编程实例大讲堂

CD-ROM

Java Web整合开发实例精通

Struts+Hibernate+Spring

闫术卓 吴峻申 等编著

本书特色

⊙ 涵盖基础知识、核心技术、典型实例等内容

⊙ 按照“技术要点→实现代码→源程序解读”的模式讲解，容易上手

⊙ 提供80余典型实例进行讲解

超值光盘内容

⊙ 本书源代码

⊙ 1200余页编程技术文档（免费赠送）

⊙ 45个编程专题视频讲座（免费赠送）

机械工业出版社
China Machine Press

Java Web整合开发实例精通

——Struts+Hibernate+Spring

闫术卓　吴峻申　著

ISBN：978-7-111-26328-9

本书纸版由机械工业出版社于2009年出版，电子版由华章分社（北京华章图文信息有限公司）全球范围内制作与发行。

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.bbbvip.com

新浪微博 @研发书局

腾讯微博 @yanfabook

目　录

丛书序

不积跬步，无以至千里。

——荀子

初学编程的人很苦恼的一件事是不知道如何上手。其实有两种思路都可以很好地上手。一种是按部就班，像大学里的C程序设计课程一样，从基本语法，到各种具体应用程序逐渐深入；还有一种方法是，把基本语法和一些函数等的用法用比较典型的实例贯穿起来，通过学习这些实例来掌握编程知识，这也是一种很好的方法。因为学习编程需要大量的实践才能学好，而这种方法正好符合这种学习特点，所以也有比较好的效果。按照这个思路我们策划了这套书。

丛书特色

作为一套以实例贯穿始终的图书，本丛书在编写上着重体现以下特色。

1.以实例引导学习，可快速入门

本丛书以全新的实例模式编写，每本书都是以实例贯穿始终，读者可以在实例引导下一步一步地学习编程，增强了编程的亲身体验，可以快速入门，达到良好的学习效果。

2.编写模式科学，讲解细致

本丛书中贯穿的实例大都是按照“技术要点→实现代码→源程序解读”的模式编写，非常科学，讲解也很细致，容易掌握。

3.实例数量丰富，实践性强

本丛书每本书都是以数以百计的实例指导读者学习，这些例子实用强，可为读者以后程序开发奠定坚实的基础。

4.代码规范，注释丰富

为了增强代码的易读性，丛书编写时对代码进行了丰富的注释，非常易于读者阅读和理解，增强学习效果。

5.光盘内容实用、超值

配书光盘提供了书中所涉及的源代码，以方便读者使用。除此之外，还特别免费提供了大量的编程入门视频和技术文档，以方便相关人员学习和教学使用。

6.提供技术支持

本丛书提供了论坛：http：//www.rzchina.net，读者可以在上面提问交流。另外，论坛上还有一些小的教程、视频动画和各种技术文章，可帮助读者提高开发水平。

丛书包含的书目

《Java实例精通》

《PHP实例精通》

《Java Web整合开发实例精通——Struts+Hibernate+Spring》

《JavaScript实例精通》

《Visual C++实例精通》

《Visual Basic实例精通》

《C#3. 0实例精通》

《ASP. NET 3.5实例精通》

本丛书读者定位

初学编程的人员；

已经入门，需要通过实例提高编程水平的人员；

大中专院校的学生；

社会培训学员；

相关程序员。

阅读本丛书的几点建议

没有基础的读者建议按顺序阅读，不要跳跃，不要跳步。

有基础的读者可以跳过一些特别基础的章节学习。

如果感觉学习本书有困难，建议先阅读机械工业出版社的“编程红宝书”丛书中的对应入门图书。

多动手，亲自完成书中的实例，加深理解。

遇到问题，除了本书的技术支持论坛，还可利用网络资源解决。例如，利用Google和Baidu搜索相关资料，或者在相关论坛上发帖提问，会有热心人给你答复。

要重点阅读源代码及其注释，可以有效提高代码理解能力。

正所谓“宝剑锋从磨砺出，梅花香自苦寒来”。编写这样的一套书也实属不易，是一个需要克服很多困难、花费大量心血才能完成的“浩大工程”。同

样，在学习编程的道路上也不会一帆风顺，肯定有许多磨难等着你。我们伟大的思想家荀子早都说过，“不积跬步，无以至千里”。做任何事都得脚踏实地，才能走得远，希望以此与各位读者共勉。看到你们能以此套书提升编程水平，便是我们最开心的事了！

丛书策划编辑

前言

本书讲述的Struts、Hibernate、Spring无论在已有项目的选用比例，还是在开发人员的认知度上，都是最有影响力和号召力的，Struts+Hibernate+Spring已经成为轻量级开发J2EE的标准配置，被称为SHS经典组合，这也是目前Java程序员必须掌握的技能。

为了能让读者以最直接的途径了解到最新版本的SHS组合，我们特编写此书。本书是一本由小实例组合成的实践书，每个案例读者都可以亲自实践，也可以参考配套光盘中的源代码。

本书特点

相比同类图书，本书具有以下明显特色。

1.版本最新，与时俱进

本书所提供的Struts+Hibernate+Spring都是使用的最新版本，因为旧版本与新版本的差距比较大，希望读者在使用时安装最新版本。本书采用的是Struts 2+Hibernate 3+Spring 2的最新版本组合。

2.层次递进，讲解清晰

本书提供了最常用的三个框架，每个框架开始前，都将这个框架进行了总体的概览，然后才通过小实例，逐步去学习框架的具体知识。讲解方式通过“代码+注释+效果图+代码说明”的方式，让读者每看完一个案例，都能明白其中的道理。

3.实例丰富，强调实践

框架本来就是从实践中不断提升归纳出来的经典程序。本书列举了大量实例进行讲解，通过这些实例，读者可更加深入地理解相关概念和语法，从而达

到灵活使用Struts、Spring、Hibernate编写程序的目的。另外，本书重点强调实践性，本书中的很多例子都来源于作者的实际开发，通过对这些例子的学习，可以增强读者的动手实践能力。

4.代码规范，注释丰富

本书所涉及程序源代码层次清楚、语句简洁、注释丰富，体现了代码优美的原则，从一开始便给读者树立良好的榜样，有利于读者养成良好的编写代码习惯。

本书内容

本书虽然没有具体的分篇，但从书名就知道应该是三大部分：Struts、Hibernate和Spring。在目录中都介绍了每节的具体意义。全书共26章，下面介绍每章的主要内容。

第1章：通过Struts的历史，了解Struts的发展，并学习最新的Struts所具备的特色。

第2章：Web开发都需要哪些技术，什么是B/S系统，什么是C/S系统，是本章要解决的问题。

第3章：Struts 2核心技术是什么，从Struts 1.X到Struts 2都发生了哪些变化，Struts 2为Web开发带来了哪些好处。

第4章：详细介绍Struts 2的拦截器，并介绍拦截器的实现原理。通过本章学习拦截器的使用和作用。

第5章：作为Web开发的利器，Struts 2提供了标签库，其中包括控制标签、数据标签、表单标签、非表单标签等。

第6章：Struts 2非JSP视图技术，注意它和JSP视图技术的区别，主要介绍velocity、freemarker、

JasperReports三个视图。

第7章：介绍了一些Struts 2类型转换技术，包括List集合转换、Set集合转换等，本章还对转换过程中的异常情况做了讲解。

第8章：介绍Struts 2输入校验技术，其中包括Struts 2提供的一些校验器，还有如何通过配置文件来校验数据等常用安全技术。

第9章：Struts 2的国际化应用技术，这在多语种版本中应用很广泛，就是允许用户选择中文或英文界面，当然也可以设计其他语种界面。

第10章：Struts 2页面布局的实现。其中会涉及一些高级的布局标签，如＜decorator：usePage＞、＜decorator：useHtmlPage＞和＜decorator：head＞标签。

第11章：Hibernate技术的简介，说明Hibernate的发展历史、结构，以及为什么要应用Hibernate。

第12章：讲解如何安装Hibernate框架，如何在MyEclispse下应用Hibernate，还介绍了一些常用的SQL方言。

第13章：Hibernate核心API的介绍，包括Session、CRUD、事务处理等，本章还介绍了Hibernate缓存的相关知识。

第14章：详细讲解Hibernate映射的技术，其中包括Set集合映射、List集合映射、Map集合映射、Bag集合映射、Component映射等。

第15章：Hibernate关系映射的介绍，因为Hibernate涉及一些数据库的操作，所以本章知识虽然抽象，但是非常重要。

第16章：学习Criteria条件查询。查询功能非常强大，本章分为几个部分介绍，如简单查询、复杂查询等。

第17章：详细介绍HQL查询。学习使用HSQL实现一些增加、删除的操作，还有一些稍微复杂的操作，如分组统计等。

第18章：Spring入门详解。通过Spring的发展历史，了解为什么需要引入Spring框架，并介绍了这个框架的组成。

第19章：为什么要使用控制反转（IOC）？本章解答这个问题，由于它不是什么具体的方法，所以理解起来有点困难。请读者自己动手多做案例。

第20章：解读IOC容器的反射机制和装载机制。IOC容器利用Java反射机制创建类、调用方法等，它是

如何把它们转载起来呢？本章主要讲解的就是这些核心技术。

第21章：分析DI注入方式。DI类型分别有接口注入、构造注入和设置注入，本章通过具体的事例介绍这些类型，同时还要注意各种类型是如何实现注入的。

第22章：学习如何合理地编写配置文件。当需要注入的Bean不多时，对XML文件的编写还可以接受，可是当注入的Bean太多时，就会出现问题。

第23章：本章介绍为什么要使用AOP。Spring出现后，完全可以利用Spring的AOP代替EJB来声明式事务。

第24章：掌握在Spring环境中实现AOP技术。当Spring的IOC为目标对象创建出动态代理类时，开发人

员能获取动态代理类的参数吗？在编写目标对象时是否必须继承接口？这些问题都在本章中给予解答。

第25章：Spring与Hibernate结合使用。主要讲解两种技术：使用编程方式实现事务和实现声明式事务。

第26章：Spring与Struts结合使用。介绍用依赖查找方式实现Spring与Struts的结合，也可以用Action注入方式实现Spring与Struts结合。

本书适合的读者

本书适合以下人员阅读：

Java开发Web应用和J2EE方面的初学者；

对于有一定基础但希望提高自己的系统设计水平的读者；

使用过Struts+Hibernate+Spring，想了解最新版本的新特性的读者；

想以最新版本替代旧版本框架的读者；

有一定基础但希望提高自己的系统设计水平的读者；

其他编程爱好者；

大专院校的学生。

本书作者

本书主要由闫术卓、吴峻申主持编写，其他参与编著和资料整理的人员有冯华君、刘博、刘燕、叶青、张军、张立娟、张艺、彭涛、徐磊、戎伟、朱毅、李佳、李玉涵、杨利润、杨春娇、武鹏、潘中

强、王丹、王宁、王西莉、石淑珍、程彩红、邵毅、郑丹丹、郑海平、顾旭光。

编者

第1章　Struts概述

Struts是目前世界上所有使用Java语言进行J2EE项目开发的人员，经常使用的基于MVC模式的Web项目开发框架之一。它也是目前最早的Web项目开发框架。由于它的易学易用，对入门者来说学习所花时间少，也容易上手，因此使用Struts的开发人群是目前所有Web项目开发框架使用人群中最大的。可是近几年，随着新的视图技术（如FreeMarker、Velocity技术），还有设计模式的大行其道，开发人员越来越觉得Struts在这些方面有先天的不足，并不能很优雅和优秀地完成Web项目开发工作。

原因有很多种，一方面是Struts出现的时间比较早，现在流行的技术都是在Struts后出现的，因此必然导致Struts对新技术的支持不够。另一方面很多新兴的Web项目开发框架都很好地体现了现有开发理念的

使用，对Struts的影响和威胁都很大，所以近几年参加工作的很多IT从业人员都不喜欢使用Struts，而是采用Tapestry、JSF等框架进行他们的开发工作。

值得庆幸的是，Struts的开发人员也意识到了Struts的这些缺点，因此在2006年Struts和另外一个Web项目开发框架WebWork进行了合并，形成了新的Web项目开发框架Struts 2。这个所谓的Struts 2其实就是WebWork的一个新版本。

一方面WebWork在IOC、基于接口编程、新的视图技术支持等方面，具有先天优势。

另一方面Struts使用人群的庞大以及学习曲线的平缓，还有它的技术延续性，使得这两个项目开发框架各取所长，互相补充，形成了一个更有竞争力，更具有健壮性的新框架。

所以Struts并没有过时，它已经进化成一个崭新的Web项目开发框架。本书这部分就针对Struts 2的具体技术细节进行了详细附例的说明。而本章则把Struts和Struts 2之间的“恩恩怨怨”，以及同类的Web项目开发框架产品和Struts 2的关系具体介绍。希望读者在学习Struts 2技术之前，能对Struts 2有清晰正确的认识和了解。

1.1 Struts历史发展过程

Struts这个名字来源于在建筑和滑翔机中用来支持的金属架。Struts的开发者大概是希望用该框架来支持JSP、Servlet、Java这些技术在Web项目中的应用。

2001年春天，Struts的第一个版本在apache网站上发布时，它只提供了一种分离视图和业务应用逻辑

的Web应用方案。因为在Struts之前，开发人员都是在JSP里写入处理业务逻辑的Java代码，尤其是涉及数据库和页面Form表单数据交互时，开发人员在每个页面都要写入像连接数据库这样的Java代码，导致了大量的代码冗余。而且每个页面显示速度和性能都不是很好，这是因为页面中存储数据的Java对象都需要从内存中读取，势必影响性能。

所以当像Struts这种Web应用方案一出现，每个开发人员都把它视为把自己从繁重的开发工作中解放出来的利器。所以在2001～2003这几年，大量的为企业做Web应用系统的IT公司，在项目架构中都采取Struts作为开发中必须使用的框架。

1.1.1 Model设计模式的演进

从市场推广的角度来看，Struts也是一个创建知名品牌的案例。由于它的知名度，很多开发人员开始熟悉Struts的应用，一批批成功应用Struts的Web项目，如雨后春笋般显现出来，这算是入门者依据前人的优秀经验，应用Struts去实现更多的Web项目。随着它作为项目开发框架的公司和人员越来越多，作为公司的领导层也让实际开发人员把注意力更加着重放在行业知识、业务领域的研究实现工作上。

其实在Struts出现之前，使用J2EE开发的Web项目都是使用Model1的设计模式，Model1模式有三种常用的开发方式：

（1）之前所述的将Java代码写在JSP中，就是第一种常见的方式。

（2）还有一种就是将部分业务逻辑实现代码封装成为JavaBean，在JSP中调用这些JavaBean。此种方式

解决之前所述冗余代码的问题，而且从内存中只读取一次封装了数据的Java对象，没必要频繁地读取Java对象。它是Model1模式的典型结构，如图1.1所示。

（3）第三种就是将Java中已经是标准的内置对象在JSP中调用。比如J2EE的JDBC（JSP中直接使用JDBC的标准类和方法）。实际上和第二种方式是类似的，只不过第二种方式中调用的是开发者自己开发的，而第三种方式调用的是Java的标准类和方法。

Model1模式对于中小项目的开发还是很有优势的，有些企业为了避免开发带来的风险，同时这些企业的相关环境决定使用其他模式也没有那么迫切，因此还在使用这种模式开发自己的企业应用。但是它有以下明显的缺点：

（1）首先，如果企业级应用需要改动原有需求或者新增需求，那势必需要改动很多代码，开发工作量

会很大。

（2）其次，就是业务逻辑代码和使用视图来表示页面的代码是在JSP中混合的，如果某个JSP页面需要重用，是根本无法实现的。因为业务逻辑在每个页面是不同的，无法用一个JSP中的业务逻辑套用到另外一个JSP页面中。

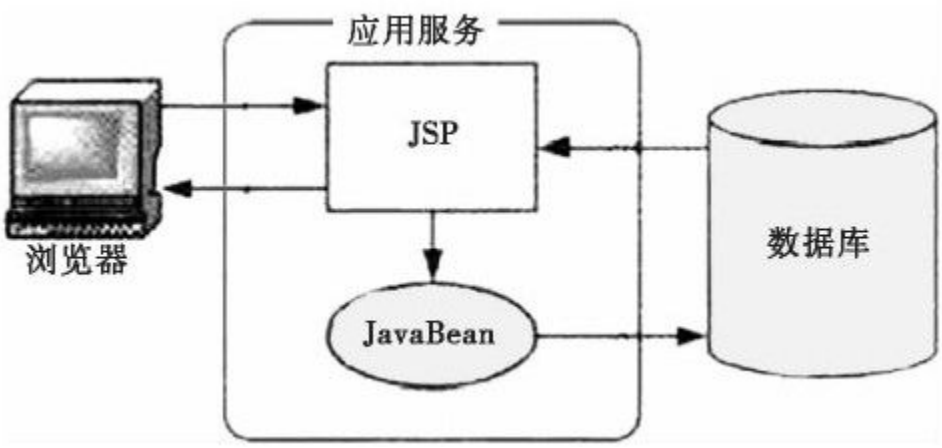

图　1.1　Model1模式结构

基于这些原因，在开发者中有人提出了Model2模式，这种模式保留了Model1的优点，又针对它的缺点进行了修正和改良。Model2模式基础概念是组件化，

它的设计者在最初的设想是开发Web项目时，如果需要实现某个业务逻辑将封装好的组件进行调用，而开发者不需要再关心该组件内部是如何实现的。

这就好比购买了一台电脑，直接开机使用就可以，没必要关心电脑内部是如何组成的，内部又是怎样实现让用户使用电脑功能这些细节化的东西。该模式采用业务逻辑、视图分离的方法，让JSP只负责展现表示功能，相应的业务逻辑由Java来实现，当开发新的需求或者修改原有需求时没必要在JSP中更改，而且JSP也可以被复用。具体结构如图1.2所示。

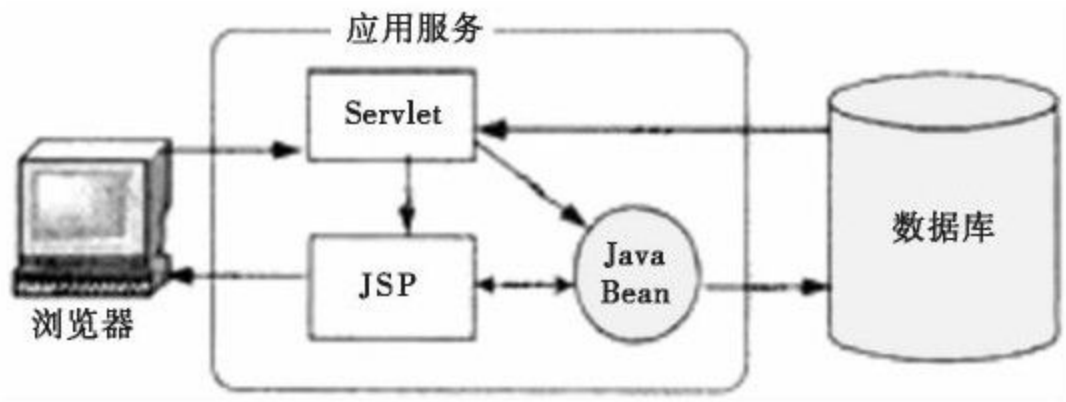

图　1.2　Model2模式结构

1.1.2 MVC模式

有时开发者也把Model2模式称为“MVC”模式。“MVC”是Model、View、Controller这三个英文单词的缩写。

“Model”代表的业务逻辑这块由Java实现的组件。

“View”则代表了表示界面，当时主要是使用JSP技术来实现，而现在还有笔者之前提到的FreeMarker、Velocity这些实现表示界面的视图技术。

“Controller”代表的是处理流程控制，主要功能是实现业务逻辑如何和表示界面相关联的技术。

为了让读者不引起歧义，以后章节中都将Model2模式称为MVC模式。由于MVC模式的提出，彻底解决了Model1模式的缺点，很多开发人员开始使用这种模式来解决他们实际工作中碰到的各种各样Web项目开发问题。也正是在这样一个大背景下，诞生了Struts这个运用MVC模式的Web项目开发框架。

从2003年开始，同类型的Web项目开发框架也开始出现在广大开发者的视野中，由于Struts对新技术的支持不够，一些Struts技术细节是否有必要实现的争论，也让开发者开始质疑Struts的优越性。

Struts的设计者也意识到Struts的一些局限性和缺点，必须进行重新设计和改良。因此在2006年和WebWork的设计者一起将这两个框架合并在一起，形成了今天的Struts 2框架，这个框架是综合了原有两个旧框架的特点，而又去除了很多它们的缺点，特别是

Struts的一些缺点，比如支持的视图技术太单一（只支持JSP），还有和Servlet关系太紧密，不适合现在的松耦合开发理念等。

Struts 2框架实际上也可以称为WebWork的一个最新版本，它的MVC模式实现的方法更多是来自于WebWork而不是Struts，Struts框架其实还发展了另外一个分支框架Shale，这个框架的设计思想更多是来自于JSF，只不过通过Struts的基础配置和代码来实现，由于JSF和Struts、WebWork的开发理念有很大的区别，JSF可以称为Web化的Swing。

让桌面应用程序开发结构在Web浏览器中实现出来，这点对现在的开发人员也很有吸引力，但是笔者认为还是有些不能和Struts、WebWork相比的弱点（在后面的小节中将详细叙述）。所以对于Struts的演变

笔者更认为Struts 2才是最正统的继承，而Shale则只能算作一个比较好的补充而已。

在下一节，将详细概述Struts 2的特点及需要改进的部分。

1.2 Struts 2简介

Struts 2应该说是第二代实现MVC模式的Web项目开发框架。它是以拦截器先拦截HTTP请求，在进入MVC模式中的“C”（就是控制器）部分前，对HTTP请求中包含的数据做校验、字符编码转化等操作，由于这些操作和业务逻辑关系不是很大，所以提前做这些事情，也让控制器可以着重处理HTTP请求和业务逻辑之间转发、处理等控制功能的实现。

这就体现了“松耦合”的开发理念，让Web项目各部分都发挥自己负责的功能，而又不互相牵涉和纠缠。开发人员进行修改、新增等功能时只需关心被开发的部分，而不需要去关心其他部分。

Struts 2按照技术细节划分，主要分为以下几块。

标签库：在视图中运用这些标签来实现网页上各种格式的显示。

拦截器：HTTP请求在进入控制器部分执行前先执行拦截器中的功能。

FilterDispatcher和Action：接收HTTP请求，根据Action的ActionMapper决定调用Action哪些方法。

Struts 2配置文件：定义控制转发流程，每个Action类的处理和结果数据如何导航到相应的表示界面都由它定义。

OGNL和类型转化：在表示界面将包含数据的Java对象进行类型转化，显示出符合页面规则的数据格式。

Struts 2校验规则：对数据输入的严格定义，保证没有垃圾数据和不符合项目需求的数据。

国际化和本地化：针对字符编码的转化，让Web项目显示各种语言版本。

SiteMesh页面布局：和以前的IFrame以及Struts中的tiles相类似的技术。使用插件式开发模式让其和Struts 2形成一个完整实体，支持Web项目中的页面布局。

下面针对这些具体细节做一下初步概念的介绍，希望初学者或入门者能对Struts 2有一个完整的认识。

1.2.1 标签库介绍

Struts 2的标签库和Struts的标签库一样，也是通过标签定义文件，也就是后缀名为tld的文件在JSP页面的定义，然后在JSP页面中调用这些标签进行表示层代码的开发。它主要分为下列几大类。

控制标签：该类标签是用来控制那些在视图中最终显示的信息，也封装了在视图中有可能根据未来需求需要调用的数据集合。

数据标签：该类标签可以修改那些动态生成的数据信息，例如Action执行后的返回结果，本地化或国际化的文本，导航指向的URL和链接等。而且该类标签还能为开发者提供调试信息。

表单标签：该类标签将原有HTML标签进行了封装，很多HTML标签都可以转为相应的表单标签，包括CheckBox、日期、下拉列表等。

非表单标签：该类标签虽然也是可以用来表示表单，但和构成表单的标签不同。主要包含错误信息、树形菜单、选择页等。

1.2.2 拦截器应用的目的

之前也对拦截器有个初步的叙述，在Struts 2框架执行Action之前拦截器对各种各样的操作都预先进行处理。这是AOP编程理念的一个实现，即对某个业务逻辑处理类或流程控制类执行之前或之后，预先或事后对一些和系统业务逻辑没有太大关系的功能进行处理。

AOP中的术语称之为“crosscut”，中文可以翻译为“横切”。而且开发人员不再需要修改原有代码就能增加很多他们想实现的需求。各组件之间也很独立，如果系统有变化部分，也能很好地对涉及变化的组件进行修改而不影响到其他组件。其实所谓的“松耦合”也就是这样的概念。

Struts 2中使用拦截器的目的有如下几点：

在Action调用之前提供系统处理流程逻辑控制。也就是Web项目中从一个视图转向或导航到目标视图的逻辑控制。

和Action交互时，提供Action执行时的一些初始信息。比如和Spring整合时，调用的一些被Spring容器管理的JavaBean类。或者是传入Action中的一切request、session中的数据。

在Action执行结束后，一些事后处理的流程逻辑也需要由拦截器实现。

修改Action中返回的Result信息，这样可以让系统导航到开发者需要它导航的目标视图。

捕获异常，保证让一些可供选择的流程被执行或者导航一些显示异常原因或错误信息的目标视图。

1.2.3 FilterDispatcher和Action概述

Action可以称之为Struts 2的核心技术。每一个URL都可以被一个特殊的Action所映射。该Action是开发人员根据Web项目特定需求进行开发所实现的。

Action中通过一个无参的execute方法来执行控制转向，并且返回一个String类型变量或者Result对象。当然也可以在Struts.xml文件中进行特殊指定方法名，这样就可以不用execute作为方法名，而是用开发人员自己指定的方法名。

（1）如果返回的是一个String类型变量，相应导航的目标视图是在Struts.xml配置文件中定义，配置文件把这个变量值和定义的目标视图名相匹配，这样系统就能正确导航到目标视图。

（2）如果返回的是Result对象，因为Struts 2本身支持返回结果是Result对象的映射导航，可是像JSP这种视图中显示的数据都是Java中的基本类型，这就涉及到类型转化概念，在Struts 2中设计者也考虑到这一点，因此也提供了类型转化的机制，在稍后章节会具体介绍。

在Web项目中，如果HTTP请求被servlet容器接收，然后导航到定义的目标视图时，有可能在之前会在Web项目中定义个一个过滤器（Filter）对它进行处理。在Struts 2中是由一个叫FilterDispatcher的类来执行该过滤器的HTTP请求处理功能。

FilterDispatcher也是Struts 2的一个核心技术，它提供了处理HTTP请求用来访问Web项目框架的访问方法。假设我们启动了某个Web项目，很多可设置的元素都已经在该Web项目框架中定义，例如Spring可管

理的JavaBean配置、Action的映射定义等。在此时FilterDispatcher会执行下列这些操作。

读取静态数据信息：比如某些文件或者JavaScript代码等。

决定需要转发或导航的Action配置。举例说明，从HTTP请求中发出的目标视图的导航定义，一般在Struts.xml配置文件中都配置了Action名字，系统会根据Action配置去配置文件中寻找这个Action名字的Action。也就是说通过FilterDispatcher去搜寻后缀名为“Action”的所有Action映射。

创建Action的Context：从HTTP请求中发出的数据信息都是存储在request或session中，而Struts 2则封装了这些HTTP的对象，创建Context对象作为存储数据之处，并且通过Context的一些内置方法可以得到HTTP请求存储在request或session中的数据。

创建Action代理：Action代理是一个附加的逻辑处理。它实际上包含了Web项目所有的配置和Context信息，用来处理HTTP请求并包含请求处理完毕后哪些要返回的Result对象。

内存清理和性能优化：为了确保在Web项目启动运行后，内存不会溢出，以至出现运行越来越慢的情况，FilterDispatcher会自动执行清理动作，主要清理Context对象，保证系统中没有冗余对象或垃圾对象。

1.2.4 Struts 2配置文件处理

Struts 2的配置文件也是Struts 2框架的核心技术。它主要将Web项目中一些属性和需要配置的机制，以及Web页面导航流程控制都配置起来。

配置文件主要分为struts.xml和struts.properties两个文件。

struts.xml文件：在该文件中主要对Action即Struts 2的主要业务逻辑控制类进行配置，除此之外对拦截器、FreeMarker等也可以进行配置。

struts.properties文件：该文件定义了Web项目整个应用程序范围的设置。并且对那些可以改变Web项目框架功能的配置参数，进行配置定义。

1.2.5　OGNL介绍和类型转换目的

OGNL全称为Object Graph Navigating Language，翻译成中文就是对象导航图语言。该语言是一种表达式语言，但是它有它的绑定方式。同一个OGNL表达式可以用于得到或者设置Java对象的属性。其实就是简化了Java中的getter、setter方法。而且表达式除了显示、得到对象属性之外，表达式也被允许带有计算功能。在Struts 2中使用OGNL的目的其实就是三个方面：

OGNL本身的类型转换机制，允许视图中的值和数据进行类型装换。

OGNL能使有些数据源可以简单映射到视图中。

OGNL能将Web组件和相应的Java对象绑定在一起。

Struts 2中的类型转换一方面需要OGNL来支持，另一方面由于它自带了类型转换器，所以开发者调用这些缺省设置的转换器配合OGNL一起使用，就能实现类型转换。而且开发者还可以自己定义符合自己Web项目需求的类型转换器。之所以类型转换是因为视图技术上显示的值一般都是String类型，而在Java对象中的属性不一定都是String类型，因此有必要让视图上的值和Java对象中的属性进行双向的类型转换。除此之外，一些判断视图上的值是否为空的操作处理，也需要类型转换才能定义空值或不空值时在视图上显示的值格式。

1.2.6　校验

通常开发者在开发自己的Web项目时，针对即将被存储进数据库的一些数据会进行校验，保证数据库中的数据不是垃圾数据，或者是不符合项目要求格式的数据。甚至在视图层对于用户在视图中输入的数据，也可能有特定的数据格式要求，如输入数据不能为空或者不能为特定字符这样的要求。Struts 2提供了一个稳固而又健壮的校验机制，使用该机制能对Struts 2中开发者开发的Action进行快速而又便捷的校验操作。

Struts 2中的校验机制有以下几种实现方式：

通过Struts.xml和Struts.properties配置文件定义校验。

继承ActionSupport类，重写它的一些方法在Java代码中进行校验。

使用校验拦截器，或者开发者自己定义的校验拦截器类，在视图上输入的数据还没有传递到Action控制层时来对其进行校验。

1.2.7　Web项目国际化原由

国际化即internationalization，因为在这个单词第一个字母i和最后一个字母n之间有18个字母，所以通常又被称为i18n。为什么在Web项目中要进行国际化，原因是世界上所有的国家和地区都是位于不同的区域，每个区域文化的不同，使两个不同国家或地区的开发者开发的项目受到文化、语言的限制也有所不同，所以当一个国家或地区的开发者希望自己开发的Web项目可以共享给其他国家和地区的开发者的，也就需要进行国际化来解决由文化、语言带来的问题。

国际化主要是针对语言和信息格式化进行处理。除了字符编码和语言之外，信息格式指的是文本、日期、时间、数据处理等信息的显示格式。

它还包含了一个本地化的问题，本地化也被称为l10n，原因和国际化称为i18n是一样的。举例说一个中国的开发者，需要把一个英语的Web项目改版成中文字符显示信息的Web项目，那就需要进行语言和信息格式的处理，这就是本地化。

在Struts 2中，有两种进行国际化操作的方式：

在Struts.properties文件中设置处理各种字符编码或者语言的属性文件，并由此扩展这些属性文件，让Web系统使用形成各种语言或信息格式化的配置信息。

直接装载Struts 2提供的自带信息资源包。

1.2.8 SiteMesh页面布局框架介绍

在Struts中对页面布局的开发工作都由tiles来解决，但在Struts 2中，设计者使用了一个专门来进行页面布局的框架。因此严格说它不应该属于Struts 2框架的，但是在Web项目中使用页面布局是无可厚非的一件事情，况且tiles在Struts中的使用也相当成功。

而SiteMesh不仅仅具有原先tiles的功能，它还支持一些属于设计模式中装饰模式的开发理念，开发者同样可以使用它对原有页面进行风格等的修改布置。有一点可以肯定的是：它和tiles一样都增强了页面的复用性。因此如果使用Struts 2开发Web项目，它也是不可或缺的。SiteMesh主要针对以下几点做了处理：

所有Web项目页面的布局。

所有Web项目的链接处理。

所有Web项目页面风格一致性。

1.3　为什么要用Struts 2

在上一节中对Struts 2的各个组成部分进行了大致的说明，也许有些读者对Action、Filter这些技术名词还只是一些模糊的概念，也不明白为什么Struts 2需要这些作为它的一部分。因此有必要对Struts 2在Web项目中进行操作的整体流程详细说明。如图1.3所示是Struts 2的整体结构图。

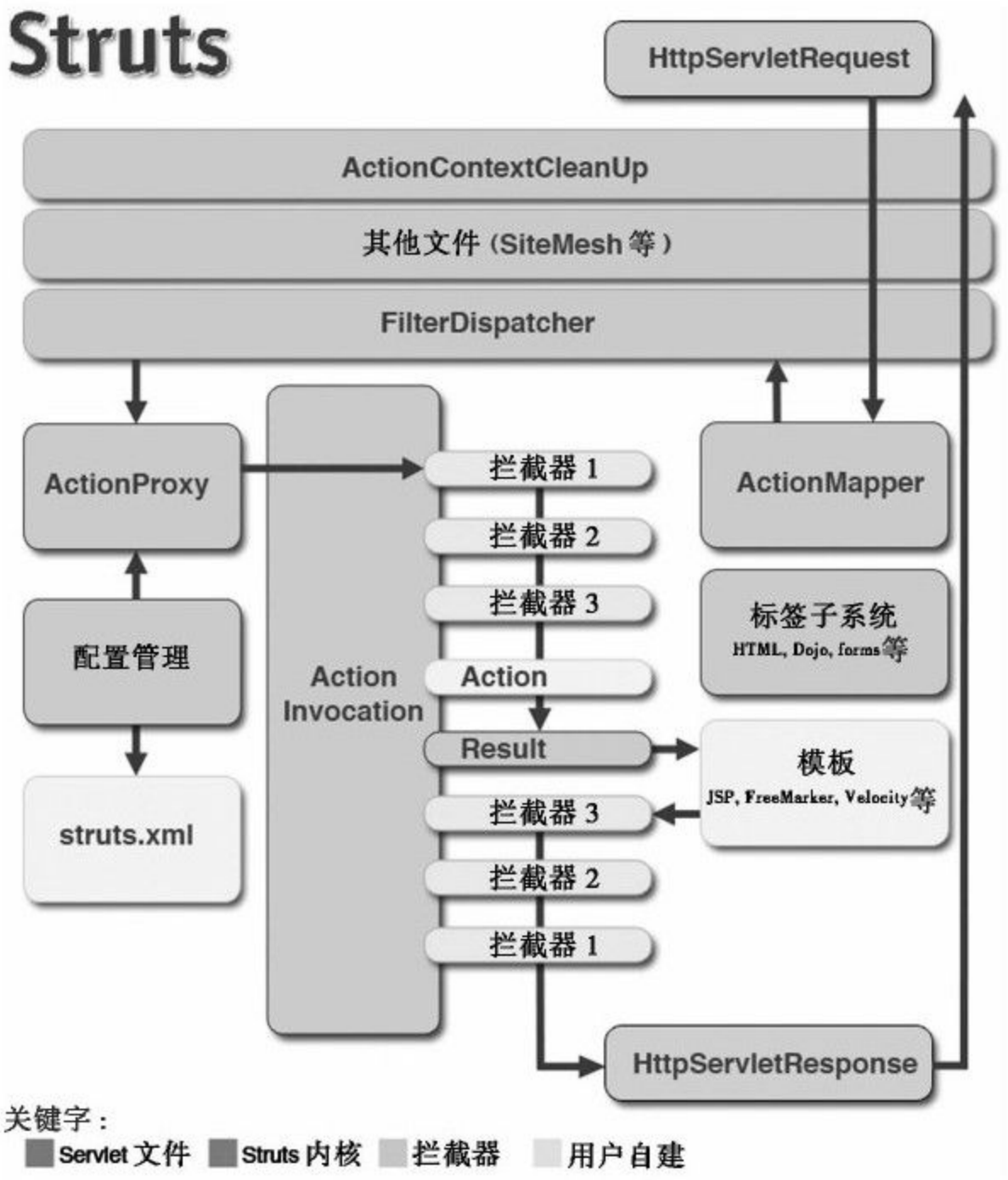

图　1.3　Struts 2整体结构图

分析图1.3的工作流程：

（1）右上角的HTTP请求发出后，如果在Context中有对象，先通过过滤器ActionContextCleanUp清理Contex对象。

（2）然后判断如果项目中使用了像SiteMesh这样的框架，那必须先通过这样的过滤器，然后再使用ActionMapper进行调度。

（3）在配置文件struts.xml中寻找相应的URL控制，通过ActionInvocation找到Action，如果被找到的Action有拦截器配置，则在Action的前后执行拦截器。

（4）Action主要功能则是调用业务逻辑类，对业务逻辑进行处理之后，则返回一个Result，在struts.xml中根据相应定义的Result导航目标视图，产生Response返回到相应的目标视图页面。如果还有过滤器设置，则还需要进行过滤器的执行和控制。

通过以上的说明，可知Struts 2还是和基本的MVC模式概念有些区别，在“VC”（视图和控制）部分，调用时都是通过相应的过滤器或拦截器组件进行处理，也就是比起传统的MVC模式，它提供了很多附加的操作处理，但是这些操作处理原先在MVC中都是一起执行的，现在只不过把他们和核心的流程分离开。

因此不仅仅是MVC的分离，甚至某些核心和非核心的控制处理也分开了，这更加体现了松耦合的开发理念。

使用Struts 2的目标其实已经很清楚了，面对以前非结构化或比较混乱的项目开发框架，Struts 2可以让项目结构化，而且对于很多新技术尤其是视图技术提供了很好的支持。对于开发人员，在Struts 2中开发自己新的符合业务逻辑的组件自由度更高，而且也不会使原有结构发生改变或混乱。

在Struts 2诞生之前，同类型中比较知名的基于MVC模式的Web项目开发框架有WebWork、Tapestry、JSF、SpringMVC。Struts 2诞生后，WebWork已经和Struts合并，剩下的还有其他三个开发框架。下面对其一一进行介绍，然后将它们和Struts 2相比较，就能明白这些框架的技术优势和劣势分别在哪里。

1.3.1 Tapestry优劣点

该框架是基于组件开发的框架。它有以下几点优势：

非常高的代码复用性。

将开发者从烦琐的JSP代码中解脱出来，取而代之的是真正面向对象的方法，而不是URL解析。

对国际化的支持。

精确地错误定位，将错误定位到源程序中的行，取代了JSP中莫名奇妙地错误提示。缺点有以下几点：

学习文档都是很概念化的，没有什么实用性。

学习曲线很陡峭，即入门者开始学习时很难理解和掌握。

产品发布周期也很长，要很长时间才会把开发者提出的技术缺陷在下一版本中解决。

几乎没有什么成功的Web项目开发示例。

IDE（集成开发环境）对其的支持很弱。在开发工具中使用它进行Web项目开发，几乎没有很好的便捷开发方法。

现在的Web项目开发中，都很重视快速开发理念，而Tapestry由于以上的缺点往往拖延了开发进度，因

此几乎没有IT公司会使用它开发Web项目。

1.3.2 JSF优劣点

该框架也是基于组件开发的框架，而且它诞生的时间比Tapestry还要早，尤其是它的设计者出身于Sun公司，因此也是Sun力推的框架。它有以下几点优势：

J2EE标准，Sun是制定行业标准和技术标准的公司，因此JSF是标准的。

易于开发，吸引了大批原C/S结构开发者，其原因也是它基于组件开发的理念。

丰富的导航框架，也让开发者从JSP代码中解放出来。缺点有以下几点：

虽然导航框架丰富，但是标签不是很丰富，特别是对原有JSP标签的支持。

JSF也没有什么好的成功项目典范，虽然有开发者使用它作为开发，但是更多的IT公司对其还是采取小心翼翼的观望态度。既然是标准，那为什么大家都不采用这个标准呢？

■作为标准，对J2EE的支持不是很好，特别是安全机制方面。

■IDE（集成开发环境）对其的支持很弱。在开发工具中使用它进行Web项目开发，几乎没有很好的便捷开发方法。

■学习曲线同样陡峭。

值得一提的是目前有个richface框架，它较好地支持了该组件，现在有很多开发者将其作为JSF的补充，但是由于它不是标准，因此推广起来并不是很容易。

1.3.3 SpringMVC优劣点

SpringMVC其实是Spring框架中对MVC模式支持Web开发的应用。它有以下几个优点：

和Struts 2一样具有一个转发过滤器，控制很灵活。

对于值绑定和校验机制可以让开发者自行开发自己的组件。

也是用IOC来实现的。

和其他视图技术的整合非常好，支持力度也很强。

该框架和Struts 2一样是基于松耦合和AOP理念实行开发，因此很多优点其实就是Struts 2的优点。但

是相比较而言缺点也很明显，比如：

JSP中要写很多代码。没有把开发者从繁重的工作中解放出来。

控制器过于灵活，缺少一个公用控制器。不像Struts 2有一个专门的FilterDispatcher来进行控制导航转发处理。

后缀名为xml的配置文件太多，让开发者很茫然。这也正是Struts 2中为什么要引入properties属性文件，配合xml文件进行项目系统参数配置的原因。

由于Struts 2的丰富标签库以及对整体控制器的开发和配置，再加上同样也是采取流行的松耦合和AOP开发理念进行开发，相信Struts 2在同类型的Web项目开发框架中应该是很有优势的。

下一节将使用最常用的开发工具MyEclipse，进一步说明Struts 2的各个技术优势，以及新建Web项目中Struts 2的基本配置。

1.4　在Web项目中使用Struts 2

在开始创建新的Web项目让其使用Struts 2技术之前，请先去Struts 2的官方网站http：//struts.apache.org/2.0.9/index.html下载Struts 2。最好下载struts2.0.11.1all.zip，因为它包含了Struts 2的所有内容，而且之后的说明和解释都是在解压缩包后的基础上进行的。

1.4.1　Struts 2开发包的结构

如果以上操作都完成了，请解压下载的Struts 2开发包（struts2.0.11.1all.zip）。会看到如图1.4所示红框中的几个文件夹。

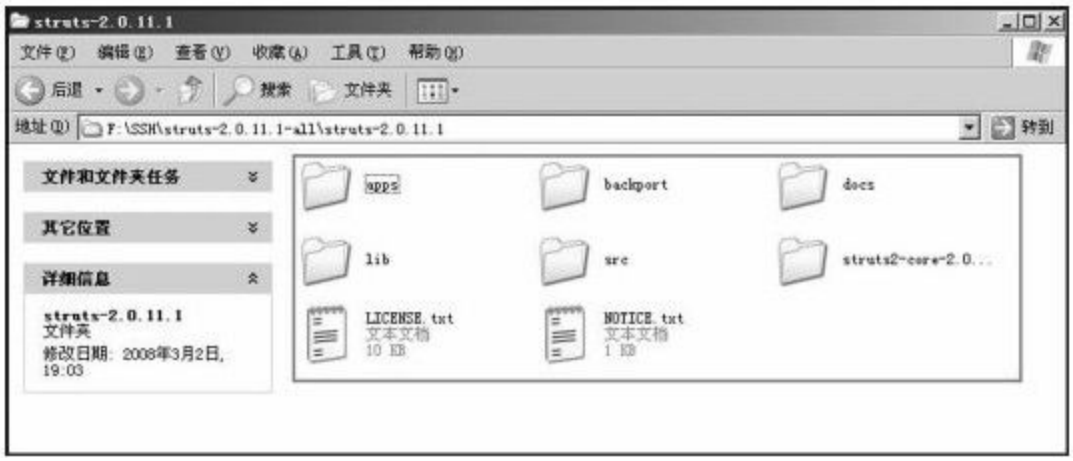

图　1.4　Struts 2文件结构图

在上图中除了两个文本文件是介绍license和官方网站介绍外，将对其他5个文件夹的内容逐一进行介绍。

（1）apps文件夹中都是使用Struts 2开发的一些官方示例。特别是Struts 2blank2.0.11.1.war是Struts 2在Web项目中基础配置的例子，如图1.5所示。

（2）backport文件夹中提供的是一些Struts 2使用Java 4和Java 5互相转化的示例。由于Struts 2主

要是支持Java 5的JDK（Java Develop Kit），而有些公司还在使用Java 4即Java 1.4的JDK。因此Struts 2设计者提供了这两个Java版本互相支持的转化工具和项目的示例，具体内容如图1.6所示。

图　1.5　apps文件结构图

图　1.6　backport文件结构图

（3）docs文件夹则是Struts 2设计者提供给入门者的所有学习文档，初学者可以使用这些文件配合自己的实际操作来加深对Struts 2的理解。如图1.7所示。

（4）lib文件夹提供了Struts 2在项目开发中所有应该支持的jar包，读者可以将这些包导入自己开发的Web项目使用的类库中。文件结构图如图1.8所示。

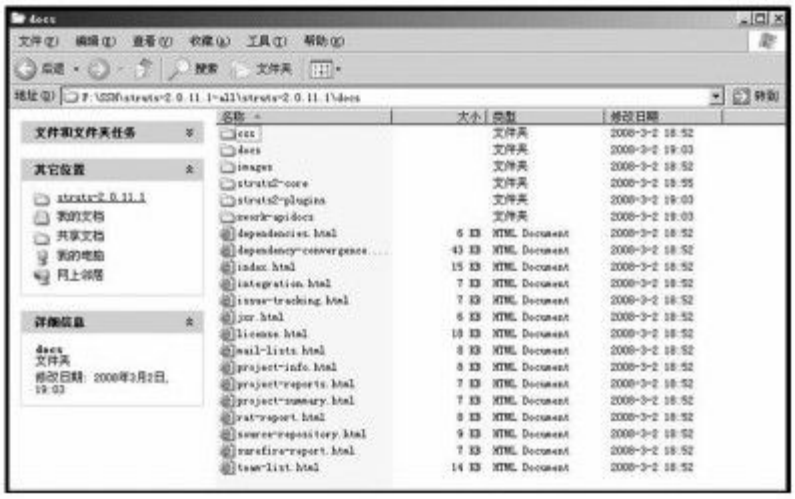

图　1.7　docs文件结构图

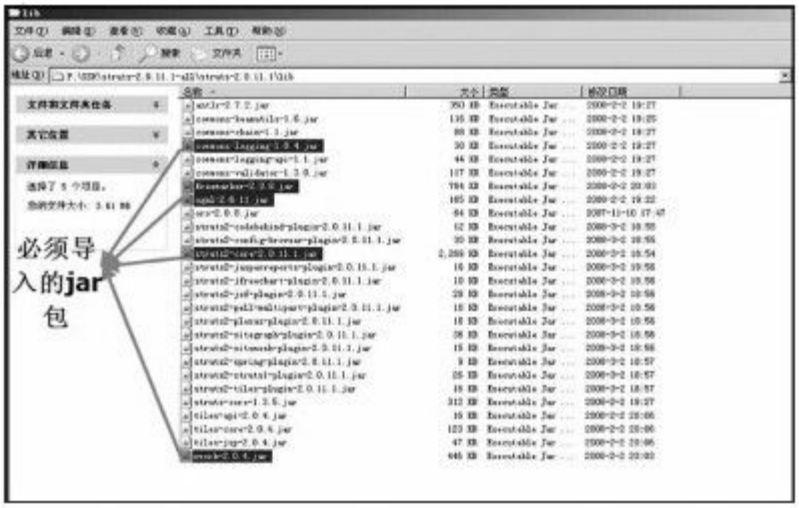

图　1.8　lib文件结构图

注意：不是所有的jar包都是在读者的Web项目中需要的，读者可以选择部分jar包导入到自己的Web项目类库中。一般情况下xwork2.0.4.jar、Struts 2core2.0.11.1.jar、ognl2.6.11.jar、freemarker2.3.8.jar、commonslogging1.0.4.jar是jar包一定需要导入的。

（5）src文件夹是Struts 2所有的源代码，因为Struts 2也是一个开源的框架之一。因此所有的源代码都开放给开发者，开发者可以查阅这些代码，了解

Struts 2的内部或底层实现机制。因此从这一角度来看，也是入门者学些Struts 2的一个很好的文档资料。如图1.9所示。

图　1.9　src文件结构图

1.4.2　创建MyEclipse项目

熟悉完Struts 2的文件架构后，打开MyEclipse，如图1.10所示是MyEclipse的整体视图。单击File | New | Project菜单命令，打开如图1.11所示的界面。选择图1.11中蓝色显示的Web Project项，单击“Next”按钮，打开如图1.12所示的界面。

在图1.12的界面中输入项目名称，比如本章是第1章，输入的项目名称是C01。项目文件一般都是放在缺省的MyEclipse设置的workspace文件夹下，当然也可以取消勾选“Use defult location”复选框，然后单击下一行右侧的“Browse”按钮，选择想要存放的项目路径。余下的都是项目中的源代码，即Java代码存放的路径和JSP等视图文件存放的路径。

URL是在项目部署成功后，按照这个URL去访问Web系统。然后可以选择支持的J2EE的版本，这里是选择Java EE 5.0单选按钮，然后单击“Finish”按钮完成Web项目初始配置。

没有任何Java文件、视图文件，以及配置文件的空白项目，如图1.13所示可以把相应的Struts 2文件配置放在这个项目中。

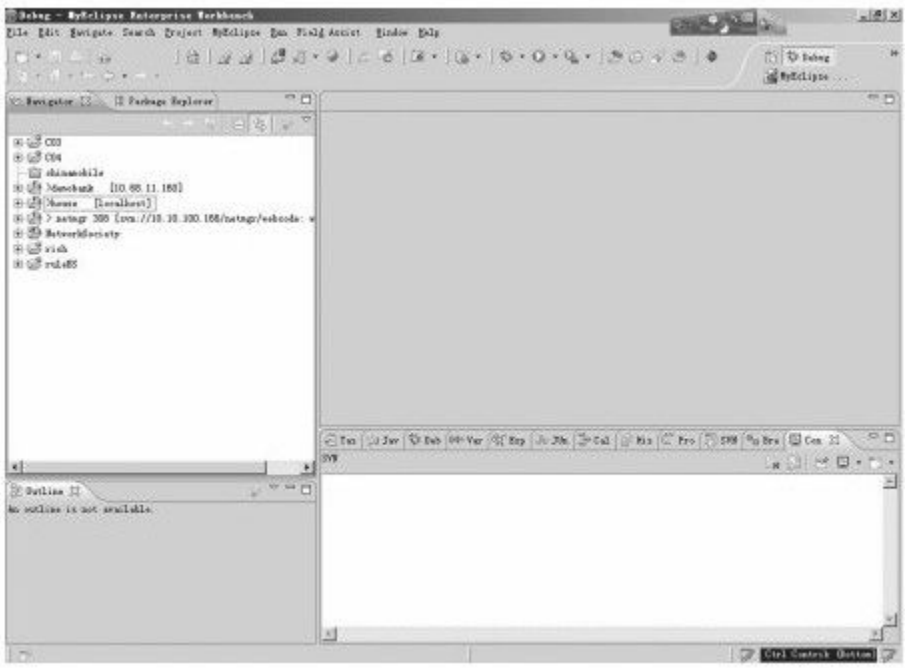

图　1.10　MyEclipse开发视图

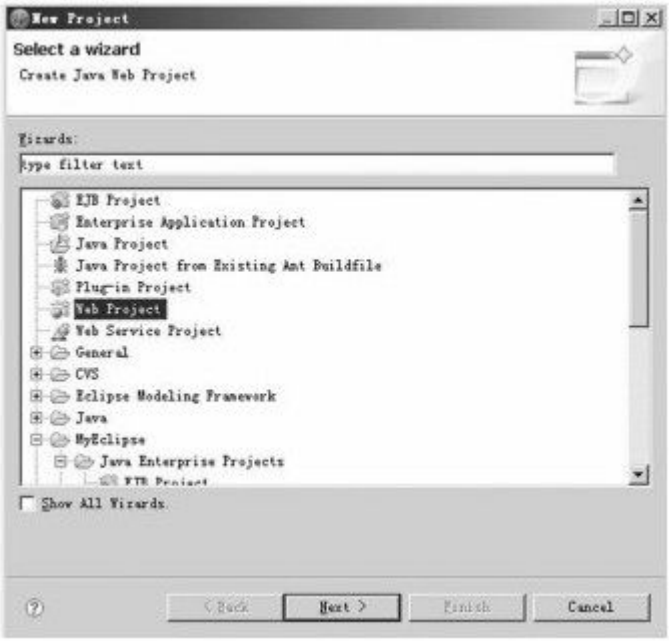

图　1.11　新建Web项目界面

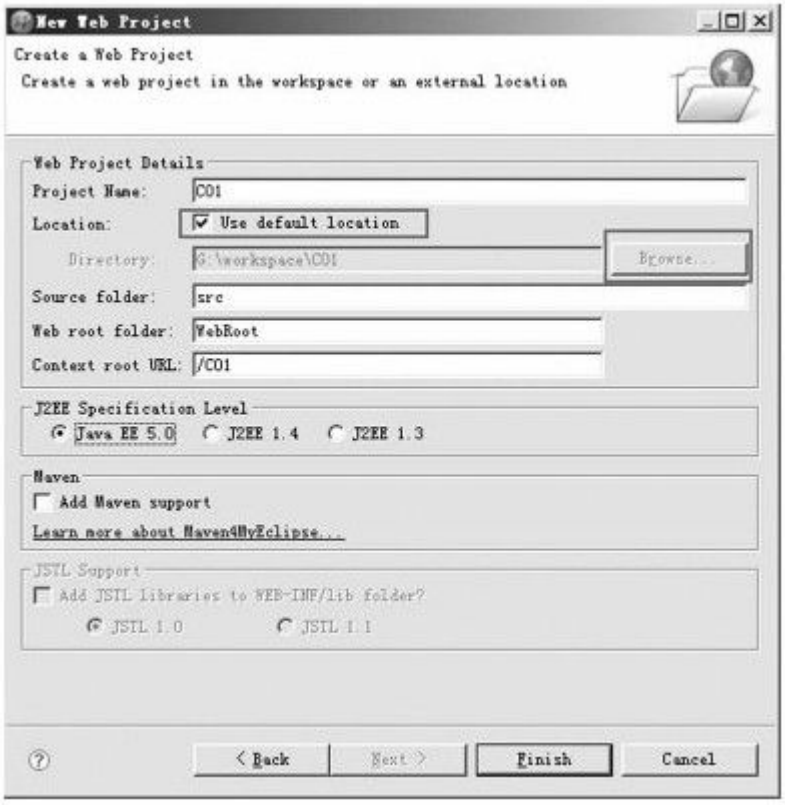

图　1.12　设置新项目属性

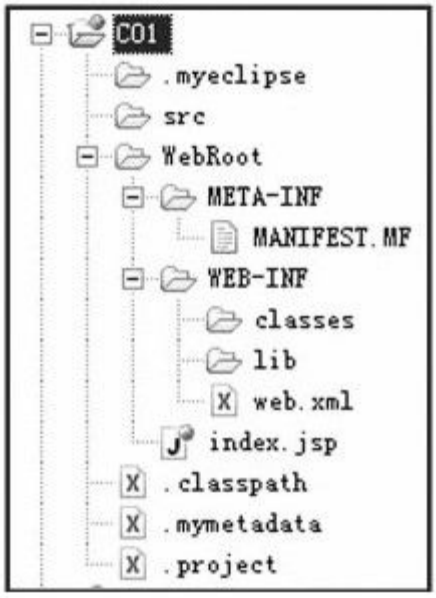

图　1.13　C01项目基础结构

1.4.3　在项目中配置Struts 2

项目创建好之后，开始学习配置Struts 2，详细步骤如下所示：

（1）将Struts 2的“lib”文件夹中最基本的类库，放到CO1项目的“WEBINF＼lib”下，如图1.14所示。也可在项目名上单击鼠标右键，在弹出的快捷菜单中选择“properties”命令，如图1.15所示。

（2）弹出如图1.16所示的对话框，在其中选择“Java Build Path”项，然后在“Libraries”选项卡中，可以看到已经导入的Struts 2的几个jar包。选择jar包路径，可以选择新的jar包或更改原有jar包的设置按钮。

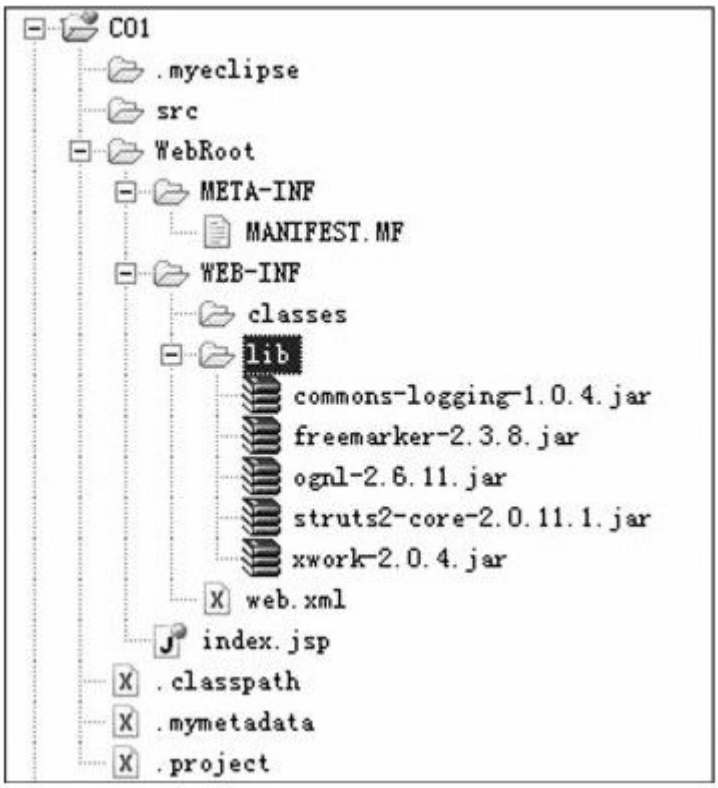

图　1.14　C01项目下的文件夹

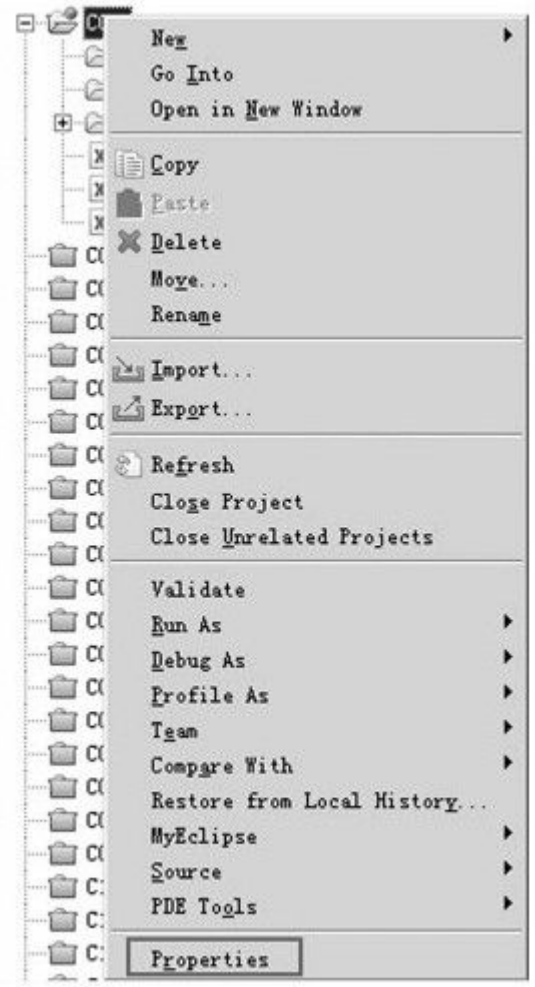

图　1.15　选择“Properties”命令

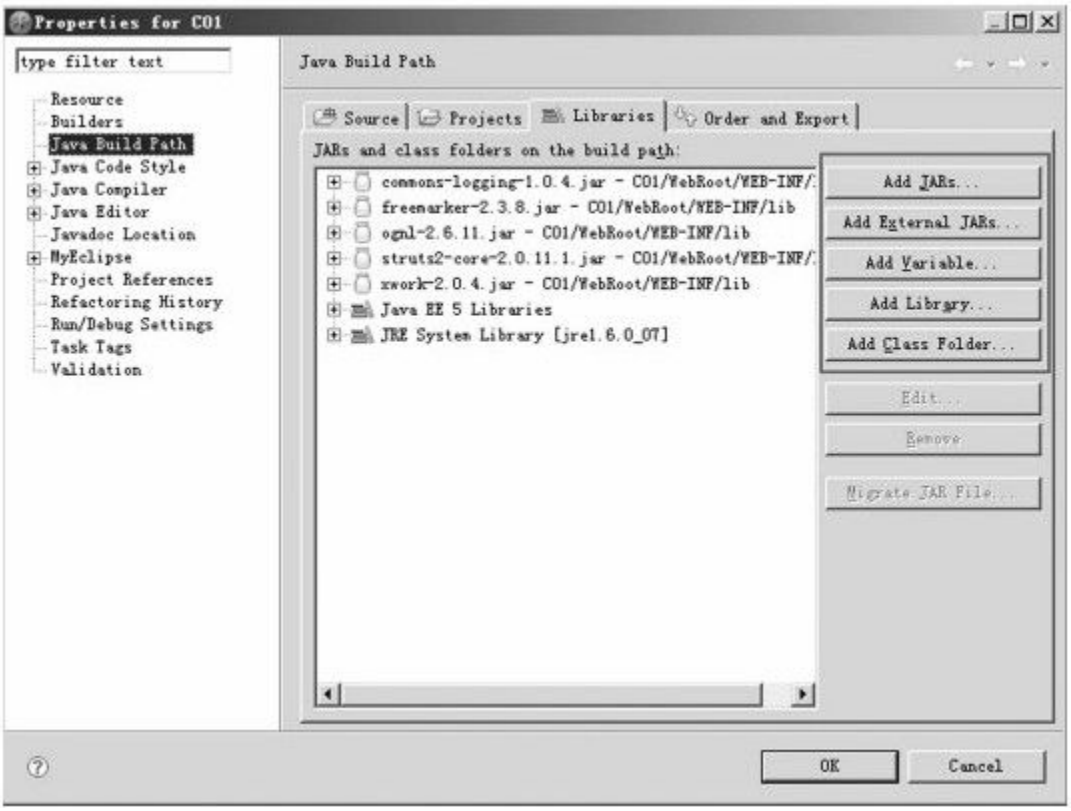

图　1.16　导入项目支持的jar包

（3）在图1.13中，“WebRoot/WEBINF”文件夹下有一个名为“web.xml”的文件。该文件是Web项目的属性整体配置文件，特别是一些过滤器、拦截器，还有Servlet的配置都在该文件中定义。如果要把Struts 2在C01的项目中使用，则要把web.xml修改成如下代码：

```
    <! ……文件名：
web.xml……>
    <? xml version="1.0"encoding="UTF8"? >
    <webapp
version="2.5"xmlns="http: //java.sun.com/xml/ns/jav
aee"
    xmlns:
xsi="http: //www.w3.org/2001/XMLSchemainstance"
    xsi:
schemaLocation="http: //java.sun.com/xml/ns/javaee
    http: //java.sun.com/xml/ns/javaee/webapp__2__
5.xsd">
    <filter>
    <! 过滤器名字>
    <filtername>Struts 2</filtername>
    <! 过滤器支持的Struts 2类>
    <filterclass>
    org.apache.Struts 2.dispatcher.FilterDispatcher
    </filterclass>
    </filter>
    <filtermapping>
    <! 过滤器拦截名字>
    <filtername>Struts 2</filtername>
    <! 过滤器拦截文件路径名字>
    <urlpattern>/</urlpattern>
    </filtermapping>
    <welcomefilelist>
    <welcomefile>index.jsp</welcomefile>
    </welcomefilelist>
    </webapp>
```

从代码中可知已经声明了Struts 2的过滤转发器的映射，以及Struts 2拦截的文件路径定义，具体的

代码含义将在后面章节介绍，这里只是让读者明白怎么在Web项目中使用Struts 2。

（4）Struts 2还有自己基本的配置文件需要放置在C01项目中，也就是之前介绍过的struts.xml和struts.properties文件。新建这两个文件，将其放在项目的“src”文件夹下，以后建立整个项目时，它们也会在“WebRoot＼WEBINF＼classes＼”中出现。放置完之后，效果如图1.17所示。

图1.17中的messageResource.properties是由struts.properties定义的扩展属性文件。它们之间关系就相当于Java中父类和子类的关系。子类扩展父类，同样也适用于这两个属性文件。这三个文件的代码内容如下所示，这里只是让读者知道里面有些什么内容，至于这些内容代表含义在后面章节会详细介绍。

```
    <！……文件名：
struts.xml……>
    <？xml version="1.0"encoding="UTF-8"？>
    <！DOCTYPE struts PUBLIC
    "//Apache Software Foundation//DTD Struts
Configuration 2.0//EN"
    "http: //struts.apache.org/dtds/struts2.0.dtd">
    <struts>
    <！Action所在包定义>
    <package name="C01"extends="strutsdefault">
    <！全局导航页面定义>
    <globalresults>
    <result name="global">/jsp/login.jsp</result
>
    </globalresults>
    <！Action名字，类以及导航页面定义>
    <！通过Action类处理才导航的Action定义>
    <action name="Login"
    class="com.example.struts.action.LoginAction">
    <result name="input">/jsp/login.jsp</result>
    <result name="success">/jsp/success.jsp
</result>
    </action>
    <！直接导航的Action定义>
    <action name="index">
    <result>/jsp/login.jsp</result>
    </action>
    </package>
    </struts>
    <！……文件名
struts.properties……>
    struts.custom.i18n.resources=messageResource
    <！……文件名
messageResource.properties……>
    user.required=请输入用户名!
    pass.required=请输入密码!
```

如图1.18所示的文件结构，就是使用Struts 2的Web项目C01的基础配置，可以在里面新建视图文件和Java文件，进行自己的Web项目开发。

通过简单的配置，可知Struts 2的配置不是很难，而且在MyEclipse帮助下开发工作也是相当快捷的，并且Struts 2提供的学习资料和示例也是非常丰富的，因此学习曲线也是相当的平缓。在后面的章节，将对Struts 2的各个组成部分和技术细节使用代码示例进行说明。

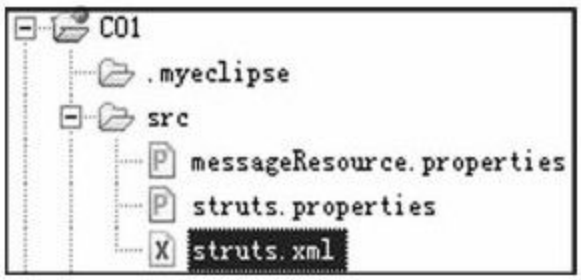

图　1.17　项目属性文件放置图

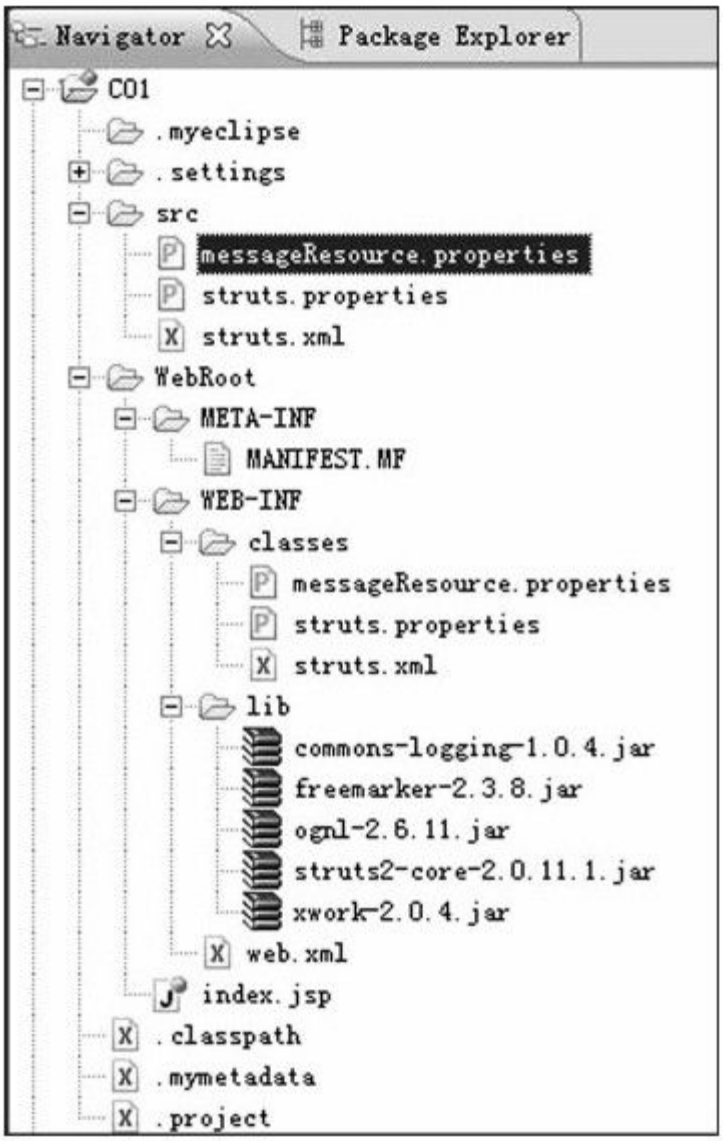

图　1.18　项目文件整体结构

第2章　Web基础技术概述

在具体介绍Struts 2的各部分之前，针对一部分B/S结构的项目开发入门者，有必要将Web开发中涉及的一些基础技术知识，特别是和J2EE有关的知识做一个说明。如果有的读者已经对Web项目开发比较熟悉，可以略过本章，继续以后章节的学习。

2.1　B/S和C/S系统区别

Web项目还有另外一个开发者耳熟能详的名字就是B/S系统。它是Brower/Server（浏览器/服务器）的缩写，客户端只要有浏览器就可以通过输入URL，看到Web项目内容。而在服务器端只需要安装数据库和Web容器，把相应的Web项目系统部署在容器中。通过客户

端浏览器，用户就可以和服务器端的数据进行数据交互操作。

所谓的C/S系统是Client/Server（客户/服务器）的缩写。服务器端都是用性能很高的PC、工作站或者小型机，并结合数据库系统、客户端安装专用的客户端软件，即可调用服务器端的数据进行业务处理。

B/S系统和C/S系统之间也有很大的不同。

硬件环境：C/S一般建立在专用网络，小范围的网络环境中，局域网通过专门服务器提供连接和数据交换服务。B/S则是建立在广域网上，没有专门的网络环境，比C/S适应范围更加广，只需要浏览器就行了。

安全要求：C/S面向相对固定的用户群，对信息安全有很强的控制性。而B/S相对较弱。

程序架构：C/S更加注重流程，对权限多层次检验，较少考虑性能和运行速度。B/S对安全和运行速度的考虑比较仔细，需要进行很多深层次的优化工作。

重用性：C/S重用性很差，而B/S因为支持松耦合的概念，因此组件相对独立，能够很好的被重用。

系统维护：C/S过于注重整体性，处理bug或者更新系统很困难。而B/S因为是组件化的，因此如果有bug或者系统更新，可以只对产生问题或者需要更新的组件进行更换或处理即可。

接口：C/S很多是建立在Windows平台上，因此表现形式有限。B/S因为只需要浏览器，表现形式多种多样，而且大部分的开发难度都不大，因此开发成本也比C/S小很多。

信息流：C/S一般都是典型的集权式处理，交互性不高。而B/S信息流向变化多端，目前流行的B2B、B2C都是很好的例子，感觉更像一个数据交互中心。

2.2　JSP和Servlet介绍

很早之前有关Web项目的技术，人们认为只是让文本、图片等信息显示在网页上，因此最早的网页内容都是用HTML来实现的，可是HTML只是提供很多静态信息的显示功能，也就是说如果和数据库交互，把存储在数据库中的数据动态地显示在网页页面上，HTML便无法实现。因此为了解决这个问题，越来越多的Web技术便应运而生，JSP和Servlet就是其中的两个技术。

2.2.1　什么是JSP

什么是JSP呢？JSP是为HTML页面添加动态信息和内容的方法之一，也是最主要的方法之一。它有自己的标签库，HTML也有自己的标签库，但是两者可以互相转化而且可以互相支持。

JSP有一大优点就是在不同的网页浏览器上，其显示在页面上的内容、信息、格式都是一样的。它支持动态信息的显示，也能很方便地传递数据到HTML的表单中。

2.2.2 什么是Servlet

什么是Servlet呢？Servlet其实是运行在服务器端的标准Java程序。Web项目通过编译，一般都是部署到容器中，而容器一般都安装在服务器端，因此可以说在容器中运行的Java程序都是支持Web项目的各种操作。比如之前所说的MVC模式，不仅仅在Model和Controller层有Java程序，而且View层的视图页面也是被编译成Java程序在容器中运行。

所以JSP编译后产生的文件也就是一个Servlet，只是两者的表现形式不同。JSP还包括很多HTML标签或其他JSP自己的标签。而Servlet就是Java程序，只是通过J2EE的类库中的打印方法调用HTML标签而已。

2.2.3 JSP内置对象

JSP提供了很多内置对象，这些对象大都是和J2EE中的一些类库中的类有关，特别是J2EE中的Javax包中的类库。前面也说了，JSP其实在经过Web项目容器编译后都是生成为一个class文件，里面的内容如果转化成Java代码就是一个Servlet类。因此Servlet类中的对象，和JSP中内置的对象有一一对应关系。

下面对JSP的内置对象进行介绍。

Request：Request表示HttpServletRequest对象。包含了有关浏览器的信息，并且提供用语获取cookie、header、session数据的方法。

Response：Response表示HttpServletResponse对象。提供了用于设置送回浏览器的相应方法，比如

request中提到的cookie、header等。

Out：Out对象是javax.jsp.JspWriter的一个实例。提供了用于向浏览器显示输出结果的方法。

pageContext：它表示的是javax.servlet.jsp.PageContext对象。用于方便存取各种范围的名字空间、Servlet有关对象的API，并且包括了通用的Servlet等相关功能的方法。

Session：session表示一个请求的javax.servlet.http.HttpSession对象。它可以存储用户的状态信息。

Application：该对象表示javax.servlet.ServletContext对象，有助于查找有关Servlet引擎和环境的信息。

Config：Config表示javax.servlet.ServletConfig对象，用于存储Servlet实例的初始化参数。

Page：Page表示从它所在的页面产生的一个Servlet实例。

Exception：针对错误网页和未捕获的例外进行处理操作，是继承于Java的异常类。JSP中还有几种常用的标签，有时也被开发者称之为“动作”，它们是：

jsp：include：在页面被请求时引入一个文件。

jsp：useBean：搜寻或实例化一个JavaBean。

jsp：setProperties：设置JavaBean属性。

jsp：getProperties：输出某个JavaBean属性。

jsp：forward：将请求转到一个新页面。

jsp：plugin：根据浏览器类型为Java插件生成OBJECT或EMBED标记。

其中include是指动态的include会检查所含文件的变化，也就是说文件内容变化了，则相应引入该文件的页面内容也会变化。适合动态页面显示，而且还能带参数。代码样式如下：

```
<jsp：include page="xx.jsp"flush="true"/>
```

而且引入文件的页面只是显示文件的结果，页面URL还是原来那个书写了上述代码的页面，类似于函数调用。而jsp：forward则会转到新页面，类似于goto语句。

2.2.4 Servlet的生命周期

Servlet不同于JSP的是因为它有它的一个生命周期。此生命周期是存在于容器中的，如果容器被重启或者停止，则生命周期重新开始或结束。它的生命周期流程如下：

根据用户发出的请求，容器根据响应创建了HttpServletRequest和HttpServletReponse对象，并且创建了Servlet的一个实例，把上述两个对象当作参数传递进来。此时会对Servlet对象进行初始化操作，调用Servlet的init方法。然后初始化完成后，会调用Servlet的核心方法service方法。根据传递进来的请求是get还是post属性调用doGet或doPost方法。在这两个方法中处理请求，完成后直接调用destroy方法，结束Servlet生命周期。

简而言之，Servlet的生命周期包括加载、实例化、初始化、处理请求、销毁服务等几个部分。其中所说的get和post都是数据传输转向的方法。它们之间的不同有如下几点：

get是从服务器上得到数据，而post是向服务器传递数据。

get将表单中的数据按照variable=value的形式，添加到action所指向的URL后面，并且两者使用“？”连接，而各个变量之间使用“＆”连接；post是将表单中的数据放在表单的数据中，按照变量和值相对应的方式，传递到action所指向的URL。

get是不安全的，在传输过程中，数据被放在请求的URL中，而如今现有的很多服务器、代理服务器或者用户代理都会将请求URL记录到日志文件中，然后放在某个地方，这样就可能会有一些隐私的信息被第三方

看到。另外，用户也可以在浏览器上直接看到提交的数据，一些系统内部消息将会一同显示在用户面前。Post的所有操作对用户来说都是不可见的。

get传输的数据量小，这主要是因为受URL长度的限制；post则可以传输大量的数据，所以在上传文件只能使用post。

get限制表单的数据集的值必须为ASCII字符；而post支持整个ISO10646字符集。

get是表单的默认数据传输方法。

仔细研究Servlet的类属性和方法，还会发现有两个API，一个是forward方法，另一个是redirect方法。它们的区别如下：

Forward表示的功能仅是容器的控制器转向，在浏览器中不会显示转向后的地址。而redirect则是完全

的跳转，浏览器显示转向后的地址，并重新发送请求。因此前者效率较高，也有助于隐藏实际的链接地址。但是如果想跳转到其他服务器的地址，那就有可能需要使用redirect方法。一般在Web项目中，不同服务器的转发情况很少发生。因此大多数情况下都用forward方法。

