理逻辑代码和实现事务的代码是分开的。

9.2.1 编程式事务管理方式

没有比较，就不会知道声明式事务的优势，所以接下来看个编程式事务的例子。

代码位置	视频位置
code/第 9 章/HibernateTransByProgram	视频/第 9 章/编程式事务的讲解

这里还是用 9.1.2 节所示的 `UserInfo` 表。在 `UserInfo.java` 里，用注解的方式，描述和数据库的关联，代码如下。

```

1 //省略必要的 package 和 import 代码
2 //用 Entity 和 Table 来表示所关联的表
3 @Entity
4 @Table(name="userinfo")
5 public class UserInfo {
6     //用 ID 来标识主键

```

```
7  @Id
8  @Column(name = "UserID")
9      private int userID;
10     //下面字段和属性之间的关联
11 @Column(name = "username")
12     private String userName;
13 @Column(name = "pwd")
14     private String pwd;
15 @Column(name = "usertype")
16     private String userType;
17 @Transient
18 private int age;
19 //省略必要的 get 和 set 方法
20 ...
21 }
```

在 Spring 的配置文件 `ApplicationContext.xml` 里，除了要配置 Hibernate 相关的数据库连接外，还加入了针对事务的配置。

```
1  <?xml version="1.0" encoding="gb2312"?>
2  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
3      "http://www.springframework.org/dtd/spring-beans.dtd">
4  <beans>
5  <bean id="dataSource"
6      class="org.springframework.jdbc.datasource.DriverManagerDataSource" >
7      <property name="driverClassName" value="com.mysql.jdbc.Driver"></property>
8      <property name="url"
9          value="jdbc:mysql://localhost:3306/hibernatechart"></property>
10     <property name="username" value="root"></property>
11     <property name="password" value="123456"></property>
12 </bean>
```

从第 5 行到第 10 行，定义了 id 为 `dataSource` 的 Bean 用来描述和数据库的连接。

```
11 <bean id="sessionFactory"
12     class="org.springframework.orm.hibernate4.LocalSessionFactoryBean"
13     lazy-init="false">
14     <property name="dataSource" ref="dataSource" />
15     <property name="hibernateProperties">
16     <props>
17     <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
18     <prop key="hibernate.show_sql">true</prop>
19     <prop key="hibernate.hbm2ddl.auto">create</prop>
20     </props>
21 </property>
```

```

20 <property name="annotatedClasses">
21 <list>
22 <value>Model.UserInfo</value>
23 </list>
24 </property>
25 </bean>

```

第11行到第25行，同前文一样定义了 SessionFactory 这个 bean。第12行，在 SessionFactory 里引入了 dataSource，由此可以成功地连接到数据库。第13行到第19行，配置了 Hibernate 的诸多属性。第20行到第24行，指定了是用 UserInfo 这个含注解的文件和数据表关联。

```

26 <bean id="transactionManager"
    class="org.springframework.orm.hibernate4.HibernateTransactionManager">
27 <property name="sessionFactory" ref="sessionFactory"></property>
28 <property name="dataSource" ref="dataSource"></property>
29 </bean>

```

第26行到第29行，指定了 Spring 的 HibernateTransactionManager 类作为事务管理器，在第27行和第28行这个事务管理器里加入了 SessionFactory 和 dataSource 这两个属性。

一旦定义了事务管理器，那么在代码里就会有一些针对事务的操作（比如提交或回滚），以后遇到事务时，就都由这个事务管理器来执行。以前经常用 JDBC 来操作事务，这里是用 HibernateTransactionManager 来管理事务。

```

30 <bean id="transactionTemplate"
    class="org.springframework.transaction.support.TransactionTemplate">
31 <property name="transactionManager" ref="transactionManager"></property>
32 <property name="readOnly" value="false"></property>
33 <property name="isolationLevelName" value="ISOLATION_DEFAULT"></property>
34 </bean>
35 </beans>

```

第30行到第34行，定义了一个事务模板 TransactionTemplate。在第31行，引入了刚才定义的事务管理器。在第32行，定义了这个事务不是只读的。在第33行，定义了这个事务的事务隔离级别。在前面数据库部分章节曾讲过这部分的知识，这里提一下，一般是有5个常量来描述事务隔离级别，级别从低到高分别如下：

- ① 读取未提交：TRANSACTION_READ_UNCOMMITTED
- ② 读取提交：TRANSACTION_READ_COMMITTED
- ③ 可重读：TRANSACTION_REPEATABLE_READ
- ④ 可串行化：TRANSACTION_SERIALIZABLE

另外还有一个，是不支持事务：TRANSACTION_NONE JDBC

把事务管理器比作饭店里的厨师，是由它具体地执行事务，而事务模板就好比是菜单。可以在菜单里加入我们需要的配置，随后提交给事务管理器来执行。

下面通过 HibernateMain 这个类来看一下事务的调用过程。

```
1 //省略必要的 package 和 import 方法
2 public class HibernateMain {
3     private TransactionTemplate transactionTemplate;
4     SessionFactory sessionFactory = null;
5     Session session = null;
6     //在这个方法里，我们通过事务来插入两个 User
7     private void updateUsers()
8     {
9         ApplicationContext context=new
10         ClassPathXmlApplicationContext("applicationContext.xml");
11         sessionFactory = (SessionFactory)context.getBean("sessionFactory");
12         session = sessionFactory.openSession();
13         transactionTemplate=(TransactionTemplate) context.getBean("transactionTemplate");
14         transactionTemplate.execute(new TransactionCallback<Boolean>() {
15             @Override
16             public Boolean doInTransaction(TransactionStatus ts) {
17                 try {
18                     UserInfo user = new UserInfo();
19                     user .setUserID(1);
20                     user .setUserName("Mike");
21                     user .setPwd("123");
22                     user .setUserType("VVip");
23                     session.save(user);
24                     UserInfo anotherUser = new UserInfo();
25                     anotherUser.setUserID(2);
26                     anotherUser.setUserName("Peter");
27                     anotherUser.setPwd("456");
28                     anotherUser.setUserType("Vip");
29                     session.save(anotherUser);
30                     session.flush();
31                 }
32                 catch (Exception e) {
33                     System.out.println("Roll Back");
34                     ts.setRollbackOnly();
35                     e.printStackTrace();
36                     return false;
37                 }
38             }
39         });
40     }
41 }
```

```

37 System.out.println("Correct");
38         return true;
39     }
40 });
41 session.close();
42 sessionFactory.close();
43 }

```

第13行到第40行，在 `transactionTemplate.execute` 里放入了和事务有关的代码。具体而言，在第15行的 `doInTransaction` 里，向数据表里存放了两个 `UserInfo` 的信息。

如果没有发生异常，那么代码能正确地执行到第40行的位置，在正确退出方法时，会提交事务（把连个 `UserInfo` 对象一起插入数据表里）。一旦出现异常，那么会在第31行的 `catch` 从句里，通过第33行的代码执行回滚操作，撤销事务。

```

44 public static void main(String[] args) {
45     try{
46         HibernateMain main = new HibernateMain();
47         main.updateUsers();
48     }
49     catch(Exception e)
50     {
51         e.printStackTrace();
52     }
53 }
54 }

```

第47行 `main` 函数里，调用了 `updateUsers` 这个方法，通过这个方法，以事务的方式插入了两个 `UserInfo` 对象。

虽然已经在 `Spring` 的配置文件里加入了针对事务的配置，但由于在代码里，显式地把事务操作的代码放入了 `transactionTemplate.execute` 这个方法里，所以是编程式事务，具体而言，还是在 `Java` 代码里（而不是配置文件里）通过编程的方式使用事务。

在编程式事务里，数据库操作的逻辑（比如这里是插入两个 `UserInfo`）和事务代码是紧密地耦合在一起的，一旦要修改事务部分的代码（比如以后换事务模板了），那么包含数据库操作的代码（这里是 `updateUsers` 方法）就不得不起跟着修改。

正是因为有这类问题，所以在当前的项目里，编程式事务用得比较少，大多还是用下面提到的声明式事务。

9.2.2 声明式事务管理方式

针对特定的项目，可以在 Spring 的配置文件里制定一个规则，以此可以指定针对特定类（一般是数据库操作的相关类）的特定方法（一般是涉及事务操作的方法）添加事务控制，并设置好事务的相关属性。

这样一来，如果单观察类和方法本身，是看不到任何事务的痕迹（因为耦合度很低），而一旦执行到具体的方法，事务就会起作用。

代码位置	视频位置
code/第 9 章/HibernateTransByConf	视频/第 9 章/声明式事务的讲解

这里还是用刚才的 UserInfo 数据表，和 9.2.1 节中一样，在 UserInfo.java 里，用注解的方式，描述和数据库的关联，代码与前面相同，这里不再给出。

在 ApplicationContext.xml 这个 Spring 的配置文件里，不仅配置了连接 MySQL 数据库的方式，更配置了针对事务的描述，代码如下。

```

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:p="http://www.springframework.org/schema/p"
5  xmlns:aop="http://www.springframework.org/schema/aop"
6  xmlns:context="http://www.springframework.org/schema/context"
7  xmlns:jee="http://www.springframework.org/schema/jee"
8  xmlns:tx="http://www.springframework.org/schema/tx"
9  xsi:schemaLocation="http://www.springframework.org/schema/aop
10 http://www.springframework.org/schema/aop/spring-aop-4.2.xsd
11 http://www.springframework.org/schema/beans
12 http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
13 http://www.springframework.org/schema/context
14 http://www.springframework.org/schema/context/spring-context-4.2.xsd
15 http://www.springframework.org/schema/jee
16 http://www.springframework.org/schema/jee/spring-jee-4.2.xsd
17 http://www.springframework.org/schema/tx
18 http://www.springframework.org/schema/tx/spring-tx-4.2.xsd">

```

至此配置了一些本配置文件里会用到的命名空间。

```

10 <bean id="dataSource"
11 class="org.springframework.jdbc.datasource.DriverManagerDataSource" >
12 <property name="driverClassName" value="com.mysql.jdbc.Driver"></property>
13 <property name="url"
14 value="jdbc:mysql://localhost:3306/hibernatechart"></property>

```

```

13 <property name="username" value="root"></property>
14 <property name="password" value="123456"></property>
15 </bean>

```

第10行到第15行，在id为dataSource的bean里，配置了连接MySQL数据库的信息。

```

16 <bean id="sessionFactory"
    class="org.springframework.orm.hibernate4.LocalSessionFactoryBean"
    lazy-init="false">
17 <property name="dataSource" ref="dataSource" />
18 <property name="hibernateProperties">
19 <props>
20 <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
21 <prop key="hibernate.show_sql">true</prop>
22 <prop key="hibernate.hbm2ddl.auto">create</prop>
23 </props>
24 </property>
25 <property name="annotatedClasses">
26 <list>
27 <value>Model.UserInfo</value>
28 </list>
29 </property>
30 </bean>

```

第16行到第30行，在id为sessionFactory的bean里，配置了Hibernate的信息。具体而言，在第17行，指定了该Hibernate需要用到dataSource的配置连接数据库。在第20行到第23行，配置了诸如“是否显示SQL语句”等Hibernate属性。在第27行，指定了是通过UserInfo这个带注解的文件来实现数据表到本地Model对象的映射。

```

31 <bean id="transactionManager"
    class="org.springframework.orm.hibernate4.HibernateTransactionManager">
32 <property name="dataSource" ref="dataSource"></property>
33 <property name="sessionFactory" ref="sessionFactory"></property>
34 </bean>
35 <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
36 <property name="dataSource" ref="dataSource" />
37 </bean>

```

第31行到第34行，在id为transactionManager的bean里，配置了事务管理器信息，由此来实现提交和回滚等操作。第35行到第37行，配置了一个封装数据库操作的jdbcTemplate对象。

```

38 <tx:annotation-driven transaction-manager="transactionManager"/>
39 <tx:advice id="txAdvice" transaction-manager="transactionManager">
40 <tx:attributes>

```

```
41 <tx:method name="update*" propagation="REQUIRED" />
42 <tx:method name="delete*" propagation="REQUIRED" />
43 <tx:method name="add*" propagation="REQUIRED" />
44 <tx:method name="*" propagation="REQUIRED" read-only="false" />
45 </tx:attributes>
46 </tx:advice>
```

第 38 行到第 46 行，配置了针对事务的一些规则，这也是声明式事务的关键代码。

具体而言，在第 39 行，说明了通过 `transactionManager` 来实现针对事务的操作。第 40 行到第 45 行，说明了事务的应用范围。比如第 41 行，说明了名字为 `update...` 的方法将采用 `REQUIRED` 这类管理方式（这个是事务传播方式，页面会详细讲解）。

```
47 <aop:config>
48 <aop:pointcut id="interceptorPointCuts" expression="execution(* com.*.*(..))" />
49 <aop:advisor advice-ref="txAdvice" pointcut-ref="interceptorPointCuts" />
50 </aop:config>
51 </beans>
```

第 47 行到第 50 行，通过 AOP 切面的方式，说明了这个事务将作用在 `com` 这个 package 下的所有文件里。

结合第 38 行到第 46 行，通过配置文件指定了在哪些文件的哪些方法里，将用某某的方式来管理事务，随后来看下在 `HibernateMain` 类里使用事务的方式。

```
1 //省略必要的 package 和 import 方法
2 public class HibernateMain {
3     private JdbcTemplate jdbcTemplate;
4     public JdbcTemplate getJdbcTemplate() {
5         return jdbcTemplate;
6     }
7     public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
8         this.jdbcTemplate = jdbcTemplate;
9     }
10    //在这个方法里，我们插入了两个 UserInfo 的对象
11    private void updateUsers()
12    {
13        ApplicationContext context=new
        ClassPathXmlApplicationContext("applicationContext.xml");
14        jdbcTemplate = (JdbcTemplate)context.getBean("jdbcTemplate");
15        String sql="INSERT INTO userInfo VALUES(?,?,?,?)";
16        jdbcTemplate.update(sql, "1","Mike","123","VIP");
17        jdbcTemplate.update(sql, "2","Peter","456","VIP");
18    }
19    public static void main(String[] args) {
```

```

20     try{
21         HibernateMain main = new HibernateMain();
22         //这里是调用
23         main.updateUsers();
24     }
25     catch(Exception e)
26     {
27         e.printStackTrace();
28     }
29 }
30 }

```

第 23 行，调用了 `updateUsers` 这个方法。在第 11 行的 `updateUsers` 方法里，仅仅是通过 `jdbcTemplate` 对象向 `UserInfo` 表里插入了两条数据，一点也看不出事务的痕迹。

但是在 `ApplicationContext.xml` 这个配置文件里，已经定义了 `com` 这个包(`HibernateMain` 在这个包里)的所有类里的以 `update` 开头的方法都将以 `propagation="REQUIRED"` 的方式执行事务，所以事实上在 `updateUsers` 方法里，已经用到了事务。

9.2.3 事务传播机制要解决的问题（适用范围）

事务传播机制，也叫事务传播行为，它是个独立的概念，和 Spring 以及 Hibernate 等语言无关。通过事务传播机制，可以定义事务之间的交互行为。

Spring 容器定义了 7 种类型的事务传播行为，由此规定了在嵌套情况下事务方法之间相互调用时的协调方式。

接下来看个案例。在如下的 `calBenefit` 方法里，为每个账户添加收益，在第 5 行里，定义了添加外汇收益的方法。

```

1 //为每个账户添加收益，参数是 1000 个账户
2 void calBenefit(List accounts)
3 {
4     ...
5     //为每个账户添加外汇收益
6     addRatsBenefit(accounts)
7     ...
8 }

```

从代码结构上来看，第 6 行的添加外汇收益这个方法嵌套地被第 2 行的 `calBenefit` 方法调用，事务传播机制正是用来协调被嵌套方法（比如添加外汇收益的方法）和调用端（比如外层的第 2 行的方法）之间的关系。

表 9.3 列出了事务传播机制各取值的含义。

表 9.3 事务传播机制各取值的含义

传播行为	在 XML 文件里的取值	含义
PROPAGATION_REQUIRED REQUIRED	ED	表示当前方法（比如计算外汇收益的方法）必须在一个具有事务的方法里运行，如果调用端（比如最外层的计算收益方法）已经被用事务的方式管理，那么被调用端（计算外汇收益的方法）可以在这个事务（比如最外层的计算收益方法所在的事务）中运行，否则就要重新开启一个事务（来运行计算外汇收益的方法）。如果被调用端（计算外汇收益的方法）发生异常，那么调用端和被调用端事务都将回滚
PROPAGATION_SUPPORTS SUPPORTS	TS	表示当前方法（比如计算外汇收益的方法）不必需要具有一个事务上下文，但是如果有一个事务的话（假设所处的最外层的计算收益方法运行在一个事务里），那么它也可以在这个事务中运行
PROPAGATION_MANDATORY MANDATORY	ATORY	表示当前方法（比如计算外汇收益的方法）必须在一个事务中运行，如果最外层的计算收益的方法没有事务在事务环境下运行，那么将抛出异常
PROPAGATION_NESTED NESTED	ED	表示如果当前方法（比如计算外汇收益的方法）刚好有一个事务（最外层的方法）在运行中，则该方法应该运行在一个嵌套事务中，被嵌套的事务可以独立于被封装的事务中进行提交或者回滚。 如果外层事务存在（比如最外层的计算收益方法处在一个事务管理下），并且外层事务抛出异常回滚，那么内层事务（计算外汇收益的方法）必须回滚。反之，内层事务并不影响外层事务，内层事务哪怕回滚了，外层事务也不会回滚。 如果封装事务不存在，则与 PROPAGATION_REQUIRED 相同
PROPAGATION_NEVER NEVER		表示当方法（比如计算外汇收益的方法）不应该在一个事务中运行时，如果存在一个事务（假设最外层的计算收益的方法处在一个事务的管理下），则抛出异常
PROPAGATION_REQUIRES_NEW REQUIRES_NEW	ES_NEW	表示当前方法（比如计算外汇的方法）必须运行在它自己独立的事务中，系统将为这个方法创建一个新的事务。如果有一个现有的事务（最外层的计算收益的方法）正在运行，则这个方法将在运行期被挂起，直到新的事务提交或者回滚才恢复执行