2.3　XML知识介绍

XML全称为Extensible Markup Language（可扩展标记语言）。其本身是一组在文档中建立和调整数据的指导方针。主要功能如下：

解析数据到内存中。

可处理分析的数据，并且可以使用样式表单进行转换。

处理数据结构。

2.3.1　XML的格式

XML的格式可以分为标记、元素和属性。

（1）标记是左尖括号（＜）和右尖括号（＞）之间的文本。有开始标记（例如＜name＞）和结束标记（例如＜/name＞）。

（2）元素是开始标记、结束标记以及位于二者之间的所有内容。

（3）属性是一个元素开始标记中的名值对。如＜name sex=″male″＞中的sex就是属性。

2.3.2　XML的文档类型

XML的文档类型有三种：

无效文档。有遵守XML规范定义的语法规则。如果开发人员已经在DTD或模式中定义了文档能够包含什么，而某个文档没有遵守那些规则时，这个文档就是无效文档。

有效文档。遵守XML语法规则也遵守在DTD或模式中定义的规则。

格式良好的文档。遵守XML语法，但没有DTD或模式。

注意：XML的元素是区分大小写的，而且不能重叠，必须有根元素和结束标记，属性取值要加引号。

2.3.3　XML的用途

XML是用来存储、传输和交换数据的，并不是用来显示数据的。主要用处如下：

将数据从HTML中分离出来。HTML文件中包含了要显示的数据，但如果使用XML，数据就可以单独存储在一个XML文件中，然后就可以将精力集中在HTML文件的布局和显示方面，并且以后修改数据时只需要修改XML文件即可，而不用去动HTML文件。XML的数据也可以作为一个数据块存储在HTML页面中。

交换数据。用XML可以在两个不兼容的系统间交换数据。通常，开发人员不得不花费大量的时间在两个不兼容的系统间交换数据，如果将数据转换为XML，则将大大降低数据交换的复杂性，并且不同类型的应用程序都可以读取它。

共享数据。因为XML是以简单的文本格式存储的，因此在共享数据方面，XML提供了一个独立于软硬件的方法。这使得创建被不同应用程序所使用的数据更容易，也使得系统的升级更容易。

存取数据。XML可用于将数据存储在文件或数据库中，应用程序能够存取和检索这些信息。一般的应用是显示这些信息。

2.3.4　XML的解析方式

XML最主要的解析方式有4种，分别是DOM、SAX、JDOM、DOM4J。

DOM是基于平台、语言无关的官方W3C标准。基于树的层次，其优点是可以移植，编程容易，开发者只需要调用建树的指令。其缺点是加载大文件不理想。

SAX是基于事件模型的，它在解析XML文档时可以触发一系列的事件，当发现给定的tag时，可以激活一个回调方法，告诉该方法制定的标签已经找到。类似于流媒体的解析方式，所以在加载大文件时效果不错。

JDOM是想成为Java特定文档的模型。它简化与XML的交互并且比使用DOM实现得更快。使用的是具体类不

使用接口，运用了大量的Collections类，方便开发者。

DOM4J是一个独立的开发结果，也是一个非常优秀的Java XML API，具有性能优异、功能强大和极端易使用的特点，同时它也是一个开放源代码的软件。

2.3.5　DOM和SAX解析XML详解

DOM是用与平台和语言无关的方式表示XML文档的官方W3C标准。DOM是以层次结构组织的节点或信息片段的集合。这个层次结构允许开发人员在树中寻找特定信息。分析该结构通常需要加载整个文档和构造层次结构，然后才能做任何工作（所以其劣势就是基于大文件的加载速度很慢，因为它是需要全部加载后才能操作）。

由于它是基于信息层次的，因而DOM被认为是基于树或基于对象的。DOM以及广义的基于树的处理具有几个优点。

（1）首先，由于树在内存中是持久的，因此可以修改它以便应用程序能对数据和结构作出更改。它还

可以在任意时间在树中上下导航，而不是像SAX是一次性的处理。DOM使用起来也要简单得多。

（2）另一方面，对于特别大的文档，解析和加载整个文档可能很慢且很耗资源，因此使用其他手段来处理这样的数据会更好。比如基于事件的模型，比如SAX。SAX这种处理的优点非常类似于流媒体的优点。分析能够立即开始，而不是等待所有的数据被处理。而且，由于应用程序只是在读取数据时检查数据，因此不需要将数据存储在内存中。这对于大型文档来说是个巨大的优点。事实上，应用程序甚至不必解析整个文档，它可以在某个条件得到满足时停止解析。一般来说，SAX比它的替代者DOM快很多。

2.3.6　JDOM和DOM4J解析XML详解

JDOM的目的是成为Java特定文档模型，它简化与XML的交互并且比使用DOM实现得更快。JDOM与DOM主要有两方面不同。

（1）首先，JDOM仅使用具体类而不使用接口。这在某些方面简化了API，但是也限制了灵活性。

（2）第二，API大量使用了Collections类，简化了那些已经熟悉这些类的Java开发者的使用。

JDOM文档声明其目的是“使用20%（或更少）的精力解决80%（或更多）Java/XML问题”（根据学习曲线假定为20%）。JDOM对于大多数Java/XML应用程序来说当然是有用的，并且大多数开发者发现API比DOM容易理解得多。JDOM还包括对程序行为的相当广泛检查以

防止用户做任何在XML中无意义的事。然而，仍需要充分理解XML以便做一些超出基本的工作（或者甚至理解某些情况下的错误）。这也许是比学习DOM或JDOM接口都更有意义的工作。

JDOM自身不包含解析器。它通常使用SAX2解析器来解析和验证输入XML文档（尽管它还可以将以前构造的DOM表示作为输入）。它包含一些转换器以将JDOM表示输出成SAX2事件流、DOM模型或XML文本文档。JDOM是在Apache许可证变体下发布的开放源码。

虽然DOM4J代表了完全独立的开发结果，但最初，它是JDOM的一种智能分支。它合并了许多超出基本XML文档表示的功能，包括集成的XPath支持、XMLSchema支持以及用于大文档或流化文档的基于事件的处理。它还提供了构建文档表示的选项，通过DOM4J API和标准DOM接口具有并行访问功能。

为支持这些功能，DOM4J使用接口和抽象基本类方法。DOM4J大量使用了API中的Collections类，但是在许多情况下，它还提供一些替代方法以允许更好的性能或更直接的编码方法。直接好处是，虽然DOM4J付出了更复杂的API的代价，但是它提供了比JDOM更大的灵活性。

在添加灵活性、XPath集成和对大文档处理的目标时，DOM4J的目标与JDOM是一样的：针对Java开发者的易用性和直观操作。它还致力于成为比JDOM更完整的解决方案，实现在本质上处理所有Java/XML问题的目标。在完成该目标时，它比JDOM更少强调防止不正确的应用程序行为。

DOM4J是一个很优秀的Java XML API，具有性能优异、功能强大和极端易使用的特点，同时它也是一个开放源代码的软件。如今可以看到越来越多的Java软

件都在使用DOM4J来读写XML，特别值得一提的是连Sun的JAXM也在用DOM4J, Hibernate也用DOM4J来读取XML配置文件。

第3章　Struts 2核心技术

Struts 2核心技术向我们展示了MVC设计模式中Control层如何通过开发手段来实现。本章通过实际开发项目中的一些简单示例向读者展现web.xml配置Struts 2和配置文件struts.xml、Action、ActionSupport校验等重要知识点，并指出在开发过程中需要注意的一些细节。

3.1　使用web.xml配置Struts 2实现Web项目Struts 2应用

在现在开发的Web项目中，都是使用web.xml来实现MVC框架的应用。既然Struts 2也属于MVC框架，因此在web.xml中必定要配置Struts 2才能实现应用。

技术要点

本节代码说明Struts 2的基本配置：

如何加载FilterDispatcher过滤器。

如何使用FilterDispatcher过滤器拦截URL。

实现代码

web. xml文件究竟是什么样子？它都包括哪些配置？下面给出了一个含有基本配置的web.xml文件。

```
    <! ……文件名：
web.xml……>
    <? xml version="1.0"encoding="GB2312"? >
    <web-app xmlns=http: //java.sun.com/xml/ns/j2ee
    xmlns:
xsi=http: //www.w3.org/2001/XMLSchemainstance
    version="2.4"
    xsi:
schemaLocation="http: //java.sun.com/xml/ns/j2ee
    http: //java.sun.com/xml/ns/j2ee/webapp__2__
4.xsd">
    <filter>
    <! 过滤器名字>
    <filtername>Struts 2</filtername>
```

```
    <! 过滤器支持的Struts 2类>
    <filterclass>org.apache.Struts
2.dispatcher.FilterDispatcher
    </filterclass>
    </filter>
    <filtermapping>
    <! 过滤器拦截名字>
    <filtername>Struts 2</filtername>
    <! 过滤器拦截文件路径名字>
    <urlpattern>/</urlpattern>
    </filtermapping>
    <welcomefilelist>
    <welcomefile>index.jsp</welcomefile>
    </welcomefilelist>
    </webapp>
```

源程序解读

（1）web. xml中对Struts1的加载都是加载一个Servlet，但是在Struts 2中，因为设计者为了实现AOP（面向方面编程）概念，都是用filter来实现的。所以web.xml里加载的都是Struts 2的FilterDispatcher类。<filtername>是定义的过滤器名字，而<class>就是Struts 2中的FilterDispatcher类。

（2）定义好过滤器，还需要在web.xml中指明该过滤器是如何拦截URL的。＜urlpattern＞＜/＜urlpattern＞中的“/*”是个通配符，它表明该过滤器是拦截所有的HTTP请求。基本不会改成其他形式，因为在开发中所有的HTTP请求都可能是在一个页面上进行业务逻辑处理的请求。就目前而言，开发人员只需要写成“/*”就可以了。

（3）本节中的示例代码是最基本的web.xml配置Struts 2的内容。其实还有＜initparam＞等设置过滤器初始化参数的配置内容。这里没有具体解释，是因为这些也可以在struts.properties文件内定义。

3.2 使用配置文件struts.xml实现页面导航定义

Struts 2中最核心的是Action，而Action的核心就是struts.xml，struts.xml集中了所有页面的导航定义。对于大型的Web项目，通过此配置文件即可迅速把握其脉络，不管是对前期的开发，还是后期的维护或升级都是大有裨益的。掌握struts.xml是掌握Struts 2的关键所在。

技术要点

本节代码向读者演示struts.xml内容的组成部分：

XML文件字符编码定义和DTD文件声明。

globalresults映射定义，如何进行全局导航页面。

package映射定义。包含的Action各属性介绍。

实现代码

Struts 2中的配置文件太多，前面通过web.xml了解了过滤器的配置。下面通过struts.xml了解页面导航的配置。

```
    <！……文件名：
struts.xml……>
    <? xml version="1.0"encoding="gb2312"? >
    <！DOCTYPE struts PUBLIC
    "//Apache Software Foundation//DTD Struts
Configuration 2.0//EN"
    "http: //struts.apache.org/dtds/struts2.0.dtd">
    <struts>
    <！Action所在包定义>
    <package name="C02"extends="strutsdefault">
    <！全局导航页面定义>
    <globalresults>
    <result name="global">/jsp/login.jsp</result
>
    </globalresults>
    <！Action名字，类以及导航页面定义>
```

```
    <! 通过Action类处理才导航的的Action定义>
    <action name="Login"
    class="com.example.struts.action.LoginAction">
    <result name="input">/jsp/login.jsp</result>
    <result name="success">/jsp/success.jsp
</result>
    </action>
    <! 直接导航的Action定义>
    <action name="index">
    <result>/jsp/login.jsp</result>
    </action>
    </package>
    </struts>
```

源程序解读

（1）struts. xml第一行是所有xml文件都具有的声明。通常以“<？”开始，以“？>”结束。version是必须指定的，该属性一般都为1.0，表明该文档遵守XML1.0规范。encoding是可选的，如果不写则默认UTF8，该文件代码中的gb2312表明该文件的编码集是gb2312，支持中文字符。常见的字符编码集有支持简体中文的gb2312，支持繁体中文的GBK，支持西欧字符的ISO88591以及通用的国际编码UTF8。

DTD文件必须被声明，它表明struts.xml是支持Struts 2的文档类型定义。DTD全称为Document Type Defination（文档类型定义）。

（2）struts. xml文件中所有的属性定义都是以“＜struts＞”开始，以“＜/struts＞”结束。主要属性有很多，这里先详细介绍package。

（3）package中定义了Action映射申明。它也可以包含很多＜action＞或者一个也不包含（当然实际开发中是不可能一个都不包含的）。其中name属性内容是开发的web项目名称，比如本章代码是CO2项目，所以代码中写的是CO2，而且它还扩展了Struts 2自带的默认文件strutsdefault.xml配置文件，在此基础上可以对Action或其他项目中需要用到的类映射进行自定义。

（4）Action是之前所述package中包含的Action映射申明。＜action＞中的name属性是在JSP页面上定义的Action名字。在Struts 2中系统主动寻找名字为它的Action，一旦找到就根据class属性中定义的Action类路径去执行该Action类。

在代码中可以看到Action名字为Login.action，系统搜索到它之后根据映射定义的class执行LoginAction类。result相当于在Struts1中的forward属性。因为Action对象都是配置对象，这些配置对象都有唯一的标识，其中name就是标识。通过检索这些标识，Action对象封装了需要指向的URL，系统就会将最后响应信息转到URL所指的JSP页面。也就是代码在＜result＞和＜/result＞中定义的JSP页面路径。

注意：Action的name一定要写成代码中显示的形式，没必要在后面加“.do”或者“.action”的后缀

名形式。因为“.do”是Struts1中定义的Action后缀名形式（当然在web.xml中也可以使用＜urlpattern＞*.do＜/urlpattern＞来定义或定义成其他后缀名形式），在Struts 2中已经废弃不用了。而不加“.action”是因为当系统运行时会自动搜寻后缀名为“.action”的Action，所以也没必要加。否则就变成搜寻“xx.action.action”这样格式的Action，会造成系统报错。

虽然Struts 2中系统只会搜寻“.action”的Action，但也可以让它只搜寻其他名字的后缀名。在Struts 2的org/apache/Struts 2目录下有个default.properties属性文件，其中有个属性名为struts.action.extension，可以将它改为“struts.action.extension=do”，这样就只搜寻“.do”的后缀名。

如果不想修改Struts 2的源文件，也可以用struts.properties，在该文件中加上“struts.action.extension=do”。而且还可以改为“struts.action.extension=do, htm”，这样就不仅只搜寻“.do”，还可以搜寻“.htm”后缀名。当中要以“，”隔开。

（5）代码中还示范了另外一种Action写法，这种Action不经过具体Action类进行业务逻辑处理，而是类似一个简单的Html链接功能，如代码所示，系统找到index.Action，根据＜result＞中定义的URL，在浏览器中直接显示login.jsp。

注意：＜result＞和＜/result＞之间定义的JSP页面要写出全路径，不能只写login.jsp、success.jsp。除非该JSP页面是在系统根目录下。

（6）＜globalresults＞是全局导航页面映射定义，这些定义的＜result＞被多个Action共用的。如果一个具体Action在＜action＞中找不到定义的＜result＞唯一标识，它就去寻找（也可称之为匹配）＜globalresults＞中的＜result＞唯一标识。如代码所示如果LoginAction返回的唯一标识不是“input”和“success”而是“global”，那它在浏览器中显示的名字为“global”的＜result＞指向的JSP，这里只是为了示范，所以还是指向login.jsp。可以新建一个error.jsp，让＜globalresults＞中名字为“global”的＜result＞指向它，则页面显示的就是error.jsp的内容。

3.3　使用Action类控制导航业务数据

Struts 2中Action充当着一个关键的角色。它解决了如何把JSP页面上的数据，根据实际开发项目中具体的业务逻辑，来进行处理的问题。

技术要点

本节代码使用登录功能作为例子，详细解析LoginAction类。

如何使用Execute方法处理业务逻辑。

如何处理Form表单数据。

HTTP的session对象在Action中的主要使用方式。

ActionMapping对象配置文件中的处理流程。

实现代码

LoginAction是一个负责登录的类，主要处理登录过程中用户属性的处理，如获取用户名和密码等。下面就是一个具体的，可以参与实际操作的LoginAction登录类。

```
    <！……文件名：
LoginAction.java……>
    public class LoginAction {
    //Action类公用私有变量，用来做页面导航标志
    private static String FORWARD=null;
    private String username; //用户名属性
    private String password; //密码属性
    public String getUsername（） {//取得用户名值
    return username;
    }
    public void setUsername（String username） {//设
置用户名值
    this.username=username;
    }
    public String getPassword（） {//取得密码值
    return password;
    }
    public void setPassword（String password） {//设
置密码值
    this.password=password;
    }
    public String execute（）throws Exception {
```

```
    username=getUsername（）；//属性值即JSP页面上输入的
值
    password=getPassword（）；//属性值即JSP页面上输入的
值
    try {
    //判断输入值是否是空对象或没有输入
    if（username! =null&&! username.equals（""）&&
password! =null
    &&! password.equals（""））{
    ActionContext.getContext（）.getSession（）.put（
"user", getUsername（））；
    FORWARD="success"; //根据标志内容导航到操作成功页面
    } else {
    FORWARD="input"; //根据标志内容导航到操作失败页面
    }
    } catch（Exception ex）{
    ex.printStackTrace（）；
    }
    return FORWARD;
    }
    }
```

Action类映射配置通过struts.xml完成，代码如下所示。

```
    <！……文件名：
struts.xml……>
    <action
name="Login"class="com.example.struts.action.Login
Action">
    <result name="input">/jsp/login.jsp</result>
    <result name="success">/jsp/success.jsp
</result>
```

```
</action>
```

源程序解读

（1）Struts 2中每一个具体的Action类其实都是将Struts1中的FormBean类和Action类代码放在一起，所以看起来就是一个简单的JavaBean类（按照现在流行说法是POJO, PlainsOrdinary Java Object。无格式普通Java对象）。比如代码中用户名和密码两个变量都是字符串类型变量，使用get和set方法可以从JSP页面上得到输入的值内容。

（2）Action类中最主要的方法为execute方法，Struts1中共有四个参数。返回一个ActionForward对象。而在Struts 2中为了不侵入Servlet的类和方法，同时为了更好地解耦，符合现在Web项目松耦合开发理念，所以这四个参数都已经不用了。Struts 2中返回

的也是一个普通字符串，此字符串内容就是之前所述的导航页面的唯一标识。

（3）代码中用户名和密码两个变量通过get方法得到JSP页面上输入的值内容，然后判断这两个变量是否为空对象或者字符串内容为“”，即没有输入任何数据。如果为空或者没有输入任何数据则mapForward赋值为“input”，这样就导航到错误页面。反之，则mapForward赋值为“success”，导航到成功页面。

细心的读者可以发现LoginAction类中给它赋的success、input两个值在struts.xml中＜action＞元素的＜result＞属性中都有定义。这就是之前所说的唯一标识指向URL中的JSP页面。

注意：之所以操作失败后mapForward赋值为input，而不是error或failure是为了Struts 2校验使用考虑，具体原因见下节。

（4）用户名变量的值在实际项目中有可能会被用到，因此把它放入session属性中。

注意：该session和Hibernate中的session是两回事情，它是HTTP请求中的session对象，getAttribute方法和setAttribute方法是开发中经常用到的。它的应用范围是整个当前HTTP请求，所以当用setAttribute方法将对象值放入后，可以在任何业务逻辑需要使用对象值时用getAttribute方法取出。至于Hibernate中的session将在后面章节中另外具体讲述。

在代码中Struts 2把session也封装起来。通过Struts 2包中自带的ActionContext类来调用，首先先得到当前HTTP应用中的内容，然后通过getSession方法得到Sessions对象，但Struts 2是用一个map对象来标识，也就是说getSession方法得到的是一个

Sessions对象封装处理后的结果。所以代码中可以使用put等map的方法，而不是setAttribute方法来将用户名变量值放入session。

（5）Action类的每一个实例都是和struts.xml中每一个＜action＞元素对应。这些struts.xml中的配置信息其实都是在系统开始运行时读入内存，以供系统运行时使用。比如此代码中通过name中的“Login”提交请求信息，control层将信息传递给LoginAction处理，LoginAction实例的execute方法被调用，将所对应的Form数据传入LoginAction，然后进行相应业务逻辑处理。

3.4 使用ActionSupport进行校验

如果From数据操作有误，比如输入的不是所需要的数据，或者没有输入等原因，Action实例执行execute方法前会使用校验来进行控制。本节就介绍如何在Struts 2中实现校验功能。

技术要点

本节代码中还是使用登录功能作为示例：

Action类中ActionSupport使用，以及validate方法重写实现。

属性文件messageResource.properties定义，以及JSP页面上的错误信息如何显示。

导航结果页面演示。

实现代码

前面学习了创建一个LoginAction登录类，此类用来处理登录过程中的一些逻辑操作。本例依然把要进行校验的方法，添加在登录类中。

```
    <！……文件名：
LoginAction.java……>
    public class LoginAction extends
ActionSupport {
    //校验方法，用来校验输入值为空或没有输入返回错误信息
    public void validate（） {
    if（getUsername（）==null｜｜
getUsername（）.trim（）.equals（""））{
    //返回错误信息键值，user.required包含具体内容见
messageResource.properties
    addFieldError（"username",
getText（"user.required"））；
    }
    if（getPassword（）==null｜｜
getPassword（）.trim（）.equals（""））{
    //返回错误信息键值，pass.required包含具体内容见
messageResource.properties
    addFieldError（"password",
getText（"pass.required"））；
    }
    }
    }
```

---

struts. properties定义显示信息文件名，代码如下所示。

```
    <!……文件名：struts.properties……>
    #支持本地化的资源文件名定义
    struts.custom.i18n.resources=messageResource
```

messageResource. properties定义出错信息，代码如下所示。

```
    <!……文件名：messageResource.properties……>
    #用key=value格式定义页面上显示的内容
    user.required=请输入用户名!
    pass.required=请输入密码!
```

下面通过登录页面login.jsp，来演示本例的验证操作。页面的详细代码如下所示。

```
    <!……文件名：login.jsp……>
    <%@page language="java"pageEncoding="gb2312"%>
    <! DOCTYPE HTML PUBLIC"//W3C//DTD HTML 4.01 Transitional//EN">
```

```
<! Struts 2标签库调用声明>
<%@taglib prefix="s"uri="/strutstags"%>
<html>
<head>
<title>登录页面</title>
</head>
<body>
<! form标签库定义，以及调用哪个Action声明>
<s: form action="Login">
<table width="60%"height="76"border="0">
<! 各标签定义>
<s: textfield name="username"label="用户名"/>
<s: password name="password"label="密码"/>
<s: submit value="登录"align="center"/>
</table>
</s: form>
</body>
</html>
```

登录页面如图3.1所示。不输入直接登录显示出错页面如图3.2所示。输入数据页面如图3.3所示。登录成功页面如图3.4所示。登录成功显示的页面是success.jsp，代码如下。

```
<! ……文件名：success.jsp……>
<%@page language="java"contentType="text/html;charset=GB2312"%>
<! DOCTYPE HTML PUBLIC"//W3C//DTD HTML 4.01 Transitional//EN">
```

```
<html>
<head>
<title>登录成功</title>
</head>
<body>
<! 取得session中的用户名值>
{sessionScope.user}，欢迎您～
</body>
</html>
```

图　3.1　登录初始页面

图　3.2　错误提示页面

图　3.3　输入数据页面

图　3.4　登录成功页面

源程序解读

（1）LoginAction类中继承ActionSupport类，此类也是Struts 2自带的类之一。该类有一个validate方法，所以继承ActionSupport类的Action子类都可以

通过重写此方法来定义自己开发的项目操作失败后的错误提示信息。

注意：Struts 2类库中有个BaseAction类，该类也是继承ActionSupport类的。因此也可以在具体Action类代码中直接继承BaseAction类。比如在LoginAction代码粗体所示处，将“ActionSupport”改为“BaseAction”，同样可以重写validate方法，不会产生任何编译错误。

如代码所示出错信息，其中addFieldError方法和getText方法也是ActionSupport类的方法，只是Action子类是直接调用没有重写它们。addFieldError方法顾名思义就是对JSP页面上Form中某个需要校验的field，操作有错误时如何进行错误信息处理的方法。比如“username”就是JSP页面上这个字段的名字，而“user.required”是错误信息属性文件中定义的。使

用getText方法可以得到属性文件中“user.required”对应的具体错误信息。

注意：validate方法一般都是在Action类执行execute方法之前执行，如果操作失败，就直接返回到struts.xml中定义的input指向的URL。而input指向的JSP页面也正是登录初始页面，如图3.2所示，在登录初始页面显示了错误提示信息。

（2）在Struts 2中还有个很重要的概念就是属性文件，一般有个名字为struts.properties的属性文件是Struts 2中全局属性配置的文件。如代码中看到的，对于本地化的处理，直接使用该文件中定义的messageResource.properties文件。

（3）messageResource. propertie文件定义的两个属性就是在Action类中显示的user.required和

pass.required，采用key=value格式，定义了具体错误信息内容。

如果result值为success则转到success.jsp。在该示例中这个页面代码显示了之前放置在HTTP的session中的用户名值，如图3.4所示。

如果result值为input则转到login.jsp。在该示例中，login.jsp将之前所述的错误提示信息值在JSP页面上显示，如图3.2所示。

（4）login. jsp中用Struts 2的标签库显示Form和它里面的各字段定义。具体标签使用方式可见后面章节内容，在此就不详述了。唯一值得的是只有在JSP页面中使用Form标签，本节叙述的校验功能才会起作用。因为Form标签本身已具备显示校验错误的能力。

第4章　Struts 2的另一核心技术——拦截器

Struts 2另一核心技术是拦截器，英文名为Interceptor。它原来是WebWork框架中一个很好的支持国际化、校验、类型转换的工具。现在WebWork和Struts合并成Struts 2之后，理所当然也成为了Struts 2的一部分。

拦截器本身是一个普通的Java对象，它的功能是动态拦截Action调用，在Action执行前后执行拦截器本身提供的各种各样的Web项目需求。当然也可以阻止Action的执行，同时也可以提取Action中可以复用的部分。

在Struts 2中还有个拦截器栈的概念，其实它就是拦截器的一个集合。它把多个拦截器集合起来，按

照在栈中配置的顺序执行，特别是针对Action可以拦截相应的方法或者字段。本章对其进行代码级别的示范，让读者对拦截器概念有进一步的认识。

4.1　拦截器在Struts 2中的默认应用

通过前几章介绍读者应该明白在Web项目中，客户需先在视图界面提交一个HTTP请求，在Struts 2的ServletDispatcher接收请求时，Struts 2会查找配置文件，如struts.xml文件。根据xml文件中定义的拦截器配置，去调用拦截器。如果配置了拦截器栈，则根据拦截器在拦截器栈中的前后顺序，一一进行调用。而Struts 2自带的源代码中也提供了默认的拦截器配置。

在第1章中介绍过Struts 2的各个文件夹内容，也说过在src文件夹中包含了Struts 2的所有底层实现源

代码，读者可到自己安装Struts 2的文件路径下找到src\core\src\main\

resources\，其中有个名为strutsdefault.xml的文件，它是Struts 2自定义的配置文件，其中有关拦截器的配置代码，正是本节需要介绍的拦截器在Struts 2中的默认应用。

技术要点

本节代码说明Struts 2中的拦截器默认配置，及其各拦截器的功能。

拦截器和拦截器栈的定义配置格式。

Struts 2定义的各个拦截器的功能介绍。

实现代码

strutsdefault.xml是安装Struts 2后自带的配置文件，它配置一些拦截器栈的默认应用。

```
    <！……文件名：
strutsdefault.xml……>
    <interceptors>
    <interceptor name="alias"
    class="com.opensymphony.xwork2.interceptor.Alia
sInterceptor"/>
    ……
    <interceptor
name="roles"class="org.apache.Struts
2.interceptor.RolesInterceptor"/>
    <！基础栈>
    <interceptorstack name="basicStack">
    <interceptorref name="exception"/>
    <interceptorref name="servletConfig"/>
    <interceptorref name="prepare"/>
    <interceptorref name="checkbox"/>
    <interceptorref name="params"/>
    <interceptorref name="conversionError"/>
    </interceptorstack>
    <！校验和工作流栈>
    <interceptorstack
name="validationWorkflowStack">
    <interceptorref name="basicStack"/>
    <interceptorref name="validation"/>
    <interceptorref name="workflow"/>
    </interceptorstack>
    <！文件上传栈>
    <interceptorstack name="fileUploadStack">
    <interceptorref name="fileUpload"/>
    <interceptorref name="basicStack"/>
    </interceptorstack>
```

```
<! 模型驱动栈>
<interceptorstack name="modelDrivenStack">
<interceptorref name="modelDriven"/>
<interceptorref name="basicStack"/>
</interceptorstack>
<! action链栈>
<interceptorstack name="chainStack">
<interceptorref name="chain"/>
<interceptorref name="basicStack"/>
</interceptorstack>
<! i18n国际化栈>
<interceptorstack name="i18nStack">
<interceptorref name="i18n"/>
<interceptorref name="basicStack"/>
</interceptorstack>
</interceptors>
```

源程序解读

（1）在xml配置文件中配置拦截器和拦截器栈都是以“<interceptors>”开头，以“</interceptors>”结尾。

（2）配置拦截器的格式如上面代码所示以“<interceptor/>”格式显示，其中两个属性name是拦

截器名字，另一个是对应的类路径，因为之前已经说过拦截器也是一个普通的Java对象。

（3）拦截器栈的格式是以“＜interceptorstack＞”开头，以“＜/interceptorstack＞”结尾。其中属性name是拦截器栈名字。在“＜interceptorstack＞”和“＜/interceptorstack＞”之间可以设置拦截器。如代码所示格式为“＜interceptorref/＞”，其中name属性也是拦截器名字。如果系统运行拦截器栈，都是按照拦截器栈中定义的拦截器先后顺序执行拦截器。请读者仔细查看基础栈配置，配置的拦截器都是在xml文件中定义的拦截器。

注意：拦截器栈中不单单可以配置拦截器，它甚至还可以配置拦截器栈。比如在validationWorkflowStack拦截器栈中就配置了

basicStack拦截器栈配置的子拦截器栈中的拦截器也会被执行，这类似于父集合和子集合的概念。

（4）针对strutsdefault.xml文件中各个拦截器配置进行介绍，因为如果使用Struts 2在Web项目开发中，这些拦截器都是默认缺省的，会被执行的。因此了解Struts 2底层的拦截器能实现什么功能对开发人员来说是很有帮助的。

alias：对于HTTP请求包含的参数设置别名。

autowiring：将某些JavaBean实例自动绑定到其他Bean对应的属性中。有点类似Spring的自动绑定，在Spring部分会详细说明。

Chain：在Web项目开发中，以前使用Struts开发时经常碰到两个Action互相传递参数或属性的情况。

该拦截器就是让前一个Action的参数可以在现有Action中使用。

conversionError：从ActionContext中将转化类型时发生的错误添加到Action的值域错误中，在校验时经常被使用来显示类型转化错误的信息。

cookie：从Struts 2.0.7版本开始，把cookie注入Action中可设置的名字或值中。

createSession：自动创建一个HTTP的Session，尤其是对需要HTTP的Session的拦截器特别有用。比如下面介绍的TokenInterceptor。

debugging：用来对在视图间传递的数据进行调试。

ExecAndWait：不显式执行Action，在视图上显示给用户的是一个正在等待的页面，但是Action其实正

在“背后”执行着。该拦截器尤其对进度条进行开发的时特别有用。

exception：将异常和Action返回的result相映射。

fileUpload：支持文件上传功能的拦截器。

i18n：支持国际化的拦截器。

logger：拥有日志功能的拦截器。

modelDriven：Action执行该拦截器时，可以将getModel方法得到的result值放入值栈中。

scopedModelDriven：执行该拦截器时，可以从一个scope范围检索和存储model值，通过调用setModel方法去设置model值。

params：将HTTP请求中包含的参数值设置到Action中。

prepare：假如Action继承了Preparable接口，则会调用prepare方法。

staticParams：对于在struts.xml文件中Action中设置的参数设置到对应的Action中。

scope：在session或者application范围中设置Action的状态。

servletConfig：该拦截器提供访问包含HttpServletResquest和HttpServletResponse对象的Map的方法。

timer：输出Action的执行时间。

token：避免重复提交的校验拦截器。

tokenSession：和token拦截器类似，但它还能存储提交的数据到session中。

validation：运行在actionvalidation.xml（校验章节将介绍）文件中定义的校验规则。

workflow：在Action中调用validate校验方法。如果Action有错误则返回到input视图。

store：执行校验功能时，该拦截器提供存储和检索Action的所有错误和正确信息的功能。

checkbox：视图中如果有checkbox存在的情况，该拦截器自动将unchecked的checkbox当作一个参数（通常值为“false”）记录下来。这样可以用一个隐藏的表单值来记录所有未提交的checkbox，而且缺省unchecked的checkbox值是布尔类型的，如果视图中

checkbox的值设置的不是布尔类型，它就会被覆盖成布尔类型的值。

profiling：通过参数来激活或不激活分析检测功能，前提是Web项目是在开发模式下（涉及调试和性能检验时使用）。

roles：进行权限配置的拦截器，如果登录用户拥有相应权限才去执行某一特定Action。

4.2　拦截器原理实现

在了解Struts 2中拦截器的使用方式之前，需要先向读者展示拦截器的底层实现原理。之前也说了拦截器是一个普通的Java对象，而被拦截的正常执行业务逻辑功能的类也是一个普通的Java对象，如何使这两个对象进行关联，这两个对象执行的先后顺序是怎么样的，本节进行详细介绍。

技术要点

本节代码向读者演示拦截器如何实现的，介绍的知识点如下：

拦截器类和被拦截类内容。

运用反射机制调用类和类方法。

设置拦截器处理类，配置拦截器在何时执行以及拦截器类和被拦截类执行的先后顺序。

设置代理对象类实现拦截器拦截功能。

测试程序运行结果显示拦截功能正常执行情况。

实现代码

功能执行类：

```
    <！……文件名：
ExecuteFunction.java……>
    //执行功能类
    public class ExecuteFunction implements
ExecuteFunctionInterface {
    public void execute（） {//执行功能类执行方法
    System.out.println（"execute something……"）；
    }
    }
```

功能执行接口：

```
    ！……文件名：
ExecuteFunctionInterface.java……>
```

```
    public interface ExecuteFunctionInterface {//执
行功能接口
    public void execute（）;
    }
```

拦截器类：

```
    <!……文件名：
Interceptor.java……>
    //拦截器类
    public class Interceptor {
    public void beforeDoing（） {//拦截执行功能类之前执
行方法
    System.out.println（"before doing
Something……"）;
    }
    public void afterDoing（） {//拦截执行功能类之后执行
方法
    System.out.println（"after doing
Something……"）;
    }
    }
```

拦截器处理类：

```
    <!……文件名：
InterceptorHandler.java……>
    import java.lang.reflect.InvocationHandler;
    import java.lang.reflect.Method;
    //拦截器处理类
```

```
    public class InterceptorHandler implements
InvocationHandler {
    private Object object;
    private Interceptor inter=new Interceptor（）;
    public void setObject（Object object） {//设置需
要拦截的执行功能类
    this.object=object;
    }
    //接口invoke方法，proxy是代理实例，method是实例方法，
args是代理类传入的方法参数
    public Object invoke（Object proxy, Method
method, Object[]args）throws
    Throwable {
    inter.beforeDoing（）;
    //object提供该方法的类实例，args是调用方法所需的参数值
的数组
    Object returnObject=method.invoke（object,
args）;
    inter.afterDoing（）;
    return returnObject;
    }
    }
```

代理对象类：

```
    <！……文件名：ProxyObject.java……
>
    import java.lang.reflect.Proxy;
    public class ProxyObject {
    private InterceptorHandler handler=new
InterceptorHandler（）;
    public Object getProxy（Object object） {//得到执
行类的代理对象实例
    handler.setObject（object）;
```

```
    //创建对象实例
    return Proxy.newProxyInstance（
    ExecuteFunction.class.getClassLoader（），
    object.getClass（）.getInterfaces（），
    handler）；
    }
    }
```

测试程序：

```
    <！……文件名：
TestInterceptor.java……>
    //测试执行类和拦截器类是否执行
    public class TestInterceptor {
    public static void main（String[]args） {
    ExecuteFunctionInterface test=new
ExecuteFunction（）；
    //得到执行类的一个代理对象实例
    ExecuteFunctionInterface
resultObject=（ExecuteFunctionInterface）
    new ProxyObject（）
    .getProxy（test）；
    resultObject.execute（）；//代理对象执行
    }
    }
```

在TestInterceptor.java文件上单击鼠标右键，然后单击Run As｜Java Aplication命令。在

MyEclipse控制台中显示代码中定义的打印方法。测试程序运行效果如图4.1所示。

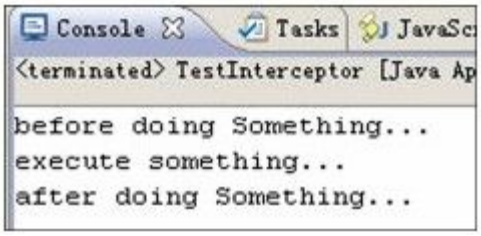

图　4.1　拦截器运行示例效果图

源程序解读

（1）ExecuteFunction是一个正常执行业务逻辑类的Java类，它是继承接口ExecuteFunctionInterface，其中的execute方法，作为示例，笔者只是调用打印方法打印了“execute something……”这一行字。还定义了Interceptor拦截器类，该类中有两个方法也是为了示例，这两个方法都是简单打印了一行字。

如图4.1所示，如果其中beforeDoing方法在ExecuteFunction类的execute方法打印“execute something……”之前打印“before doing Something……”，afterDoing方法在其后打印“after doing Something”就达到了拦截器在功能执行类前后执行拦截的目的。

（2）为了让拦截器类和被拦截的功能执行类发生关联关系，使用Java中的反射机制。在InterceptorHandler类中，通过扩展java.lang.reflect.InvocationHandler接口，覆盖重写invoke方法，该方法用method.invoke来调用再通过setObject方法传入的功能执行类对象的方法。比如这里通过setObject方法传入的是ExecuteFunction对象，该对象中包含一个execute方法，而且是无参的，因此method.invoke调用的就是execute方法，只是执行并打印出“execute something……”，同时将已经

被设置为私有类变量的拦截器类中的两个方法在其前后执行。

这样invoke方法已经将拦截器类中的两个方法在功能执行类的方法执行前后执行了，现在要做的只是如何让该类中的invoke方法被测试程序调用。

（3）创建ProxyObject对象是想通过使用设计模式中的代理模式（由于本书不是专门介绍设计模式，请读者翻阅设计模式有关资料）来生成功能执行类的一个代理对象实例。通过newProxyInstance方法调用InterceptorHandler类。说的再明白点就是如果这个代理对象也执行功能执行类的execute方法时，newProxyInstance方法的作用是把execute方法指派给InterceptorHandler类执行，通过反射，InterceptorHandler类执行execute方法是在invoke方法中执行。

因此在method.invoke方法前后的拦截器类的beforeDoing和afterDoing方法也会执行。从而就会按照图4.1所示的顺序打印三个方法的显示内容。

注意：代理模式的定义是包装一个对象，并控制它的访问。在这里就是包装了ExecuteFunction对象，并控制它的execute方法，让它做了额外的事情就是执行拦截器类的beforeDoing和afterDoing方法。

（4）测试程序通过新建一个代理对象类，并调用getProxy方法得到ExecuteFunction的一个代理对象实例，然后在这个代理对象执行execute方法时，因为在之前已经说明其实创建代理对象实例时，已经是调用了InterceptorHandler类的invoke方法，就实现了动态代理机制（所谓动态代理就是指代理对象是代码执行时创建的）。

仔细看测试程序就知道在getProxy方法执行前，只是新建了ExecuteFunctionInterface接口对象，并没有创建代理对象，这里创建代理对象的目的就是让execute方法执行拦截器类的方法，让拦截器类的方法也被执行，并且执行顺序也是根据InterceptorHandler类的invoke方法中定义的顺序执行。拦截器相当于在功能执行类前后都拦截了它，并执行了自己的方法。

4.3　在Struts 2中配置自定义的拦截器

Struts 2中的拦截器配置一般都是在struts.xml配置文件中。这里编写了三个拦截器类，通过在struts.xml配置文件中定义并查看运行效果，介绍自定义拦截器的执行顺序和配置文件中遵行的配置原理。

4.3.1　扩展拦截器接口的自定义拦截器配置

技术要点

本节代码介绍拦截器基础配置以及设置参数功能。

配置文件struts.xml中如何定义拦截器。

Action配置中拦截器参数定义和注意点。

拦截器参数的设置和配置修改过程。

实现代码

自定义拦截器类，主要为了测试拦截器的执行顺序。

```
    <！……文件名：
ExampleInterceptor.java……>
    import
com.opensymphony.xwork2.ActionInvocation;
    import
com.opensymphony.xwork2.interceptor.Interceptor;
    public class ExampleInterceptor implements
Interceptor {
    //设置新参数
    private String newParam;
    public String getNewParam（） {
    return newParam;
    }
    public void setNewParam（String newParam） {
    this.newParam=newParam;
    }
    public void destroy（） {
```

```
    System.out.println（"end doing……"）;
    }
    //拦截器初始方法
    public void init（） {
    System.out.println（"start doing……"）;
    System.out.println（"newParam is: "+newParam）;
    }
    //拦截器拦截方法
    public String intercept（ActionInvocation arg0）
throws Exception {
    System.out.println（"start invoking……"）;
    String result=arg0.invoke（）;
    System.out.println（"end invoking……"）;
    return result;
    }
    }
```

拦截器映射配置struts.xml文件。

```
    <！……文件名：
struts.xml……>
    <！拦截器配置定义>
    <interceptors>
    <interceptor name="example"
    class="com.example.struts.interceptor.ExampleIn
terceptor">
    <！参数设置>
    <param name="newParam">test</param>
    </interceptor>
    </interceptors>
    <！Action名字，类以及导航页面定义>
    <！通过Action类处理才导航的Action定义>
    <action name="Login"
    class="com.example.struts.action.LoginAction">
```

```
    <result name="input">/jsp/login.jsp</result>
    <result name="success">/jsp/success.jsp
</result>
    <! Action中拦截器定义>
    <interceptorref name="example">
    <! 改变拦截器参数值>
    <param name="newParam">example</param>
    </interceptorref>
    </action>
```

执行效果如图4.2所示。参数值显示如图4.3所示。

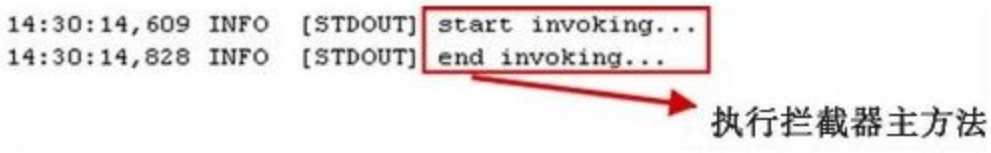

图 4.2 执行拦截器后的效果

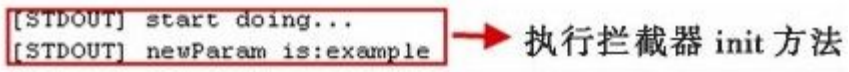

图 4.3 newParam参数值显示图

源程序解读

（1）先看struts.xml文件，在文件开始以<interceptors>开头，以</interceptors>结尾的形

式定义了拦截器，该拦截器命名为example，映射的类文件路径写在class属性中。在＜Action＞中＜result＞标签后，可以定义只在该Action执行时会拦截的拦截器定义，其实就是调用在＜Action＞前定义的example拦截器，并且还可以＜param＞标签定义拦截器属性。

（2）＜param＞标签定义拦截器属性，或者称为参数。name是参数名，而在＜param＞＜/param＞间的内容就是该定义的参数值。

注意：如果在＜Action＞中和＜Action＞外都定义＜param＞的值，比如在本实例中＜Action＞中newParam值为“example”，＜Action＞外newParam值为“test”。而在运行时显示的还是＜Action＞中的参数值，即“example”，如图4.3所示。可以理解为屏蔽了＜Action＞外参数值，因此如果定义多个

Action，每个Action都调用example拦截器，则都可以自定义自己的newParam的值。如果不定义，显示的就是“test”，否则就是各自定义的newParam值。

（3）再来看看ExampleInterceptor类代码，newParam是它的一个私有变量属性，有自己的setter、getter方法。而且它扩展了Interceptor接口，该接口是Struts 2的类库中自带的接口类。重写interceptor方法，用invoke方法执行Action，在Action前后执行两个打印方法。

启动服务器后，在网页上显示登录页面，单击“登录”按钮，然后在MyEclipse的控制台下就可以看见如图4.2所示的效果图。如果读者能看见这两行字被打印出来，就说明example拦截器拦截Login Action成功。

4.3.2　继承抽象拦截器的自定义拦截器配置

技术要点

本节代码介绍抽象拦截器配置并对默认拦截器栈做简单介绍。

继承抽象拦截器类的自定义拦截器类编写方式。

配置文件struts.xml中如何定义默认拦截器。

实现代码

重写上一小节的自定义拦截器，主要是抽象拦截器的拦截方法。

<！……文件名：ExampleInterceptor.java……>

```
    import
com.opensymphony.xwork2.ActionInvocation;
    import
com.opensymphony.xwork2.interceptor.AbstractInterc
eptor;
    public class ExampleInterceptor extends
AbstractInterceptor {
    //重写抽象拦截器的拦截方法
    @Override
    public String intercept（ActionInvocation arg0）
throws Exception {
    System.out.println（"start invoking2……"）;
    String result=arg0.invoke（）;
    System.out.println（"end invoking2……"）;
    return result;
    }
    }
```

拦截器映射配置struts.xml文件的设置。

```
    <！……文件名：
struts.xml……>
    <struts>
    <！Action所在包定义>
    <package
name="C04.3.2"extends="strutsdefault">
    <！拦截器配置定义>
    <interceptors>
    <interceptor name="example"
    class="com.example.struts.interceptor.ExampleIn
terceptor">
    </interceptor>
    <！
    拦截器栈配置定义
```

```
    <interceptorstack name="exampleStack">
    <interceptorref name="example">
</interceptorref>
    <interceptorref name="defaultStack">
</interceptorref>
    </interceptorstack>
    >
    </interceptors>
    <action name="Login"
    class="com.example.struts.action.LoginAction">
    <result name="input">/jsp/login.jsp</result>
    <result name="success">/jsp/success.jsp
</result>
    <! Action拦截器配置定义>
    <interceptorref name="example">
</interceptorref>
    <! Action拦截器栈配置定义>
    <interceptorref name="defaultStack">
</interceptorref>
    </action>
    </package>
    </struts>
```

拦截器执行效果如图4.4所示。拦截器栈执行效果如图4.5所示。

图　4.4　执行拦截器后的效果

图　4.5　缺省拦截器栈中包含的校验拦截器执行

源程序解读

（1）ExampleInterceptor类中，继承AbstractInterceptor抽象类。读者可以查看Struts 2的源代码，在AbstractInterceptor中只有intercept这一个抽象方法。因此自定义的ExampleInterceptor中只需要对这个方法进行重写。重写内容和4.3.1节类似。

（2）struts. xml配置文件中，在＜Action＞前还是定义名为“example”的拦截器。在＜Action＞

中，配置“example”拦截器。

注意：在＜Action＞中还配置了“defaultStack”拦截器栈，这是因为如果在＜Action＞中不配置该拦截器栈，则Login.action运行时只会执行配置的“example”拦截器，不会执行“defaultStack”拦截器栈。而且“defaultStack”是Struts 2配置的缺省拦截器栈，在4.1节中的strutsdefault.xml中定义的拦截器都是由它来调用执行。

Struts 2规定如果在＜Action＞中，开发人员配置了自己定义的拦截器或拦截器栈，不显示在struts.xml配置文件中配置“defaultStack”拦截器栈，则所有strutsdefault.xml中定义的拦截器都不会执行即不执行“defaultStack”拦截器栈。当然如果在＜Action＞中开发人员没有配置自己定义的拦截器

或拦截器栈，就是不显示配置“defaultStack”拦截器栈，则“defaultStack”拦截器栈也是会执行的。

（3）为了让Action被执行时，“defaultStack”拦截器栈和“example”的拦截器都执行，一种办法是如上代码所示。另一种办法也可以如struts.xml配置文件中被注释的那段自定义的拦截器栈配置。在4.1节中也说过拦截器栈中可以配置拦截器栈，因此在注释中“defaultStack”拦截器栈可以作为配置的“exampleStack”拦截器栈的子元素。在＜Action＞中配置代码可以写成如下代码中黑体所示：

```
    <! ……文件名:
struts.xml……>
    <action name="Login"
    class="com.example.struts.action.LoginAction">
    <result name="input">/jsp/login.jsp</result>
    <result name="success">/jsp/success.jsp
</result>
    <! Action拦截器栈配置定义>
    <interceptorref name="exampleStack">
</interceptorref>
    </action>
```

这样的代码形式也能保证“defaultStack”拦截器栈和“example”的拦截器都执行。例如在登录页面不输入任何登录信息，直接单击“登录”按钮。在MyEclipse的控制台下，执行结果如图4.4所示。而在页面中的显示如图4.5所示，“defaultStack”拦截器栈中包含的输入校验拦截器执行后，显示拦截后的信息。这两张图就充分证明了“defaultStack”拦截器栈和“example”的拦截器都已经执行。

4.3.3　继承方法拦截器的自定义拦截器配置

技术要点

本节代码介绍方法拦截器配置，并对缺省拦截器栈对整个Web项目的Action影响进行介绍。

继承方法拦截器类的自定义拦截器类编写方式。

配置文件struts.xml中如何定义方法拦截器和其属性。

对所有Action配置拦截器和拦截器栈。

实现代码

还是前面定义的那个自定义拦截器类，本节学习重写方法拦截器的拦截方法。

```
    <！……文件名：
ExampleInterceptor.java……>
    import
com.opensymphony.xwork2.ActionInvocation;
    import
com.opensymphony.xwork2.interceptor.MethodFilterIn
terceptor;
    public class ExampleInterceptor extends
MethodFilterInterceptor {
    //重写方法拦截器拦截方法
    @Override
    protected String doIntercept（ActionInvocation
arg0）throws Exception {
    System.out.println（”start invoking3……”）;
    String result=arg0.invoke（）;
    System.out.println（”end invoking3……”）;
    return result;
    }
```

LoginAction中增加了method方法。

```
    <！……文件名：
LoginAction.java……>
    public String method（）throws Exception {
    FORWARD="success";
    return FORWARD;
    }
```

拦截器映射配置struts.xml文件的修改。

```
    <! ……文件名：
struts.xml……>
    <struts>
    <! Action所在包定义>
    <package name="C04.3"extends="strutsdefault">
    <! 拦截器配置定义>
    <interceptors>
    <interceptor name="example"
    class="com.example.struts.interceptor.ExampleIn
terceptor">
    </interceptor>
    </interceptors>
    <!
    缺省拦截器栈配置定义
    <defaultinterceptorref name="example">
</defaultinterceptorref>
    >
    <! Action名字，类以及导航页面定义>
    <! 通过Action类处理才导航的Action定义>
    <action name="Login"
    class="com.example.struts.action.LoginAction"me
thod="method">
    <result name="input">/jsp/login.jsp</result>
    <result name="success">/jsp/success.jsp
</result>
    <! Action方法拦截器配置定义>
    <interceptorref name="example">
    <! 被拦截方法配置定义>
    <param name="includeMethods">method</param>
    <! 不被拦截方法配置定义>
    <param name="excludeMethods">method, execute
</param>
    </interceptorref>
    </action>
    </package>
    </struts>
```

“includeMethods”配置后的拦截器执行效果如图4.6所示。“includeMethods”和“excludeMethods”同时配置后的拦截器执行效果如图4.7所示。

图　4.6　执行方法拦截器后的效果

图　4.7　method方法还是被拦截器拦截

源程序解读

（1）ExampleInterceptor类中，继承MethodFilterInterceptor抽象类。读者也可以查看Struts 2的源代码，在MethodFilterInterceptor中也只有一个抽象方法，但该抽象方法名为“doIntercept”。也对这个方法进行重写。重写内容和4.3.1节类似。

（2）LoginAction. java中又定义了一个名为“method”方法，在struts.xml配置文件中，因为LoginAction中有execute方法，又有method方法，因此在＜Action＞中，请读者注意struts.xml中的黑体部分，该部分代码表示现在LoginAction只执行method方法，而execute方法不被执行。笔者在＜Action＞中增加了一个“method”属性，该属性中“=”后面的内容是Action中具体方法名，如果不写“method”属性，Action是缺省执行execute方法。如果写了“method”属性，Action就执行“=”后的具体方法，而不会执行execute方法。

“example”拦截器还是跟以前一样在＜Action＞前定义。在＜Action＞中配置“example”拦截器，笔者增加了“includeMethods”和“excludeMethods”两个param属性定义。“includeMethods”表示的是被拦截器拦截的方法。方法名写在＜param＞和＜/param

＞之间，如果有多个方法需要拦截器拦截，则方法名之间以“，”相隔。“excludeMethods”表示的是不被拦截器拦截的方法，如果有多个方法，也是以“，”相隔。

注意：如struts.xml配置文件中代码所示：

```
<param name="excludeMethods">method, execute</param>
```

这行代码被注释，则运行后在MyEclipse的控制台中看到的是如图4.6所示的运行后效果。这说明method方法被拦截。如果

```
<param name="includeMethods">method</param>
```

这行代码被注释，则MyEclipse的控制台中没有任何拦截器拦截信息显示。说明method没有被拦截器拦截即拦截器没有执行。

但是如果struts.xml配置文件中代码显示上述的两行代码都没有被注释，有时会不知道method方法到底是被拦截器拦截还是没被拦截。其实运行后的效果如图4.7所示，这说明method方法在两个属性中都被定义，Struts 2认为method方法还是被拦截的。

（3）请注意struts.xml配置文件中＜Action＞前被注释的＜defaultinterceptorref＞定义。该标签表示的是所有Action都会执行的拦截器或拦截器栈的定义。之前的代码中对于拦截器的定义是在＜Action＞前，拦截器的配置都是在＜Action＞中，比如“example”拦截器只有在LoginAction执行时才会去拦截。如果是配置＜defaultinterceptorref＞中，则不管是LoginAction还是其他struts.xml配置文件中定义的Action都会被“example”拦截。在＜defaultinterceptorref＞中，也可以配置拦截器栈。

如4.3.2小节中的“exampleStack”拦截器栈如果在＜defaultinterceptorref＞中配置，则所有Action执行时，“exampleStack”拦截器栈都会执行该栈中包含的拦截器。

注意：struts.xml配置文件中要么没有＜defaultinterceptorref＞定义，如果定义了也只能定义一次。该标签在struts.xml配置文件中只能写在＜Action＞前，而且只能写一次，不能重复定义。

4.4　Struts 2文件上传拦截器的应用

4.1　节中对所有Struts 2缺省定义的拦截器作了介绍，其中有个“fileUpload”拦截器，本节就针对该拦截器在开发中如何实现文件上传下载功能做一个简单说明。在正式说明之前，还需要在Web项目中导入支持文件上传下载和IO输入输出的两个jar包。它们的名字为commonsfileupload1.2.1.jar和commonsio1.4.jar，这两个包都可以在apache网站上下载，笔者下载的是它们的最新版本，请读者自行去下载。

4.4.1　Struts 2文件上传功能开发

技术要点

本节代码详细说明文件上传功能的开发流程。

文件上传页面和显示上传成功页面代码内容。

UploadAction类中实现上传功能方法和上传文件属性介绍。

struts.xml中UploadAction配置，以及字符编码、文件临时存放路径配置。

上传后所处路径和最终上传成功后的效果展示。

实现代码

上传文件页面，这里笔者定义的是多个文件上传。

```
    <!……文件名：
upload.jsp……>
    <%@taglib prefix="s"uri="/strutstags"%>
    <html>
    <head>
    <meta
httpequiv="ContentType"content="text/html;
charset=gb2312">
    <title>上传文件</title>
    </head>
```

```
    <body>
    <! 上传文件表单定义>
    <s: form
action="upload"method="post"enctype="multipart/form
data">
    <tr>
    <! 上传文件标签定义>
    <td>上传文件: <s: file name="file"></s: file>
</td>
    </tr>
    <tr>
    <td>再次上传文件: <s: file name="file"></s:
file></td>
    </tr>
    <tr>
    <td align="left"><s: submit
name="submit"value="提交"></s: submit></td>
    </tr>
    </s: form>
    </body>
    </html>
```

上传文件成功后结果页面的代码。

```
    <! ……文件名:
result.jsp……>
    <%@taglib prefix="s"uri="/strutstags"%>
    <html>
    <head>
    <meta
httpequiv="ContentType"content="text/html;
charset=gb2312">
    <title>上传结果</title>
    </head>
    <body>
```

```
    上传文件：
    <！显示上传成功文件名>
    <s: property value="fileFileName"/>
    </body>
    </html>
```

UploadAction类的代码。

```
    <！……文件名：
UploadAction.java……>
    import java.io.File;
    import java.io.FileInputStream;
    import java.io.FileNotFoundException;
    import java.io.FileOutputStream;
    import java.io.IOException;
    import java.io.InputStream;
    import java.io.OutputStream;
    import java.util.List;
    import org.apache.Struts
2.ServletActionContext;
    import com.opensymphony.xwork2.ActionSupport;
    //文件上传Action
    public class UploadAction extends
ActionSupport {
    private final static String
UPLOADDIR="/upload"; //上传文件存放路径
    private List<File>file; //上传文件集合
    private List<String>fileFileName; //上传文件名集
合
    private List<String>fileContentType; //上传文件
内容类型集合
    public List<File>getFile（） {
    return file;
    }
    public void setFile（List<File>file） {
```

```
    this.file=file;
    }
    public List＜String＞getFileFileName（） {
    return fileFileName;
    }
    public void setFileFileName（List＜String＞
fileFileName） {
    this.fileFileName=fileFileName;
    }
    public List＜String＞getFileContentType（） {
    return fileContentType;
    }
    public void setFileContentType（List＜String＞
fileContentType） {
    this.fileContentType=fileContentType;
    }
    public String execute（） throws Exception {
    for（int i=0; i＜file.size（）; i++） {
    uploadFile（i）; //循环上传每个文件
    }
    return"success";
    }
    //执行上传功能
    private void uploadFile（int i） throws
FileNotFoundException, IOException {
    try {
    InputStream in=new
FileInputStream（file.get（i））;
    String
dir=ServletActionContext.getRequest（） .getRealPath
（UPLOADDIR）;
    File uploadFile=new File（dir,
this.getFileFileName（） .get（i））;
    OutputStream out=new
FileOutputStream（uploadFile）;
    byte[]buffer=new byte[10241024];
    int length;
```

```
while（（length=in.read（buffer））＞0）｛
out.write（buffer，0，length）；
｝
in.close（）；
out.close（）；
｝catch（FileNotFoundException ex）｛
ex.printStackTrace（）；
｝catch（IOException ex）｛
ex.printStackTrace（）；
｝
｝
｝
```

---

struts. xml配置文件中有关文件上传的配置：

```
＜！……文件名：
struts.xml……＞
＜struts＞
＜！系统常量定义，定义上传文件字符集编码＞
＜constant
name="struts.i18n.encoding"value="gb2312"＞
＜/constant＞
＜！系统常量定义，定义上传文件临时存放路径＞
＜constant
name="struts.multipart.saveDir"value="c：＼"＞
＜/constant＞
＜！Action所在包定义＞
＜package name="C04.4"extends="strutsdefault"＞
＜！Action名字，类以及导航页面定义＞
＜！通过Action类处理才导航的的Action定义＞
＜action
name="upload"class="action.UploadAction"＞
＜result name="input"＞/jsp/upload.jsp＜/result＞
＜result name="success"＞/jsp/result.jsp＜/result
＞
```

```
</action>
</package>
</struts>
```

文件上传页面如图4.8所示。选择文件如图4.9所示。单击“提交”按钮后文件上传成功页面，并显示上传文件名，如图4.10所示。两个被上传的文件最终在服务器上存放路径，如图4.11所示。

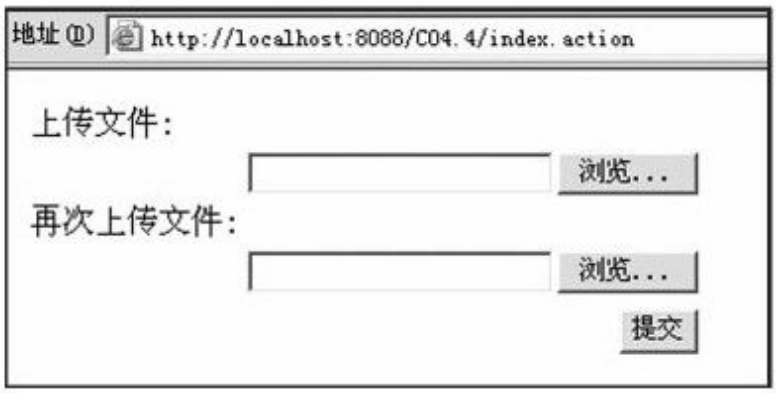

图　4.8　文件上传

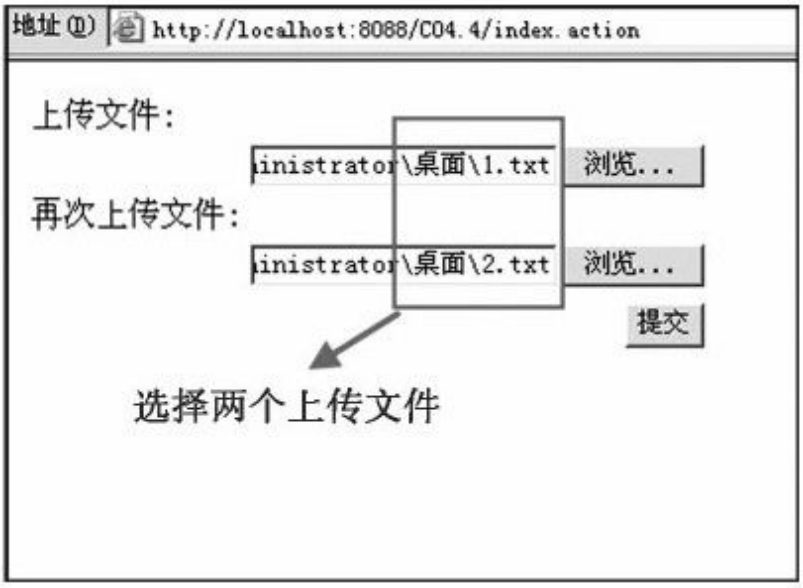

图　4.9　选择上传的文件

图　4.10　上传文件成功后的效果

图　4.11　上传文件的存放路径

源程序解读

（1）在upload.jsp中通过Form标签和File标签定义了两个上传文件。Struts 2标签会在后面章节中具体介绍，这里只是让读者知道如何使用标签显示如图4.8所示的内容。如果上传成功，在result.jsp中“[”和“]”之间显示上传文件的文件名，如果是多个文件，以“，”相隔。这些显示格式都是用Property标签定义的。

注意：如果上传文件，在JSP的Form中一定要定义如upload.jsp文件中黑体表示的部分。method和enctype属性都必须要如代码中所示，这样Form中上传文件才会起作用。

（2）UploadAction文件中先定义了常量UPLOADDIR，它是上传文件上传后存放的文件夹名字。比如这里使用的是JBoss，则在它的已部署Web项目下

的upload文件夹中，会有所有上传成功的文件。如图4.11所示可以看见上传文件最终存放路径。

注意：在MyEclipse中开发的“WebRoot”目录下也要新建一个upload文件夹，否则部署后在JBoss的已部署Web项目下将没有upload文件夹。因为部署的时会将所有“WebRoot”目录下的文件夹和文件都部署到JBoss的已部署Web项目下。

（3）定义好UPLOADDIR后，再定义上传文件的属性变量。也许看到其中的“fileFileName”和“fileContentType”会有点别扭，尤其是“fileFileName”，感觉不符合Java命名规范，但是这两个属性变量是4.1节中介绍的“fileUpload”拦截器类中的类公有变量名字，只有这样定义，UploadAction执行时会把在页面上选择的上传文件的属性值放在这两个变量中，否则调试UploadAction时会发现这两个变量都会是“null”即空值。不相信的

读者可以自行改变这两个变量名再执行上传文件功能进行调试，比较这两个变量得到的值。

注意：因为这里是进行多个文件上传功能开发，因此“file”、“fileFileName”、“fileContentType”属性变量都设定为List类型，其实还可以设定为数组类型，完全凭个人喜好而定。还有如果读者自己开发单个文件上传，就没必要把它们设定为List类型或数组类型。直接把“file”定义为Java的IO包中的File类型，将“fileFileName”定义为普通的String类型即字符串类型。

（4）之后在execute方法中，写一个循环，对所有页面中选择的上传文件单独进行上传。这里运用了重构中的“抽取方法”的方式，将上传文件的功能封装成一个私有方法，名字为“uploadFile”，其中运用了Java的IO包中的很多API方法。

（5）struts. xml中定义了＜constant＞标签，主要定义了文件名和文件内容显示的字符编码集以及这些被上传文件的临时存放路径。

先说明一下＜constant＞标签，这是定义整个Web项目的一些常量属性值，如果不定义则在Struts 2自带的default.properties（读者们可到自己安装Struts 2的文件路径src＼core＼src＼main＼resources＼org＼apache＼Struts 2＼下找到）文件中有这些常量的定义，比如在本节struts.xml文件中的“struts.i18n.encoding”和“struts.multipart.saveDir”在default.properties中定义的代码如下：

```
    <!……文件名：default.properties……>
    ###This can be used to set your default locale and encoding scheme
    #struts.locale=en__US
    struts.i18n.encoding=UTF8
    ###Parser to handle HTTP POST requests, encoded using the MIMEtype multipart/formdata
```

```
#struts.multipart.parser=cos
#struts.multipart.parser=pell
struts.multipart.parser=jakarta
#uses javax.servlet.context.tempdir by default
struts.multipart.saveDir=
```

如果不在struts.xml文件中定义，则Web项目会缺省使用default.properties文件中这两个常量属性的定义。一个将使字符编码集变为“UTF8”，另一个没有指定任何文件路径。而笔者开发的该Web项目缺省支持的字符编码集是“gb2312”，而且需要指定临时上传文件的存放路径（如果读者开发的Web项目缺省编码集就是“UTF8”，而且也不需要指定临时路径时，就没必要在struts.xml中定义这两个＜constant＞），因此有必要定义这两个属性符合项目开发要求。

注意：也可以如第3章那样，把这两个属性定义在自定义的struts.properties文件中，具体代码可以如下，

```
    <！……文件名：
struts.properties……>
    struts.i18n.encoding=gb2312
    struts.multipart.saveDir=c: \
```

个人认为这个方法比在struts.xml中定义更好，毕竟Struts 2自己也是定义在properties属性文件中，而不是定义在自己的xml配置文件中（Struts 2自带的xml配置文件为strutsdefault.xml，在4.1节中已记述）。这里是为了让读者知道struts.xml配置文件也可以配置这些属性，因此写在struts.xml配置文件中。从3.2节笔者说明struts.xml配置文件时并没有介绍＜constant＞标签，这点也可以知道笔者个人其实是不赞同这样的配置手段的，即在struts.xml中配置＜constant＞标签。

在＜Action＞标签中配置“result”，和第3章类似，将这两个JSP文件的导航流程配置好即可。

（6）开始进行文件上传功能，按照上述的步骤执行即可。笔者在桌面上新建了两个文本文件，将它们上传到JBoss已部署的Web项目中展示文件上传的upload文件夹下，如图4.11所示。

其实还可以指定上传文件的格式，让它只上传特定类型的文件。比如只能上传文本和xml文件，则在struts.xml需要显示配置“uploadFile”拦截器。代码如下：

```
    <! ……文件名：
struts.xml……>
    <struts>
    <! Action所在包定义>
    <package name="C04.4"extends="strutsdefault">
    <! Action名字，类以及导航页面定义>
    <! 通过Action类处理才导航的的Action定义>
    <action
name="upload"class="action.UploadAction">
    <result name="input">/jsp/upload.jsp</result>
    <result name="success">/jsp/result.jsp</result
>
    </action>
    <! 显示配置文件上传拦截器>
    <interceptorref name="fileUpload">
    <! 指定特定类型的上传文件>
```

```
    <param name="allowedTypes">text/plain,
application/xml</param>
    </interceptorref>
    <interceptorref name="defaultStack">
</interceptorref>
    </package>
    </struts>
```

定义了一个名为“allowedTypes”的参数，其中在<param></param>之间的是文件类型，也可以用“，”间隔，表示允许上传多个文件类型。这里允许上传文件类型为txt、xml格式的文件。如果读者不知道各个文件类型的定义，可在自己的JBoss安装目录中的server\default\deploy\jbossweb.deployer\conf\下web.xml文件中找到（搜索<mimemapping>即可）。

注意：如果显示配置Struts 2自己的缺省拦截器一定要写在“defaultStack”前，否则“fileUpload”拦截器不会执行拦截。因为Struts 2

中如果某个拦截器执行拦截时发现自己已经执行过，第二个乃至之后同名的拦截器都不会再执行。

这里因为“defaultStack”拦截器栈中包含了“fileUpload”拦截器，而“fileUpload”拦截器已经执行拦截了，则不会再执行拦截。如果把“defaultStack”拦截器栈放在“fileUpload”拦截器前配置，则只执行“defaultStack”拦截器栈中的“fileUpload”拦截器，这里没有定义“allowedTypes”，因为Struts 2默认的是支持所有文件类型，所以它会支持所有文件类型的文件上传，因此再设定“allowedTypes”也就没有任何意义。

4.4.2 Struts 2文件下载功能开发

技术要点

本节代码详细说明文件下载功能的开发流程。

上传成功页面修改后支持文件下载代码内容。

DownloadAction文件下载功能开发。

struts.xml中DownloadAction配置，以及支持文件名为中文字符的文件下载。

下载文件流程展示。

实现代码

上传成功页面，让其在每个上传文件后提供“下载”链接，代码如下：

```
    <！……文件名：
result.jsp……>
    <%@taglib prefix="s"uri="/strutstags"%>
    <body>
    上传文件：
    <table>
    <！循环显示上传成功文件名>
    <s: iterator value="fileFileName"status="fn">
    <tr>
    <td>
    <！上传成功文件名>
    <s: property/>
    </td>
    <td>
    <！下载文件链接内容为定义的下载Action>
    <！下载文件名作为链接参数fileName值，用OGNL表达式表达
>
    <a href="<s: url value=′ download.action'><
s: param name=′ fileName'
    value=′ fileFileName[#fn.getIndex（）]′ />
    </s: url>">下载</a></td>
    </tr>
    </s: iterator>
    </table>
    </body>
```

DownLoadAction类的代码如下：

```
    <！……文件名：
DownLoadAction.java……>
    import java.io.InputStream;
    import java.io.UnsupportedEncodingException;
    import org.apache.Struts
2.ServletActionContext;
```

```
    import com.opensymphony.xwork2.ActionSupport;
    public class DownLoadAction extends
ActionSupport {
    //下载文件原始存放路径
    private final static String
DOWNLOADFILEPATH="/upload/";
    private String fileName; //文件名参数变量
    public String getFileName() {
    return fileName;
    }
    public void setFileName(String fileName) {
    this.fileName=fileName;
    }
    //从下载文件原始存放路径读取得到文件输出流
    public InputStream getDownloadFile() {
    return
    ServletActionContext.getServletContext().getRe
sourceAsStream(DOWNLOADFILEPATH+fileName);
    }
    //如果下载文件名为中文，进行字符编码转换
    public String getDownloadChineseFileName() {
    String downloadChineseFileName=fileName;
    try {
    downloadChineseFileName=new
String(downloadChineseFileName.get
    Bytes(), "ISO88591");
    } catch(UnsupportedEncodingException e) {
    e.printStackTrace();
    }
    return downloadChineseFileName;
    }
    public String execute() {
    return SUCCESS;
    }
    }
```

struts. xml配置文件中有关文件下载的配置：

```
    <！……文件名：
struts.xml……>
    <struts>
    <！下载文件的Action定义>
    <action
name="download"class="action.DownLoadAction">
    <！设置文件名参数，由页面上传入>
    <param name="fileName"></param>
    <result name="success"type="stream">
    <！下载文件类型定义>
    <param name="contentType">text/plain</param>
    <！下载文件处理方法>
    <param name="contentDisposition">
    </param>
    <！下载文件输出流定义>
    <param name="inputName">downloadFile</param>
    </result>
    </action>
    </struts>
```

文件开始下载页面如图4.12所示。单击“下载”按钮，则会出现如图4.13所示的对话框，单击“保存”按钮后选择下载文件存放的路径，如图4.14所示。

图　4.12　文件下载

图　4.13　下载文件处理方式

图　4.14　下载文件选择存放路径

源程序解读

（1）在result.jsp中通过iterator标签和url标签定义了“fileFileName”的循环显示以及链接。其中有关“status”和OGNL表达式会在后面章节中具体介绍，这里只是让读者知道是如何使用标签显示如图4.12所示的内容。特别指出＜param＞标签为downloadAction定义了一个参数，该参数名为“fileName”，因为在4.4.1节中定义的“fileFileName”是个List类型的数据集合，因此利用OGNL表达式将文件名作为“fileName”参数值传入downloadAction中。

（2）DownLoadAction文件中先定义了常量DOWNLOADFILEPATH，它是下载文件在服务器存放的路

径名，也就是4.4.1节中上传文件在服务器存放的路径名。

定义好DOWNLOADFILEPATH后，在定义DownLoadAction的属性变量。因为在result.jsp中定义了参数“fileName”，则它作为DownLoadAction的属性变量，需要定义相应的getter、setter方法。

然后定义了getDownloadFile方法，它返回的是一个文件流，表明将被下载文件转换为输出流，方便下载。利用Struts 2自带的“ServletActionContext”类的API把下载文件存放路径作为方法参数，读取下载文件，将其转换为文件流。

还有一个getDownloadChineseFileName方法，该方法主要作用是将文件名为中文字符的文件进行文件名的字符编码集合转换。因为在Web系统中由JSP等视图页面传入的变量值，特别是中文字符的变量，缺省

的字符编码集合都是“ISO88591”，因此利用Java的字符串类的API，将字符编码转成开发需要的字符编码集，防止中文字符乱码问题的发生。

（3）struts. xml中定义了名为“download”的Action。其中它自己的参数“fileName”在这里的值会从JSP页面上传入，所以这里只是定义，没有具体给它赋任何值。

（4）在＜result＞标签中定义了type属性，值为“stream”。如果是下载文件功能开发，DownLoadAction一定要设置type属性，而且值为“stream”。这是因为在Struts 2自带的xml配置文件strutsdefault.xml中有关于“stream”的result返回类型的定义，代码如下：

```
    ＜！……文件名：strutsdefault.xml……＞
    ＜resulttype name="stream"class="org.apache.Struts
```

```
2.dispatcher.StreamResult"/>
```

这里Struts 2定义了result返回类型为“stream”，这个result类型主要是处理文件的输入和输出流时需要使用的。因为下载文件就是把文件转换成输入输出流，将其从一个文件路径放到另外一个文件路径中去，所以肯定要设置这个result类型的。

（5）“contentType”、“contentDisposition”、“inputName”都是这个result的属性。“contentType”就是文件类型。这里因为下载的文件是文本文件，因此设定的值为文本文件类型，具体各个文件类型如何定义，4.4.1节已经介绍过，这里不再做说明。

“contentDisposition”是指定下载文件处理方式，如图4.13就是处理方式的效果。特别指出如果“contentDisposition”定义的值把前面的

“attachment”去掉，则下载方式不是以附件方式下载，如果单击“下载”链接，则会把下载文件的内容显示在浏览器中。读者可以去试验一下。

（6）这里有个“${downloadChineseFileName}”，这就是在DownLoadAction中定义getDownloadChineseFileName方法的目的，${downloadChineseFileName}是OGNL的表达式，它显示了“downloadChineseFileName”变量的具体值，因为在DownLoadAction中定义getDownloadChineseFileName方法，则把已经转换成符合需要字符编码集的下载文件名作为下载文件方式对话框中显示的名称，不会造成任何乱码问题。

“inputName”是最关键的一个属性，也是一定要定义的属性，“inputName”参数中定义的值“downloadFile”是DownLoadAction中getDownloadFile方法返回的文件流名字。在Struts 2

中Acion用前缀名为get的方法得到各种属性的值，这些属性有些是在Action中定义，有些是在配置文件中利用OGNL表达式或直接定义。

（7）开始进行文件下载功能展示，按照如上记述的步骤执行即可。笔者将两个文本文件上传上去，然后在上传成功页面对具体的文件进行下载。在图4.13中单击“保存”按钮就会出现图4.14，选择在本机上存放下载文件的路径即可完成下载文件功能。

第5章　Struts 2标签库

在之前的章节中，很多代码示例都用到Struts 2的一些标签。Struts 2的标签库中所包含的标签是非常多的。因此有必要为Struts 2标签实现原理、机制，以及各种类型的标签在开发过程中的使用方式做介绍，这样读者就可以在开发中有信心地使用Struts 2标签，并且可以自定义自己的标签，应用到开发工作中去。

5.1　Struts 2标签使用原理解疑

在下载的Struts 2的包中，读者可以在/lib文件夹下找到struts 2core2.0.11.1.jar包，解压该包在其根目录下的/METAINF文件夹下，可以看到一个名字为“strutstags.tld”文件。该文件就是Struts 2中