续表

传播行为	在 XML 文件里的取值	含义
PROPAGATION_NOT_SUPPORTED	NOT_SUPPORTED	表示该方法（比如计算外汇的方法）不应该在一个事务中运行。如果有一个事务（最外层的计算收益的方法）正在运行，那么这个方法将在运行期被挂起，直到这个事务提交或者回滚才恢复执行

9.3 针对 Spring 整合数据库的总结

在项目里，一般不会直接使用 Hibernate，而是通过 Spring 注入 Hibernate 里的 SessionFactory。所以大家面试时，应该让面试官清晰地知道你了解如下知识点。

第一，在简历中，明确地写明项目里是用 Spring 整合 Hibernate，当然，项目里也可能是用其他的 ORM 语言，比如 MyBatis 甚至是 JPA，但一般也会由 Spring 用自由组装的方式来管理访问数据库的对象（比如 Hibernate 里是 DataSource 和 SessionFactory）。

同时，要理解用 Spring 整合 Hibernate 的优势，具体是可以让业务逻辑和数据库操作逻辑以比较低耦合的方式组装到一起。

第二，一个商业项目里一定会包含事务，你需要掌握至少一种管理事务的方式，比如在本章里提到的用声明式事务。

如果你确实在项目里用到了编程式事务，那么也不要紧，不过你最好说明下理由。比如项目里事务部分的代码比较少，用编程式事务写起来比较直观，而且在开发过程中需求不大变更。第三，理解事务相关的知识点，比如前文里提到的事务隔离级别和本章提到的事务传播机制。针对事务传播机制，需要结合项目的实际需求说明，比如你项目的业务是什么，所以使用 NESTED 这种方式，而且请特别地提到你用到的方式里，如果遇到异常会怎么办？（外层和内层究竟是哪个会回滚？）

第 10 章

Web 框架案例分析

本章将介绍 Struts+Spring+Hibernate (SSH) 框架的整合方式。由于现在 Struts 用得并不多，都是些历史项目还在用 Struts，所以大家对此了解一下即可。案例中，将应用 Spring MVC+Hibernate 的开发框架。

在实际项目中，不论业务多复杂，都可以归纳成“增删改查”四个方面的应用，所以通过这四个方面的应用来走通从前端到后端的所有流程。

10.1 Struts、Spring 与 Hibernate 的整合

Struts、Spring 与 Hibernate 三者的整合简称为 SSH 整合。在这个框架里，是用到 Struts 的 MVC，通过 Spring 的 IoC（依赖注入）来降低模块之间的耦合，通过 Hibernate 来和数据库关联。

目前 Struts MVC 不如 Spring MVC 流行，所以这套框架用得也不多，大多是在历史项目里，不过这毕竟也是一套 Web 框架，之前曾流行过，从中大家也可以了解到基于框架的开发方式。

10.1.1 SSH 整合案例的说明

比如，我们要开一个便利店，一种方式是，从设计装修到联系货源再到进货等工作全都自己干；另外一种方式是加盟（比如加盟全家等超市），使用商家的铺货方式、进货方式等一整套的方案。

基于 Web 的框架和这相似，一般的公司更关注网站带来的附加价值，比如通过网站买卖东西，提供服务，所以一般没必要太多地关注网站细节，（比如前端怎么向后端传输数据，后端又如何把计算结果返回给前端），一般会采用现成的框架技术，如 SSH 框架。

这里用到的数据如表 10.1 所示，用的是 MySQL 里的 UserInfo 表，这个表是放在 projectchart 这个数据库里的。

表 10.1 UserInfo 表结构

字段名	类型	含义
Id int		主键，用户的 ID
username var	char	用户名
password var	char	密码

在这个案例中，要做一件非常简单的事情：通过登录界面登录，判断验证通过后，是否跳转到一个新页面，在其中显示登录名。

本项目的目录结构如表 10.2 所示。

表 10.2 本项目的目录结构

目录名/package 名	功能
Src/com.spring	存放 Struts Action 类
Src/com.model	存放 model 类和 hbm.xml 文件
Src/com.service	存放 Service 类
Src/com.dao	存放 dao 类
Src/conf	存放 Spring 的配置文件
WebRoot	存放 JSP 文件
WebRoot/Web-INF	存放 web.xml

其中的项目架构如图 10.1 所示。

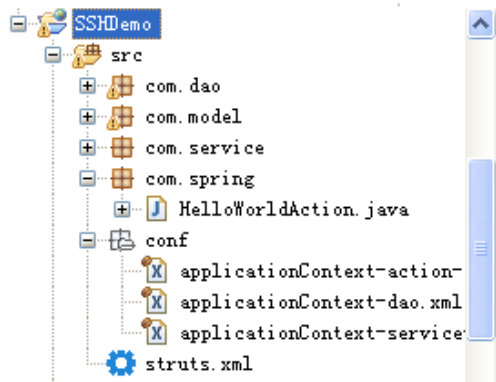


图 10.1 本项目的目录结构示意图

这里采用了对传统的项目层级结构，把业务方法存放到 Service 类里，而在 dao 类里存放数据库相关的方法(这个项目里用到了 Hibernate)，这样就可以把修改限制在比较小的范围里，比如出现数据库变更的情况，只需修改 Dao 层，而可以不用修改 Service 以及调用 Service 的方法。

代码位置	视频位置
code/第 10 章/SSHDemo	视频/第 10 章/SSH 整合案例的讲解

10.1.2 编写登录页面和 Web.xml 配置文件

前端登录的 login.jsp 页面相对简单，代码如下。

```

1  <%@ page language="java" contentType="text/html; charset=UTF-8" %>
2  <%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
3  <%@ taglib uri="/struts-tags" prefix="s" %>
4  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
5  <html>
6  <head>
7    <title>login</title>
8  </head>
9  <body>
10   <div>
11     <s:form method="post" action="demo/login.do" id="userInfo">
12       <s:textfield name="userName" label="User Name"/>
13       <br>
14       <s:password name="userPwd" label="Password"/>
15       <br>
16       <s:submit value="Login" />

```

```

17 </s: form>
18 </body>
19 </html>

```

在第 3 行, 引入了 uri 为 "/struts-tags", 前缀是 s 的 struts 标签, 之后从第 11 行到第 17 行, 通过这个标签定义了一个 form。

在 form 里, 第 12 行和 14 行, 定义了 User Name 和 Password 两大输入框, 第 16 行, 定义了一个 submit 类型的按钮。

由于这里侧重于讲述 SSH 流程, 所以就省略了一些诸如输入验证等次要因素。这个 form 的样式如图 10.2 所示, 一旦我们完成输入单击 Login 按钮后, 根据第 11 行的定义, 将以 post 的方式跳转到 demo/login.do 这个 Action 上。

Login

User Name:

Password:

图 10.2 登录界面的 Form 样式图

根据 web.xml 里的定义, 将使用 Struts MVC, 这部分的代码如下所示。

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
5     http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
6   <welcome-file-list>
7     <welcome-file>login.jsp</welcome-file>
8   </welcome-file-list>

```

第 6 行到第 8 行, 设置了 login.jsp 是默认页。也就是说, 如果输入 localhost:8080/SSHDemo, 而不输入具体的 JSP 文件名, 那么其效果等同于 localhost:8080/SSHDemo/login.jsp。

```

9 <filter>
10   <filter-name>struts2</filter-name>
11   <filter-class>
12     org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-
13     class>
14 </filter>
15 <filter-mapping>

```

```
14     <filter-name>struts2</filter-name>
15     <url-pattern>/*</url-pattern>
16 </filter-mapping>
```

第 14 行和第 15 行，定义了任何（/*）请求都将由 **struts2** 这个 class 来处理。

在第 9 行到第 12 行的名为 **struts2** 的 filter 元素里，第 11 行指定了由 **Struts** 的 dispatcher 来处理前端请求。由此可见，这里将使用 **Struts** 的 MVC 来处理和转发前端的请求。

```
17 <listener>
18   <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
19 </listener>
20 <context-param>
21   <param-name>contextConfigLocation</param-name>
22   <param-value>
23     classpath*:conf/applicationContext-*.xml
24   </param-value>
25 </context-param>
26 </web-app>
```

第 17 行到第 25 行，引入了 **Spring** 的 listener，第 23 行，指定了 **Spring** 配置文件的位置。根据这个配置，一旦启动 Web 服务，系统将自动装载定义在 **conf** 目录下的所有 xml 配置文件。

10.1.3 编写 Struts 的配置文件 struts.xml

在前文，用的是 **Struts** MVC，而且一旦单击登录页面的按钮，将跳转到 **demo/login.do** 这个 Action 上。也就是说，在 **struts.xml** 这个文件里，将对此配置具体的 Action，现在来看下代码。

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE struts PUBLIC
3    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
4    "http://struts.apache.org/dtds/struts-2.0.dtd">
5  <struts>
6    <constant name="struts.action.extension" value="do"/>
7    <package name="train" namespace="/demo" extends="struts-default">
8      <action name="login" class="helloworld" method="login">
9        <result name="success">/index.jsp</result>
10       <result name="failure">/login.jsp</result>
11     </action>
12   </package>
13 </struts>
```

第 6 行, 指定了 do 作为 Action 的后缀。第 7 行, 指定了 Action 的命名空间是/demo。第 8 行, 指定了名为 login 的 Action class 是 helloworld, 在这个 action 里, 将默认调用 login 方法 (而不是之前经常看到的 execute 方法)。第 9 行和第 10 行, 制定了 login 方法里不同返回字符串所对应的页面。

所以, login.jsp 里提交的 demo/login.do 将由 helloworld 这个 class 来处理。这个 helloworld 其实不是类名, 而是 bean 的 id, 而它所对应的 class 类将在 Spring 的配置文件里指定。

10.1.4 编写 Spring 的配置文件

在 applicationContext-action-api.xml 的第 7 行到第 9 行中, 定义了 struts.xml 里用到的 id 是 helloworld 的这个 Bean, 具体代码如下。

从第 8 行中能看到, 在 bean 所对应的 com.spring.HelloWorldAction 类里, 通过 Spring 的依赖注入方式引入了 name 为 hws 的 Service 类, 通过 ref, 能看到这个 Service 类的 id 是 helloworldservice。

```

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  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-2.5.xsd
9  http://www.springframework.org/schema/aop
10 http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
11 http://www.springframework.org/schema/context
12 http://www.springframework.org/schema/context/spring-context-2.5.xsd
13 http://www.springframework.org/schema/tx
14 http://www.springframework.org/schema/tx/spring-tx-2.5.xsd"
15 default-lazy-init="true">
16   <bean id="helloworld" class="com.spring.HelloWorldAction">
17     <property name="hws" ref="helloworldservice"/>
18   </bean>
19 </beans>

```

在 applicationContext-service-api.xml 这个配置文件里的第 9 行到第 11 行, 定义了 Action 里用到的 id 为 helloworldservice 的 bean。在第 10 行, 能看到这个类里将自动引入 id 为 userInfoDao 的一个 Bean。

```

1  <beans xmlns="http://www.springframework.org/schema/beans"

```

```
2  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:aop="http://www.springframework.org/schema/aop"
3  xmlns:context="http://www.springframework.org/schema/context"
   xmlns:tx="http://www.springframework.org/schema/tx"
4  xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
   http://www.springframework.org/schema/aop
   http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
   http://www.springframework.org/schema/context
   http://www.springframework.org/schema/context/spring-context-2.5.xsd
   http://www.springframework.org/schema/tx
   http://www.springframework.org/schema/tx/spring-tx-2.5.xsd"
5  default-lazy-init="true">
6  <bean id="helloworldservice" class="com.service.HelloWorldService">
7    <property name="userInfoDao" ref="userInfoDao"/>
8  </bean>
9  </beans>
```

而在 applicationContext-dao.xml 里，一方面定义了在服务里用到的 userInfoDao 这个 Bean，另一方面还定义了用来创建数据库连接的 sessionFactory 对象。

```
1  <?xml version="1.0" encoding="gb2312"?>
2  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
3    "http://www.springframework.org/dtd/spring-beans.dtd">
4  <beans>
5    <bean id="dataSource"
6      class="org.springframework.jdbc.datasource.DriverManagerDataSource" >
7      <property name="driverClassName" value="com.mysql.jdbc.Driver"></property>
8      <property name="url"
9        value="jdbc:mysql://localhost:3306/projectchart"></property>
10     <property name="username" value="root"></property>
11     <property name="password" value="123456"></property>
12  </bean>
```

第 5 行到第 10 行，在 id 为 dataSource 的这个 bean 里定义了连接到 MySQL 的各种配置，包括驱动、URL、用户名和密码。

```
11 <bean id="sessionFactory"
12   class="org.springframework.orm.hibernate5.LocalSessionFactoryBean"
13   lazy-init="false">
14   <property name="dataSource" ref="dataSource" />
15   <property name="hibernateProperties">
16     <props>
17       <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
18       <prop key="hibernate.show_sql">true</prop>
```

```

17 <prop key="hibernate.hbm2ddl.auto">Validate</prop>
18 </props>
19 </property>
20 <property name="mappingResources">
21     <list>
22         <value>com/model/UserInfo.hbm.xml
23     </value>
24     </list>
25 </property>
26 </bean>

```

第 11 行到第 26 行，定义了 SessionFactory 这个 Bean，其中在第 12 行，指定了这个 SessionFactory 用到的 dataSource。也就是说，这里的 sessionFactory 是根据 dataSource 里的定义将连接到 MySQL 上。

第 14 行到第 18 行，配置了各种 Hibernate 的诸如方言等属性，请大家重点注意，第 17 行，指定了 hibernate.hbm2ddl.auto 值是 Validate，而不是会导致全表数据丢失的 Create。

在第 20 行到第 25 行中，指定了 UserInfo.hbm.xml 文件的路径。

```

27 <bean id="userInfoDao" class="com.dao.UserInfoDao" >
28 <property name="sessionFactory" ref="sessionFactory"/>
29 </bean>
30 </beans>

```

第 27 行到第 29 行，定义了在服务里会用到的 userInfoDao 这个 Bean，同时第 28 行指定了这个 Bean 里将自动装载 sessionFactory。

10.1.5 编写 Struts 的 Action 类

通过配置文件，可知 login.jsp 里发出的/demo/login.do 请求将由 com.spring 这个 package 里的 HelloWorldAction 类的 login 方法来处理，现在来看下这个类的代码。

```

1 package com.spring;
2 import com.model.UserInfo;
3 import com.service.HelloWorldService;
4 public class HelloWorldAction {
5     private String message;
6     private HelloWorldService hws;
7     public HelloWorldService getHws() {
8         return hws;
9     }
10    public void setHws(HelloWorldService hws) {
11        this.hws = hws;

```



```
12 }
13 //这两个属性需要和 login.jsp 里 form 的 text 相对应
14 private String userName;
15 private String userPwd;
16 public String login() {
17     UserInfo user = hws.getUserInfoByLoginName(userName,userPwd);
18     if(user != null)
19     {
20         message = user.getUsername();
21         return "success";
22     }
23     else
24         return "failure";
25 }
26 省略针对 userName、userPwd 和 message 三个属性的 get 和 set 方法
27 }
```

第 16 行到第 25 行，定义了该 Action 里的主要方法 login 方法，其中第 17 行，通过 HelloWorldService 类型的 hws 对象，调用定义在 Service 层的 getUserInfoByLoginName 方法。如果根据输入的用户名和密码能从数据表里找到对应的记录，则会进入第 18 行的流程，将登录名存入 message 对象后随后返回 success，否则将走第 24 行的流程，返回 failure。

由于在相关配置文件里，已经通过如下代码在 Action 类里注入了 hws 对象，所以在这里无须通过 new 等方式初始化该对象，而是可以直接使用，这样可以降低 Action 和 Service 层的耦合关系。

```
1 <bean id="helloworld" class="com.spring.HelloWorldAction">
2 <property name="hws" ref="helloworldservice"/>
3 </bean>
```

同样，根据 struts.xml 里的定义，针对不同 login 方法的返回值，将自动跳转到 index.jsp（成功登录）或 login.jsp（验证没成功）这两个不同的页面上。

10.1.6 编写 Service 和 DAO 类

项目里通常采用在 Service 类里定义具体的业务方法，而这些业务方法一般是调用 DAO 层的方法实现针对数据库的操作。这里的 Service 是 HelloWorldService.java，代码如下。

```
1 package com.service;
2 import com.dao.UserInfoDao;
3 import com.model.UserInfo;
4 public class HelloWorldService {
5     //定义了 userInfoDao 对象，它是通过配置文件注入的
```

```

6  private UserInfoDao userInfoDao;
7  public UserInfoDao getUserInfoDao() {
8      return userInfoDao;
9  }
10 public void setUserInfoDao(UserInfoDao userInfoDao) {
11     this.userInfoDao = userInfoDao;
12 }
13 public UserInfo getUserInfoByLoginName(String loginName,String pwd) {
14     return userInfoDao.getUserInfoByLoginName(loginName,pwd);
15 }
16 }

```

第 13 行到第 15 行,定义了了在 Struts Action 里用到的 `userInfoDao.getUserInfoByLoginName` 方法。同样,由于在配置文件里,可通过如下代码在 Service 类里引入了 `userInfoDao`,所以这里也不需要初始化该 Dao 对象。

```

1  <bean id="helloworld" class="com.spring.HelloWorldAction">
2      <property name="hws" ref="helloworldservice"/>
3  </bean>

```

一般在 Dao 里直接和数据库打交道,下面来看下 `UserInfoDao` 的代码。

```

1  package com.dao;
2  import java.util.List;
3  import javax.annotation.Resource;
4  import org.hibernate.Criteria;
5  import org.hibernate.Query;
6  import org.hibernate.SQLQuery;
7  import org.hibernate.Session;
8  import org.hibernate.SessionFactory;
9  import org.hibernate.criterion.DetachedCriteria;
10 import org.hibernate.criterion.Restrictions;
11 import com.model.UserInfo;
12 public class UserInfoDao {
13     //这里的 sessionFactory 是注入的,所以无须初始化
14     private SessionFactory sessionFactory = null;
15     public SessionFactory getSessionFactory() {
16         return sessionFactory;
17     }
18     public void setSessionFactory(SessionFactory sessionFactory) {
19         this.sessionFactory = sessionFactory;
20     }
21     public Session getSession() {
22         return sessionFactory.openSession();
23     }