所有自带的标签库定义。本节通过对该文件代码的介绍，来让读者知晓Struts 2内部是如何使用这些标签来进行工作的，并简单说明JSP中是如何用其来书写标签代码的。

技术要点

本节代码说明Struts 2内部定义标签的格式和在JSP中的使用方式。

strutstags.tld文件标签定义配置格式。

JSP中使用标签功能介绍。

实现代码

Struts 2中所有自带的标签库定义文件如下所示，详细注释参考后面的代码解释。

```
    <！……文件名：
strutstags.tld……>
```

```
    <taglib>
    <tlibversion>2.2.3</tlibversion>
    <jspversion>1.2</jspversion>
    <shortname>s</shortname>
    <uri>/strutstags</uri>
    <displayname>"Struts Tags"</displayname>
    <description>……</description>
    <tag>
    <name>action</name>
    <tagclass>org.apache.Struts
2.views.jsp.ActionTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <description><! [CDATA[Execute an action from
within a view]]></description>
    <attribute>
    <name>executeResult</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
    <description><! [CDATA[Whether the result of
this action（probably a view）should be executed/
    rendered]]></description>
    </attribute>
    ……
    <attribute>
    <name>namespace</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
    <description><! [CDATA[Namespace for action
to call]]></description>
    </attribute>
    </tag>
    </taglib>
```

源程序解读

（1）strutstags.tld是Struts 2自标签定义文件。所有标签定义都是在＜tablib＞和＜/taglib＞之间定义。＜tag＞＜/tag＞用来定义一个具体标签。每个标签都可以有很多自己的属性，这些属性的定义都是以＜attribute＞＜/attribute＞来定义。

（2）＜tlibversion＞＜/tlibversion＞之间定义的是标签库的版本。＜jspversion＞＜/jspversion＞定义的是标签库的标签支持JSP的哪个版本。＜shortname＞＜/shortname＞其实是标签库的默认名，也可以认为是其昵称。＜uri＞＜/uri＞定义的是标签库的URI，在JSP中会使用到。＜displayname＞＜/displayname＞是显示名。＜description＞＜/description＞是标签库的记述，记述标签库的使用用途等。

（3）＜attribute＞中＜name＞＜/name＞是属性名称定义。＜required＞＜/required＞表示的该属性是否是必须的属性，如果是必须的则＜required＞＜/required＞之间为true，否则为false。＜rtexprvalue＞＜/rtexprvalue＞表示的是可否使用表达式，大多数标签都是为false。这里不是不能使用表达式，而是恰恰相反表示可以使用表达式。＜description＞＜/description＞定义和前面介绍相同。

（4）在JSP中，如之前章节的演示代码所示，都是在文件头有个使用标签的声明，代码如下：

```
    <!……文件名: .jsp……>
    <%@taglib prefix="s"uri="/strutstags"%>
```

有了这个声明，在JSP文件中就可以使用Struts 2的标签。比如form标签的定义代码如下：

```
    <s: form action="upload"……>
```

记住一定要用“s”，它是Struts 2中标签的默认名也是相当于一个昵称，当然读者也可以把它改为自己想取的名字，不过在标签声明中的“prefix”中也要改成自己所取的名字。

注意：因为笔者使用的Servlet版本是2.3之上的版本，因此没必要在web.xml中定义标签库。如果读者使用的Servlet版本比较低，则在web.xml文件中需要定义如下的代码：

```
    <！……文件名：
web.xml……>
    <taglib>
    <！定义URI>
    <tagliburi>/Struts 2tags</tagliburi>
    <！定义标签库支持的jar包位置>
    <tagliblocation>/WEBINF/lib/Struts
2core2.0.11.1.jar</tagliblocation>
    </taglib>
```

只有这样标签库才会在Servlet版本比较低的情况下使用。

5.2 OGNL表达式语言介绍

在进行Struts 2的标签库介绍之前，有必要着重对OGNL（Object Graph Navigating Language）对象导航语言做一详细解析。因为在之后的演示代码中经常会用到一些有关OGNL的代码，为了更好地学习Struts 2的标签库，笔者把OGNL当作学习Struts 2标签库的基础知识来介绍，让大家学习Struts 2的基础打得更加扎实。

技术要点

本节代码对OGNL一些常用特性进行分析，用演示代码演示这些特性。

常用特性介绍。

OGNL在Struts 2页面中的应用。

OGNL特殊符号介绍。

实现代码

利用OGNL进行应用的页面：

```
    <！……文件名：
ognl.jsp……>
    <%@taglib prefix="s"uri="/strutstags"%>
    <body>
    <！OGNL显示request、response中的值>
    <h3 align="left">Session和Request值</h3>
    request.materialName：<s：property
value="#request.materialName"/><br/>
    session.materialName：<s：property
value="#session.get（′materialName'）"/><br/>
    <！OGNL显示条件表达式过滤的数据>
    <h3 align="left">根据条件显示数据</h3>
    <p>价格小于50元的建材</p>
    <ul>
    <s：iterator value="materials.｛? #this.mainbid
<50｝">
    <li><s：property value="materialName"/>建材价
格是<s：property value="mainbid"/>
    元！</li>
    </s：iterator>
    </ul>
    <p>"人造石台面"的库存数量是：<s：property
value="materials.｛? #this.materialName
    ==′人造石台面′｝.{mount}[0]"/></p>
    <！OGNL新建Map类型数据集合，显示子元素值>
    <h3 align="left">Map数据显示</h3>
```

```
    <s: set name="frank"value="# {' material': ' 欧龙无苯油漆（六度）' , ' mount': ' 500' } "/>
    <p>供销商frank手里还有建材<s: property value="#frank[' material']"/></p>
    <p>库存量为<s: property value="#frank[' mount']"/></p>
    </body>
```

---

struts. xml文件配置：

```
    <! ……文件名: struts.xml……>
    <package name="C05.2"extends="strutsdefault">
    <! 创建Action>
    <action name="ognl"class="action.OgnlExampleAction">
    <result name="success">/jsp/ognl.jsp</result>
    </action>
    </package>
```

OGNL示例Action代码：

```
    <! ……文件名: OgnlExampleAction.java……>
    public class OgnlExampleAction extends ActionSupport {
    //设置request、response参数和需要显示的数据集合定义
    private HttpServletRequest request;
    private HttpSession session;
    private List<Material>materials;
```

```
    public Strin xecute（）throws Exception｛
    request=ServletActionContext.getRequest（）；
    session=request.getSession（）；
    //设置request、session存放值
    request.setAttribute（"materialName"，"人造石台面
From request"）；
    session.setAttribute（"materialName"，"欧龙无苯油
漆（六度）From session"）；
    //初始化数据集合，集合类型为List
    materials=new ArrayList＜Material＞（）；
    materials.add（new Material（"欧龙无苯油漆（六
度）"，100，2000））；materials.add（new
Material（"610mm门套线红影木夹板饰面（单面）"，20，
2900））；
    materials.add（new Material（"人造石台面"，56，
800））；
    return SUCCESS;
    }
    //setter，getter方法
    public HttpServletRequest getRequest（）｛
    return request;
    }
    public void setRequest（HttpServletRequest
request）｛
    this.request=request;
    }
    ……
    }
```

材料数据类：

```
    ＜！……文件名：
Material.java……＞
    //材料对象
```

```
    public class Material {
    private String materialName; //材料名
    private int mainbid; //材料价格
    private int mount; //材料数量
    //构造初始化数据
    public Material（String materialName, int
mainbid, int mount） {
    super（）;
    this.materialName=materialName;
    this.mainbid=mainbid;
    this.mount=mount;
    }
    //setter, getter方法
    public String getMaterialName（） {
    return materialName;
    }
    public void setMaterialName（String
materialName） {
    this.materialName=materialName;
    }
    ……
    }
```

运行效果如图5.1所示。

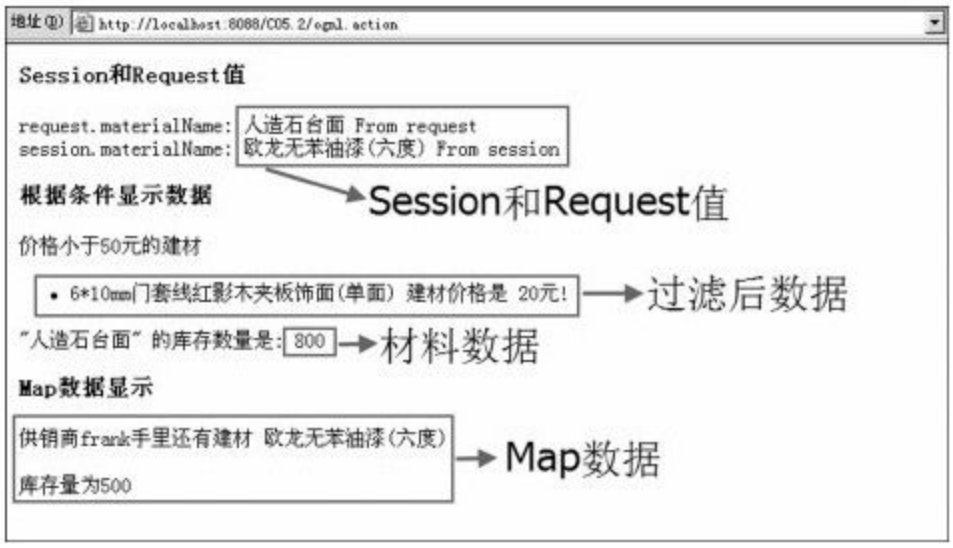

图　5.1　OGNL运行示例效果

源程序解读

（1）Struts 2中将应用范围中的数据以及Action处理的数据都存储在一个区域中。在Struts 2中叫做“valueStack”，中文名为“值栈”。而OGNL就是从“值栈”中取出数据，并在某些条件下进行数据过滤和计算的表达式语言。在Struts 2中值栈作为OGNL的根对象，从值栈中取出自己需要的数据，而且值栈存取数据是按照先入后出的概念。因此查询数据时往往是值栈中最顶部的数据先被查询出来。更加需要指出

的是在值栈中也可以使用索引，可以在指定的索引位置开始搜索数据。

（2）在OGNL中如果搜索的数据不是值栈中存储的数据，而且其他对象中，特别是Struts 2的ActionContext中的对象，则在访问这些对象时，前面要加“#”。比如ognl.jsp中访问session和request对象时代码中在request和session之前就有“#”。

（3）对于数据集合类型，如果根据条件查询该集合中的数据，形成子集合的时，常用的OGNL还有“？”、“^”、“$”，此三个符号中“？”是取得所有符合条件的数据时使用。“^”是取这些符合条件的数据索引中第一个或第一条数据。而“$”则正好相反，是取最后一个或最后一条数据。

（4）在OGNL中，还有lamuda表达式和很多计算公式的表达式，笔者认为如果坚持松耦合理念的话，最

好不要使用。毕竟OGNL更多地使用在MVC模式中view和control两层，而计算公式等OGNL知识更多的和业务逻辑联系关联比较大，所以本书就不做过多介绍。

（5）本示例中，首先创建了建筑材料的模型对象。其中有材料名、材料价格、材料库存量三个属性。在OgnlExampleAction中做了两件事情，一件事情是在request、session中各放入key为“materialName”的值。另一件事情是初始化三个材料对象，将这些对象放在List类型的数据集合中。

（6）然后在struts.xml中对OgnlExampleAction做定义，将其导航到ognl.jsp中。在该jsp中，读者参看其代码。首先将request、session中“materialName”的值用OGNL取出。因为request、session都不是值栈，所以在它们名字之前都以“#”开头。在代码中将之前List类型的数据集合中所有材

料价格小于50的数据显示出来。这里用了“？”也就是之前所说去取符合条件的数据集时使用。

（7）在取材料名为“人造石台面”的材料库存量时，OGNL先利用条件过滤得到符合条件的材料对象集然后用“｛｝”去取库存量属性，再取该子集中第一个索引，这里索引次序和Java中相同，也是从“0”开始表明集合中第一个元素。

（8）在ognl.jsp中OGNL也新建了一个Map对象，然后定义了该对象中的key和value。OGNL对于显示Map对象中某个key的值采取ognl.jsp代码中所示。“[]”中写key的名字，并用‘’包含。

（9）OGNL中经常会有“#”、“%”、“$”三个符号。一般“#”都是像前文所述代表那些非值栈的对象。还有就是ognl.jsp中创建Map对象时用到。而“%”则是显示对象中的值。至于“$”则是用来显示

属性文件中定义的值。比如某属性文件中定义了“kkk=10”则在Struts 2的struts.xml或者JSP文件中，用“｛kkk｝”时则系统会读取“10”这个值作为显示值。

5.3 Struts 2控制标签简介

Struts 2的控制标签主要用处是控制JSP等视图中流程的转向问题。笔者将一些常用的控制标签进行说明。

5.3.1 append标签使用介绍

技术要点

本节代码具体介绍append控制标签的使用方式介绍。

append标签使用。

append标签功能演示。

实现代码

一个使用标签的JSP文件：

```
    <!……文件名：
appendTag.jsp……>
    <body>
    <h3 align="left">
    append标签使用范例
    </h3>
    <s: set name="frank"value="# {′ material': ′ 欧龙
无苯油漆（六度）′ , ′ mount': ′ 500′ } "/>
    <s: set name="jakcy"value="# {′ material': ′ 进户
门套油漆′ , ′ mount': ′ 800′ } "/>
    <s: append id="SP">
    <s: param value="frank"/>
    <s: param value="jakcy"/>
    </s: append>
    <p>
    供销商frank和jakcy手里还有建材
    <s: iterator value="#SP">
    <tr>
    <td>
    <p>
    <s: property/>
    </p>
    </td>
    </tr>
    </s: iterator>
    </body>
```

功能演示如图5.2所示。

图　5.2　append标签范例

源程序解读

append标签的功能是将多个集合合并成一个集合。其中id是命名了多个集合的集合名。所有被合并的集合都是在＜param＞里定义的。如代码所示，将已定义的两个Map集合合并成名为“SP”的集合中。在＜param＞的value属性中就是这两个被合并的集合名字。

5.3.2　generator标签使用介绍

技术要点

本节代码具体介绍generator控制标签的使用方式介绍。

generator标签使用。

generator标签功能演示。

实现代码

一个使用标签的JSP文件：

```
    <! ……文件名：generatorTag.jsp……>
    <body>
    <h3 align="left">
    generator标签使用范例
    </h3>
    <s: set name="frank"value=" {′ 欧龙无苯油漆（六度）；进户门套油漆；踢脚线；奥普浴霸′ } "/>
```

```
    <s: generator
separator="; "val="frank"id="example"count="2">
</s: generator>
    <s: iterator value="#attr.example">
    <tr>
    <td>
    <p>
    <s: property/>
    </p>
    </td>
    </tr>
    </s: iterator>
    </body>
```

功能演示如图5.3所示。

图　5.3　generator标签范例

源程序解读

generator标签的功能是将一个字符串按照指定的分隔符将该字符串分割成多个字符串集合。其中id是分割后的字符串集合名。separator是分隔符值。val

是要被分割的字符串名。count是定义显示分割后字符串集合中元素的个数。如代码所示，被分割后的字符串集合元素共有4个，但定义的count是2，则表明只需要显示其中两个元素。

注意：id、separator、val、count等属性中定义时的值一定要以""包含。特别是count，开发者容易写成count=2，而不是count="2"。前者写法是不正确的，在IDE中也会给出错误提示。希望读者仔细注意这一点。

5.3.3 if、else、elseif标签使用介绍

技术要点

本节代码具体介绍if、else、elseif控制标签的使用方式介绍。

if、else、elseif标签使用。

if、else、elseif标签功能演示。

实现代码

使用标签的JSP文件：

```
    <!……文件名：ifelseifTag.jsp……>
    <body>
    <h3 align="left">
    ifelseif标签使用范例
    </h3>
    <s: set name="frank"value="{' 欧龙无苯油漆（六度）',' 进户门套油漆',' 踢脚线',' 奥普浴霸'}"/>
```

```
    <s: if test="% {#frank[1]==' 踢脚线' } ">
    <s: property value="% {#frank[1]} "/>
    </s: if>
    <s: elseif test="% {#frank[1]==' 进户门套油漆
' } ">
    <s: property value="% {#frank[1]} "/>
    </s: elseif>
    <s: else>
    不是
    </s: else>
    </body>
```

功能演示如图5.4所示。

图　5.4　if、else、elseif标签范例

源程序解读

（1）if标签的功能判断条件是否符合if中定义的条件，和Java中if的功能相同。

（2）else标签的功能判断条件是否符合else中定义的条件，和Java中else的功能相同。

（3）elseif标签的功能判断条件是否符合elseif中定义的条件，和Java中else if的功能相同。

（4）上述三个标签中test都是判断的条件定义。返回的是一个boolean值，如果判断条件为真即返回true值，则标签中定义的内容才会在页面上显示。如代码所示，网页中显示的是elseif标签中的内容。因为只有该标签的test返回的是true，而且一旦返回true值，后面的标签内便不执行。和Java中if、else的功能是完全相同的。

5.3.4　iterator标签使用介绍

技术要点

本节代码具体介绍iterator控制标签的使用方式介绍。

iterator标签使用。

iterator标签功能演示。

实现代码

使用标签的JSP文件：

```
    <!……文件名：iteratorTag.jsp……>
    <body>
    <h3 align="left">
    iterator标签使用范例
    </h3>
    <s: set name="frank"value="{' 欧龙无苯油漆（六度）',' 进户门套油漆',' 踢脚线',' 奥普浴霸'}"/>
```

```
<s: iterator value="#frank">
<p>
<s: property/>
</p>
</s: iterator>
</body>
```

功能演示如图5.5所示。

图　5.5　iterator标签范例

源程序解读

（1）iterator标签的功能是对某个集合中的所有属性进行迭代遍历。和Struts中的iterator标签功能完全相同。其中id也和之前记述标签相同。而value则可以指定需要被迭代遍历的集合，如果不显式指定，则表示是使用在值栈的栈顶的数据集合。还有一个

status属性，它其实表示的是Struts 2中的IteratorStatus对象的具体某实例。其中有一些API都是和集合的顺序索引有关。有兴趣的读者可以去参看Struts 2中IteratorStatus的源代码。

（2）代码所示显示了字符串集合frank的所有字符串。

5.3.5　merge标签使用介绍

技术要点

本节代码具体介绍merge控制标签的使用方式。

merge标签使用。

merge标签功能演示。

实现代码

使用标签的JSP文件：

```
    <! ……文件名：
mergeTag.jsp……>
    <body>
    <h3 align="left">
    merge标签使用范例
    </h3>
    <s: set name="frank"value="# {' material': ' 欧龙
无苯油漆（六度）' , ' mount': ' 500′ } "/>
    <s: set name="jakcy"value="# {' material': ' 进户
门套油漆′ , ' mount': ' 800′ } "/>
    <s: merge id="SP">
```

```
<s: param value="frank"/>
<s: param value="jakcy"/>
</s: merge>
<p>
供销商frank和jakcy手里还有建材
<s: iterator value="#SP">
<tr>
<td>
<p>
<s: property/>
</p>
</td>
</tr>
</s: iterator>
</body>
```

功能演示如图5.6所示。

图　5.6　merge标签范例

源程序解读

merge标签的功能和append标签功能相同。唯一的不同点在于merge标签合并的集合中各个元素的排列顺序是不同的，通过图5.6和图5.2的比较可知。其属性和append标签完全相同。

5.3.6　sort标签使用介绍

技术要点

本节代码具体介绍sort控制标签的使用方式。

sort标签使用。

sort标签功能演示。

实现代码

使用标签的JSP文件，代码如下：

```
    <!……文件名：
sortTag.jsp……>
    <body>
    <h3 align="left">
    sort标签使用范例
    </h3>
    <s: set name="frank"value="{' 欧龙无苯油漆（六度）
' , ' 进户门套油漆' , ' 踢脚线' , ' 奥普浴霸' } "/>
    <s: sort source="frank"comparator="sort">
    <s: iterator>
    <tr>
```

```
    <td>
    <p>
    <s: property/>
    </p>
    </td>
    </tr>
    </s: iterator>
    </s: sort>
    </body>
```

定义的分类条件代码：

```
    <！……文件名：
SortAction.java……>
    public class SortAction extends ActionSupport {
    //新建内部类comparator，定义分类条件：按照字符串长度从
短到长排列
    public Comparator getSort（） {
    return new Comparator（） {
    public int compare（Object arg1，Object arg2） {
    return（（String）arg1）.length（）（（String）
arg2）.length（）；
    }
    }；
    }
    }
```

功能演示如图5.7所示。

图　5.7　sort标签范例

源程序解读

（1）sort标签的功能是对指定的数据集合进行排序或分类。可以自定义自己的排序或分类条件。id和之前技术标签相同。comparator返回一个java.util.Comparator类型实例，用来定义排序或分类的条件。source定义的是被排序或分类的集合名。

（2）如果需要自定义comparator，则可以在Action中扩展Comparator接口，重写compare方法即可。如代码所示，通过定义内部类返回Comparator类

型对象，在内部类中重写compare方法，定义了按照字符串长度由短到长排序的条件。

有对内部类不是很熟悉的读者，可以去查看Java相关资料自行研究学习。这里不具体解释。

5.3.7　subset标签使用介绍

技术要点

本节代码具体介绍subset控制标签的使用方式。

subset标签使用。

subset标签功能演示。

实现代码

使用标签的JSP文件，代码如下：

```
    <!……文件名：
subsetTag.jsp……>
    <body>
    <h3 align="left">
    subset标签使用范例
    </h3>
    <s: set name="frank"value="{' 欧龙无苯油漆（六度）
' , ' 进户门套油漆' , ' 踢脚线' , ' 奥普浴霸' } "/>
    <s: subset source="frank"start="2"count="2">
    <s: iterator>
    <p>
```

```
    <s: property/>
    </p>
    </s: iterator>
    </s: subset>
    <h3 align="left">
    使用decider的使用范例
    </h3>
    <s: subset
source="frank"decider="selfDecider">
    <s: iterator>
    <p>
    <s: property/>
    </p>
    </s: iterator>
    </s: subset>
    </body>
```

定义的读取子集条件代码：

```
    <!……文件名：
SelfDecider.java……>
    import org.apache.Struts
2.util.SubsetIteratorFilter.Decider;
    public class SelfDecider implements Decider {
    public boolean decide（Object arg1）throws
Exception {
    String condition=（String）arg1;
    //读取元素中包含"（"字符的字符串
    return condition.indexOf（"（"）>0;
    }
    }
```

功能演示如图5.8所示。

图　5.8　subset标签范例

源程序解读

（1）subset标签的功能是根据读取子集条件将某个集合的子集读取出来。也可以自定义自己的读取子集条件。id和之前技术标签相同。source定义的是父集合名。start表示是从父集合哪一个元素的索引序号开始读取。count表示的是读取父集合中多少个元素。decider就是读取子集的条件，可以不写，如果写了，则表明一个Decider对象实例，需要开发者自行定义此对象实例。

（2）如代码所示，第一个示例表明的是从字符串集合中的第三个也就是索引号为2（索引号为0表示的是第一个元素）的元素开始读取，读取元素个数为2即只需要从第三个元素开始的两个元素作为子集合的元素。因此页面上显示的是两个元素。

第二个示例则自定义了Decider对象实例，代码中表示将元素中包含字符"（"的元素读取出来作为子集合的元素。因此在字符串集合中只有"欧龙无苯油漆（六度）"包含"（"字符，所以子集合中显示在页面上的元素只有它一个。

5.4 Struts 2数据标签简介

Struts 2的数据标签主要用处是提供数据访问。笔者将一些常用的数据标签进行说明，这些数据标签包括action、bean等非常关键的内容。

5.4.1 action标签使用介绍

技术要点

本节代码具体介绍action数据标签的使用方式。

action标签使用。

action标签功能演示。

实现代码

显示action结果视图的JSP文件，代码如下：

```
    <! ……文件名：
actionTag.jsp……>
    <body>
    <! 显示Action参数>
    <s: property value="param"/>
    </body>
```

使用action标签的JSP文件，代码如下：

```
    <! ……文件名：
showActionTag.jsp……>
    <head>
    <title>显示Action视图和参数？</title>
    </head>
    <body>
    <h3 align="left">
    显示结果视图
    </h3>
    <s: action
name="actionTag"executeResult="true"></s: action>
    <h3 align="left">
    不显示结果视图
    </h3>
    <s: action
name="actionTag"executeResult="false"></s: action
>
    <h3 align="left">
    忽略传递的Action参数
    </h3>
    <s: action
name="actionTag"executeResult="true"ignoreContextP
arams="true"></s: action>
    </body>
```

定义action的配置文件内容：

```
    <! ……文件名：
struts.xml……>
    <package name="C05.4"extends="strutsdefault">
    <action
name="actionTag"class="action.ActionAction">
    <result name="success">/jsp/actionTag.jsp
</result>
    </action>
    </package>
```

功能演示如图5.9所示。

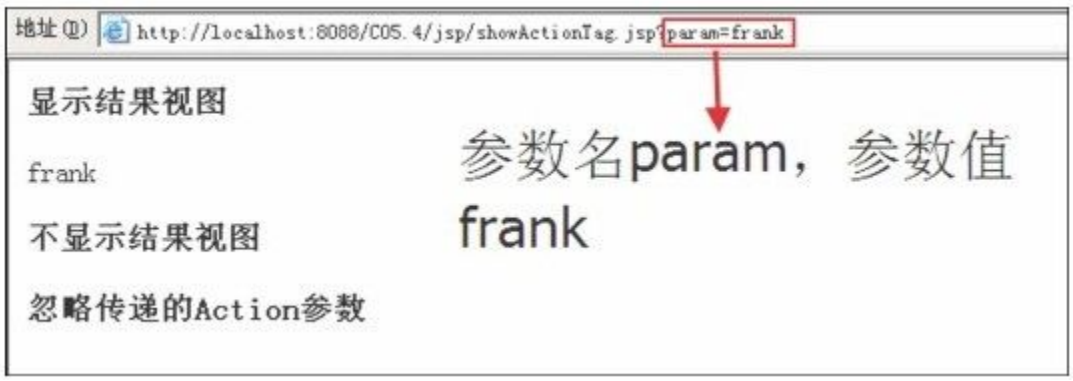

图　5.9　action标签范例

源程序解读

（1）action标签的功能是实现在JSP中调用Struts 2的Action。其中id是需要调用的Action的标识定义。name才是Action的名字，这两个属性开发者经常会混淆，所以需要搞清楚。除了这两个属性，namespace是用来指定调用的Action的命名空间。

（2）除了上述属性之外。Action标签还有两个比较重要的属性。第一个是executeResult属性。它是一个布尔型的值。如果为“true”则表示在JSP页面中返回调用的Action的结果视图。反之则不显示，默认值是false。还有一个是ignoreContextParams，该属性是来决定视图中请求的参数是否需要传递到Action中去。顾名思义，它也是一个布尔型的值。如果为false则需要把参数传递到Action中，反之则不传递。它的默认值也是flase。

注意：在笔者对Struts 2标签的研究中，几乎所有标签的属性中如果是一个表示布尔型的值，默认值都为false。因此如果需要定义这些属性，都需要在视图中（大多数情况下是JSP）显式定义为“true”。

（3）如代码所示，在图5.9中红框框中的就是需要传递到Action的参数内容。因为在JSP代码中最后使用了ignoreContextParams属性，并且设定为“true”，因此并没有将参数值显示在页面上。

5.4.2　bean标签使用介绍

技术要点

本节代码具体介绍bean数据标签的使用方式。

bean标签使用。

bean标签功能演示。

实现代码

使用bean标签的JSP文件：

```
    <!……文件名：
beanTag.jsp……>
    <body>
    <h3 align="left">
    bean标签内访问数据
    </h3>
    <s: bean name="model.Material">
    <s: param name="materialName"value="′进户门套油
漆′"/>
    <s: param name="mainbid"value="70"/>
    <s: param name="mount"value="200"/>
```

```
    <p>
    <s: property value="materialName"/>
    </p>
    <p>
    <s: property value="mainbid"/>
    </p>
    <p>
    <s: property value="mount"/>
    </p>
    </s: bean>
    <h3 align="left">
    bean标签外访问数据
    </h3>
    <s: bean name="model.Material"id="material">
    <s: param name="materialName"value="' 进户门套油
漆' "/>
    <s: param name="mainbid"value="70"/>
    <s: param name="mount"value="200"/>
    </s: bean>
    <p>
    <s: property value="#material.materialName"/>
    </p>
    <p>
    <s: property value="#material.mainbid"/>
    </p>
    <p>
    <s: property value="#material.mount"/>
    </p>
    </body>
```

使用Java定义的bean文件内容：

```
    <! ……文件名：
material.java……>
```

```
    //材料对象
    public class Material {
    private String materialName; //材料名
    private int mainbid; //材料价格
    private int mount; //材料数量
    public String getMaterialName() {
    return materialName;
    }
    public void setMaterialName(String
materialName) {
    this.materialName=materialName;
    }
    ……}
```

功能演示如图5.10所示。

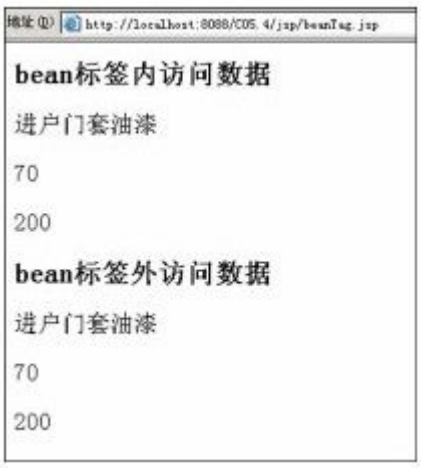

图　5.10　bean标签范例

源程序解读

（1）bean标签的功能是用来创建一个具体的JavaBean实例。其中id是对需要创建的JavaBean实例对象进行标识定义。name是JavaBean的名字，即类名。

（2）可以在bean标签中使用＜param＞来指定JavaBean的属性值。如果用＜param＞来指定属性值其实和JavaBean中的setter方法是具有同等功能的。

注意：如果需要输出定义的属性值的话，在bean标签内部定义是可以直接输出的。假设在外部定义的话，bean标签一定要定义id属性，在代码中笔者已经用黑体标明。这样在外部定义输出，可以直接使用该JavaBean的id来调用属性，注意要在id前使用“#”，在代码中也有说明，请读者仔细研究。

5.4.3　date标签使用介绍

技术要点

本节代码具体介绍date数据标签的使用方式。

date标签使用。

date标签功能演示。

实现代码

使用date标签的JSP文件：

```
    <!.....文件名：
dateTag.jsp.....>
    <body>
    <h3 align="left">
    date标签使用范例
    </h3>
    <h3 align="left">
    date类型两种表达方式
    </h3>
    <p>
```

```
    北京奥运会将于<s: date
name="omplicDate"format="MMddyyyy"/>召开
    </p>
    <p>
    北京奥运会将于<s: date
name="omplicDate"format="yyyy年MM月dd日"/>召开
    </p>
    <h3 align="left">
    默认日期显示
    </h3>
    <p>
    <s: date name="omplicDate"nice="false"/>
    </p>
    <h3 align="left">
    当前日期和定义日期时间间隔
    </h3>
    <p>
    北京奥运会开幕<s: date
name="omplicDate"nice="true"/>后
    </p>
    </body>
```

定义Action的配置文件：

```
    <! ……文件名：
struts.xml……>
    <package name="C05.4"extends="strutsdefault">
    <action
name="dateTag"class="action.DateAction">
    <result name="success">/jsp/dateTag.jsp
</result>
    </action>
    </package>
```

定义日期的Action内容：

```
    <！……文件名：
DateAction.java……>
    public class DateAction extends ActionSupport {
    //定义北京奥运会开幕日期的日期属性
    private Date omplicDate;
    public String execute（）throws Exception {
    setOmplicDate（DateUtil.stringToDate（"20080808"
））；
    return SUCCESS;
    }
    public Date getOmplicDate（） {
    return omplicDate;
    }
    public void setOmplicDate（Date omplicDate） {
    this.omplicDate=omplicDate;
    }
    }
```

功能演示如图5.11所示。

图　5.11　date标签范例

源程序解读

（1）date标签的功能是用来在JSP等视图界面上显示日期，该日期的格式可以自己定义。其中id也是对需要显示的日期进行标识定义。name是日期变量名。

（2）format属性是让开发者自行定义这个需要显示的日期的显示格式。虽然Struts 2也提供了一个默认的日期显示格式，但是在实际开发中不一定符合具体情况，因此提供了自定义日期格式的功能，方便开发者开发。

注意：在代码中需要指明的是年、月、日的定义一定要按照“y”、“M”、“d”这样的字母大小写，这也是Java中日期格式定义中要求的。

（3）date标签还提供了一个比较特殊的属性nice，初一看很多人（包括刚开始学习Struts 2的笔者）都不知道这个属性是用来干嘛的。其实它是用来输出当前日期和需要显示的日期之间的时间差。特别是在开发中如果需要开发倒计时这样的功能时，该属性特别有用。如代码所示，在JSP页面上显示了离北京奥运会开幕还有多少时间的信息。

注意：nice也是一个布尔型的值，需要使用它的时也是显示定义为“true”。

5.4.4　debug标签使用介绍

技术要点

本节代码具体介绍debug数据标签的使用方式。

debug标签使用。

debug标签功能演示。

实现代码

使用debug标签的JSP文件：

```
    <! ……文件名：
debugTag.jsp……>
    <body>
    <s: debug/>
    <h3 align="left">
    debug标签使用范例
    </h3>
    ……
    </body>
```

功能演示如图5.12所示。单击“debug”链接按钮后页面显示如图5.13所示。

源程序解读

（1）debug标签的功能是给开发人员提供一个在视图上调试代码的功能。它没有具体属性可以介绍。

（2）仔细看图5.13，在该图中显示了值栈以及许多context中的对象、变量的使用情况。它是通过图5.12中那个“debug”链接展现出来的。其实图5.12和代码中也表明笔者只是在前面使用date标签的JSP代码中写上“＜s：debug/＞”，其他没有任何改动。如果开发者需要实时知道值栈和其他相关信息，debug标签是个很好的工具。

图　5.12　debug标签范例

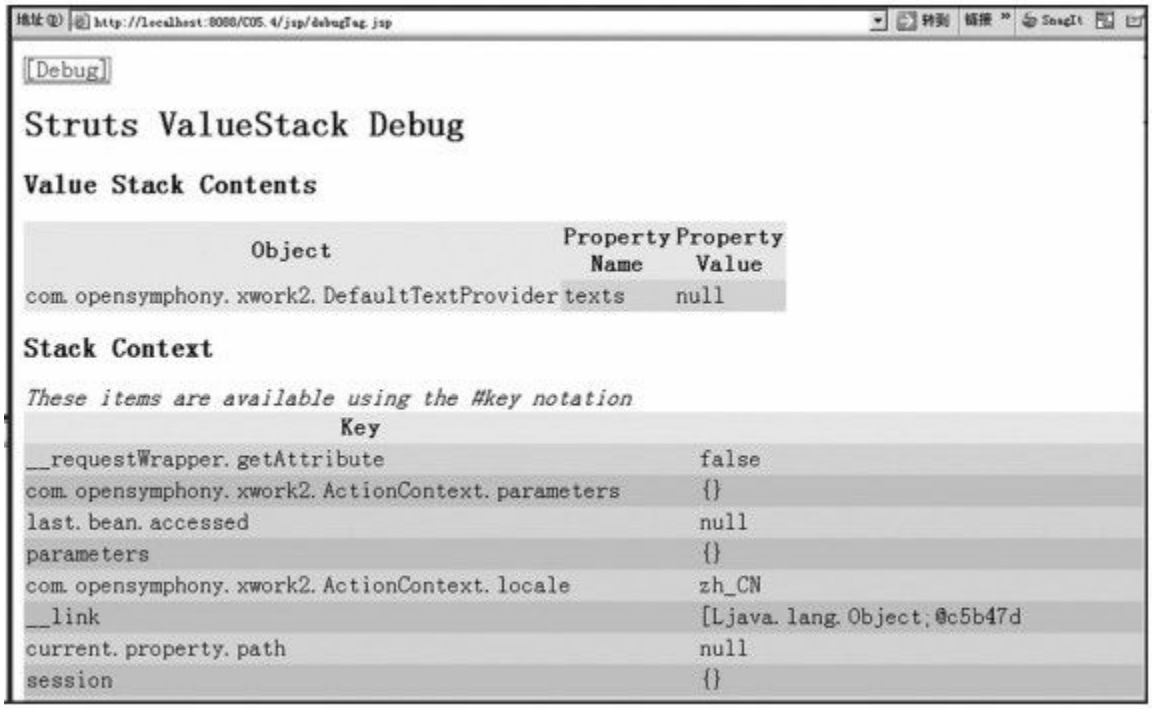

图　5.13　debug页面

5.4.5　include标签使用介绍

技术要点

本节代码具体介绍include数据标签的使用方式。

include标签使用。

include标签功能演示。

实现代码

使用include标签的JSP文件：

```
    <! ……文件名：
includeTag.jsp……>
    <body>
    <h3 align="left">
    include标签使用范例
    </h3>
    <h3 align="left">
    如下显示dateTag.jsp内容：
    </h3>
    <s: include value="dateTag.jsp"></s: include>
    </body>
```

定义Action的配置文件内容：

```
    <! ……文件名：
struts.xml……>
    <constant
name="struts.i18n.encoding"value="gb2312">
</constant>
    <package name="C05.4"extends="strutsdefault">
    <action
name="includeTag"class="action.DateAction">
    <result name="success">/jsp/includeTag.jsp
</result>
    </action>
    </package>
```

功能演示如图5.14所示。

图　5.14　include标签范例

源程序解读

（1）include标签的功能和JSP中的include功能相同，都是在一个页面包含另外一个页面文件显示的内容。是给开发人员提供一个在视图上调试代码的功能，它没有具体属性可以介绍。

（2）其实如果被包含的页面没有动态的数据，比如像HTTP请求中需要传递的参数，是可以在页面上直接使用include标签。这里为什么还在struts.xml中定义了一个Action，其目的是被包含的使用date标签的JSP文件有动态的需要显示的日期数据。

（3）include使用value属性来指定被包含的视图文件名，这里笔者还是include使用date标签的JSP文件。

注意：在struts.xml配置文件中又定义了字符编码属性，这是因为被include标签包含的页面的字符编码有可能和包含的页面的字符编码不一致，因为在这个示例中使用的是字符编码集为gb2312，因此显示声明了字符编码，使它们相一致。

5.4.6　push标签使用介绍

技术要点

本节代码具体介绍push数据标签的使用方式。

push标签使用。

push标签功能演示。

实现代码

使用push标签的JSP文件：

```
    <!……文件名：
pushTag.jsp……>
    <body>
    <s: bean name="model.Material"id="material">
    <s: param name="materialName"value="′进户门套油
漆′"/>
    <s: param name="mainbid"value="70"/>
    <s: param name="mount"value="200"/>
    </s: bean>
    <s: push value="material">
    <p>
```

```
<s: property value="materialName"/>
</p>
<p>
<s: property value="mainbid"/>
</p>
<p>
<s: property value="mount"/>
</p>
</s: push>
</body>
```

功能演示如图5.15所示。

图　5.15　push标签范例

源程序解读

（1）push标签的功能是将某个具体的值放在值栈的最顶处。id属性也是对需要放置在值栈栈顶的值的标识定义。name是该值的名字。

注意：被放置在值栈栈顶的值可能是一个字符串类型，也可能是一个Java对象实例。

（2）如代码所示，定义了材料这个Java对象，然后把它放置在值栈栈顶。然后通过property标签显示该对象的各个属性值。

5.4.7　set标签使用介绍

技术要点

本节代码具体介绍set数据标签的使用方式。

set标签使用。

set标签功能演示。

实现代码

使用set标签的JSP文件：

```
    <!……文件名：
setTag.jsp……>
    <body>
    <s: bean name="model.Material"id="material">
    <s: param name="materialName"value="′进户门套油
漆′"/>
    <s: param name="mainbid"value="70"/>
    <s: param name="mount"value="200"/>
    </s: bean>
    <s: set
name="anotherMaterial"value="#material"></s: set>
```

```
<p>
<s: property value="#material.materialName"/>
</p>
<p>
<s: property value="#material.mainbid"/>
</p>
<p>
<s: property value="#material.mount"/>
</p>
</body>
```

功能演示如图5.16所示。

图　5.16　set标签范例

源程序解读

（1）set标签的功能是对一个变量指定一个值。name属性是对该变量名字的定义。value属性就是这个给该变量指定的值。如果不指定，Struts 2是自动将

值栈中栈顶的值指定给它。还有scope属性，它表示的是类似session、request这样的范围定义。也就是该变量被指定值是在什么范围下才有效（request、session知识在第二章中有介绍）。如果不指定它则该变量是放在值栈的context中。

（2）如代码所示，还是使用之前定义的材料JavaBean。笔者定义了一个“anotherMaterial”变量，然后将material值指定给这个变量。

5.4.8　url标签使用介绍

技术要点

本节代码具体介绍url数据标签的使用方式。

url标签使用。

url标签功能演示。

实现代码

使用url标签的JSP文件：

```
    <!……文件名：
urlTag.jsp……>
    <body>
    <h3 align="left">
    url标签使用范例
    </h3>
    <h3 align="left">
    action由value属性指定，不显示全路径URL
    </h3>
    <p>
    <s: url value="actionTag.action">
```

```
    <s: param name="param"value="′ frank'"/>
    </s: url>
    </p>
    <h3 align="left">
    action由action属性指定，显示全路径URL
    </h3>
    <p>
    <s: url action="actionTag">
    <s: param name="param"value="′ frank'"/>
    </s: url>
    </p>
    <h3 align="left">
    value、action属性同时指定，以value指定为准即不显示全路径URL
    </h3>
    <p>
    <s: url
action="actionTag"value="actionTag.action">
    <s: param name="param"value="′ frank'"/>
    </s: url>
    </p>
    <h3 align="left">
    <p>value、action属性都不指定，则显示当前浏览器中URL内容。</p>
    <p>若有参数定义则URL后以"？"开头，使用"参数名=参数值"格式显示参数名和参数值</p>
    </h3>
    <p>
    <s: url includeParams="get">
    <s: param name="param"value="′ frank'"/>
    </s: url>
    </p>
    </body>
```

功能演示如图5.17所示。

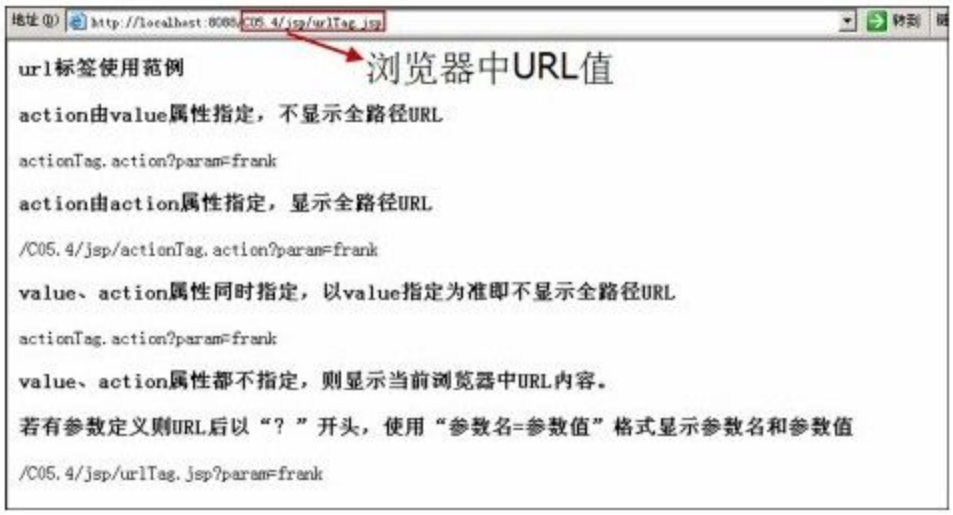

图　5.17　url标签范例

源程序解读

（1）url标签的功能是生成一个url地址。param作为它的子标签定义了需要传递到url的HTTP请求参数内容。它的属性比较多，笔者将自己认为最重要的几个属性通过代码示例介绍。

（2）如代码所示，value属性是指定生成的url地址，但是它只显示指定的Action或jsp，而并不是把url的全路径地址显示出来。而action属性则是指定url地址是哪一个action，然后将在struts.xml配置文

件中该action的result中指定的路径全部显示出来。如果同时显式定义action、value，则Struts 2是以value指定为准。如果都不显式定义，则以当前视图界面的url为指定生成的url地址。

图5.17中红框选中的正是该url，可以从图中可知和最后一行打印出来的信息是相同的，并且通过includeParams属性将定义的参数值也显示出来。

（3）includeParams属性是表明是否包含HTTP请求的参数，它的内容只能是none、get、all这三个。如代码所示，笔者定义的是get该参数的值在图5.17中的几个小例子都已经显示出来。

5.4.9 param标签和property标签使用介绍

param标签在前面这些标签介绍的代码中都有介绍，它可以作为很多标签的子标签来使用，当它作为子标签使用时的功能是视每个标签的具体功能而定。

property标签和上一小节介绍的param类似，前面的示例中都有代码演示。不再列具体代码和图例。

5.5 Struts 2表单标签简介

如果有对HTML比较熟悉的读者，肯定知道诸如select、checkbox这些HTML标签的使用方式。这些标签其实是可以归类为表单标签。在Struts 2中除了这些基本的HTML标签的定义外，它还定义了许多特殊的但又基于前述的HTML表单标签的个性化标签。这些表单标签各具特色，是很值得介绍的。在开发中负责view层功能开发的开发人员如果使用这些标签，就能发现Struts 2提供了多么丰富的view层功能。

5.5.1 基础表单标签使用介绍

技术要点

本节代码具体介绍各个基础表单标签的使用方式。这些标签其实都可以用HTML的表单标签来实现。

读者可以了解Struts 2的这部分表单标签和原有HTML表单标签的实现功能几乎是相同的。

基础表单标签使用。

基础表单标签功能演示。

实现代码

基础表单标签的JSP文件：

```
    <！……文件名：basicFormTag.jsp……>
    <%@page language="java"contentType="text/html;charset=gb2312"
    pageEncoding="gb2312"%>
    <%@taglib prefix="s"uri="/strutstags"%>
    ……
    <body>
    <h3 align="left">
    基础表单标签使用范例（可使用HTML标签替代）
    </h3>
    <h3 align="left">
    checkbox标签使用范例
    </h3>
    <p>
    <s：checkbox label="别墅"name="bieshu"value="true"/>
```

```
    <s: checkbox label="公寓"name="gongyu"/>
    </p>
    <h3 align="left">
    checkboxlist标签使用范例
    </h3>
    <p>
    <s: checkboxlist label="材料"list="{' 进户门套油
漆',' 踢脚线',' 大理石',' 吊顶'}"
    name="material"/>
    </p>
    <h3 align="left">
    file标签使用范例
    </h3>
    <p>
    <s: file name="file"accept="text/"/>
    <s: file name="file"accept="image/jpeg"></s:
file>
    </p>
    <h3 align="left">
    select标签使用范例
    </h3>
    <p>
    <s: select label="下拉框示
例"name="material"headerKey="0"headerValue=""
    list="#{' 01' : ' 进户门套油漆',' 02' : ' 踢脚线
',' 03' : ' 大理石',' 04' : ' 吊顶'}"/>
    </p>
    <h3 align="left">
    optgroup标签使用范例
    </h3>
    <p>
    <s: select label="示
例"name="material"headerKey="0"headerValue=""
    list="#{' 01' : ' 进户门套油漆',' 02' : ' 踢脚线
',' 03' : ' 大理石',' 04' : ' 吊顶'}">
    <s: optgroup label="材料品牌"
```

```
    list="#{'01':'多乐士超易洗','02':'绿太阳
'}"/>
    </s:select>
    </p>
    <h3 align="left">
    radio标签使用范例
    </h3>
    <p>
    <s:radio label="材料"list="{'进户门套油漆','
踢脚线','大理石','吊顶'}"
    name="material"/>
    </p>
    <h3 align="left">
    textarea标签使用范例
    </h3>
    <p>
    <s:textarea label="输入文
本"labelposition="left"name="textarea"cols="40"rows
="
    10"/>
    </p>
    <h3 align="left">
    textfield标签使用范例
    </h3>
    <p>
    <s:textfield label="输入文
本"name="textfield"size="40"maxlength="10"/>
    </p>
    <h3 align="left">
    password标签使用范例
    </h3>
    <p>
    <s:password label="输入密
码"name="password"size="20"maxlength="8"/>
    </p>
    </body>
    </html>
```

功能演示如图5.18和图5.19所示。

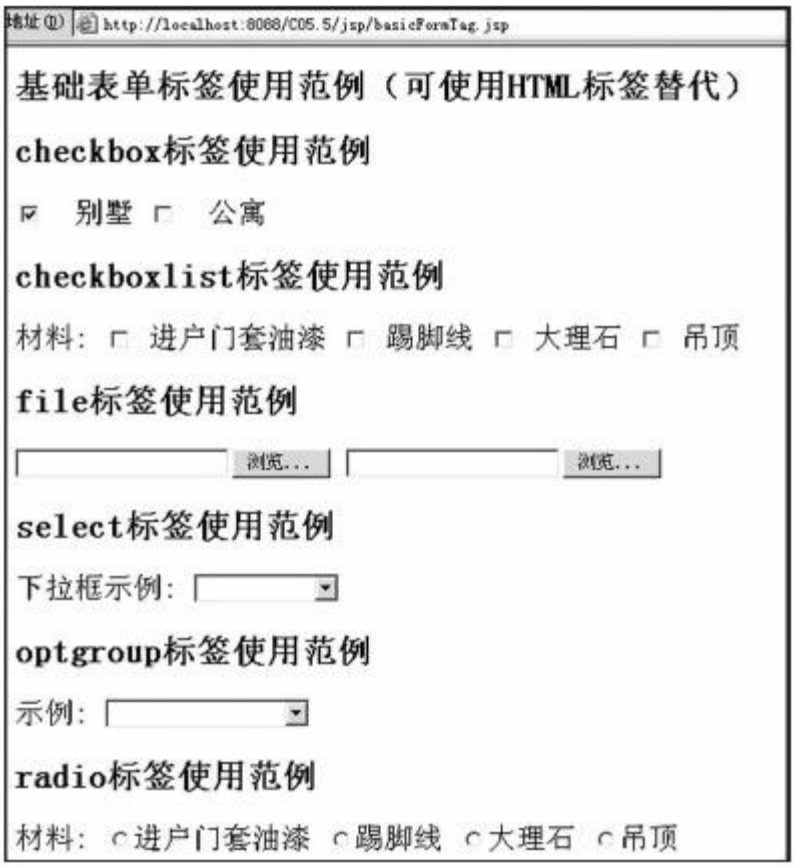

图　5.18　基础表单标签范例1

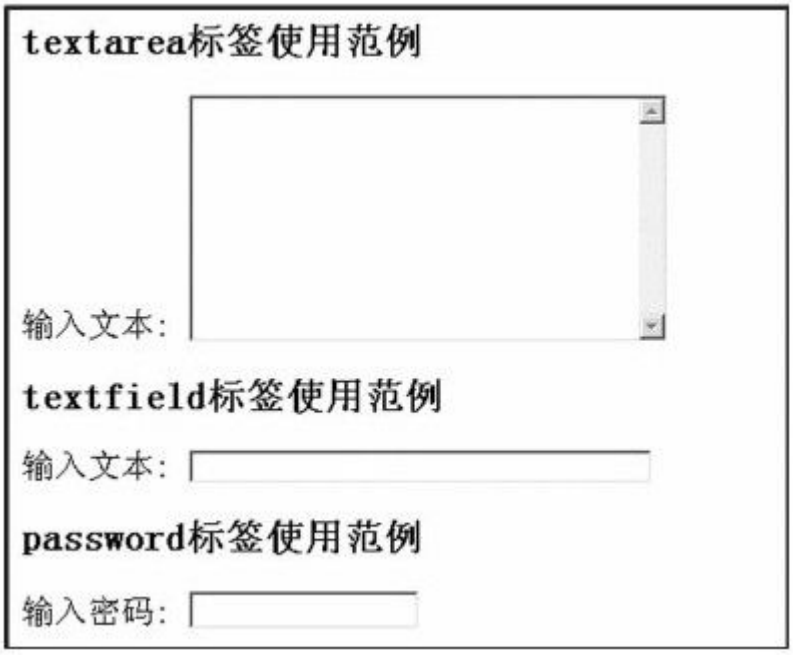

图　5.19　基础表单标签范例2

源程序解读

（1）checkbox标签的功能是显示一个可以让用户选择的复选框。其中label属性是显示复选框在页面上的选择项名字。name是定义该复选框的名字。而value属性则是表示是否选中。其中如果为“true”则表示被选中，在页面上该复选框中则会以“√”显示。反之则复选框中为空。

（2）checkboxlist标签的功能是根据一个数据集合来显示多个可以让用户选择的复选框。其中label属性是显示复选框在页面上的选择项名字。List是来指定数据集合，它实际上是一个map类型的数据集合。

默认情况下，它的key就赋值给value属性，它自己的value则对应页面上显示的复选框内容。在代码中笔者向读者表示了基本的checkboxlist用法。它还有两个可以不写的属性，一个是listKey，它指定了集合中哪个属性作为复选框的value。另外一个属性为listValue，该属性指定集合中哪个属性作为在页面上显示的复选框内容。

（3）file标签的功能是显示文件上传的输入框。该标签在前面文件上传下载章节的代码中有过记述。这里重新记述，除了让读者加深理解之外，还向读者记述它的accept属性。该属性限定了上传的文件类

型，文件类型的表示在前面章节也已说明。如果读者在file标签中上传了非accept属性指定的文件类型，则表单提交时会报错。

（4）select标签的功能是提供一个下拉框。其中list属性也是指定一个数据集合，以map或list类型显示。如果是map类型，则key和value可以显示指定为下拉框中每个元素的值。集合中的数据就是下拉框中的内容。headerKey和headerValue属性是表明下拉框缺省显示的值和内容。listKey和listValue属性则和checkboxlist标签中的属性相同，都是表明值和在页面上显示的内容。它还有一个multiple属性，代码中没有显示给读者看，它是表明下拉框中内容是否可以多选即同时选中下拉框中的多项元素。

（5）optgroup标签的功能是配合select标签，在下拉框中显示一个选项组。它在页面上的显示和

select标签相同。所不同的是下拉框中选择内容的显示，如图5.20和图5.21所示。

图　5.20　select标签下拉框内容

图　5.21　select和optgroup标签联合使用中下拉框内容

它的list属性其实就是指定了选择组中的内容。在该示例中就是显示了“材料品牌”这个选项组。选中“材料品牌”即表明下面两个品牌内容都被选中。

（6）radio标签的功能是显示一个单选框。它的属性和checkboxlist标签属性相同。

（7）textarea标签的功能是显示一个文本输入框。它的label和name属性和之前标签的属性相同。它还有rows和cols两个属性，这两个属性表示该文本输入框的行数和每行允许显示字数的多少。

（8）textfield标签的功能是显示一个单行文本输入框。在之前章节有很多代码示例，这里不再赘述。

（9）password标签的功能是显示一个输入密码的文本输入框。它输入的文字都不会显式的显示在页面上，而是用“*”来显示。

5.5.2　复杂表单标签使用介绍

技术要点

本节代码具体介绍各个复杂表单标签的使用方式。这些标签在不使用Struts 2的情况下，都是用基础表单标签和JavaScript代码联合起来使用，才能达到这些标签的效果。

复杂表单标签使用。

复杂表单标签功能演示。

实现代码

使用doubleselect和optiontransferselect标签需要显示的材料类别类文件代码：

```
    <!……文件名：Item.java……>
```

```
    public class Item implements
java.io.Serializable {
    private int itemId; //材料类别ID
    private String item; //材料类别名称
    ……
    public int getItemId（） {//获取或设置ID
    return itemId;
    }
    public void setItemId（int itemId） {
    this.itemId=itemId;
    }
    public String getItem（） {//获取或设置材料类别
    return this.item;
    }
    public void setItem（String item） {
    this.item=item;
    }
    }
```

使用doubleselect和optiontransferselect标签

需要显示的材料类文件代码：

```
    <！……文件名：
Material.java……>
    public class Material implements
java.io.Serializable {
    private int materialId; //材料ID
    private int itemid; //类别ID
    private String material; //材料名称
    ……
    public int getMaterialId（） {//获取材料ID
    return this.materialId;
    }
```

```
    public void setMaterialId（int materialId）｛//
设置材料ID
    this.materialId=materialId;
    }
    ……
    public String getMaterial（）｛//获取或设置材料名称
    return this.material;
    }
    public void setMaterial（String material）｛
    this.material=material;
    }
    }
```

使用doubleselect和optiontransferselect标签的Action文件代码：

```
    ＜！……文件名：
ComplexFormTagAction.java……＞
    public class ComplexFormTagAction extends
ActionSupport｛
    private List＜Item＞itemList；//级联第一个下拉框数
据
    private Map＜Integer，List＜Material＞＞
materialMap；//级联第二个下拉框数据
    private String[]leftMaterials；//左边材料数据
    private String[]rightMaterials；//右边材料数据
    public String execute（）throws Exception｛
    itemList=new ArrayList＜Item＞（）；
    //循环新建10个类别
    for（int j=0；j＜10；j++）｛
    Item item=new Item（）；
    item.setItemId（j+1）；
    item.setItem（“类别”+（j+1））；
```

```
    itemList.add（item）；
    }
    materialMap=new HashMap＜Integer, List＜Material
＞＞（）；
    //循环新建每个类别中的10个材料
    for（int j=0；j＜10；j++）｛
    List＜Material＞materialList=new ArrayList＜
Material＞（）；
    for（int i=0；i＜10；i++）｛
    Material material=new Material（）；
    material.setItemid（i+1）；
    material.setMaterialId（i）；
    material.setMaterial（"类别"+（j+1）+"＞"+"材料"+
（i+1））；
    materialList.add（material）；
    }
    materialMap.put（（j+1），materialList）；
    }
    int tempCount；
    ArrayList＜Material＞leftList=new ArrayList＜
Material＞（）；
    for（int i=0；i＜10；i++）｛
    Material leftMaterial=new Material（）；
    leftMaterial.setMaterialId（i+1）；
    leftMaterial.setMaterial（"材料"+（i+1））；
    leftList.add（leftMaterial）；
    }
    tempCount=10；
    ArrayList＜Material＞rightList=new ArrayList＜
Material＞（）；
    for（int i=0；i＜10；i++）｛
    Material rightMaterial=new Material（）；
    rightMaterial.setMaterialId（i+1）；
    rightMaterial.setMaterial（"材料"+
（tempCount+i+1））；
    rightList.add（rightMaterial）；
    }
```

```
    getRequest（）.setAttribute（"rightList",
rightList）;
    getRequest（）.setAttribute（"leftList",
leftList）;
    return SUCCESS;
    }
    private HttpServletRequest getRequest（） {
    return ServletActionContext.getRequest（）;
    }
    public Map＜Integer, List＜Material＞＞
getMaterialMap（） {
    return materialMap;
    }
    public void setMaterialMap（Map＜Integer, List＜
Material＞＞materialMap） {
    this.materialMap=materialMap;
    }
    ……
    public String[]getRightMaterials（） {
    return rightMaterials;
    }
    public void
setRightMaterials（String[]rightMaterials） {
    this.rightMaterials=rightMaterials;
    }
    }
```

复杂表单标签的JSP文件：

```
    ＜!……文件名：
complexFormTag.jsp……＞
    ＜%@taglib prefix="s"uri="/strutstags"%＞
    ＜html xmlns="http: //www.w3.org/1999/xhtml"＞
    ＜head＞
```

```
    <title>复杂表单标签使用范例</title>
    <s: head/>
    </head>
    <body>
    <h3 align="left">
    复杂表单标签使用范例（不使用Struts 2时需要结合
JavaScript和HTML标签共同开发的功能）
    </h3>
    <h3 align="left">
    combobox标签使用范例
    </h3>
    <p>
    <s: combobox label="材
料"name="material"readonly="false"
    headerValue=""headerKey="0"list="{' 进户门套油漆
' , ' 踢脚线' , ' 大理石' , ' 吊顶' } "/>
    </p>
    <h3 align="left">
    doubleselect标签使用范例
    </h3>
    <p>
    <s: form name="doubleselectExample">
    <s: doubleselect label="材
料"headerValue=""headerKey="0"
    list="itemList"listKey="itemId"listValue="item"
    doubleName="materialId"doubleList="materialMap.
get（top.itemId）"
    doubleListKey="materialId"doubleListValue="mate
rial"/>
    </s: form>
    </p>
    <h3 align="left">
    datetimepicker标签使用范例
    </h3>
    <p>
    <s: datetimepicker label="日
历"name="calendar"value="today"
```

```
    toggleType="plain"toggleDuration="300"language=
"zh_CN"type="date"
    displayWeeks="5"displayFormat="dd/MM/yyyy"forma
tLength="long"/>
    </p>
    <h3 align="left">
    optiontransferselect标签使用范例
    </h3>
    <p>
    <s: optiontransferselect label="材
料"name="leftMaterials"
    leftTitle="已选择材
料"list="#request.leftList"listKey="materialId"
    listValue="material"multiple="true"headerKey="h
eaderKey"
    headerValue="请选
择"emptyOption="false"allowUpDownOnLeft="true"
    rightTitle="未选择材
料"doubleList="#request.rightList"
    doubleListKey="materialId"doubleListValue="mate
rial"
    doubleName="rightMaterials"doubleHeaderKey="dou
bleHeaderKey"
    doubleHeaderValue="请选
择"doubleEmptyOption="false"
    doubleMultiple="true"allowUpDownOnRight="true"/
>
    </p>
    <h3 align="left">
    updownselect标签使用范例
    </h3>
    <p>
    <s: form name="updownselectExample">
    <s: updownselect
    list="#{'01':'进户门套油漆','02':'踢脚线
','03':'大理石','04':'吊顶'}"
    name="material"
```

```
    headerKey="-1"
    headerValue="选择"
    moveUpLabel="up"moveDownLabel="down"selectAllLa
bel="all"
    />
    </s: form>
    </p>
    </body>
    </html>
```

功能演示如图5.22～图5.24所示。

图　5.22　复杂表单标签范例1

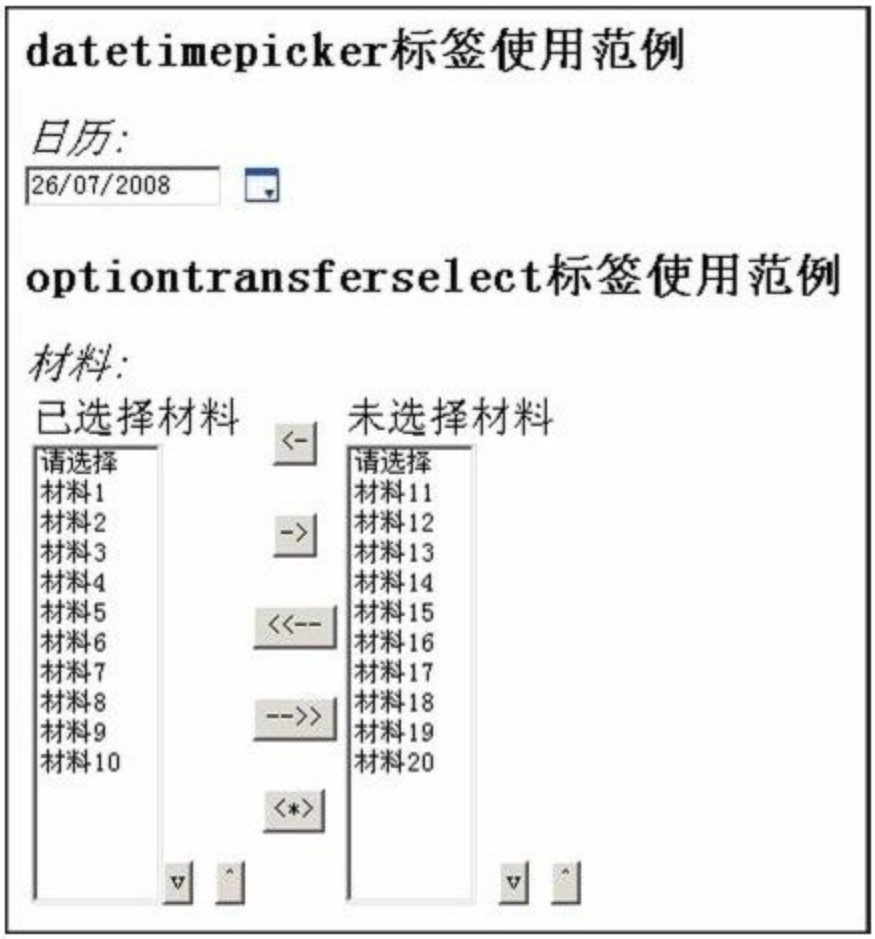

图　5.23　复杂表单标签范例2

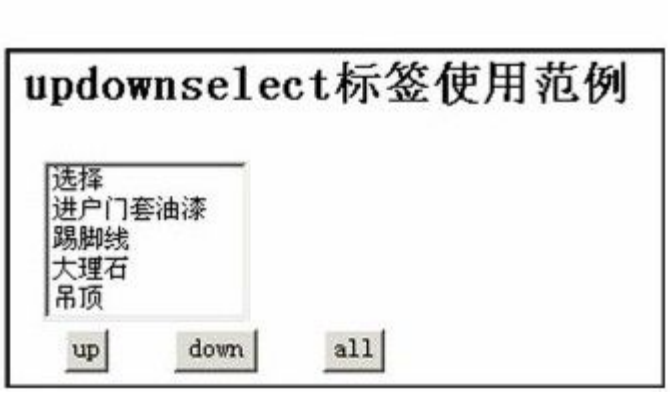

图　5.24　复杂表单标签范例3

源程序解读

（1）combobox标签的功能是显示一个可以让用户选择的复选框，和一个可以让用户输入的文本输入框组合。这两个表单标签在HTTP请求中同时被指定同一个请求参数。list属性是指定一个下拉框选择内容的数据集合。headerValue和headerKey属性和前述的标签中属性功能相同。该标签具有一个readonly属性，它是表明文本输入框中是否允许输入文本，如果readonly属性值为“true”则不允许用户输入，只能查看该文本输入框中显示的内容。

（2）doubleselect标签的功能是提供两个有级联关系的下拉框。用户选中第一个下拉框中的某选项，则第二个下拉框中的选项根据第一个下拉框被选中的某选项内容来决定它自己的下拉框选项内容。

由于该标签名字中包含“double”，因此它只提供两个级联关系的下拉框。如果开发者想利用该标签开发多个级联关系下拉框，则还需要自己编写JavaScript或Java代码联合使用该标签开发。笔者很希望Struts 2的设计者在以后的版本中能提供一个multiselect标签，这样开发者就可以直接使用该标签来快速开发多个级联关系下拉框。

该标签属性有很多。其中list属性是指定第一个下拉框中选项的数据集合。listKey和listValue属性是分别指定第一个下拉框中的value和页面上显示的内容。doubleList是指定第二个下拉框中选项的数据集合。doubleListKey和doubleListValue属性是分别指定第二个下拉框中的value和页面上显示的内容。doubleName是第二个下拉框的名字属性。其他诸如headerKey和headerValue属性都和之前讲述标签中的属性相同。

（3）在本示例中，笔者先定义了两个有级联关系的Java类，一个是材料类别，另一个是材料。每个材料类别分别拥有多种材料。因此第一个下拉框中的选项都是类别数据，第二个下拉框中的选项都是材料数据。用户选中一个类别，则相应的第二个下拉框根据这个类别显示该类别中所包含的所有材料选项。如图5.22所示，选中“类别2”，然后第二个下拉框中材料的选项都是“类别2”的材料，然后再选中“材料5”。在Action代码中，笔者也利用循环新建了10个类别和每个类别拥有的10个材料。然后将类别放入一个list类型数据集合中，材料放在map类型数据集合中，该map集合的key就是类别的id。然后在JSP中利用OGNL表达式配合doubleselect标签来将这两个集合的数据在级联下拉框中显示。

注意：JSP中有“materialMap.get（top.itemId）”这行代码。其中

的top其实返回的是材料类别map集合的value即材料list集合中的材料类对象实例，而不是材料list集合。而且在使用该标签时，该标签一定要包含在form表单中，否则功能不会实现。

（4）datetimepicker标签的功能是显示一个可以选择时间日期的选择框。如果不使用Struts 2，以前都是用JavaScript来完成这个选择框实现工作。现在Struts 2底层已经集成了该选择框，开发者只要使用该标签就能完成开发工作，节省了大量时间。

该标签属性也很多。value属性可以指定具体的时间日期，也可以用“today”表示当前日期。type属性只能是“date”和“time”，分别表示是日期选择还是时间选择。language属性是指定选择的国家文字类型，比如在该示例中的“zh_CN”表明显示的是简体中文。displayWeeks属性表明日历显示的星期数，一

般如果要把一个月的日期都显示，需要显示星期数为5，所以该示例中笔者对该属性的值是赋为5。

（5）displayFormat属性指定日期显示格式。formateLength属性则表明显示的日期格式类型，它一共有四个可以赋给它的值，分别为long、short、medium、full。toggleType和toggleDuration属性都是和日期选择框有关。toggleType属性是指定日期选择框的显示、关闭方式。它也有四个可以赋的值，分别是plain、wipe、explode、fade。示例中笔者用的是plain。而toggleDuration属性是指定日期选择框显示、关闭的时间间隔，单位是毫秒。示例中定义的是300即日期选择框显示、关闭时间是300毫秒。

如图5.25所示是日期选择框显示的效果。读者可以看到显示了2008年7月的日历，白色显示的26号即笔

者写作这章的当前日期。选择框最下方是可以让用户选择的纪年。

图　5.25　plain型的日期选择框显示图

注意：在使用该标签时，JSP中一定要显示定义在下一小节中将会介绍的head标签。因为head标签是调用了某些AJAX的框架dojo定义的文件，该标签显示的日期格式等其实都是由dojo来实现的。所以不使用head标签，该标签就不能在页面上正常显示日期选择。

（6）optiontransferselect标签的功能是提供两个可以左右转移以及下拉的列表项。该标签中提供了很多上下左右移动选择项的功能。而且还可以进行多项选择项移动。以前笔者曾经开发过这样的页面组件，当时也是使用JavaScript和基础HTML标签类开发的。用了两个工作日才完成这样的页面组件。如果使用Struts 2的话，就可以不需要浪费这么多时间仅仅完成一个页面组件，而忽略了业务逻辑功能的开发。

（7）在本示例中，笔者也是在Action中新建了两个材料数据集合。这两个集合的元素都是字符串类型的。将这两个集合放置在request中，然后在JSP中配合OGNL来实现该标签。

因为该标签在页面上的显示是分成左右两部分，笔者先介绍设置左边选择列表的属性。其中leftTitle属性是指定左边列表标题。其中list属性是指定左边

列表中被选择的数据集合，以map或list类型显示。如果是map类型，则key和value可以显示指定为左边列表中每个元素的值。集合中数据就是左边列表中的内容。headerKey和headerValue属性是表明左边列表缺省显示的值和内容。

（8）listKey和listValue属性则和checkboxlist标签中的相同，都是表明左边列表值和在页面上显示的内容。它还有一个multiple属性表明左边列表中内容是否可以多选即同时选中左边列表中的多项元素。allowUpDownOnLeft属性是指定左边列表中各个元素是否可以上下移动，如果值为“true”就表明可以上下移动，反之则否。emptyOption属性是指定左边列表中是否有空元素出现即没有任何文字的选项。如果值为“true”就表明有空元素出现，反之则否。

（9）rightTitle属性是指定右边列表标题。其中doubleList属性是指定右边列表中被选择的数据集合，以map或list类型显示。如果是map类型，则key和value可以显示指定为右边列表中每个元素的值。集合中的数据就是右边列表中的内容。doubleHeaderKey和doubleHeaderValue属性是表明右边列表缺省显示的值和内容。doubleListKey和doubleListValue属性则和前面段落中listKey和listValue属性相同，只不过是表明右边列表值和在页面上显示的内容。它还有一个doubleMultiple属性表明右边列表中内容是否可以多选即同时选中右边列表中的多项元素。

allowUpDownOnRight属性是指定右边列表中各个元素是否可以上下移动，如果值为“true”就表明可以上下移动，反之则否。doubleEmptyOption属性是指定右边列表中是否有空元素出现即没有任何文字的选

项。如果值为“true”就表明有空元素出现，反之则否。

（10）updownselect标签的功能是显示选项可以上下移动的选择列表框。它的list属性也是指定选择列表的选择项数据集合。moveUpLabel属性是显示上移按钮。moveDownLabel属性是显示下移按钮。selectAllLabel属性是显示全选按钮，用来全部选择列表中的选择项。上述三个属性可以定义为自己想显示在页面上的按钮文字内容。如果不显式定义它们，则上移、下移、全选按钮文字内容默认显示为“∨”、“∧”、“*”。

该标签还有三个属性，它们是allowMoveUp、allowMoveDown、allowSelectAll。默认都为“true”，因此除非不想让选择列表中的选项被上

移、下移和全选，可以显式定义他们为“false”，否则不会被显式定义。

注意：在使用该标签时，该标签一定要包含在form表单中，否则功能不会实现。

5.5.3　其他表单标签使用介绍

技术要点

本节代码具体介绍一些很难分类的表单标签的使用方式。

其他表单标签使用。

其他表单标签功能演示。

实现代码

其他表单标签的JSP文件：

```
    <!……文件名：anotherFormTag.jsp……>
    <%@taglib prefix="s"uri="/strutstags"%>
    <body>
    ……
    <h3 align="left">
    head标签使用范例
    </h3>
```

```
<s: head/>
<h3 align="left">
token标签使用范例
</h3>
<s: token/>
<h3 align="left">
hidden标签使用范例
</h3>
<s: hidden name="frank"/>
<s: hidden name="frank"value="frank"/>
</body>
```

由于这些标签在页面上看不出实现效果，可单击鼠标右键，在弹出菜单中选择“查看源文件”命令来查看。具体代码的实现如图5.26所示。

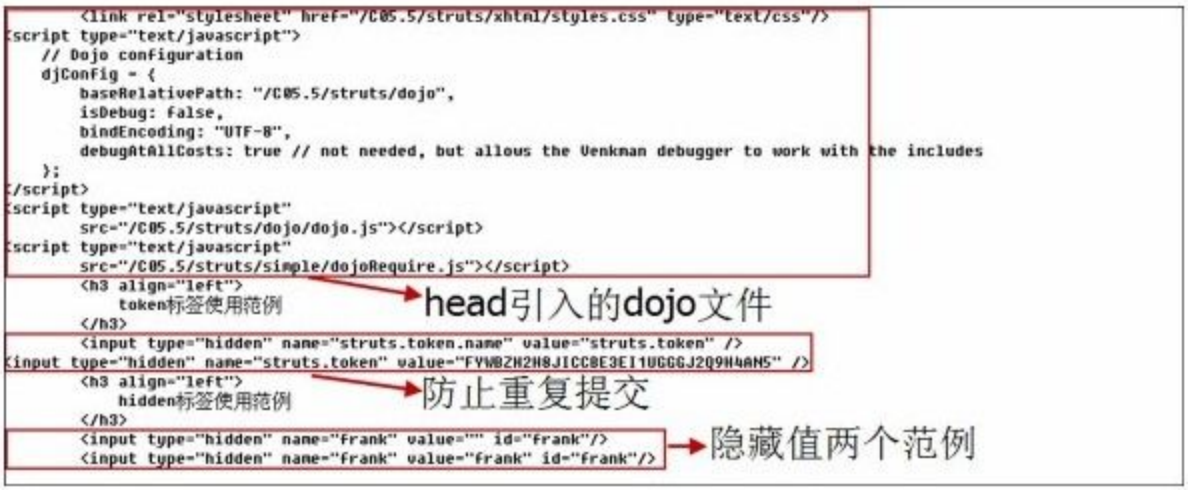

图　5.26　其他表单标签范例

源程序解读

（1）head标签的功能是生成页面文件的HEAD部分。但是它和一般的JSP、HTML页面文件HEAD部分又不同。它生成了对Ajax框架dojo的配置文件的引用或代码。具体都是有关CSS和JavaScript文件引用或代码。图5.26中第一个红框范围内显示内容就是这些文件引用和代码演示。在上一小节中有关日期选择的标签中也着重说明过。请读者仔细体会，有兴趣的读者也可以参考dojo的文档资料。

（2）token标签的功能是防止重复提交一个表单数据。有时一些用户会有恶意提交的现象（黑客？），使用该标签就是为了避免这种现象发生。

其实在前面讲述拦截器的章节中也介绍过Struts 2一些缺省的拦截器。该标签就是利用了TokenInterceptor或TokenSessionStoreInterceptor拦截器。该标签在页面中增加了一个隐藏值，每次提

交时该值应该是不同的，如果上述两个拦截器拦截HTTP请求时发现两次隐藏值相同，则判断有人重复提交同一界面同一表单，拦截器就不允许表单提交。图5.26中第二个红框范围内代码也表明了防止重复提交的该原理。

（3）hidden标签的功能是在表单中增加一个隐藏输入的值域。图5.26中第三个红框范围内代码表明了两个示例。这两个示例不同之处在于一个value属性被赋予特定的值，另一个则没有显式定义value属性的值。由代码可知如果没有显式定义value属性，则value属性默认值为空。