```

在第 21 行到第 23 行的 `getSession` 方法里，通过 `sessionFactory.openSession`，返回了一个 `Session` 对象。

```
24 public UserInfo getUserInfoByLoginName(String loginName,String pwd) {  
25     Criteria crit = getSession().createCriteria(UserInfo.class);  
26     crit.add(Restrictions.eq("username", loginName));  
27     crit.add(Restrictions.and( Restrictions.eq("password", pwd) ));  
28     List<UserInfo> list = crit.list();  
29     if (null == list || list.size() == 0)  
30         return null;  
31     else  
32         return list.get(0);  
33 }  
34 }
```

第 24 行到第 34 行，定义了 `Service` 层里用到的 `getUserInfoByLoginName` 方法。其中，在第 25 行里定义了一个 `Criteria` 对象，通过 `createCriteria` 方法指定查询结果的类型为 `UserInfo`，并通过第 26 行和第 27 行的两个 `add` 方法，设置了两个条件。

也就是说，根据从 `login.jsp` 传递过来的用户名和密码这两个参数，到数据表里查询是否有这个用户，并通过第 28 行的 `crit.list` 方法返回查询结果。

第 29 行，如果 `list` 对象为空，那么说明输入的用户名和密码不对，将返回 `null` 对象，否则将在第 32 行里返回查询到的 `UserInfo` 对象。

根据 `Action` 类里的如下代码，如果通过验证，则会返回 `success` 并随后跳转到 `index.jsp`，否则将返回 `failure` 并跳转到 `login.jsp` 页面。

```
1  if(user !=null)  
2  {  
3      mess    age = user.getUsername();  
4      return "success";  
5  }  
6  else  
7      return "failure";
```

10.1.7 编写 Model 类和映射文件

在 `UserInfoDao` 这个类里，用到了 `Hibernate` 这个 `ORM` 技术，所以需要编写和 `userinfo` 数据表相对应的 `UserInfo` 类以及 `UserInfo.hbm.xml` 映射文件。

`UserInfo` 类的代码相对简单，从第 3 行到第 5 行定义了 3 个和数据表里相对应的属性，

并需要编写它们的 `get` 和 `set` 方法。

```

1 package com.model;
2 public class UserInfo implements java.io.Serializable {
3     private Long id;
4     private String username;
5     private String password;
6     省略针对 id、username 和 password 的 get 和 set 方法
7 }

```

而在 `UserInfo.hbm.xml` 文件里，需要定义 `UserInfo` 数据表和 `UserInfo.java` 这个 `Model` 类之间的映射关系，代码如下。

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3 "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
4 <hibernate-mapping package="com.model">
5 <class name="UserInfo" table="userinfo">
6 <id name="id" type="java.lang.Long">
7 <column name="id" />
8 <generator class="identity" />
9 </id>
10 <property name="username" type="java.lang.String">
11 <column name="username" length="20" not-null="true" unique="true" />
12 </property>
13 <property name="password" type="java.lang.String">
14 <column name="password" length="20" />
15 </property>
16 </class>
17 </hibernate-mapping>

```

其中第 4 行指定了数据表和 `Model` 类的映射关系。第 6 行到第 9 行，通过 `id` 元素定义了 `UserInfo` 表里 `id` 这个主键和 `UserInfo.java` 这个类里 `id` 的映射关系。随后在第 10 行到第 15 行中，定义了 `username` 和 `password` 这两个字段的映射关系。

10.1.8 编写显示返回结果的 `index.jsp`

如果通过验证，则会跳转到 `index.jsp` 上，这个页面也比较简单，将通过第 8 行，在页面里显示 `Welcome:xxx` 的字样，代码如下。

```

1 <%@ page language="java" import="java.util.*" pageEncoding="utf-8"%>
2 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
3 <html>
4 <head>

```

```
5 <title>Welcome</title>
6 </head>
7 <body>
8     Welcome ${message}<br>
9 </body>
10 </html>
```

其中的 message 字段，是从 HelloWorldAction 的 login 方法里传递而来的。

10.1.9 对 SSH 框架的分析

这个例子并不复杂，一般的商业项目里，如下模块会得到扩展。

① 前端方面，会更多地使用 Struts 标签、JS+DIV+CSS 或其他前端标签来完善 Web 界面，本书重点是后端而不是前端，所以界面相对简单。

② 本案例中，调用的层次是 Action 到 Service 再到 Dao，其他的商业项目里，会根据具体的业务扩展出更多的 Service 和 DAO 类，并在其中封装业务和数据库相关的方法。

③ 会扩展 Action 类以及对应的 Struts.xml 配置文件，通过不同的 Action 来处理不同前端页面的请求。

④ 可能会用到声明式事务等高级知识点。

但是由于目前 SSH 框架并不流行，我们在面试时，如果候选人（尤其是工作经验小于 3 年的初级程序员）没用过，那么我们认为他有任何的欠缺。不过如果用过这套框架，至少说明该候选人有框架经验。我们一般会通过如下考查点来确认候选人对 SSH 框架的理解。

考查点 1: Spring 和 Struts 的整合，Spring 和 Hibernate 的整合。

大家都知道，SSH 框架用的是 Struts MVC，但在 Struts.xml 里，Action 一般是 bean id，而不会像直接用 Struts MVC 时那样直接指定 Java 类名，而且，一般会通过 Spring 的配置文件，在 Action 里注入 Service，在 Service 里注入 DAO，在 DAO 里注入 SessionFactory（或其他能操作数据库）的对象。

另外，在这套 SSH 框架里，一般会在配置文件里定义 sessionFactory（或 JdbcTemplate 之类的 Hibernate 对象），同时把包含数据库连接属性的 dataSource 注入到 SessionFactory 里，同时会把 SessionFactory 注入 DAO 里。

考查点 2，结合你在项目里的一个具体业务，说明 SSH 的流程。

不少候选人可能仅仅在简历上写了他们做过 SSH，但实际上没做过，这些人遇到这个问

题就可能露馅。

考点点 3，你们项目为什么还会用到 SSH 框架。

还是考查你的 SSH 经验是否是真实的项目经验，如果你项目里真的是用，那么总有理由，我们听到的合理的理由是，这是一个处于维护阶段的历史项目，或者是客户要求用 SSH，因为要和他们其他的系统整合，我们听到的不合理的理由有，这个框架比较流行，或者项目经理选用的（目前很少会选用这个框架）。

10.2 基于 Spring MVC 的 Web 框架分析

由于 Spring MVC 比 Struts MVC 更加轻便，所以当前有不少项目都选择 Spring MVC。仔细观察项目就会发现，各种需求点都离不开“增删改查”这四个功能，下面将用目前比较流行的 Spring MVC+Hibernate 这套 Web 框架实现一个包含四个功能点的一个案例。

麻雀虽小，五脏俱全，在这个功能不算复杂的项目里，除了 Spring 整合 Hibernate 和 Spring MVC 这些基本的技术点外，还包含了拦截器、声明式事务和 Rest 风格 URL 等常用技术点。

10.2.1 Spring MVC 案例的说明

这里用到的数据如表 10.3 所示，用的是 MySQL 里的 user 表，这个表是放在 projectchart 这个数据库里的。

表 10.3 user 表结构

字段名	类型	含义
Id int		主键，用户的 ID，自增长
name var	char	姓名
age int		年龄

在这个项目里，将实现如下功能点。

功能点 1，展示所有的用户信息，如图 10.3 所示。

添加用户

名字	年龄	操作
Peter	12	更新 删除
Mike	13	更新 删除
Peter	21	更新 删除
Zhang	15	更新 删除
Fang	20	更新 删除

图 10.3 展示所有用户信息

功能点 2，单击“删除”按钮后，能删除当前用户，并刷新显示页面。

功能点 3，单击“添加用户”按钮后，能进入到如图 10.5 所示的界面，在其中输入用户名和年龄后，单击“添加用户”按钮，能往数据表里插入所添加的用户信息，并跳转到图 10.4 所示的展示所有用户信息界面。

用户名：

年 龄：

添加用户

图 10.4 添加用户的界面

功能点 4，在图 10.4 中，单击“更新”按钮后，能进入如图 10.5 所示的界面，在其中能看到用户的当前信息。

在完成修改后，单击“修改”按钮，能把修改后的信息更新到数据库里，并能跳转到如图 10.5 所示的界面，在其中展示所有的用户信息。

用户名：

年 龄：

修改

图 10.5 更改用户的界面

本项目的目录结构如表 10.4 所示。

表 10.4 S pring MVC 项目的目录结构

目录名/package 名	功能
Src/com.mvc	存放 Spring 的 Controller 类
Src/com.model	存放 Model 类
Src/com.interceptor	存放拦截器类
Src/com.service	存放 Service 类
Src/com.dao	存放 Dao 类
Src/conf	存放 Spring 的配置文件
WebRoot	存放 displayUsers.jsp,addUser.jsp 和 editUser.jsp 这三个前端文件
WebRoot/Web-INF	存放 web.xml

这里同样采用了 Service+DAO 的开发模式。在 conf 目录下，分别用两个 xml 文件来存放基于 Hibernate 的配置和基于 Spring MVC 的配置，这样做的目的是为了隔离修改。

代码位置	视频位置
code/第 10 章/SpringMVCProj	视频/第 10 章/Spring MVC+Hibernate 整理案例的讲解

10.2.2 在 web.xml 里定义使用 Spring MVC

在 web.xml 文件里，需要告诉系统使用 Spring 的 MVC，具体代码如下。

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5      http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
6  <!--系统级别指定 Spring 配置文件的位置 -->
7  <context-param>
8  <param-name>contextConfigLocation</param-name>
9  <param-value>/WEB-INF/classes/conf/spring*.xml</param-value>
10 </context-param>
11 <!-- 配置 Spring 的 listener -->
12 <listener>
13 <listener-class>org.springframework.web.context.ContextLoaderListener
14 </listener-class>
15 </listener>
16 <!-- 配置 springMVC 的 DispatcherServlet -->
17 <servlet>
18 <servlet-name>springMVC</servlet-name>
19 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-cla
20 <init-param>
```



```

21 <!--指定 Controller 级别配置文件的路径-->
22 <param-name>contextConfigLocation</param-name><param-value>classpath:conf/spr
    ing-*.xml</param-value>
23 </init-param>
24 <load-on-startup>1</load-on-startup>
25 </servlet>
26 <servlet-mapping>
27 <servlet-name>springMVC</servlet-name>
28 <url-pattern>/</url-pattern>
29 </servlet-mapping>
30 </web-app>

```

第 7 行到第 10 行，指定了系统级配置文件的路径。第 12 行到第 15 行，指定了 Spring 的 Listener，这里用到的是 ContextLoaderListener，它的作用是，启动 Web 容器时，会自动装配 ApplicationContext 的配置信息。如果不在第 7 行到第 10 行指定系统级配置文件的路径，那么 ContextLoaderListener 会自动去找/WEB-INF/applicationContext.xml 这个文件。如果没有指定配置文件的路径，而且也没有把 applicationContext.xml 放到指定的路径，那么启动 Web Server（这里是 Tomcat）时，系统会报错。

第 17 行到第 25 行，定义了名为 SpringMVC 的一个 Servlet，它的处理类是基于 Spring 的 DispatcherServlet，由此配置了整个项目将用到 Spring 的 MVC，20 行到第 23 行，通过 init-param 配置了基于 Servlet（也就是 Spring 里的 Controller）配置文件的路径，若不做这个配置，默认的文件是 WEB-INF/（Servlet 名）-servlet.xml，这里是 SpringMVC-servlet.xml。同样，如果没有准备这个文件，而且也没有通过指定 init-param 来指定 Servlet 级别的配置文件，那么启动 Web 服务器时，也会报错。

第 26 行到第 29 行，通过 servlet-mapping 来指定了 SpringMVC 这个 Servlet 的处理范围，也就是说，任何请求（/）都将用 Spring MVC 来处理。

10.2.3 编写整合 Hibernate 的 xml 文件和 Model 类

这里还是用 Spring 来整合 Hibernate，这部分的配置文件放在 conf 目录下的 spring-orm.xml 文件里，代码如下。

```

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     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

```

```

7 http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
8 http://www.springframework.org/schema/context
9 http://www.springframework.org/schema/context/spring-context-2.5.xsd
10 http://www.springframework.org/schema/aop
11 http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
12 http://www.springframework.org/schema/tx
13 http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">

```

第 2 行到第 13 行，在 Spring 的头文件里，指定了这里将要用到的命名空间和 Schema，请大家照着写，否则在后继定义时会出错。

```

14 <!-- 配置数据源 -->
15 <bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
16 <property name="driverClassName" value="com.mysql.jdbc.Driver"></property>
17 <property
    name="url" value="jdbc:mysql://localhost:3306/projectchart"></property>
18 <property name="username" value="root"></property>
19 <property name="password" value="123456"></property>
20 </bean>

```

第 15 行到 20 行，通过了 id 为 dataSource 的 Bean，定义了连接到 MySQL 数据库的属性。请注意这里是连接到 MySQL 里 projectchart 这个数据库，用户名和密码分别是 root 和 123456。

```

21 <!-- 配置 sessionFactory -->
22 <bean id="sessionFactory"
    class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
23 <property name="dataSource" ref="dataSource" />
24 <property name="hibernateProperties">
25 <props>
26 <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
27 <prop key="hibernate.hbm2ddl.auto">update</prop>
28 <prop key="hibernate.show_sql">true</prop>
29 <prop key="hibernate.format_sql">true</prop>
30 </props>
31 </property>

```

从第 22 行开始，定义了 SessionFactory，其中，第 23 行指定该 SessionFactory 用到前面定义的 dataSource，第 26 行到第 29 行，定义了针对 Hibernate 的诸多属性。

```

32 <!-- 注解扫描的包 -->
33 <property name="annotatedClasses">
34 <list>
35 <value>com.model.User</value>
36 </list>

```

```
37 </property>
38 </bean>
```

第 33 行到 37 行，定义了用于描述映射数据表的 `user` 类，而且指定了通过注解的方式来描述映射关系。

```
39 <!-- 声明式容器事务管理 ,transaction-manager 指定事务管理器为 transactionManager -->
40 <tx:advice id="txAdvice" transaction-manager="transactionManager">
41 <tx:attributes>
42 <tx:method name="add*" propagation="REQUIRED" />
43 <tx:method name="del*" propagation="REQUIRED" />
44 <tx:method name="get*" propagation="REQUIRED" />
45 <tx:method name="modify*" propagation="REQUIRED" />
46 </tx:attributes>
47 </tx:advice>
48 <aop:config>
49 <aop:pointcut id="txPointcut" expression="execution(* com.service..*(..))" />
50 <aop:advisor pointcut-ref="txPointcut" advice-ref="txAdvice"/>
51 </aop:config>
```

第 39 行到第 47 行，用声明的方式定义了本项目里的事务。第 40 行，指定了事务管理器。第 42 行到第 45 行，分别定义了以不同字符串开头方法名的事务处理方式。

第 49 行，指定了事务将作用到 `com.service` 这个包下所有的类文件。

```
52 <!-- 配置事务管理器 -->
53 <bean id="transactionManager"
    class="org.springframework.orm.hibernate4.HibernateTransactionManager">
54 <property name="sessionFactory" ref="sessionFactory" />
55 </bean>
```

第 53 行到第 55 行，定义了一个事务管理器。第 53 行，定义了它的 `class`，通过第 54 行的代码，在这个事务管理器里引入了 `sessionFactory`。

```
56 <bean id="userService" class="com.service.UserService">
57 <property name="userDao" ref="userDao"/>
58 </bean>
59 <bean id="userDao" class="com.dao.UserDao">
60 <property name="sessionFactory" ref="sessionFactory"/>
61 </bean>
62 </beans>
```

第 56 行到第 58 行，在 `Service` 层的代码里引入了 `Dao` 类。第 59 行到第 61 行，在 `Dao` 类里引入了 `SessionFactory`。

从上述配置文件里可知，需要用 `com.model.User` 这个 `Model` 类来映射 `MySQL` 里的数据

表，文件的代码如下。

```

1  //省略必要的 package 和 import 代码
2  @Entity
3  @Table(name="User")
4  public class User {
5      //通过@Id 指定主键，而且设置主键是自增的
6      @Id
7      @GeneratedValue(strategy=GenerationType.AUTO)
8      @Column(length=32)
9      private Integer id;
10     @Column(length=32)
11     private String name;
12     @Column(length=32)
13     private Integer age;
14     //省略针对 id、name 和 age 的 getter 和 setter 方法
15 }

```

在 Model 里的第 2 行和第 3 行，通过 @Entity 和 @Table 指定了 User 这个类将映射到 MySQL 里的 User 表。第 6 行到第 9 行，通过 @Id 指定了主键，第 7 行，通过 strategy 设置了主键 Id 是自增长的。

第 10 行到第 13 行，指定类文件里 name 和 age 两个属性和 MySQL 表里两个相关字段的映射关系。

10.2.4 配置 Spring MVC 的 xml 文件

在 conf 目录下的 spring-mvc.xml 文件里，设置了和 Spring MVC 相关的配置信息，代码如下。

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3  xmlns:context="http://www.springframework.org/schema/context"
4  xmlns:p="http://www.springframework.org/schema/p"
5  xmlns:mvc="http://www.springframework.org/schema/mvc"
6  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
7  xsi:schemaLocation="http://www.springframework.org/schema/beans
8  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
9  http://www.springframework.org/schema/context
10 http://www.springframework.org/schema/context/spring-context.xsd
11 http://www.springframework.org/schema/mvc
12 http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

```

从第 2 行到第 8 行，定义了 Spring 的头文件。

```
9 <context:component-scan base-package="com.mvc" />
10 <mvc:annotation-driven />
```

第 9 行，指定了需要扫描 com.mvc 这个包，因为在其中定义了 Spring 的 Controller。第 10 行，注册了两个 Spring MVC 里@Controller（控制器）分发请求所必须用到的两个 Bean，即 DefaultAnnotationHandlerMapping 和 AnnotationMethodHandlerAdapter。

```
11 <!-- 配置解析器 -->
12 <bean id="viewResolver"
13 class="org.springframework.web.servlet.view.InternalResourceViewResolver">
14 <property name="prefix" value="/" />
15 <property name="suffix" value=".jsp" />
16 </bean>
```

第 12 行到第 16 行，配置了一个视图解析器，同时第 13 行中指定了具体的实现类。在第 14 行和第 15 行中，分别设置了前缀和后缀字符，这样，一旦有请求返回到前端，会自动地给这个请求添加指定的前缀（/）和后缀（即.jsp）。

```
17 <mvc:interceptors>
18 <mvc:interceptor>
19 <mvc:mapping path="/*" />
20 <bean class="com.interceptor.SpringMVCInterceptor"></bean>
21 </mvc:interceptor>
22 </mvc:interceptors>
23 </beans>
```

第 17 行到第 22 行，配置了一个 Spring 的拦截器。从第 18 行来看，它将拦截所有的请求。第 19 行，配置了这个拦截器的具体实现类。

10.2.5 编写前端的增改查 JSP 文件

这里我们分别需要开发三个 JSP 前端文件用于实现显示、添加和更新功能，其中在显示页面里，包含了删除功能。

1. 在 displayUsers.jsp 里实现显示和删除效果

这个页面的代码如下。

```
1 <%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>
2 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
3 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

在第 2 行，引入了 JSTL 的标签库，同时指定它的前缀是 `c`。这部分属于前端的知识，属于很容易查询到的知识点，所以本书没有具体展开，如果对这方面大家想要深入了解，可以到任何搜索引擎查询“JSTL 用法”等关键字即可。

```

4  <html>
5  <head>
6  <title>用户列表</title>
7  </head>
8  <body>
9  <a href=/SpringMVCProj/addUser.jsp>添加用户</a>
10 <table border="1">
11 <tbody>
12 <tr>
13 <th>名字</th>
14 <th>年龄</th>
15 <th>操作</th>
16 </tr>
17 <c:forEach items="${users}" var="u">
18 <tr>
19 <td>${u.name }</td>
20 <td>${u.age }</td>
21 <td>
22 <a href="/SpringMVCProj/getUser/${u.id}">更新</a>
23 <a href="/SpringMVCProj/deleteUser/${u.id}">删除</a>
24 </td>
25 </tr>
26 </c:forEach>
27 </tbody>
28 </table>
29 </body>
30 </html>

```

在第 9 行，通过 `<a>` 的方式定义了一个超链接，由此可以跳转到添加用户的页面上。

第 17 行到 26 行，通过 `<c:forEach>` 的标签，实现了迭代展示用户的效果。从第 17 行得知，在后继的第 18 行到第 25 行里，将以此展示 `users` 对象里的各用户，而 `users` 这个对象将用后端（也就是 Java 端）传来。

在第 22 行和第 23 行，能看到在每条用户之后，会有“更新”和“删除”两个超链接，它们分别向 `/SpringMVCProj/getUser/` 和 `/SpringMVCProj/deleteUser` 上发请求，而这两类请求都将经过 Spring 的控制器类转发到适当的处理类上。

2. 在 addUser.jsp 里添加效果

```

1  <%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>
2  <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
4  <html>
5  <head>
6  <title>添加用户</title>
7  </head>
8  <%
9  String path = request.getContextPath();
10 String basePath =
    request.getScheme()+"://"+request.getServerName()+":"+request.getServerPort()+
    path+"/";
11 %>
12 <body>
13 <form action="<%=basePath%>/addUser" method="post">
14 用户名:<input type="text" name="name"><br/>
15 年&nbsp;龄:<input type="text" name="age"><br/>
16 <input type="submit" value="添加用户">
17 </form>
18 </body>
19 </html>

```

从第 13 行到第 17 行是一个 form，在其中可以供用户输入用户名和年龄。当完成输入，单击第 16 行的“添加用户”按钮后，能根据第 17 行的定义，以 post 的方式把整个表单（form）提交到<%=basePath%>/addUser 这个处理请求上。

从第 8 行到第 11 行，得到了当前请求所在的相对路径和绝对路径，所以这里的提交路径是<%=basePath%>/addUser，和前文里直接输入了项目名的方式（/SpringMVCProj/getUser）相比，这种方式更加灵活。

3. 在 editUser.jsp 里修改效果

```

1  <%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>
2  <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
4  <html>
5  <head>
6  <title>修改用户</title>
7  </head>
8  <%
9  String path = request.getContextPath();
10 String basePath =
    request.getScheme()+"://"+request.getServerName()+":"+request.getServerPort()+

```

```

    +path+"/";
11  %>
12  <body>
13  <form action="<%=basePath%>/modifyUser" method="post">
14  <input type="hidden" name="id" value="{user.id }">
15  用户名:<input type="text" name="name" value="{user.name }"><br/>
16  年&nbsp;年龄:<input type="text" name="age" value="{user.age }"><br/>
17  <input type="submit" value="修改">
18  </form>
19  </body>
20  </html>

```

这个页面和前文的添加用户页面很相似，区别点有三个。

区别点 1，在第 13 行，请求跳转的路径修改为<%=basePath%>/modifyUser。

区别点 2，在第 14 行，以 hidden 的方式定义了 id，虽然这个 id 不显示在页面上，但会提交到后台，这样就能以此 id 知道应该修改哪个 User 对象。

区别点 3，在用户名和年龄的文本框里，通过类似于 value="{user.name }" 的方式，事先放置了修改前的值。

10.2.6 编写拦截器类和控制器类

前端的请求首先将经过拦截器，在没有被拦截的前提下（本案例拦截器的作用只是打印请求，没有拦截），这个请求将交由控制器转发，下面来看下这两个关键性部件的代码。

1. 编写拦截器类

从配置文件里得知拦截器的处理类是 com.interceptor.SpringMVCInterceptor，这部分的代码如下。

```

1  //省略必要的 package 和 import 代码
2  public class SpringMVCInterceptor implements HandlerInterceptor {
3      @Override
4      public boolean preHandle(HttpServletRequest request,
5          HttpServletResponse response, Object handler) throws Exception {
6          String requestUri = request.getRequestURI();
7          String contextPath = request.getContextPath();
8          String url = requestUri.substring(contextPath.length());
9          System.out.println("url is: " + url);
10         return true;
11     }
12     @Override

```



```

13 public void postHandle(HttpServletRequest request,
14     HttpServletResponse response, Object handler,
15     ModelAndView modelAndView) throws Exception {    }
16 @Override
17 public void afterCompletion(HttpServletRequest request,
18     HttpServletResponse response, Object handler, Exception ex)
19     throws Exception {    }

```

这里通过实现（implements）HandlerInterceptor 类来实现拦截器，其中需要重写三个方法。

- preHandle 方法是在请求被处理前调用的，这里是打印出请求的 URL，该方法的返回值是 true，表示继续执行。如果返回 false，则该请求就会被拦截掉，不会往下走了。在实际项目中，还可以在这个方法里实现权限监测，比如发现权限不对时就拦截下该请求。
- postHandle 方法是在处理请求后，渲染视图前被调用，这里没有写任何动作。
- afterCompletion 方法是在视图渲染完毕后被调用，这里也没放任何代码。在实际项目中，可以通过记录结束时间来监控性能，还可以进行资源释放的动作。

2. 编写控制器类

根据配置文件里的设置，在 com.mvc 包的 UserController 类里实现了 Spring 的控制器部分的代码。

根据之前我们对控制器类的了解，从前端发来的各种请求，会根据@RequestMapping 的设置，提交给适当的方法来处理。

```

1 //省略必要的 package 和 import 方法
2 //通过@Controller 注解来说明该方法是控制器类
3 @Controller
4 public class UserController {
5     //通过注入的方式引入 userService 类
6     @Resource(name="userService")
7     private UserService userService;
8     @RequestMapping("/addUser")
9     public String addUser(User user){
10         userService.addUser(user);
11         return "redirect:/displayUsers";
12     }

```

根据@RequestMapping 的配置，如果请求的样式是/addUser，就会被第 9 行的 addUser 方法处理，即通过第 10 行调用 userService 的 addUser 方法来实现添加用户的功能。

根据第 11 行该方法的返回值，它将通过 `redirect` 的方式发出 `/displayUsers` 请求，从这个控制器类的后继代码来看，这个 `/displayUsers` 请求将由 `getAll` 方法来处理。

```
13     @RequestMapping("/displayUsers")
14     publicString getAll(HttpServletRequest request){
15         List<User> users = userService.getAll();
16         request.setAttribute("users", users);
17         return"/displayUsers";
18     }
```

从第 13 行的 `@RequestMapping` 的设置来看，一旦有 `/displayUsers` 这样的请求，就将交给第 14 行的 `getAll` 方法来处理。在第 15 行，将通过 `userService` 的 `getAll` 方法拿到所有用户的信息，并通过第 17 行的 `return` 代码，直接跳转到 `/displayUsers` 这个请求上。

根据 `spring-mvc.xml` 里解析处理器的配置，将给这个请求加上 `.jsp` 后缀，也就是说，这个 `getAll` 方法会在第 16 行把所有用户的信息（也就是 `users` 对象）通过 `request.setAttribute` 方法放入 `users` 这个属性，随后会跳转到 `/displayUsers.jsp` 这个页面。

```
19     @RequestMapping("/deleteUser/{id}")
20     publicString deleteUser(@PathVariable String id, HttpServletResponse
    response){
21         userService.deleteUser(id);
22         return"/redirect:/displayUsers";
23     }
```

从第 19 行的 `@RequestMapping` 的设置来看，一旦有 `/deleteUser` 这样的请求，就将交给第 20 行的 `deleteUser` 方法来处理。从第 19 行还能看到，这里是通过 `/deleteUser/{id}` 的方式来传入待删除的用户 ID 号。

在第 21 行，是通过 `userService` 的 `delete` 方法来删除用户的，之后将通过第 22 行的 `return` 代码，跳转到 `/displayUsers.jsp` 这个页面（这里会根据解析处理器的配置，给请求加上 `.jsp` 后缀）。

```
24     @RequestMapping("/getUser/{id}")
25     publicString getUser(@PathVariable String id, HttpServletRequest request){
26         User user = userService.getUser(id);
27         request.setAttribute("user", user);
28         return"/editUser";
29     }
```

从第 24 行的 `@RequestMapping` 的设置来看，一旦有 `/getUser` 这样的请求，就将交给第 25 行的 `getUser` 方法来处理。这里同样是通过 `/getUser/{id}` 的方式来传入待查询用户的 ID 号。

在第 26 行，是通过 `userService` 的 `getUser` 方法来得到用户。在第 27 行，通过 `request.setAttribute` 方法把这个 `user` 对象放入 `user` 这个属性，并通过第 28 行的 `return` 代码，跳转到 `/editUser.jsp` 这个页面（这里同样会根据解析处理器的配置，加上 `.jsp` 后缀）。

```
30     @RequestMapping("/modifyUser")
31     public String modifyUser(User user) {
32         userService.modifyUser(user);
33         return "redirect:/displayUsers";
34     }
35 }
```

从第 30 行的 `@RequestMapping` 的设置来看，一旦有 `/modifyUser` 这样的请求，就将交给第 31 行的 `modifyUser` 方法来处理。

在这个方法的第 32 行，是通过 `userService` 的 `modifyUser` 方法来修改用户，并通过第 33 行的 `return` 代码，跳转到 `/displayUser.jsp` 这个页面（这里同样会根据解析处理器的配置，加上 `.jsp` 后缀）。

10.2.7 编写 Service 层的代码

在控制器 `UserController` 类里，会调用 `UserService` 里的方法来实现添加用户等操作。

`UserService` 是 `Service` 层的代码，在实际项目中，一般在其中存放业务代码，而不会放置操作数据库的代码，这样做的目的是隔离业务逻辑和数据库访问逻辑。

`Service` 类起到的是一个承上启下的作用，封装在其中的方法会被前端或控制器部分的代码调用，而它一般会调用 `DAO` 层的代码实现针对数据库层的操作。

这里的 `UserService` 代码如下所示。

```
1  //省略必要的 package 和 import 代码
2  public class UserService {
3      //通过注解的方式引入 userDao 对象
4      @Resource(name="userDao")
5      private UserDao userDao;
6      public void setUserDao(UserDao userDao) {
7          this.userDao = userDao;
8      }
9      //添加用户的方法
10     public void addUser(User user) {
11         userDao.addUser(user);
12     }
13     //获得所有用户的方法
```

```

14     publicList<User> getAll() {
15         return userDao.getAll();
16     }
17     //根据 id 删除用户的方法
18     publicvoid deleteUser(String id) {
19         userDao.deleteUser(id);
20     }
21     //根据 id 得到指定用户的方法
22     publicUser getUser(String id) {
23         return userDao.getUser(id);
24     }
25     //修改用户的方法
26     publicvoid modifyUser(User user) {
27         userDao.modifyUser(user);
28     }
29 }

```

在这个类里，定义了诸多针对用户的操作方法，所有的方法都无一例外地调用 `userDao` 这个 DAO 类来实现数据库层的操作。

而且，根据 `spring-orm.xml` 里关于事务的定义，定义在 `UserService` 类里的相关方法都将适用于事务，从中我们能看到声明式事务的优点，在具体的代码里看不到任何关于事务的痕迹，这样一旦需要取消事务或者变更事务的设置，将不会影响到具体的 `Service` 方法。

10.2.8 编写 DAO 层的代码

我们一般把针对数据库层的操作放入 DAO 层的代码，换句话说，在其他的层面（比如控制器层、拦截器层和 `Service` 层），将看不到任何数据库操作的痕迹。

这样做的好处是，一旦有针对数据库层的修改，就可以把这个修改只限定在 DAO 层，而一些不相干的业务代码不会受到影响。

`userDAO` 部分的代码如下。

```

1     //省略必要的 package 和 import 方法
2     public class userDao {
3         //在配置文件里，我们已经在 userDao 里注入了 sessionFactory
4         private SessionFactory sessionFactory;
5         publicvoid setSessionFactory(SessionFactory sessionFactory) {
6             this.sessionFactory = sessionFactory;
7         }
8         //通过 Session 的 save 方法添加用户
9         publicvoid addUser(User user) {

```

```

10      sessionFactory.getCurrentSession().save(user);
11    }
12    public void deleteUser(String id) {
13        String hql = "delete from User where id=?";
14        Query query = sessionFactory.getCurrentSession().createQuery(hql);
15        query.setString(0, id);
16        query.executeUpdate();
17    }

```

第 12 行到第 17 行，在 `deleteUser` 方法里定义了删除用户的操作。其中，在第 13 行定义了删除的语句，第 15 行设置了待删除用户的 `id`，之后通过第 16 行的 `executeUpdate` 方法完成了删除操作。

```

18 public List<User> getAll() {
19     String hql = "from User";
20     Query query = sessionFactory.getCurrentSession().createQuery(hql);
21     return query.list();
22 }

```

第 18 行到第 22 行，在 `getAll` 方法里定义了得到所有用户的操作。其中，在第 16 行定义了得到用户的 HQL 语句，随后在第 21 行通过 `query.list` 方法以 `List<User>` 的形式，得到并返回所有的用户。

```

23 public User getUser(String id) {
24     String hql = "from User where id=?";
25     Query query = sessionFactory.getCurrentSession().createQuery(hql);
26     query.setString(0, id);
27     return (User) query.uniqueResult();
28 }

```

第 23 行到第 28 行，在 `getUser` 方法里定义了得到指定用户的操作。

其中，在第 24 行定义了得到指定用户的 HQL 语句，在第 26 行设置了指定用户的 `id`，随后在第 27 行通过 `query.uniqueResult` 方法得到并返回指定的用户。

```

29 public void modifyUser(User user) {
30     String hql = "update User set name=?,age=? where id=?";
31     Query query = sessionFactory.getCurrentSession().createQuery(hql);
32     query.setString(0, user.getName());
33     query.setInteger(1, user.getAge());
34     query.setInteger(2, user.getId());
35     query.executeUpdate();
36 }
37 }

```

第 29 行到第 36 行，在 `modifyUser` 方法里定义了修改用户的操作。

其中，在第 30 行定义了得到修改用户的 HQL 语句，并通过第 32 行到第 35 行的语句设置了待修改用户的 `id`、`name` 和 `age` 值，最后在第 35 行通过 `query.executeUpdate` 修改用户。

10.3 描述商业项目案例经验

目前比较多的项目会用到 Spring MVC 的框架（或者有些项目会在此基础上再封装一层，说到底也是 MVC），虽然实际的商业项目要比这个案例复杂得多，但是均能在本案例提供的增删改查的功能基础上扩展。这里就拿我们经常提问的问题为例，来向大家说明下如何描述基于 Spring MVC 的项目经验。

在面试的刚开始，我们会问些基本的项目情况，一般会问，你最近的项目（或者你感觉做得最好的项目）用到哪些框架？数据库是用什么？用了多久？有多少人在做？

这个属于描述综合项目，我们不关心具体的业务，只关心框架技术。大家可以这样回答，我的项目用到了 Spring MVC（顺便把 Spring MVC 介绍一下），数据库用的是 Oracle（或 MySQL），通过 Hibernate（或 MyBatis 等）来关联数据库。

然后最好大致提下项目里用到的各个技术，因为你不说我们后面也会问。比如可以说，在项目里，我们用到了拦截器和声明式事务，在项目里，我们把业务方法封装到 Service 层，把和数据库关联的操作放到 DAO 层。

最后可以简单说一下你做的事情，比如可以这样回答，我做的是 xx 和 xx 模块，其中有 xx 亮点。

这个属于开放性问题，没答案没边界，回答时请避免两种极端情况。第一类是说得太过简单，需要我们不断追问，对于这种候选人，如果后继面试还这样，我们就要写上“表达能力不强”之类的评语了。第二种是过于泛泛而谈，回答时跳跃性太强，思路不连贯，这样我们就可能要写上“思路不清晰（甚至是思路有些混乱），逻辑性不强”之类的评语了。

在了解大致项目情况后，我们会让候选人结合他在项目里做过的事情，具体描述下 Spring MVC 的流程，具体的问题可以是，请结合你开发过的一个实际例子，描述 Spring MVC 从前端到后端的工作流程。

通过这个问题，我们不仅想要考查候选人是否掌握了 Spring MVC 的流程，还想了解候选人主要用过 Spring MVC 里的哪些组件。

大家可以用这里给出的删除用户为例（任何项目都会有删除的业务需求），可以说，一旦

发生删除动作时，前端会以 post（或 get）的方式发送请求，这个请求先会被拦截器处理，在我们的拦截器代码里，做了 xx 事情。经过拦截器后，会根据针对控制器方法的 `@RequestMapping` 配置，把这个请求交由特定的控制器方法来处理，处理后会由视图解析器解析处理该方法的返回值，最终返回到指定的页面。