5.6 Struts 2非表单标签简介

除了表单标签之外，Struts 2也提供了很多显示可视化控件的标签，甚至还允许开发人员自定义自己的控件标签。因为这些标签分类比较难，所以又通通归类为非表单标签。

5.6.1 主题和模板介绍

技术要点

在介绍非表单标签前，有必要介绍Struts 2中主题和模板这两个概念，因为很多非表单标签都涉及Ajax技术或除了JSP视图外freemarker、velocity视图技术。在Struts 2中定义了一些主题和模板来方便开发者使用非表单标签，而又无缝的结合上述几种技术。

主题和模板使用。

Struts 2中主题和模板使用原理。

实现代码

使用主题的JSP文件：

```
    <!……文件名：
login.jsp……>
    <%@taglib prefix="s"uri="/strutstags"%>
    <html>
    <head>
    <title>登录页面</title>
    <s: head/>
    </head>
    <body>
    <s: form action="Login"theme="simple">
    <table width="60%"height="76"border="0">
    <s: textfield name="username"label="用户名"/>
    <s: password name="password"label="密码"/>
    <s: submit value="登录"align="center"/>
    </table>
    </s: form>
    <s: form action="Login">
    <table width="60%"height="76"border="0">
    <s: textfield name="username"label="用户名"/>
    <s: password name="password"label="密码"/>
    <s: submit value="登录"align="center"/>
    </table>
    </s: form>
```

```
<s: form action="Login"theme="css_xhtml">
<table width="60%"height="76"border="0">
<s: textfield name="username"label="用户名"/>
<s: password name="password"label="密码"/>
<s: submit value="登录"align="center"/>
</table>
</s: form>
<s: form action="Login"theme="ajax">
<table width="60%"height="76"border="0">
<s: textfield name="username"label="用户名"/>
<s: password name="password"label="密码"/>
<s: submit value="登录"align="center"/>
</table>
</s: form>
</body>
</html>
```

使用各主题的页面效果如图5.27所示。

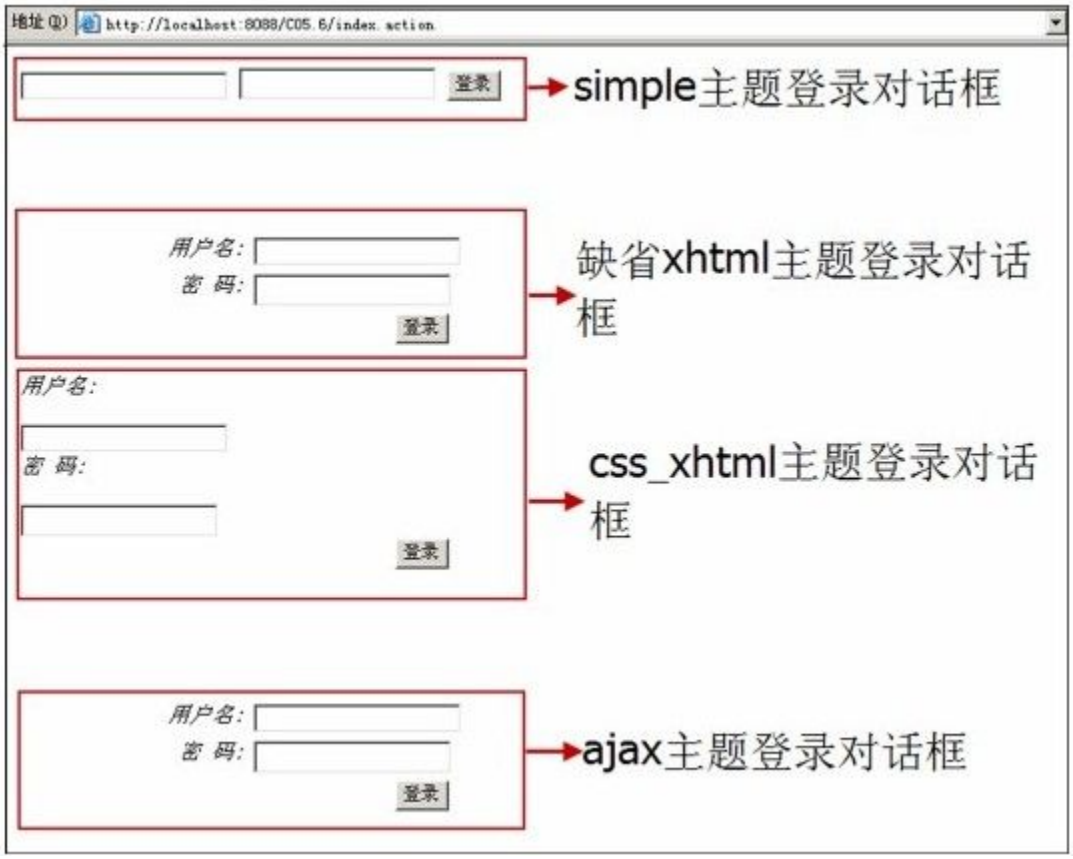

图　5.27　主题使用范例

源程序解读

（1）在Struts 2框架中提供了四种主题。分别为“simple”、“xhtml”、“css_xhtml”、“ajax”。其中“xhtml”主题是缺省显示的。即不显示定义theme属性，Struts 2默认为“xhtml”主题。在JSP中可以像示例中的代码所示使用主题。除了像代

码中所示在form标签中使用sheme属性外。还可以在各个标签中使用theme属性。

注意：theme属性是按照特定顺序来确定主题的。

界面标签（Struts 2中除了数据和控制标签之外其他标签都可归类为界面标签，也称之为UI标签）中使用theme属性指定主题。

表单标签中使用theme属性指定主题。

page范围内命名中使用theme属性指定主题。

request范围内命名中使用theme属性指定主题。

session范围内命名中使用theme属性指定主题。

application范围内命名中使用theme属性指定主题。

struts.propertie或struts.xml文件中定义struts.uti.theme常量来指定主题。

页面中定义主题按照上述顺序来定义主题，如果在上述中同时定义多个主题。则以第1个为准。比如界面标签中有theme属性定义，表单标签中又有theme属性定义。则以界面标签中定义的theme属性为准。

（2）simple主题只是很简单地生成一些HTML的基本元素。从图5.27中可以看出没有任何文字说明，只是提供一些基础的文本框、按钮。

（3）xhtml主题是默认缺省显示的。它除了simple主题一些功能的显示之外，还提供了错误校验、文字等附加功能。

（4）css__xhtml主题其实和xhtml主题功能类似。只是又增加了css的一些控制。

（5）ajax主题是在前三个主题上又做了深度扩展。它以ajax框架dojo为基础，增加了很多有关ajax的功能。

注意：使用ajax主题的JSP页面也需要定义head标签。原理之前也已讲述，可参见前面章节中对head标签的解释说明。

（6）模板其实是主题中一个个子元素。即多个模板构成1个主题。在Struts 2的源代码包中可以清晰看见主题和模板的关系。比如在“struts 2core2.0.11.1\template”路径下有所有主题的目录，单击任一目录，可看见一些后缀名为ftl文件，如图5.28所示。

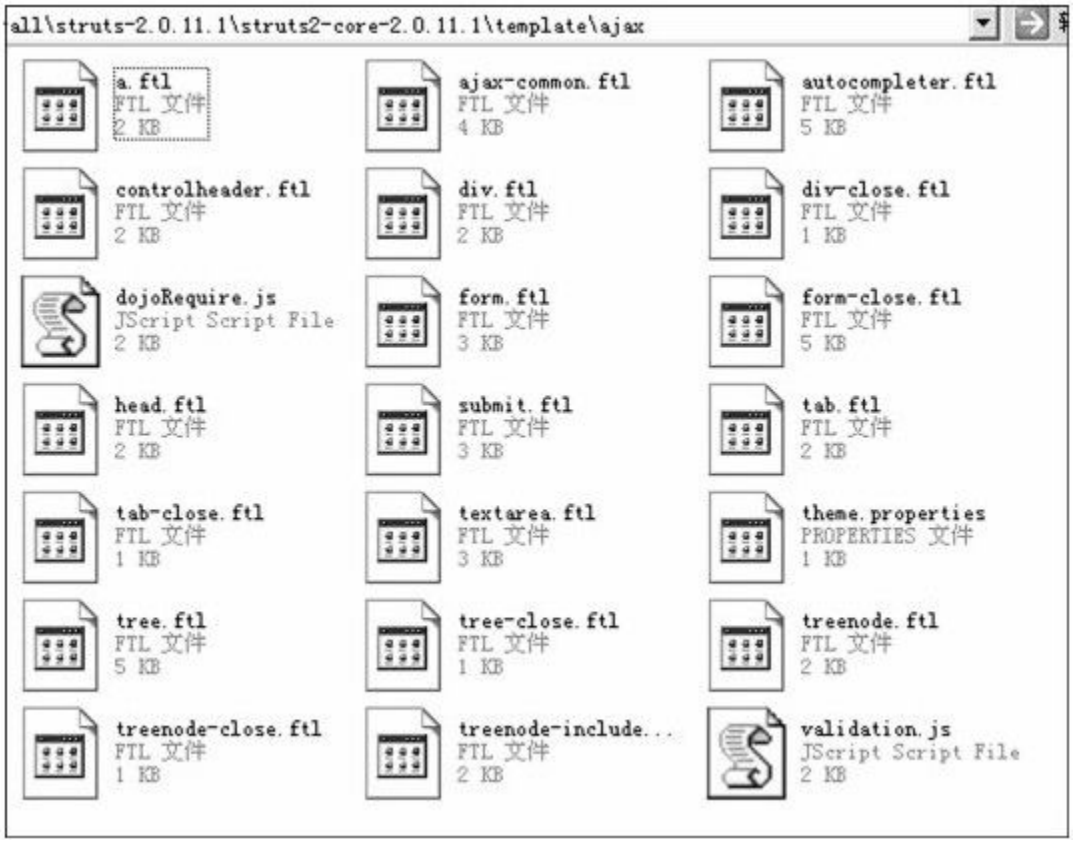

图　5.28　ajax目录下的模板文件

（7）由于Struts 2默认模板基于FreeMarker视图技术，而它的模板文件都是以“ftl”为后缀名。因此可以理解为模板文件默认是FreeMarker视图技术。除了FreeMarker这个默认视图技术之外，在第1章介绍过Struts 2还可以使用velocity和JSP视图技术。如果读

者想使用这两种视图技术，则可以使用后缀名为vm和jsp的模板文件。

通常可以在struts.propertie或struts.xml文件中定义struts.uti.templateSuffix常量来制定使用这三种中的哪一个视图技术。赋予该常量的值就是三种视图技术的模板文件后缀名：ftl、vm和jsp。

5.6.2　非表单标签介绍

技术要点

本节代码具体介绍一些非表单标签的使用方式。

非表单标签使用。

非表单标签功能演示。

实现代码

使用actionError和actionMessage标签的JSP文件：

```
    <!……文件名：
ErrorAndMessage.jsp……>
    <%@taglib prefix="s"uri="/strutstags"%>
    ……
    <h3 align="left">
    actionerror标签使用范例
    </h3>
    <p>
```

```
<s: actionerror/>
</p>
<h3 align="left">
actionmessage标签使用范例
</h3>
<p>
<s: actionmessage/>
</p>
……
```

使用actionError和actionMessage标签的Action文件：

```
<！……文件名：ErrorAndMessageAction.java……>
public class ErrorAndMessageAction extends ActionSupport {
public String execute（）throws Exception {
//调用Struts 2API，设置error和Message信息
addActionError（"Action的错误信息"）；
addActionMessage（"Action的消息信息"）；
return SUCCESS;
}
}
```

使用tree和treenode标签的JSP文件：

```
<！……文件名：tree.jsp……>
<%@taglib prefix="s"uri="/strutstags"%>
```

```
    ……
    <head>
    <title>登录页面</title>
    <s: head theme="ajax"/>
    </head>
    <body>
    <h3 align="left">
    tree和treenode标签使用范例
    </h3>
    <p>
    <s: tree id="root"label="HTML"theme="ajax">
    <s: treenode label="<b>html1</b
>"id="html1"theme="ajax">
    <s: treenode label="subhtml1"
    id="subhtml1"theme="ajax"></s: treenode>
    <s: treenode label="subhtml2"
    id="subhtml2"theme="ajax"></s: treenode>
    </s: treenode>
    <s: treenode label="<b>html2</b
>"id="html2"theme="ajax"/>
    </s: tree>
    </p>
    </body>
```

使用actionError和actionMessage标签的效果如图5.29所示。使用tree和treenode标签的效果如图5.30所示。

图　5.29　actionError和actionMessage标签使用范例

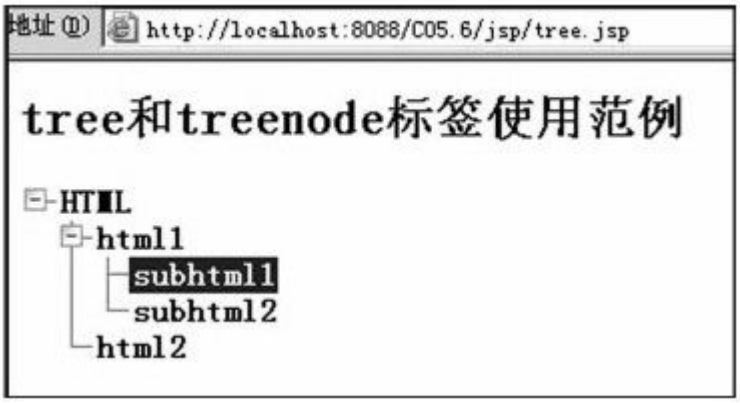

图　5.30　tree和treenode标签标签使用范例

源程序解读

（1）在本示例中可知actionerror标签是输出Struts 2的API方法getActionError（）中的信息。而actionmessage标签则是输出Struts 2的API方法

getActionMessage（）中的信息。这两个方法返回的信息都是一个字符串类型的变量。

由图5.29可知，这两个标签显示在页面上的内容就是在Action代码中封装进去的字符串内容。

（2）treenode和tree标签不但需要联合使用而且都是需要指定ajax主题才能实现树形结构功能。具体使用方式如JSP代码中所示。

（3）component标签功能主要是让开发者自定义自己的Struts 2标签，在下一小节将着重说明，这里暂时不给出示例。

5.7 Struts 2自定义标签实现图形验证功能

本节将介绍如何自定义开发者自己的Struts 2标签来实现登录时图形验证功能。

技术要点

本节先以JSP视图技术作为模板文件使用的视图技术，介绍如何实现图形验证功能。最后结合component标签来说明该标签的重要属性。

JSP模板文件如何实现图形验证功能。

component标签调用模板文件。

实现代码

JSP模板文件：

```
    <!……文件名：
image.jsp……>
    <%@taglib uri="Self"prefix="s"%>
    ……
    <s: self></s: self>
    ……
```

自定义标签Self类文件代码：

```
    <!……文件名：
SelfDefinationTag.java……>
    public class SelfDefinationTag extends
TagSupport {
    ……
    public int doStartTag（）throws JspException {
    JspWriter out=pageContext.getOut（）；
    try {
    out.println（"<img src=\"..\\validateImage
\"/>"）；
    } catch（IOException ioe1） {
    ioe1.printStackTrace（）；
    }
    return EVAL__BODY__INCLUDE;
    }
    ……
```

自定义标签Self模板定义文件（TLD文件）：

```
    <! ……文件名：
self.tld……>
    <taglib>
    <tlibversion>1.0</tlibversion>
    <jspversion>1.2</jspversion>
    <shortname>map</shortname>
    <tag>
    <name>self</name>
    <tagclass>servlet.SelfDefinationTag</tagclass
>
    <bodycontent>JSP</bodycontent>
    </tag>
    </taglib>
```

自定义标签Self在web.xml文件中的部署定义：

```
    <! ……文件名：
web.xml……>
    <servlet>
    <servletname>validateImage</servletname>
    <servletclass>servlet.ValidateImage
</servletclass></servlet>
    <servletmapping>
    <servletname>validateImage</servletname>
    <urlpattern>/validateImage</urlpattern>
</servletmapping>
    ……
    <taglib>
    <tagliburi>Self</tagliburi>
    <tagliblocation>/WEBINF/tlds/self.tld
</tagliblocation></taglib>
```

使用component标签调用模板文件的JSP：

```
    <!……文件名：
SelfDefinationTag.jsp……>
    <%@taglib prefix="s"uri="/strutstags"%>
    <html>
    <head>
    <title>使用JSP模板自定义标签</title>
    </head>
    <body>
    <h3 align="left">
    使用JSP模板实现图形验证标签
    </h3>
    <p>
    <s: component template="/components/image.jsp"/
>
    </p>
    <h3 align="left">
    指定特定主题实现图形验证标签
    </h3>
    <p>
    <s: component
theme="ajax"template="/components/image.jsp"/>
    </p>
    <h3 align="left">
    指定特定模板目录实现图形验证标签
    </h3>
    <p>
    <s: component
theme="ajax"templateDir="MyTemplate"
    template="/components/image.jsp"/>
    </p>
    </body>
    </html>
```

自定义图形验证标签页面效果如图5.31所示。

图　5.31　自定义图形验证标签范例

源程序解读

（1）在本示例中，随机生成动态数字的图形不是本节重点讲述的要点。因此有关生成动态数字图的servlet，读者可以参看示例代码。这里通过实现Struts 2的TagSupport接口定义了一个标签处理类，重写了doStartTag（）方法。在该代码中读者也可知道这个自定义的标签只是插入了一段HTML图像代码，其中的validateImage就是笔者写的生成动态数字图的

servlet类。在配置文件web.xml中也有该servlet的映射定义。

然后在示例项目的WEB__INF目录下新建了“tlds”目录，在该目录下新建了self.tld文件，定义了self标签。并且也在配置文件web.xml中声明了该标签。之后又新建了image.jsp文件，引入了self标签的声明。该文件就作为实现图形验证标签的JSP模板文件。在代码中使用self标签定义。

（2）因为Struts 2中的component标签是用来方便开发者定义自己开发的标签。因此笔者定义了一个新的JSP文件，在该文件中使用component标签来调用image.jsp这个模板文件。

（3）在示例项目的WebRoot根目录下，新建了几个文件夹，用来特殊说明component标签如何使用这些主题和模板。目录结构如图5.32所示。

图　5.32　主题和模板结构

如代码所示，该标签有三个属性比较重要，而且它们都可以不显式申明，并且由图5.32可知，笔者在这些目录下都存放了image.jsp这个模板文件。因此component标签中template属性就是指定该模板文件所在的路径。细心的读者可以发现template属性中的模板文件路径不是全路径，前面缺少了模板根目录和主题文件夹名字的定义。这是因为component标签另外两个属性是分别来定义模板根目录和主题文件夹名字的。这两个属性是templateDir属性和theme属性。

其中theme属性就是定义之前所记述的主题。上一小节也提到它可以默认是“xhtml”主题。因此如果它

没有显式声明，表明主题为“xhtml”，则系统调用的模板文件是“xhtml”文件夹下的JSP模板。

同理，templateDir属性是定义模板文件所在的根目录名。如果它不显式声明，则默认为“template”。因此假设templateDir属性和theme属性都不显式声明，则系统调用的模板文件就是/template/xhtml下的模板文件。

注意：component标签内也可以使用param标签。这个标签可以作为很多标签的子标签。前面也有所提及，请读者仔细品味。

（4）本示例调用JSP模板文件的JSP代码中。第一个小例子同时不显式声明templateDir属性和theme属性，则调用的image.jsp文件是/template/xhtml/components/下的那个模板文件。第二个小例子中显式声明theme属性为“ajax”，则它

调用的是/template/ajax/components/下的模板文件。第三个例子中theme属性为“ajax”，templateDir属性为“MyTemplate”，则它调用的是/MyTemplate/ajax/components/下的模板文件。

（5）component标签调用的模板文件可以不是JSP文件，它还可以是vm和ftl文件。这两个文件分别是velocity和FreeMarker默认的模板文件后缀名。因此template属性中也可以指定这两种文件类型的模板文件。同理，也可以使用这两种类型文件来调用模板文件。在下一章笔者将具体介绍如何在Struts 2框架中使用velocity和FreeMarker这两种视图技术。

第6章　Struts 2非JSP视图技术

在之前的章节中，代码示例都是用JSP来实现的。其实Struts 2不仅仅支持JSP这一种视图技术。作为MVC架构中view层技术，velocity和FreeMarker视图技术也很优秀，近年来很多Web系统项目的架构师和开发者，都开始放弃使用JSP作为view层显示技术，转向使用velocity和FreeMarker。因此有必要介绍Struts 2框架是如何支持这两种视图技术的，希望本章能带领读者进入这两种视图技术的学习。

6.1　velocity视图技术使用介绍

velocity翻译成中文就是“速度”的意思，它的缩写“v”就是物理学中代表速度的符号。由它的名字可以想像出它的设计者有什么样的期望。毕竟有时使

用JSP来实现view层，对于开发者来说是件很麻烦又费时的事情。velocity就是为开发者节约大量view层开发时间而准备的视图技术。它使用模板文件来显示视图界面。除了它自身的书写格式外，模板文件中其他代码就是普通的html代码。因此它也是实现松耦合，让美工和开发人员各司其职进行view层开发。下面通过示例来介绍它在Struts 2中的使用方式。

技术要点

本节代码说明velocity在Struts 2中的使用方式。

velocity基本语义和书写格式说明。

模板文件vm介绍。

Struts 2中如何使用velocity。

实现代码

显示数据的vm模板文件代码：

```
    <!……文件名：
show.vm……>
    <html>
    <head>
    <title>Velocity使用范例</title>
    <meta
httpequiv="contenttype"content="text/html;
charset=GB2312"/>
    </head>
    <body>
    <h3 align="left">
    Velocity使用范例
    </h3>
    <h1>装修材料信息列表</h1>
    <table border>
    <tr align="center">
    <td>材料名</td>
    <td>材料价格（单位：元）</td>
    <td>材料库存量（单位：个）</td>
    </tr>
    #foreach（Material inmList）
    <tr align="center">
    <td>Material.material</td>
    <td>Material.bid</td>
    <td>Material.mount</td>
    </tr>
    #end
    </table>
    </body>
    </html>
```

Struts 2使用velocity的Action代码：

```
    <！……文件名：
VelocityAction.java……>
    public class VelocityAction extends
ActionSupport {
    private List mList;
    ……
    public String execute（）throws Exception {
    mList=new ArrayList（）；
    for（int i=0；i＜4；i++）{//循环显示材料
    Material m=new Material（）；
    m.setMaterial（"材料"+（i+1））；
    m.setMount（10（i+1））；
    m.setBid（1.0+（i+1））；
    mList.add（m）；
    }
    return SUCCESS;
    }
    }
```

配置文件对Action的配置内容：

```
    <！……文件名：
struts.xml……>
    ……
    ＜package name="C06.1"extends="strutsdefault"＞
    ＜action name="velocity"
    class="action.VelocityAction"＞
    ＜result
name="success"type="velocity"＞/velocity/show.vm
```

```
</result>
    </action>
    </package>
```

对支持中文字符的字符编码集设置：

```
    <!……文件名：
struts.properties……>
    #支持本地化的资源文件名定义
    struts.i18n.encoding=gb2312
```

示例效果显示如图6.1所示。

图　6.1　velocity显示效果图

源程序解读

（1）本节示例是将一些装修材料用list形式显示在页面上。之前笔者讲述的示例完全可以由JSP来完成，而该示例是使用velocity技术来实现。读者也可以将其理解为一种和JavaScript相同的脚本语言。它的语义和语法中有以下几个注意点。

“#”标识：该标识用来表明velocity中的控制语句。比如“#if”、“#foreach”等，都是用来控制流程转向的。

“$”标识：该标识来表示对象或变量。比如示例中的“$Material”。如果这些对象还有自己的属性，则可以在其后面再加“.”，如示例中的“$Material.material”。

“｛｝”标识：该标识用来为某对象或变量赋具体的值。比如“｛frank｝”

“！”标识：假设某变量值为null或者还没被定义。则在该变量前加上“！”，这样在页面上调用该变量的地方一律显示为空白即“”。

“#”标识：velocity语言的注释标识。

（2）velocity模板文件后缀名都是“vm”结尾。该类型文件中除了可以书写velocity自己的语言代码之外，也可以使用HTML语言的标签。在本示例中查看show.vm文件就可以知道它和JSP文件还是很相像的。而且Struts 2的标签也可以在vm文件中使用，只需要在原有标签前加“#s”就可以了，参数之间使用″″分隔。如果读者需要在vm文件中引入其他文件，这些被引入文件是JSP或者vm类型的话，将会显示文件中定义的各种动态数据。如果是其他类型的文件被引入，只是显示这些文件的文本即静态引入。一般是使用“#inclue（″文件名″）”格式进行引入。

如果需要引入多个文件则在“（）”中以“，”相隔。还有一个引入文件的书写代码格式是“#parse”，这个只能引入vm类型文件，而且只能引入一个。不过这样引入也有一个好处，就是被引入vm文件中不仅可以使用它自己定义的变量也可以使用引入它的vm文件中的变量。这样引入文件和被引入文件关系有点像JAVA中父类和子类关系。还有如果想让界面中显示velocity中这些标识，而不被velocity解析，那可以在它们前面加上“＼”。比如想显示“$”，则可以写成“＼$”，这样在“$”后面的内容，velocity不会将其解析成velocity的变量而只是普通文本。

注意：velocity只会按照getXXX（）解析变量，例如Material.material实际上找到的Material.getMaterial（）方法，读者可以试着在

Action中写一个变量而不使用getXXX（）方法，看看会不会在vm中取到值，答案是变量照样会被输出。

（3）在Struts 2中使用velocity技术，首先要在项目中导入velocity的jar包，具体jar包名请读者参考项目示例。在该示例中Action无需过多说明。值得说的是struts.xml文件，比如在示例代码中笔者用黑体注明了result的返回类型必须是“velocity”，这样才可以调用vm模板文件。然后就可以像使用JSP一样，使用vm文件显示Action定义的材料list数据。

注意：由于velocity默认显示中文字符用的字符编码集合是“ISO88591”。如果开发中使用的项目字符编码集合不是“ISO88591”，则需要在struts.properties中定义“struts.i18n.encoding”为开发中使用的字符编码集。比如在本示例中就定义为“gb2312”。

6.2 freemarker视图技术使用介绍

freemarker和velocity一样也是一种可以替代JSP的视图技术。而且Struts 2中对它的支持要比对velocity强大得多。

技术要点

本节代码说明freemarker在Struts 2中使用方式。

freemarker基本语义和书写格式说明。

模板文件ftl介绍。

Struts 2中如何使用freemarker。

实现代码

显示数据的ftl模板文件代码：

```
    <! ……文件名：
show.ftl……>
    <html>
    <head>
    <title>FreeMarker使用范例</title>
    <meta
httpequiv="contenttype"content="text/html;
charset=GB2312"/>
    </head>
    <body>
    <h3 align="left">
    FreeMarker使用范例
    </h3>
    <h1>装修材料信息列表</h1>
    <table border>
    <tr align="center">
    <td>材料名</td>
    <td>材料价格（单位：元）</td>
    <td>材料库存量（单位：个）</td>
    </tr>
    <#list mList as Material>
    <tr align="center">
    <td> {Material.material} </td>
    <td> {Material.bid} </td>
    <td> {Material.mount} </td>
    </tr>
    </#list>
    </table>
    </body>
    </html>
```

Action代码同velocity示例中Action代码，这里就不显示了。配置文件对Action的配置内容：

```
    <! ……文件名：
struts.xml……>
    ……
    <package name="C06.1"extends="strutsdefault">
    <action name="velocity"
    class="action.VelocityAction">
    <result
name="success"type="freemarker">/velocity/show.vm
</result>
    </action>
    </package>
```

对支持中文字符的字符编码集设置也如velocity示例。示例效果如图6.2所示。

图　6.2　freemarker显示效果

源程序解读

（1）本节示例也是将一些装修材料用list形式显示在页面上。只不过现在使用的是freemarker。它其实是将Java代码中定义好的类，在底层使用map类型显示在页面上。因此在ftl模板中看见的“#”标识除了表明控制转向之外，其他都是map的key值，而每个key值对应的value就是Java中定义的类，这些类有可能是JavaBean也有可能像本示例中的材料列表一样是list和其他数据集合类。“$”标识和velocity中“$”标识类似用来标识具体对象或变量。

（2）freemarker模板文件后缀名都是“ftl”结尾。它也可以在文件中定义各种各样的html标签和代码。不过它要调用Struts 2的标签则调用方法要比vm文件调用Struts 2标签稍微复杂点。

首先在它的文件头部使用assign来导入Struts 2的标签定义。代码如下：

```
    <#assign
s=JspTagLibs["/WEBINF/strutstags.tld"]/>
```

另外在每次调用Struts 2标签时要加上“@”。

（3）在Struts 2中使用freemarker技术，首先要在项目中导入freemarker的jar包。struts.xml文件中，同样用黑体注明了result的返回类型必须是“freemarker”，这样才可以调用ftl模板文件。关于中文字符支持问题的解决方法也如笔者介绍velocity小节中的相同。

注意：ftl文件中对英文字符的大小写是很敏感的，因此定义某些对象和变量时，请读者注意大小写问题。

6.3 JasperReports报表视图技术使用介绍

在实际的Web项目开发中，往往需要在视图界面中生成各种文件格式的报表文件以供有数据分析需求的客户查看。有一定工作经验的读者知道，同类型的报表生成软件工具有很多，其中最著名的是JasperReports工具。Struts 2中也对该报表工具提供了很好的支持。因此有包含大量数据需要操作或查看的Web项目开发中，就可以将两者结合起来。形成可以查看数据的报表视图文件。本小节将具体介绍如何使用Struts 2和JasperReports报表工具，生成特定的报表视图文件。

技术要点

本节代码使用前几小节的相同示例来说明JasperReports在Struts 2中的使用方式。

JasperReports在Web项目中基本使用方式。

struts.xml中设置使用JasperReports的Action配置定义。

使用JasperReports的Action如何利用JasperReports模板jrxml文件来生成jasper文件。

实现代码

显示数据的JasperReports模板jrxml文件代码（这里仅将重点部分代码罗列给读者阅读）：

```
    <!……文件名：jasper__
template.jrxml……>
    ……
    <reportFont name="宋
体"isDefault="true"fontName="宋
体"size="12"pdfFontName="STSong
    Light"pdfEncoding="UniGBUCS2H"isPdfEmbedded="tru
e"/>
```

```
    <field name="material"class="java.lang.String"/
>
    <field name="bid"class="java.lang.Double"/>
    <field name="mount"class="java.lang.Integer"/>
    ……
    <columnHeader>
    <band height="20">
    <staticText>
    <reportElement
x="0"y="0"width="150"height="20"/>
    <textElement textAlignment="Center">
    <font isUnderline="false"/>
    </textElement>
    <text><! [CDATA[材料名]]></text>
    </staticText>
    <staticText>
    <reportElement
x="150"y="0"width="150"height="20"/>
    <textElement textAlignment="Center">
    <font isUnderline="false"/>
    </textElement>
    <text><! [CDATA[材料价格（单位：元）]]></text>
    </staticText>
    <staticText>
    <reportElement
x="300"y="0"width="150"height="20"/>
    <textElement textAlignment="Center">
    <font isUnderline="false"/>
    </textElement>
    <text><! [CDATA[材料库存量（单位：个）]]></text>
    </staticText>
    </band>
    </columnHeader>
    <detail>
    <band height="15">
    <textField
isBlankWhenNull="true"hyperlinkType="None">
```

```
    <reportElement
x="0"y="0"width="150"height="15"/>
    <textElement textAlignment="Center"/>
    <textFieldExpression
    class="java.lang.String"><!
[CDATA[F {material} ]]></textFieldExpression>
    </textField>
    <textField
isBlankWhenNull="true"hyperlinkType="None">
    <reportElement
x="150"y="0"width="150"height="15"/>
    <textElement textAlignment="Center"/>
    <textFieldExpression
    class="java.lang.Double"><! [CDATA[F {bid} ]]>
</textFieldExpression>
    </textField>
    <textField
isBlankWhenNull="true"hyperlinkType="None">
    <reportElement
x="300"y="0"width="150"height="15"/>
    <textElement textAlignment="Center"/>
    <textFieldExpression
    class="java.lang.Integer"><!
[CDATA[F {mount} ]]></textFieldExpression>
    </textField>
    </band>
    </detail>
    ……
```

配置文件对Action的配置内容：

```
    <! ……文件名：
struts.xml……>
    ……
    <! 设置使用JasperReports的Action>
```

```
    <package name="C06.3"
    extends="strutsdefault, jasperreportsdefault">
    <action name="PDF"class="action.JasperAction">
    <! result类型设置为jasper>
    <result name="success"type="jasper">
    <! 编译后的jasper文件路径>
    <param name="location">
    /jasperreports/compiled_jasper_template.jasper
    </param>
    <! 视图界面中显示的数据的数据源>
    <param name="dataSource">mList</param>
    <! 报表生成格式>
    <param name="format">PDF</param>
    </result>
    </action>
    </package>
    ……
```

将JasperReports模板文件编译成jasper文件的Action代码：

```
    <! ……文件名：
JasperAction.java……>
    public class JasperAction extends
ActionSupport {
    private List mList;
    ……
    public String execute（）throws Exception {
    ……
    //读取jasper_template.jrxml文件，将其编译成.jasper
文件，以供Action调用将数据显示在视图界
    面中
    try {
```

```
    String reportSource;
    reportSource=ServletActionContext.getServletCont
ext（）.getRealPath（"/jasperreports/jasper
    __template.jrxml"）;
    File parent=new
File（reportSource）.getParentFile（）;
    JasperCompileManager.compileReportToFile（report
Source, new File（parent, "compiled__jasper__
    template.jasper"）
    .getAbsolutePath（））;
    } catch（Exception e） {
    e.printStackTrace（）;
    return ERROR;
    }
    ……
    }
```

生成PDF格式报表文件的视图界面如图6.3所示。

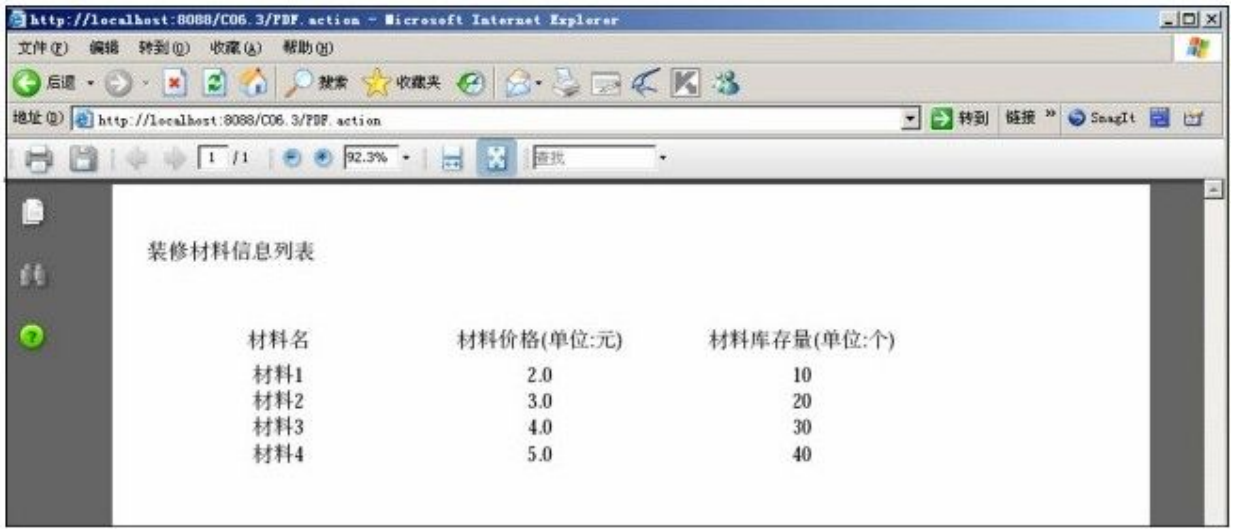

图　6.3　使用jasper生成的PDF报表文件效果图

源程序解读

（1）使用JasperReports工具时，在Web项目的/WEBINF/lib文件夹下必须导入jasperreports3.0.1.jar工具包，这样在Java代码中才能使用相关JasperReports的操作类。如果需要让JasperReports支持中文字符编码集，则还需要导入jasperreports3.0.1.jar、itext1.4.jar两个支持中文字符编码的jar包。除此之外要在Struts 2中和JasperReports结合使用还要导入Struts 2jasperreportsplugin2.0.11.2.jar这个包。

（2）JasperReports要对报表的格式或者数据内容进行定义。一般都是使用后缀名为jrxml的模板文件。其实该模板文件就是一个xml文件，其中使用的标签都是由JasperReports的dtd文件来定义（关于dtd文件概念请见第2章）。

（3）由示例代码可知，该xml文件支持定义Java变量。而“＜！[CDATA[]]＞”中的字符内容都是可以在视图界面上显示的静态文字。除此之外还可以用“$F”来定义各个需要显示的动态数据字段。用“$V”来显示变量值。当然还有其他显示格式，读者可参考相关的JasperReports文档来使用。

如果要让视图文件显示中文字符，在该jrxml模板文件头部分需要声明如下：

```
    ＜reportFont name="宋
体"isDefault="true"fontName="宋
体"size="12"pdfFontName="STSong
    Light"pdfEncoding="UniGBUCS2H"isPdfEmbedded="tru
e"/＞
```

这样就可以在生成的报表文件中显示中文字符或数据。

（4）在一般情况下，编辑和修改jrxml模板文件可以使用一个著名的可视化工具iReport，这样就避免

手工在jrxml文件中进行修改，尤其是在要显示大量数据的报表中，手工修改jrxml文件是件很累人又效率很低的工作。有关iReport工具的具体介绍读者也可自行参阅相关资料。因为不是本书重点，这里只是稍微涉及。

（5）在Struts 2配置文件中，定义使用JasperReports的Action配置。请参看代码，首先package扩展的文件除了“strutsdefault”之外还要有“jasperreportsdefault”，两者用“，”相隔。这是因为Struts 2使用JasperReports时调用的jar包是之前所说的“struts 2jasperreportsplugin2.0.11.2.jar”这个包。

在该包中对于JasperReports的result配置文件是包中的strutsplugin.xml文件。其中用到的package名字就是“jasperreportsdefault”。因此要用到JasperReports的Action，它的result类型必定设为

“jasper”。这样只有在package中声明了“jasperreportsdefault”，用到相关jasper类型的result时才会生效，导航到相应的视图文件。否则，则不会找到相应的视图文件，在界面上就会报错。

（6）在jasper类型的result中，还可以定义一些相关的参数。这些参数中location是显示经过编译后生成的后缀名为jasper的文件存放路径。也可以理解为JasperReports的视图文件显示的URL。而dataSource则是定义报表视图文件中显示的数据是取自哪个数据源。

（7）示例中的mList就是在Action代码中的mList属性。在Action的execute方法中和前几小节相同都把材料数据作为mList元素统一组合起来放入mList数据集合中。因此示例中报表视图文件显示的数据就是mList中包含的数据。

由图6.3显示的数据可以看出的确如此。format是定义报表生成的格式，因为JasperReports不仅可以生成PDF格式还可以生成XLS（显示微软excel文件）、CSV、XML、HTML等格式的报表文件。在jasper类型的result中默认缺省的格式是PDF格式。除了这些参数还有parse、contentDisposition等属性，因为实际开发中用处不大，这里也不作介绍。

（8）在使用JasperReports的Action代码中，参阅代码可知，主要是调用了JasperReports的jasperreports3.0.1.jar工具包中的JasperCompileManager类来将jrxml模板文件编译成jasper文件。这里的jasper文件其实可以理解为Servlet或JSP文件。报表视图文件正是使用该文件才将jrxml文件中定义的数据显示格式、动态数据和静态文本都经过编译后显示在视图界面中。

（9）在示例的Action代码中，JasperCompileManager类使用compileReportToFile方法将jrxml文件编译成jasper文件。这里使用了相关的Java输入输出方法。将编译后生成的jasper文件和jrxml文件放置在同一目录下。

注意：compileReportToFile方法中，两个参数一个是jrxml文件存放的实际目录路径，另外一个是将要生成的jasper文件存放的实际目录路径。这不是两个文件的相对路径，而是它们的绝对路径。

第7章　Struts 2类型转换技术

在Web项目开发中，很多数据在页面上显示时，都是以字符串类型来显示。而在控制层或者model层中，开发人员使用Java开发时，对于这些从页面上传入或者需要传到页面上显示的数据类型，不一定都是字符串类型。因此常常需要在视图和非视图之间进行类型转换，最明显的例子就是显示当前日期。可是这些开发工作往往是无关紧要或者说是犹如“鸡肋”，而很多开发时间却都白白浪费在这上面。基于此，Struts 2的设计者提供了类型转换的功能。

通过前面章节的学习，也应该知道类型转换也是用拦截器来实现的。这里笔者使用Struts 2类型转换功能，看看在Struts 2中是如何实现类型转换的。

7.1 Struts 2类型转换使用介绍

Struts 2的类型转换几乎支持Java中各种数据类型的转换。甚至开发者还可以自定义自己的类型转换功能。我们不推荐开发人员开发自定义的类型转换功能。原因有二。一是遵循IT界著名名言“不重复发明轮子”，不在前人的成果上再次浪费时间。二是类型转换本身在开发工作中就不应该占用大量时间和人力。况且自定义自己的类型转换，项目风险也有可能增加。从项目管理角度上讲对时间、成本、风险的管理都存在负面效应。

因此在本节中笔者具体介绍Struts 2本身所具有的类型转换功能。大致分为以下类型转换：

int、boolean、double等Java基本类型转换。

Date类型转换。

List类型转换。

Set类型转换。

数组类型转换。

除了数组的类型转换不大实用以外，其他几种类型转换都是比较常用的。而且Date类型转换也是属于单个Java变量的转换。而List、Set可以算作集合类型的转换即多个Java变量封装成单个集合的类型转换。下面就依次介绍有关知识点和一些需要注意的细节问题。

7.1.1 基本数据类型转换功能

技术要点

本节代码具体介绍Java基本数据类型和Date类型转换的使用方式。

基本类型转换Action中的使用方式。

基本类型转换在视图界面中的使用方式。

实现代码

使用的Action文件：

```
    <！……文件名：
AddMaterialAction.java……>
    public class AddMaterialAction extends
ActionSupport {
    //属性类型需要类型转换的对象
    private Material material;
    public Material getMaterial（） {
    return material;
    }
    public void setMaterial（Material material） {
    this.material=material;
    }
    public String execute（）throws Exception {
    return SUCCESS;
    }
    }
```

配置文件中的导航定义：

```
    <！……文件名：
struts.xml……>
    <！Action所在包定义>
    <package
name="C07.1.1"extends="strutsdefault">
    <action name="addMaterial"
    class="com.action.AddMaterialAction">
    <result name="input">/jsp/addMaterial.jsp
</result>
    <result name="success">/jsp/showMaterial.jsp
</result>
    </action>
    </package>
```

类型转换的数据输入JSP文件：

```
    <！……文件名：
addMaterial.jsp……>
    ……
    <！材料输入表单>
    <table>
    <s: form
id="materialForm"action="addMaterial">
    <s: textfield name="material.material"label="材
料名"></s: textfield>
    <s: textfield name="material.bid"label="价格">
</s: textfield>
    <s: textfield name="material.mount"label="库存
量"></s: textfield>
    <s: datetimepicker
name="material.expireDate"label="过期日期"></s:
datetimepicker>
    <s: submit value="提交"></s: submit>
    </s: form>
```

```
    </table>
    ……
```

类型转换的显示数据JSP文件：

```
    <!……文件名：showMaterial.jsp……>
    ……
    <!材料数据显示>
    <table>
    材料名：<s：property value="material.material"></s：property>
    价格：<s：property value="material.bid"></s：property>
    库存量：<s：property value="material.mount"></s：property>
    过期日期：<s：property value="material.expireDate"></s：property>
    </table>
    ……
```

数据输入如图7.1所示。显示数据如图7.2所示。请注意各种Java类型数据在显示页面和输入页面的格式及显示的不同之处。

图　7.1　各种基本类型数据输入

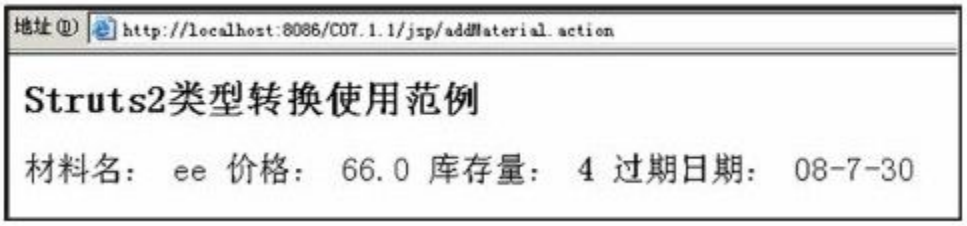

图　7.2　各种基本类型数据显示

源程序解读

（1）在页面上输入一个Material对象的所有属性值，并将它的属性值显示在页面上。由于之前很多示例中已经有过Material这个JavaBean对象代码，这里就没有写出给读者参阅。由已显示的示例代码也可

知，Material对象由材料名、价格、库存量和材料过期日期四个属性组成。

恰好这4个属性的数据类型分别是String、double、int、date四个Java类型，因此由该示例可以知道Struts 2的类型转换是如何转换这些Java类型。由于Struts 2也是使用拦截器来进行类型转换，因此对这些基本的Java类型转换根本不需要开发人员编写任何类型转换代码。

（2）在输入数据的页面，使用OGNL和Struts 2标签来建立一个数据输入的表单。使用的material对象在Action中已定义完成，并建立getter、setter方法。这样在JSP页面可以设置该对象。在页面中按“提交”按钮后，就相当于“setMaterial（）”方法。在系统根据struts.xml配置文件执行Action之前，Struts 2自带的类型转换拦截器就已经将Material对

象中几个属性变量的类型由页面上输入时的String类型转换为Material对象属性变量被定义的数据类型。

（3）在数据显示页面上显示数据时，实际上做的事情是上述操作的一个逆向操作。Material对象中每个属性变量的数据类型又都转换为页面上需要显示的String类型。

（4）实际上在开发工作中，除非有根据特殊需求需要类型转换之外，在Struts 2中绝大部分类型转换都已经由Struts 2自己完成。因此给开发者节省了大量开发时间，除了Java基本类型转换之外，有时候在页面上需要批量输入一些数据，如果这些数据也像本示例的Material对象一样，那么可以使用Struts 2自带的对集合类型的转换功能来完成类型转换。

7.1.2 List集合类型数据类型转换功能

技术要点

本节代码具体介绍包含多个Java对象的List集合类型如何进行类型转换。

List集合类型转换Action中的使用方式。

List集合类型转换在视图界面中的使用方式。

实现代码

使用的Action文件：

```
    <!……文件名：AddMaterialAction.java……>
    public class AddMaterialAction extends ActionSupport {
    private List<Material>materialList; //定义List类型变量
    public String execute() throws Exception {
    return SUCCESS;
```

```
    }
    public List<Material>getMaterialList() {//用
泛型来表明List集合中的元素都是
    Material对象return materialList;
    }
    public void setMaterialList(List<Material>
materialList) {
    this.materialList=materialList;
    }
    }
```

配置文件中的导航定义：

```
    <!……文件名：
struts.xml……>
    <! Action所在包定义>
    <package
name="C07.1.2"extends="strutsdefault">
    <action name="addMaterial"
    class="com.action.AddMaterialAction">
    <result name="input">/jsp/addMaterial.jsp
</result>
    <result name="success">/jsp/showMaterial.jsp
</result>
    </action>
    </package>
```

指定List类型中元素是哪何种对象属性文件：

```
    <!……文件名：
AddMaterialActionconversion.properties……>
```

```
    #集合属性-List类型
    Element__materialList=com.model.Material
```

类型转换的数据输入JSP文件：

```
    <! ……文件名：addMaterial.jsp……>
    ……
    <! 材料输入表单>
    <table>
    <s: form id="materialForm"action="addMaterial"theme="simple">
    <table>
    <tr>
    <td>材料名</td>
    <td>价格</td>
    <td>库存量</td>
    <td>过期日期</td>
    </tr>
    <s: iterator value="new int[4]"status="m">
    <tr>
    <td><s: textfield name="% {' materialList[' +#m.index+' ].material'} "/></td>
    <td><s: textfield name="% {' materialList[' +#m.index+' ].bid'} "/></td>
    <td><s: textfield name="% {' materialList[' +#m.index+' ].mount'} "/></td>
    <td><s: datetimepicker name="% {' materialList[' +#m.index+' ].expireDate'} "/></td>
    </tr>
    </s: iterator>
    <tr>
```

```
    <td colspan="4"><s: submit value="提交"></s:
submit>
    </tr>
    </table>
    </s: form>
    </table>
    ……
```

类型转换的显示数据JSP文件：

```
    <! ……文件名：
showMaterial.jsp……>
    ……
    <! 材料数据显示>
    <table>
    <tr>
    <td>材料名</td>
    <td>价格</td>
    <td>库存量</td>
    <td>过期日期</td>
    </tr>
    <s: iterator value="materialList"status="m">
    <tr>
    <td><s: property value="material"></s:
property></td>
    <td><s: property value="bid"></s: property>
</td>
    <td><s: property value="mount"></s: property
></td>
    <td><s: property value="expireDate"></s:
property></td>
    </tr>
    </s: iterator>
    </table>
```

……

数据输入如图7.3所示。显示数据如图7.4所示。注意各种Java类型数据在显示页面和输入页面的格式及显示的不同之处。

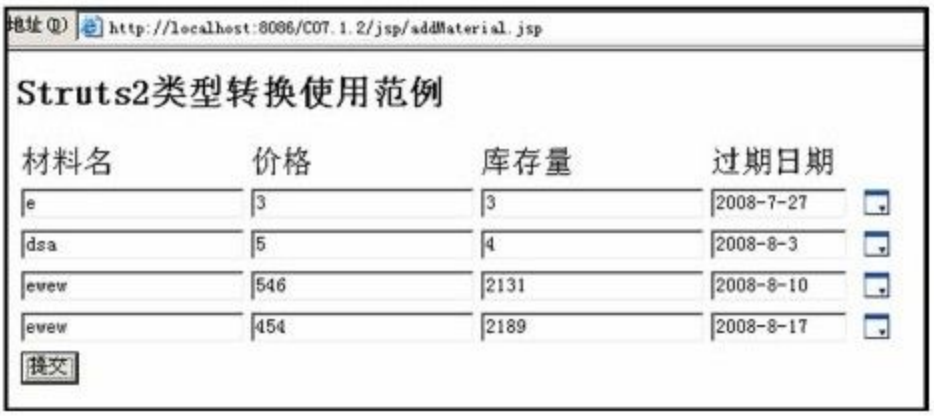

图　7.3　多个material对象数据输入

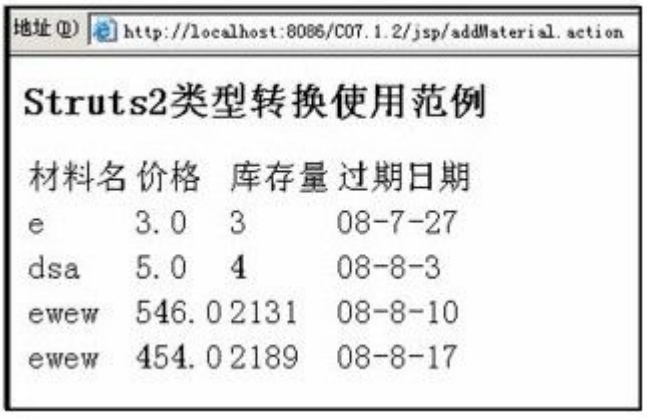

图　7.4　多个material对象数据显示

源程序解读

（1）和上一小节示例相比。本示例虽然输入的也是Material对象，不过却是多个Material对象输入。这里页面上的显示是四个Material对象输入。将这些Material对象组装成List集合类型，然后也可以让Struts 2自带的类型转换器进行类型转换。

首先在Action代码中定义了List类型变量。并且运用Java 5中的泛型来表明该List集合类型变量中的元素都是Material对象。当然，也可以在Action代码中不使用泛型来标明是何种对象作为List集合类型的元素。另一种方法就是示例中写明的在属性文件中指定List集合类型的元素是何种对象。

注意：该属性文件属于局部类型转换属性定义文件。文件名要以ActionName开头，然后以“conversion.properties”结尾。这表示是对该

Action中的List集合类型指定元素类型。而且该属性文件一定要和Action放在同一目录下。否则运行系统时Struts 2是不会知道该Action的List集合类型变量元素是何种类型对象。

（2）试设想有一种情况，多个Action都需要将某一变量的类型进行转换。此时可以像之前所述使用属性文件来定义被转换类型的变量。但是一个Action定义一个属性文件则太浪费时间。因此Struts 2中还有个全局类型转换属性定义文件，这样所有需要类型转换的Action都可以调用该文件中定义的需要类型转换的变量。全局属性文件名必须为“xworkconversiion.properties”。文件中定义的内容其实和局部属性文件中大同小异，只是这些被定义的变量可以在所有Action中进行类型转换。全局属性文件没必要和具体的Action代码文件放在一起，只需

要放在源代码根目录下即可，也就是说全局属性文件直接放在“src”文件夹下即可。

（3）在页面输入的JSP中，千万不能将List集合类型的变量名写错，否则Action得不到具体在页面上输入的值，而且因为是List类型，所以可以利用OGNL来循环遍历，这样在页面上可依次输入数据。数据显示的JSP页面上没有什么特别需要注意的，读者可以看到只是利用Struts 2的标签来显示这些数据。

（4）读者可由图7.3和图7.4看出，批量的Material对象数据输入其实和单个Material对象输入本质上没有多大区别，在视图界面上只是利用OGNL、Struts 2标签来保证数据可以输入和显示。在Action这一层和普通的Struts 2的Action导航没有多大区别，最重要的是全局和局部类型转换属性文件的定义。让系统明白集合类型中包含的元素是何种对象。

不过也请读者不要误解，以为Set集合类型转换也是如此。

7.1.3　Set集合类型数据类型转换功能

技术要点

本节代码具体介绍包含多个Java对象的Set集合类型如何进行类型转换。

Set集合类型转换在Action中的使用方式。

Set集合类型转换在视图界面中的使用方式。

实现代码

使用的Action文件：

```
    <!……文件名：AddMaterialAction.java……>
    public class AddMaterialAction extends ActionSupport {
    private Set<Material>materialSet=new HashSet（）; //定义Set集合类型变量
    public String execute（）throws Exception {
    System.out.println（"fuck”+materialSet）;
```

```
    return SUCCESS;
    }
    public Set<Material>getMaterialSet() {//用泛型
来表明集合中的元素都是Material对象
    return materialSet;
    }
    public void setMaterialSet(Set<Material>
materialSet) {
    this.materialSet=materialSet;
    }
    }
```

配置文件中的导航定义：

```
    <!……文件名：
struts.xml……>
    <! Action所在包定义>
    <package
name="C07.1.3"extends="strutsdefault">
    <action name="addMaterial"
    class="com.action.AddMaterialAction">
    <result name="input">/jsp/addMaterial.jsp
</result>
    <result name="success">/jsp/showMaterial.jsp
</result>
    </action>
    </package>
```

指定Set类型中元素是哪种对象属性文件：

```
    <！……文件名：
AddMaterialActionconversion.properties……>
    #集合属性-Set类型
    Element__materialSet=com.model.Material
    #制定索引
    KeyProperty__materialSet=material
```

类型转换的数据输入JSP文件：

```
    <！……文件名：
addMaterial.jsp……>
    ……
    <！材料输入表单>
    <table>
    <s: form
id="materialForm"action="addMaterial"theme="simple
">
    <table>
    <tr>
    <td>材料名</td>
    <td>价格</td>
    <td>库存量</td>
    <td>过期日期</td>
    </tr>
    <s: iterator value="new int[4]"status="m">
    <tr>
    <td><s: textfield name="% {'
materialSet.makeNew[' +#m.index+' ].material'} "/>
</td>
    <td><s: textfield name="% {'
materialSet.makeNew[' +#m.index+' ].bid'} "/></td
>
    <td><s: textfield name="% {'
materialSet.makeNew[' +#m.index+' ].mount'} "/>
```

```
</td>
    <td><s: datetimepicker name="% {'
materialSet.makeNew[' +#m.index+' ].expireDate'} "/
>
    </td>
    </tr>
    </s: iterator>
    <tr>
    <td colspan="4"><s: submit value="提交"></s:
submit>
    </tr>
    </table>
    </s: form>
    </table>
    ……
```

类型转换的显示数据JSP文件：

```
    <! ……文件名:
showMaterial.jsp……>
    <! 材料数据显示><table>
    <tr>
    <td>材料名</td>
    <td>价格</td>
    <td>库存量</td>
    <td>过期日期</td>  </tr>
    <s: iterator value="materialSet"status="m">  <
tr>
    <td><s: property value="material"></s:
property></td>  <td><s: property value="bid">
</s: property></td>
    <td><s: property value="mount"></s: property
></td>
```

```
    <td><s: property value="expireDate"></s:
property></td>  </tr>
    </s: iterator>  </table>
    ……
```

数据输入如图7.5所示。显示数据如图7.6所示。

图　7.5　多个material对象数据输

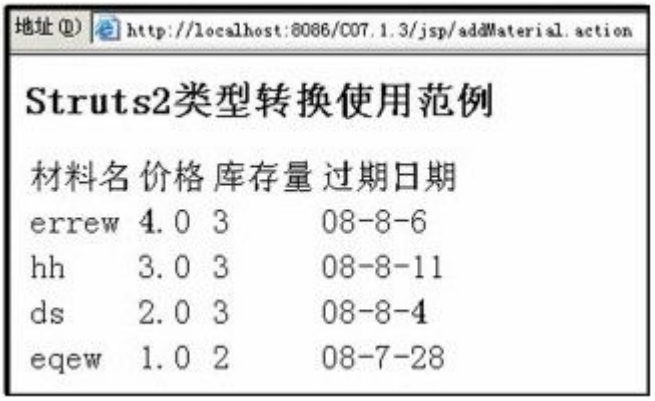

图　7.6　多个material对象数据显示

源程序解读

（1）本小节还是以输入多个Material对象为例，先说明在Action文件中，像List集合类型示例一样笔者定义了一个Set集合类型变量materialSet。不过该变量必须显示定义它的类型。如代码中所示“new HashSet（）”一样。这是必须的，否则Struts 2不知道该变量是哪一种Set集合，这样系统会找不到在属性文件中定义的索引和元素对象类型。

（2）属性文件中除了继续定义元素对象类型之外，还需要定义该集合类型变量的索引即代码中所示的“KeyProperty__materialSet=material”。其实定义索引的通用格式为“KeyProperty__SetName=属性”。也许有的读者很难理解，其实也不很难。首先在Action中定义的Set集合类型变量名要写在“KeyProperty__”之后。然后“=”后面写的是元素对象中开发者想指定为索引的属性。

比如示例中元素对象是Material对象。该对象有几个属性，其中有个属性为“material”是代表Material对象的材料名字，就把它定义为索引。这里笔者的定义想表明的意思就是materialSet这个Set集合类型变量的索引是组成它的元素Material对象中的material属性，如果属性文件中写成“KeyProperty__materialSet=bid”则表明把Material对象中表示材料价格的bid属性作为materialSet的索引。

（3）在数据输入的JSP页面中，笔者使用了OGNL中的“makeNew”API。由它来建立materialSet中的material对象。然后依次输入material对象的各个属性数据。

（4）除了上述几点之外，Set集合类型转换和List集合类型转换没太大的区别。

7.2　类型转换发生异常的处理方案

类型转换其实也就是调用某些Struts 2已经定义的Java代码。不过只要由Java这门语言书写的程序都不可避免异常处理。因此万一在类型转换中发生异常，必须要快速解决。本节就介绍笔者日常中对类型转换的异常进行处理的一些办法和方案。

7.2.1　Struts 2自带异常提示

技术要点

Struts 2自带对于类型转换发生异常错误时的提示。

无任何操作时的错误提示。

输入fielderror标签时的错误提示。

实现代码

使用7.1.1示例代码没有进行任何数据异常处理时的数据输入，显示如图7.7所示。输入价格类型为字符串类型，而价格类型其实是double类型，此时会出现错误提示，如图7.8所示。

图　7.7　各种基本类型数据输入

图　7.8　价格类型输入错误图

引入Struts 2标签fielderror的数据输入JSP代码：

```
    <！……文件名：addMaterial.jsp……>
    ……
    <s：fielderror/>
    ……
```

引入标签后类型转换错误的提示如图7.9所示。

图　7.9　引入标签fielderror后价格类型输入错误图

源程序解读

（1）本节代码大致和7.1.1小节的代码类似。在没有引入fielderror标签之前。笔者只是简单地把7.1.1代码copy过来，没有增加任何代码。当在页面中输入数据，输入的价格数据为字符串类型数据，由于类型转换时，价格作为Material对象的一个属性，它的Java代码定义的类型是double类型，因此如果页面上输入的价格不是数字组成的，则转换类型时必定会发生类型转换异常。此时Struts 2自带的处理类型转换异常的机制就会如图7.8所示，显示红色的错误提示信息和出错的数据输入框提示。

（2）Struts 2本身还有个提示表单中field输入错误的提示标签。该标签名字为“fielderror”，具体写法如代码中所示。之所以在前面介绍Struts 2标签的章节中没有记述该标签，只是因为很多Struts 2标签有特定的使用意义和使用场合，因此在介绍

Struts 2其他部分时会顺便介绍这些有特殊使用场合的标签。

如图7.9所示，JSP代码中加了该标签后，显示的错误提示信息是在页面头部分。也就是说fielderror标签在页面上引入后，显示的错误提示除了显示位置不同之外，其他都和不引入标签时Struts 2自带提示类型转换错误提示相同。

注意：错误提示中的“material.bid”是JSP中定义的价格，field的name。

7.2.2 Struts 2局部异常提示定义属性文件使用介绍

技术要点

定义局部属性文件，在该文件中定义类型转换发生异常时的错误提示。

局部属性文件定义。

实现代码

局部属性文件代码：

```
    <!.....文件名：AddMaterialAction.properties.....>
    #Action属性类型转换错误提示
    invalid.fieldvalue.material.bid=材料价格非法输入
```

异常错误提示如图7.10所示。

图　7.10　价格类型输入错误图

源程序解读

如同之前类型转换属性文件定义一样，在和Action文件同目录下定义了名为“ActionName.properties”的属性文件，在该属性文件中对类型转换错误的属性进行定义。其中“material.bid”就是输入数据的JSP页面中定义的field的name。而“invalid.fieldvalue”则是固定不变的格式。

这样设置后，系统就会在发生类型转换错误时在页面上显示属性文件中定义的类型转换错误提示信息，不会显示之前使用fielderror标签时显示的提示信息。

7.2.3 Struts 2全局异常提示定义属性文件使用介绍

技术要点

定义全局属性文件，在该文件中定义类型转换发生异常时的错误提示。

全局属性文件定义。

实现代码

全局属性文件代码：

```
    <!……文件名：messageResource.properties……>
    #类型转换提示
    xwork.default.invalid.fieldvalue={0}类型转换错误~
```

异常错误提示如图7.11所示。

图　7.11　价格类型输入错误图

源程序解读

在源代码根目录下笔者定义了名为“messageResource.properties”的属性文件，该文件是用来显示国际化目的而使用的属性文件。但关于类型转换的错误定义也可以在该属性文件中定义。这样在所有JSP页面中只要有关于Material对象的bid属性的输入类型转换都可以调用该属性文件中对bid属性类型转换异常提示的信息。

注意：属性文件中的“{0}”其实是占位符号，如果有多个属性的类型转换有异常发生，可依次以“{0}”，“{1}”定义多个属性。然后这些属性名会自动一一对应于属性文件中的定义。比如该示例中是“material.bid”，则“{0}”代表的就是它。所以在异常信息提示的页面上显示的是“material.bid类型转换错误”。

第8章　Struts 2输入校验

前一章讲述的类型转换异常处理情况，其实就笔者认为也可以算是Struts 2的输入校验中的一种方式。在记述Struts 2的核心技术时，笔者也简单介绍过一些输入校验的Struts 2的自带类和方法。现在本章重新整理一下，将Struts 2中所有输入校验的使用做个完整而又详细的介绍。

8.1　validate输入校验方式再谈

在Web系统项目中有大量的视图页面需要用户自行输入很多数据。这些数据的类型有很多种。为了防止某些客户的恶意输入以及对Web项目的恶意破坏。必须引入输入校验像Windows操作系统中的防火墙一样，把一些“垃圾”数据过滤，挡在Web系统之外。

Struts 2的输入校验是以上一章的类型转换为基础。而且输入校验一般和Web系统的业务逻辑息息相关。所以在阅读本章前，笔者建议读者能仔细参看类型转换章节。

在前面的章节中笔者也曾对Struts 2输入校验中，最基本的使用validate方式做过简单介绍，本小节再次就这一话题进行讨论。

8.1.1 复习validate方法进行输入校验

技术要点

本节代码就一个简单的用户注册功能，具体介绍利用validate方法对数字、字符串、日期等类型数据进行输入校验方式的介绍。

几种基本Java数据类型输入校验。

针对具体业务逻辑进行输入校验。

实现代码

使用的Action文件：

```
    <！……文件名：
RegisterAction.java……>
    public class RegisterAction extends
ActionSupport {
    private static String FORWARD=null; //Action类公
用私有变量，用来做页面导航标志
    private String username; //用户名属性
    ……
    private int age; //年龄属性
    public String getUsername（） {//取得用户名值
    return username;
    }
    public void setUsername（String username） {//设
置用户名值this.username=username;
    }
    ……
    public int getAge（） {//取得年龄值
    return age; }
    public void setAge（int age） {//设置年龄值
    this.age=age; }
    public String execute（）throws Exception {//执行
注册方法FORWARD="success";
    return FORWARD; }
    //校验方法，用来输入校验
    public void validate（） {
    //校验是否输入用户名
```

```
    if（getUsername（）==null｜｜
getUsername（）.trim（）.equals（""））
 {addFieldError（"username"，"请输入用户名"）；
    }
    if（getBirthday（）==null）｛//校验是否输入生日
addFieldError（"birthday"，"请输入生日日期"）；
    } else
    if（getBirthday（）.after（new Date（））） ｛//校验
是否输入正确的生日日期addFieldError（"birthday"，"请不要
输入未来日期"）；
    }
    if（getPassword（）==null｜｜
getPassword（）.trim（）.equals（""））｛//校验是否输入
密码addFieldError（"password"，"请输入密码"）；
    }
    //校验是否输入确认密码
    if（getRepassword（）==null｜｜
getRepassword（）.trim（）.equals（""））
 {addFieldError（"repassword"，"请输入确认密码"）；
    }
    //校验输入的密码和确认密码是否一致
    if（！
getPassword（）.equals（getRepassword（））） {
    addFieldError（"repassword"，"确认密码和密码输入不
一致"）；}
    if（getMobile（）.length（）！=11）｛//校验输入的手
机号码长度是否正确addFieldError（"mobile"，"请输入正确的
手机号码"）；
    }
    if（getAge（）＜1｜｜getAge（）＞99）｛//校验输入的年
龄是否正确addFieldError（"age"，"请输入您的真实年龄"）；
    }
    }
    }
```

配置文件中的导航定义：

```
    <！……文件名：
struts.xml……>
    <struts>
    <！Action所在包定义>
    <package
name="C08.1.1"extends="strutsdefault">
    <！Action名字，类以及导航页面定义>
    <！通过Action类处理才导航的的Action定义>
    <action name="Register"
    class="action.RegisterAction">
    <result name="input">/jsp/register.jsp
</result>
    <result name="success">/jsp/success.jsp
</result>
    </action>
    <！直接导航的的Action定义>
    <action name="index">
    <result>/jsp/register.jsp</result>
    </action>
    </package>
    </struts>
```

输入校验的数据输入JSP文件：

```
    <！……文件名：
register.jsp……>
    ……
    <！用户信息注册form表单>
    <s: form action="Register">
    <table width="60%"height="76"border="0">
```

```
    <! 各标签定义>
    <s: textfield name="username"label="用户名"/>
    <s: password name="password"label="密码"/>
    <s: password name="repassword"label="密码确认"/
>
    <s: textfield name="birthday"label="生日"/>
    <s: textfield name="mobile"label="手机号码"/>
    <s: textfield name="age"label="年龄"/>
    <s: submit value="注册"align="center"/>
    </table>
    </s: form>
```

数据不进行任何输入显示的出错信息如图8.1所示。输入密码不一致时的出错信息如图8.2所示。

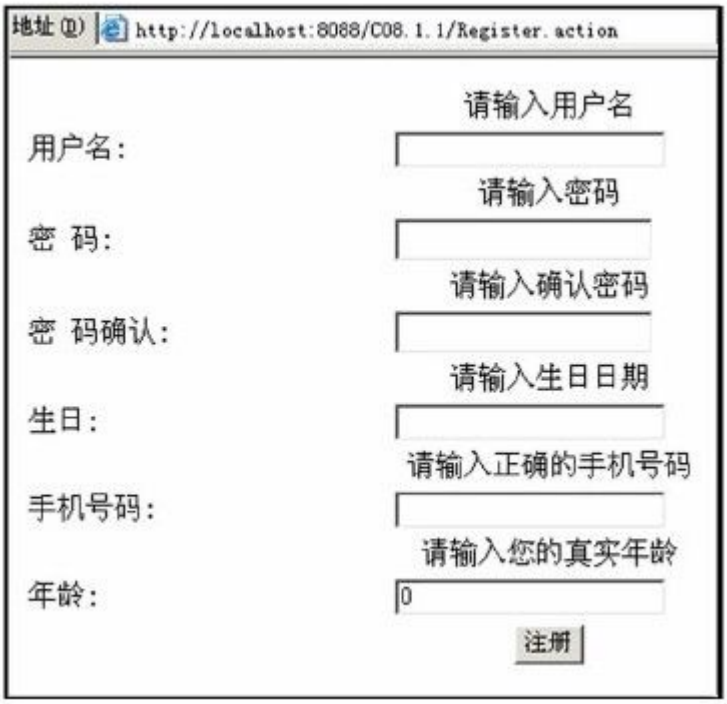

图 8.1 输入校验发现数据没有进行任何输入

图　8.2　密码输入不一致时的信息显示

源程序解读

（1）Struts 2对输入校验这方面采用的最基本方法是在每个Action中继承ActionSupport类，并且重写它的输入校验方法validate（）。本示例中的RegisterAction代码中也显示，根据页面上输入的各种校验将所有不符合输入校验规则的错误信息都由ActionSupport类中另一个方法addFieldError方法将错误信息加入到表单错误信息，并且在输入数据的页面显示，不会再由Action导航到注册成功页面。

struts. xml也定义了一个名字为“input”的result，它表明将所有输入失败的错误信息导航到一个特定页面。本示例中笔者还是将这个特定页面定义为数据输入的页面。

（2）再次阅读RegisterAction代码，可以发现在validate方法中笔者编写了很多if语句。每一个if语句中都针对表单中某一字段进行输入校验。如果发现不符合输入校验规则则都调用addFieldError方法。该方法中有两个参数，第一个参数都是表单中字段名，这里所有的名字都和输入数据的页面中每一个字段的name属性中内容相同。否则Struts 2找不到具体的错误信息是针对哪一个字段。第二个参数则是错误信息的内容。

这些内容就是在输入校验失败时显示在之前所说的特定页面中，由图7.1和图7.2可以看到这些内容在

页面上是如何显示的。

（3）validate方法中的各个if语句判断了表单中各个字段的输入数据是否符合输入校验的规则，这些规则也是开发人员根据特定业务逻辑定义的。比如其中数据是否输入，输入的生日信息是否在当前日期之前等。细心读者又可以发现并没有对这些字段进行类型转换，但在Action中某些字段类型都已经变成Java的一些基本类型。比如生日字段，页面上输入时是字符串，在Action中已经变成Java中的Date类型。

之前在类型转换章节也已说明：页面上输入的数据已经都由字符串类型转换成Action中指定的Java类型。因此从这一点更加说明类型转换是输入校验的基础，也可以说是一种特定的输入校验。

8.1.2 validateXXX方法进行输入校验

技术要点

本节代码也就一个简单的用户注册功能，具体介绍利用validateXXX方法对Action中某一特定的方法进行校验。

Action具体方法的validateXXX方法介绍。

实现代码

使用的Action文件：

```
    <!……文件名：RegisterAction.java……>
    public class RegisterAction extends ActionSupport {
    private static String FORWARD=null; //Action类公用私有变量，用来做页面导航标志
    private String username; //用户名属性
    ……
    private int age; //年龄属性
```

```
    public String getUsername（）｛//取得用户名值
    return username;
    }
    public void setUsername（String username）｛//设
置用户名值
    this.username=username;
    }
    ……
    public int getAge（）｛//取得年龄值return age;
    }
    public void setAge（int age）｛//设置年龄值
    this.age=age;
    }
    public String Register（）throws Exception｛//执
行注册方法
    FORWARD="success";
    return FORWARD;
    }
    //校验方法，用来输入校验
    public void validateRegister（）｛
    //校验是否输入用户名
    if（getUsername（）==null｜｜
getUsername（）.trim（）.equals（""））｛
    addFieldError（"username"，"请输入用户名"）；
    }
    //校验是否输入生日
    if（getBirthday（）==null）｛
    addFieldError（"birthday"，"请输入生日日期"）；
    ｝else
    //校验是否输入正确的生日日期
    if（getBirthday（）.after（new Date（））｛
    addFieldError（"birthday"，"请不要输入未来日期"）；
    }
    //校验是否输入密码
    if（getPassword（）==null｜｜
getPassword（）.trim（）.equals（""））｛
    addFieldError（"password"，"请输入密码"）；
```

```
    }
    //校验是否输入确认密码
    if（getRepassword（）==null｜｜
getRepassword（）.trim（）.equals（""））{
    addFieldError（"repassword"，"请输入确认密码"）；
    }
    //校验输入的密码和确认密码是否一致
    if（！
getPassword（）.equals（getRepassword（））{
    addFieldError（"repassword"，"确认密码和密码输入不
一致"）；
    }
    //校验输入的手机号码长度是否正确
    if（getMobile（）.length（）！=11）{
    addFieldError（"mobile"，"请输入正确的手机号码"）；
    }
    //校验输入的年龄是否正确
    if（getAge（）＜1｜｜getAge（）＞99）{
    addFieldError（"age"，"请输入您的真实年龄"）；
    }
    }
    }
```

配置文件中的导航定义同8.1.1小节。输入校验的数据输入JSP文件同8.1.1小节有一点不同，具体代码如下：

```
    ＜！……文件名：
register.jsp……＞
    ……
    ＜！ fielderror标签显示所有校验错误信息＞＜s：
fielderror＞＜/s：fielderror＞
```

```
    <! 用户信息注册form表单>
    <s: form action="Register! Register.action">
    <table width="60%"height="76"border="0"><! 各
标签定义>
    <s: textfield name="username"label="用户名"/><
s: password name="password"label="密码"/>
    <s: password name="repassword"label="密码确认"/
><s: textfield name="birthday"label="生日"/>
    <s: textfield name="mobile"label="手机号码"/><
s: textfield name="age"label="年龄"/>
    <s: submit value="注册"align="center"/></table
>
    </s: form>
```

数据不进行任何输入显示的出错信息如图8.3所示。输入密码不一致时的出错信息如图8.4所示。

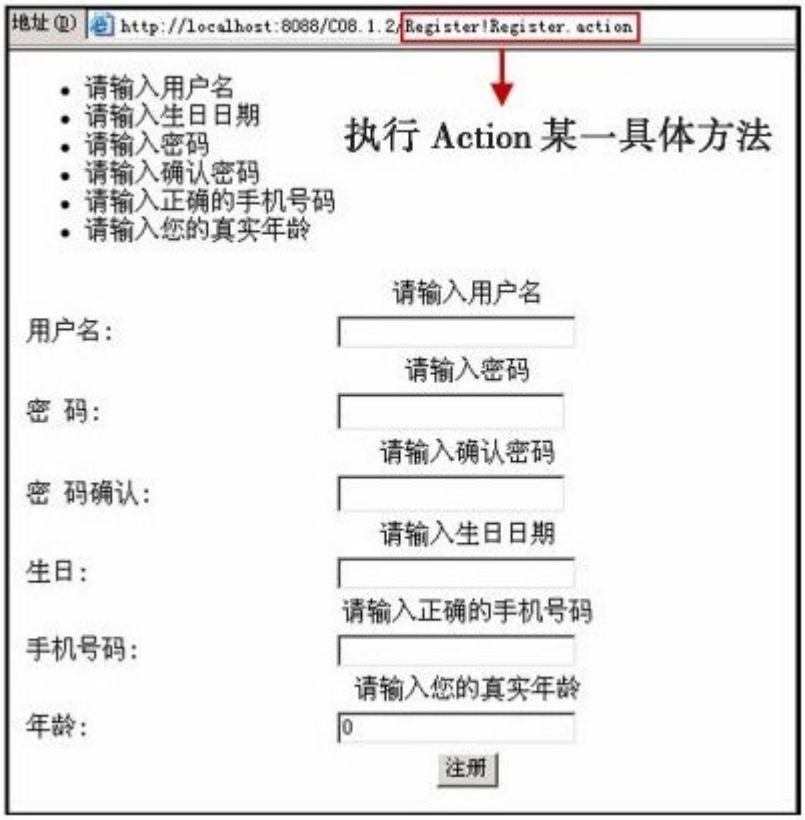

图 8.3 输入校验发现数据没有进行任何输入

图 8.4 密码输入不一致时出错信息显示

源程序解读

（1）Struts 2中除了validate方法之外它还有一种validateXXX方法针对Action中某一特定方法进行该方法的各种字段的输入校验。其中XXX就是该特定方法名。在本示例中笔者定义了一个Register方法，该方法和上一小节中的execute方法类似只是一个简单的导航。但是在该RegisterAction中就没有了validate方法，取而代之的是validateRegister方法。

如果读者使用validateRegister方法，最好不要再使用validate方法。虽然和上一小节示例代码比较这两个方法中的内容是一模一样的，但是validate方法是对所有Action中方法的输入校验都进行校验，validateRegister方法只对Register方法进行校验。因此两者不能重复使用，都使用会造成两个方法都进行了校验的结果。执行顺序是先validateRegister后

validate。如果validateRegister方法有特殊的输入校验则就会被validate方法“覆盖”，因此达不到预期的输入校验目的。推荐读者自己进行试验，在这两个方法中设置断点运行一下就知道了。

（2）请读者阅读数据输入的页面代码，在代码中用黑体标注的是一个特殊的运行Action的示例。这里笔者故意把名字都写为“Register”来让读者加深理解。首先第1个“Register”是RegisterAction中的方法名，一定要和方法名写成一样。而在“！”后的“Register”则是在struts.xml配置文件中定义的RegisterAction的映射中的“name”内容。黑体的内容表明该表单的Action是执行RegisterAction中的Register方法。如果在页面中直接写“！”后面的内容则表示执行的是RegisterAction中的execute方法。

在图8.3所示浏览器中的被红框框中的URL，表明该表单数据输入完成后提交时执行的是Register方法。这是Struts 2中一个特殊的使用方式。如果开发者以后在使用Struts 2的开发工作中，根据特定业务逻辑不想执行execute方法而是执行另外一个开发完成的方法。则在视图页面（包括velocity和freemarker）中可以这种方式让表单提交后执行该开发完成的方法。

（3）validateRegister方法中各个if语句定义和上一小节的validate方法内容相同。这里笔者只是作为示例，其实也可以和上一小节中的validate方法的内容不相同，用来进行Register方法中特定的表单字段输入校验。

（4）在数据输入的视图界面笔者又增加了Struts 2的标签fielderror。从图8.3和图8.4中可以看出，在

具体字段输入校验出错信息显示之外，在表单头部也有出错信息显示，这其实和Struts 2的校验顺序有关。

在之前说明validateRegister方法和validate方法时也记述了两者的执行校验顺序是先validateRegister后validate。其实在视图界面进行表单提交后。输入校验顺序是以如下的顺序进行的：

查找Action中是否有validateXXX方法。如果有则执行该方法，将校验产生的错误信息放置到ActionContext对象中。

查找Action中是否有validate方法。如果有则执行该方法，将校验产生的错误信息放置到ActionContext对象中。

查找视图界面是否有fielderror标签定义。如果有则返回到result为“input”的视图。同时ActionContext对象中有关的输入校验的错误信息也显示在该视图中。

Struts 2的输入校验顺序就是按照如上说明来先后执行的，这也更好地说明了validateRegister方法和validate方法并存在Action中时输入校验是如何进行的，这也是图8.3和图8.4会产生显示两遍错误信息的原因。

8.2　利用配置文件进行输入校验方法说明

8.1　节中说明的输入校验方法会让程序产生很多代码。如果是一个项目生命周期很短的项目，必然对时间的要求会很高。这时利用Struts 2来对该项目进行开发，开发者势必不喜欢8.1小节中介绍的输入校验方法。因为代码太多，工作量一下子就增大了不少。而且它是一种硬编码的形式，增大了系统各个模块之间的耦合度，也不利于项目后期的维护和实施。因此本节介绍的利用配置文件进行输入校验的方式则很好地解决了上述问题。

利用配置文件进行校验主要是增加了一些xml文件来对具体表单中各个字段进行输入校验，当然配置文件能做的事情不止这些。其本身的输入校验规则的书

写格式也有字段和非字段两种，下面笔者就来具体介绍这些知识点。

8.2.1 Struts 2字段校验的配置文件形式

技术要点

Struts 2输入校验配置文件字段校验格式介绍。

输入校验配置文件命名方式和相关注意点。

字段校验格式和相关注意点。

实现代码

数据输入JSP代码：

```
    <!……文件名：register.jsp……>
    ……
```

```
    <! 用户信息注册form表单>
    <s: form
action="Register.action"validate="true">
    <table width="60%"height="76"border="0">
    <! 各标签定义>
    <s: textfield name="username"label="用户名"/>
    <s: password name="password"label="密码"/>
    <s: password name="repassword"label="密码确认"/
>
    <s: textfield name="birthday"label="生日"/>
    <s: textfield name="mobile"label="手机号码"/>
    <s: textfield name="age"label="年龄"/>
    <s: submit value="注册"align="center"/>
    </table>
    </s: form>
    ……
```

输入校验文件代码：

```
    <! ……文件名：
RegisterActionvalidation.xml……>
    <? xml version="1.0"encoding="gb2312"? >
    <! DOCTYPE validators PUBLIC
    "//OpenSymphony Group//XWork Validator
1.0.2//EN"
    "http: //www.opensymphony.com/xwork/xworkvalida
tor1.0.2.dtd">
    <validators>
    <field name="username">
    <! 检验用户名的长度>
    <fieldvalidator type="stringlength">
    <param name="minLength">6</param>
    <param name="maxLength">8</param>
```

```
    <message>用户名长度必须在 {minLength}
 {maxLength} 位之间</message>
    </fieldvalidator>
    <! 检验用户名是否已输入>
    <fieldvalidator type="requiredstring">
    <message>请输入用户名</message>
</fieldvalidator>
    </field>
    <field name="password">
    <! 检验密码的长度>
    <fieldvalidator type="stringlength">
    <param name="minLength">6</param>
    <param name="maxLength">8</param>
    <message>密码长度必须在 {minLength} {maxLength}
位之间</message>
    </fieldvalidator>
    <! 检验密码是否已输入>
    <fieldvalidator type="requiredstring">
    <message>请输入密码</message>
    </fieldvalidator>
    </field>
    <field name="repassword">
    <! 检验确认密码的长度>
    <fieldvalidator type="stringlength">
    <param name="minLength">6</param>
    <param name="maxLength">8</param>
    <message>确认密码长度必须在 {minLength}
 {maxLength} 位之间</message>
    </fieldvalidator>
    <! 检验确认密码是否已输入>
    <fieldvalidator type="requiredstring">
    <message>请输入确认密码</message>
</fieldvalidator>
    <! 检验密码和确认密码的输入内容是否一致>
    <fieldvalidator type="fieldexpression">
    <param name="expression">password==repassword
</param>
```

```
    <message>确认密码和密码输入不一致</message>
    </fieldvalidator></field>
    <field name="birthday">
    <! 检验生日是否已输入>
    <fieldvalidator type="required">
    <message>请输入生日日期</message>
</fieldvalidator>
    <! 检验输入日期是否在一个有效日期范围内>
    <fieldvalidator type="date">
    <param name="min">19280101</param>
    <param name="max">20040101</param>
    <message>输入生日日期无效</message>
    </fieldvalidator></field>
    <field name="mobile">
    <! 检验手机号码是否已输入>
    <fieldvalidator type="requiredstring">
    <message>请输入手机号码</message>
    </fieldvalidator>
    <! 检验输入手机号码长度是否是11位有效手机号码>
    <fieldvalidator type="stringlength">
    <param name="minLength">11</param>
    <message>请输入正确的手机号码，号码位数必须为11位
</message>
    </fieldvalidator>
    </field>
    <field name="age">
    <! 检验年龄是否已输入>
    <fieldvalidator type="required">
    <message>请输入年龄</message>
    </fieldvalidator>
    <! 检验输入年龄是否符合特定年龄范围>
    <fieldvalidator type="int">
    <param name="min">1</param>
    <param name="max">80</param>
    <message>年龄必须在{min}{max}岁之间</message
>
    </fieldvalidator>
```

```
</field>
</validators>
```

源程序解读

（1）使用配置文件来完成输入校验这种方式，首先在Action代码中去除所有validate和validateXXX方法的代码。然后在和Action类文件同一级目录下增加XXXvalidation.xml配置文件。这里“XXX”是Action类文件名字，表示该XML文件中所有输入校验的规则定义和错误信息显示方式都只针对该Action有效。

（2）该输入校验的配置文件有两种书写格式，一种是本节笔者要说的字段校验格式。另一种是下一小节要介绍的非字段格式。首先来说明字段校验格式。如代码所示。在＜validators＞和＜/validators＞之间使用＜field＞来对输入界面表单中每一个字段进行输入校验规则定义和错误信息定义。

（3）＜field＞中的name属性就是表单中字段名字。它里面包含＜fieldvalidator＞标签，它的type属性表明是何种类型的输入校验。这些输入校验的type都是在Struts 2中默认定义的。被称之为校验器。具体这些校验器是如何形成的以及如何生效，在之后章节有介绍，这里读者只是知道一下就可以了。

（4）在＜fieldvalidator＞标签内可以有两种标签。一种是＜param＞标签，该标签定义了一些输入校验规则需要用到的参数。这些参数更可以以“${参数名}”格式显示在视图页面上。除此之外还有＜message＞标签，该标签定义的是输入校验出错后的出错信息，这些信息是可以显示在视图界面上的。

（5）值得说的是代码中黑体表明的参数名字。因为Struts 2中特定的校验类型的参数名是已经在Struts 2代码中缺省定义过的。因此如果开发者定义

的参数名字和它本身缺省定义的不符合。那么输入校验时在配置文件中定义的该校验规则是不会生效的。比如代码中的“stringlength”类型校验，如果定义的最大长度不是“maxLength”而是其他的。则在输入数据界面，如果输入的数据超过最大长度，则视图界面不会给出特定的出错信息。

（6）如果在输入数据的视图界面的表单中输入上述JSP文件中的黑体“validate=″true″”，则其实是另外一种输入校验的方式，这种方式称之为“客户端输入校验方式”，它会自动在试图页面中生成很多JavaScript代码，但是它是有其局限性的，因此这些代码的适合程度不是对Struts 2中所有的主题都适合的（有关主题的基本概念翻阅第5章）。因此笔者本身不赞同这样的输入校验方式，这里只是希望读者对Struts 2的输入校验有个完整的印象。

8.2.2 Struts 2非字段校验的配置文件形式

技术要点

Struts 2输入校验配置文件非字段校验格式介绍。

非字段校验格式和相关注意点。

实现代码

与上一小节代码相比，只是输入校验配置文件中的内容有所不同。代码如下：

```
    <!……文件名：RegisterActionvalidation.xml……>
    <validators>
    <! 检验输入用户名长度是否在一个有效范围内>
    <validator type="stringlength">
    <param name="fieldName">username</param><param name="minLength">6</param>
```

```
    <param name="maxLength">8</param><message>
用户名长度必须在{minLength}~{maxLength}位之间
</message>
    </validator>
    <!检验用户名是否已输入>
    <validator type="requiredstring">
    <param name="fieldName">username</param><
message>请输入用户名</message>
    </validator>
    <!检验密码的长度>
    <validator type="stringlength">
    <param name="fieldName">password</param><
param name="minLength">6</param>
    <param name="maxLength">8</param><message>
密码长度必须在{minLength}~{maxLength}位之间
</message>
    </validator>
    <!检验密码是否已输入>
    <validator type="requiredstring">
    <param name="fieldName">password</param><
message>请输入密码</message>
    </validator>
    <!检验确认密码的长度>
    <validator type="stringlength">
    <param name="fieldName">repassword</param><
param name="minLength">6</param>
    <param name="maxLength">8</param>
    <message>确认密码长度必须在{minLength}
~{maxLength}位之间</message>
    </validator>
    <!检验确认密码是否已输入>
    <validator type="requiredstring">
    <param name="fieldName">repassword</param>
    <message>请输入确认密码</message>
    </validator>
    <!检验密码和确认密码的输入内容是否一致>
    <validator type="fieldexpression">
```

```
    <param name="fieldName">password</param>
    <param name="fieldName">repassword</param>
    <param name="expression">password==repassword
</param>
    <message>确认密码和密码输入不一致</message>
    </validator>
    <! 检验生日是否已输入>
    <validator type="required">
    <param name="fieldName">birthday</param>
    <message>请输入生日日期</message>
    </validator>
    <! 检验输入日期是否在一个有效日期范围内>
    <validator type="date">
    <param name="fieldName">birthday</param>
    <param name="min">19280101</param>
    <param name="max">20040101</param>
    <message>输入生日日期无效</message>
    </validator>
    <! 检验手机号码是否已输入>
    <validator type="requiredstring">
    <param name="fieldName">mobile</param>
    <message>请输入手机号码</message>
    </validator>
    <! 检验输入手机号码长度是否是11位有效手机号码>
    <validator type="stringlength">
    <param name="fieldName">mobile</param>
    <message>请输入正确的手机号码，号码位数必须为11位
</message>
    </validator>
    <! 检验年龄是否已输入>
    <validator type="required">
    <param name="fieldName">age</param>
    <message>请输入年龄</message>
    </validator>
    <! 检验输入年龄是否符合特定年龄范围>
    <validator type="int">
    <param name="fieldName">age</param>
```

```
    <param name="min">1</param>
    <param name="max">80</param>
    <message>年龄必须在{min} {max}岁之间</message
>
    </validator>
    </validators>
```

源程序解读

（1）非字段格式的输入校验方式与上一小节相比主要是它的书写格式不像之前在<validators>和</validators>之间包含<field>标签，而是<validator>标签。每个<validator>标签定义的type属性还是Struts 2自带的输入校验器的类型格式。

（2）在<validator>标签之内包含的还是<param>和<message>两种标签。但是在<param>中多定义了一个fieldName属性。这个属性定义的就是输入校验的表单字段名字，其他和之前字段格式都类似。

（3）由于所有数据输入校验出错的错误信息和8.1节相同，因此所有的校验出错的信息显示读者可以参看图8.1到图8.4。

8.2.3 Struts 2输入校验出错信息的国际化配置形式

技术要点

Struts 2输入校验配置文件中定义了＜message＞标签，该标签定义了很多输入校验的出错信息。之前是使用硬编码写在配置文件中，其实可以使用Struts 2的国际化配置将这些错误信息写在属性文件中。本小节介绍如何将错误信息定义在属性文件中，也为之后Struts 2国际化章节做个基础介绍。

配置文件中修改代码介绍，介绍出错信息的key属性定义。

使用中文的属性文件出错信息定义代码。

实现代码

配置文件代码如下：

```
    <! ……文件名：
RegisterActionvalidation.xml……>
    <validators>
    <! 检验输入用户名长度是否在一个有效范围内>
    <validator type="stringlength">
    <param name="fieldName">username</param>
    <param name="minLength">6</param>
    <param name="maxLength">8</param>
    <message key="userNameLengthRange"></message
>
    </validator>
    <! 检验用户名是否已输入>
    <validator type="requiredstring">
    <param name="fieldName">username</param>
    <message key="userNameRequired"></message>
    </validator>
    <! 检验密码的长度>
    <validator type="stringlength">
    <param name="fieldName">password</param>
    <param name="minLength">6</param>
    <param name="maxLength">8</param>
    <message key="passwordLength"></message>
    </validator>
    <! 是否已经输入密码>
    <validator type="requiredstring">
    <param name="fieldName">password</param>
    <message key="passwordRequired"></message>
    </validator>
    <! 检验确认密码的长度>
    <validator type="stringlength">
    <param name="fieldName">repassword</param>
    <param name="minLength">6</param>
    <param name="maxLength">8</param>
```

```
    <message key="repasswordLength"></message>
    </validator>
    <! 是否已经输入确认密码>
    <validator type="requiredstring">
    <param name="fieldName">repassword</param>
    <message key="repasswordRequired"></message>
    </validator>
    <! 检验密码和确认密码的输入内容是否一致>
    <validator type="fieldexpression">
    <param name="fieldName">password</param>
    <param name="fieldName">repassword</param>
    <param name="expression">password==repassword
</param>
    <message key="repasswordEquals"></message>
    </validator>
    <! 是否已经输入日期>
    <validator type="required">
    <param name="fieldName">birthday</param>
    <message key="birthdayRequired"></message>
    </validator>
    <! 检验输入日期是否在一个有效日期范围内>
    <validator type="date">
    <param name="fieldName">birthday</param>
    <param name="min">19280101</param>
    <param name="max">20040101</param>
    <message key="birthdayRange"></message>
    </validator>
    <! 检验是否输入手机号码>
    <validator type="requiredstring">
    <param name="fieldName">mobile</param>
    <message key="mobileRequired"></message>
    </validator>
    <! 检验输入手机号码长度是否正确>
    <validator type="stringlength">
    <param name="fieldName">mobile</param>
    <message key="mobileLength"></message>
    </validator>
```

```
    <! 检验年龄是否输入>
    <validator type="required">
    <param name="fieldName">age</param>
    <message key="ageRequired"></message>
    </validator>
    <! 检验输入年龄是否符合特定年龄范围>
    <validator type="int">
    <param name="fieldName">age</param>
    <param name="min">1</param>
    <param name="max">80</param>
    <message key="ageRange"></message>
    </validator>
    </validators>
```

支持中文的国际化配置属性文件代码如下：

```
    <! ……文件名：messageResource.properties……>
    #配置文件中<message>的key属性定义
userNameLengthRange=用户名长度必须在{minLength}{maxLength}位之间
    userNameRequired=请输入用户名
    passwordLength=密码长度必须在{minLength}{maxLength}位之间passwordRequired=请输入密码
    repasswordLength=确认密码长度必须在{minLength}{maxLength}位之间
    repasswordRequired=请输入确认密码
    repasswordEquals=确认密码和密码输入不一致
    birthdayRequired=请输入生日日期
    birthdayRange=输入生日日期无效
    mobileRequired=请输入手机号码
    mobileLength=请输入正确的手机号码，号码位数必须为11位
    ageRequired=请输入年龄
    ageRange=年龄必须在{min}{max}岁之间
```

源程序解读

（1）在配置文件中对＜message＞标签做了修改。使用了key属性，这些key属性定义的内容都是出错信息名字，而这些名字是在国际化配置的属性文件中定义的。这里笔者使用了非字段定义的格式来写配置文件中的代码。其实字段格式也是相同，都是修改＜message＞标签中的内容。

（2）在支持中文的国际化配置属性文件“messageResource.properties”中，笔者对所有key属性定义的出错信息名字做了一一具体相关的出错信息定义。因为该文件是中文国际化的属性文件，因此都是中文的出错信息。

（3）有关Struts 2国际化的配置在后面章节会具体完整的介绍，这里只是让读者知道输入校验的出错

信息都是可以国际化的赋予各个语言版本。

8.3　集合类型输入校验介绍

与之前类型转化章节中记述的集合类型的对象类型转化相同。Struts 2也支持集合类型对象的输入校验。因此批量的对同一对象包含的属性数据的输入校验也是笔者需要向读者说明的，读者可以与类型转化中相关章节一起对照来学习。

8.3.1　Struts 2中单个Java对象的输入校验形式

技术要点

在介绍集合类型的输入校验之前，必须向读者介绍单个Java对象输入校验的形式。因为每个集合类型中包含的元素都是一个个单独的java对象，因此Java

对象的输入校验是集合类型对象数据输入校验的基础，请读者必须掌握。

Visitor校验器的介绍和使用方式。

Action和Java对象的输入校验配置文件介绍。

实现代码

还是使用第7章添加材料的示例。具体的Material对象代码和第7章相同。Action代码如下：

```
    <！……文件名：AddMaterialAction.java……>
    ……
    public class AddMaterialAction extends ActionSupport {
    //属性类型需要输入校验的材料对象
    private Material material;
    public Material getMaterial（） {
    return material;
    }
    public void setMaterial（Material material） {
    this.material=material;
    }
    public String execute（）throws Exception {
    return SUCCESS;
```

```
    }
    }
```

添加材料和显示添加材料成功的JSP视图界面代码也和第7章相同。修改Action的输入校验配置文件AddMaterialActionvalidation.xml文件，代码如下：

```
    <! ……文件名：
AddMaterialActionvalidation.xml……>
    <validators>
    <field name="material">
    <! 单个JAVA对象校验>
    <fieldvalidator type="visitor">
    <param name="context">materialContext</param
>
    <param name="appendPrefix">true</param>
    <message>添加材料输入校验：</message>
    </fieldvalidator>
    </field>
    </validators>
```

除了Action校验配置文件之外，本示例中还要增加一个对于Material这个Java对象的输入校验配置文件，名字叫

MaterialmaterialContextvalidation.xml，至于为什么取这个名字稍后在代码解释中说明。代码如下：

```
    <！……文件名：
MaterialmaterialContextvalidation.xml……>
    <validators>
    <field name="material">
    <！校验材料是否输入>
    <fieldvalidator type="requiredstring">
    <message>请输入材料名</message>
    </fieldvalidator>
    </field>
    <field name="bid">
    <！校验价格是否输入>
    <fieldvalidator type="double">
    <param name="minExclusive">0.1</param>
    <message>请输入价格</message>
    </fieldvalidator>
    </field>
    <field name="mount">
    <！校验库存量是否输入>
    <fieldvalidator type="int">
    <param name="min">1</param>
    <message>请输入库存量</message>
    </fieldvalidator>
    </field>
    <field name="expireDate">
    <！校验过期日期是否输入>
    <fieldvalidator type="required">
    <message>请输入过期日期</message>
    </fieldvalidator>
    <！校验过期日期是否在指定日期范围内>
    <fieldvalidator type="date">
    <param name="min">20090101</param>
```

```
    <param name="max">20190101</param>
    <message>输入过期日期无效</message>
    </fieldvalidator>
    </field>
    </validators>
```

注意：该输入校验配置文件要和Material这个Java对象的代码文件放在同一目录下。

笔者还增加了一个struts.properties文件，方便输入支持本示例的字符编码集GB2312。代码如下：

```
    <!……文件名：
struts.properties……>
    #支持本地化的资源文件名定义
    struts.i18n.encoding=gb2312
```

如图8.5所示，没有输入任何信息时的输入校验错误信息提示。如果输入的过期日期不是MaterialmaterialContextvalidation.xml文件中定义的日期范围时出错信息如图8.6所示。

图　8.5　输入校验发现数据没有进行任何输入

图　8.6　输入校验发现日期不符合所定义的日期范围

源程序解读

（1）其实本示例代码很多都和类型转换中复合类型对象的类型转换示例很相似。所不同的就是两个输入校验时使用的xml文件。

（2）第一个文件是Action的校验文件。该文件中由于Action的私有变量是一个Material对象。因此代码中fieldname是该变量名字，定义了两个参数。一个是context参数，所定义的名字是有开发者自己自由定义。但是之后Material对象的输入校验配置文件名字中必须有这个名字。而appendPrefix参数缺省是false，定义为true时则表明在输入校验出错信息之前可以加上message所的定义内容。

图8.5和图8.6也显示了每个出错信息前都有message中定义的“添加材料输入校验：”这几个字，其原因就是笔者把appendPrefix参数设置了true才会有这样的效果。

（3）第二个文件是本示例重点，这个输入校验配置文件名字命名格式是“Java对象名context参数validation.xml”，而在本示例中是

“MaterialmaterialContextvalidation.xml”，而且之前也已经说了该文件一定要和Java对象的类代码文件放在同一目录下。其中对输入校验规则的定义和前几节类似，都是使用了Struts 2内置的输入校验器。其中用到了一个前几节没有用到的double类型，在后面的小节综合Struts 2内置的输入校验器时一起说明。

8.3.2 Struts 2对象集合即批量输入的校验形式

技术要点

Struts 2中也支持对List、Set等数据集合的输入校验，在视图页面上即是对同一Java对象进行批量的输入。这里笔者只介绍List数据集合类型的校验。其他数据集合类型校验依此类推。

批量输入校验格式和相关注意点。

实现代码

其实和上一小节代码是相同的。这里只将不同的代码罗列出来。支持List类型转换的属性文件如下：

```
<！……文件名：
AddMaterialActionconversion.properties……>
```

```
#集合属性-List类型
Element__materialList=com.model.Material
```

Action代码如下：

```
<！……文件名：
AddMaterialAction.java……>
public class AddMaterialAction extends
ActionSupport {
private List<Material>materialList;
public String execute（）throws Exception {
return SUCCESS;
}
public List<Material>getMaterialList（） {
return materialList;
}
public void setMaterialList（List<Material>
materialList） {
this.materialList=materialList;
}
}
```

输入的JSP视图界面代码如下：

```
<！……文件名：
addMaterial.jsp……>
<s: form
id="materialForm"action="addMaterial"theme="simple
">
<table>
<tr>
```

```
    <td>材料名</td>
    <td>价格</td>
    <td>库存量</td>
    <td>过期日期</td>
    </tr>
    <s: iterator value="new int[4]"status="m">
    <tr>
    <td><s: textfield name="% {' materialList['
+#m.index+' ].material'} "/></td>
    <td><s: textfield name="% {' materialList['
+#m.index+' ].bid'} "/></td>
    <td><s: textfield name="% {' materialList['
+#m.index+' ].mount'} "/></td>
    <td><s: datetimepicker name="% {'
materialList[' +#m.index+' ].expireDate'} "/></td
>
    </tr>
    </s: iterator>
    <tr>
    <td colspan="4"><s: submit value="提交"></s:
submit>
    </tr>
    </table>
    </s: form>
```

输入校验显示出错信息如图8.7所示。

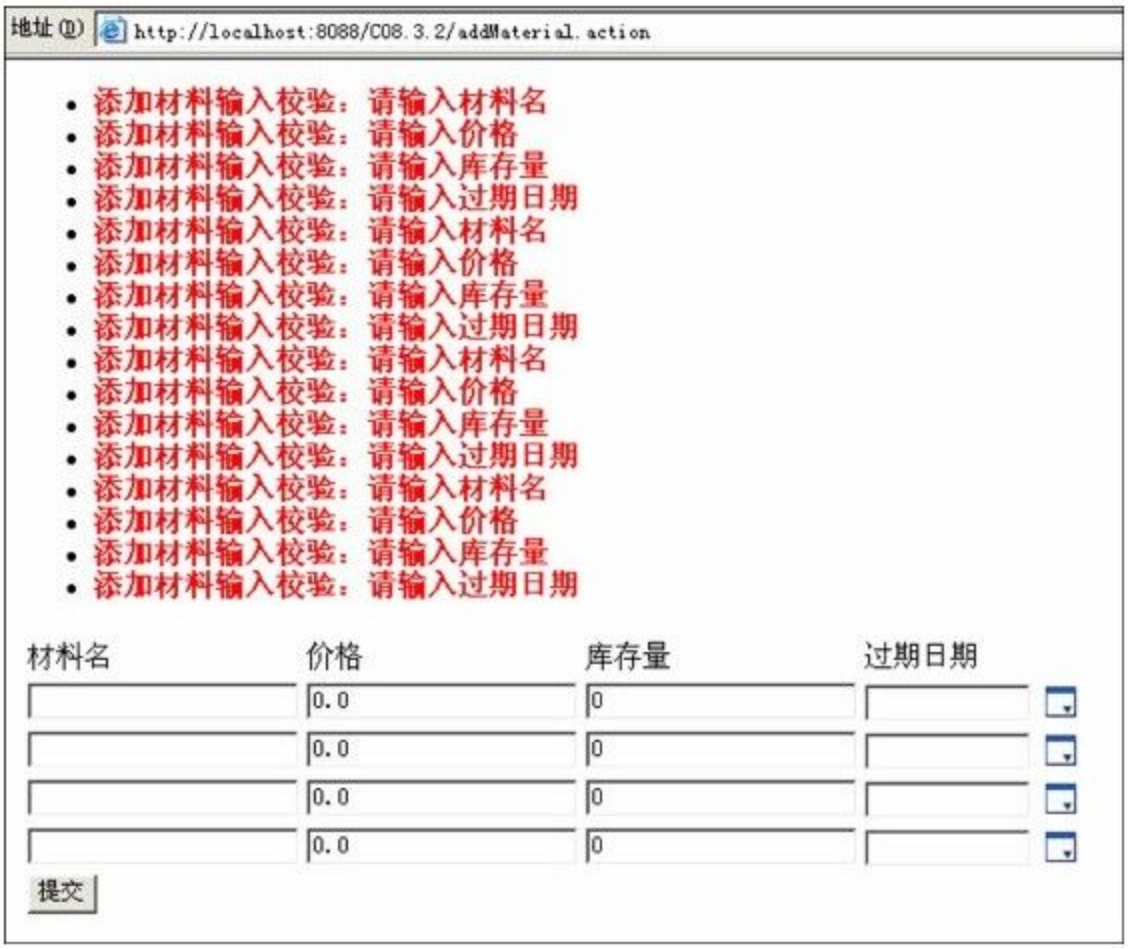

图　8.7　输入校验所有数据为输入时候的出错信息

源程序解读

（1）Action中私有变量是List类型的Material对象的集合。但是Action和Material对象的输入校验配置文件内容和前一小节相同。

（2）AddMaterialActionconversion.properties文件定义了List的元素都是Material对象（请读者翻阅List集合类型转换章节，回忆一下这么做的原因）。

（3）批量输入材料的JSP视图界面由代码可知和前一章类型转换中的界面相同。笔者在本章一开始就说明了类型转换其实也是输入校验的一种，从这里也可以佐证这一论点，在下一小节也有说明。

（4）图8.7显示了所有信息未输入时的出错信息，表明批量输入校验的确是在起作用。

8.4 Struts 2输入校验器大全

在前几节输入校验的配置文件中，读者应该看到很多类型的Struts 2的输入校验器。下面就对这些输入校验器做详细介绍。

技术要点

介绍Struts 2自带的输入校验器。

实现代码

在xwork2.0.4.jar包中，请读者在\com\opensymphony\xwork2\validator\validators路径下找一个名字为“default.xml”的xml文件。在该文件中有所有Struts 2自带的输入校验器定义。具体代码如下：

```
    <! ……文件名：
default.xml……>
    ……
    <validators>
    <validator name="required"
    class="com.opensymphony.xwork2.validator.valida
tors.RequiredFieldValid
    ator"/>
    <validator name="requiredstring"
    class="com.opensymphony.xwork2.validator.valida
tors.RequiredStringVali
    dator"/>
    <validator name="int"
    class="com.opensymphony.xwork2.validator.valida
tors.IntRangeFieldValid
    ator"/>
    <validator name="double"
    class="com.opensymphony.xwork2.validator.valida
tors.DoubleRangeFieldVa
    lidator"/>
    <validator name="date"
    class="com.opensymphony.xwork2.validator.valida
tors.DateRangeFieldVali
    dator"/>
    <validator name="expression"
    class="com.opensymphony.xwork2.validator.valida
tors.ExpressionValidator"/>
    <validator name="fieldexpression"
    class="com.opensymphony.xwork2.validator.valida
tors.FieldExpressionVal
    idator"/>
    <validator name="email"
    class="com.opensymphony.xwork2.validator.valida
tors.EmailValidator"/>
    <validator name="url"
    class="com.opensymphony.xwork2.validator.valida
tors.URLValidator"/>
```

```
    <validator name="visitor"
    class="com.opensymphony.xwork2.validator.valida
tors.VisitorFieldValida
    tor"/>
    <validator name="conversion"
    class="com.opensymphony.xwork2.validator.valida
tors.ConversionErrorFie
    ldValidator"/>
    <validator name="stringlength"
    class="com.opensymphony.xwork2.validator.valida
tors.StringLengthField
    Validator"/>
    <validator name="regex"
    class="com.opensymphony.xwork2.validator.valida
tors.RegexFieldValidator"/>
    </validators>
```

以上代码是所有Struts 2输入校验器的定义，下面将这些输入校验器（一共有13个）的字段和非字段格式的校验形式写在如下，本例中没有程序代码示例，所有输入校验器的应用代码都是笔者自己定义的。

```
    <! 必填校验>
    <! 非字段校验>
    <validator type="required">
    <param name="fidleName">field</param>
    <message>请输入数据</message>
    </validator>
```

```
    <! 字段校验>
    <field name="field">
    <fieldvalidator type="required">
    <message>请输入数据</message>
    </fieldvalidator>
    </field>
    <! 必填字符串校验>
    <! 非字段校验>
    <validator type="requiredstring">
    <param name="fidleName">field</param>
    <param name="trim">true</param>
    <message>请输入数据</message>
    </validator>
    <! 字段校验>
    <field name="field">
    <fieldvalidator type="requiredstring">
    <param name="trim">true</param>
    <message>请输入数据</message>
    </fieldvalidator>
    </field>
    <! 整数校验>
    <! 非字段校验>
    <validator type="int">
    <param name="fidleName">field</param>
    <param name="min">1</param>
    <param name="max">80</param>
    <message>数字必须在 {min} {max} 岁之间</message
>
    </validator>
    <! 字段校验>
    <field name="field">
    <fieldvalidator type="int">
    <param name="min">1</param>
    <param name="max">80</param>
    <message>数字必须在 {min} {max} 岁之间</message
>
    </fieldvalidator>
```

```
</field>
<! 浮点校验>
<! 非字段校验>
<validator type="double">
<param name="fidleName">field</param>
<param name="minExclusive">0.1</param>
<param name="maxExclusive">10.1</param>
<message>输入浮点无效</message>
</validator>
<! 字段校验>
<field name="field">
<fieldvalidator type="double">
<param name="minExclusive">0.1</param>
<param name="maxExclusive">10.1</param>
<message>输入浮点无效</message>
</fieldvalidator>
</field>
<! 日期校验>
<! 非字段校验>
<validator type="date">
<param name="fidleName">field</param>
<param name="min">20090101</param>
<param name="max">20190101</param>
<message>日期无效</message>
</validator>
<! 字段校验>
<field name="field">
<fieldvalidator type="date">
<param name="min">20090101</param>
<param name="max">20190101</param>
<message>日期无效</message>
</fieldvalidator>
</field>
<! 表达式校验>
<! 非字段校验>
<validator type="expression">
```

```
    <param name="expression">password==repassword
</param>
    <message>两者输入不一致</message>
    </validator>
    <! 字段表达式校验>
    <! 非字段校验>
    <validator type="fieldexpression">
    <param name="expression">password==repassword
</param>
    <message>两者输入不一致</message>
    </validator>
    <! 字段校验>
    <field name="field">
    <fieldvalidator type="fieldexpression">
    <param name="expression"><!
[CDATA[#password==#repassword]]></param>
    <message>两者输入不一致</message>
    </fieldvalidator>
    </field>
    <! 邮件校验>
    <! 非字段校验>
    <validator type="email">
    <param name="fidleName">field</param>
    <message>非法邮件地址</message>
    </validator>
    <! 字段校验>
    <field name="field">
    <fieldvalidator type="email">
    <message>非法邮件地址</message>
    </fieldvalidator>
    </field>
    <! 网址校验>
    <! 非字段校验>
    <validator type="url">
    <param name="fidleName">field</param>
    <message>无效网址</message>
    </validator>
```

```
<! 字段校验>
<field name="field">
<fieldvalidator type="url">
<message>无效网址</message>
</fieldvalidator>
</field>
<! visitor校验>
<! 非字段校验>
<validator type="visitor">
<param name="fidleName">field</param>
<param name="context">fieldContext</param>
<param name="appendPrefix">true</param>
<message>输入校验</message>
</validator>
<! 字段校验>
<field name="field">
<fieldvalidator type="visitor">
<param name="context">fieldContext</param>
<param name="appendPrefix">true</param>
<message>输入校验</message>
</fieldvalidator>
</field>
<! 类型转换校验>
<! 非字段校验>
<validator type="conversion">
<param name="fidleName">field</param>
<message>类型转换错误</message>
</validator>
<! 字段校验>
<field name="field">
<fieldvalidator type="conversion">
<message>类型转换错误</message>
</fieldvalidator>
</field>
<! 字符串长度校验>
<! 非字段校验>
<validator type="stringlength">
```

```
    <param name="fidleName">field</param>
    <param name="minLength">1</param>
    <param name="maxLength">10</param>
    <param name="trim">true</param>
    <message>字符串长度必须为10位</message>
    </validator>
    <! 字段校验>
    <field name="field">
    <fieldvalidator type="stringlength">
    <param name="minLength">1</param>
    <param name="maxLength">10</param>
    <param name="trim">true</param>
    <message>字符串长度必须为10位</message>
    </fieldvalidator>
    </field>
    <! 正则表达式校验>
    <! 非字段校验>
    <validator type="regex">
    <param name="fidleName">field</param>
    <param name="expression"><! [CDATA[（/ ^
13[13567890]（ \d {8} ）/）]]></param>
    <message>手机号码必须为数字并且是11位</message>
    </validator>
    <! 字段校验>
    <field name="field">
    <fieldvalidator type="regex">
    <param name="expression"><! [CDATA[（/ ^
13[13567890]（ \d {8} ）/）]]
    ></param>
    <message>手机号码必须为数字并且是11位</message>
    </fieldvalidator>
    </field>
```

源程序解读

（1）必填校验器required是用来判断输入的字段是否为空。如果未输入任何数据则会显示错误信息。fieldName属性是指定校验的字段名。这个属性是所有Struts 2自带的输入校验器都具有的属性。因此介绍其他输入校验器时，笔者略过不谈。但是读者自己要知道该属性是输入校验器共有的。该校验器其他标签在前几章节中有过介绍，笔者也略过不谈。

（2）必填字符串校验器requiredstring用来判断输入字段是否是一个非空字符串。如果不是也显示错误信息。其中的trim属性是在校验之前对字符串进行处理。默认是“true”。

（3）整数校验器int判断输入的字段数据是在一个整数范围内。min属性是最小值，max是最大值。＜message＞标签内可用“${属性名}”格式类表示他们具体的值。

（4）浮点校验器double是判断输入的字段数据是在一个浮点数范围内。minInclusive表示这个范围的最小值。max Inclusive表示这个范围的最大值。还有minExclusive和maxExclusive两个属性，前者表明在浮点范围之外的最小值，后者表示是在浮点范围之外的最大值。注意：以上四个属性如果没有声明，则输入校验不会去检查。

（5）日期校验器date判断输入的字段的日期值是否在一个日期范围内。min是该范围的最小值，max是最大值。他们两个属性也和浮点校验器的四个属性相同，如果没有声明则输入校验不检查。

（6）表达式校验器expression只有非字段校验格式。不能在字段校验中声明。它的属性也是expression。如代码所示开发者可以使用OGNL表达式来定义校验规则。

（7）字段表达式校验器fieldexpression判断字段是否满足一个表达式。如代码所示，当用来判断输入的密码和确认密码值是否一致就可以使用该校验器。它的属性expression和表达式校验器相同。不过它可以用在字段校验中。

（8）邮件校验器email来判断输入的字段是一个email时是否符合email的格式。

（9）网址校验器url来判断输入的字段是一个网址时是否符合网址的格式。

（10）visitor校验器就是判断集合类型的字段。前面章节有所介绍。这里再重申一下context属性是可以应用的集合类型中元素对象的别名。appendPrefix属性是指定在错误信息中前面是否加上特定前缀。该前缀内容可在＜message＞标签中定义。另外这两个属性没有声明时，校验器也不会去执行检查。

（11）类型转换校验器conversion用来判断输入字段是否进行类型转换。它有一个repopulateField属性，如果为true表明如果发生类型转换错误，返回到struts.xml中指定的Action的input视图界面时还是否显示原来错误的输入内容。值为false则相反。从这点可以看出类型转换也属于输入校验的一种是有理论依据的。

（12）字符串长度校验器stringlength用于判断输入的字符串长度是否是指定的长度范围。其中minLength是最小字符串长度，maxLength是最大字符串长度。trim属性和上述必填字符串校验器requiredstring中trim属性拥有相同的功能。这三者也属于不声明就不执行检查的可选属性。

（13）正则表达式校验器regex检查字段输入值是否和一个正则表达式匹配。expression属性中的内容

就是该正则表达式。还有个caseSensitive属性，为true则表明匹配时对字母大小写敏感，反之则不敏感。如代码所示用了一个判断输入的值是否是11位、全部由数字组成的正则表达式。对于输入手机号码数据的字段，该校验规则是最适用的。

第9章　Struts 2国际化

在第1章综述中，已经将国际化的基本概念做了简单说明。国际化即internationalization，因为在这个单词第一个字母i和最后一个字母n之间有18个字母，所以通常又被称为i18n。本章扩展国际化内容，向读者展示在Struts 2中应用国际化的各种范例。

9.1　Struts 2国际化基础应用

本节首先介绍国际化在Struts 2中的基础使用方法。主要介绍基础使用方式的原理以及如何使用占位符号来实现Web项目国际化功能的开发。

9.1.1　国际化基础使用方式

技术要点

以登录功能为例，介绍如何实现中文、英文的国际化。

国际化属性文件定义原理。

ActionSupport类中getText方法的基础说明。

视图界面中如何显示国际化信息。

实现代码

使用的Action文件：

```
    <！……文件名：
LoginAction.java……>
    public class LoginAction extends
ActionSupport {
    ……
    public String execute（）throws Exception {
    username=getUsername（）；//属性值即JSP页面上输入的
值
    password=getPassword（）；//属性值即JSP页面上输入的
值
    try {
    //判断输入值是否是空对象或没有输入
    if（username！=null＆＆！username.equals（""）＆＆
password！=null＆＆！pass
```

```
    word.equals（""））{
    ActionContext.getContext（）.getSession（）.put（
"user"，getUsername（））；
    //打印getText方法，取得属性文件中定义的值
    System.out.println（getText（"username"）
+username）；
    System.out.println（getText（"password"）
+password）；
    FORWARD="success"；//根据标志内容导航到操作成功页面
    } else {
    FORWARD="input"；//根据标志内容导航到操作失败页面
    }
    } catch（Exception ex）{
    ex.printStackTrace（）；
    }
    return FORWARD;
    }
    ……
    }
```

属性文件中国际化定义：

```
    <！……文件名：
struts.properties……>
    #支持本地化的资源文件名定义
    struts.custom.i18n.resources=messageResource
```

支持中文的属性文件内容：

```
    <！……文件名：messageResource_zh_
CN.properties……>
```

```
#中文属性定义文件
#用key=value格式定义页面上显示的内容
username=用户名
password=密码
loginSubmit=登录
loginPage=登录页面
successPage=操作成功页面
welcome=欢迎您
user.required=请输入用户名!
pass.required=请输入密码!
```

支持英文的属性文件内容：

```
<!……文件名：messageResource_en_US.properties……>
#英文属性定义文件
#用key=value格式定义页面上显示的内容
username=User Name
password=User Password
loginSubmit=login
loginPage=login page
successPage=success page
welcome=Welcome You
user.required=please input your name!
pass.required=please input your password!
```

国际化的登录login.jsp文件：

```
<!……文件名：login.jsp……>
……
```

```
    <html>
    <head>
    <title><s: text name="loginPage"></s: text>
</title>
    </head>
    <body>
    <! form标签库定义，以及调用哪个Action声明>
    <s: form action="Login">
    <table width="60%"height="76"border="0">
    <! 各标签定义>
    <s: textfield name="username"key="username"/>
    <s: password name="password"key="password"/>
    <s: submit key="loginSubmit"align="center"/>
    </table>
    </s: form>
    </body>
    </html>
```

国际化的登录成功success.jsp文件：

```
    <! ……文件名：
success.jsp……>
    ……
    <head>
    <title><s: text name="successPage"></s: text
></title>
    </head>
    <body>
    <! 取得session中用户名值>
     {sessionScope.user}，<s: text name="welcome">
</s: text>
    </body>
    ……
```

中文的登录界面如图9.1所示。英文的登录界面如图9.2所示。中文登录成功界面如图9.3所示。英文登录成功界面如图9.4所示。LoginAction代码中打印在中文属性值控制台用户名和密码值如图9.5所示。LoginAction代码中打印在英文属性值控制台用户名和密码值如图9.6所示。

图　9.1　中文登录界面

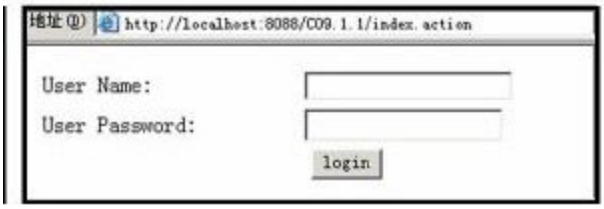

图　9.2　英文登录界面

图　9.3　中文登录成功界面

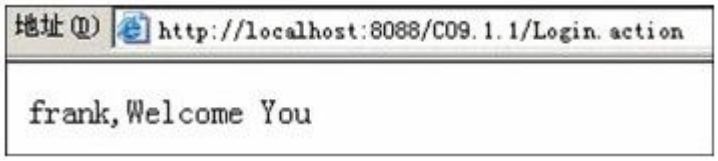

图　9.4　英文登录成功界面

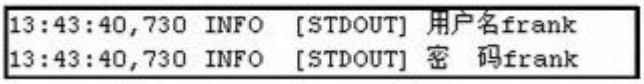

图　9.5　中文属性值控制台打印信息

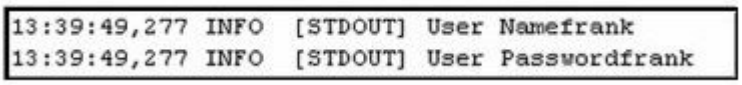

图　9.6　英文属性值控制台打印信息

源程序解读

（1）本节是Struts 2国际化最基本的应用。首先请读者阅读struts.properties文件中的代码。“struts.custom.i18n.resources”并不是随意取的名字，它是Struts 2设计者定义的使用全局国际化属性文件的常量名。如果对它进行了定义，则开发者可以在Web项目中使用类似XXX__language__

country.properties这样的属性文件名格式来创建在不同语言环境中的属性文件。

这也是在上述代码中有“messageResource__zh__CN.properties”和“messageResource__en__US.properties”这两个各支持中文和英文的属性文件名的由来。因为笔者在struts.properties文件中定义了“struts.custom.i18n.resources”值为messageResource，这样messageResource作为属性文件名，可以实现各个支持不同国家语言的国际化属性文件。

注意：关于国际化属性文件名的定义不仅仅可以在struts.properties中定义，也可以在struts.xml和web.xml这两个配置文件中定义。之所以不在这里介绍，具体原因在第4章介绍如何实现上传下载功能的拦截器篇章已经讲述，这里不再重复说明。如果读者对

在两个xml文件中定义国际化属性文件名兴趣，可自行查阅其他技术资料。

（2）LoginAction继承ActionSupport类，可以使用ActionSupport中getText方法。这里getText方法中的字符串参数都是在国际化属性文件中定义的key。之前章节也说明过属性文件中的定义格式都是以“key=value”格式定义。getText方法得到key作为方法参数，返回的结果就是value。从图9.5和图9.6也可以看出在两个国际化属性文件中“用户名”和“密码”的value是不相同的，因此在不同的语言环境下得到的值就不同了。一个是中文，另一个是英文。

注意：getText方法在ActionSupport中是重载方法。因此它还有几个重名的getText方法，具体使用在有占位符的国际化属性文件中。稍后章节将具体介绍。

（3）在login.jsp和success.jsp中，笔者使用Struts 2标签＜text＞，该标签的name属性定义为属性文件中的“key”，在Struts 2的Action中getText方法就可以将value的值显示在视图界面上。而在其他Struts 2标签中，增加key属性的定义，也可以将国际化属性文件中的value值作为各个标签的值显示在视图界面上。

比如“＜s：textfield name=″username″key=″username″/＞”在原先可以使用label属性定义显示值，现在使用key属性，在不同语言环境下username的key得到的value值就不同，见图9.1和图9.2。

（4）Struts 2中有个名字为i18n的拦截器，它的使用目的就是在Action执行前查看Session中的request__locale参数的值是什么。如果是“zh__CN”

它就将之后执行的所有Action的语言环境设置为中文语言环境。如果是“en__US”则是英文语言环境。在本示例和接下来的示例中，读者可以在操作系统下的“控制面板→区域和语言选项”中选择区域来查看在英文和中文语言环境下示例显示的异同。

当然也可以直接在浏览器的地址栏中在action请求后面写上“？request__locale=zh__CN”这样的数据。具体可以这么写的原因就是通过手工输入的request__locale值，让i18n拦截器得到该值并进行相关语言设置。

9.1.2　占位符国际化使用方式

占位符的使用目的，是可以让开发者可以动态地填入某些国际化的值。也许这句话读者看的不是很明白。试举一例，在我们平时日常语言中常常会说“我……”。在“我”之后，我们可以根据不同情况加上不同的内容。因此占位符就充当了“我”之后的那些内容表达意义的角色。在不同语言环境和不同的业务逻辑下，占位符所代表的含义可以千变万化。相应地也可以说动态地显示了不同的信息。在Struts 2国际化应用中，使用占位符可以展现各种语言字符。

技术要点

以登录功能为例，介绍占位符的国际化。

国际化属性文件中占位符的使用方式。

多个getText方法介绍。

text标签中指定占位符位置。

实现代码

使用的Action文件：

```
    <！……文件名：
LoginAction.java……>
    ……
    public String execute（）throws Exception｛
    username=getUsername（）；//属性值即JSP页面上输入的
值
    password=getPassword（）；//属性值即JSP页面上输入的
值
    //定义getText方法需要的参数变量
    List valueList=new ArrayList（）；
    String[]valueArray=｛null, null｝；
    String defaultValue="default";
    try｛
    //判断输入值是否是空对象或没有输入
    if（username！=null＆＆！username.equals（""）＆＆
password！=null
    ＆＆！password.equals（""））｛
    ActionContext.getContext（）.getSession（）.put（
"user", getUsername（））；
    //对getText方法需要的参数变量赋予开发需要的值
    valueList.add（username）；
    valueList.add（password）；
    valueArray[0]=username;
```

```
    valueArray[1]=password;
    //打印各个getText方法，显示占位符中的属性值
    System.out.println（getText（"successMessage"））
;
    System.out.println（getText（"successMessage",
valueList））;
    System.out.println（getText（"successMessage",
valueArray））;
    System.out.println（getText（"successMessage",
defaultValue, valueList））;
    System.out.println（getText（"successMessage",
defaultValue, valueArray））;
    FORWARD="success"; //根据标志内容导航到操作成功页面
    } else {
    FORWARD="input"; //根据标志内容导航到操作失败页面
    }
    } catch（Exception ex） {
    ex.printStackTrace（）;
    }
    return FORWARD;
    public class LoginAction extends ActionSupport {
    }
    ……
    }
```

使用占位符的中文属性文件内容：

```
    ＜！……文件名：messageResource__zh__
CN.properties……＞
    ……
    successMessage=｛0｝，欢迎您！您的密码为｛1｝，请注意
保存
```

使用占位符的英文属性文件内容：

```
    <！……文件名：messageResource__en__
US.properties……>
    ……
    successMessage={0}，Welcome! Your password
is {1}，please remember it。
```

使用占位符的登录成功success.jsp中文件：

```
    <！……文件名：
success.jsp……>
    ……
    <！取得属性文件中定义的值>
    <s：text name="successMessage">
    <！占位符{0}的值由用户名值填充>
    <s：param><s：property value="username"/>
</s：param>
    <！占位符{1}的值由密码值填充>
    <s：param><s：property value="password"/>
</s：param>
    </s：text>……
    ……
```

显示占位符的中文登录成功界面如图9.7所示。显示占位符的英文登录成功界面如图9.8所示。

LoginAction代码中打印在中文属性值控制台用户名和

密码值如图9.9所示。LoginAction代码中打印在英文属性值控制台用户名和密码值如图9.10所示。

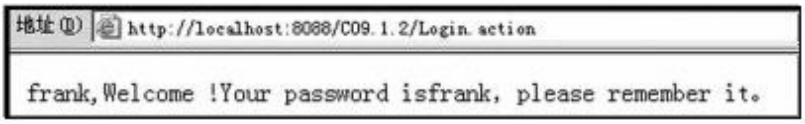

图　9.7　占位符中文登录成功界面

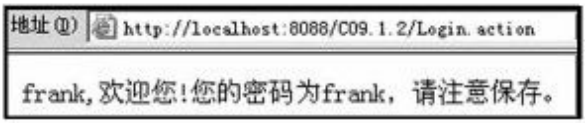

图　9.8　占位符英文登录成功界面

图　9.9　中文属性值控制台打印信息

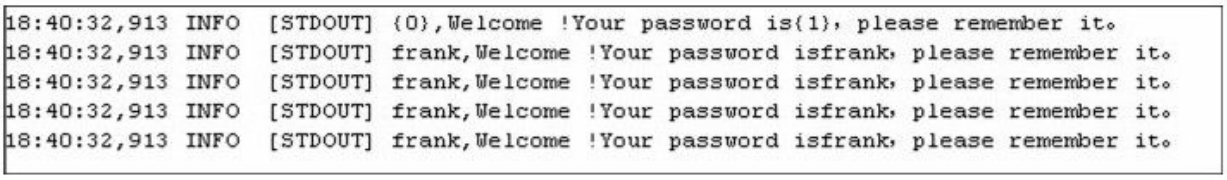

图　9.10　英文属性值控制台打印信息

源程序解读

（1）在本示例中在登录成功的显示信息中使用了两个占位符。占位符在属性文件中的写法都是以“{}”格式，第一个可以在“{}”中写上0，表示是第一个占位符位置。依此类推后面依次是“{1}”，“{2}”等。本示例中是想把登录成功后的用户名数据填充到“{0}”之中，将用户密码填充到“{1}”之中。所以在两个国际化属性文件中的“successMessage”的value值中都有“{0}”和“{1}”。

（2）之前也已说过LoginAction继承ActionSupport类后可以使用ActionSupport中getText方法。而getText方法是个重载方法，因此它的同名方法又很多，具体都是使用在有占位符的属性文件中。

将前一小节中介绍过的getText方法一起列出来。各个getText方法使用用途和方法申明如下：

String getText（String key）：上一小节已说明将属性文件中的key作为参数得到属性文件中value值。

String getText（String key, List valueList）：将属性文件中key映射的value中所有占位符封装成List变量，返回有占位符的value值。

String getText（String key, String[]valueArray）：将属性文件中key映射的value中所有占位符封装成字符串数组变量，返回有占位符的value值。

String getText（String key, String defaultValue, List valueList）：将属性文件中key映射的value中所有占位符封装成List变量，返回有占位符的value值。如果List变量为空或者没有将占位符代表的值赋予它，则返回缺省的defaultValue。

String getText（String key, String defaultValue, String[]valueArray）将属性文件中key映射的value中所有占位符封装成字符串数组变量，返回有占位符的value值。如果字符串数组变量为空或者没有将占位符代表的值赋予它，则返回缺省的defaultValue。

请读者查看本示例中LoginAction的打印方法，successMessage中如果有值赋予占位符则打印出这些值。如果没有，就如图9.9和图9.10第一行所示打印出“{0}”，“{1}”。而因为valueList和valueArray在LoginAction中都有值赋予，因此调用String getText（String key, String defaultValue, String[]valueArray）和String getText（String key, String defaultValue, List valueList）时打印出来的结果值并不是参数

defaultValue的值。如果没有则打印出来的内容应该为default，即Action中定义的defaultValue值。

（3）在success.jsp中，还是使用Struts 2标签＜text＞。只不过在text标签中加了＜param＞标签，在Struts 2标签章节中也已介绍过这个＜param＞，该标签可以嵌套在其他很多标签中作为参数定义。这里由于successMessage中有两个占位符定义，因此定义了两个＜param＞标签。使用＜property＞标签显示属性文件中username和password两个key映射的value值，见图9.7和图9.8。

9.2 Struts 2国际化使用范围说明

在上一小节中介绍的Struts 2国际化应用中，定义了两个国际化属性文件来应用国际化。读者应该注意到这两个属性文件都是直接放在src根目录下，而且都是根据struts.proerties文件中struts.custom.i18n.resources常量的定义来命名自己的国际化属性文件名。其实它们被称之为全局国际化信息属性文件。但是由于开发中的Web项目都很庞大，拥有很多数量的Java和JSP代码，因此对于整个Web项目的国际化，把所有信息都定义在全局属性文件中有很大的弊病。

其一是装载时需要浪费很多时间。

其二就是万一需要只更新某个Action或者JSP文件中的国际化属性定义，势必造成其他不需要更新的文

件一起进行变更，引出很多不必要的错误和麻烦。因此Struts 2设计者还引入了国际化使用范围的概念，让某些国际化属性文件只在某一特定范围内有效果。就算定义的key值相同，在不同的范围内Action和JSP显示的value值也各不相同。

9.2.1 Struts 2包范围属性文件国际化应用

技术要点

以介绍占位符章节的示例，介绍包范围国际化应用。

包范围属性文件命名。

实现代码

包范围的中文属性文件内容：

```
    <！……文件名：package__zh__
CN.properties……>
    #中文属性定义文件
    #用key=value格式定义页面上显示的内容
    username=用户名
    password=密码
    loginSubmit=登录
    loginPage=登录页面
    successPage=操作成功页面
    welcome=欢迎您
    user.required=请输入用户名！
    pass.required=请输入密码！
    successMessage=｛0｝，欢迎您！您的密码为｛1｝，请注意
保存
```

包范围的英文属性文件内容：

```
    <！……文件名：package__en__
US.properties……>
    #英文属性定义文件
    #用key=value格式定义页面上显示的内容
    username=User Name
    password=User Password
    loginSubmit=login
    loginPage=login page
    successPage=success page
    welcome=Welcome You
    user.required=please input your name!
    pass.required=please input your password!
    successMessage= {0} , Welcome! Your password
is {1} , please remember it
```

源程序解读

（1）以上节介绍占位符的示例作为本小节示例。这里只是将两个国际化属性文件的位置由src根目录下移到src/com文件夹下。相对于JSP页面没有影响。但是对于Java文件的国际化有很大影响。假设在src下新建其他文件夹，并在新建文件夹下创建了几个Java文件，则两个国际化属性文件并不会对这些新的Java文件进行国际化，而只对src/com下的Java文件进行国际化。

这样和全局国际化属性文件相比就更加松耦合，并且可以将src/com下特有的需要国际化的信息进行国际化，没必要写在全局国际化属性文件中。读者可以理解为将全局国际化属性文件分割成一个个包范围的国际化。

（2）将这两个属性文件转移目录后，名字也需要相应改变。这里都改为“package__language__country.properties”格式，而属性文件中的内容不必要再做修改。运行后的效果和之前雷同，因此这里也不再显示。

注意：一定要写成package，它是Struts 2中的固定名称，不能更改的，否则包范围的国际化会完全失效。

9.2.2 Struts 2Action范围属性文件国际化应用

技术要点

以介绍占位符章节的示例来介绍Action范围国际化应用。

Action范围属性文件命名。

实现代码

Action范围的中文属性文件内容：

```
    <!……文件名：LoginAction__zh__
CN.properties……>
    #中文属性定义文件
    #用key=value格式定义页面上显示的内容
    username=用户名
    password=密码
    loginSubmit=登录
    loginPage=登录页面
    successPage=操作成功页面
```

```
    welcome=欢迎您
    user.required=请输入用户名!
    pass.required=请输入密码!
    successMessage=｛0｝，欢迎您！您的密码为｛1｝，请注意保存
```

Action范围的英文属性文件内容：

```
    ＜！……文件名：LoginAction__en__US.properties……＞
    #英文属性定义文件
    #用key=value格式定义页面上显示的内容
    username=User Name
    password=User Password
    loginSubmit=login
    loginPage=login page
    successPage=success page
    welcome=Welcome You
    user.required=please input your name!
    pass.required=please input your password!
    successMessage= {0} , Welcome! Your password is {1} , please remember it
```

源程序解读

（1）Action范围的国际化就是指在Action同目录下设置国际化属性文件，然后只对该Action有国际化

效果。它是进一步细化了包范围的国际化属性文件，对于每个Action设置自己的国际化属性文件。

（2）Action范围内对某一Action设置国际化属性文件，文件名必须是XXXAction__language__country.properties格式，这里的XXXAction是该Action的类名。

（3）下面还是使用介绍占位符的示例来进行Action范围国际化的说明。这里只是把两个国际化属性文件移至和LoginAction同目录的文件夹下，并且将属性文件名改为“LoginAction__zh__CN.properties”和“LoginAction__en__US.properties”，属性文件中内容不变。

9.2.3 Struts 2临时范围属性文件国际化应用

技术要点

以介绍占位符章节的示例来介绍临时范围国际化应用。

临时范围属性文件命名。

JSP文件中制定临时文件存放目录。

实现代码

临时范围的中文属性文件内容：

```
    <！……文件名：struts__zh__
CN.properties……>
    #中文属性定义文件
    #用key=value格式定义页面上显示的内容
    username=用户名
```

```
    password=密码
    loginSubmit=登录
    loginPage=登录页面
    successPage=操作成功页面
    welcome=欢迎您
    user.required=请输入用户名!
    pass.required=请输入密码!
    successMessage=｛0｝，欢迎您！您的密码为｛1｝，请注意保存
```

临时范围的英文属性文件内容：

```
    <！……文件名：struts__en__US.properties……>
    #英文属性定义文件
    #用key=value格式定义页面上显示的内容
    username=User Name
    password=User Password
    loginSubmit=login
    loginPage=login page
    successPage=success page
    welcome=Welcome You
    user.required=please input your name!
    pass.required=please input your password!
    successMessage= {0} , Welcome! Your password is {1} , please remember it
```

制定临时国际化属性文件范围的登录login.jsp：

```
    <！……文件名：login.jsp……>
```

```
……
    <! name值指定com.example.struts目录下的struts打头的属性文件>
    <s: i18n name="com.example.struts.struts">
    <title><s: text name="loginPage"></s: text></title>
    </s: i18n>
    </head>
    <body>
    <! name值指定com.example.struts目录下的struts打头的属性文件>
    <s: i18n name="com.example.struts.struts">
    <! form标签库定义，以及调用哪个Action声明>
    <s: form action="Login">
    <table width="60%"height="76"border="0">
    <! 各标签定义>
    <s: textfield name="username"key="username"/>
    <s: password name="password"key="password"/>
    <s: submit key="loginSubmit"align="center"/>
    </table>
    </s: form>
    </s: i18n>
    ……
```

制定临时国际化属性文件范围登录成功的

success. jsp:

```
    <! ……文件名：success.jsp……>
    ……
    <! name值指定com.example.struts目录下的struts打头的属性文件>
    <s: i18n name="com.example.struts.struts">
```

```
    <title><s: text name="successPage"></s: text
>
    </title>
    </s: i18n>
    </head>
    <body>
    <! name值指定com.example.struts目录下的struts打头
的属性文件>
    <s: i18n name="com.example.struts.struts">
    <! 取得属性文件中定义的值>
    <s: text name="successMessage">
    <! 占位符 {0} 的值由用户名值填充>
    <s: param>
    <s: property value="username"/>
    </s: param>
    <! 占位符 {1} 的值由密码值填充>
    <s: param>
    <s: property value="password"/>
    </s: param>
    </s: text>
    </s: i18n>
    ……
```

源程序解读

（1）Struts 2中可以定义临时的国际化属性文件。然后让Action和JSP使用这些临时的国际化属性文件进行国际化。在本示例中可先阅读两个临时的国际化属性文件。它们存放在src/com/example/struts目

录下。具体内容和之前示例中的内容相同，只是把名字改成struts_zh_CN.properties和struts_en_US.properties，这里的“struts”是可以自由命名的，和包范围中国际化属性文件必须命名为“package”不同，本示例中只是命名为“struts”。

（2）在JSP中需要使用Struts 2的＜i18n＞标签来使临时国际化属性文件国际化生效。该标签在之前介绍Struts 2标签章节并没有介绍过。它的作用是在它本身的标签范围内访问临时国际化属性文件。其中有个name属性，该属性命名时一定要标明临时属性文件的存放路径。从src根目录开始，依次以Java包命名格式定义，直到临时国际化属性文件名字。

在本示例中，临时国际化属性文件名字命名的是struts，而它们是存放在src/com/example/struts目

录下，以Java的包命名格式从src开始就是JSP代码中黑体标明的“com.example.struts.struts”。

注意：两个struts表达含义不同，一个是存放的文件夹名字，一个是临时国际化属性文件的名字。

（3）使用临时国际化属性文件可以在单个JSP中使用多个国际化属性文件，而且可以充分利用＜i18n＞标签来进行国际化。具体生成的效果和前述相同，此处不再显示。

值得说的是在这三种范围的国际化属性文件的应用中，开发者可以同时使用这几种范围的国际化应用。但是他们执行国际化是有先后顺序的。介绍如下：

首先国际化执行顺序要分Action方面和JSP（视图）两种方面。也就是说Action方面是Java类代码执

行国际化还是JSP执行国际化是有所区别的。执行国际化顺序也可以理解成Struts 2项目装载这些国际化属性文件的先后顺序。

JSP（视图）方面，JSP中没有＜i18n＞标签时：

查找sruts.properties中对“struts.custom.i18n.resources”定义的国际化属性文件即全局国际化属性文件。

如果找不到全局国际化属性文件中的key则把JSP中定义的key属性定义的值显示在页面上。有＜i18n＞标签时：

查找＜i18n＞name属性定义的临时国际化属性文件。

如果找不到临时国际化属性文件，则查找sruts.properties中对

“struts.custom.i18n.resources”定义的国际化属性文件即全局国际化属性文件。

如果还找不到则把JSP中定义的key属性定义的值显示在页面上。Action（Java类）方面：

查找Action范围的国际化属性文件。

如果找不到则查找该Action所属的包范围的国际化属性文件。

如果找不到则查找上一级目录的包范围的国际化属性文件。如果还找不到则一直往上找包范围的国际化属性文件。直到最顶层包范围。

如果都找不到则查找sruts.properties中对“struts.custom.i18n.resources”定义的国际化属性文件即全局国际化属性文件。

如果找不到全局国际化属性文件中的key则把JSP中定义的key属性定义的值显示在页面上。

由此可知Action方面国际化属性文件执行顺序比JSP方面要多几个步骤，而且Action范围和包范围国际化都要一一遍历寻找国际化属性文件。而且可以总结临时范围国际化主要应用在JSP（视图）方面的国际化。Action和包范围国际化应用在Action方面国际化。

9.3　用户主动选择国际化应用介绍

在之前章节需要在URL中自行定义request_locale参数值或者在操作系统中自行修改区域和语言选择来进行国际化。这样在用户体验度方面并不可取，因此可以在JSP或其他视图界面定义语言选项，用户只要在浏览器中自行点击语言选项链接就可以在适合自己的语言中进行业务等方面的操作。本节就介绍如何实现这样的国际化应用。

技术要点

还是以登录作为本节示例，只不过在登录界面提供选择中文或英文的语言选项。并且依旧有占位符的使用。

语言选择的Action定义。

JSP中语言选择的代码定义。

实现代码

国际化属性文件本示例使用的是全局属性文件，内容和之前章节中相同。在struts.xml配置文件中增加一个用于语言选择的Action，直接连向登录的login.jsp，并不需要Action代码，内容如下：

```
    <! ……文件名：struts.xml……>
    ……
    <! 切换中英文的语言设置的Action定义>
    <action name="loginLanguage">
    <result>/jsp/login.jsp</result>
    </action>
    ……
```

设置语言选项的login.jsp代码：

```
    <! ……文件名：login.jsp……>
    ……
    <! 指定URL为英文的语言设置>
    <s: url id="english"action="loginLanguage">
    <! 参数request_locale设置英文>
```

```
    <s: param name="request__locale">en__US</s:
param>
    </s: url>
    <! 英文语言设置的链接定义>
    <s: a href="% {english} ">English</s: a>
    <! 指定URL为中文的语言设置>
    <s: url id="chinese"action="loginLanguage">
    <! 参数request__locale设置中文>
    <s: param name="request__locale">zh__CN</s:
param>
    </s: url>
    <! 中文语言设置的链接定义>
    <s: a href="% {chinese} ">中文</s: a>
    ……
```

如图9.11所示，有语言选项的登录页面。如图9.12所示，选择英文后的登录界面，注意查看地址栏中的URL。

图　9.11　语言选项的登录界面

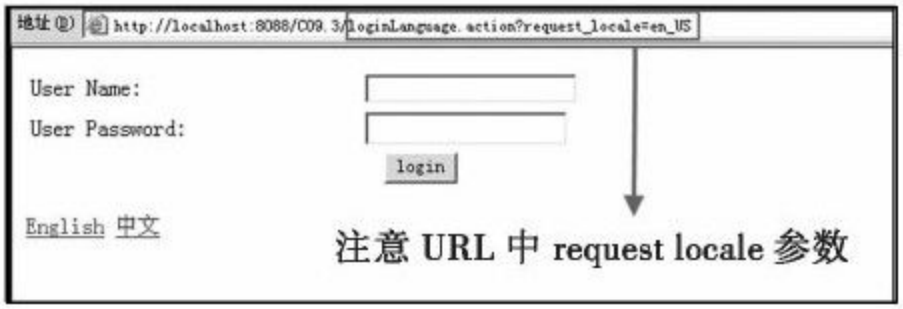

图　9.12　URL中指定了参数request__locale的值

源程序解读

（1）在struts.xml文件中定义了一个不经过Action的类映射的Action。它相当于一个页面链接。这里使用的全局国际化属性文件定义，目的其实就是在链接中给系统传入不同的参数request__locale值。

（2）在login.jsp中，使用了Struts 2标签＜url＞，该标签其实就是定义一个链接中需要指向的值。这里设置id属性是方便使用OGNL表达式来标明在Struts 2标签＜a＞中使用哪一个URL。而login.jsp中定义了两个URL，它们只是参数request__locale的值不一样。一个是zh__CN，另外一个是en__US。使用的

Action都是之前在struts.xml中定义的“loginLanguage”。

接着定义两个Struts 2标签＜a＞，它们的href属性都是OGNL表达式（关于OGNL表达式前面章节有介绍）指向定义好的两个URL。

注意：这里两个URL的id属性名最好不同，若是重名，会让链接不能指向自己需要的语言选项的登录界面。笔者使用了“chinese”和“english”分别说明这两个URL一个是传递值为zh__CN的request__locale参数值，一个是传递值为en__US的request__locale参数值。

（3）在登录界面单击“English”后指向的login.jsp中显示的都是英文。仔细看红框选中的URL，就会发现已经实现了单击“English”链接后，向“loginLanguage”这个Action传递了值为“en__

US”的request_locale参数值，之后输入用户名和密码后，按“login”按钮后显示的登录成功界面的信息也是英文的。

（4）在很多Web网站上都在首页某一部分有可以让用户选择网站的语言选项。如果点击了某一语言后，整个网站所有的数据信息都显示该语言文字（比如51job中简历的“中文”和“英文”语言版本的选择）。其原理就是把Session中的request_locale参数值设置为该语言选项。这样用户也没必要在操作系统中选择语言或者使用其他办法选择语言选项。

在Struts 2中国际化的应用其实也可以实现这一功能，具体实现就如本节所记述。请读者好好参透本小节，毕竟国际化的应用也是实际开发中很重要的一部分。

第10章　Struts 2页面布局实现

传统的Web项目中，经常使用Iframe来进行Web视图页面的布局。在Struts中也提供了一个名为tiles的插件来实现页面布局。而在Struts 2中则提供了一个名为sitemesh的开源产品整合在Struts 2中进行页面布局。本章就详细介绍sitemesh在Struts 2中的整合使用方法。

10.1　sitemesh基本使用方法

笔者在征得客户同意情况下，使用一个真正在使用项目作为本章的示例。我们修改了其中部分代码，用以实现sitemesh的使用方式。

技术要点

以一个装修网站首页为示例，介绍sitemesh的使用方式。

装饰模式的Web布局实现。

web.xml文件中调用sitemesh。

struts.xml中有关result类型和sitemesh关系。

实现代码

首先看web.xml文件涉及sitemesh的部分代码：

```
    <! ……文件名：
web.xml……>
    ……
    <filter>
    <! 定义sitemesh过滤器>
    <filtername>sitemesh</filtername>
    <! 定义sitemesh过滤器类文件路径>
    <filterclass>
    com.opensymphony.module.sitemesh.filter.PageFil
ter
    </filterclass>
    </filter>
    <filtermapping>
    <! 定义sitemesh过滤器映射名>
```

```
    <filtername>sitemesh</filtername>
    <! 定义sitemesh过滤器对根目录下所有文件进行过滤>
    <urlpattern>/</urlpattern>
    </filtermapping>
    <welcomefilelist>
    <welcomefile>jsp/body/firstPage.jsp
</welcomefile>
    </welcomefilelist>
    ……
```

Sitemesh定义装饰器的decorators.xml文件：

```
    <! ……文件名：
decorators.xml……>
    <? xml version="1.0"encoding="gb2312"? >
    <decorators defaultdir="/jsp/template">
    <! excludes标签中定义不会被装饰的JSP视图界面>
    <excludes>
    <pattern>/jsp/layout/</pattern>
    <pattern>/jsp/template/</pattern>
    <pattern>/jsp/image.jsp</pattern>
    </excludes>
    <! 定义装饰器，该装饰器为一个页面格式分为上中下三块的
JSP视图界面>
    <decorator
name="template"page="3PartLayoutTemplate.jsp">
    <pattern>/jsp/</pattern>
    <pattern>/jsp/body/</pattern>
    </decorator>
    </decorators>
```

被定义的装饰器JSP视图界面代码：

```
    <！……文件名：
3PartLayoutTemplate.jsp……>
    <%@page contentType="text/html;
charset=gb2312"%>
    <%@taglib
uri="http: //www.opensymphony.com/sitemesh/decorato
r"prefix="decorator"%>
    <html>
    ……
    <head>
    <！被装饰页面的<title>中内容在这里填充>
    <title>
    <decorator: title default="sitemesh页面"/>
    </title>
    ……
    </head>
    <body
bgcolor="#f7c800"leftmargin="0"topmargin="0"margin
width="0"marginheight="0">
    <jsp: include
page="/jsp/layout/head.jsp"flush="true"/>
    <！被装饰页面的<body>中内容在这里填充>
    <decorator: body/>
    <jsp: include
page="/jsp/layout/bottom.jsp"flush="true"/>
    </body>
    </html>
```

定义Action的struts.xml文件代码：

```
    <！……文件名：
struts.xml……>
    ……
```

```
    <action name="index">
    <result
type="redirect">/jsp/body/firstPage.jsp</result>
    </action>
    ……
```

显示效果如图10.1所示。

图　10.1　sitemesh使用后页面布局显示效果图

源程序解读

（1）首先说明开发中如果要使用sitemesh来进行页面布局功能开发，在WEBINF/lib文件夹下一定要放入sitemesh支持的jar包。目前sitemesh最新版本是sitemesh2.3.jar。

（2）然后在web.xml中必须定义sitemesh为过滤器。这样Web项目中的JSP或其他视图界面都可以被sitemesh调用进行页面布局。然后新建decorators.xml文件。该文件是用来定义所有装饰器视图界面。因为sitemesh是使用设计模式中的装饰模式来进行页面布局功能开发（具体装饰模式定义读者可查看相关设计模式资料）。在这里笔者只能简单说明sitemesh将视图页面分为装饰页面和被装饰页面两种。

装饰页面又称之为装饰器。都是在decorators.xml文件定义，一般定义装饰器时都指定

了被装饰页面或者某文件路径，在该文件路径下的所有视图界面文件都是被装饰页面。

（3）读者可以查看decorators.xml文件代码看到＜excludes＞标签，在该标签中一般都是指定不需要被装饰的视图文件。以＜pattern＞标签来一个个定义。有代码也可知道“/jsp/layout/”标明在/jsp/layout/目录下所有的视图界面都是不需要被装饰的。如果想指定某个视图文件不被装饰，则可以写成代码中“/jsp/image.jsp”的形式，这样在jsp目录下只有image.jsp文件不被装饰，其他视图界面都是要被装饰的。

（4）＜decorator＞标签则是相关装饰器页面定义需要使用的标签。其中name属性是让开发者定义装饰器名字，而page属性则指定装饰器文件的路径和具

体名字。在本示例代码中可知3PartLayoutTemplate.jsp是一个装饰器文件。

（5）在＜decorators＞标签中有个defaultdir属性，它是表明装饰器文件的路径。本示例代码中定义的是“/jsp/template”，也就是说之前所定义的3PartLayoutTemplate.jsp是在该目录下。如果defaultdir定义的是“/”，则装饰器定义的page属性中定义的装饰器文件是在Web项目根目录下。在开发环境中就是在WebRoot文件夹下（笔者使用的是MyEclipse，所以读者使用的开发工具不是MyEclipse，则web根目录的文件夹名字不一定是WebRoot）。如果读者想让3PartLayoutTemplate.jsp还是在/jsp/template目录下，而defaultdir想定义为“/”，则page属性中的定义需要写成“/jsp/template/3PartLayoutTemplate.jsp”形式。

（6）在装饰器定义中，可以仍使用＜pattern＞标签来指定需要被装饰的视图界面。具体做法和＜excludes＞中的＜pattern＞用法相同，只不过一个是指定被装饰的，另外一个指定不被装饰的。当然也可以不使用＜pattern＞标签，这样可以使用＜page：applyDecorator＞标签在页面中指定需要被装饰的视图界面文件（下一小节将具体介绍如何使用），而不是在decorators.xml文件中指定被装饰的视图界面文件。

（7）在3PartLayoutTemplate.jsp该装饰器页面中，读者查看代码可以看见在该视图文件中，笔者先定义了“＜%@taglib uri="http：//www.opensymphony.com/sitemesh/decorator"prefix="decorator"%＞”用以在页面中使用＜decorator＞标签。这里只是使用了＜decorator：

title＞和＜decorator：body＞两个decorator的基础标签。

这里＜decorator：title＞中有个default属性，使用该属性的目的是当被装饰视图界面文件中没有定义自己的＜title＞标签内容时，则装饰和被装饰视图界面一起显示的页面布局界面的title就显示该属性中的内容。在图10.1中，笔者也用红框特别注明了显示的title的内容。

而＜decorator：body＞就是表明被装饰视图界面的＜body＞中内容显示。因此可以这么理解在装饰器视图界面文件中，如果看到以decorator开头的标签，则是显示被装饰视图界面文件的内容。如果“：”后写的是title则显示的是被装饰文件中＜title＞标签中的内容。如果是body则显示的是被装饰文件中＜body＞标签中的内容。