随后，我们会具体问些 Spring MVC 框架的基础技术，只要候选人用过 Spring MVC，就应该能说上来，以此来确认候选人对这套框架的熟悉程度。经常问的技术问题如下。

① 如何标识控制器类？（可以通过 `@Controller` 注解）

② 在控制器方法里重定向有两种方式，一种是 forward，另一种是 redirect，有什么差别？

③ 你一般是通过实现什么方法来实现拦截器，一般在拦截器里，你会重写哪些方法？在拦截器的方法里，你们的项目放的是什么业务？

④ 在拦截器的 `preHandle` 方法里，如果我们不想让这个请求继续往下走（也就是说想拦截这个请求），该如何处理？

⑤ 你的项目里有多少个控制器类？如果一个项目有多个不同的控制器类，该怎么把请求定位到指定的控制器类上？

⑥ 你用的是哪个视图解析器？其中该怎么给请求加后缀？

大家可以阅读第 6 章来回顾下第 2 题到第 6 题该怎么回答。

在问完 Spring MVC 框架后，我们会问一些关于数据库和 ORM 的知识点，经常问到的问题如下：

① 你是用注解的方式还是配置文件的方式来描述数据表和 Model 类的对应关系？这两个方式各有什么优缺点？

② 在你们项目里，是怎么管理主键的？你知道的生成主键的方式有哪些？

③ 结合一个实际案例说明一下你们项目里如何管理一对一、一对多和多对多的关系。我们可能还会扩展问到 `inverse` 和 `cascade` 这两个关键字的用法。

④ 你们项目里有没有用到事务？用哪个类来实现事务的管理？用到的是编程式事务还是声明式事务？这两种事务的实现方式有什么差别？或者如果用到的是其他的事务管理方式，也请说一下怎么用的。

⑤ 事务的传播机制有哪些？它的作用是什么？我们会经常用到 `REQUIRED` 这个值，它是用什么样的方式来管理事务？除此之外，你们项目还用到过哪些值？有时我们还会把事务

隔离级别和事务传播方式结合起来提问，比如问它们各自适用于哪些场合？

⑥ 你们的项目里，怎么实现 Spring 和 Hibernate 的整合？

⑦ 你的项目里，是通过 HQL 还是 SQL 来获取数据？或者说，请结合你项目里的实际做法，告诉我们哪些场景下会用到 HQL，哪些场景下用到了 SQL？

上述问题我们都分析过，从中大家可以看到，只要你真的做过 Spring MVC 框架，而且平时能不断总结，大致就没什么问题了。

可能在有些项目里会采用些其他的 ORM 的方式来管理数据表和 Model 类之间的映射关系，比如会用到 MyBatis 或者是 JPA，在其中可能还直接通过注解来注入 SQL 语句，但请注意，万变不离其宗，各种 ORM 技术都是相通的，我们不在乎你在之前的项目里用到的是哪种技术，但我们需要确认你对此有一定的了解。

在问完 Spring MVC 框架和数据库 ORM 后，我们还会问一些前端的问题，对于后端开发者而言，这部分的问题你回答不上也不要紧，但如果你知道，这就是加分项。

① 你知道 Post 和 Get 两种请求的方式有什么差别吗？结合具体的实际项目分别举两个例子来说明。

② Restful 格式的 URL 是什么？有什么好处？

③ 你有没有用过 Ajax，Ajax 一般是怎么开发的？

④ 同样的一套前端代码，但它在不同种类的浏览器里渲染的效果会不同，你有没有遇到过这种情况？遇到后你是怎么解决的？

最后，我们还会问些诸如性能调优和项目亮点之类的高级问题，虽说这些问题是适用于高级程序员的，但对于初级程序员而讲，你多少也该知道些。

① 在数据库层面，你们做了哪些优化动作？如果针对某个业务，SQL 语句非常复杂，你们该怎么办？

② 你们平时是怎么监控数据库的，比如出现了长时间运行的 SQL 语句，或者死锁情况，或者表分区不足，你们是怎么知道的？一旦出现这些问题，你们是怎么解决的？

③ 你们项目有没有用到缓存？

④ 你们项目是怎么监控内存的？你们项目有没有遇到过 OOM 异常，遇到后该怎么分析、怎么解决？

⑤ 你们网站的访问量是多少？如果遇到高并发、大访问量的情况，你们具体做了哪些优

化措施？

⑥ 说说你在项目里具体解决了哪些技术难题？

对于这类问题，不少人直接告诉我们没有，这就失去了一个展示自己分析解决问题的机会，我其实不在乎你具体解决了什么问题，但公司一般都想招一些具有实际分析解决问题能力的人进来。

第 11 章

简历面试那些事

公司是通过筛选简历和面试来确认候选人能力的，如果确定要录用了，再通过背景调查来核实候选人在简历上写的信息。在此过程中，筛选简历的人事和面试官没有义务来帮你挖掘亮点，讲得再实际一些，在候选人无法自证是否达标时，一般认为是没有，所以有不少技能达标的人倒在了面试路上，同样也有不少技能欠缺但会面试的人“涉险”过关。

如果是因为面试方法不对而错失良机，那么就太可惜了。当然，大家多参加几次面试，也一定能“久病成良医”，不过时间就耽搁了。

笔者在这里，将根据面试多个候选人（至今有 100 以上）的经验，来向大家展示各种关键技能，从而帮助大家完成最终的临门一脚。

11.1 不要让你的简历进回收站

找工作时可以多投简历，这样能获得足够多的面试机会，一般来说，平均每投 5 份简历至少能获得一次面试机会，如果你没有达到这个比例，而且你的学历、经验、技能等硬指标不算差的话，那么你的简历就有问题了。

11.1.1 面试的基本流程

知己知彼，百战不殆，这里我们来讲述一下面试的一般流程。

首先是人事筛选简历，人事一般不熟悉具体技术，但会根据业务部门给出的关键字来筛选。

比如有个职务是 Java 后端开发工程师，要求是本科以上，最好是 985 或 211 大学，至少有 1 年经验，要有 Spring MVC 或类似经验，最好有 Hibernate 经验，有 Sonar、Jenkins 的优先考虑，因为项目是保险背景的，所以有这方面经验的优先考虑。

那么人事就会根据必备项和加分项综合比较，把满足要求的简历留下，并根据综合条件做个排序，交给技术面试官，面试官开始优先面试条件好的。

随后是技术面试，最常见的是两轮：一轮电话面试，一轮现场面试。如果通过了，那么项目经理和部门经理会介入（如果公司规模较小，则老板或者副总会直接聊）。项目经理和部门经理一般不会再问技术了，而是会从稳定性，人际交往沟通能力和职业规划等方面看看候选人是否适合这个岗位。

如果项目经理和部门经理觉得可以，那么人事会再出面来谈工资和到岗时间等细节，之后就可以安排入职了。

11.1.2 根据职务介绍再针对性地准备简历

从上文描述的过程来看，为了更高效地得到面试机会，最好根据每份不同的工作描述准备对应的简历。

我们和一些学员沟通后发现，针对不同的公司，大多数人会用同一份简历。要知道，一个商业项目持续时间很长，在简历上你可能只列了其中的重要技术，但不同的工作岗位对技术要求的侧重点肯定不同的。

比如某人做了 1 年的 Spring MVC 项目，在简历上他侧重描述框架，但某个职位描述上写明了“有数据库调优经验的优先考虑”，虽然他也做过，但没写到简历上，那么就很不吃亏。

所以在投每份简历前，请务必先“审题”，根据不同的需求微调你的简历，挖掘出你和经验和职位需求的匹配项。

11.1.3 哪类简历比较难获面试机会

在表 11.1 里，列出了一些很难得到面试机会的简历。

如果出现了如表 11.1 所示的情况，而且你也没有额外的弥补项，比如学校特别好，项目经验特别丰富，有海外工作经验，或者有英语环境的工作能力等（一般刚毕业或工作经验不长的人不会有这些特长项），那么你得到面试机会的几率就很低。

表 11.1 很难得到面试机会的简历原因分析表

问题点	很难获得面试机会的原因
学历不符，比如要求是本科以上，但学历是大专	学历是硬指标，所以达不到学历要求的一般很难得到面试机会
相关工作经验严重欠缺，比如 ①要求是 3 年以上经验，但简历上才 1 年。 ②要求 Java 后端有 3 年经验，虽然简历上有 3 年，但只有 1 年是 Java 后端	公司需要招进来的人要立即能干活，其中一个重要的考查指标是工作年限，所以年限不足，一般不会考虑
最近半年的工作经验和职务描述不匹配	由于最近没有相关技能经验，那么相关经验就会生疏，而公司要求是入职就能干活的
投简历时看上去很敷衍，比如就一份邮件带一份简历，邮件正文里也没有求职信之类的文字	如果遇到比较挑剔的，会认为这个人没有足够的尊重感，如果有可以替代的，估计很难得到面试机会
从简历上的项目信息等有效信息上，无法看出符合职务需求	技能不匹配
简历上大量充斥和工作无关的描述，比如我们看到有些毕业生的简历超过 5 页，前 2 页会写从高中到大学的生长经历，再用 2 页写性格分析、兴趣爱好还有学生会工作介绍，最后 1 页才是技能和实习经历	用在关键信息上的篇幅较少，所以能展示的有效信息也少，这样会导致人事认为你的技能不匹配
简历上显示频繁换工作，比如 ① 大多数工作持续时间不到半年 ② 1 年换了 2 份以上或 2 年换了 3 份以上工作 ③ 最近一份工作还没干半年就又开始换工作了	公司一般需要比较稳定的员工，经常换工作说明这个人能力有问题或是稳定性不强

11.1.4 准备简历时该注意哪些

第一，简历需要短小精悍，别做成花花绿绿的，黑白打印即可。

第二，人事（或相关人员）在筛选简历时，往往要在比较短的时间里阅读很多简历，所以平摊到每份简历上的阅读时间不会很长。对此，建议大家可以根据职位介绍，然后在第一页里写上综合能力介绍，让阅读者能第一时间感觉你和这个职位很般配，这样的话他就有读下去的意愿了。综合介绍可以采用如下形式。

- 有 xx 年多软件开发工作经验，有 xx 年后端经验，有 xx 年银行等方面的金融项目经验。
- 技能方面，熟悉 xx、xx 技术。
- 数据库方面，有 xx 年以上 Oracle 调优经验，用过 xx 数据库。
- 框架方面，用过 xx 框架。
- 外语方面，能独立和老外开会。
- 其他能帮助你申请到该职位的亮点

第三，出于和第二点相同的理由，工作过的公司和项目经验倒叙写，让读者第一时间就能看到你最近的项目和公司。

第四，在你的项目描述等方面，多列出和申请职位需求相匹配的关键字，第一眼筛选简历的人事未必熟悉技能，你列出的关键字要尽量和需求描述上的一样。

比如需求上写，要有 3 年以上 Java 核心开发经验，你就别写，有 3 年以上 Java Core 开发经验，虽然核心开发和 Core 开发是一回事，但万一筛选简历的人不是专业人士，那么你可能就吃亏了。

11.2 面试之前，你要做哪些准备

面试官是人，不是神，拿到你的简历的时候，是没法核实你的项目细节的（一般公司会到录用后，用背景调查的方式来核实）。

你在面试的时候，不需要告诉面试官你以前干过的项目细节，因为面试官根本不关心，而是需要展示你以前项目里用过的技术、框架，以及你对这些技术的熟悉程度，面试官会由此来确认你是否适合这个岗位。

11.2.1 准备项目经验描述，别害怕，因为面试官什么都不知道

面试官会通过交叉提问的技巧来验证你的项目和技能描述，所以大家别抱有任何蒙混过关的想法。不过，由于信息不对称，相对而言你优势的会大些。从表 11.2 中，你能看到你的优势。

表 11.2 面试时，你和面试官的情况分析

	你	面试官
对你以前的项目和技能	很了解	只能听你说，只能根据你说的内容做出判断
在面试过程中的职责	在很短的时间内防守成功即可	如果找不出漏洞，就只能算你以前做过
准备时间	面试前你有充足的时间准备	一般在面试前用 10 分钟阅读你的简历
沟通过程	你可以出错，但别出关键性的错误	不会太为难你，除非你太差
技巧	你有足够的技巧，也可以从网上找到足够多的面试题	其实就问些通用的有规律的问题

所以你根本没必要紧张，假设候选人的工作经验比面试官还丰富的话，甚至还可以控制整个面试流程。

11.2.2 面试官的策略——如何通过提问，找出你回答中的矛盾

既然面试官无法了解你的底细，那么他们怎么来验证你的项目经验和技术？表 11.3 里总结了一些常用的提问方式。

表 11.3 面试官的基本提问方式

提问方式	目的
让你描述工作经验和项目（极有可能是最近的），看看你说的是否和简历上一致	看你是否真的做过这些项目
看你简历上项目里用到的技术，比如框架、数据库，然后针对这些技术提些基本问题	还是验证你是否做过项目，同时看你是否了解这些技术，为进一步提问做准备
针对某个项目，不断深入地问一些技术上的问题，或者从不同侧面问一些技术实现，看你前后回答里面是否有矛盾	深入核实你的项目细节
针对某技术，问些项目里一定会遇到的问题，比如候选人说做过数据库，那么就会问索引方面的问题	通过这类问题，核实候选人是否真的有过项目经验（或者还仅仅是学习经验）

11.3 面试陈述篇：充满自信地描述你的项目经验

在你被提问前，一般会让你介绍最近的项目经验，这部分将直接关系到后面的问题。既然面试官不知道你底细，那么项目描述部分，你说什么，那就是什么。

自信些，因为这部分你说了算，流利些，因为你经过充分准备后，可以知道你要说些什么。而且这些是你实际的项目经验（不是学习经验，也不是培训经验），那么一旦让面试官感觉你都说不上来，那么可信度就很低了。

11.3.1 准备项目的各种细节，一旦被问倒了，就说明你没做过

不少人是拘泥于“项目里做了什么业务，以及代码实现的细节”，这就相当于把后继提问权直接交给面试官。表 11.4 列出了一些不好的回答方式。

表 11.4 描述项目经验时一些不好的回答

回答方式	后果
我在 XX 软件公司做了 XX 门户网站项目，这个项目做到了 XX 功能，具体是 XX 和 XX 模块，各模块做了 XX 功能，客户是 XX，最后这个项目挣了 XX 钱	直接打断，因为业务需求我不需要了解，我会直接问他项目里的技术
（需要招聘一个 Java 后端开发，会 Spring MVC） 最近一个项目我是用 C#（或其他非 Java 技术）实现的，实现了……或者我最近做的不是开发，而是测试……或者我最近的项目没有用到 Spring MVC	提问，你最近用到 SSH 技术的项目是什么时候，然后在评语上写：最近 XX 时间没接触过 SSH
在毕业设计的时候（或者在读书的时候，在学习的时候，在 XX 培训学校，在 XX 实训课程中），……	直接打断，提问你这个是否是商业项目，如果不是，你有没有其他的商业经验。如果没商业项目经验，除非是校招，否则就直接结束面试
描述项目时，一些关键要素（比如公司、时间、所用技术等）和简历上的不匹配	我们会深究这个不一致的情况，如果是简历造假，那么可能直接中断面试，如果真的是笔误，那么就需要提供合理的解释

大家可以按表 11.5 所示的步骤说出项目关键性的要素，如果可以，你准备一下用英语描述。其实刚毕业的学生，或者工作经验较少的人，英语能力都差不多。

我们在外企做面试官的时候，如果遇到不愿说英语或无法用英语回答完问题的候选人，就会写上“不能用英语表达”之类的评语，英语能力在外企是个硬指标，这种人一般就很难得到外企的 offer 了。不过候选人只要说了，而且能完成简单的交流（问题无非是介绍上个项目，介绍上个公司等常见问题），最差也能得到“有基本的英语沟通能力”这个评价。

表 11.5 描述项目经验时需要包含的基本要素

要素	样式
控制在 1 分钟里面讲出项目基本情况，比如项目名称，背景，给哪个客户做，完成了基本的事情，做了多久，项目规模多大，用到哪些技术，数据库用什么，然后酌情简单说一下模块。重点突出背景，技术，数据库与其他和技术有关的信息。	我在 XX 公司做了 XX 外汇保证金交易平台，客户是 XX 银行，主要完成了挂盘、实盘成交以及保证金杠杆成交等功能，数据库是 Oracle，前台用到 JS 等技术，后台用到 Java 的 SSH，几个人做了 X 个月。不需要详细描述各功能模块，不需要说太多和业务有关但和技术无关的。如果面试官感兴趣，等他问

续表

要素	样式
要主动说出你做了哪些事情，这部分的描述一定要和你的技术背景一致	我做了外汇实盘交易系统，挂单成交系统，XXX 模块，做了 X 个月
描述你在项目里的角色	我主要是做了开发，但在开发前，我在项目经理的带领下参与了业务调研、数据库设计等工作，后期我参与了测试和部署工作
可以描述用到的技术细节，特别是你用到的技术细节，这部分尤其要注意，你说出口的，一定要知道，因为面试官后面就是根据这个提问的。 你如果做了 5 个模块，宁可只说你最熟练的 2 个	用到了 Java 里面的集合、JDBC……等技术，用到了 Spring MVC 等框架，用技术连接数据库
这部分风险自己承担，如果可以，不露声色地说出一些热门的要素，比如 Linux、大数据、大访问压力等。但一旦你说了，面试官就会直接问细节	这个系统里，部署在 Linux 上，每天要处理的数据量是 XX，要求是在 4 小时，1GB 内存的情况下处理完 5 千万条数据。平均访客是每分钟 XXX

面试前，一定要准备，一定要有自信，但也要避免如下一些情况。

表 11.6 描述项目经验时需要避免的情况

要避免的情况	正确的做法	原因
回答很简单。问什么答什么，往往就用一句话回答	把你知道的都说出来，重点突出你知道的思想、框架	问：你 SSH 用过吗？ 答：用过。 问：在什么项目里用到？ 答：一个保险项目。 问：你做了哪方面的事情？ 答：开发。 我直接不问了
说得太流利	适当停顿，边思考边说	让面试官感觉你在背准备的东西，这样后面问题就很难
项目介绍时什么都说	就说些刚才让准备的一些，而且要有逻辑地说	会让面试官感觉你思路太乱
别太多介绍技术细节，就说你熟悉的技术	技术面点到为止，等面试官来问	你说到的所有技术要点，都可能会被深问。面试官一般会有自己的面试节奏，如果你在介绍时就说太多技术细节，则很有可能被打断，从而没法说出你准备好的亮点

11.3.2 充分准备你要说的项目的框架、数据库

当你了解项目介绍时的注意要点后，你就可以在面试前准备，在准备的时候，一定要对你项目的框架、数据库等关键要素准备充分。

下面列出一些框架方面可能会问到的问题：

- ① 这个项目用到哪些框架？通过一个模块，说明一下框架的流程。
- ② 你是否参与过搭建框架？
- ③ 做下来以后你对这个框架有什么了解？有什么优点和缺点？
- ④ 你感觉这个框架的扩展性如何？

数据库方面的问题：

- ① 数据库用的是哪个？为什么要用这个数据库？
- ② 项目里是怎么连接数据库的？
- ③ 整个项目数据量有多大？
- ④ 你有没有处理过大数据量情况？如果数据量大了，你怎么办？
- ⑤ 你有没有参与数据表的设计？你用什么原则来设计数据表结构？
- ⑥ 你是怎么用数据库的？是简单的增删改查还是有其他的？
- ⑦ 数据库优化方面，你做了哪些事情？有哪些经验？
- ⑧ 如果做过大数据技术，一定会问到这个。

一旦遇到开放性的提问，不论你说什么，都可以遵循表 11.7 所列出的原则。

表 11.7 回答框架和数据库方面问题时可以遵循的原则

等级	数据库方面的要求 We	b 框架方面的要求
高级	<div>①会设计数据表，能说出设计表的原则：别用 3 范式，采用尽量少连接的设计原则。</div> <div>②知道对数据库的优化操作，比如索引、集群、读写分离，而且知道怎么监控数据库，从而知道哪些语句该优化</div> <div>③知道大数据量下的处理方法，比如建索引、SQL 语句调优、缓存等</div> <div>④知道诸如 Oracle、MySQL 的高级应用</div> <div>⑤知道不同数据库下，锁、事务等高级应用</div> <div>⑥最好了解大数据相关的技术</div>	<div>①在项目里自主搭建过框架</div> <div>②了解主流框架的优点和缺点，比如 Spring 有什么优缺点</div> <div>③知道如何让你的框架适应业务变更</div> <div>④参与过处理大数据、高并发的框架</div>

续表

等级	数据库方面的要求 We	b 框架方面的要求
初级	①知道设计数据表的注意点 ②知道简单的索引、存储过程、锁、事务。 ③对一种数据库了解比较深，不仅仅是简单的增删改查 ④能写出相对复杂的 SQL 语句 ⑤有基本的数据库优化技能	①能熟练搭建一个简单的框架 ②知道框架的调用流程 ③会通过继承、面向对象等方法，适当改写框架
勉强能找到工作	①只会基本的增删改查 ②没听说过大数据量处理 ③根本不知道 SQL 调优	只能在项目经理搭建好框架的基础上干活，不会更改框架

11.3.3 不露痕迹地说出面试官爱听的话

在项目介绍的时候（当然包括后继的面试），面试官其实很想听一些关键点，只要你说出来，而且能很好地回答后继的提问，这绝对是加分项。

我们在面试别人的时候，一旦这些关键点得到确认，是绝对会在评语上加上一笔的。

表 11.8 能切实帮到你的加分项

关键点	说辞
能考虑到代码的扩展性，有参与框架设计的意识	我的 XX 保险项目，用到 SSH 技术，数据库是 Oracle（这个是铺垫），开发的时候，我会先和项目经理一起设计框架，并参与了框架的构建。连接数据库的时候，我们用到了 DAO，这样做的理由是，把 SQL 语句封装到 DAO 层，以便在扩展功能模块时，可以不用做太多的改动
有调优意识，能通过监控发现问题点，然后解决	在开发阶段，我就注意到内存的性能问题和 SQL 运行的时间问题，在压力测试阶段，我会通过 xx 工具来监控内存和数据库，发现待提升的代码点，然后通过查资料来优化。最后等项目上线后，我们会部署监控系统，一旦发现内存和数据库有问题，我们会第一时间解决
动手能力很强，肯干活，会的东西比较多，团队合作精神比较好	在项目里，我不仅要开发的工作，而且需要自己测试，需要自己根据一些日志的输出到数据库或 Java 端去 debug。当我开发一个模块时，需要自己部署到 Linux 上测试或者，一旦遇到问题，如果是业务方面的，我会及时和项目经理沟通；如果是技术方面的，我会自己查资料；如果是测试方面的，我会及时和测试的人沟通
责任心比较强，能适应大压力的环境	被问“你如果在项目里遇到问题怎么办？” 回答：遇到问题我先查资料，如果实在没法解决，不会拖，会及时问相关的人，即使加班，也会在规定的时间内解决
有主见，能不断探索新的知识	在项目里，我会在保证进度的前提下和项目经理说出我的想法，提出我的解决方案。在开发过程中，我会先思考一下，用一种比较好的方式，比如效率最高的方法实现 另外你要找机会说出：平时我会不断看一些新技术（比如大数据 Hadoop），会不断深入了解一些框架和技术的实现底层

11.3.4 一定要主动，面试官没有义务挖掘你的亮点

我们去面试应聘者的时候，往往会特别提问：你项目里有什么亮点？或者你作为应聘者，有什么其他加分项能帮你成功应聘到这个岗位。即使这样问，还是有些人直接说没有。

我们这样问已经是处于角色错位了，作为面试者，应当主动说出，而不是等着问，但请注意，说的时候要有技巧，找机会说，通常是找一些开放性的问题说。

比如：在这个项目里用到了什么技术？你除了说一些基本的技术，比如 Spring MVC、Hibernate，还有数据库方面的常规技术时，还得说，用到了 Java 内存管理，这样能减少对虚拟机内存的压力，或者说用到了大数据处理技术等。也就是说，得找一切机会说出你拿得出手的而且当前也非常热门的技术。

或者找个相关的问题做扩展性说明，比如被问到：你有没有用到过一对多和多对多？你除了说出基本知识点外，还可以说，一般我还会根据需求适当地设置 `cascade` 和 `inverse` 关键字，随后通过一个实际的案例来说明合理设计对你项目的帮助，这样就能延伸性地说明你的技能了。相反，如果你不说，面试官一定会认为你只会简单的一对一和一对多操作。

面试的时候，如果候选人回答问题很简单，有一说一，不会扩展，或者用非常吝啬的语句来回答我们提的问题，那么我们一般会给他们机会让他们深入讲述（但也许不是每个面试官都会深入提问），如果回答再简洁，那么也会很吝啬地给出好的评语。

记住：面试官不是你的亲戚，面试官很忙，能挖掘出你的亮点的面试官很少，而说出你的亮点是你的义务。

我们在面试应聘者的过程中，会根据不同的情况给出不同的评语。

① 回答很简单，但回答里能证明出他对框架等技术确实用过，评语里会写道：“对框架了解一般，不知道一些深层次的知识（我们问了多次了，你都回答很简单，那么对不起了，我们只能这么写，或许你确实技术很强，那也没办法，谁让你不肯说呢？）”，同时会加一句“表达能力一般，沟通能力不强”。这样即使他通过技术面试，后面的面试他也会很吃力。

② 回答很简单，通过回答我们无法验证他是否在项目里做过这个技术，还是仅仅在平时学习中学过这个技术。就会写道：“在简历中说用过 XX 技术，但对某些细节说不上来，无法看出在项目里用到这个技术”。如果这个技术是职务必需点，那么他通过面试的可能性就非常小。

③ 回答很简单，而且只通过嗯啊之类的虚词回答，经过提醒还这样，我们会敷衍几句结束面试，直接写道：“技术很薄弱，没法通过面试”。

④ 虽然通过回答能很好地展示自己的技能，但逻辑条理不清晰，那么我们会让他通过技术面试，但会写上“技能很好，但表达能力一般，请后继面试经理斟酌”。这样通过后继综合面试的可能性就一般了，毕竟综合面试会着重考查表达能力、交往能力等非技术因素。

不管怎样，一旦回答简单，不主动说出你的擅长点，或没有条理很清楚地说出你的亮点，就算我们让你通过面试，也不会写上“框架细节了解比较深，数据库应用比较熟练”等之类的好的评语，即使通过技术面试和后面的综合面试，工资也是比较低的。

11.3.5 一旦有低级错误，可能会直接出局

面试过程中有些方面你绝对不能出错，所以在准备过程中需要尤其注意如下因素。

表 11.9 会导致你直接出局的错误列表

错误类型	导致的后果
前后矛盾，后面的回答无法证明你的项目描述，比如一开始说用到了 Spring MVC，后面没法说出最基本的实现，比如不知道 Spring 有哪些类，或者没法说出项目的细节	我们会怀疑这个项目的真实性，从而会进一步问：数据库用什么，数据量多少？多少人做了多少时间，一旦再出现明显漏洞，比如一个小项目用到非常多的时间，那么就不仅仅是技术问题，而是在面试过程中企图“蒙混过关”的性质了
项目里一定会用到的基本概念性问题都回答不上，Spring 的依赖注入概念是什么，怎么用的，或者 Hibernate 的一对多怎么实现	一旦被我们发现概念不知道，我们会通过更多问题确认，如果我们确认很弱，这就相当严重，因为技术能力差和技术没用过是两个截然不同的状况，技术没用过会导致直接出局
面试时说出的工作经验和简历上的不一致	我们会直接怀疑简历是编的，我们会让候选人解释，即使是说简历写错了，我们也会问比较深入的问题来核实他的技术和能力
简历上的技能描述和回答出来的明显不一致，比如明明只会简单的 Linux，但吹得天花乱坠	我们会通过一些比较深的问题核实其他技能，找出其他方面吹嘘的水分 所以建议，你可以适当夸张，但别过分，比如你在项目里没搭建框架，但平时学习时搭建过，你可以写“XX 项目的框架是你搭建的”，但你不能说你是一个架构师，非常了解项目的底层
让面试官感觉你不稳定，很浮躁，比如说话不稳重，口头禅太多，或者面试时打扮非常不正规，穿着背心来的	即使你技术再好，这也可能会导致你直接出局 我们对油嘴滑舌的候选人一般会直接写上不好的评语，这样很难过后面项目经理的面试 我们遇到过这样的一个人，简历上写明工作是半年一换，我们问他为什么经常换，他直接说是待遇问题，这种人我们是让他直接出局
明说不能加班，不能出差	其实虽然有这一问，但公司里未必真的会大量加班、会出差。但听到这类回答，说明这个人不能承受大压力的工作，或者责任心不强，大多数公司是不会要这种人的

11.4 面试引导篇：把问题引入准备好的范围

上文里作者列出了在项目介绍阶段你要说出的要素和你需要注意的事情。一般来说，面试过程中问题都会根据简历上的项目经验描述和第一阶段的项目介绍来展开，所以大家在准备简历和做项目介绍时，可以在一定程度上引导面试官的问题。

11.4.1 项目要素、框架设计和数据库，这些是必须要准备的

你可以引导问题，并不是意味着你可以逃避一些问题，刚才提到的一些框架、数据库等因素，你即使不说，也会被问。下面给出一些框架方面必然要考查的知识点。

表 11.10 框架方面的必问点

问题点	回答要点
你 XX 项目用到什么框架	从前台到后台都要回答，重点说你做的，比如前台用 JS，后台用 SSH，还用到 XX 开源框架
你是否参与搭建过框架	最好说是搭建过，如果你确实是在项目经理的带领下开发的，没参与搭建，你也可以在了解框架细节的基础上说搭建过
为什么要用这个框架	任何框架都要和项目匹配，比如一个项目很简单，用简单的 JSP+JDBC 即可，如果一个 Team 里都只会用 Spring，那干脆就用 Spring，记住，框架越复杂，成本越高，还未必匹配项目
在这个项目里，你大致说一下工作流程	一般是 MVC，你就举个业务上的例子，说前台怎么传输数据到后台，后台怎么得到结果返回给前台
这个项目有什么优点和缺点？或者有什么特点	优点一般是结合 Struts、spring、Hibernate 等的好处，比如 Spring 能靠依赖注入减少耦合，Struts 的好处是流程清晰，Hibernate 的好处是封装底层实现，具体的大家可以到网上再查些细节 缺点也请对应框架，比如 Struts 是侵入式的。然后再结合项目，比如好处是现成的技术，开发周期短，扩展性高，缺点是大数据处理不大好
你是怎么用到这个框架的	不仅要说编写代码，最好还说些参与搭建框架等。比如在搭建前，通过代码来测试这个框架的数据处理量、处理时间，了解诸如依赖注入等细节。搭建时，在项目经理带领下编写一些总体的 xml 文件，然后在开发过程中完成自己的部分
这个框架用下来有什么体会	因为我们面试的时候，发现不少人是为了用框架而用框架，其实框架只有在适合项目时，才能发挥价值，所以这里大家可以说，框架不是完成的，它抽象出了流程，实现了低耦合，但最好说出你自己感觉出来的框架的缺点，有什么不便利的因素

表 11.11 数据库方面的必问点

问题点	回答要点
你用到什么数据库？为什么要用这个	如实回答数据库，为什么要用 ①结合项目背景，一般银行、保险是用 Oracle，如果结合 IBM 的产品，就会用到 DB2，如果是微软产品就用 SQL Server ②你可以看一下招聘公司需要的数据库，从而了解这方面的细节 ③看项目需求，比如这个项目需要大数据，那么可以用 Oracle；如果开源因素比较多，可以用 MySQL
你是怎么用数据库的	①怎么用代码连接数据库，比如是用 JDBC 还是 Hibernate，这样做是为了什么目的，比如用 DAO 是为了抽象，用简单的 JDBC 是因为项目周期比较短 ②连接数据库的代码，你做了什么工作？比如代码是你写的，或者连接数据库的框架是你设计的 ③找机会说出连接数据库部分代码的亮点，比如 DAO 有什么好处，又比如你用到了 try...catch...finally，或者你用到了 Hibernate 里的缓存 ④有能力的话，一定要说出数据库调优方面的技能
你有没有参与设计数据表结构	如实回答。一般设计数据表需要考虑的因素： ①客户定的 ②由需求定 ③别用三范式，在设计的时候，尽量少让表做连接，因为一旦数据量很大，做连接就很消耗资源
怎么处理大数据量的情况	这是个开放性问题，请根据自己的能力酌情回答。如果工作经验比较少，可以说，是技术经理做了，你知道些。 ① 用开源的大数据的框架，比如 Hadoop ②在代码中，先把大数据装载到内存里 ③设计数据表少做连接 ④设计的时候酌情建索引 ⑤用连接池、集群或者其他负载均衡技术或架构

表 11.12 项目管理方面的必问点

问题点	回答要点
项目多大，做了多久，是给谁做的，以及其他基本因素	需要和简历上的匹配，特别是一些细节问题
在项目里你做了哪些事情	首先是开发，其实可以酌情说些调研需求、数据库设计、框架设计、测试、上线等事情，但一旦你说了，就得准备回答这方面的问题
你在项目里遇到问题怎么办	其实是考查你实际解决问题的能力，你可以说，先自己思考，然后实在无法解决就要问相关的人 千万不能说，没遇到问题，或者说很简单

续表

问题点	回答要点
项目背景是什么，具体有什么需求	这个你需要事先做好准备，了解面试公司是否有银行、支付、保险、门户网站、电商等背景，如果你在以前公司里有，一定要准备好，如果没有，可以直说，但可以说了解过一些业务知识
能不能加班，上个公司压力大不大，能不能出差	在面试的时候，不论项目组是否需要加班，是否需要出差，我们都会问这个问题，毕竟公司也想招一个能适应大压力的人。一般可以迎合公司说，但别说太过火

11.4.2 准备些加分点，在介绍项目时有意提到，但别说全

在回答必考点的时候，或者是在做项目介绍的时候，你可以穿插说出一些你的亮点，但请记住，不论在介绍项目还是在回答问题，你当前的职责不是说明亮点而是介绍项目，一旦你详细说，很可能会让面试官感觉你跑题了。

所以这时你可以一笔带过，比如你可以说，“我们的项目对数据要求比较大，忙的时候平均每小时要处理几十万条数据”，这样就可以把面试官引入“大数据”的方向。

你在面试前可以根据职位需求，准备好这种“一笔带过”的话。比如这个职位的需求点是 Spring MVC 框架，大数据、高并发，要有数据库调优经验，那么介绍以往项目时，最好突出你在这些方面的实际技能。

实在不行，你也可以说“我除了做开发，也做了了解需求、测试和部署的工作，因为这个项目人手比较少，压力比较大”，这样你也能展示你有过独当一面的经历。

我们在面试过程中，一旦听到有亮点，就会等到他说好当前问题后，顺口去问，一般技术面试最多半小时，你把时间用在回答准备好的问题点上的时候，被问其他问题的时间自然就少了。

11.4.3 对于面试官的圈套，别顺口回答

总结下来，面试官的圈套有如下几种情况：

① 当你在说项目基本情况时，我们会问一个和你简历上不匹配的问题（比如你上个项目是半年前做的？），了解你只用到 Spring，那我们就会问，你这个时间段的项目是否用到了 Struts 技术？

这种属于交叉提问，你既然写了只用到 Spring，那么就不该说还用到 Struts。

这个用来考查基本的项目情况，因为项目是你做的，你应该比谁都清楚，一旦你说出和简历不匹配的情况，我们就要你解释。

② 你在说用到某个技术时，我们会故意问一个本不该这个技术负责的问题。比如有人提到项目里就用到 Struts，没用 Spring，我们会问，你在项目里怎么实现模块间低耦合的，或者有没有通过依赖注入的方式来实现低耦合？

大家知道，实现模块间的低耦合是 Spring 的擅长点，而不是 Struts，其实这里是考查技术掌握程度和技术要点，一旦说错了，我们会认为你对 Struts 很不熟悉，从而会质疑你是否在项目里真的用过 Struts。