（8）在本示例中笔者将/jsp/body/中的firstPage.jsp作为被装饰文件，在web.xml中定义缺省显示的就是该被装饰文件。因此运行该示例就可以看到firstPage.jsp不仅仅显示了自己的内容，而且将装饰器文件3PartLayoutTemplate.jsp中的内容也显示出来了，特别是两个＜jsp：include＞的JSP文件。给人感觉就是在3PartLayoutTemplate.jsp显示的内容上面又加了firstPage.jsp内容，而被装饰的文件还可以是其他视图文件。

因此在别的界面中则显示的不一定是firstPage.jsp内容，但是3PartLayoutTemplate.jsp中的内容还是会显示出来的，这样就达到了页面布局的效果。

（9）开发人员还可以定义Action来显示页面布局。在笔者使用的该示例struts.xml中定义了

index.action，而其中result指向的就是firstPage.jsp，不过这时result的type类型一定要定义为“redirect”（代码中黑体注明，另外说一句result缺省的类型是dispatcher，可以不显示声明）。

因为如果不定义该类型，则index.action指向的只是单纯firstPage.jsp文件的内容，不会显示装饰器文件3PartLayoutTemplate.jsp中的内容。这和redirect是一个重定向有关系，有关重定向的概念笔者在第2章也做过简单介绍，读者可以翻阅之前章节。

10.2 sitemesh高级应用

在前面小节中笔者简单介绍了一些sitemesh中特有的装饰标签。本节继续介绍开发中比较实用的几个sitemesh装饰标签，其目的是让读者能针对Struts 2页面布局有更深刻的认识。

10.2.1 ＜page：applyDecorator＞和＜decorator：getProperty＞标签

技术要点

运用＜page：applyDecorator＞和＜decorator：getProperty＞两个标签的目的如下：前者是允许开发者可以自行选择使用哪一个装饰器视图界面，后者是

让指定的装饰器页面中定义的参数在被装饰的视图界面中显示参数值内容。

＜page＞标签的介绍和相关注意点。

读取装饰器页面的参数值内容，并显示在被装饰页面的使用方式介绍。

实现代码

定义可以由开发者自行指定的装饰器视图界面的代码：

```
    <！……文件名：decorators.xml……>
    ……
    <！定义装饰器，该装饰器没有被指定可装饰的视图界面，而是在视图界面中来制定需要它装饰的视图界面>
    <decorator name="panel"page="panel.jsp">
</decorator>
    ……
```

---

decorators. xml文件定义的panel.jsp装饰器视图文件代码：

```
    <! ……文件名：panel.jsp……>
    ……
    <head>
    <title>
    <! 被装饰页面的<title>中内容在这里填充>
    <decorator: title default="panel页面"/>
    </title>
    </head>
    <body bgcolor="#f7c800"leftmargin="0"topmargin="0"marginwidth="0"marginheight="0">
    <! 被装饰页面的<body>中内容在这里填充>
    <decorator: body/><br>
    <! 得到已定义的参数内容并使用下列标签在这里显示>
    <decorator: getProperty property="email">
</decorator: getProperty>
    </body>
    </html>
```

在原有装饰器视图文件中定义可以指定其他装饰器并带有具体参数值内容定义的代码：

```
    <! ……文件名：3PartLayoutTemplate.jsp……>
    ……>
```

```
    <td
width="217"height="48"valign="top"align="center">
    <! 在JSP视图界面指定由panel装饰器装饰的页面>
    <page: applyDecorator
page="/jsp/body/page.jsp"name="panel">
    <! 设置参数值内容>
    <page: param
    name="email">frank__wjs@hotmail.com</page:
param>
    </page: applyDecorator>
    </td>
    <td>
    <decorator: body></decorator: body>
    </td>
    ……
```

效果如图10.2所示。

图　10.2　sitemesh自行指定装饰器的效果图

源程序解读

（1）如果要使用＜page＞标签则需要在装饰器页面声明“＜%@taglib uri="http：//www.opensymphony.com/sitemesh/page "prefix="page"%＞”。

注意：sitemesh采取松耦合理念，因此被装饰的视图界面往往都是JSP、HTML等简单的视图文件。不需要使用＜page＞、＜decorator＞标签。这些sitemesh的装饰标签往往都是在装饰器视图文件中定义。

（2）在本示例中还是使用类似于上一小节的示例代码。这里只是在decorators.xml中又定义了一个没有特殊指定被装饰页面的装饰器panel。而在原来装饰器视图文件3PartLayoutTemplate.jsp代码中指定了需要被panel装饰器装饰的视图界面文件page.jsp。由代码可知已调用了＜page：applyDecorator＞标签。page属性指定了被装饰的page.jsp视图文件，而且其

中又嵌套了＜page：param＞标签，定义了一个email参数，该参数值是MSN邮箱。name属性就是参数名内容，这里是“email”。

这样其实原来装饰器视图界面3PartLayoutTemplate.jsp中包含了另外一个装饰器视图界面文件panel.jsp。因此实现多个装饰器文件同时装饰文件的功能。如果开发中根据特定需求，需要在页面布局中某一部分动态装饰视图文件，其他部分不需要装饰视图文件时，这样的功能恰好能满足特定需求。

因为只需要使用＜page：applyDecorator＞标签来指定需要装饰的那一部分调用的装饰器文件，就能达到局部装饰的目的。而且如果开发者还想实现一个视图文件由唯一一个装饰器文件装饰，又可以将这段

使用＜page：applyDecorator＞标签的代码删去或注释掉即可，实现了之前曾经说过的松耦合理念。

（3）在装饰器文件panel.jsp代码中也显示，使用＜decorator：getProperty＞标签可以在被装饰的视图界面文件中显示指定的装饰器文件中使用的参数值。property属性中的内容就是在＜page：param＞标签中定义的参数name属性。在图10.2中左边红框中也显示了被panel装饰器装饰的page.jsp显示了MSN邮箱，达到了局部装饰的目的。

10.2.2 ＜decorator：usePage＞、＜decorator：useHtmlPage＞和＜decorator：head＞标签

技术要点

＜decorator：head＞标签的作用和普通html页面中＜head＞标签相同，是在页面布局中显示被装饰视图界面文件的头部分属性定义。而＜decorator：usePage＞和＜decorator：useHtmlPage＞标签的作用相同都是显示被装饰页面中使用的标签属性。

这两个标签存在的原因是因为siteMesh的＜decorator：body＞、＜decorator：head＞、＜decorator：title＞标签只能解析普通视图界面中＜body＞、＜head＞、＜title＞等几个标签中的内容，

不能直接读取这些标签的属性内容，所以用这两个标签来扩展该功能。

＜decorator：head＞标签的介绍和相关注意点。

使用＜decorator：usePage＞和＜decorator：useHtmlPage＞标签读取被装饰页面中标签属性的使用方法。

实现代码

被装饰页面firstPage.jsp代码：

```
    <！……文件名：
firstPage.jsp……>
    <%@page contentType="text/html;
charset=gb2312"%>
    <%@taglib
uri="http: //www.opensymphony.com/sitemesh/decorato
r"prefix="decorator"%>
    <html>
    <！定义被装饰的JSP页面的meta、body等属性的内容>
    <head>
    <title>使用标签decorator: usePage</title>
    <meta name="description"content="记述">
    <meta name="author"content="作者">
```

```
    </head>
    <body onload="调用某JavaScript方法">
    </body>
    </html>
```

显示标签属性内容和调用<decorator：head>标签装饰器视图文件代码：

```
    <！……文件名：3PartLayoutTemplate.jsp……>
    <%@page contentType="text/html;charset=gb2312"%>
    <%@taglib uri="http: //www.opensymphony.com/sitemesh/decorator"prefix="decorator"%>
    <%@taglib uri="http: //www.opensymphony.com/sitemesh/page"prefix="page"%>
    <html>
    ……
    <！使用decorator: head显示被装饰页面的head标签中内容（除title以外）>
    <decorator: head/>
    ……
    <！使用decorator: usePage标签，打印出被装饰页面的body等标签的属性内容>
    <！usePage相当于JSP中<useBean>标签定义的JavaBean类名>
    <decorator: usePage id="usePage"/>
    <tr>
    <td align="center"colspan=2>
    <h2>使用decorator: usePage标签各属性内容：</h2>
    </td>
```

```
    </tr>
    <%
    //取得被装饰页面的标签属性名，作为key值组合成一字符串数
组
    String[]element=usePage.getPropertyKeys（）；
    for（int i=0; i<element.length; i++）｛%>
    <tr>
    <td align="right">
    <%=i+1%>
    </td>
    <td>
    <！各元素为被装饰页面标签属性名，作为key值，由
getProperty方法得到value即属性内容>
    <%=element[i]%>:
<%=usePage.getProperty（element[i]）%>
    </td>
    </tr>
    <%｝%>
    <！使用decorator: useHtmlPage标签，打印出被装饰页面
的body等标签的属性内容>
    <！HTMLPage相当于JSP中<useBean>标签定义的
JavaBean类名>
    <decorator: useHtmlPage id="HTMLPage"/>
    <tr>
    <td align="center"colspan=2>
    <h2>使用decorator: useHtmlPage标签各属性内容:
</h2>
    </td>
    </tr>
    <%
    //取得被装饰页面的标签属性名，作为key值组合成一字符串数
组
    String[]elementOfHTMLPage=HTMLPage.getPropertyK
eys（）；
    for（int i=0; i<elementOfHTMLPage.length; i++）
 ｛%>
    <tr>
```

```
    <td align="right">
    <%=i+1%>
    </td>
    <td>
    <！各元素为被装饰页面标签属性名，作为key值，由
getProperty方法得到value即属性内容>
    <%=elementOfHTMLPage[i]%>:
<%=HTMLPage.getProperty（elementOfHTMLPage
    [i]）%>
    </td>
    </tr>
    <%} %>
    ……
```

效果如图10.3所示。

图　10.3　sitemesh使用＜decorator：usePage＞和＜decorator：useHtmlPage＞标签的效果图

在图10.3显示的页面中单击鼠标右键，在弹出菜单中选择“查看源文件”命令，可查看生成的＜head

＞标签内容，如图10.4所示。

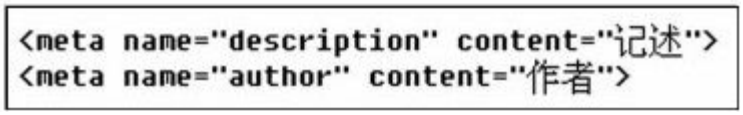

图　10.4　sitemesh使用＜decorator：head＞标签的效果图

源程序解读

（1）该示例中将被装饰文件firstPage.jsp内容重新做了调整。在＜head＞和＜body＞标签中增加了一些属性。之前也已经说过原来的sitemesh装饰标签值能显示＜head＞和＜body＞

之间的内容，不能显示它们的属性。但是使用＜decorator：usePage＞和＜decorator：useHtmlPage＞就能显示这些属性的内容。

（2）请仔细看装饰器文件3PartLayoutTemplate.jsp中的代码。这里使用了＜

decorator：usePage＞和＜decorator：useHtmlPage＞标签，其中的id属性定义了使用这两个标签的id名，而这些id名竟然可以在Java代码中作为一个对象来使用。如果有JSP基础的读者就能发现这两个标签其实和JSP中的＜jsp：useBean＞使用方式如出一辙。说简单点就是利用这两个标签各自定义了一个Java对象，而且其中的getPropertyKeys和getProperty方法都是可以得到属性和属性代表的值内容。

如图10.3所示，通过循环遍历将被装饰视图文件firstPage.jsp中＜head＞和＜body＞属性名和代表的值内容都显示出来，这也更加证明之前所说的利用＜decorator：usePage＞和＜decorator：useHtmlPage＞标签可以直接读取被装饰视图文件中＜head＞和＜body＞属性值。

（3）在3PartLayoutTemplate.jsp中的代码还使用了＜decorator：head＞标签。在图10.4说明中也介绍了通过查看源文件方式可以知道它显示了被装饰视图文件firstPage.jsp中＜head＞中的内容，唯一没有显示的是＜title＞内容。而title内容正是由前一小节中介绍的＜decorator：title＞标签来显示。

（4）sitemesh的装饰标签还有其他一些，但不是很实用，而且使用的目的有点多此一举，因此这里就不介绍了，有兴趣的读者可以查阅其他sitemesh资料。

第11章　Hibernate技术简介

Hibernate是一个免费的开源Java项目，它使程序员使用关系数据库变得十分容易，使数据库表的访问同普通Java对象访问一样简单和快捷。更加方便的是程序员不必考虑如何将这些Java对象和数据库表中的数据保持同步，这些工作由Hibernate来完成。这样解放了程序员，使其可以专注于应用程序的对象和功能，而不必担心如何保存它们或者如何存取数据库。Hibernate目前已经成为流行的Java开源项目。

11.1　什么是ORM

对象-关系映射（Object/Relational Mapping，简称ORM）是一种为了解决面向对象与关系数据库存在的互不匹配的问题的技术。简单的说，ORM是连接对象

和数据库之间的桥梁，通过使用描述对象和数据库之间映射的元数据，将Java程序中的对象自动持久化到关系数据库中。

ORM本质上就是将数据从一种形式转换到另外一种形式。虽然增加了一些额外的执行程序，但是ORM作为一种中间件实现，则提供很多机会做优化处理。

11.1.1 ORM基础

对象-关系映射是随着面向对象的软件开发方法发展而产生的。面向对象的开发方法是当今企业级应用开发环境中的主流开发方法，关系数据库是企业级应用环境中永久存放数据的主流数据存储系统。对象和关系数据是业务实体的两种表现形式，业务实体在内存中表现为对象，在数据库中表现为关系数据。

内存中的对象之间存在关联和继承关系，而在数据库中，关系数据无法直接表达多对多关联和继承关系。因此，对象-关系映射系统一般以中间件的形式存在，主要实现程序对象到关系数据库数据的映射。

面向对象是从软件工程基本原则（如耦合、聚合、封装）的基础上发展起来的，而关系数据库则是从数学理论发展而来的，两套理论存在显著的区别。为了解决这个不匹配的现象，对象关系映射技术应运而生。

字母O起源于“对象”（Object），而R则来自于“关系”（Relational）。几乎所有程序中，都存在对象和关系数据库。在业务逻辑层和用户界面层中，是面向对象的。当对象信息发生变化时，我们需要把对象的信息保存在关系数据库中。

当开发一个应用程序的时（不使用ORM），程序员会编写很多数据访问层的代码，用来从数据库保存、删除、读取对象信息等，而这些代码写起来总是重复的。

更好地访问数据库的方式是ORM。引入一个ORM，实质上，一个ORM会为你生成数据库访问语句。程序员使用ORM保存、删除、读取对象，ORM负责生成SQL，这样，程序员只需要关心对象就可以了，极大地简化了数据库访问的复杂度。

对象关系映射成功运用在不同的面向对象持久层产品中，如Torque、OJB、Hibernate、TopLink、Castor JDO和TJDO等。

11.1.2 ORM组成

一般的ORM包括以下4部分：

一个对持久类对象进行CRUD操作的API。

一个语言或API用来规定与类和类属性相关的查询。

一个规定mapping metadata的工具。

一种技术可以让ORM的实现同事务对象一起进行dirty checking、lazy association fetching以及其他优化操作。

如图11.1所示是ORM示意图。

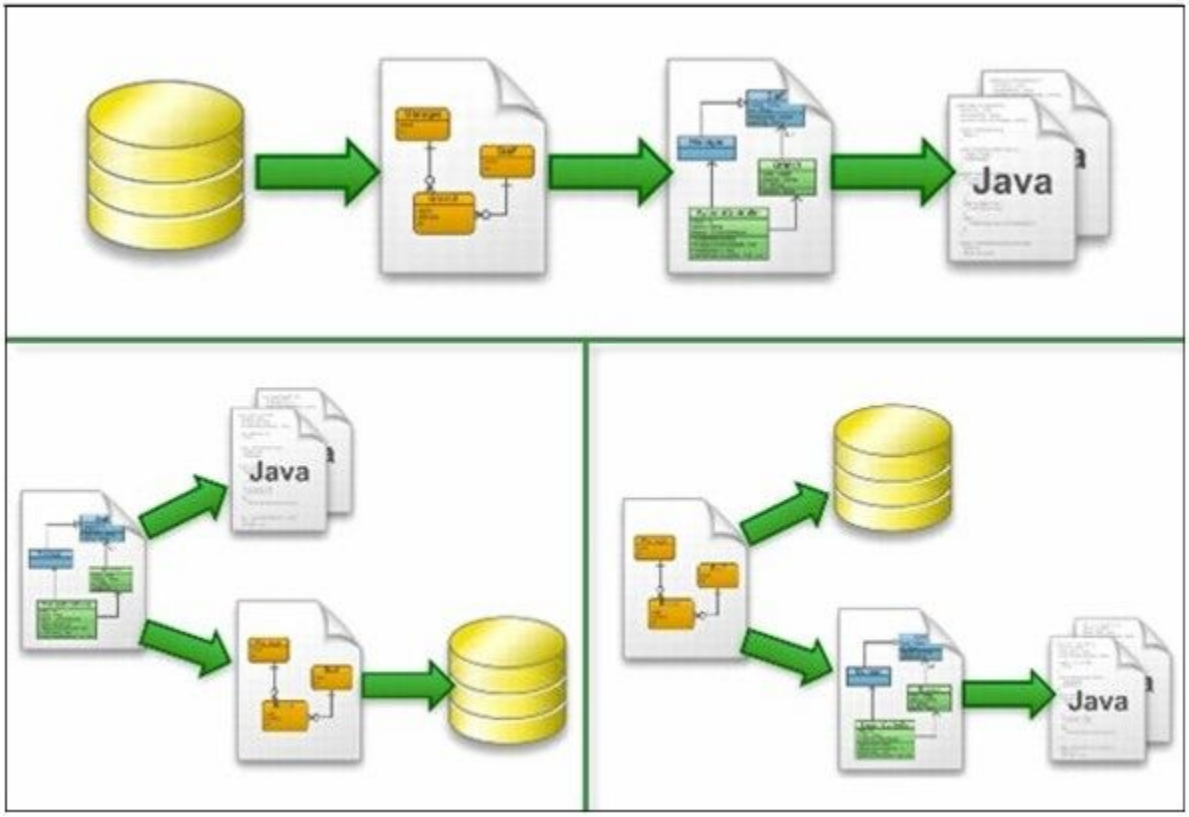

图　11.1　ORM示意图

11.1.3　流行的ORM架构

目前众多厂商和开源社区都提供了持久层框架的实现，常见的有：

Apache OJB。

Cayenne。

Jaxor。

Hibernate。

iBatis。

jRelationalFramework。

mirage。

SMYLE。

TopLink。

其中TopLink是Oracle的商业产品，其他均为开源项目。

其中Hibernate的轻量级ORM模型逐步确立了在Java ORM架构中的领导地位，甚至取代了复杂而又烦琐的EJB模型而成为事实上的Java ORM工业标准。而且其中的许多设计均被J2EE标准组织吸纳而成为最新EJB 3.0规范的标准，这也是开源项目影响工业领域标准的有力见证。

11.2 Hibernate概述

Hibernate是一种Java语言下的对象关系映射解决方案。它是一种自由、开源的软件。它用来把对象模型表示的对象映射到基于SQL的关系模型结构中去，为面向对象的领域模型到传统的关系型数据库的映射，提供了一个使用方便的框架。

目前Hibernate已经发展到Hibernate 3.X版本，读者注意它与Hibernate 2.0的区别。

11.2.1 Hibernate用途

Hibernate不仅管理Java类到数据库表的映射（包括从Java数据类型到SQL数据类型的映射），还提供数据查询和获取数据的方法，可以大幅度减少开发时人工使用SQL和JDBC处理数据的时间。

它的设计目标是将软件开发人员从大量相同的数据持久层相关编程工作中解放出来。无论是从设计草案还是从一个遗留数据库开始，开发人员都可以采用Hibernate。

Hibernate对JDBC进行了非常轻量级的对象封装，使得Java程序员可以随心所欲地使用对象编程思维来操纵数据库。Hibernate可以应用在任何使用JDBC的场合，它既可以在Java的客户端程序使用，也可以在Servlet/JSP的Web应用中使用。最具革命意义的是，Hibernate可以在应用EJB（即Enterprise JavaBeans是Java应用于企业计算的框架）的J2EE架构中取代CMP，完成数据持久化的重任。

11.2.2 Hibernate架构

Hibernate不会对你造成妨碍，也不会强迫你修改对象的行为方式。它们不需要实现任何不可思议的接口便能够持续存在，唯一需要做的就是创建一份XML“映射文档”，告诉Hibernate你希望能够保存在数据库中的类，以及它们如何关联到该数据库中的表和列，然后就可以要求它以对象的形式获取数据，或者把对象保存为数据。与其他解决方案相比，它几乎已经很完美了。如图11.2所示是Hibernate体系结构图。

从图11.2中可以看出，Hibernate使用数据库和配置信息来为应用程序提供持久化服务（以及持久的对象）。

下面详细地看一下Hibernate运行时的体系结构。由于Hibernate非常灵活，且支持多种应用方案，所以在这里介绍一下两种极端的情况。

简单的体系结构方案，要求应用程序提供自己的JDBC连接并管理自己的事务。这种方案使用了Hibernate API的最小子集，如图11.3所示。

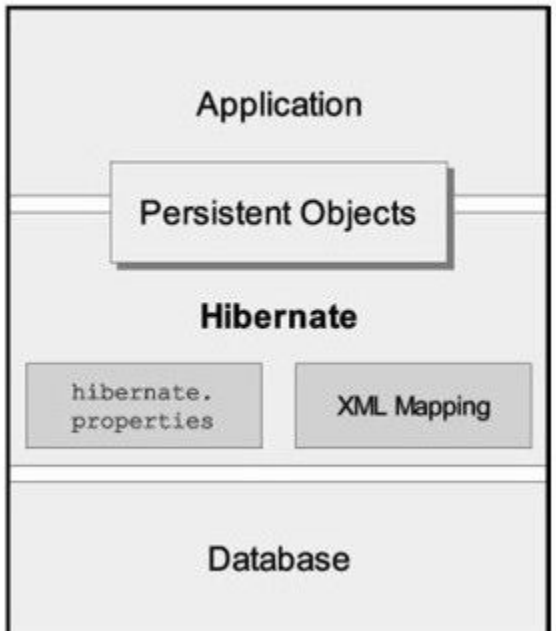

图　11.2　Hibernate体系结构图

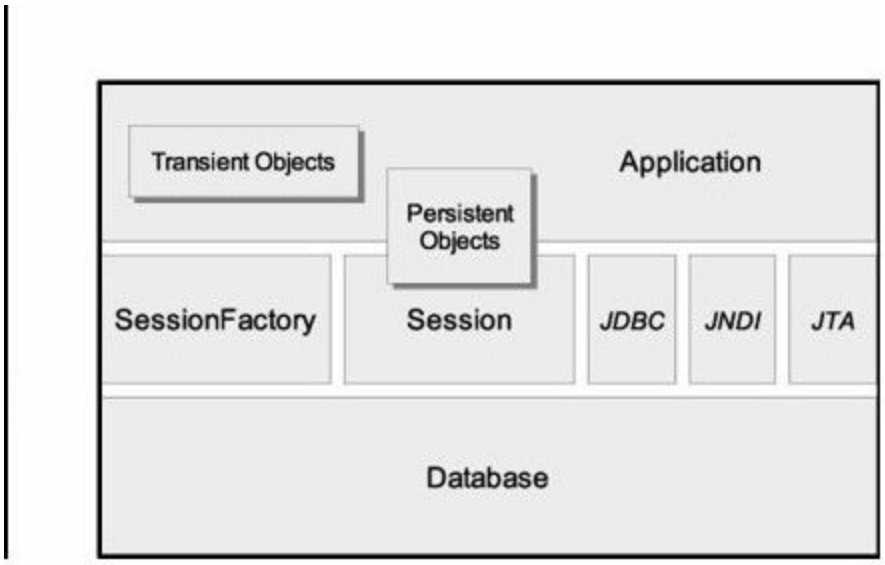

图　11.3　简单的体系结构图

复杂的体系结构方案，将应用层从底层的JDBC/JTA API中抽象出来，而让Hibernate来处理这些细节，如图11.4所示。

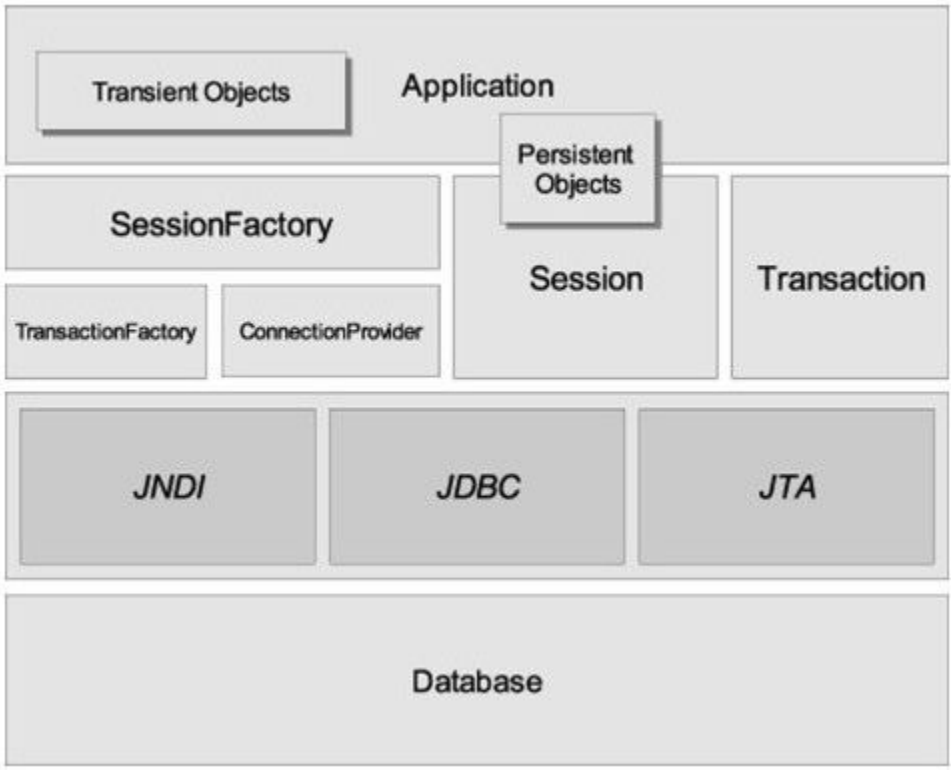

图　11.4　复杂的Hibernate体系结构图

图11.4中涉及下面的几个内容：

SessionFactory：它针对单个数据库映射关系经过编译后的内存镜像，是线程安全的（不可变）。它是生成Session的工厂，本身要用到ConnectionProvider。该对象可以在进程或集群的级别上，为那些事务之间可以重用的数据提供可选的二级缓存。

Session：表示应用程序与持久存储层之间交互操作的一个单线程对象，此对象生存期很短。其隐藏了JDBC连接，也是Transaction的工厂。它会持有一个针对持久化对象的必选（第一级）缓存，在遍历对象图或者根据持久化标识查找对象时会用到。

持久的对象及其集合：带有持久化状态的、具有业务功能的单线程对象，此对象生存期很短。这些对象可能是普通的JavaBeans/POJO，唯一特殊的是它们正与（仅仅一个）Session相关联。一旦这个Session被关闭，这些对象就会脱离持久化状态，这样就可被应用程序的任何层自由使用（例如，用作跟表示层打交道的数据传输对象）。

瞬态（Transient）和脱管（Detached）的对象及其集合：目前没有与Session关联的持久化类实例。它们可能是在被应用程序实例化后，尚未进行持久化的

对象，也可能是因为实例化它们的Session已经被关闭而脱离持久化的对象。

事务（Transaction）：应用程序用来指定原子操作单元范围的对象，它是单线程的，生命周期很短。它通过抽象将应用从底层具体的JDBC、JTA以及CORBA事务隔离开。某些情况下，一个Session之内可能包含多个Transaction对象。尽管是否使用该对象是可选的，但无论是使用底层的API还是使用Transaction对象，事务边界的开启与关闭是必不可少的。

ConnectionProvider：生成JDBC连接的工厂（同时也起到连接池的作用）。它通过抽象将应用从底层的Datasource或DriverManager隔离开，仅供开发者扩展/实现用，并不暴露给应用程序使用。

TransactionFactory：生成Transaction对象实例的工厂。仅供开发者扩展/实现用，并不暴露给应用程

序使用。

11.2.3　Hibernate核心接口

应用Hibernate框架到项目当中，第一步就是要了解Hibernate的核心接口。Hibernate接口位于业务层和持久化层，如图11.5所示为Hibernate核心接口的关系和它们在应用模型中所处的位置。

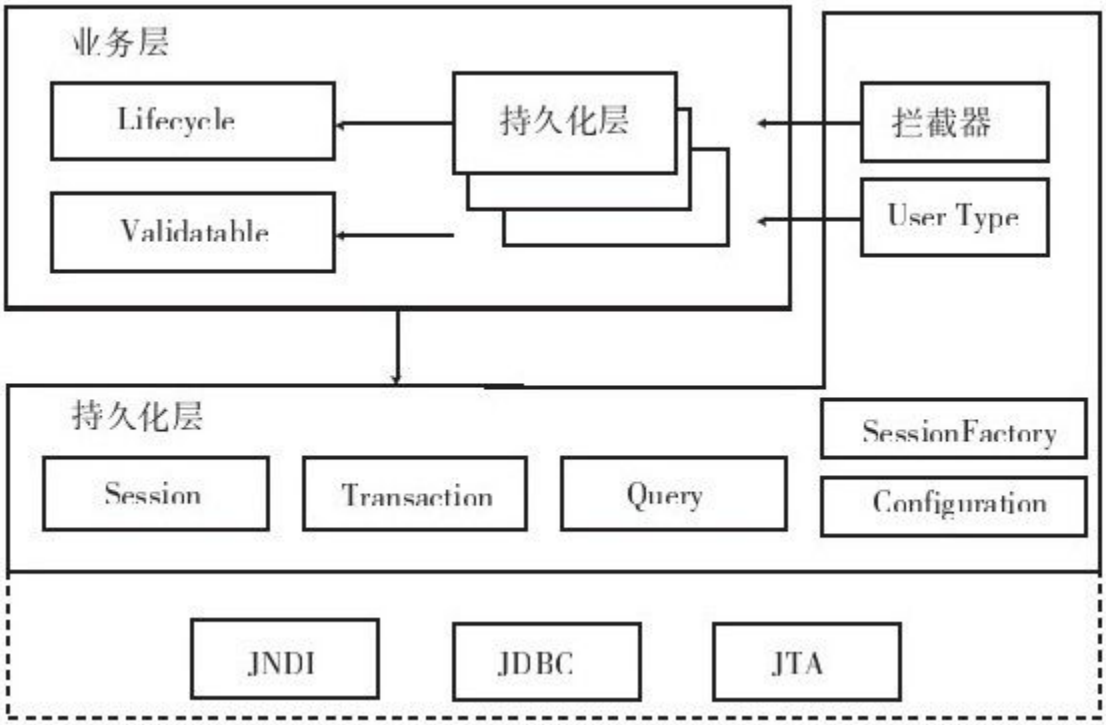

图　11.5　Hibernate核心接口的层次架构关系

Hibernate的核心接口可以分为如下几类：

执行基本的CRUD和查询操作的接口。这些接口是应用程序的业务逻辑对Hibernate框架的主要依赖点，接口包括Session、Transaction和Query。

执行Hibernate配置的接口，包括对Hibernate框架本身的配置和需要被持久化的类的配置信息。

允许应用程序响应Hibernate事件的回调接口。Hibernate事件是指类似持久化对象被加载、插入、更新及删除等事件。回调接口包括截取器、生命周期和有效性验证等。

允许对Hibernate功能进行扩展的接口，例如UserType、CompositeUserType和IdentifierGenerator等接口。

Hibernate的大多数的核心接口位于net.sf.hibernae包中，主要包括以下五个接口。

Session接口：负责执行被持久化对象的CRUD操作。

SessionFactory接口：初始化Hibernate，充当数据存储源的代理，并负责创建Session对象。

Configuration接口：负责配置并启动Hibernate，创建SessionFactory对象。

Transaction接口：负责事务相关的操作。

Query和Criteria接口：执行各种数据库查询。

回调接口允许应用程序在某些对象上发生一些持久化事件时得到通知，例如对象被加载、更新或删除时。Hibernate应用通过回调接口来响应这些事件。回

调接口按实现方式可以分为生命周期和数据验证接口，即Lifecycle和Validatable两类。这两个接口由持久化类实现。Lifecycle接口使得持久化对象能响应被加载、保存或删除时间。Validatable接口使得持久化对象在被保存之前进行数据验证。这种实现需要持久化类实现Hibernate特定的接口，使应用程序会过多依赖Hibernate，破坏了Hibernate非侵入的特点，因此不推荐使用。

拦截器接口，即Intercepor。Intercepor接口并不由持久化类实现，应用程序可以定义专门实现Intercepor接口的类。Intercepor接口负责响应持久化对象的CRUD事件。

Type接口表示Hibernate映射类型，用于把Java对象映射为数据库中的关系数据（可能不止一行）。Hibernate为Type接口提供了许多内置的实现类，代表

具体的Hibernate映射类型，这些类型覆盖了所有的Java原生类型、大多数的JDK类，包括java.util.Currency、java.util.Calendar、byte[]和java.io.Serializable。Hibernate还支持自定义的类型，UserType和CompositeUserType接口提供了将自定义的类型添加到Hibernate映射类型。

Hibernate提供的很多功能是可配置的，允许用户选择适当的Hibernate内置策略。当Hibernate内置的策略不能满足需求时，就可以进行扩展。Hibernate的扩展点包括以下内容：

定制主键的生成策略（IdentifierGenerator接口）。

SQL方言支持（Dialect抽象类）。

缓存策略（Cache和CacheProvider接口）。

JDBC连接管理（ConnectionProvider接口）。

事务管理（TransactionFactory、Transaction和TransactionManagerLookup接口）。

ORM策略接口（ClassPersister及其扩展接口）。

属性访问策略（PropertyAccessor接口）。代理工厂（ProxyFactory接口）。

11.2.4　持久化对象的状态

当应用程序通过new语句创建一个对象，这个对象的生命周眯期就开始了，当不再有任何引用变量引用它时，这个对象就结束了生命周期，它占用的内存就可以被JVM的垃圾回收器回收。对于需要被持久化的Java对象，在它的生命周期中，可处于以下三个状态之一：

临时状态（Transient）。刚刚用new语句创建，还没有被持久化，不处于Session的缓存中。处于临时状态的Java对象被称为临时对象。

持久化状态（Persistent）。已经被持久化，加入到Session的缓存中。处于持久化状态的Java对象被称为持久化对象。

游离状态（Detached）。已经被持久化，但不再处于Session的缓存中。处于游离状态的Java对象被称为游离对象。

11.3 Hibernate优点

Hibernate具有下面的几个优点：

Hibernate是JDBC的轻量级的对象封装。它是一个独立的对象持久层框架，和App Server、EJB没有什么必然的联系。Hibernate可以用在任何JDBC可以使用的场合，例如Java应用程序的数据库访问代码，DAO接口的实现类，甚至可以是BMP里面的访问数据库的代码。从这个意义上来说，Hibernate和EJB不是一个范畴的东西，也不存在非此即彼的关系。

Hibernate是一个和JDBC密切关联的框架，所以Hibernate的兼容性和JDBC驱动，和数据库都有一定的关系，但是和使用它的Java程序，和App Server没有任何关系，也不存在兼容性问题。

Hibernate不能用来直接和Entity Bean做对比。只有放在整个J2EE项目的框架中才能比较。并且即使是放在软件整体框架中来看，Hibernate也是作为JDBC的替代者出现的，而不是Entity Bean的替代者出现的。

Hibernate发展前景看好，新版的Hibernate不但支持Java架构，同时还提供了对微软的.NET的支持。

第12章　Hibernate入门

Hibernate是目前流行的Java框架ORM开源项目，已经被广泛应用在各类软件开发中，同时得到了广大Java程序员的追捧。本章向读者介绍Hibernate的基本知识，读者通过本章学习，可以基本了解Hibernate的运行过程，并通过一个示例，体验Hibernate技术的魅力。当然在使用Hibernate之前，需要首先安装数据库系统。

12.1　准备工作

使用Hibernate，需要下载Hibernate项目必需的开发包，另外需要安装数据库，用来测试Hibernate。下面将一步一步详细引导读者来完成相关操作。

12.1.1　安装Hibernate

Hibernate是一个Java开源项目，读者可以登录官方网站：www.hibernate.org，界面如图12.1所示。进入Hibernate Core栏目，如图12.2所示。

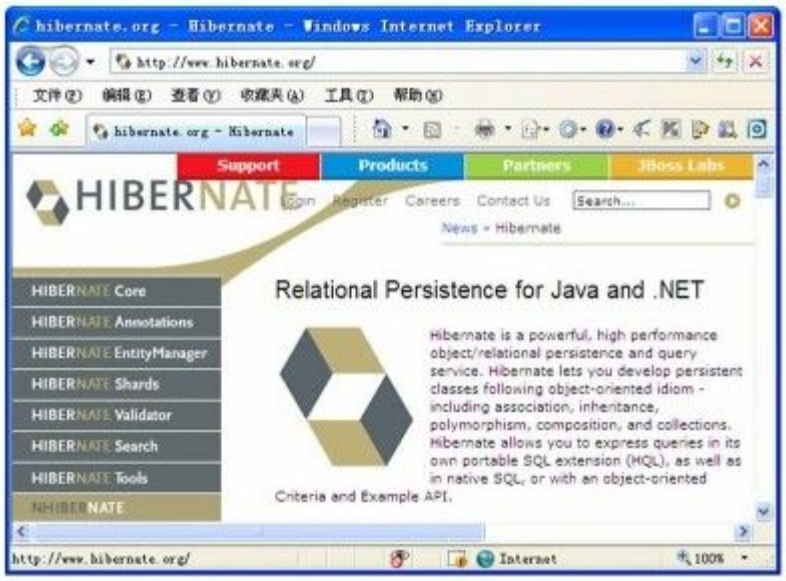

图　12.1　Hibernate官方网站

图　12.2　Hibernate Core下载界面

Hibernate下载界面中，读者可以发现网站提供了Linux版本和Windows版本的Hibernate，要根据开发环境的不同来选择。这里下载了Windows版本的ZIP压缩包，解压缩后目录结构如图12.3所示。

其中，hibernate3.jar是Hibernate的核心类库，lib目录中包含了其他辅助的jar文件，要使用Hibernate，只需要将相关类库保存到系统的环境变量CLASSPATH中即可，必需的jar文件如图12.4所示。

注意：Hibernate并不是只能在Java Web项目中使用，也可以在普通的Java程序中访问，本章示例均使用普通的Java程序访问Hibernate。Hibernate在Web开发中的使用，后面案例中将会详细介绍。

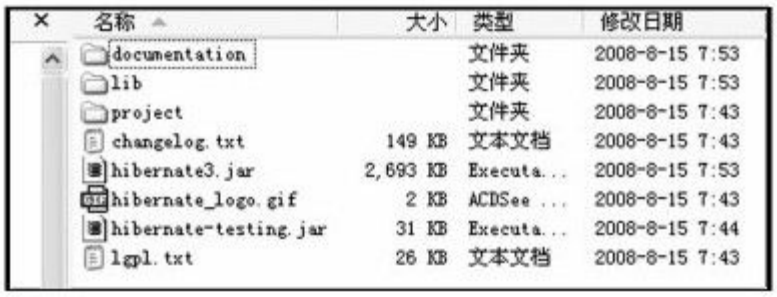

图　12.3　Hibernate压缩包文件结

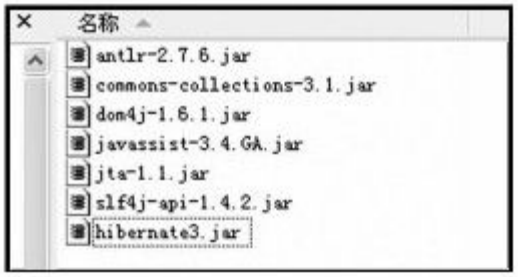

图　12.4　Hibernate必需的类库

12.1.2 MyEclipse中使用Hibernate

MyEclipse是一个Eclipse插件，集成了许多强大的功能，例如缺省的Hibernate支持，读者如果使用MyEclipse作为开发工具，则已经内置了Hibernate支持，无需在额外下载Hibernate类库。

（1）在MyEclipse中使用Hibernate非常简单，在项目工程上单击鼠标右键，在弹出菜单中选择了MyEclipse“Add Hibernate Capabilities”命令，即增加项目的Hibernate支持，如图12.5所示。

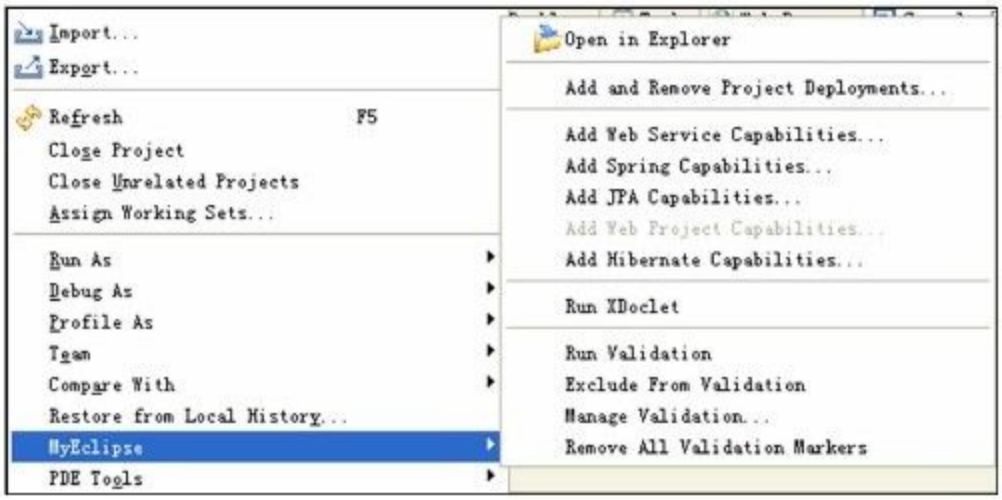

图　12.5　添加Hibernate支持

（2）接下来会让开发者选择相关的配置信息，如图12.6所示。读者在该界面中可以选择Hibernate的版本，笔者编写此书时，MyEclipse最新版本最高支持Hibernate 3.1版本，以后会有更新的版本。

（3）该界面中还可以让读者选择Hibernate使用的jar类库，如果读者下载了自己的jar类库，可以选择“User Libraries”复选框，并指定类库的位置；如果不使用自己的类库，MyEclipse已经提供了Hibernate相关版本的类库。

（4）单击“Next”按钮，将会出现数据库连接配置界面，如图12.7所示。读者可以在这里配置数据库的相关信息，例如，是使用JDBC驱动还是使用JNDI驱动、连接的URL、数据库驱动jar、数据库连接的用户

名和密码等信息。读者还可以选择是否能够动态生成数据库表。

MyEclipse提供了良好的Hibernate代码生成功能，可以根据数据库表自动生成对应的Java对象，并自动完成XML文件的配置，同时提供基本功能的Java对象DAO模板。

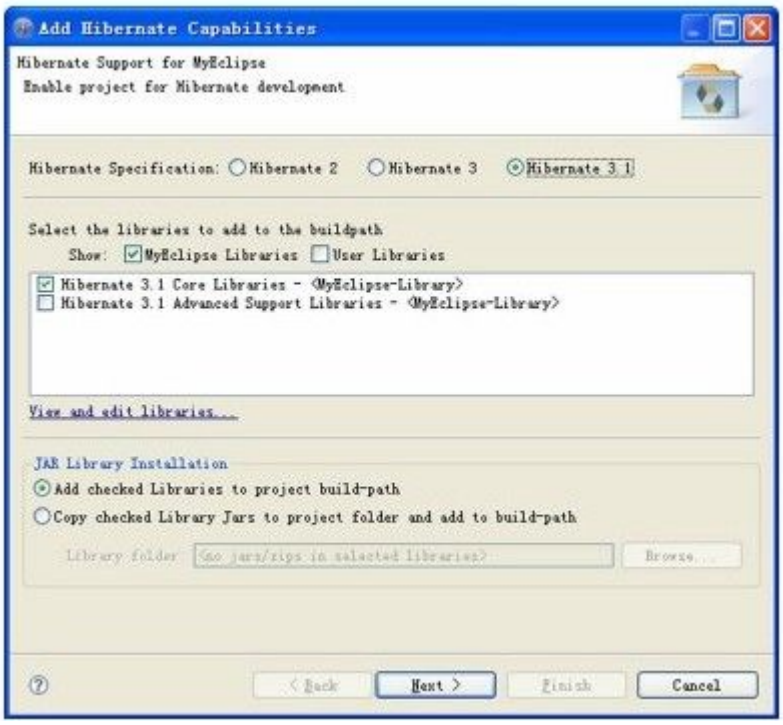

图　12.6　选择Hibernate版本

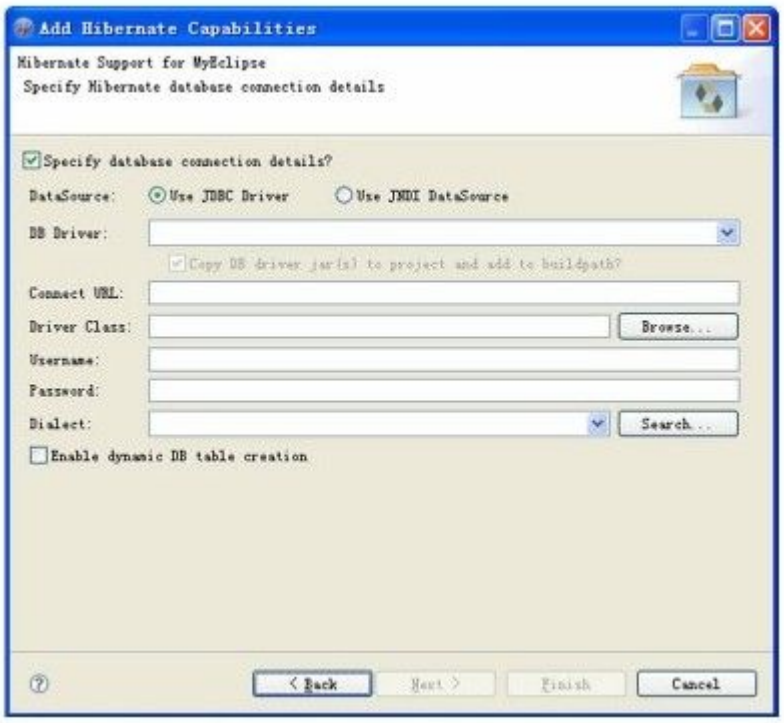

图　12.7　配置数据库连接

12.1.3　安装MySQL数据库

为了使用Hibernate，需要安装数据库，本书使用MySQL数据库作为基本数据库，对于MySQL安装等过程请参考本书中对MySQL技术的专门介绍部分。

本章需要建立一个示例数据库，这里需要建立一个名为pla的数据库，在MySQL界面使用如下命令：

```
create database pla;
```

为了访问MySQL数据库，需要下载MySQL数据库的驱动程序，读者可以访问MySQL数据库官方网站www.mysql.com，下载数据库连接驱动，如图12.8所示。

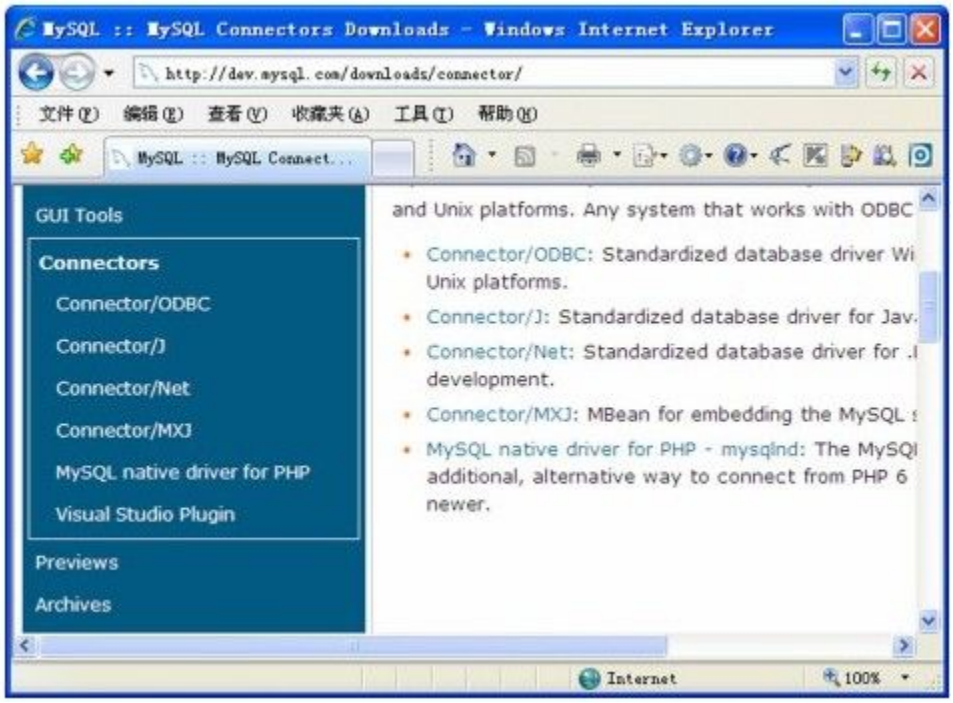

图　12.8　MySQL数据库驱动下载界面

下载相关的jar驱动文件，并复制到项目的lib目录下，或者是系统的CLASSPATH目录下。这样，就可以通过Hibernate来访问MySQL数据库了。

12.2　第一个Hibernate应用

下面以一个简单的Hibernate应用来体验Hibernate的魅力，假设数据库pla中存在一个用户表（user），本应用就是演示如何使用Hibernate来访问数据库。

技术要点

本节以一个简单的Hibernate应用来引导读者创建Hibernate应用，为读者带来Hibernate体验。

基本的Hibernate配置。

使用Java对象实现对数据库表的添加记录功能。

实现代码

首先登录MySQL数据库，建立表user表，SQL语句如下：

```
CREATE TABLE'user'（
'id'int（11）NOT NULL auto__increment，
'name'char（20）default NULL，
'age'int（11）default NULL，
PRIMARY KEY（'id'）
）ENGINE=InnoDB DEFAULT CHARSET=gbk
```

Hibernate配置文件hibernate.cfg.xml：

```
<? xml version='1.0'encoding='UTF8'? >
<! DOCTYPE hibernateconfiguration PUBLIC
"//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http: //hibernate.sourceforge.net/hibernateconfiguration3.0.dtd">
<hibernateconfiguration>
<sessionfactory>
<! 数据库用户名>
<property name="connection.username">root</property>
<! 数据库连接URL>
<property name="connection.url">
jdbc: mysql: //localhost: 3306/ssh
</property>
<property name="dialect">
org.hibernate.dialect.MySQLDialect
</property>
```

```
    <property name="myeclipse.connection.profile">
Mysql</property>
    <property name="hbm2ddl.auto">update
</property>
    <! 数据库密码>
    <property name="connection.password">pla
</property>
    <! 数据库驱动>
    <property name="connection.driver__class">
    com.mysql.jdbc.Driver
    </property>
    <! 数据库表Java对象映射文件表>
    <mapping
resource="base/helloworld/User.hbm.xml"/>
    </sessionfactory>
    </hibernateconfiguration>
    Java对应的类User.java:
    package base.helloworld;
    public class User implements
java.io.Serializable {
    private Integer id; //ID属性变量
    private String name; //用户名name属性变量
    private Integer age; //用户年龄age属性变量
    public User（） {} //构造器
    public Integer getId（） {//id属性的Getter方法和
Setter方法
    return this.id;
    }
    public void setId（Integer id） {
    this.id=id;
    }
    public String getName（） {//name属性的Getter方法
和Setter方法
    return this.name;
    }
    public void setName（String name） {
    this.name=name;
```

```
    }
    public Integer getAge（）｛//age属性的Getter方法和
Setter方法
    return this.age;
    }
    public void setAge（Integer age）｛
    this.age=age;
    }
    }
```

User类的配置文件user.hbm.xml：

```
    ＜? xml version="1.0"encoding="utf8"? ＞
    ＜! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd"＞
    ＜hibernatemapping＞
    ＜! 指定类和对象的表＞
    ＜class
name="base.helloworld.User"table="user"catalog="pl
a"＞
    ＜! 指定主键＞
    ＜id name="id"type="java.lang.Integer"＞
    ＜! 对应的表字段＞
    ＜column name="id"/＞
    ＜! 定义主键生成方式Hibernate自行决定＞
    ＜generator class="native"/＞
    ＜/id＞
    ＜! 定义name字段＞
    ＜property name="name"type="java.lang.String"＞
    ＜column name="name"length="20"/＞
    ＜/property＞
    ＜! 定义age字段＞
```

```
    <property name="age"type="java.lang.Integer">
    <column name="age"/>
    </property>
    </class>
    </hibernatemapping>
```

至此，第一个Hibernate应用就完成了，下面编写一个测试程序，向user表中加入一条记录，TestHelloWorld.java文件：

```
    package base.helloworld;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.Transaction;
    import org.hibernate.cfg.Configuration;
    public class TestHelloWorld {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（）.configure（）;
    //根据config建立SessionFactory
    //SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）;
    User user=new User（）; //建立持久化对象
    user.setName（"测试"）;
    user.setAge（new Integer（30））;
    //建立Session，相当于建立JDBC的Connection
    Session session=sessionFactory.openSession（）;
    //Transaction表示一组数据库事务处理
    Transaction tx=session.beginTransaction（）;
    session.save（user）; //将对象保存到数据库当中
```

```
    tx.commit（）；
    session.close（）；
    sessionFactory.close（）；
    System.out.println（"增加用户数据成功，请查看数据库数据！"）；
    }
    }
```

运行TestHelloWorld，结果如下：

增加用户数据成功，请查看数据库数据！

读者可以登录MySQL数据库，查询user表，可以看到已经增加了一条记录。

源程序解读

（1）hibernate. cfg.xml是Hibernate配置文件。基本的Hibernate配置文件，可以使用XML或Properties属性文件，这里使用XML，文件名缺省预设为hibernate.cfg.xml。

（2）User类文件中除了定义对应的属性，还定义了构造器，其中id属性很重要，是主键属性。

（3）为了定义id主键的生成方式，需要在user.hbm.xml文件中定义generator属性，这里定义为native，表示主键的生成方式由Hibernate根据数据库Dialect的定义来决定。

（4）在测试程序中，没有出现SQL语句，完全是标准的对象操作，其中session.save（user）语句告诉Hibernate，将user对象存入数据库表。这里非常重要的是session和Transaction，后面会详细介绍。

12.3 Hibernate配置

Hibernate配置文件可以有两种格式，一种是properties属性文件：hibernate.properties，另一种是xml文件：hibernate.cfg.xml。

开发者可以使用任何一种配置文件来配置Hibernate，后者稍微方便一些，当增加hbm映射文件时，可以直接在hibernate.cfg.xml中增加，不必像hibernate.properties必须在初始化代码中加入。

但不管怎么说，两种格式的配置项都是一样的，下面详细介绍Hibernate的配置属性，在Hibernate的src目录下有一个hibernate.properties模板，模板中已经列出了Hibernate的相关属性，开发者只要修改这些属性值即可。下面介绍常见的属性值。

12.3.1　配置数据库连接

从前面的示例中可以看到，配置文件中可以配置数据库连接信息，下面给出一个xml格式的配置文件hibernate.cfg.xml：

```
    <? xml version="1.0"encoding="utf8"? >
    <! DOCTYPE hibernateconfiguration PUBLIC
    "//Hibernate/Hibernate Configuration DTD
3.0//EN"
    "http: //hibernate.sourceforge.net/hibernateconf
iguration3.0.dtd">
    <hibernateconfiguration>
    <sessionfactory>
    <! 显示实际操作数据库时的SQL>
    <property name="show__sql">
    true
    </property>
    <! SQL方言，这边设定的是MySQL>
    <property name="dialect">
    org.hibernate.dialect.MySQLDialect
    </property>
    <! JDBC驱动程序>
    <property name="connection.driver__class">
    com.mysql.jdbc.Driver
    </property>
    <! JDBC URL>
    <property name="connection.url">
    jdbc: mysql: //localhost/demo
    </property>
    <! 数据库使用者>
```

```
    <property name="connection.username">
    caterpillar
    </property>
    <! 数据库密码>
    <property name="connection.password">
    123456
    </property>
    <! HHibernate预设的Connection pool>
    <property name="connection.pool__size">
    2
    </property>
    <! 对象与数据库表格映像文件>
    <mapping
resource="onlyfun/caterpillar/User.hbm.xml"/>
    </sessionfactory>
    </hibernateconfiguration>
```

这是xml格式的配置文件，下面是hibernate.properties格式配置内容：

```
    #显示实际操作数据库时的SQL
    hibernate.show__sql=true
    #SQL方言，这边设定的是MySQL
    hibernate.dialect=org.hibernate.dialect.MySQLDia
lect
    #JDBC驱动程序
    hibernate.connection.driver__
class=com.mysql.jdbc.Driver
    #JDBC URL
    hibernate.connection.url=jdbc:
mysql: //localhost/demo
    #数据库使用者
    hibernate.connection.username=caterpillar
```

```
#数据库密码
hibernate.connection.password=123456
#Hibernate预设的Connection pool
hibernate.connection.pool__size=2
```

注意：配置文件中的数据库连接信息是可选的，并不是必须的，Hibernate支持多种数据库连接配置方式。

另一种方式是使用C3P0连接池配置数据库连接。C3P0是随Hibernate发行包一起发布的一个开放源代码JDBC连接池，开发者可以在lib目录中找到。假若你设置了hibernate.c3p0.*属性，Hibernate会使用内置的C3P0 ConnectionProvider作为连接池。对Apache DBCP和Proxool的支持也是内置的。开发者必须设置hibernate.dbcp.属性（DBCP连接池属性）和hibernate.dbcp.ps.（DBCP语句缓存属性）才能使用DBCP ConnectionProvider。

说明：要详细了解相关资料，请查阅Apache commonspool的文档。要使用C3P0，必须将c3p0.jar文件放在CLASSPATH路径下。

如果开发者想要用Proxool，需要设置hibernate.proxool.系列属性。下面给出使用C3P0连接池配置数据库连接的示例hibernate.cfg.xml：

```
    <? xml version="1.0"encoding="utf8"? >
    <! DOCTYPE hibernateconfiguration PUBLIC
    "//Hibernate/Hibernate Configuration DTD
3.0//EN"
    "http: //hibernate.sourceforge.net/hibernateconf
iguration3.0.dtd">
    <hibernateconfiguration>
    <sessionfactory>
    <! 显示实际操作数据库时的SQL>
    <property name="show__sql">
    true
    </property>
    <! SQL方言，这边设定的是MySQL>
    <property name="dialect">
    org.hibernate.dialect.MySQLDialect
    </property>
    <! JDBC驱动程序>
    <property name="connection.driver__class">
    com.mysql.jdbc.Driver
    </property>
    <! JDBC URL>
```

```
    <property name="connection.url">
    jdbc: mysql: //localhost/demo
    </property>
    <! 数据库使用者>
    <property name="connection.username">
    caterpillar
    </property>
    <! 数据库密码>
    <property name="connection.password">
    123456
    </property>
    <! C3P0连接池设定>
    <property name="c3p0.min__size">
    5
    </property>
    <property name="c3p0.max__size">
    20
    </property>
    <property name="c3p0.timeout">
    1800
    </property>
    <property name="c3p0.max__statements">
    50
    </property>
    <! 对象与数据库表格映像文件>
    <mapping
resource="onlyfun/caterpillar/User.hbm.xml"/>
    </sessionfactory>
    </hibernateconfiguration>
```

下面是hibernate.properties格式配置内容：

```
    #显示实际操作数据库时的S Q L
    hibernate.show__sql=true
    #SQL方言，这边设定的是MySQL
```

```
    hibernate.dialect=org.hibernate.dialect.MySQLDia
lect#JDBC驱动程序
    hibernate.connection.driver__
class=com.mysql.jdbc.Driver#JDBC URL
    hibernate.connection.url=jdbc:
mysql: //localhost/demo
    #数据库使用者
    hibernate.connection.username=caterpillar
    #数据库密码
    hibernate.connection.password=123456#C3P0连接池设
定
    hibernate.c3p0.min__size=5
    hibernate.c3p0.max__size=20
    hibernate.c3p0.timeout=1800
    hibernate.c3p0.max__statements=50
```

如果在Application Server内使用时，Hibernate可以从JNDI中注册的javax.sql.Datasource取得连接。需要设置如下属性，如表12.1所示。

表 12.1　Hibernate 数据源(Datasource)属性

属性名称	说　明
hibernate. connection. datasource	JNDI 名字
hibernate. jndi. url	提供者的 URL（可选）
hibernate. jndi. class	InitialContextFactory 的类名（可选）
hibernate. connection. username	数据库用户名（可选）
hibernate. connection. password	数据库密码（可选）

例如，现在有一个数据源test，下面是相关的配置属性：

```
    #SQL方言，这边设定的是MySQL
    hibernate.dialect
net.sf.hibernate.dialect.MySQLDialect
    #指定数据源名称
    hibernate.connection.datasource test
    #指定provider__class类
    hibernate.connection.provider__class
net.sf.hibernate.connection.Datasource
    ConnectionProvider
```

其他参数就不必写了，因为已经在App Server配置连接池时指定好了。如果你不是在App Server环境中使用Hibernate，例如远程客户端程序，但是你又想用App Server的数据库连接池，那么你还需要配置JNDI的参数，例如Hibernate连接远程Weblogic上的数据库连接池：

```
    #SQL方言，这边设定的是MySQL
    hibernate.dialect
net.sf.hibernate.dialect.MySQLDialect
    #指定数据源名称
    hibernate.connection.datasource test
    #指定provider__class类
    hibernate.connection.provider__class
net.sf.hibernate.connection.Datasource
    ConnectionProvider
    hibernate.jndi.class
weblogic.jndi.WLInitialContextFactory
```

```
    #指定JNDI URL地址
    hibernate.jndi.url t3: //servername: 7001/
```

最后，如果你需要在EJB或者JTA中使用Hibernate，需要取消下行的注释：

```
    hibernate.transaction.factory__class
net.sf.hibernate.transaction.JTATransa
    ctionFactory
```

12.3.2　其他配置

表12.2是一些在运行时可以改变Hibernate行为的其他配置。所有这些都是可选的，也有合理的默认值。

系统级别的配置只能通过javaDproperty=value或者在hibernate.properties文件中配置，而不能通过传递给Configuration的Properties实例来配置。

表 12.2　Hibernate 配置属性

属　性　名	用　　途
hibernate. dialect	Hibernate 方言(Dialect)的类名，可以让 Hibernate 使用某些特定的数据库平台的特性 取值：full. classname. of. Dialect
hibernate. default_ schema	在生成的 SQL 中，schemla/tablespace 的全限定名 取值：SCHEMA_ NAME
hibernate. session_ factory_ name	把 SessionFactory 绑定到 JNDI 中去 取值：jndi/composite/name
hibernate. use_ outer_ join	允许使用外连接抓取 取值：true | false
hibernate. max_ fetch_ depth	设置外连接抓取树的最大深度 取值：建议设置为 0 到 3 之间
hibernate. jdbc. fetch_ size	一个非零值，用来决定 JDBC 的获取量大小，(会调用 calls Statement. setFetchSize( ))
hibernate. jdbc. batch_ size	一个非零值，会开启 Hibernate 使用 JDBC2 的批量更新功能 取值：建议值在 5 和 30 之间
hibernate. jdbc. use_ scrollable_ resultset	允许 Hibernate 使用 JDBC2 提供的可滚动结果集。只有在使用用户自行提供的连接时，这个参数才是必需的。否则 Hibernate 会使用连接的元数据(metadata)。 取值：true | false
hibernate. jdbc. use_ streams_ for_ binary	在从 JDBC 读写 binary(二进制)或者 serializable(可序列化)类型时，是否使用 stream(流) 这是一个系统级别的属性。 取值：true | false
hibernate. cglib. use_ reflection_ optimizer	是否使用 CGLIB 来代替运行时反射操作。(系统级别属性，默认为在可能时都使用 CGLIB)在调试的时候有时候使用反射会有用。 取值：true | false
hibernate. jndi. < propertyName >	把 propertyName 这个属性传递到 JNDI InitialContextFactory 去 (可选)
hibernate. connection. isolation	事务隔离级别 (可选) 取值：1，2，4，8
hibernate. connection. < propertyName >	把 propertyName 这个 JDBC 属性传递到 DriverManager. getConnection( ) 去
hibernate. connection. provider_ class	指定一个自定义的 ConnectionProvider 类名 取值：classname. of. ConnectionProvider
hibernate. cache. provider_ class	指定一个自定义的 CacheProvider 缓存提供者的类名 取值：classname. of. CacheProvider
hibernate. cache. use_ minimal_ puts	优化第二层缓存操作，减少写操作，代价是读操作更频繁(对于集群缓存很有用) 取值：true | false
hibernate. cache. use_ query_ cache	打开查询缓存 取值：true | false
hibernate. cache. region_ prefix	用于第二层缓存区域名字的前缀 取值：prefix
hibernate. transaction. factory_ class	指定一个自定义的 TransactionFactory 类名，Hibernate Transaction API 将会使用 取值：classname. of. TransactionFactory

（续）

属 性 名	用 途
jta. UserTransaction	JTATransactionFactory 用来获取 JTA UserTransaction 的 JNDI 名 取值：jndi/composite/name
hibernate. transaction. manager_ lookup_ class	TransactionManagerLookup 的类名，当在 JTA 环境中，JVM 级别的缓存被打开时使用 取值：classname. of. TransactionManagerLookup
hibernate. query. substitutions	把 Hibernate 查询中的一些短语替换为 SQL 短语（比如说短语可能是函数或者字符） 取值：hqlLiteral = SQL_ LITERAL，hqlFunction = SQLFUNC
hibernate. show_ sql	把所有的 SQL 语句都输出到控制台（可以作为 log 功能的一个替代） 取值：true | false
hibernate. hbm2ddl. auto	自动输出 schema 创建 DDL 语句 取值：update | create | create-drop

12.3.3 SQL方言

Hibernate使用方言（dialect）来实现数据库操作跨平台性。用户必须在Hibernate的配置文件中比如hibernate.cfg.xml中对所使用的Hibernate方言加以设置。例如：

```
    <property name="dialect">
org.hibernate.dialect.OracleDialect</property>
```

表12.3给出了Hibernate的SQL方言列表。

表 12.3　Hibernate SQL 方言（hibernate. dialect）

RDBMS	方　言
DB2	org. hibernate. dialect. DB2Dialect
DB2 AS/400	org. hibernate. dialect. DB2400Dialect
DB2 OS390	org. hibernate. dialect. DB2390Dialect
PostgreSQL	org. hibernate. dialect. PostgreSQLDialect
MySQL	org. hibernate. dialect. MySQLDialect
MySQL with InnoDB	org. hibernate. dialect. MySQLInnoDBDialect
MySQL with MyISAM	org. hibernate. dialect. MySQLMyISAMDialect
Oracle（any version）	org. hibernate. dialect. OracleDialect
Oracle 9i/10g	org. hibernate. dialect. Oracle9Dialect
Sybase	org. hibernate. dialect. SybaseDialect
Sybase Anywhere	org. hibernate. dialect. SybaseAnywhereDialect
Microsoft SQL Server	org. hibernate. dialect. SQLServerDialect
SAP DB	org. hibernate. dialect. SAPDBDialect
Informix	org. hibernate. dialect. InformixDialect
HypersonicSQL	org. hibernate. dialect. HSQLDialect
Ingres	org. hibernate. dialect. IngresDialect
Progress	org. hibernate. dialect. ProgressDialect

（续）

RDBMS	方　言
Mckoi SQL	org. hibernate. dialect. MckoiDialect
Interbase	org. hibernate. dialect. InterbaseDialect
Pointbase	org. hibernate. dialect. PointbaseDialect
FrontBase	org. hibernate. dialect. FrontbaseDialect
Firebird	org. hibernate. dialect. FirebirdDialect

12.3.4　查询语言中的替换

可以使用hibernate.query.substitutions在Hibernate中定义新的查询符号，例如：

```
hibernate.query.substitutions true=1, false=0
```

将导致符号true和false在生成的SQL中被翻译成整数常量。

```
hibernate.query.substitutions toLowercase=LOWER
```

将允许重命名SQL中的LOWER函数。

12.3.5 日志

Hibernate使用Apache commonslogging来为各种事件记录日志，commonslogging将直接输出到Apache Log4j（如果在类路径中包括log4j.jar）或JDK1.4 logging（如果运行在JDK1.4或以上的环境下）。

在前面介绍的Hibernate类库中，已经包含了Log4j的类库，如果没有，你可以从http：//jakarta.apache.org下载Log4j。要使用Log4j，需要将log4j.properties文件放置在类路径下，随Hibernate一同分发的样例属性文件在src/目录下。表12.4给出了Log4j的日志类别。

表 12.4　Log4j 的日志类别

类　别	功　能
org. hibernate. SQL	在所有 SQL DML 语句被执行时为它们记录日志
org. hibernate. type	为所有 JDBC 参数记录日志
org. hibernate. tool. hbm2ddl	在所有 SQL DDL 语句执行时为它们记录日志
org. hibernate. pretty	在 session 清洗(flush)时，为所有与其关联的实体(最多 20 个)的状态记录日志
org. hibernate. cache	为所有二级缓存的活动记录日志
org. hibernate. transaction	为事务相关的活动记录日志
org. hibernate. jdbc	为所有 JDBC 资源的获取记录日志
org. hibernate. hql. AST	在解析查询时，记录 HQL 和 SQL 的 AST 分析日志
org. hibernate. secure	为 JAAS 认证请求做日志
org. hibernate	为任何 Hibernate 相关信息做日志（信息量较大，但对查错非常有帮助）

说明：在使用MyEclipse作为开发工具时，运行Hibernate应用都会显示一些警告，提示没有配置Log4j，这是因为项目的lib中有Log4j类库，但是开发者没有配置Log4j的属性文件，读者可以从Hibernate的安装包的project＼etc目录下找到log4j.properties配置文件，复制到项目的路径中。

该配置文件内容如下：

```
    ###输出日志信息到控制台###
    log4j.appender.stdout=org.apache.log4j.ConsoleAp
pender
    log4j.appender.stdout.Target=System.out
```

```
    log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
    log4j.appender.stdout.layout.ConversionPattern=%d {ABSOLUTE} %5p%c {1} : %L%m%n
    ###输出日志到hibernate.log文件###
    #log4j.appender.file=org.apache.log4j.FileAppender
    #log4j.appender.file.File=hibernate.log
    #log4j.appender.file.layout=org.apache.log4j.PatternLayout
    #log4j.appender.file.layout.ConversionPattern=%d {ABSOLUTE} %5p%c {1} : %L%m%n
    ###设置日志优先级###
    log4j.rootLogger=warn, stdout
    log4j.logger.org.hibernate=debug
    ###配置JDBC绑定参数###
    log4j.logger.org.hibernate.type=info
```

读者可以将该文件复制到本章的示例项目路径中重新运行程序，输出如下：

```
    11: 23: 45，543 INFO Environment: 479Hibernate 3.1.3
    11: 23: 45，558 INFO Environment: 509hibernate.properties not found
    11: 23: 45，574 INFO Environment: 525using CGLIB reflection optimizer
    ……
    11: 23: 48，355 DEBUG AbstractEntityPersister: 2450Snapshot select: select user__.id, user__.name as name0
    __，user__.age as age0__from user user__where user__.id=?
```

```
    11: 23: 48，433 DEBUG AbstractEntityPersister:
2452Insert 0: insert into user（name, age, id）
values（？，？，
    ？）
    11: 23: 48，433 DEBUG AbstractEntityPersister:
2453Update 0: update user set name=？，age=？where
id=?
    11: 23: 48，433 DEBUG AbstractEntityPersister:
2454Delete 0: delete from user where id=?
    11: 23: 48，480 DEBUG AbstractEntityPersister:
2457Identity insert: insert into user（name, age）
values（？，
    ？）
    11: 23: 48，590 DEBUG EntityLoader: 79Static
select for entity base.helloworld.User: select
user0__.id as id0__
    0__，user0__.name as name0__0__，user0__.age as
age0__0__from user user0__where user0__.id=?
    11: 23: 48，621 DEBUG EntityLoader: 79Static
select for entity base.helloworld.User: select
user0__.id as id0__
    0__，user0__.name as name0__0__，user0__.age as
age0__0__from user user0__where user0__.id=?
    ……
    11: 23: 49，168 INFO SessionFactoryImpl:
729closing
    11: 23: 49，168 INFO
DriverManagerConnectionProvider: 147cleaning up
connection pool: jdbc: mysql: //lo
    calhost: 3306/ssh
    增加用户数据成功，请查看数据库数据!
```

读者将会在日志中看到JDBC执行的SQL语句，而在示例代码中并没有出现这些语句，这对于开发者理解

对象持久化过程有着重要意义。

第13章　Hibernate核心API

Hibernate是一种对JDBC做了轻量级封装的对象——关系映射工具，所谓轻量级封装，是指Hibernate并没有完全封装JDBC，Java应用即可以通过Hibernate API访问数据库，还可以绕过Hibernate API，直接通过JDBC API来访问数据库。下面介绍Hibernate的核心API，使读者详细了解Configuration接口、SessionFactory接口和Session接口。并通过示例使读者快速掌握其用法。

13.1　Session介绍

Session是Hibernate运作的中心，对象的生命周期、事务的管理、数据库的存取都与Session息息相关。在Hibernate中使用持久化对象PO（Persistent

Object）完成持久化操作，对PO的操作必须在Session管理下才能同步到数据库。

13.1.1 Configuration

Hibernate具有六个核心接口或者是类，如图13.1所示。

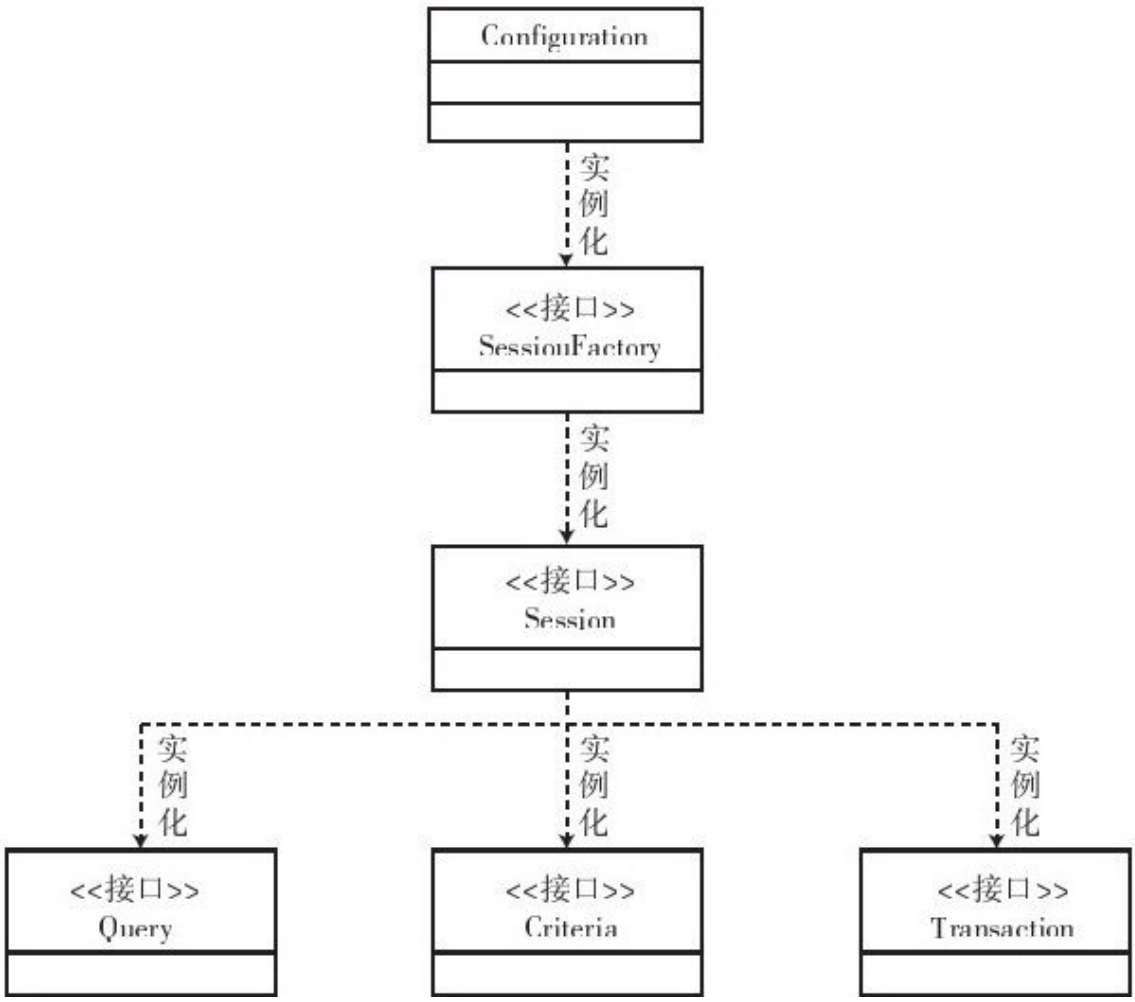

图　13.1　Hibernate核心接口及其关系

从图13.1中可以看到，首先要做的是需要创建Configuration对象。Configuration对象用于配置和启动Hibernate。Hibernate应用通过Configuration实例来指定对象/关系映射文件的位置或者动态配置Hibernate的属性，然后创建SessionFactory实例。