③ 针对某个技术点，故意给出错误的概念。比如某人说到建索引能提升数据表查找数据的速度，我们会问，在任何情况下，索引是不是都能提升数据库性能？

其实索引是适用于数据量比较大的情况，索引本身也是有代价的，如果数据量很少，用索引就得不偿失了。如果候选人对概念比较熟悉，或者真用过，就应该说，只在大数据量下用，即使在小数据下用，也要给出一个合适的说法。

④ 根据候选人错误的回答，推论性地给出错误的结论。比如某人告诉我们，在 Hibernate 里使用 HQL 的性能要比使用 SQL 好，那么我们会问，所以在你们的项目里就都是用 HQL，而不会用 SQL 了？

我们知道，如果查询语句很复杂，比如 Join 和 group by 之类很多，那么我们宁可写 SQL，更严谨的回答可以是，比如查询语句很简单，我们可以用 HQL，否则可以用 SQL。

这时我们的问题中的以偏概全的错误已经很明显了，候选人如果再听不出，那么我们只能认为候选人对这方面的知识点很不熟悉。

不管圈套怎样，总之记得一点：回答问题慎重一些，就根据你的准备，你实际用过的说。如果不知道，千万别猜，你可以说，这个是项目经理定的，我没有参与，但我知道 XX 细节。因为一旦被问出某技术你没用过，那么要比某技术掌握的很差要严重得多。

11.4.4 别自作聪明地回答面试官没问到的但你很想说的亮点

我们在面试的时候，也会遇到一些有准备的人，其实如果你真的想应聘的话，一定要事先准备，这点我们能理解，甚至赞同，你只要别露出太明显的痕迹，我们不会写上“似乎有准备，没法考查真实技能”这种话。

但你不能凭着有准备而太强势，毕竟面试是面试官主导的。

我们遇到过个别面试的人，他们说话太多，一般会主动扩展，比如问他数据库用什么，他不仅回答数据库是什么，自己做了什么，甚至顺便会把大数据处理技术都说出来。

其实过犹不及，这样我们更会重点考查你说的每个细节，因为我们怀疑你说的都是从网上看的，而不是你项目中用到的，我们甚至会直接威胁：“你和我说实话，这个技术你真在项目里用过？我后面会重点考查，一旦被认为你没用过，这个就想属于蒙混过关了。”往往这些人会主动坦白。

不过话说回来，他如果仅仅说，数据量比较大，但点到为止，不继续说后面的话，我们就会深入去问，他自然有机会表达。

同时请注意，一般在面试过程中，一旦你亮出加分点，但面试官没接话，这个加分点可能就不是项目必备的，也不是他所关注的，那么你就别再说了，或者等到你提问题的时候再说。

11.5 必问的问题：这些非技术问题你逃不掉的

这里的“非技术问题”不仅仅是项目基本情况之类的问题，而是项目开发流程方面的问题。

你技术好仅仅是一方面，但开发项目是团队工作，公司也不想找一个能力很强，但没法和别人协作的人进来。

11.5.1 如何描述你在项目里的作用？别单说你仅仅 coding

一般这种问题是：“你在项目里干了什么”，或者类似的问题。

回答的时候可以适当夸张，但要量力而行，一般项目里有如下角色。

① 需求调研，除非是内部项目，否则一般是项目经理做的，调研的时候会写一些需求和设计文档，也会用 UML 之类的工具画点图。

你可以说，在项目经理的带领下了解需求；也可以适当说写了一些文档，绘制过一些 UML 图；也可以说，如果在开发过程中遇到不清晰的业务，会及时和项目经理或相关人员确认。

② 设计，包括文档上的设计和代码上框架的设计以及数据表的设计。一般是比较资深的人做的。你可以在了解各细节的基础上，说参与过部分设计。

③ 开发，这个就不说了。

④ 测试，你自己的模块需要你自己测试，此外还有专职的 Test 人，如果你了解一些黑盒、白盒以及自动化测试工具，或者是 JUNIT 等技术，可以说出来。

⑤ 部署上线，包括打包上传发布到 Linux 之类的工作，这个很考验一个人的动手能力，你可以去专门了解这方面的技术，比如怎么写 Ant，怎么 deploy 到 Tomcat，websphere，怎么到 Linux 上运行，然后可以根据你的项目情况适当准备一下。

除了写代码，程序员在项目里或多或少会做些其他的事情，比如测试或者部署上线，但我们在面试别人的时候，不少人真就说他只参与了编码工作。对此我们一般会深入提问，比如会继续问，你是否参与了设计、测试等。如果他在我们提问后能准确说出还做了其他事情，那么不会有什么不好的影响。但万一你遇到一个不大擅长挖掘候选人能力的面试官，而且你只会说 coding，那他就真会写上“项目中除了 coding 基本没干过其他事”这样的评语。

这种评语的潜台词是，除了写代码，你没有单元测试、数据库设计、模块设计、系统发布等各种其它的经验，大家可以自己比较下其中的后果。

11.5.2 一定要找机会说出你的团队合作能力

面试过程中，我们一般会随口问一些和项目相关的情况，比如这个项目有多少人？都是做什么的？然后再进一步问，如果你在开发的时候遇到问题怎么办？不管问题是什么，其实我们想了解的内容用大白话说出来就是：“你是不是能和别人一起协作开发，会不会是刺头，遇到问题你是积极主动地解决还是消极地得过且过”。

很多人都会漫不经心随口回答，但一般来说，面试官在得不出候选人团队合作能力之前，是不会终止提问的，因为在软件开发里，不可能一人把什么事情都做了。

我们也遇到过一些人，在回答这类问题时过于简单，实在没法得出结论，就只能写“沟通和理解能力有待提升（因为你总没法理解我们提的问题，或者未表达出你的意思），没法考查团队合作精神”。如果技术可以，即使他进下一轮面试，也会因为这个原因出局。

下面列出一些关于团队合作方面需要大家展示出的能力，其实这些能力大家应该在谈吐中展示，因为这方面没什么太多的问题。

表 11.13 团队合作方面你需要展示的技能

需要讲出（或者表现出） 的要素	回答样式和面试官的考查点
沟通能力如何	如果有问题，我会及时和项目经理（或者测试人员，或者其他相关人员）确认和沟通，如果遇到我不太明白的问题，我也会及时沟通，不会按我理解的做下去

续表

需要讲出（或者表现出）的要素	回答样式和面试官的考查点
	同时，你需要在面试过程中展示出比较好的沟通能力，比如有疑问点赶紧确认，说话别吞吞吐吐，别太自大
理解和表达能力如何	这个没什么样式，表现形式是候选人能很好理解面试官提出的问题，即使个别地方不理解也能及时问懂，然后说出来的话有条理，能让面试官听明白
解决问题的方式	你需要想办法说明你是会积极主动地解决问题，而不会消极地回避问题 比如你可以说下在之前项目里解决问题的方式，你可以说，你是先通过查询网络资料尝试自己解决，如果再有问题，可以和你的同事或者领导一起协商解决
是否适合和其他人一起协作开发	候选人不会傻到自己说团队合作能力不行，一般面试官的考查点是： ①看看这个人的说话方式是不是很冲，是不是过于自大，是不是有什么沟通障碍，从中推断出这个人为人处世的大致情况 ②直接问对待问题的态度 ③有时候深入一个问题不断问，做类似的压力测试，看看候选人在被逼急了的情况下能否还心平气和
能不能适应在外派环境下工作	外派一般钱会多些，但可能压力会大些，因为毕竟甲方的压力会转嫁到外派的头上 如果你应聘的是一个外派的岗位，最好别太有个性，最好让面试官感觉你是“逆来顺受”的，这样你的成功几率就会大得多

我们在招聘的时候，如果某个项目比较着急，或者是某个人员流动比较多的项目，或者在招聘的淡季，往往会降低技术上标准，比如本来要对 Spring MVC 很熟悉的，现在只要做过就行，但不论怎么降低要求，团队协作能力不会降。也就是说，即使你能力稍微欠缺些，但很擅长和别人一起协作开发，机会就要比技能很强但沟通协作有问题的人要多得多。

11.5.3 当问项目周期规模和技术时，是在考查你值多少钱

我们往往会随口问，XX 项目你做了多久，多少人做，或者是，你项目里用到了什么新技术？

这时请大家注意，一旦你有大项目经验，或者用到某个新技术，你的身价会适当往上涨一些，但如果你不注意这方面的回答，让面试官感觉你只有小作坊、小团队的经验，则很可能会给你一个比较低的工资，因为某些技术、经验只有从大项目中才能得到。下面列出一些大项目的常用标准。

- ① 客户是比较知名的银行保险等大公司。
- ② 外派到某个著名大公司。

③ 展现出在项目里用到一些比较值钱的软件，比如 IBM 的 WebSphere、Oracle 之类的，而不是用免费的，因为大的项目能负担得起比较大的软件花费。

④ 一般在知名公司做的项目都可以理解成大项目。

⑤ 时间周期比较长，或者用到了比较热门的技术。

如果你实在没有，可以往“长项目”上靠，比如一个项目大概 10 个人规模（别多说，不是大项目，人多了别人未必信，而且 10 个人我们都嫌多，可以说在 7 个人左右），做了 8 个月以上，你是从需求分析开始做起的，一直做到上线和维护，毕竟这样你还能深入了解软件开发的周期，而且在项目里做长了你积累也多，这个时候你在谈价格的时候也有底气。

如果我们在面试的时候，别人告诉我，他不断换项目，每个项目都做不长，比如三个月到半年，而且做的项目客户背景总换，我们就没法写上“有大项目经验”，或者是“做一个比较长的完整项目”这种评语了。

11.5.4 想尽办法展示你的责任心和学习能力

责任心和学习能力看上去都是虚的，似乎没法衡量，但在面试的时候，我们会通过一些问题旁敲侧击地问出来，相信有经验的其他面试官也都会做到这点。

表 11.14 责任心方面你需要展示的能力

提问方式	需要展示的要素
如果你的项目进度比较紧/或需要加班/或需要出差/总之要你额外付出，你会怎么样	①先别问回报，比如别问加班是否有钱，因为不管你问不问，该给的总会给你 ②态度很诚恳，语气很平常，表达出愿意的意思 ③不仅要完成手头的事情，而且要经常和对应的人沟通协商，或者帮助其他开发人员、测试人员，或者帮助项目经理一起想办法解决问题 ④可以适当准备几个在上个项目里你责任心很强的例子，一般只要候选人语气措辞可以，说得头头是道，面试官听着像就会让他过关

表 11.15 学习能力方面你需要展示的能力

考查点	需要展示的要素
你在以前的项目里，有没有遇到问题？你是怎么解决的	①总会遇到问题的，不可能不遇到 ②先自己用点时间看，但别无限期看 ③和相关的人协商，协商的时候要能说出你哪块不懂，或者说出你的想法 ④同样可以准备几个例子，但别夸张，比如遇到过在一周内学好一个大数据框架的，面试官就有些半信半疑了

这些方面其实是考查一个人的潜力，如果公司对候选人要求不高，只想招个初级的，其

实也知道人和人之间差别不会太大。若这些方面你表现得很好，那么你入职的机会就会大很多。

11.6 开放性问题篇：面试官想摆脱你的准备，别慌，有技巧

在做培训的时候，我们经常被问：“开放性问题怎么准备，怎么回答”？

这里仅说技术方面的开放性问题，开放性问题的难度就好比是中考、高考的难题目，我们给出的经验是：①多去了解基础知识，②说出你所有知道的，③实在不知道时用你掌握的方法去推断。

先给大家交个底，面试的人群里，只要技术可以，有一定的项目经验，回答开放性问题时只要别错得太离谱，一般是不会因此而导致面试失败的。

常见的开放性问题有：

① 场景分析，对于一个需求，问你怎么实现。

② 你对某个技术，比如 Spring、Hibernate 等有什么理解？或者 Spring MVC 和 Struts MVC 各有什么好处？

③ 高并发大数据下，你一般怎么处理？

④ 数据库方面，你怎么做到 SQL 调优。

⑤ 一个程序已经在运行了，你是怎么查看程序的运行效率和 SQL 语句的执行效率的？

话说回来，你该展现的已经被面试官问得差不多了，这个阶段是了解你分析和解决问题的能力，只要别被认为你什么都不懂，或者干脆不知道，就不会对面试的结果有太多的影响。

11.7 当你有权提问时，别客气，这是个逆转的好机会

在面试官问完所有的技术问题后，一般会说“我这边没问题了，你有什么问题？”此时就进入到“角色转换”的阶段，你问我们答。

我们遇到不少人直接就说没问题了，然后面试就结束了。但也遇到不少人，通过这个阶段让我们改变了对他的看法，了解更多的加分项，或者干脆直接逆转面试结果。

11.7.1 通过提问，进一步展示你和职位的匹配度

你可以问，如果我成功应聘，我将进入哪个项目组？在这个项目组里需要哪些技术？

当面试官告诉你后，你就可以说，XX 技术是我以前用过的，然后谈点在以前项目里用这个技术的细节。

或者面试官说的项目你以前有过类似的经验，你也可以说一下你的经验。

如果面试官听到有类似的经验，并确认核实发现是真实的，就会再加上“以前做过类似项目”的评语，这可以起到锦上添花的作用，能帮助候选人拿到更多的工资。

有一次我们给一个保险项目招人，有个候选人刚开始的时候可能是紧张，回答问题很简单，我们已经写了“技术一般，属于可用可不用的，请后继经理斟酌”这种话，但在这个阶段，或许是因为他说到自己比较熟悉的方面，因而比较顺畅，而且知识点也展示得比较全面，我们就立即修改了评语为“技术可以，有过相似经历”这种话了。

这里给大家的建议是，你面试前还是请了解一下公司的项目，想办法套近乎，你可以适当修改你以前的项目经验，使之与公司未来要做的项目有一定的相似度，这样效果会更好。

11.7.2 通过提问，说出你未被问到的亮点

这里不建议直接说“我其实还掌握 XX 技术，但你刚才没问”，因为有些直接，此时你还是可以通过问项目情况来说。

比如你可以问，这个项目是不是用到 Spring MVC 框架？当得到肯定的回答时候，你就可以说，“我以前还在项目经理的帮助下搭建过框架，有 XX 的体会”。

或者说，你 XX 项目里某个需求点是不是涉及数据库调优？我以前有类似的经验，然后说一些。这里的建议是，你在面试前准备好一些你有把握的亮点，这里的有把握是指你非常熟悉、了解细节，而且能结合项目的实际来举例说明，想好切入点，然后就可以找机会在面试中说了。

11.7.3 可以展示一些非技术的特长

除了技术外，你还可以问，你们公司加班多吗？客户是谁？项目紧不紧之类的非技术问题，你问这些问题的目的有两个：

① 展示自己吃苦耐劳，善于沟通，善于和比较认真的客户打交道，如果你到外包公司面

试，这个能力相当重要。

② 看一下这个公司是不是血汗工厂，会不会没日没夜连轴转，会不会压力过大，如果是这种情况，你就需要酌情看了。

另外，一旦你感觉自己的面试有可能不成，那么你还可以在提问阶段说如下亡羊补牢的话。

11.8 亡羊补牢：万一你回答不好，该怎么办

即便在充分准备的前提下，也没法保证你能在面试中很好地发挥，或者说，面试主要考查你的综合能力，你准备得再充分，如果能力确实不行，那么面试也不大可能成功。请记住：面试不成是常态，对于一个有过 5 年以上工作经验的人，能保证面试三个公司成功一个，这就很不错了，更何况对于毕业生或者工作经验不多的人。

所以你一定要有良好的心态：面试成了最好，不成就当面试官给你免费提供了一次锻炼的机会，而且免费告诉了你一些面试题，你也是赚的。

但如果你一方面因为技术原因而没法通过面试，同时另一方面又什么都不做，而是幻想撞大运找到合适的公司，那么我们认为，你面试次数再多也不大可能提升你自己的能力，即便你进了公司，你的薪资待遇也不会高。

11.8.1 坦诚相对，说明你的擅长点，让面试官给次机会

我们遇到过个别候选人，他技术点知道一点，并非什么都不知道，属于可上可下的。比如项目是要 Spring MVC，这方面他只有学习经验，没有商用项目经验，但他的 Java Core 和数据库方面很不错，他就直说，Spring MVC 确实不行，但亮出了他的长处，比如举例说明他学习能力很强，或者很能吃苦，沟通能力很好，然后表达出强烈想入职的愿望，我们一般都会给出“技术可以（或技术勉强可以），能参加后继面试”的评语。

大家在面试的时候，回答问题好坏自己能估计出来。如果太差，属于一问三不知的，即使说这种话也没用。但如果你感觉回答的时候并非一无是处，那么就可以找机会说出这种话。

在表 11.16 里，列出了一些补救措施。

表 11.16 面试不好时可以采取的补救措施列表

补救因素	可以列出的证据
虽然没有 XX 项目经验，但在平时学习过，自己动手写过代码	我看过 XX 书，自己了解过这种技术，或者了解过同类技术，同时说出对这种技术的理解
学习能力很强，有强烈的学习新技术的愿望	我本来不熟悉毕业设计用到的技术，但我用了很短时间就掌握了，或者以前在公司里我属于什么都不懂的，但我肯问，用了 XX 时间就知道了 或者，最近比较热门的 XX 技术，虽然在我的项目里用不到，但我自己已经学过了，然后说说学习情况
肯吃苦，能加班，能出差，能适应大压力下的环境	列出以前公司加班，压力大的一些情况
很擅长和别人沟通，在项目里遇到不熟悉的，肯问别人	在以前公司的时候，遇到问题我不会积压，有需求上的问题找 XXX，技术上不懂会找 XX，遇到有 Bug 能找测试
事先了解到这个公司的项目背景，然后说自己知道这方面的知识	比如 XX 公司做云计算的，你即使没有项目经验，甚至没有动手写代码的经验，但你可以说，了解过这方面的知识，知道开发流程，知道入手点
说明你对 Java 里某个技术点研究特别深入，肯钻研	比如很了解 Java 的内存管理，说明你是通过看文档或者看底层代码自己研究的，那么面试官想想即使你没他需要的技能，但有自己的的一套研究方法，肯钻研，也会适当考虑
说明你的责任心、稳定性比较强，肯在一个岗位上钻研下去	这个自己想办法说明