一个org.hibernate.cfg.Configuration实例代表了一个应用程序中Java类型到SQL数据库映射的完整集合。Configuration被用来构建一个（不可变的（immutable））SessionFactory，映射定义则由不同的XML映射定义文件编译而来。读者可以查看Configuration的源代码，它的configure（）方法是这样实现的：

```
    public Configuration configure（）throws
HibernateException {
    configure（"/hibernate.cfg.xml"）; //此处指定了ORM
文件的位置
    return this;
    }
```

这里指定了ORM文件的位置，这就是为什么Hibernate总是默认到classpath下去寻找hibernate.cfg.xml文件的原因了。实际上我们还可以通过configure（String resource）来动态地指定配置文件，只不过通常我们都是采用默认设置罢了。这

样的话我们的配置文件就都被读取了，同时配置文件中通过＜mapping＞元素引入的映射文件也被读取了。

可以直接实例化Configuration来获取一个实例，并为它指定XML映射定义文件，如果映射定义文件在类路径（classpath）中，请使用addResource（）方法添加：

```
    Configuration cfg=new Configuration（）
    .addResource（"box.hbm.xml"）//增加映射文件
box.hbm.xml
    .addResource（"user.hbm.xml"）; //增加映射文件
user.hbm.xml
```

除了可以指定xml文件，还可以指定被映射的类，让Hibernate帮你寻找映射定义文件，如：

```
    Configuration cfg=new Configuration（）
    .addClass（org.hibernate.auction.Box.class）//指
定被映射的类Box.class
    .addClass（org.hibernate.auction.User.class）; //
指定被映射的类User.class
```

Hibernate将会在类路径（classpath）中寻找名字为/org/hibernate/auction/box.hbm.xml和/org/hibernate/auction/user.hbm.xml映射定义文件，这种方式消除了任何对文件名的硬编码，推荐使用。

Configuration也允许开发者指定Hibernate配置属性，例如：

```
    Configuration cfg=new Configuration（）
    .addClass（org.hibernate.auction.Box.class）//增加映射文件
    .addClass（org.hibernate.auction.User.class）
    //增加Hibernate配置参数
    .setProperty（"hibernate.dialect", "org.hibernate.dialect.MySQLInnoDBDialect"）
    .setProperty（"hibernate.connection.datasource", "java: comp/env/jdbc/test"）
    .setProperty（"hibernate.order__updates", "true"）;
```

读者可以参考上一章关于Hibernate配置项的介绍。配置Hibernate有如下几种方式：传一个

java.util.Properties实例给Configuration.setProperties（）。

将hibernate.properties放置在类路径（classpath）的根目录下（root directory）。

通过javaDproperty=value来设置系统（System）属性。

在hibernate.cfg.xml中加入元素＜property＞。

注意：Configuration实例是一个启动期间（startuptime）对象，一旦SessionFactory创建完成它就被丢弃了。

13.1.2 SessionFactory

Configuration负责创建SessionFactory实例，是通过Configuration的buildSessionFactory方法来创建的，buildSessionFactory方法把Configuration对象所包含的所有配置信息都复制到SessionFactory对象的缓存当中。同时SessionFactory对象创建之后就不再和Configuration对象关联了，即Configuration完成了使命，被丢弃了。

一个SessionFactory对象就代表一个数据库存储源，通常一个应用程序只需要创建一个SessionFactory实例即可。由此看出SessionFactory具备如下两个特点。

线程安全：整个应用共用一个SessionFactory实例。

重量级：在SessionFactory中存放了Hibernate配置信息以及映射元素据信息，这些都需要很大的缓存。

一般来说，SessionFactory的主要作用就是创建Session对象，所有线程都是从SessionFactory中获取Session对象来处理客户请求的。下面是创建SessionFactory实例的代码：

```
    Configuration config=new
Configuration（）.configure（）; //Configuration管理
Hibernate配置
    //根据Configuration建立SessionFactory
    //SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）;
```

注意：一般情况下，一个项目通常只需要一个SessionFactory就够，当需要操作多个数据库时，可以为每个数据库指定一个SessionFactory。

13.1.3 创建Session

Session接口负责执行被持久化对象的CRUD操作（CRUD的任务是完成与数据库的交流，包含了很多常见的SQL语句）。但需要注意的是Session对象是非线程安全的。同时，Hibernate的Session不同于JSP应用中的HttpSession。这里用Session术语时，其实指的是Hibernate中的Session，而以后会将HttpSesion对象称为用户Session。

Session的生命周期绑定在一个物理的事务（tansaction）上面，Session的主要功能是提供对映射的实体类实例的创建、读取和删除操作。实例可能以下面三种状态存在：

自由状态（transient）。不曾进行持久化，未与任何Session相关联。

持久化状态（persistent）。仅与一个Session相关联。

游离状态（detached）。已经进行过持久化，但当前未与任何Session相关联。

游离状态的实例可以通过调用save（）、persist（）或者saveOrUpdate（）方法进行持久化。持久化实例可以通过调用delete（）变成游离状态。通过get（）或load（）方法得到的实例都是持久化状态的。游离状态的实例可以通过调用update（）、saveOrUpdate（）、lock（）或者replicate（）进行持久化。游离或者自由状态下的实例可以通过调用merge（）方法成为一个新的持久化实例。

save（）和persist（）将会引发SQL的INSERT操作，delete（）会引发SQL的DELETE操作，而update（）或merge（）会引发SQL的UPDATE操作。对

持久化（persistent）实例的修改在刷新提交时会被检测到，它也会引起SQL的UPDATE操作。

saveOrUpdate（）或者replicate（）会引发SQL的INSERT或者UPDATE操作。如图13.2所示显示了实例的三种状态及其转换关系。

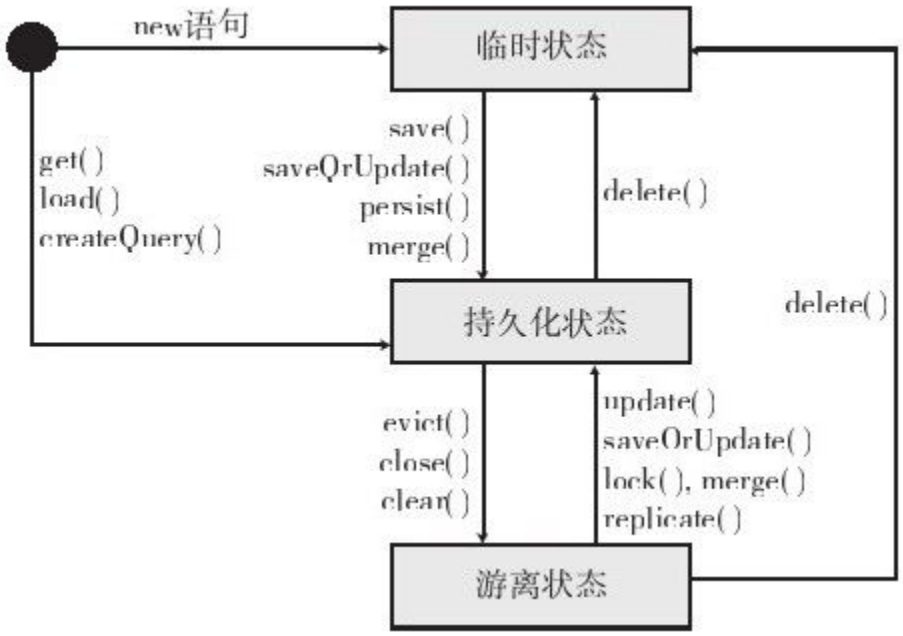

图　13.2　实例的三种状态及其转换关系

其具体实现并不一定是线程安全的。每个线程/事务应该从一个SessionFactory获取自己的Session实例。如果其持久化对象类是可序列化的，则Session实

例也是可序列化的。一个典型的事务应该使用下面的形式：

```
    Session sess=factory.openSession（）; //创建
session
    Transaction tx;
    try {
    tx=sess.beginTransaction（）;
    //业务逻辑处理
    ……
    //事务提交
    tx.commit（）;
    }
    catch（Exception e） {
    if（tx! =null）tx.rollback（）;
    throw e;
    }
    finally {
    sess.close（）;
    }
```

如果Session抛出了异常，事务必须回滚而Session会被废弃。在异常发生后Session的内部状态可能会与数据库失去同步。

Hibernate在对数据库进行操作之前，必须先取得Session实例，相当于JDBC在对数据库操作之前，必须

先取得Connection实例，Session是Hibernate操作的基础，它不是设计为线程安全（Threadsafe），一个Session由一个线程来使用。下面给出Session的常用方法。

Transaction beginTransaction（）：开始一个工作单元并且返回相关联的事务（Transaction）对象。

void cancelQuery（）：终止执行当前查询。

void clear（）：完整地清除这个Session。

Connection close（）：停止这个Session，通过中断JDBC连接并且清空（cleaning up）它。

Connection connection（）：获取这个Session的JDBC连接。如果这个Session使用了积极的collection释放策略（如CMT容器控制事务的环境

下），关闭这个调用的连接的职责应该由当前应用程序负责。

boolean contains（Object object）：检查这个对象实例是否与当前的Session关联（即是否为Persistent状态）。

Criteria createCriteria（Class persistentClass）：为给定的实体类或它的超类创建一个新的Criteria实例。

Query createQuery（String queryString）：根据给定的HQL查询条件创建一个新的Query实例。

SQLQuery createSQLQuery（String queryString）：根据给定的SQL查询条件创建一个新的SQLQuery实例。

void delete（Object object）：从数据库中移除持久化（persistent）对象的实例。

void delete（String entityName, Object object）：从数据库中移除持久化（persistent）对象的实例。

void flush（）：强制提交清理（flush）Session。

Object get（Class clazz, Serializable id）：根据给定标识和实体类返回持久化对象的实例，如果没有符合条件的持久化对象，实例则返回null。

SessionFactory getSessionFactory（）：获取创建这个Session的SessionFactory实例。

Transaction getTransaction（）：获取与这个Session关联的Transaction（事务）实例。

Object load（Class the Class, Serializable id）：在符合条件的实例存在的情况下，根据给定的实体类和标识返回持久化状态的实例。

Object merge（Object object）：将给定的对象的状态复制到具有相同标识的持久化对象上。

void persist（Object object）：将一个自由状态（Transient）的实例持久化。

void replicate（Object object, ReplicationMode replicationMode）：使用当前的标识值持久化给定的游离状态（Transient）的实体。

Serializable save（Object object）：首先为给定的自由状态（Transient）的对象（根据配置）生

成一个标识并赋值，然后将其持久化。

void saveOrUpdate（Object object）：根据给定的实例的标识属性的值（注：可以指定unsavedvalue。一般默认null）来决定执行save（）或update（）操作。

void setReadOnly（Object entity, boolean readOnly）：将一个未经更改的持久化对象设置为只读模式，或者将一个只读对象标记为可以修改的模式。

void update（Object object）：根据给定的detached（游离状态）对象实例的标识，更新对应的持久化实例。

void update（String entityName, Object object）：根据给定的detached（游离状态）对象实

例的标识，更新对应的持久化实例。

13.2　简单的CRUD示例

CRUD是指在做计算处理时的增加、查询（重新得到数据）、更新和删除几个单词的首字母简写。主要被用在描述软件系统中数据库或者持久层的基本操作功能。下面示例演示了如何创建session实例，并使用session的相关方法，实现对数据库表记录的添加、修改、删除和查找。

技术要点

数据库表的操作，基本上是对数据的增加、删除、修改和查询操作，也就是CRUD，读者掌握基本的CRUD操作非常重要，也是进一步进行复杂数据库操作的基础，本节将以一个示例介绍如何使用Hibernate技术实现简单的CRUD操作，为读者进一步理解和掌握Hibernate打下良好基础。

基本的Hibernate配置。

基本的session创建。

使用session完成数据的CRUD操作。

实现代码

首先登录MySQL数据库，建立表box，SQL语句如下：

```
CREATE TABLE'box'（
'id'int（11）NOT NULL auto__increment，
'width'float default NULL，
'length'float default NULL，
'height'float default NULL，
'name'varchar（20）default NULL，
PRIMARY KEY（'id'）
）ENGINE=InnoDB DEFAULT CHARSET=gbk
```

Hibernate配置文件hibernate.cfg.xml：

```
＜? xml version='1.0'encoding='UTF8'? ＞
＜! DOCTYPE hibernateconfiguration PUBLIC
```

```
    "//Hibernate/Hibernate Configuration DTD
3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatecon
figuration3.0.dtd">
    <! Hibernate配置.
    <hibernateconfiguration>
    <sessionfactory>
    <! 数据库用户名>
    <property name="connection.username">root
</property>
    <! 数据库连接URL>
    <property name="connection.url">
    jdbc: mysql: //localhost: 3306/ssh
    </property>
    <property name="dialect">
    org.hibernate.dialect.MySQLDialect
    </property>
    <property name="myeclipse.connection.profile">
Mysql</property>
    <property name="hbm2ddl.auto">update
</property>
    <! 数据库密码>
    <property name="connection.password">pla
</property>
    <! 数据库驱动>
    <property name="connection.driver_class">
    com.mysql.jdbc.Driver
    </property>
    <! 数据库表Java对象映射文件表>
    <mapping
resource="h1/session/test/Box.hbm.xml"/>
    </sessionfactory>
    </hibernateconfiguration>
```

Java对应的类Box. java：

```
    package helloworld.session.test;
    public class Box implements
java.io.Serializable {
    private Integer id; //ID属性变量
    private Float width; //宽度width属性变量
    private Float length; //长度length属性变量
    private Float height; //高度height属性变量
    private String name; //名称name属性变量
    public Box（） {//构造器
    }
    public Integer getId（） {//ID属性的Getter（）方法
和Setter（）方法
    return this.id;
    }
    public void setId（Integer id） {
    this.id=id;
    }
    public Float getWidth（） {//width属性的Getter（）
方法和Setter（）方法
    return this.width;
    }
    public void setWidth（Float width） {
    this.width=width;
    }
    public Float getLength（） {//length属性的
Getter（）方法和Setter（）方法
    return this.length;
    }
    public void setLength（Float length） {
    this.length=length;
    }
    public Float getHeight（） {//height属性的
Getter（）方法和Setter（）方法
    return this.height;
    }
    public void setHeight（Float height） {
```

```
    this.height=height;
    }
    public String getName（） {//name属性的Getter（）
方法和Setter（）方法
    return this.name;
    }
    public void setName（String name） {
    this.name=name;
    }
    }
```

Box类的配置文件box.hbm.xml：

```
    <? xml version="1.0"encoding="utf8"? >
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
    <! 配置映射文件>
    <class
name="helloworld.session.test.Box"table="box"catal
og="ssh">
    <id name="id"type="java.lang.Integer"><! id映
射为Integer类型>
    <column name="id"/><! 对应数据库表的id字段>
    <generator class="native"/><! 主键定义为自动生成
>
    </id>
    <property name="width"type="java.lang.Float">
<! 映射宽度属性变量>
    <column name="width"precision="12"scale="0"/>
    </property>
```

```
    <property name="length"type="java.lang.Float">
<! 映射长度属性变量>
    <column name="length"precision="12"scale="0"/>
    </property>
    <property name="height"type="java.lang.Float">
<! 映射高度属性变量>
    <column name="height"precision="12"scale="0"/>
    </property>
    <property name="name"type="java.lang.String">
<! 映射名称属性变量>
    <column name="name"length="20"/>
    </property>
    </class>
    </hibernatemapping>
```

至此，第一个Hibernate应用就完成了，下面编写一个测试程序，实现box表的CRUD操作。Test.java文件：

```
    package helloworld.session.test;
    import java.util.List;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.Transaction;
    import org.hibernate.cfg.Configuration;
    public class Test {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（）.configure（）;
    //根据Configuration建立SessionFactory
    //SessionFactory用来建立Session
```

```
    SessionFactory
sessionFactory=config.buildSessionFactory（）；
    //显示所有记录
    //建立Session，相当于建立JDBC的Connection
    Session session=sessionFactory.openSession（）；
    List result=session.createQuery（"from
Box"）.list（）；//查询所有记录
    //显示记录
    System.out.println（"数据内容：id名称长度宽度高
度"）；
    for（int i=0；i＜result.size（）；i++）｛
    Box box=（Box）result.get（i）；
    System.out.println（""+box.getId（）
+""+box.getName（）+""
    +box.getLength（）+""+box.getWidth（）+""
    +box.getHeight（））；
    ｝
    session.close（）；
    //增加记录
    Box box=new Box（）；
    box.setHeight（24.3f）；
    box.setLength（100.00f）；
    box.setWidth（45.00f）；
    box.setName（"My Box"）；
    session=sessionFactory.openSession（）；
    Transaction tx=session.beginTransaction（）；
    //将对象保存到数据库当中，并获得id值
    Integer boxid=（Integer）session.save（box）；
    tx.commit（）；
    session.close（）；
    System.out.println（"id为"+boxid+"的记录已经添
加！"）；
    session=sessionFactory.openSession（）；//查询添
加的记录数据
    tx=session.beginTransaction（）；
    box=（Box）session.load（Box.class, boxid）；
```

```
    System.out.println（"id为"+boxid+"数据修改前：id名称长度宽度高度"）；
    System.out.println（""+box.getId（）+""
    +box.getName（）+""+box.getLength（）+""+box.getWidth（）
    +""+box.getHeight（））；
    //修改对象
    box.setName（"Update Box！"）；
    box.setWidth（35.6f）；
    session.update（box）；//更新数据库
    tx.commit（）；//提交事务
    //查看更新结果
    box=（Box）session.load（Box.class, boxid）；
    System.out.println（"id为"+boxid+"数据修改前：id名称长度宽度高度"）；
    System.out.println（""+box.getId（）+""
    +box.getName（）+""+box.getLength（）+""+box.getWidth（）
    +""+box.getHeight（））；
    session.close（）；
    //删除记录
    session=sessionFactory.openSession（）；
    tx=session.beginTransaction（）；
    box=（Box）session.load（Box.class, boxid）；
    session.delete（box）；
    tx.commit（）；//提交事务
    System.out.println（"id为"+boxid+"的记录已经删除！"）；
    //查看结果，如果没有记录，get返回null，如果使用load方法，则抛出异常
    box=（Box）session.get（Box.class, boxid）；
    if（box==null）
    System.out.println（"找不到对应的记录："+boxid）；
    session.close（）；
    sessionFactory.close（）；//关闭sessionFactory
    }
    }
```

运行Test，结果如下：

```
数据内容：id名称长度宽度高度
3 Update Box! 100.0 null 24.3
4 Update Box! 100.0 null 24.3
5 Update Box! 100.0 35.6 24.3
6 My Box 100.0 null 24.3
id为31的记录已经添加!
id为31数据修改前：id  名称长度  宽度  高度
31 My Box 100.0 45.0 24.3
id为31数据修改前：id  名称长度  宽度  高度
31 Update Box! 100.0 35.6 24.3
id为31的记录已经删除!
找不到对应的记录：31
```

源程序解读

（1）hibernate. cfg.xml是Hibernate配置文件。其中配置了数据库连接的相关信息，并把Box.hbm.xml加入到数据库表Java对象映射文件表中。

（2）Box类文件中除了定义对应的属性，还定义了构造器，其中id属性很重要，是主键属性。

（3）为了定义id主键的生成方式，需要在b.hbm.xml文件中定义generator属性，这里定义为native，表示主键的生成方式由Hibernate根据数据库Dialect的定义来决定。本例中使用键表定义的自动增长值。

（4）Test. java测试程序中，使用一个HQL（Hibernate Query Language-Hibernate查询语言）查询语句来从数据库中加载所有存在的Box对象。

（5）使用了一个boxid值来标记添加的记录主键，在修改、删除、查找等操作中，均使用了该主键值。

（6）session的load方法查询对象，如果查询不到，则抛出ObjectNotFoundException异常；使用get则返回null。

（7）从示例可以看到，开发者只需要对Java对象实例进行相应的操作，就可以实现数据库表的相应改变，简化了开发难度，数据库对开发者变得透明。

（8）Hibernate 3中，取消了find（）方法，必须通过Query或Criteria来进行数据查询。

（9）这里需要理解box.hbm.xml对象映射文件中的Hibernate主键生成方式Key Generator（主键生成器）。

该属性可以有以下几种选择。

assigned：主键由外部程序负责生成，无需Hibernate参与。

hilo：通过hi/lo算法实现的主键生成机制，需要额外的数据库表保存主键生成历史状态。

seqhilo：与hilo类似，通过hi/lo算法实现的主键生成机制，只是主键历史状态保存在Sequence中，适用于支持Sequence的数据库，如Oracle。

increment：主键按数值顺序递增。此方式的实现机制为在当前应用实例中维持一个变量，以保存着当前的最大值，之后每次需要生成主键时将此值加1作为主键。这种方式可能产生的问题是，如果当前有多个实例访问同一个数据库，那么由于各个实例各自维护主键状态，不同实例可能生成同样的主键，从而造成主键重复异常。因此，如果同一数据库有多个实例访问，此方式必须避免使用。

identity：采用数据库提供的主键生成机制。如DB2、SQL Server、MySQL中的主键生成机制。

sequence：采用数据库提供的sequence机制生成主键。如Oralce中的Sequence。

native：由Hibernate根据底层数据库自行判断采用identity、hilo、sequence其中一种作为主键生成方式（本例使用该选项）。

uuid.hex：由Hibernate基于128位唯一值产生算法生成16进制数值（编码后以长度32的字符串表示）作为主键。

uuid.string：与uuid.hex类似，只是生成的主键未进行编码（长度16）。在某些数据库中可能出现问题（如PostgreSQL）。

foreign：使用外部表的字段作为主键。

说明：一般而言，利用uuid.hex方式生成主键将提供最好的性能和数据库平台适应性。

（10）在标识主键时，建议使用相应的类对象，例如Long、Integer等，而不是相应的元数据类型long

和int，因为Hibernate是基于对象操作的。对于有Java编程经验的开发人员来说尤其需要注意。

13.3 Save还是Update

前面介绍了一个简单的示例，演示如何使用Hibernate来完成最简单的数据记录添加、删除、修改和查询。Hibernate使数据库开发变得透明，开发者只需要操作对象就能够实现数据库表的联动。我们知道在Hibernate中，对象有三种状态：自由状态、持久化状态和游离状态。那么对于对象的改变，Hibernate根据什么来判断对象是否被修改，如果被修改，是应该增加一条新记录还是修改某个记录？

技术要点

Hibernate是一个ORM工具，将Java对象和数据库对应起来，程序员通过操作Java对象来实现数据库的操作，而对象的持久化工作有Hibernate来实现。那么对于一个Java对象，Hibernate在将这个对象同数据库

表关联时，是更新一条记录还是插入一条记录那？判断的依据是什么？如何配置？本节将详细介绍。

对象的3种状态及其转换。

saveOrUpdate和unsavedvalue属性。

使用session完成数据的CRUD操作。

实现代码

本节的代码是在前一节代码的基础上做一些小的改动，在box.hbm.xml文件中增加了unsavedvalue属性，box.hbm.xml内容：

```
    <? xml version="1.0"encoding="utf8"? >
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
    <class
name="helloworld.session.test.Box"table="box"catal
og="ssh">
    <! 定义unsavedvalue为null>
```

```
    <id
name="id"type="java.lang.Integer"unsavedvalue="nul
l"><! 定义>
    <column name="id"></column>
    <generator class="native"></generator>
    </id>
    <property name="width"type="java.lang.Float">
<! 映射宽度属性变量>
    <column name="width"precision="12"scale="0"/>
    </property>
    <property name="length"type="java.lang.Float">
<! 映射长度属性变量>
    <column name="length"precision="12"scale="0"/>
    </property>
    <property name="height"type="java.lang.Float">
<! 映射高度属性变量>
    <column name="height"precision="12"scale="0"/>
    </property>
    <property name="name"type="java.lang.String">
<! 映射名称属性变量>
    <column name="name"length="20"/>
    </property>
    </class>
    </hibernatemapping>
```

增加一个Hibernate测试类SaveOrUpdate.java：

```
    package helloworld.session.test;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.Transaction;
    import org.hibernate.cfg.Configuration;
    public class SaveOrUpdate {
    public static void main（String[]args） {
```

```
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（）.configure（）；
    //根据Configuration建立SessionFactory
    //SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）；
    Box box=new Box（）；//box对象为自由状态
    box.setHeight（24.3f）；
    box.setLength（100.00f）；
    box.setWidth（45.00f）；
    box.setName（"自由状态box"）；
    //建立Session，相当于建立JDBC的Connection
    Session session=sessionFactory.openSession（）；
    Transaction tx=session.beginTransaction（）；
    //将对象保存到数据库当中，并获得id值
    Integer boxid=（Integer）session.save（box）；
    //在本session中，box对象为自由状态
    tx.commit（）；
    session.close（）；//session关闭，box对象为游离状态
    //重新创建session
    session=sessionFactory.openSession（）；
    tx=session.beginTransaction（）；
    //box对象为持久化状态，不用调用update，Hibernate会自
动同步对象到数据库
    box=（Box）session.load（Box.class，boxid）；
    box.setName（"持久化状态box"）；
    session.flush（）；
    tx.commit（）；
    session.close（）；
    box.setName（"游离状态box"）；//session关闭，box对
象为游离状态
    session=sessionFactory.openSession（）；//重新创
建session
    tx=session.beginTransaction（）；
    //必须显式调用update、save或者是saveOrUpdate
```

```
    //使用saveOrUpdate方法，需要设置主键的unsavedvalue
属性
    session.saveOrUpdate（box）;
    tx.commit（）;
    session.close（）;
    sessionFactory.close（）; //关闭sessionFactory
    }
    }
```

其他代码不变，运行测试类，数据库记录如下：

```
    22持久化状态box 100.0 45.0 24.3
    23游离状态box 100.0 45.0 24.3
```

源程序解读

（1）session的saveorUpdate方法是由Hibernate来判断被操作的对象究竟是一个已经持久化过的持久对象还是临时自由状态对象。这需要在对象映射文件的主键id中定义unsavedvalue属性，如果不显式定义，则默认为unsavedvalue=null。

（2）unsavedvalue可以是下面的几个选项。

null：主键是对象类型，Hibernate判断操作对象的主键是否为null，来判断断操作对象是否已被持久化，如果是，调用save方法，生成insert语句，在数据库中增加一条记录；如果不是，设置主键则直接生成update的SQL语句，发送update，如果数据库中没有那条记录，则抛出异常。

none：由于不论主键属性为任何值，都不可能为none，因此Hibernate总是对被操作对象发送update。

any：由于不论主键属性为任何值，都肯定为any，因此Hibernate总是对project对象发送save，Hibernate生成主键。

（3）显式的使用session.save（）或者session.update（）操作一个对象时，实际上是用不到unsavedvalue的。

（4）在一个session中，持久化对象的变化，不需要调用update等显式语句，由flush方法就可以实现数据库表的更新。

（5）不同session之间的对象，也就是游离状态的对象，必须使用update显式更新数据库表。

（6）修改一个对象，最好的方法是使用session的load方法进行持久化，然后使用set方法实现属性的修改，一般情况下，不要使用setId方法来修改对象的主键值，避免产生意想不到的错误。

（7）使用Hibernate的id generator来生成无业务意义的主键，不使用有业务含义的字段做主键，不建议使用assigned。

（8）使用对象类型（String/Integer/Long/……）来做主键，而不使用基础类型（int/long/

……）做主键。

13.4　实体对象的识别

Hibernate将数据库数据同Java实体对象关联在一起，使开发者以操作对象的方式实现数据库的访问。相对于数据库，比较两个记录是否相同，一般比较其主键值是否相同就可以了，那么Hibernate持久化的实体对象如何比较呢？

技术要点

Hibernate让程序员通过操作Java对象的方式来操作数据库，那么如何比较两个Java实体对象？也就是说，如何通过Java实体对象的比较来比较两条数据记录。这在数据库应用项目中经常遇到，是一个基础问题。本节示例代码介绍如何对Hibernate持久化对象进行识别。

Java的对象比较方式。

如何结合业务逻辑进行持久化对象识别。

实现代码

本节的代码在前一节代码的基础上做一些小的改动，增加了一个测试类：DataIdentity.java，代码如下：

```
    package helloworld.session.test;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.cfg.Configuration;
    public class DataIdentity {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（） .configure（）；
    //根据Configuration建立SessionFactory
    //SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）；
    //建立Session，相当于建立JDBC的Connection
    Session session=sessionFactory.openSession（）；
    //同一个session中建立2个实体对象
    Box box1=（Box）session.load（Box.class, new
Integer（1））；
```

```
    Box box2=（Box）session.load（Box.class, new
Integer（1））;
    System.out.println（box1==box2）;
    System.out.println（box1.equals（box2））;
    session.close（）;
    session=sessionFactory.openSession（）; //跨
session的实体对象比较
    box2=（Box）session.load（Box.class, new
Integer（1））;
    System.out.println（box1==box2）;
    System.out.println（box1.equals（box2））;
    session.close（）;
    sessionFactory.close（）; //关闭sessionFactory
    }
    }
```

运行该测试类，结果如下：

```
    true
    true
    false
    false
```

接下来修改Box.java代码，增加equals（）、hasCode（）两个方法，代码如下：

```
    package helloworld.session.test;
    public class Box implements
java.io.Serializable {
    private Integer id; //主键ID属性变量
```

```
    private Float width; //宽度width属性变量
    private Float length; //长度length属性变量
    private Float height; //高度height属性变量
    private String name; //名称name属性变量
    public Box（） {//构造器
    }
    //getter和setter方法
    public Integer getId（） {
    return this.id;
    }
    public void setId（Integer id） {
    this.id=id;
    }
    ……
    public boolean equals（Object o） {//重写equals方法，可以根据业务需要自行修改比较逻辑
    if（this==o）return true;
    if（id==null｜｜！（o instanceof Box））return false;
    final Box box=（Box）o;
    return this.id.equals（box.getId（））;
    }
    public int hasCode（） {//重写hasCode（）方法
    return id==null? System.identityHashCode（this）: id.hashCode（）;
    }
    }
```

修改后，重新运行DataIdentity类，结果如下：

```
    true
    true
    false
    true
```

源程序解读

（1）对数据库而言，其识别一笔数据唯一性的方式是根据主键值，如果手上有两份数据，它们拥有同样的主键值，则它们在数据库中代表同一个字段的数据。

（2）对Java而言，要识别两个对象是否为同一个对象有两种方式，一种是根据对象是否拥有同样的内存位置来决定，在Java语法中就是透过==运算来比较，一种是根据equals（）、hasCode（）中的定义来比较。

（3）Object类中已经定义了equals（）、hasCode（）方法，所有类都可以根据需要重载这些方法。

（4）在同一个session中，box1和box2对应用一个Java对象的引用，所以比较结果为true。

（5）当第一个session关闭之后，重新建立session，这时重新给box2赋值，实际上box1和box2已经不是同一个对象引用了。虽然从逻辑上，box1和box2是完全相同的，但是打印却是false。

（6）如果要根据持久化对象的主键值作为对象识别的依据，在同一个session是正确的，当跨session时，就会出现问题。这就需要开发者重载Box类中的equals（）、hasCode（）方法，这样在比较2个持久化对象时就会给出正确结果。

（7）通常情况下，Java程序不推荐使用==来比较2个对象，而是使用equals（）方法。

（8）重载equals（）、hasCode（）方法，对一些业务逻辑来说非常有用，例如，本示例中的box对象，只要id相同，就认为是相同的。但是如果每个box的name值都是唯一的，我们也可以在equals（）、hasCode（）方法中使用name值来判断2个box是否相等。

13.5 Hibernate一级缓存

缓存（Cache）是计算机领域非常通用的概念。它介于应用程序和永久性数据存储源（如硬盘上的文件或者数据库）之间，其作用是降低应用程序直接读写永久性数据存储源的频率，从而提高应用的运行性能。缓存中的数据是数据存储源中数据的副本，应用程序在运行时直接读写缓存中的数据，只在某些特定时刻按照缓存中的数据来同步更新数据存储源。

技术要点

正确使用缓存，可以大幅度提高系统的性能，但是错误使用缓存，反而会造成意想不到的结果，所以，理解Hibernate的缓存机制非常重要。Hibernate的缓存按照作用范围可以分为一级缓存和二级缓存，本节介绍一级缓存。

Hibernate的一级缓存是由Session提供的，因此它只存在于Session的生命周期中，当程序调用save（）、update（）、saveorupdate（）等方法，及调用查询接口list、filter、iterate时，如session缓存中还不存在相应的对象，Hibernate会把该对象加入到一级缓存中，当Session关闭时，该Session所管理的一级缓存也会立即被清除。

注意：Hibernate的一级缓存是Session所内置的，不能被卸载，也不能进行任何配置。

一级缓存采用的是keyvalue的Map方式来实现的，在缓存实体对象时，对象的主关键字ID是Map的key，实体对象就是对应的值。所以说，一级缓存是以实体对象为单位进行存储的，在访问时使用的是主关键字ID，虽然，Hibernate对一级缓存使用的是自动维护的功能，没有提供任何配置功能，但是可以通过Session

中所提供的方法来对一级缓存的管理进行手工干预。本节代码演示Hibernate如下知识点：

使用Hibernate一级缓存。

了解Session使用缓存的常见方法。

理解Session的load方法和get方法的区别。

使用Log4j日志查看SQL执行情况。

实现代码

为了查看Hibernate如何访问数据库表，需要使用Log4j日志，读者可以将Hibernate安装包中project\etc目录下的log4j.properties属性文件复制到项目路径中，默认的log4j.properties文件会造成大量的调试信息，这里对该属性文件进行了修改，只显示SQL访

问信息，便于查看。log4j.properties文件内容代码如下：

```
    ###定义标准输出###
    log4j.appender.stdout=org.apache.log4j.ConsoleA
ppender
    log4j.appender.stdout.Target=System.out
    log4j.appender.stdout.layout=org.apache.log4j.P
atternLayout
    log4j.appender.stdout.layout.ConversionPattern=
%d {ABSOLUTE} %5p%c {1} : %L%m%n
    ###定义日志文件hibernate.log###
    #log4j.appender.file=org.apache.log4j.FileAppen
der
    #log4j.appender.file.File=hibernate.log
    #log4j.appender.file.layout=org.apache.log4j.Pa
tternLayout
    #log4j.appender.file.layout.ConversionPattern=%
d {ABSOLUTE} %5p%c {1} : %L%m%n
    ###设置日志优先级###
    #log4j.rootLogger=warn, stdout
    log4j.rootLogger=debug, stdout
    #log4j.logger.org.hibernate=info
    log4j.logger.org.hibernate=warn
    ###log HQL query parser activity
    #log4j.logger.org.hibernate.hql.ast.AST=debug
    ###log just the SQL显示SQL信息
    log4j.logger.org.hibernate.SQL=debug
    ###log JDBC bind parameters###
    #log4j.logger.org.hibernate.type=info
    log4j.logger.org.hibernate.type=warn
    ###log schema export/update###
    log4j.logger.org.hibernate.tool.hbm2ddl=warn
    ###log HQL parse trees
```

```
log4j.logger.org.hibernate.hql=debug
###log cache activity###显示Hibernate缓存信息
log4j.logger.org.hibernate.cache=debug
###log transaction activity
log4j.logger.org.hibernate.transaction=warn
###log JDBC resource acquisition
log4j.logger.org.hibernate.jdbc=warn
```

注意：Log4j的类库必须在CLASSPATH路径或者是项目的lib路径下，才能正常运行，默认的Hibernate类库已经包含了Log4j类库。

增加1个Hibernate一级缓存的测试类：SessionCache.java，内容如下：

```
package helloworld.session.test;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
public class SessionCache {
public static void main（String[]args） {
//Configuration管理Hibernate配置
Configuration config=new
Configuration（）.configure（）;
//根据Configuration建立SessionFactory
//SessionFactory用来建立Session
SessionFactory
sessionFactory=config.buildSessionFactory（）;
//建立session，相当于建立JDBC的Connection
```

```
    Session session=sessionFactory.openSession（）；
    //同一个session中建立2个实体对象
    Box box1=（Box）session.get（Box.class, new
Integer（1））；
    //box2使用了session缓存，并重新没有访问数据库
    Box box2=（Box）session.get（Box.class, new
Integer（1））；
    session.close（）；//session关闭，session缓存随即清
除
    System.out.println（"第一个session关闭"）；
    session=sessionFactory.openSession（）；//重新建
立session
    System.out.println（"创建第二个session"）；
    box1=（Box）session.get（Box.class, new
Integer（1））；//重新访问数据库
    session.clear（）；//清除session缓存
    box2=（Box）session.get（Box.class, new
Integer（1））；
    session.close（）；
    sessionFactory.close（）；//关闭sessionFactory
    }
```

运行该示例，结果如下：

```
    14: 12: 42，734 DEBUG SQL: 346select box0__.id as
id0__0__，box0__.width as width0__0__，box0__.length
as
    length0__0__，box0__.height as height0__0__，box0
__.name as name0__0__from ssh.box box0__where box0
__.id=?
    第一个session关闭
    创建第二个session
    14: 12: 42，796 DEBUG SQL: 346select box0__.id as
id0__0__，box0__.width as width0__0__，box0__.length
```

```
as
    length0__0__, box0__.height as height0__0__, box0
__.name as name0__0__from ssh.box box0__where box0
__.id=?
    14: 12: 42, 843 DEBUG SQL: 346select box0__.id as
id0__0__, box0__.width as width0__0__, box0__.length
as
    length0__0__, box0__.height as height0__0__, box0
__.name as name0__0__from ssh.box box0__where box0
__.id=?
```

上面测试代码获得对象使用了session的get方法，下面将get方法替换为load方法，测试类为SessionCacheLoad.java，内容如下：

```
    package helloworld.session.test;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.cfg.Configuration;
    public class SessionCacheLoad {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（） .configure（）;
    //根据Configuration建立SessionFactory
    //SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）;
    //建立session，相当于建立JDBC的Connection
    Session session=sessionFactory.openSession（）;
    //同一个session中建立2个实体对象
```

```
    Box box1=（Box）session.get（Box.class, new
Integer（1））;
    System.out.println（"第一个box"）;
    //box2使用了session缓存，并重新没有访问数据库
    Box box2=（Box）session.get（Box.class, new
Integer（1））;
    System.out.println（"第二个box"）;
    session.close（）; //session关闭，session缓存随即清
除
    session=sessionFactory.openSession（）;
    System.out.println（"创建第二个session"）;
    box1=（Box）session.load（Box.class, new
Integer（1））; //重新访问数据库
    System.out.println（"第一个box"）;
    box1.getName（）;
    box2=（Box）session.load（Box.class, new
Integer（1））;
    System.out.println（"第二个box"）;
    box2.getName（）;
    session.close（）;
    sessionFactory.close（）; //关闭sessionFactory
    }
    }
```

运行该测试程序，结果如下：

```
    14: 25: 04, 750 DEBUG SQL: 346select box0__.id as
id0__0__, box0__.width as width0__0__, box0__.length
as
    length0__0__, box0__.height as height0__0__, box0
__.name as name0__0__from ssh.box box0__where box0
__.id=?
    第一个box
    第二个box
```

```
    创建第二个session
    第一个box
    14: 25: 04, 906 DEBUG SQL: 346select box0__.id as
id0__0__, box0__.width as width0__0__, box0__.length
as length0__0__, box0__.height as height0__0__, box0
__.name as name0__0__from ssh.box box0__where box0
__.id=?
    第二个box
```

源程序解读

（1）为了查看Hibernate对数据库的访问情况，使用了Log4j日志，log4j.properties属性文件中，设置log4j.logger.org.hibernate.SQL=debug，即可输出SQL语句信息，本例中，同时会输出hql和cache信息，但是本例并没有配置Hibernate的二级缓存，所以不会有cache信息输出。

（2）第一个测试程序中的第一个session中，有2个ID都是1的box对象，从控制台的日志可以看到，这个session只访问一次数据库。第一个box对象的数据是从数据库中查询而来的，然后该对象就保存在

session的缓存中，当第二个box对象调用get方法时，首先检查session缓存中是否有ID为1的对象，如果有，就从缓存中获得数据，并不会再次访问数据库。

（3）第二个session中，不同的是加入了clear方法，即将session缓存数据清空，这样在创建第二个box对象时，由于缓存中找不到该对象，则会再次使用SQL通过JDBC查询数据库获得数据。

（4）第二个测试程序帮助我们来区分load和get方法之间的区别。

（5）使用get方法获得持久化对象时，首先查找session缓存（Hibernate一级缓存）是否有该对象，如果有，则获得该对象；如果没有，就会访问数据库，如果数据库中找不到数据，则返回null。

（6）load方法也是获得数据，但是不同的地方是load方法已经假定数据库中一定存在该数据的，如果在数据库中找不到该数据，则会抛出一个org.hibernate.ObjectNotFoundException异常。

（7）load方法获得对象的过程：load方法首先在session缓存中查找对象，如果找不到则查找sessionfactory缓存（Hibernate二级缓存），如果再找不到则访问数据库。

（8）值得注意的是，load方法是假定数据库中一定有该数据的，所以使用代理来延迟加载对象，只有在程序中使用到了该对象的属性（非主键属性）时，Hibernate才会进入load方法的获得对象过程。所以说，如果数据库中不存在该记录，异常是在程序访问该对象属性时抛出的，而不是在创建这个对象时就抛出。

（9）第二个程序中的第二个session中可以看到，只有在访问box对象的name属性时才执行数据库查询的，而不是在创建box1时执行。

13.6 Hibernate二级缓存

二级缓存是SessionFactory级别的全局缓存，这在Hibernate应用中非常有益于提高数据库访问效率，但是值得注意的是只有用好了二级缓存才有帮助，不然会导致非常多的问题，例如死锁、数据同步等。技术要点Hibernate二级缓存比较重要，程序员必须理解Hibernate二级缓存的机制和使用方法，这对提高Hibernate应用性能非常重要。Hibernate的二级缓存使用技巧性比较强，使用不当往往造成性能的严重下降。本节代码试图对Hibernate的二级缓存进行介绍，二级缓存是Hibernate的重点内容，读者需要深入掌握二级缓存的机制和使用，本节代码涉及下面几个知识点：

配置Hibernate二级缓存。

EhCache的简单配置。几种缓存策略的区别。对二级缓存使用的建议。

实现代码

本节代码是在前一个示例代码基础上实现的，增加了二级缓存的相关配置和一个测试类。

配置Hibernate二级缓存我们使用了EhCache，需要将Hibernate开发包lib\optional\ehcache目录中的ehcache1.2.3.jar类库复制到CLASSPATH或者是项目的lib目录下。

另外需要EhCache的配置文件，可以在Hibernate开发包的project\etc路径下找到，文件名为ehcache.xml，将该文件复制到项目路径下。修改Hibernate配置文件，增加hibernate.

cache. provider__class属性，修改后的hibernate.cfg.xml内容如下：

```
    <? xml version='1.0'encoding='UTF8'? >
    <! DOCTYPE hibernateconfiguration
PUBLIC"//Hibernate/Hibernate Configuration DTD
3.0//EN""http: //hibernate.
sourceforge.net/hibernateconfiguration3.0.dtd">
    <hibernateconfiguration>
    <sessionfactory><! 数据库用户名>
    <property name="connection.username">root
    </property>
    <! 数据库连接URL>
    <property name="connection.url">jdbc:
mysql: //localhost: 3306/ssh
    </property>
    <property name="dialect">org.
hibernate.dialect.MySQLDialect
    </property>
    <property name="myeclipse.connection.profile">
Mysql
    </property>
    <property name="hbm2ddl.auto">update
    </property>
    <property name="connection.password">pla
    </property>
    <! 数据库密码>
    <property name="connection.driver__class">
    <! 数据库驱动>com. mysql.jdbc.Driver
    </property>
    <! 配置二级缓存>
    <property name="hibernate.cache.provider__
class">org.hibernate.cache.EhCacheProvider
    </property>
```

```
    <！数据库表Java对象映射文件表>
    <mapping
resource="h1/session/test/Box.hbm.xml"/>
    </sessionfactory>
    </hibernateconfiguration>
```

注意：本书使用了Hibernate的3.3.0.GA版本，不同版本的hibernate.cache.provider__class会有差异。

本例将ehcache.xml文件内容做了修改，并加入了注释，方便读者理解配置信息，内容如下：

```
    <ehcache>
    <！设置缓存文件cache.data存放位置>
    <diskStore path="java.io.tmpdir"/>
    <！设置缓存配置信息
    maxElementsInMemory：缓存中最大允许创建的对象数
    eternal：缓存中对象是否为永久的，如果是，超时设置将被忽略，对象从不过期
    timeToIdleSeconds：缓存数据钝化时间（设置对象在它过期之前的空闲时间）
    timeToLiveSeconds：缓存数据的生存时间（设置对象在它过期之前的生存时间）
    overflowToDisk：内存不足时，是否启用磁盘缓存
    >
    <defaultCache maxElementsInMemory="10000"
    eternal="false"
    timeToIdleSeconds="120"
    timeToLiveSeconds="120"
```

```
    overflowToDisk="true"/>
    </ehcache>
```

对box.hbm.xml进行修改，增加二级缓存相关设置，内容如下：

```
    <? xml version="1.0"encoding="utf8"? >
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemapp
ing3.0.dtd">
    <hibernatemapping>
    <class
name="helloworld.session.test.Box"table="box"catalo
g="ssh">
    <! 设置缓存为只读属性>
    <cache usage="readonly"/>
    <id
name="id"type="java.lang.Integer"unsavedvalue="null
">
    <column name="id"></column>
    <generator class="native"></generator>
    </id>
    <property name="width"type="java.lang.Float">
<! 映射宽度属性变量>
    <column name="width"precision="12"scale="0"/>
    </property>
    <property name="length"type="java.lang.Float">
<! 映射长度属性变量>
    <column name="length"precision="12"scale="0"/>
    </property>
    <property name="height"type="java.lang.Float">
<! 映射高度属性变量>
```

```
    <column name="height"precision="12"scale="0"/>
    </property>
    <property name="name"type="java.lang.String">
<! 映射名称属性变量>
    <column name="name"length="20"/>
    </property>
    </class>
    </hibernatemapping>
```

再增加1个二级缓存测试类：

SessionFactoryCache.java，内容如下：

```
    package helloworld.session.test;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.cfg.Configuration;
    public class SessionFactoryCache {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（）.configure（）;
    //根据Configuration建立SessionFactory
    //SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）;
    //建立session，相当于建立JDBC的Connection
    Session session=sessionFactory.openSession（）;
    //访问数据库，并将对象存入二级缓存
    Box box1=（Box）session.get（Box.class, new
Integer（1））;
    //box2使用了session缓存
    Box box2=（Box）session.get（Box.class, new
Integer（1））;
```

```
    session.close（）；//session关闭，session缓存随即清
除
    System.out.println（"第一个session关闭"）；
    session=sessionFactory.openSession（）；//重建
session
    System.out.println（"创建第二个session"）；
    //一级缓存中没有，查找二级缓存
    box1=（Box）session.get（Box.class, new
Integer（1））；
    session.close（）；
    sessionFactory.close（）；//关闭sessionFactory
    }
    }
```

运行该测试程序，输出结果如下，为了方便说明运行机制，下面对日志信息进行了注释，供参考：

```
    15: 49: 08，328 INFO ASTQueryTranslatorFactory:
24Using ASTQueryTranslatorFactory
    #建立缓存管理器
    15: 49: 08，421 DEBUG CacheManager: 191Creating
new CacheManager with default
    config
    15: 49: 08，437 DEBUG CacheManager:
164Configuring ehcache from classpath.
    #配置文件
    15: 49: 08，437 DEBUG Configurator:
121Configuring ehcache from ehcache.xml found in
the classpath:
    file: /D: /SSHcode/hibernate/bin/ehcache.xml
    #设置磁盘缓存文件位置
    15: 49: 08，453 DEBUG ConfigurationDiskStore:
185Disk Store Path: C: ＼DOCUME～1＼NSFC＼LOCALS
```

～1＼Temp＼
#初始化Box缓存为只读
15: 49: 08，500 DEBUG CacheFactory: 39instantiating cache region: helloworld.
session.test.Box usage strategy: readonly
15: 49: 08，500 WARN CacheFactory: 43readonly cache configured for mutable class: hello
world.session.test.Box

15: 49: 08，500 WARN EhCacheProvider: 103Could not find configuration[helloworld.sess
ion.test.Box]; using defaults.
#随着sessionfactory的创建初始化二级缓存
15: 49: 08，515 DEBUG DiskStore: 194Deleting data file helloworld.session.test.Box.
data
15: 49: 08，578 DEBUG MemoryStore: 147helloworld.session.test.Box Cache: Using Spooling LinkedHash
Map implementation
15: 49: 08，578 DEBUG MemoryStore: 128initialized MemoryStore for helloworld.session.
test.Box
15: 49: 08，578 DEBUG Cache: 277Initialised cache: helloworld.session.test.Box
15: 49: 08，578 DEBUG EhCacheProvider: 106started EHCache region: helloworld.session.
test.Box
15: 49: 09，578 DEBUG EhCache: 104key: helloworld.session.test.Box#1
#get方法在缓存中查找ID=1的对象，找不到
15: 49: 09，578 DEBUG MemoryStore: 204helloworld.session.test.BoxCache: MemoryStore miss for hel
loworld.session.test.Box#1

15: 49: 09，578 DEBUG Cache:
370helloworld.session.test.Box cacheMiss
15: 49: 09，578 DEBUG EhCache: 113Element for
helloworld.session.test.Box#1 is null
#Hibernate访问数据库，获得数据
15: 49: 09，578 DEBUG SQL: 346select box0__.id as
id0__0__，box0__.width as width0__0__，box0__.length
as
length0__0__，box0__.height as height0__0__，box0
__.name as name0__0__from ssh.box box0__where box0
__.id=?
#缓存中存入对象box1
15: 49: 09，703 DEBUG ReadOnlyCache: 58Caching:
helloworld.session.test.Box#1
15: 49: 09，703 DEBUG Cache:
878helloworld.session.test.Box#1 now: 1219996149703
15: 49: 09，703 DEBUG Cache:
879helloworld.session.test.Box#1 Creation Time:
1219996149703 Next To
Last Access Time: 0
15: 49: 09，703 DEBUG Cache:
881helloworld.session.test.Box#1 mostRecentTime:
1219996149703
15: 49: 09，703 DEBUG Cache:
882helloworld.session.test.Box#1 Age to Idle:
120000 Age Idled: 0
15: 49: 09，703 DEBUG Cache:
906helloworld.session.test.Box: Is element with key
helloworld.session.test.Box#1 expired? : false
#box2获得一级缓存对象，并没有查找二级缓存
第一个session关闭
创建第二个session
#第二个session中box1、box2查找二级缓存，获得数据，不会
访问数据库
15: 49: 09，703 DEBUG EhCache: 104key:
helloworld.session.test.Box#1

```
    15: 49: 09, 859 DEBUG MemoryStore:
201helloworld.session.test.BoxCache: MemoryStore
hit for
    helloworld.session.test.Box#1
    15: 49: 09, 859 DEBUG Cache:
878helloworld.session.test.Box#1 now: 1219996149859
    15: 49: 09, 859 DEBUG Cache:
879helloworld.session.test.Box#1 Creation Time:
121999614
    9703 Next To Last Access Time: 0
    15: 49: 09, 859 DEBUG Cache:
881helloworld.session.test.Box#1 mostRecentTime:
    1219996149703
    15: 49: 09, 875 DEBUG Cache:
882helloworld.session.test.Box#1 Age to Idle:
120000 Age Idled: 156
    15: 49: 09, 875 DEBUG Cache:
906helloworld.session.test.Box: Is element with key
    helloworld.session.test.Box#1 expired? : false
    15: 49: 09, 875 DEBUG ReadOnlyCache: 34Cache
hit: helloworld.session.test.Box#1
    15: 49: 09, 875 DEBUG CacheManager: 196Attempting
to create an existing instance.Existing instance re
    turned.
    #关闭sessionfactory后，清理二级缓存
    15: 49: 09, 875 DEBUG DiskStore: 444Deleting file
helloworld.session.test.Box.data
    15: 49: 09, 875 DEBUG DiskStore:
649helloworld.session.test.BoxCache: Expiry thread
interrupted on Disk
    Store.
```

源程序解读

（1）Hibernate通过hibernate.cache.provider__class属性中指定org.hibernate.cache.CacheProvider的某个实现的类名，开发者可以选择让Hibernate使用哪个缓存实现。Hibernate打包一些开源缓存实现，提供对它们的内置支持（表13.1）。除此之外，也可以使用自己定义的类，将它们插入到系统中。

注意：从3.2版本开始，不再默认使用EhCache作为缓存实现。

表 13.1　缓存策略提供商(Cache Providers)

Cache	Provider class	Type	Cluster Safe	Query
				Cache Supported
Hashtable (not intended for production use)	org.hibernate.cache.HashtableCacheProvider	memory		yes
EHCache	org.hibernate.cache.EhCacheProvider	memory, disk		yes
OSCache	org.hibernate.cache.OSCacheProvider	memory, disk		yes
SwarmCache	org.hibernate.cache.SwarmCacheProvider	clustered (ip multicast)	yes (clustered invalidation)	
JBoss TreeCache	org.hibernate.cache.TreeCacheProvider	clustered (ip multicast), transactiona	yes (replication)	yes (clock sync req.)

在使用不同缓存策略时，需要用到相应的类库和配置文件，初学者一定需要注意这点。

（2）本实例使用了EhCache缓存配置策略，需要配置ehcache.xml文件，该文件是EhCache的配置文件，有下面几个重要的属性。

maxElementsInMemory：缓存中最大允许创建的对象数。

eternal：缓存中对象是否为永久的，如果是，超时设置将被忽略，对象从不过期。

timeToIdleSeconds：缓存数据钝化时间（设置对象在它过期之前的空闲时间）。

timeToLiveSeconds：缓存数据的生存时间（设置对象在它过期之前的生存时间）。

overflowToDisk：内存不足时，是否启用磁盘缓存。

（3）在配置完Hibernate后，还需要配置映射文件的cache元素，例如下面代码：

```
    <cache
    usage="transactional|readwrite|
nonstrictreadwrite|readonly"
    region="RegionName"
    include="all|nonlazy"
    />
```

属性说明如下。

usage（必需）：说明了缓存的策略transactional、readwrite、nonstrictreadwrite或readonly。

region：（可选，默认为类或者集合的名字（class or collection role name））指定第二级缓

存的区域名（name of the second level cache region）。

include：（可选，默认为all）nonlazy当属性级延迟抓取打开时，标记为lazy=“true”的实体的属性可能无法被缓存。

（4）usage说明了该映射的缓存策略。

只读缓存（Strategy：read only）：如果你的应用程序只需读取一个持久化类的实例，而无需对其修改，那么就可以对其进行只读缓存。这是最简单，也是实用性最好的方法。甚至在集群中，它也能完美地运作。

```
<class name="eg.Immutable"mutable="false">
<cache usage="readonly”/>
……
</class>
```

读/写缓存（Strategy：read/write）：如果应用程序需要更新数据，那么使用读/写缓存比较合适。如果应用程序要求“序列化事务”的隔离级别（serializable transaction isolationlevel），那么就决不能使用这种缓存策略。如果在JTA环境中使用缓存，必须指定hibernate.transaction.manager__lookup__class属性的值，通过它，Hibernate才能知道该应用程序中JTA的TransactionManager的具体策略。在其他环境中，必须保证在Session.close（）或Session.disconnect（）调用前，整个事务已经结束。如果你想在集群环境中使用此策略，必须保证底层的缓存实现支持锁定（locking）。Hibernate内置的缓存策略并不支持锁定功能。

```
<class name="eg.Cat"……>
<cache usage="readwrite"/>
……
<set name="kittens"……>
<cache usage="readwrite"/>
……
</set>
```

</class>

非严格读/写缓存（Strategy：nonstrict read/write）：如果应用程序只偶尔需要更新数据（也就是说，两个事务同时更新同一记录的情况很不常见），也不需要十分严格的事务隔离，那么比较适合使用非严格读/写缓存策略。如果在JTA环境中使用该策略，必须为其指定hibernate.transaction.manager__lookup__class属性的值，在其他环境中，必须保证在Session.close（）或Session.disconnect（）调用前，整个事务已经结束。

事务缓存（transactional）：Hibernate的事务缓存策略提供了全事务的缓存支持，例如对JBoss TreeCache的支持。这样的缓存只能用于JTA环境中，必须指定为hibernate.transaction.manager__lookup__class属性。

（5）没有一种缓存提供商能够支持上列的所有缓存并发策略。表13.2列出了各提供商及其各自适用的并发策略。

表 13.2　各缓存提供商对缓存并发策略的支持情况

Cache	read-only	nonstrict-read-write	read-write	transactional
Hashtable（not intended for production use）	yes	yes	yes	
EHCache	yes	yes	yes	
OSCache	yes	yes	yes	
SwarmCache	yes	yes		
JBoss TreeCache	yes			yes

（6）本节实例中，使用了3次load或者get来装载Box对象，并且是跨session的，但是从Log4j的日志输出可以看到，实际上只是访问了1次数据库。

（7）第一个session中，box2对象是从一级缓存中找到的；第二个session中，对象是从二级缓存找到的。因为跨session应用，一级缓存已经清空了。

（8）从日志的输出可以看到，sessionfactory创建时，系统建立了二级缓存的磁盘文件和内存模块；第一次查询box对象时，将数据库的返回结果保持在二

级缓存中；而sessionfactory关闭时，缓存文件删除，占用内存清空。

（9）Hibernate二级缓存可以提升系统的数据访问性能，但是前提是合理使用，并不是所有的数据都可以使用二级缓存。

13.7 Hibernate事务处理

Hibernate是对JDBC的轻量级对象封装，Hibernate本身是不具备Transaction处理功能的，Hibernate的Transaction实际上是底层的JDBC Transaction的封装，或者是JTA Transaction的封装。

技术要点

事务是数据库应用开发的核心部分，对事务的掌握和理解往往是衡量一个程序员水平高低的标准之一，也往往是初级程序员所忽略的部分。例如一个复杂的金融业务操作，涉及到多个数据的改变，如果在处理过程中出现异常情况，必须保证数据的完整性，这对数据库应用尤为重要。Hibernate提供2种数据库事务支持，本节介绍Hibernate的事务处理机制。

Hibernate如何支持JDBC事务。

Hibernate如何支持JTA事务。

Hibernate如何封装JDBC事务。

实现代码

在前面代码的基础上，建立一个Hibernate事务测试类HibernateTx.java，内容如下：

```
    package helloworld.session.test;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.Transaction;
    import org.hibernate.cfg.Configuration;
    public class HibernateTx {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（）.configure（）;
    //根据Configuration建立SessionFactory
    //SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）;
    Box box=new Box（）; //建立box对象
    box.setHeight（24.3f）;
    box.setLength（100.00f）;
    box.setWidth（45.00f）;
```

```
    box.setName（"Hello Box! "）;
    //建立Session，相当于建立JDBC的Connection
    Session session=sessionFactory.openSession（）;
    //保存对象，默认Hibernate的AutoCommit是关闭的，不会自动提交数据
    session.save（box）;
    session.close（）;
    session=sessionFactory.openSession（）;
    Transaction tx=session.beginTransaction（）; //开始一个事务
    session.save（box）; //保存对象
    tx.commit（）; //提交事务，数据才会提交
    session.close（）;
    sessionFactory.close（）; //关闭sessionFactory
    }
    }
```

运行该测试类，查看数据库表，结果发现只增加了一条记录。

源程序解读

（1）Hibernate可以配置为JDBCTransaction或者是JTATransaction，这取决于你在hibernate.properties中的配置：

```
    #hibernate.transaction.factory__class net.sf.hibernate.transaction.JTATransa
```

```
    ctionFactory
    #hibernate.transaction.factory__class
net.sf.hibernate.transaction.JDBCTrans
    actionFactory
```

如果不做任何配置，缺省情况是使用JDBCTransaction。你可以在xml文件中配置该属性。

（2）简单介绍一下JTA。JTA（Java Transaction API）为J2EE平台提供了分布式事务服务。要用JTA进行事务界定，应用程序要调用javax.transaction.UserTransaction接口中的方法。例如：

```
    utx.begin（）; //开始事务
    ……
    DataSource ds=obtainXADataSource（）;
    Connection conn=ds.getConnection（）;
    pstmt=conn.prepareStatement（"UPDATE
MOVIES……"）;
    pstmt.setString（1, "Spinal Tap"）;
    pstmt.executeUpdate（）;
    ……
    utx.commit（）; //提交事务
```

Java事务API（JTA）及其同门兄弟Java事务服务（Java Transaction Service JTS）为J2EE平台提供了分布式事务服务。一个分布式的事务涉及一个事务管理器和一个或者多个资源管理器。一个资源管理器是任何类型的持久性的数据存储。事务管理器负责协调所有事务参与者之间的通信。

与本地事务相比，XA协议的系统开销相当大，因而应当慎重考虑是否确实需要分布式事务。只有支持XA协议的资源才能参与分布式事务。如果事务须登记一个以上的资源，则需要实现和配置所涉及的资源（适配器、JMS或JDBC连接池）以支持XA。

（3）缺省情况下，Hibernate的JDBC事务的AutoCommit是关闭的，必须手动提交才能够真正完成一个事务。本实例的第一个save就是这样的例子。

（4）看看使用JDBC Transaction时的代码例子：

```
session=sessionFactory.openSession（）；
//开始一个事务
Transaction tx=session.beginTransaction（）；
session.save（box）；//保存对象
tx.commit（）；//提交事务，数据才会提交
session.close（）；
```

这是默认的情况，当在代码中使用Hibernate的Transaction时实际上就是JDBCTransaction。那么JDBCTransaction究竟是什么东西呢？来看看源代码就清楚了。

Hibernate源代码中的类net.sf.hibernate.transaction.JDBCTransaction：

```
public void begin（）throws HibernateException {
……
if（toggleAutoCommit）
session.connection（）.setAutoCommit（false）；
……
}
```

在源代码中我们看到了熟悉的JDBC语法，connection（）.setAutoCommit（false）这句将JDBC

连接的自动提交关闭了。下面是commit方法的源代码：

```
    public void commit（）throws HibernateException {
    ……
    try {
    if（session.getFlushMode（）！=FlushMode.NEVER）
session.flush（）;
    try {
    session.connection（）.commit（）;
    committed=true;
    }
    ……
    toggleAutoCommit（）;
    }
```

（5）Hibernate实际上只是对JDBC的简单封装，表13.3所示显示了事务处理的对比。

表 13.3　Hibernate 和 JDBC 事务处理对比

JDBC	Hibernate
Connection conn	session = sf. openSession( )
conn. setAutoCommit(false)	tx = session. beginTransactioin( )
conn. commit( ) ; conn. setAutoCommit(true) ;	tx. commit( ) ;
conn. close( ) ;	session. close( ) ;

（6）如果在EJB中使用Hibernate，是最简单不过的了，什么Transaction代码都不用写，只需直接在

EJB的部署描述符上配置该方法是否使用事务即可。

13.8 使用复合主键

前面的数据库表都是单一主键，本节介绍对于复合主键的表Hibernate如何处理。一般情况下，Hibernate有两种方式处理复合主键。

技术要点

数据库表可以定义复合主键，记录通过多个键值组合来确定其唯一性，例如可以使用楼编号、层编号、房间编号来确定一个房间的位置，这是一个简单的复合主键示例。复合主键在数据库应用中使用频繁，是程序员必须掌握的内容。对于复合主键，Hibernate有两种方式来处理，第一种是基于实体类属性的复合主键，另一种是基于主键类的复合主键。本节示例使用基于主键类的复合主键。

创建主键类。

配置复合主键。

复合主键对象的CRUD。

实现代码

建立一个表名为dog的表，其中id和dogid为复合主键，建表语句如下：

```
CREATE TABLE'dog'（
′ id'int（11）NOT NULL，
′ dog__id'int（11）NOT NULL，
′ name'varchar（20）default NULL，
′ color'varchar（10）default NULL，
PRIMARY KEY（′ id'，′ dog__id'）
）ENGINE=InnoDB DEFAULT CHARSET=gbk
```

建立一个主键类DogId，代码如下：

```
package helloworld.compositeid;
public class DogId implements
java.io.Serializable {
private Integer id; //主键ID属性变量
```

```
    private Integer dogId; //Dogid属性变量
    public DogId（） {//构造器
    }
    public Integer getId（） {//ID属性的Getter（）方法
和Setter（）方法
    return this.id;
    }
    public void setId（Integer id） {
    this.id=id;
    }
    public Integer getDogId（） {//dogid属性的
Getter（）方法和Setter（）方法
    return this.dogId;
    }
    public void setDogId（Integer dogId） {
    this.dogId=dogId;
    }
    public boolean equals（Object other） {//重写
equals（）方法
    if（（this==other））//比较条件
    return true;
    if（（other==null））
    return false;
    if（！（other instanceof DogId））
    return false;
    DogId castOther=（DogId）other;
    //根据业务需要，设定比较条件
    return（（this.getId（）==castOther.getId（））|
|（this.getId（）！=null
    &&castOther.getId（）！=null&&
this.getId（）.equals（
    castOther.getId（））））
    &&（（this.getDogId（）
==castOther.getDogId（））||（this
    .getDogId（）！=null
    &&castOther.getDogId（）！=null&&
this.getDogId（）
```

```
    .equals（castOther.getDogId（））））；
    }
    public int hashCode（） {//重写hashCode（）方法
    int result=17;
    result=37result+（getId（）==null？0：
this.getId（）.hashCode（））；
    result=37result
    +（getDogId（）==null？0：
this.getDogId（）.hashCode（））；
    return result;
    }
    }
```

建立一个Dog类，内容如下：

```
    package helloworld.compositeid;
    public class Dog implements
java.io.Serializable {
    private DogId id; //主键ID属性变量
    private String name; //名称nameshxbl
    private String color; //颜色color属性变量
    public Dog（） {//构造器
    }
    public DogId getId（） {//ID属性的Getter（）方法和
Setter（）方法
    return this.id;
    }
    public void setId（DogId id） {
    this.id=id;
    }
    public String getName（） {//name属性的Getter（）
方法和Setter（）方法
    return this.name;
    }
```

```
    public void setName（String name） {
    this.name=name;
    }
    public String getColor（） {//color属性的
Getter（）方法和Setter（）方法
    return this.color;
    }
    public void setColor（String color） {
    this.color=color;
    }
    }
```

建立映射文件Dog.hbm.xml，并在hibernate.cfg.xml文件中加入该映射文件，Dog.hbm.xml内容如下：

```
    <? xml version="1.0"encoding="utf8"? >
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
    <class
name="helloworld.compositeid.Dog"table="dog"catalo
g="ssh">
    <! 定义复合主键>
    <compositeid
name="id"class="helloworld.compositeid.DogId">
    <keyproperty
name="id"type="java.lang.Integer">
    <column name="id"/>
    </keyproperty>
```

```
    <keyproperty
name="dogId"type="java.lang.Integer">
    <column name="dog__id"/>
    </keyproperty>
    </compositeid>
    <property name="name"type="java.lang.String">
<! 名称name映射>
    <column name="name"length="20"/>
    </property>
    <property name="color"type="java.lang.String">
<! 颜色color映射>
    <column name="color"length="10"/>
    </property>
    </class>
    </hibernatemapping>
```

编写一个测试类，实现基本的CRUD操作，

Test.java内容如下：

```
    package helloworld.compositeid;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.Transaction;
    import org.hibernate.cfg.Configuration;
    public class Test {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（）.configure（）;
    //根据Configuration建立SessionFactory
    //SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）;
```

```
    //增加记录
    DogId dogid=new DogId（）; //建立主键类实例
    dogid.setId（1）;
    dogid.setDogId（1）;
    Dog dog=new Dog（）; //建立Dog对象
    dog.setName（"Tom"）;
    dog.setColor（"yellow"）;
    dog.setId（dogid）;
    Session session=sessionFactory.openSession（）;
    Transaction tx=null;
    try {
    tx=session.beginTransaction（）;
    session.save（dog）;
    tx.commit（）;
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）;
    throw e;
    } finally {
    session.close（）;
    }
    session=sessionFactory.openSession（）; //查找数
据
    dog=（Dog）session.get（Dog.class, dogid）;
    System.out.println（dog.getId（） .getId（）
+""+dog.getId（） .getDogId（）
    +""+dog.getName（） ）;
    session.close（）;
    dog.setName（"Kitty"）; //修改数据
    session=sessionFactory.openSession（）;
    tx=null;
    try {
    tx=session.beginTransaction（）;
    session.update（dog）;
    tx.commit（）;
    } catch（RuntimeException e） {
    if（tx! =null）
```

```
    tx.rollback（）；
    throw e;
    } finally {
    session.close（）；
    }
    session=sessionFactory.openSession（）；//删除数
据
    tx=null;
    try {
    tx=session.beginTransaction（）；
    session.delete（dog）；
    tx.commit（）；
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）；
    throw e;
    } finally {
    session.close（）；
    }
    //关闭sessionFactory
    sessionFactory.close（）；
    }
    }
```

运行该测试类，结果如下：

```
    ……
    11: 32: 49, 562 DEBUG SQL: 346insert into
ssh.dog（name, color, id, dog__id）
values（? , ? , ? , ? ）
    11: 32: 49, 656 DEBUG SQL: 346select dog0__.id as
id1__0__, dog0__.dog__id as dog2__1__0__, dog0__.name
as name1
```

```
    __0__, dog0__.color as color1__0__from ssh.dog
dog0__where dog0__.id=? and dog0__.dog__id=?
    1 1 Tom
    11: 32: 49, 687 DEBUG SQL: 346update ssh.dog set
name=? , color=? where id=? and dog__id=?
    11: 32: 49, 750 DEBUG SQL: 346delete from
ssh.dog where id=? and dog__id=?
    ……
```

源程序解读

（1）本实例使用了单独的主键类DogId。

（2）主键类DogId重写了equals方法和hashCode方法，这一点非常重要。

（3）Hibernate要求主键类必须实现Serializable接口。

（4）复合主键的值是一个主键类，而不是一个普通的常见数值。

（5）复合主键的映射文件使用compositeid表示。

第14章　Hibernate集合映射

本章介绍Hibernate常见的集合映射中Set、List、Map和Bag等集合类型。本章结合简单的实例，让读者快速掌握Hibernate的集合映射。最简单的集合映射示例就是每个用户可以有多个电子邮件地址，那么对用户来说，电子邮件就是一个集合，则在用户的实体类中就可以通过定义一个集合类型的属性来表达。

14.1　Set集合映射

Set集合的特点是集合的元素不可以重复，例如用户表（user）和电子邮件表（email）之间的关系：每个用户只能拥有一个电子邮件地址，不能重复。关系如图14.1所示。

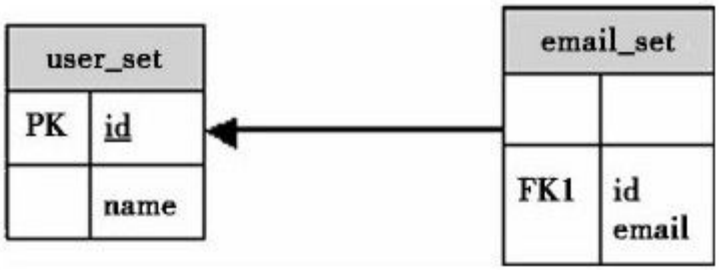

图　14.1　Set集合映射关系

技术要点

Set集合是Hibernate中基础的集合类型，元素数据一般使用外键同主表关联，Set集合非常适用于集合元素不能重复的情况。本节代码演示如何在Hibernate中建立Set集合映射。注意Set集合元素的特点是不能重复。

熟悉Set集合映射配置。

简单的对象操作。

实现代码

首先建立三个对应表格，语句如下：

```
CREATE TABLE'email__set'（
′ id'int（11）NOT NULL，
′ address'varchar（100）NOT NULL
）ENGINE=InnoDB DEFAULT CHARSET=gbk
CREATE TABLE'user__set'（
′ id'int（11）NOT NULL auto__increment，
′ name'varchar（100）NOT NULL default''，
PRIMARY KEY（′ id'）
）ENGINE=InnoDB DEFAULT CHARSET=gbk
```

建立角色的实体类Userset.java，代码如下：

```
package collect.set;
import java.util.HashSet;
import java.util.Set;
public class UserSet implements
java.io.Serializable {
private Integer id; //主键ID属性变量
private String name; //姓名name属性变量
private Set emails=new HashSet（）; //邮件集合
emails属性变量
public UserSet（） {//构造器
}
public Integer getId（） {//ID属性的Getter（）方法
和Setter（）方法
return this.id;
}
public void setId（Integer id） {
this.id=id;
}
public String getName（） {//name属性的Getter（）
方法和Setter（）方法
return this.name;
```

```
    }
    public void setName（String name） {
    this.name=name;
    }
    public Set getEmails（） {//emails属性的Getter（）
方法和Setter（）方法
    return emails;
    }
    public void setEmails（Set emails） {
    this.emails=emails;
    }
    }
```

建立Set类型的映射文件UserSet.hbm.xml，内容如下：

```
    <? xml version="1.0"encoding="utf8"? >
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
    <class name="collect.set.UserSet"table="user__
set"catalog="ssh">
    <id name="id"type="java.lang.Integer">
    <column name="id"/>
    <generator class="native"/>
    </id>
    <property name="name"type="java.lang.String">
    <column name="name"length="100"notnull="true"/
>
    </property>
    <! Set类型映射>
```

```
    <set name="emails"table="email__set"><! 定义对
应的表>
    <key column="id"></key><! 定义表的主键>
    <element type="java.lang.String">
    <column name="address"></column></element>
    </set>
    </class>
    </hibernatemapping>
```

将该映射文件加入到Hibernate配置文件中，建立测试类Test.java，代码如下：

```
    package collect.set;
    import java.util.Iterator;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.Transaction;
    import org.hibernate.cfg.Configuration;
    public class Test {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（）.configure（）;
    //根据Configuration建立SessionFactory
    //SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）;
    UserSet user1=new UserSet（）; //创建实例
    user1.setName（"user1"）;
    user1.addEmail（"email1"）;
    user1.addEmail（"email2"）;
    //加入2个相同的元素，实际会覆盖原来的
    user1.addEmail（"email2"）;
```

```
    UserSet user2=new UserSet（）；
    user2.setName（"user2"）；
    user2.addEmail（"email3"）；
    Integer pid; //定义主键变量
    Session
session=sessionFactory.openSession（）；//添加数据
    Transaction tx=null;
    try {
    tx=session.beginTransaction（）；
    pid=（Integer）session.save（user1）；//创建主键变
量
    session.save（user2）；
    tx.commit（）；
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）；
    throw e;
    } finally {
    session.close（）；
    }
    session=sessionFactory.openSession（）；//修改数
据
    tx=null;
    try {
    tx=session.beginTransaction（）；
    user1=（UserSet）session.get（UserSet.class,
pid）；
    //修改user名字
    user1.setName（"user1 update"）；
    user1.removeEmail（"email1"）；
    user1.addEmail（"email4"）；
    session.update（user1）；
    tx.commit（）；
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）；
    throw e;
```

```
    } finally {
    session.close();
    }
    session=sessionFactory.openSession(); //查询数
据
    user1=(UserSet) session.get(UserSet.class,
pid);
    System.out.println("user
name: "+user1.getName());
    Iterator iter=user1.getEmails().iterator();
    while(iter.hasNext()) {
    System.out.println("email name: "+(String)
iter.next());
    }
    session.close();
    session=sessionFactory.openSession(); //删除数
据
    tx=null;
    try {
    tx=session.beginTransaction();
    user1=(UserSet) session.get(UserSet.class,
pid);
    session.delete(user1);
    tx.commit();
    } catch(RuntimeException e) {
    if(tx!=null)
    tx.rollback();
    throw e;
    } finally {
    session.close();
    }
    sessionFactory.close(); //关闭sessionFactory
    }
    }
```

运行该测试类，结果如下：

```
    08: 28: 50, 562 DEBUG SQL: 346insert into
ssh.user__set（name）values（? ）
    08: 28: 50, 609 DEBUG SQL: 346insert into
ssh.user__set（name）values（? ）
    08: 28: 50, 656 DEBUG SQL: 346insert into email__
set（id, address）values（? , ? ）
    08: 28: 50, 671 DEBUG SQL: 346insert into email__
set（id, address）values（? , ? ）
    08: 28: 50, 671 DEBUG SQL: 346insert into email__
set（id, address）values（? , ? ）
    08: 28: 50, 734 DEBUG SQL: 346select userset0
__.id as id0__0__, userset0__.name as name0__0__from
ssh.user__
    set userset0__where userset0__.id=?
    08: 28: 50, 750 DEBUG SQL: 346select emails0__.id
as id0__, emails0__.address as address0__from email
__set
    emails0__where emails0__.id=?
    08: 28: 50, 750 DEBUG SQL: 346update ssh.user__
set set name=? where id=?
    08: 28: 50, 750 DEBUG SQL: 346delete from email__
set where id=? and address=?
    08: 28: 50, 765 DEBUG SQL: 346insert into email__
set（id, address）values（? , ? ）
    08: 28: 50, 812 DEBUG SQL: 346select userset0
__.id as id0__0__, userset0__.name as name0__0__from
ssh.user__
    set userset0__where userset0__.id=?
    08: 28: 50, 812 DEBUG SQL: 346select emails0__.id
as id0__, emails0__.address as address0__from email
__set
    emails0__where emails0__.id=?
    user name: user1 update
```

```
    email name: email4
    email name: email2
    08: 28: 50, 812 DEBUG SQL: 346select userset0
__.id as id0__0__, userset0__.name as name0__0__from
ssh.user__
    set userset0__where userset0__.id=?
    08: 28: 50, 828 DEBUG SQL: 346delete from email__
set where id=?
    08: 28: 50, 828 DEBUG SQL: 346delete from
ssh.user__set where id=?
```