11.8.2 展示你以前的亮点，让面试官相信你的潜力和能力

如果你的实际工作经验少于 3 年，那么面试官其实对你不会要求太苛刻，而是更关心你的学习能力、工作责任心、承受压力的情况。责任心和稳定性这些，刚才提到的补救措施你一定要有证据说明，记得用事实讲话，毕竟空口无凭。

下面列出一些我们面试过程中听到的别人说出的一些亮点，大家可以举一反三，灵活掌握。

① “我虽然对您刚才说到的 Spring MVC 技术了解不深入（事实上，他是会在项目经理搭建好框架的基础上开发，还能知道一点，如果一点也不知道，说了也没用），但我对 MVC 框架了解过，我以前做过的项目是用 JSP+Servlet+JDBC 实现的，也单独用过 Struts 的框架，所以我很快能上手”。（这样我们会适当地问他 JSP+Servlet+JDBC 里 MVC 的流程，如果他能说上来，我们就会在评语上写“了解基本的 SSH，了解 MVC 框架，知道 MVC 的开发方式”。但如果他不额外说明，或许我们就会写，“只会在项目经理搭建好的基础上了解 SSH，不了解框架细节。”这样即使他通过了技术面试，后继的项目经理看到评语也不会对他有太多的好感）。

② “最近的项目我是在做前台，没用到 SSH，但一年前用到过（这样回答有些危险了，最好是在半年前用过这个技术或者相关类似技术，不过话说回来，你即使最近没用 SSH，但在简历上说用过，只要你能回答出基本问题，面试官也没办法核实），但我对 SSH 框架了解很深，我知道 Spring 里 MVC 的底层实现，感觉 Struts 的 MVC 有一定的缺陷，也在商业项目里搭建过 SSH，所以我能很快上手”。（这样我们会细问他提到的 SSH 的底层细节，如果他确实对底层细节了解得不错，那么我们会写上“最近一年没用过 SSH，但对 SSH 底层有一定了解，在商业项目里搭建过 SSH”。否则，我们仅仅会写“最近没用过 SSH，SSH 的项目经验仅限于一年前”，大家可以对比一下两个评语之间的差别）。

③（为一个保险项目招人）“我在 Spring MVC 方面的经验不多，所以有些问题没回答好，以前大多是用 JSP+Servlet+JDBC 这套模式开发的（这是大实话，不过如果他面试前好好准备的话，那么就不应该说这种话），但我以前做过保险相关的项目，客户是 XX，实现了保险项目里的 XX 流程，而且我知道一些背景的业务”。（这样我们会把决定权交给二面的经理，否则，我们将直接写“不了解 Spring MVC，没法通过面试”）。

④ “我对框架技术了解一般（确实一般，根据问题的答复我们能感觉出他自己没搭建过框架，只能在人家搭建好的框架上被动开发），但我知道怎么让我的代码效率更高，我通过看文档和底层代码，知道 Java 内存管理的细节，知道多线程的实现细节，知道 SQL 调优的方法，了解过一些设计模式，思路是相通的，所以我能很快上手，而且能很快了解 SSH 的底层”。（这样我们会逐一确认他说的，是否真的对这些加分项有了解，如果是，由于这些亮点比写代码本身更重要，我们甚至会掩饰他 SSH 一般这个事实，会在评语上写“知道 SQL 调优，Java 代码调优的一些方法，学习能力和学习意识比较强，个人的综合能力可以”）。

⑤ “我对 Java 技术了解一般，（确实一般，只会用语法，不能融会贯通），这是因为我在上个项目里压力很大，需要直接和客户交流，直接了解需求，自己开发，自己测试，最后打个 jar 包给客户，所以我感觉我的综合能力很强”。（这样我们会关于这方面问一些细节问题，比如怎么打 jar 包，测试的时候是怎么做的，如果确实能说上来，我们会在评语上写“Java 能力一般，但知道整个开发的流程，能独立完成某个模块的任务”。否则我们只会写“Java 能力很一般，不了解一些深入的知识点”）。

⑥ “虽然我没有商业项目的经验（是个应届毕业生，简历上的项目被我们问出是毕业设计或者是课程设计项目，但他如果直接把这些技术写成在读书时在外面公司里做的，我们是没法核实的），但我自学能力比较强，我学习的时候走了不少弯路，这也让我现在很了解 JDBC 的底层实现，我知道最近热门的一些技术，所以你们公司的一些技术我能很快上手”。（这样我们会在评语上写，“没有商业项目经验，但学习能力很强，请后继面试官斟酌”，这总比“没商业项目经验，不建议通过面试”的评语要好）。

11.8.3 记下所有的面试题，迎接下次面试

当你感觉你成功应聘这个岗位的希望有些渺茫时，你需要做如下的事情。

① 记录下所有的技术面试题，回家查资料，为下次同样问题做准备。

② 举一反三，回家以后要赶紧学习了，最好通过动手实践，通过运行代码来了解相关的知识点。

③ 找出没成功的原因，比如这个岗位需要有项目经验的，你所描述的项目经验最终被认为是非商业项目，那你就需要更新项目描述，下次面试的时候，也要更改说辞，想办法证明你的项目确实是商业项目。

如果是因为你没有回答好具体某个技术，那么一定要去找一个真实的项目，看看这些技术在项目里是如何实现的。

即使一些工作经验 5 年以上的资深者，在刚开始换工作的几家面试公司里，也未必能回答好，因为他即使做了很多准备，也不知道面试会问些什么，所以面试前你要做好“不成功”的准备，成了最好，一旦没成，积累经验，下次或许你就成了。

11.9 基础差，不知道怎么应对面试时的对策

面试技巧确实能帮到你，但如果你通过几次面试，发现你当前的能力确实有待提升，那么你首先要做的不是提升面试技巧，而是要多学多练。

11.9.1 有计划的学习和实践

这里给大家介绍一些具体的步骤。

第一，确定你的发展方向。比如你想做 Java 后端的 Web 开发，那么你可以找一些不同公司的相关职务描述，列出这些职位的具体需求。

第二，综合整理这些需求，并列出你的学习计划。比如打算在最近的三个月里学好 Java Web 开发技术，然后再具体细化每周、每天要做的事情，对此，你可以给自己列份计划表。

第三，你可以到网上找相关的成系统的教学视频，也可以去买相关的书籍。切记，一定要边学边练，提升自己的实际动手能力。

当你看过 3 套以上教学视频，或通过运行代码通读 3 本以上书籍后，你的能力会得到显

著的提升。根据我们的培训经验，一些比较上进的学生（大多数是一般学校的本科，偶尔是专科），他们只要保证每周有 5 个小时的学习时间，那么一般只需要用 3 个月（遇到一些特别好学的学生，估计也就 2 个月），就能从零基础提升到“看上去有 1 年半左右经验”的水平。

11.9.2 多挖掘你之前的项目经验和技術点

其实不少候选人不是真的技术差，也不是真的缺乏项目经验，只是没有在简历中正确全面的描述，大家可以通过如下列出的确认点来对比一下，看一下你的简历是否真实地反映了你的水平。

确认点 1，在你的简历中，是否完整地列出了你的商业项目经验。

大家一般不会忽视毕业后的工作经验，但我们和一些候选人交流时，发现虽然他们也有毕业前的相关实习经验，但在他们的简历上，只列出毕业后的工作经验。要知道，实习经验年限也能勉强地算入总的工作年限。

确认点 2，如果你的项目介于商业项目和学习项目之间，那么你可以当作商业项目来写。

比如有些学生的毕业设计项目的需求是来自真实的商业项目，在毕业设计中要实现其中的若干个模块，如果最后这些设计直接或通过少量修改用到了商业项目里，那么这个项目就可以认定是商业项目。

又如某些培训学校里会让学生做实训项目，就是从真实商业项目里抽出一些模块让学生做，如果最终的成果也能落实到商业项目里，那么这就不是学习项目，而是商业项目。

确认点 3，你的项目描述是基于需求实现，还是列上了你用到的技术。

我们看到不少简历上列的是做了什么功能，而不是偏重用了什么技术，在这点上大家千万别客气，你技术都用了，那为什么不写？之前也讲过，列举技术时要尽量往你申请的职位需求上靠。

确认点 4，你简历上的项目经验是否真的和职位需求一致。

比如某人有 5 年经验（不算少），但前 3 年他本职是做网络管理，后 2 年才做 Web 开发，按理说相关经验才 2 年。

但他在做网络管理期间也用 Java 编写过一些管理程序，而且在他公司项目忙的时候，他也会被借调到开发部门做 Web 后端开发（这种情况确实存在）。

如果他就按前 3 年网管后 2 年开发来写简历，那么很难应聘一些高级岗位（高级岗位一

般需要 3 年以上经验)，但他适当改进了简历，在前 3 年的描述里，着重描述开发经历（当然网络管理经验也写了，但不是以此为重点），后来他也成功应聘上了高级开发的岗位。

也就是说，如果你比较用心，在同一时间段里在这个公司做了多种类型的工作，那么在描述简历时，这段时间的描述就要往职位需求上靠，这样能增加你的相关工作经验的年限。

11.9.3 及时提升你项目里用过的知识

经常见到这样的情况，在有些候选人的项目描述里，我们看到了不少能和岗位相匹配的技术点，但经过细问，他说有些技术虽然在项目里用到，但是别人做的，他不了解很细节的东西，或者他虽然用过某种技术（比如 Spring 生命周期），但给人的感觉只是修改别人的代码，自己不知道其中的细节知识点，甚至还不知道这部分代码属于“Spring 生命周期”这个知识点。

如果职位需求里提到的技术点在你的项目里都已经用到了，那么你就没有理由回答不好相关的问题，大家可以通过如下确认点来完善你针对应聘职位需求点的能力。

确认点 1，应聘职位里提到的技能是否在你的项目里用过（而不是你在项目里做过）。

如果有，你可以通过问别人，或者看具体代码，或者结合看相关资料来完善这部分的技能。

确认点 2，对于应聘职位里提到的技能，虽然你在项目里用到过，但你实际掌握得如何？

我们见过一些得过且过的人，由于他只是修改别人的代码来实现自己的目的，一个项目做好了，他甚至不知道项目用的是 Spring MVC，导致他不知道自己在这方面的经验，从而不敢去应聘这方面的岗位。

这方面给大家的建议是，首先你要从框架和整体的角度去了解你的项目，其次你还要能通过具体的代码，说出一个技术的工作流程、大致的开发步骤和关键性的代码。

11.10 背景调查的一般流程

在前文中我们提到，如果某候选人相关的工作经验不足，或者经常换工作，那么会对他后继找工作带来一定的困难，有时甚至没法通过初步的筛选，所以有些人就会通过修改简历描述来避免这些问题。

一般来说，当你入职后，公司会进行背景调查来确认你面试时提供的简历是否真的和你的实际经验相一致。

11.10.1 技术面试阶段，着重甄别是否是商业项目

其实在技术面试时，有经验的面试官会剔出非商业项目，因为非商业性项目的含金量不高。一般甄别是否是商业项目的方式如表 11.17 所示。

表 11.17 甄别商业项目的检查列表

检查方向	检查点
时间范围	从时间范围来看，如果某个阶段候选人还在学习，或者在培训学校里进修，那么这些项目是学习项目，除非能提出合理的说明
项目种类	比如某系统是图书管理系统，或者人事管理系统等，毕竟正规的机构一般会用比较成熟的整套软件，而不会在投入使用后还让别人开发相关的软件
时间范围和开发人数	一般中型项目是半年到一年，大概 5 到 10 个人左右，毕竟软件公司要挣钱，对于每个商业项目不可能无限延迟，也不可能投入过多的人 我们经常在简历上看到一些不合理的时间范围和人数，可能是简历的当事人没有很好地去了解行情

11.10.2 关键因素一旦不对，立即出局

全面的背景调查一般是在确认录用和入职这段时间内进行，有些公司也会在入职后进行，表 11.18 里列出了背景调查的常规项目。

表 11.18 背景调查常规项列表

确认点	确认方式	确认不符后的后果
基本信息，比如年龄、学历、学校、专业、毕业时间、外语水平等	提供比如身份证、学历学位证等证明材料	这类关键信息不符，而且没有合理的解释，可能会导致立即出局
工作过的公司情况，着重确认一下你上家公司，还有确认一下你在具体每个公司的工作年限是否和你简历上的一致	①在劳动手册（或其他相关的材料）上一般有之前公司的工作年限记录 ②会让你提供之前公司的联系人电话，会打电话确认	如果存在不符的情况，比如某人在 2 年里换了 4 份工作，但简历上他通过拉长其中 2 份工作的年限，只写了 2 个公司，那么公司就要你解释了，如果没有合理的解释，很可能立即出局
看一下你之前的工作经历是否和你简历描述的一致	会让你提供之前公司的联系人电话，会打电话确认	如果你试用期的表现并不好，那么说明你能力真的有问题了，可能存在辞退风险
你上份工作的工资情况	①可能会让你提供联系人的电话，通过电话确认 ②可能让你提上家公司的人事开的证明 ③最严格的是让你提供银行流水清单	我们不是人事部门，所以不确定这个后果，但如果没有合理的解释，至少这个属于不诚信

11.11 面试评分的一般依据

虽然面试的主观性很强，但也有一定的确认点，这里我们整理了技术面试和综合面试阶段比较通用的评分表供大家参考。

11.11.1 技术面试的考查要点

不同公司的考查项可能不同，这里以某个国际大公司的考查项为例。

第一部分是考查基本技能。在技能方面，一般会通过如表 11.19 所示的表格给候选人打分，打分可以分 5 个档次：1 分属于什么都不知道，2 分属于可上可下，3 分属于能力一般，4 分属于高于平均水平，5 分属于特别优秀。

表 11.19 面试评分考查点列表

考查点	参考因素	打分
基本技能	①最近是否有该技能（或者是类似技能）的项目经验 ②是否在大的项目里使用过这个技能 ③是简单使用，还是会一些比较深入核心的知识点	
技术（或框架）的基本功	①是否了解底层实现细节 ②是否知道调优方式 ③是否会用高效率方式编写代码 ④代码安全性、高并发的考虑 ⑤是否能全面了解各种框架和技术，是否对框架和技术有自己的想法，能否综合使用各种技术和框架	
数据库方面	①有没有建表的经验 ②是否会 SQL 调优 ③是否了解索引等技能 ④是否有大数据方面的经验	
项目开发和实施的能力	①团队能力如何 ②是否有管理项目的经验 ③是否能承受大压力下的工作 ④对项目的整个开发周期是否了解 ⑤掌握诸如敏捷开发之类的项目开发方式 ⑥是否有在外派公司的工作经验，是否有大公司的工作经验	

这里的要求是，项目开发和实施的能力必须要 3 分以上，其他能力，初级人员则要求两个 2 分一个 3 分，高级人员则需要都在 3 分上，而且需要一个 4 分以上。

第二部分那是考查英语能力，在英语读写能力方面，一般也是 5 个档次：过 4 级外加有基本读写能力可以给 3 分；过 4 级外加沟通没问题（能和外国人沟通）是 4 分；没过 4 级但有基本的读写能力是 2 分；没 4 级但沟通没问题 3 分。如果有海外经验，而且沟通没问题，不论是否过 4 级，都给 4 分。

这个要看项目组的要求，如果有和外国人交流需求的，要求是 3 分，国内项目一般不要求英语能力。

第三部分是考查综合能力，比如学习能力、沟通表达能力、是否可以加班、是否可以出差、责任心、是否聪明、稳定性、入职时间等方面，这方面没有打分要求，面试官可以写上具体的评价，包括优点和缺点。

虽然说后继的项目经理会着重考查综合能力，但在技术面试阶段也会先大致了解一下，如果遇到实在不合适情况，比如沟通明显有问题，就可以直接过滤掉，以节省面试时间。

第四部分专门写加分项。

第五部分专门写不足点。

第六部分是下结论，包括评语（也就是理由）和结果两种，最终的结果包括如下 3 种。

① 不适合当前岗位，但可以转给其他岗位面试。

② 不适合本公司的所有岗位。

③ 适合当前岗位，建议录用。

作为技术面试官，我们一般只针对技术考查，会根据如下不同的情况给出不同的结论。

① 技术过关而且没有大的不好的因素的，让过。

② 技术可过可不过，同时也没有大的不好的因素的，让过，但需要在评语中写明具体的情况，让二面经理考虑。

③ 技术不行，但有明显的优势，比如做过类似的项目，有海外工作经验，或者干脆项目组着急要人，也需要推荐，但需要在评语中写明，让二面经理综合考虑。

④ 技术可过可不过，同时有明显的缺点（比如沟通能力不行或表达能力不行），我们这边有权利直接淘汰，但需要上报具体的理由。

⑤ 技术很差，没有其他的加分点，可以在上报理由的前提下直接淘汰。

⑥ 技术很好，但综合能力不怎么好，感觉招进来会有障碍的，需要写上对应的评语让二

面经理考查决定。

11.11.2 综合能力面试的考查方式

一般是资深程序员或架构师来进行技术面试，如果候选人通过了技术面试，那么项目经理或部门经理就要进行综合面试。

此时，技术能力不再是综合面试的考查项，项目经理一般会从沟通能力和稳定性方面来看候选人是否适合这个岗位。表 11.20 中列了一些考查点，其实综合面试的主观性更强一些，导致综合面试失败的原因大多是感觉候选人不适合当前项目，或者稳定性不强。

表 11.20 综合面试的考查点列表

考查点	参考因素	评语
团队协作能力	①从说话方式语气等方面看看这个人给人的感觉是否好相处，有没有明显的性格方面的问题 ②询问你之前的项目经验，比如会问如果出现问题了，你是怎么和别人协调解决的 ③确认你不会成为项目组里的刺头	
沟通和表达能力	①谈吐和思路是否清晰，表达方面是否存在明显的问题 ②遇到问题，能不能主动沟通别人来解决，而不是被动地等别人来问	
职业规划	问下你之后 3 到 5 年想干什么，如果你想干的和你的实际岗位有落差，说明你稳定性不强，这可能会导致你被淘汰	
稳定性	通过询问之前公司的离职原因，考查你稳不稳定，如果感觉到你会经常因为待遇或者压力大而辞职，那么可能会导致你被考查	
是否能承受大压力的环境	①问你能不能加班 ②了解你之前公司的加班情况	