源程序解读

（1）由于一个User可以有多个电子邮件，而且每个电子邮件不能重复。所以在User set类中定义Set类型的变量，用来保存电子邮件。

（2）使用Set映射元素来关联email__set表，Set映射元素有如下配置。

name：集合属性的名称。

table：（可选，默认为属性的名称）这个集合表的名称（不能在一对多的关联关系中使用）。

schema（可选）：表的schema的名称，它将覆盖在根元素中定义的schema。

lazy（可选，默认为true）：可以用来关闭延迟加载（false），指定一直使用预先抓取，或者打开“extralazy”抓取，此时大多数操作不会初始化集合类（适用于非常大的集合）。

inverse（可选，默认为false）：标记这个集合作为双向关联关系中的方向一端。

cascade（可选，默认为none）：让操作级联到子实体。

sort（可选）：指定集合的排序顺序，可以为自然的（natural）或者给定一个用来比较的类。

orderby（可选，仅用于jdk1.4）：指定表的字段（一个或几个）再加上asc或者desc（可选），定义

Map、Set和Bag的迭代顺序。

where（可选）：指定任意的SQLwhere条件，该条件将在重新载入或者删除这个集合时使用（当集合中的数据仅仅是所有可用数据的一个子集时这个条件非常有用）。

fetch（可选，默认为select）：用于在外连接抓取、通过后续select抓取和通过后续subselect抓取之间选择。

batchsize（可选，默认为1）：指定通过延迟加载取得集合实例的批处理块大小（"batchsize"）。

access（可选，默认为属性property）：Hibernate取得集合属性值时使用的策略。

乐观锁（可选，默认为true）：对集合状态的改变会否导致其所属实体的版本增长。（对一对多关联

来说，关闭这个属性是有道理的）。

mutable（可变）（可选，默认为true）：若值为false，表明集合中的元素不会改变（在某些情况下可以进行一些小的性能优化）。

（3）本示例中，Set元素中的table属性指定了Set元素对应的表为email__set。

（4）Set集合属性特点是无序、不可重复的集合。所以，Set元素不必使用index元素来指定集合元素的次序。

（5）在执行测试类的添加模块时，数据库表内容如下：

```
User__set:
I dname
1  user1
2  user2
Email__set:
Idaddress
```

```
1  email1
1  email2
2  email3
```

执行修改模块后，数据库表内容如下：

```
User__set:
Idname
1user1 update
2user2
Email__set:
Idaddress
1email2
2email3
1email4
```

可见在修改数据时，删除了email1，增加了email4。

14.2 List集合映射

前一节介绍了Set映射，本节介绍List映射。List映射是有序的，所以需要增加一个列email__id来表示List的序号。关系如图14.2所示。

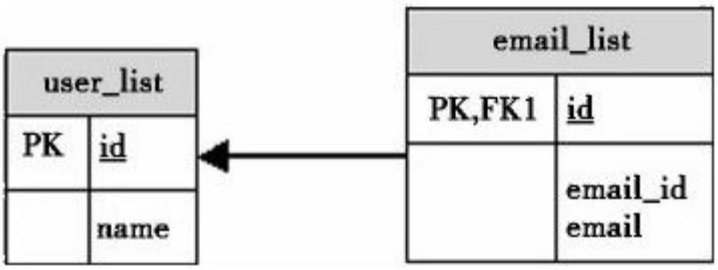

图 14.2 List映射关系图

技术要点

Set集合是无序的，集合元素并不是按照一定的顺序排列，而List集合是有序的，每个集合元素需要有一个表示集合序号的标识，这样可以使用该标识来获得该集合元素。在理解Set映射的基础上，理解List集

合映射，重点注意List集合是有序的，区别于Set集合。List集合映射的元素可以重复。

实现代码

建立两个相应的表，代码如下：

```
CREATE TABLE'email__list'（
′ id'int（11）NOT NULL，
′ email__id'int（11）NOT NULL，
′ email'varchar（100）NOT NULL default"
）ENGINE=InnoDB DEFAULT CHARSET=gbk
CREATE TABLE'user__list'（
′ id'int（11）NOT NULL auto__increment，
′ name'varchar（100）NOT NULL default"，
PRIMARY KEY（′ id'）
）ENGINE=InnoDB DEFAULT CHARSET=gbk
```

建立Userlist.java实体类，代码如下：

```
package collect.list;
import java.util.ArrayList;
import java.util.List;
public class UserList implements
java.io.Serializable {
private Integer id; //主键ID属性变量
private String name; //姓名name属性变量
```

```
    private List emails=new ArrayList（）；//List类型
属性变量
    public UserList（） {//构造器
    }
    public Integer getId（） {//ID属性的Getter（）方法
和Setter（）方法
    return this.id;
    }
    public void setId（Integer id） {
    this.id=id;
    }
    public String getName（） {//name属性的Getter（）
方法和Setter（）方法
    return this.name;
    }
    public void setName（String name） {
    this.name=name;
    }
    public List getEmails（） {//emails属性的
Getter（）方法和Setter（）方法
    return emails;
    }
    public void setEmails（List emails） {
    this.emails=emails;
    }
    public void addEmail（String email） {//添加一个
email的方法
    emails.add（email）；
    }
    public void removeEmail（String email） {//删除一
个email的方法
    emails.remove（email）；
    }
    }
```

建立映射文件UserList.hbm.xml，内容如下：

```
    <? xml version="1.0"encoding="utf8"? >
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
    <class name="collect.list.UserList"table="user
__list"catalog="ssh">
    <id name="id"type="java.lang.Integer">
    <column name="id"/>
    <generator class="native"/>
    </id>
    <property name="name"type="java.lang.String">
    <column name="name"length="100"notnull="true"/
>
    </property>
    <! List类型映射>
    <list name="emails"table="email__list"><! 配置
对应表>
    <key column="id"></key><! 配置主键>
    <index column="email__id"></index><! 配置List
索引>
    <element
type="java.lang.String"column="email"></element>
    </list>
    </class>
    </hibernatemapping>
```

将该映射文件加入到Hibernate配置文件中，建立测试类test.java，代码如下：

```
    package collect.list;
    import java.util.List;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.Transaction;
    import org.hibernate.cfg.Configuration;
    public class Test {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（） .configure（）；
    //根据Configuration建立SessionFactory,
SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）；
    UserList user1=new UserList（）；//创建实例
    user1.setName（"user1"）；
    user1.addEmail（"email1"）；
    user1.addEmail（"email2"）；
    UserList user2=new UserList（）；
    user2.setName（"user2"）；
    user2.addEmail（"email3"）；
    user2.addEmail（"email4"）；
    Integer pid; //定义主键变量
    Session
session=sessionFactory.openSession（）；//添加数据
    Transaction tx=null;
    try {
    tx=session.beginTransaction（）；
    pid=（Integer）session.save（user1）；//创建主键变
量
    session.save（user2）；
    tx.commit（）；
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）；
```

```
    throw e;
    } finally {
    session.close();
    }
    session=sessionFactory.openSession(); //修改数
据
    tx=null;
    try {
    tx=session.beginTransaction();
    user1=(UserList) session.get(UserList.class,
pid);
    //修改user名字
    user1.setName("user1 update");
    user1.removeEmail("email1");
    user1.addEmail("email5");
    session.update(user1);
    tx.commit();
    } catch(RuntimeException e) {
    if(tx! =null)
    tx.rollback();
    throw e;
    } finally {
    session.close();
    }
    session=sessionFactory.openSession(); //查询数
据
    user1=(UserList) session.get(UserList.class,
pid);
    System.out.println("user
name: "+user1.getName());
    List list=user1.getEmails();
    for(int j=0; j<list.size(); j++) {
    System.out.println("email
name: "+list.get(j));
    }
    session.close();
```

```
    session=sessionFactory.openSession（）; //删除数
据
    tx=null;
    try {
    tx=session.beginTransaction（）;
    user1=（UserList）session.get（UserList.class,
pid）;
    session.delete（user1）;
    tx.commit（）;
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）;
    throw e;
    } finally {
    session.close（）;
    }
    sessionFactory.close（）; //关闭sessionFactory
    }
    }
```

运行该测试类，结果如下：

```
    08: 59: 08, 421 DEBUG SQL: 346insert into
ssh.user__list（name）values（? ）
    08: 59: 08, 453 DEBUG SQL: 346insert into
ssh.user__list（name）values（? ）
    08: 59: 08, 468 DEBUG SQL: 346insert into email__
list（id, email__id, email）values（? , ? , ? ）
    08: 59: 08, 468 DEBUG SQL: 346insert into email__
list（id, email__id, email）values（? , ? , ? ）
    08: 59: 08, 468 DEBUG SQL: 346insert into email__
list（id, email__id, email）values（? , ? , ? ）
    08: 59: 08, 468 DEBUG SQL: 346insert into email__
list（id, email__id, email）values（? , ? , ? ）
```

```
    08: 59: 08, 546 DEBUG SQL: 346select userlist0
__.id as id0__0__, userlist0__.name as name 0__0__
from
    ssh.user__list userlist0__where userlist0
__.id=?
    08: 59: 08, 562 DEBUG SQL: 346select emails0__.id
as id0__, emails0__.email as email0__, emails0
__.email__id as
    email3__0__from email__list emails0__where
emails0__.id=?
    08: 59: 08, 578 DEBUG SQL: 346update ssh.user__
list set name=? where id=?
    08: 59: 08, 593 DEBUG SQL: 346update email__list
set email=? where id=? and email__id=?
    08: 59: 08, 640 DEBUG SQL: 346select userlist0
__.id as id0__0__, userlist0__.name as name 0__0__
from ssh.user__
    list userlist0__where userlist0__.id=?
    user name: user1 update
    08: 59: 08, 640 DEBUG SQL: 346select emails0__.id
as id0__, emails0__.email as email0__, emails0
__.email__id as
    email3__0__from email__list emails0__where
emails0__.id=?
    email name: email2
    email name: email5
    08: 59: 08, 640 DEBUG SQL: 346select userlist0
__.id as id0__0__, userlist0__.name as name 0__0__
from ssh.user__
    list userlist0__where userlist0__.id=?
    08: 59: 08, 656 DEBUG SQL: 346delete from email__
list where id=?
    08: 59: 08, 656 DEBUG SQL: 346delete from
ssh.user__list where id=?
```

源程序解读

（1）相对于Set集合，List集合是有序的，需要增加email__id字段记录List元素序号。（2）其他配置同Set集合一致。

（3）执行添加模块后，数据库内容如下：

```
User__list:
Idname
1user1
2user2
Email__set:
Idemail__i ddress
10 email1
11 email2
20 email3
21 email4
```

执行修改模块后，内容如下：

```
User__list:
Idname
1user1 update
2user2
Email__set:
Idemail__i ddress
```

```
10 email2
11 email5
20 email3
21 email4
```

14.3 Map集合映射

Map集合的特点就是使用了键值对，即KeyValue结构来存放集合元素，这样就需要对应的集合元素数据库表中包含对于Key的列，其表的结构与图14.2类似。

技术要点

熟悉Java开发的读者对Map集合类型不会陌生，Map集合的特点是使用了KeyValue的方式，即键值对来组织集合元素，Hibernate支持Map集合类型的映射。本节代码演示如何使用Map集合映射。本示例中Map的key使用了字符串。Map集合的特点就是使用键值对。

实现代码

建立相应的数据库表，语句如下：

```
CREATE TABLE'email__map'（
```

```
′ id'int（11）NOT NULL，
′ email__id'varchar（20）NOT NULL，
′ email'varchar（100）NOT NULL default"
）ENGINE=InnoDB DEFAULT CHARSET=gbk
CREATE TABLE'user__map'（
′ id'int（11）NOT NULL auto__increment，
′ name'varchar（100）NOT NULL default"，
PRIMARY KEY（′ id'）
）ENGINE=InnoDB DEFAULT CHARSET=gbk
```

建立实体类Usermap.java，代码如下：

```
package collect.map;
import java.util.HashMap;
import java.util.Map;
public class UserMap implements
java.io.Serializable {
private Integer id; //主键ID属性变量
private String name; //姓名name属性变量
private Map emails=new HashMap（）; //Map类型集合
属性变量
public UserMap（） {//构造器
}
public Integer getId（） {//ID属性的Getter（）方法
和Setter（）方法
return this.id;
}
public void setId（Integer id） {
this.id=id;
}
public String getName（） {//name属性的Getter（）
方法和Setter（）方法
return this.name;
}
```

```
    public void setName（String name） {
    this.name=name;
    }
    public Map getEmails（） {//emails属性的Getter（）
方法和Setter（）方法
    return emails;
    }
    public void setEmails（Map emails） {
    this.emails=emails;
    }
    }
```

建立映射文件UserMap.hbm.xml，内容如下：

```
    <? xml version="1.0"encoding="utf8"? >
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
    <class name="collect.map.UserMap"table="user__
map"catalog="ssh">
    <id name="id"type="java.lang.Integer">
    <column name="id"/>
    <generator class="native"/>
    </id>
    <property name="name"type="java.lang.String">
    <column name="name"length="100"notnull="true"/
>
    </property>
    <! Map类型映射>
    <map name="emails"table="email__map"><! 配置对
应表>
    <key column="id"></key><! 配置主键>
```

```
    <! 配置键值对>
    <mapkey column="email__
id"type="java.lang.String"></mapkey>
    <element
type="java.lang.String"column="email"></element>
    </map>
    </class>
    </hibernatemapping>
```

将该映射文件加入到Hibernate配置文件中，建立测试类test.java，内容如下：

```
    package collect.map;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.Transaction;
    import org.hibernate.cfg.Configuration;
    public class Test {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（）.configure（）;
    //根据Configuration建立SessionFactory
    //SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）;
    UserMap user1=new UserMap（）; //创建实例
    user1.setName（"user1"）;
    user1.getEmails（）.put（"no.1", "email1"）;
    user1.getEmails（）.put（"no.2", "email2"）;
    UserMap user2=new UserMap（）;
    user2.setName（"user2"）;
    user2.getEmails（）.put（"no.3", "email3"）;
```

```
    user2.getEmails（）.put（"no.4", "email4"）;
    Integer pid; //定义主键变量
    Session
session=sessionFactory.openSession（）; //添加数据
    Transaction tx=null;
    try {
    tx=session.beginTransaction（）;
    pid=（Integer）session.save（user1）; //主键变量
    session.save（user2）;
    tx.commit（）;
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）;
    throw e;
    } finally {
    session.close（）;
    }
    session=sessionFactory.openSession（）; //修改数
据
    tx=null;
    try {
    tx=session.beginTransaction（）;
    user1=（UserMap）session.get（UserMap.class,
pid）;
    //修改user名字
    user1.setName（"user1 update"）;
    user1.getEmails（）.put（"no.5", "email5"）;
    user1.getEmails（）.remove（"no.1"）;
    user1.getEmails（）.put（"no.2", "emai2
update"）;
    session.update（user1）;
    tx.commit（）;
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）;
    throw e;
    } finally {
```

```
    session.close（）；
    }
    session=sessionFactory.openSession（）；//查询数
据
    user1=（UserMap）session.get（UserMap.class,
pid）；
    System.out.println（"user
name: "+user1.getName（））；
    System.out.println（"email
name: "+user1.getEmails（））；
    session.close（）；
    session=sessionFactory.openSession（）；//删除数
据
    tx=null;
    try {
    tx=session.beginTransaction（）；
    user1=（UserMap）session.get（UserMap.class,
pid）；
    session.delete（user1）；
    tx.commit（）；
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）；
    throw e;
    } finally {
    session.close（）；
    }
    sessionFactory.close（）；//关闭sessionFactory
    }
    }
```

运行该测试类，结果如下：

```
    09: 18: 04, 875 DEBUG SQL: 346insert into
ssh.user__map（name）values（? ）
    09: 18: 04, 906 DEBUG SQL: 346insert into
ssh.user__map（name）values（? ）
    09: 18: 04, 921 DEBUG SQL: 346insert into email__
map（id, email__id, email）values（? , ? , ? ）
    09: 18: 04, 921 DEBUG SQL: 346insert into email__
map（id, email__id, email）values（? , ? , ? ）
    09: 18: 04, 921 DEBUG SQL: 346insert into email__
map（id, email__id, email）values（? , ? , ? ）
    09: 18: 04, 921 DEBUG SQL: 346insert into email__
map（id, email__id, email）values（? , ? , ? ）
    09: 18: 05, 015 DEBUG SQL: 346select usermap0
__.id as id0__0__, usermap0__.name as name0__0__from
    ssh.user__map usermap0__where usermap0__.id=?
    09: 18: 05, 062 DEBUG SQL: 346select emails0__.id
as id0__, emails0__.email as email0__, emails0
__.email__id as
    email3__0__from email__map emails0__where
emails0__.id=?
    09: 18: 05, 078 DEBUG SQL: 346update ssh.user__
map set name=? where id=?
    09: 18: 05, 109 DEBUG SQL: 346delete from email__
map where id=? and email__id=?
    09: 18: 05, 109 DEBUG SQL: 346update email__map
set email=? where id=? and email__id=?
    09: 18: 05, 125 DEBUG SQL: 346insert into email__
map（id, email__id, email）values（? , ? , ? ）
    09: 18: 05, 156 DEBUG SQL: 346select usermap0
__.id as id0__0__, usermap0__.name as name0__0__from
    ssh.user__map usermap0__where usermap0__.id=?
    user name: user1 update
    09: 18: 05, 171 DEBUG SQL: 346select emails0__.id
as id0__, emails0__.email as email0__, emails0
__.email__id as
    email3__0__from email__map emails0__where
emails0__.id=?
```

```
    email name: {no.2=emai2 update, no.5=email5}
    09: 18: 05, 203 DEBUG SQL: 346select usermap0
__.id as id0__0__, usermap0__.name as name0__0__from
    ssh.user__map usermap0__where usermap0__.id=?
    09: 18: 05, 218 DEBUG SQL: 346delete from email__
map where id=?
    09: 18: 05, 218 DEBUG SQL: 346delete from
ssh.user__map where id=?
```

源程序解读

（1）Map使用了String类型作为Key。

（2）从日志输出可以看到，Key和Value的对应关系。

（3）执行添加模块后，数据库内容如下：

```
    User__map:
    Idname
    1user1
    2user2
    Email__map:
    Idemail__i ddress
    1no.1 email1
    1no.2 email2
    2no.3 email3
    2no.4 email4
```

执行修改模块后，内容如下：

```
User__list:
Idname
1user1 update
2user2
Email__set:
Idemail__i ddress
1no.2 email2 update
2no.3 email3
2no.4 email4
1no.5 email5
```

14.4　Bag集合映射

前面介绍的Set集合是无序、不能有重复元素的，Hibernate提供了一个Bag集合，用来处理重复元素的情况。值得注意的是，Bag并不是Java API，而是Hibernate提供的。本节还是以用户（user）和电子邮件（email）的关系为例，一个用户可以包含重复的电子邮件地址，可以使用Bag类型的集合映射。

技术要点

前面介绍的集合类型都是在Java API中定义的，本节介绍Hibernate提供的一个集合类型Bag，顾名思义，该集合的元素是可以重复的。Bag集合映射与List不同，List的集合元素是有序的，需要有一个集合序号来标识集合元素的位置，List集合元素可以重复；

而Bag集合元素不需要元素序号标识，元素也是可以重复的。

实现代码

建立两个相关的数据库表，语句如下：

```
CREATE TABLE'email__bag'（
′ id'int（11）NOT NULL，
′ address'varchar（100）NOT NULL
）ENGINE=InnoDB DEFAULT CHARSET=gbk
CREATE TABLE'user__bag'（
′ id'int（11）NOT NULL auto__increment，
′ name'varchar（100）NOT NULL default"，
PRIMARY KEY（′ id'）
）ENGINE=InnoDB DEFAULT CHARSET=gbk
```

建立User的实体类Userbag.java，代码如下：

```
package collect.bag;
import java.util.ArrayList;
import java.util.List;
public class UserBag implements
java.io.Serializable {
private Integer id; //主键ID属性变量
private String name; //姓名name属性变量
private List emails=new ArrayList（）; //List类型
属性变量
```

```
    public UserBag（）｛//构造器
    }
    public Integer getId（）｛//ID属性的Getter（）方法和Setter（）方法
    return this.id;
    }
    public void setId（Integer id）｛
    this.id=id;
    }
    public String getName（）｛//name属性的Getter（）方法和Setter（）方法
    return this.name;
    }
    public void setName（String name）｛
    this.name=name;
    }
    public List getEmails（）｛//emails属性的Getter（）方法和Setter（）方法
    return emails;
    }
    public void setEmails（List emails）｛
    this.emails=emails;
    }
    public void addEmail（String email）｛//增加一个email元素
    emails.add（email）;
    }
    public void removeEmail（String email）｛//产出一个email元素
    emails.remove（email）;
    }
    }
```

建立映射文件UserBag.hbm.xml并加入到Hibernate配置文件中，内容如下：

```
    <? xml version="1.0"encoding="utf8"? >
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
    <class name="collect.bag.UserBag"table="user__
bag"catalog="ssh">
    <id name="id"type="java.lang.Integer">
    <column name="id"/>
    <generator class="native"/>
    </id>
    <property name="name"type="java.lang.String">
    <column name="name"length="100"notnull="true"/
>
    </property>
    <! bag类型映射>
    <bag name="emails"table="email__bag"><! 配置对
应表>
    <key column="id"></key><! 定义主键>
    <element type="string"column="email">
</element><! 定义元素类型>
    </bag>
    </class>
    </hibernatemapping>
```

建立测试类Test.java，代码如下：

```
    package collect.bag;
    import java.util.List;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.Transaction;
    import org.hibernate.cfg.Configuration;
    public class Test {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（） .configure（）；
    //根据Configuration建立SessionFactory,
SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）；
    UserBag user1=new UserBag（）; //创建实例
    user1.setName（"user1"）；
    user1.addEmail（"email1"）；
    user1.addEmail（"email1"）；
    UserBag user2=new UserBag（）；
    user2.setName（"user2"）；
    user2.addEmail（"email2"）；
    user2.addEmail（"email2"）；
    Integer pid; //定义主键变量
    Session
session=sessionFactory.openSession（）; //添加数据
    Transaction tx=null;
    try {
    tx=session.beginTransaction（）；
    pid=（Integer）session.save（user1）; //创建主键变
量
    session.save（user2）；
    tx.commit（）；
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）；
    throw e;
```

```
    } finally {
    session.close();
    }
    session=sessionFactory.openSession(); //修改数
据
    tx=null;
    try {
    tx=session.beginTransaction();
    user1=(UserBag)session.get(UserBag.class,
pid);
    user1.setName("user1 update"); //修改user名字
    user1.removeEmail("email1"); //删除一个email1
    user1.addEmail("email5");
    session.update(user1);
    tx.commit();
    } catch(RuntimeException e) {
    if(tx! =null)
    tx.rollback();
    throw e;
    } finally {
    session.close();
    }
    session=sessionFactory.openSession(); //查询数
据
    user1=(UserBag)session.get(UserBag.class,
pid);
    System.out.println("user
name: "+user1.getName());
    List list=user1.getEmails();
    for(int j=0; j<list.size(); j++) {
    System.out.println("email
name: "+list.get(j));
    }
    session.close();
    sessionFactory.close(); //关闭sessionFactory
    }
    }
```

运行该测试类，结果如下：

```
    09: 35: 31, 281 DEBUG SQL: 346insert into
ssh.user__bag（name）values（? ）
    09: 35: 31, 312 DEBUG SQL: 346insert into
ssh.user__bag（name）values（? ）
    09: 35: 31, 328 DEBUG SQL: 346insert into email__
bag（id, email）values（? , ? ）
    09: 35: 31, 328 DEBUG SQL: 346insert into email__
bag（id, email）values（? , ? ）
    09: 35: 31, 328 DEBUG SQL: 346insert into email__
bag（id, email）values（? , ? ）
    09: 35: 31, 328 DEBUG SQL: 346insert into email__
bag（id, email）values（? , ? ）
    09: 35: 31, 406 DEBUG SQL: 346select userbag0
__.id as id0__0__, userbag0__.name as name0__0__from
ssh.user
    __bag userbag0__where userbag0__.id=?
    09: 35: 31, 421 DEBUG SQL: 346select emails0__.id
as id0__, emails0__.email as email0__from email__bag
    emails0__where emails0__.id=?
    09: 35: 31, 437 DEBUG SQL: 346update ssh.user__
bag set name=? where id=?
    09: 35: 31, 437 DEBUG SQL: 346delete from email__
bag where id=?
    09: 35: 31, 453 DEBUG SQL: 346insert into email__
bag（id, email）values（? , ? ）
    09: 35: 31, 453 DEBUG SQL: 346insert into email__
bag（id, email）values（? , ? ）
    09: 35: 31, 515 DEBUG SQL: 346select userbag0
__.id as id0__0__, userbag0__.name as name0__0__from
ssh.user
    __bag userbag0__where userbag0__.id=?
    user name: user1 update
```

```
    09: 35: 31，531 DEBUG SQL: 346select emails0__.id
as id0__，emails0__.email as email0__from email__bag
    emails0__where emails0__.id=?
    email name: email1
    email name: email5
```

源程序解读

（1）在User实体类中，List类型的成员可以被映射为Bag，由于List中的元素是有序号的，而Bag则会忽略该序号。

（2）在添加数据模块中，数据库表内容如下：

```
    User__bag:
    Idname
    1user1
    2user2
    Email__bag:
    Idaddress
    1email1
    1email1
    2email2
    2email2
```

执行修改模块后，数据库表内容如下：

```
User__bag:
Idname
1user1 update
2user2
Email__bag:
Idaddress
1email1
1email5
2email2
2email2
```

（3）在更新数据时，注意日志输出：

```
09: 35: 31, 437 DEBUG SQL: 346delete from email__bag where id=?
09: 35: 31, 453 DEBUG SQL: 346insert into email__bag（id, email）values（? , ? ）
09: 35: 31, 453 DEBUG SQL: 346insert into email__bag（id, email）values（? , ? ）
```

Bag类型映射中，由于集合元素中有重复的元素，在更新数据时，系统先删除集合表中的所有数据，然后重新将集合中的数据插入到数据表中。这样的机制效率不高。Hibernate提供了一个iBag映射，使用collectionid属性来定位数据表的位置，这一点与List集合映射类似。

14.5 Component映射

Component映射可以称为组件映射，本节介绍最简单的组件映射，假设有一个用户表c__user，表结构如图14.3所示。

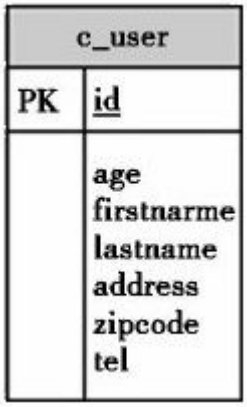

图 14.3 组件映射表

该表是一个用户信息表，可以将用户信息归纳为两个部分：一个部分是name（姓名），包含firstname（姓）和lastname（名）；另一部分是Contact（联系方式），包括了address（地址）、zipcode（邮编）和tel（电话）等信息。在创建实体

类时，可以将name和Contact分别封装到2个独立的类中，这样就提高了系统的复用性和灵活性。也就是说，需要使用Component映射，将其他的实体类映射在一起。

技术要点

Component映射就是将一个复杂的实体分解为多个简单的，或者是易于管理的组件，然后组合在一起，形成一个完整的实体。本节介绍基本的Component映射配置。

实现代码

建立1个相关的数据库表，语句如下：

```
CREATE TABLE'c__user'（
′ id'int（11）NOT NULL auto__increment,
′ age'int（11）default NULL,
′ firstname'varchar（50）default NULL,
′ lastname'varchar（50）default NULL,
′ address'varchar（200）default NULL,
```

```
′ zipcode'varchar（10）default NULL，
′ tel'varchar（20）default NULL，
PRIMARY KEY（′ id'）
）ENGINE=InnoDB DEFAULT CHARSET=gbk
```

接着建立一个Name.java类，内容如下：

```
package collect.component;
import java.io.Serializable;
public class Name implements Serializable {
private String firstname; //first name属性变量
private String lastname; //last name属性变量
public String getFirstname（） {//firstname属性的
Getter（）方法和Setter（）方法
return firstname;
}
public void setFirstname（String firstname） {
this.firstname=firstname;
}
public String getLastname（） {//lastname属性的
Getter（）方法和Setter（）方法
return lastname;
}
public void setLastname（String lastname） {
this.lastname=lastname;
}
}
```

建立联系方式相关的类Contact.java，代码如下：

```
package collect.component;
import java.io.Serializable;
public class Contact implements Serializable {
private String address; //地址属性变量
private String zipcodes; //编码属性变量
private String tel; //电话属性变量
public String getAddress（） {//address属性的
Getter（）方法和Setter（）方法
return address;
}
public void setAddress（String address） {
this.address=address;
}
public String getZipcodes（） {//zipcodes属性的
Getter（）方法和Setter（）方法
return zipcodes;
}
public void setZipcodes（String zipcodes） {
this.zipcodes=zipcodes;
}
public String getTel（） {//tel属性的Getter（）方
法和Setter（）方法
return tel;
}
public void setTel（String tel） {
this.tel=tel;
}
public Contact（） {
}
}
```

最后建立数据库表对应的实体类Cuser.java，代码如下：

```
    package collect.component;
    public class Cuser {
    private Integer id; //主键ID属性变量
    private Integer age; //年龄age属性变量
    private Name name; //姓名name属性变量
    private Contact contact; //通信类contact属性变量
    public Integer getId() {//ID属性的Getter()方法
和Setter()方法
    return id;
    }
    public void setId(Integer id) {
    this.id=id;
    }
    public Integer getAge() {//age属性的Getter()方
法和Setter()方法
    return age;
    }
    public void setAge(Integer age) {
    this.age=age;
    }
    public Name getName() {//name属性的Getter()方
法和Setter()方法
    return name;
    }
    public void setName(Name name) {
    this.name=name;
    }
    public Contact getContact() {//contact属性的
Getter()方法和Setter()方法
    return contact;
    }
    public void setContact(Contact contact) {
    this.contact=contact;
    }
    }
```

建立映射文件Cuser.hbm.xml，内容如下：

```
    <? xml version="1.0"? >
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
    <class name="collect.component.Cuser"table="c__
user"catalog="ssh">
    <id name="id"type="integer">
    <column name="id"/>
    <generator class="native"/>
    </id>
    <property name="age"type="integer">
    <column name="age"/>
    </property>
    <! 配置组件映射>
    <component
name="name"class="collect.component.Name"><! 组件
对应的类>
    <property name="firstname"type="string">
    <column name="firstname"length="50"/>
    </property>
    <property name="lastname"type="string">
    <column name="lastname"length="50"/>
    </property>
    </component>
    <component
name="contact"class="collect.component.Contact">
<! 组件对应的类
    <property name="address"type="string">
    <column name="address"length="200"/>
    </property>
    <property name="zipcodes"type="string">
```

```
    <column name="zipcode"length="10"/>
    </property>
    <property name="tel"type="string">
    <column name="tel"length="20"/>
    </property>
    </component>
    </class>
    </hibernatemapping>
```

将该映射文件加入到Hibernate的配置文件中，建立一个测试类Test.java，代码如下：

```
    package collect.component;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.Transaction;
    import org.hibernate.cfg.Configuration;
    public class Test {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（）.configure（）;
    //根据Configuration建立SessionFactory,
SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）;
    Name name=new Name（）; //创建实例
    name.setFirstname（"闫"）;
    name.setLastname（"术卓"）;
    Contact contact=new Contact（）;
    contact.setAddress（"北京"）;
    contact.setTel（"42689334"）;
    contact.setZipcodes（"100085"）;
```

```
Cuser user=new Cuser（）；
user.setAge（33）；
user.setName（name）；
user.setContact（contact）；
Integer pid; //定义主键变量
//添加数据
Session session=sessionFactory.openSession（）；
Transaction tx=null;
try {
tx=session.beginTransaction（）；
pid=（Integer）session.save（user）; //创建主键变量
tx.commit（）；
} catch（RuntimeException e） {
if（tx! =null）
tx.rollback（）；
throw e;
} finally {
session.close（）；
}
sessionFactory.close（）; //关闭sessionFactory
}
}
```

运行该示例，结果如下：

```
14: 16: 00，421 DEBUG SQL: 346insert into ssh.c__
user（age, firstname, lastname, address, zipcode,
tel）
values（? , ? , ? , ? , ? , ? ）
```

源程序解读

（1）映射文件使用Component元素将name类、Contact类同数据库表c__user联系起来。

（2）执行添加模块后，数据库内容如下：

---

idagefirstnamelastnameaddresszipcodetel 1 30闫术卓北京100085 42689334

14.6 Compositeelement映射

假设有一个团队（team）和成员（teammembers）表，其关系如图14.4所示。可以看到，每一个team都可以拥有多个teammembers。使用Compositeelement映射能够完成这种需求。

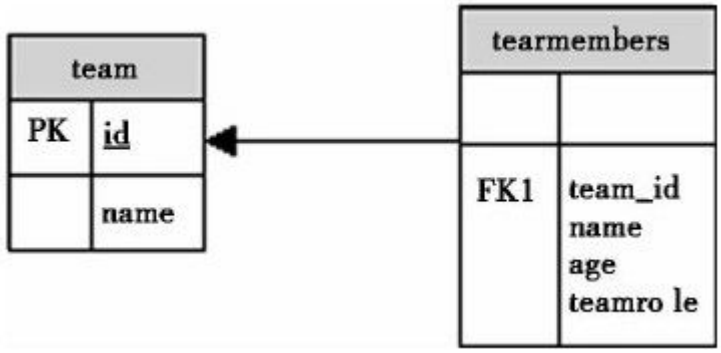

图 14.4 Compositeelement映射

技术要点

Compositeelement映射非常类似于一对多的关系映射，配置Compositeelement映射，可以实现简单的

一对多关系，本节以一个简单的示例帮助读者理解Compositeelement映射。

实现代码

建立两个相关的数据库表，语句如下：

```
CREATE TABLE'team'（
′ id'int（11）NOT NULL auto__increment，
′ name'varchar（50）default NULL，
PRIMARY KEY（′ id'）
）ENGINE=InnoDB DEFAULT CHARSET=gbk
CREATE TABLE'teammembers'（
′ team__id'int（11）default NULL，
′ name'varchar（20）default NULL，
′ age'int（3）default NULL，
′ teamrole'varchar（20）default NULL
）ENGINE=InnoDB DEFAULT CHARSET=gbk
```

建立实体类Team.java，代码如下：

```
package collect.composite__element;
import java.util.HashMap;
import java.util.Map;
public class Team implements
java.io.Serializable {
private Integer id; //主键ID属性变量
private String name; //姓名name属性变量
```

```
    private Map memebers=new HashMap（）; //Map类型属性变量
    public Team（） {//构造器
    }
    public Integer getId（） {//ID属性的Getter（）方法和Setter（）方法
    return this.id;
    }
    public void setId（Integer id） {
    this.id=id;
    }
    public String getName（） {//name属性的Getter（）方法和Setter（）方法
    return this.name;
    }
    public void setName（String name） {
    this.name=name;
    }
    public Map getMemebers（） {//memebers属性的Getter（）方法和Setter（）方法
    return memebers;
    }
    public void setMemebers（Map memebers） {
    this.memebers=memebers;
    }
    }
```

建立团队成员的实体类Members.java，代码如下：

```
    package collect.composite__element;
    import java.io.Serializable;
    public class Member {
```

```
    private String id; //主键ID属性变量
    private String name; //姓名属性变量
    private Team team; //所属团队属性变量
    private String age; //年龄属性变量
    public String getAge（） {//age属性的Getter（）方法和Setter（）方法
    return age;
    }
    public void setAge（String age） {
    this.age=age;
    }
    public String getName（） {//name属性的Getter（）方法和Setter（）方法
    return name;
    }
    public void setName（String name） {
    this.name=name;
    }
    public String getId（） {//ID属性的Getter（）方法和Setter（）方法
    return id;
    }
    public void setId（String id） {
    this.id=id;
    }
    public Team getTeam（） {//team属性的Getter（）方法和Setter（）方法
    return team;
    }
    public void setTeam（Team team） {
    this.team=team;
    }
    }
```

建立一个映射文件Team.hbm.xml，内容如下：

```
    <? xml version="1.0"encoding="utf8"? >
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
    <import class="collect.composite__
element.Member"/>
    <class name="collect.composite__
element.Team"table="team"
    catalog="ssh">
    <id name="id"type="java.lang.Integer">
    <column name="id"/>
    <generator class="native"/>
    </id>
    <property name="name"type="java.lang.String">
    <column name="name"length="50"/>
    </property>
    <! 配置Map集合映射>
    <map name="memebers"table="teammembers"><! 指
定集合元素对应表>
    <key column="team__id"></key><! 指定主键>
    <mapkey
column="teamrole"type="java.lang.String"></mapkey
>
    <! compositeelement映射配置>
    <compositeelement class="collect.composite__
element.Member">
    <parent name="team"/>
    <property name="name"/>
    <property name="age"></property>
    </compositeelement>
    </map>
    </class>
    </hibernatemapping>
```

将该映射文件加入到Hibernate配置文件中，建立一个测试类Test.java，代码如下：

```
    package collect.composite__element;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.Transaction;
    import org.hibernate.cfg.Configuration;
    import collect.map.UserMap;
    public class Test {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（）.configure（）;
    //根据Configuration建立SessionFactory,
SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）;
    Team t=new Team（）; //创建实例
    t.setName（"team1"）;
    Member m1=new Member（）;
    m1.setName（"m1"）;
    m1.setAge（"33"）;
    Member m2=new Member（）;
    m2.setName（"m2"）;
    m2.setAge（"22"）;
    t.getMemebers（）.put（"程序员", m1）;
    t.getMemebers（）.put（"测试工程师", m2）;
    Session
session=sessionFactory.openSession（）; //添加数据
    Transaction tx=null;
    try {
    tx=session.beginTransaction（）;
    pid=（Integer）session.save（t）; //创建主键变量
```

```
    tx.commit（）；
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）；
    throw e;
    } finally {
    session.close（）；
    }
    Member m3=new Member（）；//修改数据
    m3.setName（"m3"）；
    m3.setAge（"33"）；
    session=sessionFactory.openSession（）；
    tx=null;
    try {
    tx=session.beginTransaction（）；
    t=（Team）session.get（Team.class, pid）；
    t.setName（"team update"）；//修改名字
    t.getMemebers（）.put（"项目经理", m3）；
    t.getMemebers（）.remove（"测试工程师"）；
    session.update（t）；
    tx.commit（）；
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）；
    throw e;
    } finally {
    session.close（）；
    }
    session=sessionFactory.openSession（）；//查询数
据
    t=（Team）session.get（Team.class, pid）；
    System.out.println（"team
name: "+t.getName（））；
    System.out.println（"member
name: "+t.getMemebers（））；
    session.close（）；
    sessionFactory.close（）；//关闭sessionFactory
```

```
    }
    }
    Integer pid; //定义主键变量
```

运行该测试类，结果如下：

```
    14: 38: 44, 468 DEBUG SQL: 346insert into
ssh.team（name）values（？）
    14: 38: 44, 515 DEBUG SQL: 346insert into
teammembers（team__id, teamrole, name, age）
values（？，？，？，
    ？）
    14: 38: 44, 515 DEBUG SQL: 346insert into
teammembers（team__id, teamrole, name, age）
values（？，？，？，
    ？）
    14: 38: 44, 593 DEBUG SQL: 346select team0__.id
as id0__0__, team0__.name as name0__0__from ssh.team
    team0__where team0__.id=?
    14: 38: 44, 625 DEBUG SQL: 346select memebers0
__.team__id as team1__0__, memebers0__.name as name0
__,
    memebers0__.age as age0__, memebers0__.teamrole
as teamrole0__from teammembers memebers0__where
    memebers0__.team__id=?
    14: 38: 44, 625 DEBUG SQL: 346update ssh.team
set name=? where id=?
    14: 38: 44, 656 DEBUG SQL: 346delete from
teammembers where team__id=? and teamrole=?
    14: 38: 44, 656 DEBUG SQL: 346insert into
teammembers（team__id, teamrole, name, age）
values（？，？，？，
    ？）
```

```
    14: 38: 44, 703 DEBUG SQL: 346select team0__.id
as id0__0__, team0__.name as name0__0__from ssh.team
    team0__where team0__.id=?
    team name: team update
    14: 38: 44, 718 DEBUG SQL: 346select memebers0
__.team__id as team1__0__, memebers0__.name as name0
__,
    memebers0__.age as age0__, memebers0__.teamrole
as teamrole0__from teammembers memebers0__where
    memebers0__.team__id=?
    member name: {程序员=collect.composite__
element.Member@ea48be, 项目经理=collect.composite__
ele
    ment.Member@14dd758}
```

源程序解读

（1）映射文件中使用了map集合映射，key指定了对应的key为team__id, mapkey则指定了Map集合元素的索引，这里是teamrole，即团队角色（这里假定角色不能重复，即Key值唯一）。

（2）使用compositeelement元素将Map集合中的每个元素映射给teammember的相应字段，这里映射了Members实体类。

（3）运行添加模块后，数据库内容如下：

```
Team:
Idname
1team1
teammembers:
team__i ameageteamrole
1 m1 33程序员
1 m2 22测试工程师
```

执行修改模块后，数据库内容如下：

```
Team:
Idname
2team update
teammembers:
team__i ameageteamrole
1 m1 33程序员
1 m3 33项目经理
```

第15章　Hibernate关系映射

作为一种轻量级封装的关系映射工具，Hibernate支持各种关系映射，例如多对一、一对多、多对多和一对一的数据库表关系，通过映射文件的灵活配置即可轻松实现。Hibernate的重要部分就是关系映射。笔者将在本章介绍各种映射及其单向、双向的配置和示例，每个示例演示了如何进行简单的CRUD操作。

15.1　单向多对多映射

假设有角色（trole）和用户组（tgroup）两个表，是多对多的关系，即一个角色可以多个用户组拥有，一个用户组也可以拥有多个角色。这里需要增加一个角色-组的对应表tgroup_role，用来记录多对多的对应关系，如图15.1所示。

例如一个网站的用户角色可以有查看、添加、删除、修改等功能角色，用户又可以分为管理员、版主、注册用户和匿名用户等，那么这里的用户和角色就是多对多的关系，即一个用户可以拥有多个角色，每个角色也可以赋给多个用户。本示例演示单向的多对多关系，即trole表作为主动方。

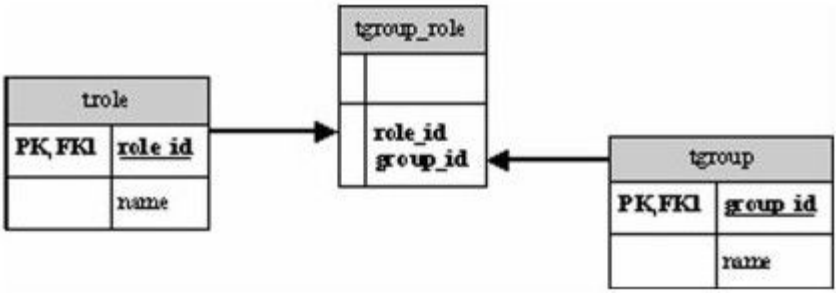

图　15.1　多对多关系表

技术要点

单向多对多关系一般都是通过关联表实现关联，在一般的数据库应用中使用比较多，也是读者必须掌握的。本节代码演示如何建立单向的多对多关系，并简单进行操作。

熟悉manytomany配置。

简单的对象操作。

实现代码

首先建立三个对应表格，语句如下：

```
CREATE TABLE'tgroup'（
′ group__id'int（11）NOT NULL auto__increment，
′ name'varchar（16）NOT NULL default"，
PRIMARY KEY（′ group__id'）
）ENGINE=InnoDB DEFAULT CHARSET=gbk
CREATE TABLE'trole'（
′ role__id'int（11）NOT NULL auto__increment，
′ name'varchar（16）NOT NULL default"，
PRIMARY KEY（′ role__id'）
）ENGINE=InnoDB DEFAULT CHARSET=gbk
CREATE TABLE'tgroup__role'（
′ group__id'int（11）NOT NULL，
′ role__id'int（11）NOT NULL
）ENGINE=InnoDB DEFAULT CHARSET=gbk
```

建立角色的实体类Role.java，代码如下：

```
package relation.unidirectional.manytomany;
import java.util.HashSet;
import java.util.Set;
```

```
public class Role {
private int id; //主键ID属性变量
private String name; /项目属性变量
private Set groups=new HashSet（）; //集合类型属性变量
public Role（） {//构造器
}
public int getId（） {//ID属性的Getter（）方法和Setter（）方法
return id;
}
public void setId（int id） {
this.id=id;
}
public String getName（） {//name属性的Getter（）方法和Setter（）方法
return name;
}
public void setName（String name） {
this.name=name;
}
public Set getGroups（） {//groups属性的Getter（）方法和Setter（）方法
return groups;
}
public void setGroups（Set groups） {
this.groups=groups;
}
}
```

建立用户组的实体类Group.java，代码如下：

```
package relation.unidirectional.manytomany;
import java.util.HashSet;
```

```
import java.util.Set;
public class Group {
private int id; //主键ID属性变量
private String name; //姓名属性变量
private Set roles=new HashSet（）; //集合类型属性变量
public Group（） {//构造器
}
public int getId（） {//ID属性的Getter（）方法和Setter（）方法
return id;
}
public void setId（int id） {
this.id=id;
}
public String getName（） {//name属性的Getter（）方法和Setter（）方法
return name;
}
public void setName（String name） {
this.name=name;
}
public Set getRoles（） {//roles属性的Getter（）方法和Setter（）方法
return roles;
}
public void setRoles（Set roles） {
this.roles=roles;
}
}
```

Role是多对多的控制方，其映射文件

Role.hbm.xml内容如下：

```
    <? xml version="1.0"encoding="utf8"? >
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
    <class
name="relation.unidirectional.manytomany.Role"tabl
e="trole">
    <id name="id"column="role__
id"unsavedvalue="0">
    <generator class="native"/>
    </id>
    <property name="name"type="string"/>
    <set name="groups"table="tgroup__
role"cascade="saveupdate">
    <key column="role__id"/>
    <! 配置多对多关系映射>
    <manytomany
class="relation.unidirectional.manytomany.Group"
    column="group__id"/>
    </set>
    </class>
    </hibernatemapping>
```

Group是被控制方，Group.hbm.xml内容如下：

```
    <? xml version="1.0"encoding="utf8"? >
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
```

```
    <class
name="relation.unidirectional.manytomany.Group"tab
le="tgroup">
    <id name="id"column="group__
id"unsavedvalue="0">
    <generator class="native"/><! 定义主键生成方式>
    </id><! 配置主键映射>
    <property name="name"type="string"/><! 配置
name映射>
    </class>
    </hibernatemapping>
```

将上面的两个映射文件加入到Hibernate配置文件中。建立一个测试类Test.java，代码如下：

```
    package relation.unidirectional.manytomany;
    import java.util.Iterator;
    import
relation.unidirectional.manytoone.Person;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.Transaction;
    import org.hibernate.cfg.Configuration;
    public class Test {
    public static void main（String[]args） {
    Role role1=new Role（）; //创建对象
    role1.setName（"Role1"）;
    Role role2=new Role（）; //创建对象
    role2.setName（"Role2"）;
    Role role3=new Role（）; //创建对象
    role3.setName（"Role3"）;
    Group group1=new Group（）; //创建对象
    group1.setName（"group1"）;
```

```
    Group group2=new Group（）; //创建对象
    group2.setName（"group2"）;
    Group group3=new Group（）; //创建对象
    group3.setName（"group3"）;
    group1.getRoles（）.add（role1）; //相互赋值
    group1.getRoles（）.add（role2）;
    group2.getRoles（）.add（role2）;
    group2.getRoles（）.add（role3）;
    group3.getRoles（）.add（role1）;
    group3.getRoles（）.add（role3）;
    role1.getGroups（）.add（group1）; //相互赋值
    role1.getGroups（）.add（group3）;
    role2.getGroups（）.add（group1）;
    role2.getGroups（）.add（group2）;
    role3.getGroups（）.add（group2）;
    role3.getGroups（）.add（group3）;
    Integer pid; //定义主键变量
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（）.configure（）;
    //根据Configuration建立SessionFactory,
SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）;
    Session
session=sessionFactory.openSession（）; //正向添加数
据
    Transaction tx=null; //开始事务
    try {
    tx=session.beginTransaction（）;
    pid=（Integer）session.save（role1）;
    session.save（role2）; //保存对象
    session.save（role3）;
    tx.commit（）; //提交事务
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）;
```

```
    throw e;
    } finally {
    session.close();
    }
    Group group4=new Group(); //修改role1数据
    group4.setName("group4"); //赋值
    session=sessionFactory.openSession();
    tx=null; //事务开始
    try {
    tx=session.beginTransaction();
    role1=(Role) session.get(Role.class, pid);
    role1.getGroups().add(group4); //role1增加
group4
    session.update(role1);
    tx.commit();
    } catch(RuntimeException e) {
    if(tx! =null)
    tx.rollback();
    throw e;
    } finally {
    session.close();
    }
    session=sessionFactory.openSession(); //查询数
据
    role1=(Role) session.get(Role.class, pid);
    System.out.println("role
name: "+role1.getName());
    System.out.println("groups: ");
    Iterator iter=role1.getGroups().iterator();
    while(iter.hasNext()) {//循环打印
    group1=(Group) iter.next();
    System.out.println("group
name: "+group1.getName());
    }
    session.close();
    session=sessionFactory.openSession(); //删除
role1数据
```

```
tx=null; //开始事务
try {
tx=session.beginTransaction（）;
role1=（Role）session.get（Role.class, pid）;
session.delete（role1）;
tx.commit（）;
} catch（RuntimeException e） {
if（tx! =null）
tx.rollback（）;
throw e;
} finally {
session.close（）;
}
//反向添加数据，只增加了group数据，对role没有影响
session=sessionFactory.openSession（）;
tx=null; //开始事务
try {
tx=session.beginTransaction（）;
session.save（group1）;
session.save（group2）;
tx.commit（）;
} catch（RuntimeException e） {
if（tx! =null）
tx.rollback（）;
throw e;
} finally {
session.close（）;
}
sessionFactory.close（）; //关闭sessionFactory
}
}
```

执行该测试类，可以看到控制台信息：

14: 01: 06，671 DEBUG SQL: 346insert into trole (name) values (?)
14: 01: 06，718 DEBUG SQL: 346insert into tgroup (name) values (?)
14: 01: 06，718 DEBUG SQL: 346insert into tgroup (name) values (?)
14: 01: 06，718 DEBUG SQL: 346insert into trole (name) values (?)
14: 01: 06，734 DEBUG SQL: 346insert into tgroup (name) values (?)
14: 01: 06，734 DEBUG SQL: 346insert into trole (name) values (?)
14: 01: 06，765 DEBUG SQL: 346insert into tgroup__role (role__id, group__id) values (? , ?)
14: 01: 06，765 DEBUG SQL: 346insert into tgroup__role (role__id, group__id) values (? , ?)
14: 01: 06，765 DEBUG SQL: 346insert into tgroup__role (role__id, group__id) values (? , ?)
14: 01: 06，765 DEBUG SQL: 346insert into tgroup__role (role__id, group__id) values (? , ?)
14: 01: 06，765 DEBUG SQL: 346insert into tgroup__role (role__id, group__id) values (? , ?)
14: 01: 06，765 DEBUG SQL: 346insert into tgroup__role (role__id, group__id) values (? , ?)
14: 01: 06，875 DEBUG SQL: 346select role0__.role__id as role1__13__0__, role0__.name as name13__0__ from trole
role0__where role0__.role__id=?
14: 01: 06，890 DEBUG SQL: 346select groups0__.role__id as role1__1__, groups0__.group__id as group2__1__,
group1__.group__id as group1__12__0__, group1__.name as name12__0__from tgroup__role groups0__ left outer join
tgroup group1__on groups0__.group__id=group1__.group__id where groups0__.role__id=?

```
    14: 01: 06, 890 DEBUG SQL: 346insert into
tgroup（name）values（? ）
    14: 01: 06, 906 DEBUG SQL: 346insert into tgroup
__role（role__id, group__id）values（? , ? ）
    14: 01: 06, 937 DEBUG SQL: 346select role0__.role
__id as role1__13__0__, role0__.name as name13__0__
from trole
    role0__where role0__.role__id=?
    role name: Role1
    groups:
    14: 01: 06, 953 DEBUG SQL: 346select groups0
__.role__id as role1__1__, groups0__.group__id as
group2__1__,
    group1__.group__id as group1__12__0__, group1
__.name as name12__0__from tgroup__role groups0__
left outer join
    tgroup group1__on groups0__.group__id=group1
__.group__id where groups0__.role__id=?
    group name: group3
    group name: group4
    group name: group1
    14: 01: 06, 953 DEBUG SQL: 346select role0__.role
__id as role1__13__0__, role0__.name as name13__0__
from trole
    role0__where role0__.role__id=?
    14: 01: 06, 968 DEBUG SQL: 346delete from tgroup
__role where role__id=?
    14: 01: 06, 968 DEBUG SQL: 346delete from trole
where role__id=?
    14: 01: 07, 015 DEBUG SQL: 346insert into
tgroup（name）values（? ）
    14: 01: 07, 015 DEBUG SQL: 346insert into
tgroup（name）values（? ）
```

源程序解读

（1）控制方需要建立一个Set集合类型属性，用来保存多个Group对象。

（2）在Role.hbm.xml文件中，配置set元素。set的name属性对应Role类的groups变量。table指明其对应的表，key指定了集合中对象同Role主键关联的键值。

（3）多对多关系通过manytomany配置，其属性如下。

column（可选）：这个元素的外键关键字段名。

formula（可选）：用于计算元素外键值的SQL公式。

class（必需）：关联类的名称。

outerjoin（可选，默认为auto）：在Hibernate系统参数中hibernate.use__outer__join被打开的情况下，该参数用来允许使用outer join来载入此集合的数据。为此关联打开外连接抓取或者后续select抓取。这是特殊情况；对于一个实体及其指向其他实体的多对多关联进行预先抓取（使用一条单独的SELECT），不仅需要对集合自身打开join，也需要对＜manytomany＞这个内嵌元素打开此属性。

notfound（可选，默认为exception）：指明引用的外键中缺少某些行该如何处理：ignore会把缺失的行作为一个空引用处理。

entityname（可选）：被关联的类的实体名，作为class的替代。

propertyref（可选）：被关联到此外键（foreign key）类中的对应属性的名字。若未指定，

使用被关联类的主键。

（4）从SQL日志可以看到，这里的多对多是单向的，当反向添加Group对象时，对role并没有影响。

15.2　双向多对多映射

前一节介绍了单向的多对多映射，本节介绍双向多对多映射，表结构没有改变。如果示例中用户和角色是相互制约的，即每个用户都有相关角色，每个角色都必须赋给多个用户，这时就是双向多对多的关系。

技术要点

前面介绍了单向的多对多映射，本节介绍双向的多对多映射，读者需要区分双向多对多映射相对于单向的不同之处。多对多关系也是数据库表常见的关联方式，与单向多对多映射相比，读者必须掌握inverse的用法。

实现代码

相对于单向多对多映射，这里需要修改Group.hbm.xml文件，内容如下：

```
    <? xml version="1.0"encoding="utf8"? >
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
    <class
name="relation.bidirectional.manytomany.Group"tabl
e="tgroup">
    <id name="id"column="group__
id"unsavedvalue="0">
    <generator class="native"/>
    </id>
    <property name="name"type="string"/>
    <set name="roles"table="tgroup__
role"cascade="saveupdate">
    <key column="group__id"/>
    <! 配置多对多关系映射>
    <manytomany
class="relation.bidirectional.manytomany.Role"
    column="role__id"/>
    </set>
    </class>
    </hibernatemapping>
```

Role. hbm.xml内容如下：

```
    <? xml version="1.0"encoding="utf8"? >
```

```
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
    <class
name="relation.bidirectional.manytomany.Role"table
="trole">
    <id name="id"column="role__
id"unsavedvalue="0">
    <generator class="native"/>
    </id>
    <property name="name"type="string"/>
    <set name="groups"table="tgroup__
role"cascade="saveupdate"
    inverse="true"><! 双向映射inverse需要为true>
    <key column="role__id"/>
    <! 配置多对多关系映射>
    <manytomany
class="relation.bidirectional.manytomany.Group"
    column="group__id"/>
    </set>
    </class>
    </hibernatemapping>
```

将这两个映射文件替换到Hibernate配置文件，增加测试类test.java，内容如下：

```
    package relation.bidirectional.manytomany;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.Transaction;
    import org.hibernate.cfg.Configuration;
```

```
    public class Test {
    public static void main（String[]args） {
    Role role1=new Role（）; //建立对象
    role1.setName（"Role1"）;
    Role role2=new Role（）;
    role2.setName（"Role2"）;
    Role role3=new Role（）;
    role3.setName（"Role3"）;
    Group group1=new Group（）; //建立对象
    group1.setName（"group1"）;
    Group group2=new Group（）;
    group2.setName（"group2"）;
    Group group3=new Group（）;
    group3.setName（"group3"）;
    group1.getRoles（）.add（role1）; //相互赋值
    group1.getRoles（）.add（role2）;
    group2.getRoles（）.add（role2）;
    group2.getRoles（）.add（role3）;
    group3.getRoles（）.add（role1）;
    group3.getRoles（）.add（role3）;
    role1.getGroups（）.add（group1）; //相互赋值
    role1.getGroups（）.add（group3）;
    role2.getGroups（）.add（group1）;
    role2.getGroups（）.add（group2）; //相互赋值
    role3.getGroups（）.add（group2）;
    role3.getGroups（）.add（group3）;
    Integer pid; //定义主键变量
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（）.configure（）;
    //根据Configuration建立SessionFactory
    //SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）;
    Session
session=sessionFactory.openSession（）; //添加数据
    Transaction tx=null; //开始事务
```

```
try {
tx=session.beginTransaction（）;
pid=（Integer）session.save（role1）;
session.save（role2）;
session.save（role3）;
tx.commit（）;
} catch（RuntimeException e） {
if（tx! =null）
tx.rollback（）;
throw e;
} finally {
session.close（）;
}
session=sessionFactory.openSession（）; //删除数
据
tx=null; //开始事务
try {
tx=session.beginTransaction（）;
role1=（Role）session.get（Role.class, pid）;
session.delete（role1）;
tx.commit（）;
} catch（RuntimeException e） {
if（tx! =null）
tx.rollback（）;
throw e;
} finally {
session.close（）;
}
Role role4=new Role（）; //反向添加数据
role4.setName（"role4"）;
Group group4=new Group（）;
group4.setName（"group4"）;
group4.getRoles（） .add（role4）;
role4.getGroups（） .add（group4）;
session=sessionFactory.openSession（）;
tx=null; //开始事务
try {
```

```
tx=session.beginTransaction（）；
session.save（group4）；
tx.commit（）；
} catch（RuntimeException e） {
if（tx! =null）
tx.rollback（）；
throw e;
} finally {
session.close（）；
}
sessionFactory.close（）；//关闭sessionFactory
}
}
```

运行该测试类，结果如下：

```
    14: 04: 26，656 DEBUG SQL: 346insert into
trole（name）values（？）
    14: 04: 26，687 DEBUG SQL: 346insert into
tgroup（name）values（？）
    14: 04: 26，703 DEBUG SQL: 346insert into
trole（name）values（？）
    14: 04: 26，703 DEBUG SQL: 346insert into
tgroup（name）values（？）
    14: 04: 26，703 DEBUG SQL: 346insert into
trole（name）values（？）
    14: 04: 26，703 DEBUG SQL: 346insert into
tgroup（name）values（？）
    14: 04: 26，734 DEBUG SQL: 346insert into tgroup
__role（group__id，role__id）values（？，？）
    14: 04: 26，734 DEBUG SQL: 346insert into tgroup
__role（group__id，role__id）values（？，？）
    14: 04: 26，734 DEBUG SQL: 346insert into tgroup
__role（group__id，role__id）values（？，？）
```

```
    14: 04: 26，734 DEBUG SQL: 346insert into tgroup
__role（group__id, role__id）values（？，？）
    14: 04: 26，750 DEBUG SQL: 346insert into tgroup
__role（group__id, role__id）values（？，？）
    14: 04: 26，750 DEBUG SQL: 346insert into tgroup
__role（group__id, role__id）values（？，？）
    14: 04: 26，843 DEBUG SQL: 346select role0__.role
__id as role1__14__0__，role0__.name as name 14__0__
from trole
    role0__where role0__.role__id=?
    14: 04: 26，859 DEBUG SQL: 346delete from trole
where role__id=?
    14: 04: 26，921 DEBUG SQL: 346insert into
tgroup（name）values（？）
    14: 04: 26，921 DEBUG SQL: 346insert into
trole（name）values（？）
    14: 04: 26，921 DEBUG SQL: 346insert into tgroup
__role（group__id, role__id）values（？，？）
```

实现了双向的多对多映射。

源程序解读

（1）相对于上一节，本节两个映射文件都需要配置manytomany元素。

（2）增加inverse属性。inverse属性负责控制关系，默认为false，也就是关系的两端都能控制，但这

样会造成一些问题，更新时会因为两端都控制关系，于是重复更新。一般来说有一端要设为true。

（3）role映射文件增加inverse=true，说明有group一方来维护它们之间的多对多关系。在双向映射中，一方需要配置inverse=true。

15.3　单向多对一映射

在数据库表关系中，经常出现多对一的映射，例如一个人只能有一个住址（假定），一个住址可以住多个人。那么人（person）和住址（address）就是多对一的关系，如图15.2所示。本节演示单向多对一实例，从“多”的一方控制“一”的一方。

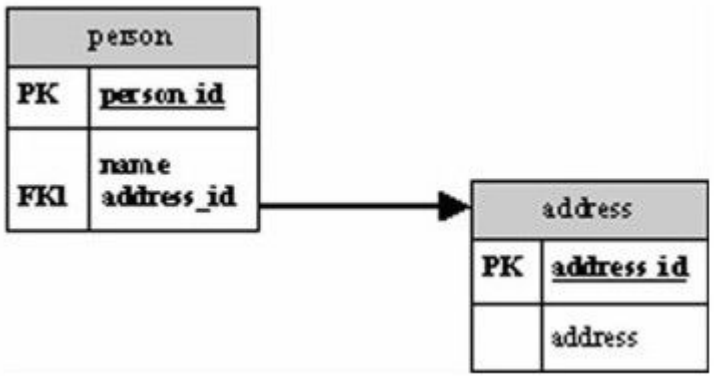

图　15.2　多对一关系

技术要点

人-地址是典型的单向一对多的映射关系，特点是映射为单向，“多”的一方为主动方。一般都是“多”的一方设置一个外键来实现关联。本节介绍如何配置单向多对一的映射关系，熟悉manytoone配置。

实现代码

建立相应的数据库表，语句如下：

```
CREATE TABLE'person'（
′person__id'int（11）NOT NULL auto__increment，
′name'varchar（20）default NULL，
′address__id'int（11）default NULL，
PRIMARY KEY（′person__id'）
）ENGINE=InnoDB DEFAULT CHARSET=gbk
CREATE TABLE'address'（
′address__id'int（11）NOT NULL auto__increment，
′address'varchar（50）default NULL，
PRIMARY KEY（′address__id'）
）ENGINE=InnoDB DEFAULT CHARSET=gbk
```

建立Person实体类Person.java，代码如下：

```
package relation.unidirectional.manytoone;
public class Person implements
java.io.Serializable {
```

```
    private Integer personId; //人员ID属性变量
    private String name; //姓名属性变量
    private Address address; //地址属性变量
    public Person（） {//构造器}
    public Integer getPersonId（） {//ID属性的
Getter（）方法和Setter（）方法
    return this.personId;
    }
    public void setPersonId（Integer personId） {
    this.personId=personId;
    }
    public String getName（） {//name属性的Getter（）
方法和Setter（）方法
    return this.name;
    }
    public void setName（String name） {
    this.name=name;
    }
    public Address getAddress（） {//address属性的
Getter（）方法和Setter（）方法
    return address;
    }
    public void setAddress（Address address） {
    this.address=address;
    }
    }
```

建立Address实体类Address.java，代码如下：

```
    package relation.unidirectional.manytoone;
    public class Address implements
java.io.Serializable {
    private Integer addressId; //地址id属性变量
    private String address; //地址属性变量
```

```
    public Address（）｛//构造器
    }
    public Integer getAddressId（）｛//addressid属性
的Getter（）方法和Setter（）方法
    return this.addressId;
    }
    public void setAddressId（Integer addressId）｛
    this.addressId=addressId;
    }
    public String getAddress（）｛//address属性的
Getter（）方法和Setter（）方法
    return this.address;
    }
    public void setAddress（String address）｛
    this.address=address;
    }
    }
```

建立映射文件Person.hbm.xml，内容如下：

```
    ＜? xml version="1.0"encoding="utf8"? ＞
    ＜! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd"＞
    ＜hibernatemapping＞
    ＜class
name="relation.unidirectional.manytoone.Person"tab
le="person"
    catalog="ssh"＞
    ＜id name="personId"type="java.lang.Integer"＞
    ＜column name="person__id"/＞
    ＜generator class="native"＞＜/generator＞
    ＜/id＞
```

```
    <property name="name"type="java.lang.String">
    <column name="name"length="10"/>
    </property><!配置多对一关联关系>
    <manytoone name="address"column="address__id"
    class="relation.unidirectional.manytoone.Addres
s"outerjoin="true"
    cascade="saveupdate"><!配置save/update>
    </manytoone>
    </class>
    </hibernatemapping>
```

建立映射文件Address.hbm.xml，内容如下：

```
    <?xml version="1.0"encoding="utf8"?>
    <!DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
    <class
name="relation.unidirectional.manytoone.Address"ta
ble="address"catalog="ssh">
    <id name="addressId"type="java.lang.Integer">
    <column name="address__id"/>
    <generator class="native"></generator>
    </id><!配置主键映射>
    <property
name="address"type="java.lang.String">
    <column name="address"length="50"/><!配置地址
映射>
    </property>
    </class>
    </hibernatemapping>
```

将映射文件添加到Hibernate配置文件中，建立测试类Test.java，代码如下：

```
    package relation.unidirectional.manytoone;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.Transaction;
    import org.hibernate.cfg.Configuration;
    public class Test {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（）.configure（）;
    //根据Configuration建立SessionFactory
    //SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）;
    Address add1=new Address（）; //创建实例
    add1.setAddress（"address1"）;
    Address add2=new Address（）;
    add2.setAddress（"address2"）;
    Person person1=new Person（）; //建立person对象
    Person person2=new Person（）;
    Person person3=new Person（）;
    person1.setName（"peson1"）; //赋值
    person1.setAddress（add1）;
    person2.setName（"person2"）;
    person2.setAddress（add2）; //对象赋值
    person3.setName（"person3"）;
    person3.setAddress（add1）;
    Integer pid; //定义主键变量
    Session
session=sessionFactory.openSession（）; //添加数据
    Transaction tx=null; //开始事务
```

```
    try {
    tx=session.beginTransaction（）;
    //创建主键变量
    pid=（Integer）session.save（person1）;
    session.save（person2）;
    session.save（person3）;
    tx.commit（）;
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）;
    throw e;
    } finally {
    session.close（）;
    }
    //修改person2，并修改对应的地址为address2
    session=sessionFactory.openSession（）;
    tx=null;
    try {//开始事务
    tx=session.beginTransaction（）;
    person1=（Person）session.get（Person.class,
pid）;
    person1.setName（"person2 updated"）;
    person1.getAddress（）.setAddress（"address2
update! "）;
    session.update（person1）;
    tx.commit（）;
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）;
    throw e;
    } finally {
    session.close（）;
    }
    session=sessionFactory.openSession（）; //查询数
据
    person1=（Person）session.get（Person.class,
pid）;
```

```
    System. out.println ("person
name: "+person1.getName ( ) ) ;
```

```
    System.out.println ("address: "+person1.getAddre
ss ( ) .getAddress ( ) ) ;
    session.close ( ) ;
    session=sessionFactory.openSession ( ) ; //删除数
据
    tx=null; //开始事务
    try {
    tx=session.beginTransaction ( ) ;
    person1= (Person) session.get (Person.class,
pid) ;
    session.delete (person1) ;
    tx.commit ( ) ;
    } catch (RuntimeException e) {
    if (tx! =null)
    tx.rollback ( ) ;
    throw e;
    } finally {
    session.close ( ) ;
    }
    Address add3=new Address ( ) ; //反向添加数据
    add3.setAddress ("address3") ;
    session=sessionFactory.openSession ( ) ;
    tx=null; //开始事务
    try {
    tx=session.beginTransaction ( ) ;
    session.save (add3) ;
    tx.commit ( ) ;
    } catch (RuntimeException e) {
    if (tx! =null)
```

```
    tx.rollback（）；
    throw e;
    } finally {
    session.close（）；
    }
    sessionFactory.close（）；//关闭sessionFactory
    }
```

运行该测试类，结果如下：

```
    14: 08: 14, 390 DEBUG SQL: 346insert into
ssh.address（address）values（？）
    14: 08: 14, 421 DEBUG SQL: 346insert into
ssh.person（name, address__id）values（？，？）
    14: 08: 14, 421 DEBUG SQL: 346insert into
ssh.address（address）values（？）
    14: 08: 14, 437 DEBUG SQL: 346insert into
ssh.person（name, address__id）values（？，？）
    14: 08: 14, 437 DEBUG SQL: 346insert into
ssh.person（name, address__id）values（？，？）
    14: 08: 14, 531 DEBUG SQL: 346select person0
__.person__id as person1__2__1__, person0__.name as
name2__1__,
    person0__.address__id as address3__2__1__,
address1__.address__id as address1__3__0__, address1
__.address as ad
    dress3__0__from ssh.person person0__left outer
join ssh.address address1__on person0__.address__
id=address1__
    .address__id where person0__.person__id=?
    14: 08: 14, 546 DEBUG SQL: 346update ssh.address
set address=? where address__id=?
    14: 08: 14, 546 DEBUG SQL: 346update ssh.person
set name=?，address__id=? where person__id=?
```

```
    14: 08: 14, 593 DEBUG SQL: 346select person0
__.person__id as person1__2__1__, person0__.name as
name2__1__,
    person0__.address__id as address3__2__1__,
address1__.address__id as address1__3__0__, address1
__.address as ad
    dress3__0__from ssh.person person0__left outer
join ssh.address address1__on person0__.address__
id=address1__
    .address__id where person0__.person__id=?
    person name: person2 updated
    address: address2 update!
    14: 08: 14, 609 DEBUG SQL: 346select person0
__.person__id as person1__2__1__, person0__.name as
name2__1__,
    person0__.address__id as address3__2__1__,
address1__.address__id as address1__3__0__, address1
__.address as ad
    dress3__0__from ssh.person person0__left outer
join ssh.address address1__on person0__.address__
id=address1__
    .address__id where person0__.person__id=?
    14: 08: 14, 609 DEBUG SQL: 346delete from
ssh.person where person__id=?
    14: 08: 14, 687 DEBUG SQL: 346insert into
ssh.address（address）values（? ）
```

源程序解读

（1）这里关键是理解Person.hbm.xml映射文件。

（2）由于是多对一关系，控制方Person需要增加Address属性，用来表示对应一个Address。

（3）多对一使用manytoone元素。

（4）cascade属性表示当控制方（Person）在进行添加、修改、删除时，是否关联被控制方（Address）。这里是saveupdate，即在添加、修改时关联。

（5）outerjoin设置为true，即在查询关联对象时，一个SQL语句就可以查出；如果设置为false，则使用多个SQL语句查出数据。

15.4　单向一对多映射

典型的一对多映射就是公司与雇员的关系，即一个公司可以有多个雇员，关系如图15.3所示。

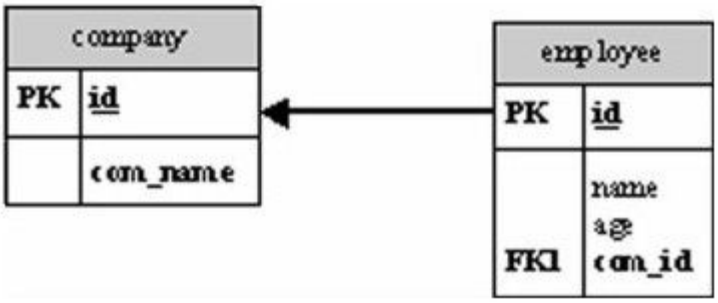

图　15.3　一对多关系

技术要点

单向一对多的映射关系不是很常见，其配置同单向多对一非常相似，其主动方为"一"的一方，使用onetomany配置属性。本节代码演示如何配置单向一对多关系，并使用一个测试类，完成简单的CRUD操作和onetomany配置。

实现代码

建立相关表格，语句如下：

```
CREATE TABLE'company'（
′ id'int（11）NOT NULL auto__increment，
′ com__name'varchar（20）default NULL，
PRIMARY KEY（′ id'）
）ENGINE=InnoDB DEFAULT CHARSET=gbk
CREATE TABLE'employee'（
′ id'int（11）NOT NULL auto__increment，
′ name'varchar（20）default NULL，
′ age'int（11）default NULL，
′ com__id'int（11）default NULL，
PRIMARY KEY（′ id'）
）ENGINE=InnoDB DEFAULT CHARSET=gbk
```

建立公司实体类company.java，代码如下：

```
package relation.unidirectional.onetomany;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Set;
public class Company implements
java.io.Serializable {
private Integer id; //主键ID属性变量
private String comName; //公司名称属性变量
private Set employees=new HashSet（）; //雇员属性
变量
public Company（） {//构造器
}
```

```
    public Integer getId（） {//ID属性的Getter（）方法
和Setter（）方法
    return this.id;
    }
    public void setId（Integer id） {
    this.id=id;
    }
    public String getComName（） {//comname属性的
Getter（）方法和Setter（）方法
    return this.comName;
    }
    public void setComName（String comName） {
    this.comName=comName;
    }
    public Set getEmployees（） {//employees属性的
Getter（）方法和Setter（）方法
    return employees;
    }
    public void setEmployees（Set employees） {
    this.employees=employees;
    }
    public void addEmployees（Employee employee）
{//增加一个employee
    employees.add（employee）;
    }
    public void removeEmployees（Employee employee）
{//删除一个employee
    employees.remove（employee）;
    }
    }
```

建立雇员的实体类employee.java，内容如下：

```
    package relation.unidirectional.onetomany;
```

```
    public class Employee implements
java.io.Serializable {
    private Integer id; //主键ID属性变量
    private String name; //姓名属性变量
    private Integer age; //年龄属性变量
    public Employee（） {//构造器
    }
    public Integer getId（） {//id属性的Getter（）方法
和Setter（）方法
    return this.id;
    }
    public void setId（Integer id） {
    this.id=id;
    }
    public String getName（） {//name属性的Getter（）
方法和Setter（）方法
    return this.name;
    }
    public void setName（String name） {
    this.name=name;
    }
    public Integer getAge（） {//age属性的Getter（）方
法和Setter（）方法
    return this.age;
    }
    public void setAge（Integer age） {
    this.age=age;
    }
    }
```

建立公司映射文件Company.hbm.xml，内容如下：

```
    ＜? xml version="1.0"encoding="utf8"? ＞
```

```
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
    <class
name="relation.unidirectional.onetomany.Company"ta
ble="company"
    catalog="ssh">
    <id name="id"type="java.lang.Integer">
    <column name="id"/>
    <generator class="native"/>
    </id><! 配置主键映射>
    <property
name="comName"type="java.lang.String">
    <column name="com__name"length="20"></column>
    </property><! 配置公司名称映射>
    <set
name="employees"table="employee"cascade="all">
    <key column="com__id"/>
    <! 配置多对一关系映射>
    <onetomany
    class="relation.unidirectional.onetomany.Employ
ee"></onetomany>
    </set>
    </class>
    </hibernatemapping>
```

建立雇员映射文件Employee.hbm.xml，内容如下：

```
    <? xml version="1.0"encoding="utf8"? >
```

```
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
    <class
name="relation.unidirectional.onetomany.Employee"t
able="employee"catalog="ssh">
    <id name="id"type="java.lang.Integer">
    <column name="id"/>
    <generator class="native"/>
    </id><! 配置主键映射>
    <property name="name"type="java.lang.String">
    <column name="name"length="20"/>
    </property><! 配置姓名映射>
    <property name="age"type="java.lang.Integer">
    <column name="age"/>
    </property><! 配置年龄映射>
    </class>
    </hibernatemapping>
```

建立测试类test.java，代码如下：

```
    package relation.unidirectional.onetomany;
    import java.util.Iterator;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.Transaction;
    import org.hibernate.cfg.Configuration;
    public class Test {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（） .configure（）;
```

```
    //根据Configuration建立SessionFactory
    //SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）；
    Employee e1=new Employee（）；//创建实例
    e1.setName（"Employee1"）；//对象赋值
    e1.setAge（30）；
    Employee e2=new Employee（）；//创建对象
    e2.setName（"Employee2"）；
    e2.setAge（33）；//对象赋值
    Employee e3=new Employee（）；//创建对象
    e3.setName（"Employee3"）；
    e3.setAge（35）；//对象赋值
    Company com1=new Company（）；//创建对象，赋值
    com1.setComName（"Company1"）；
    Company com2=new Company（）；
    com2.setComName（"Company2"）；
    com1.addEmployees（e1）；//赋值
    com1.addEmployees（e2）；
    com2.addEmployees（e3）；
    Integer pid；//定义主键变量
    Session
session=sessionFactory.openSession（）；//添加数据
    Transaction tx=null;
    try｛//开始事务
    tx=session.beginTransaction（）；
    //创建主键变量
    pid=（Integer）session.save（com1）；
    session.save（com2）；
    tx.commit（）；
    ｝catch（RuntimeException e）｛
    if（tx！=null）
    tx.rollback（）；
    throw e;
    ｝finally｛
    session.close（）；
    ｝
```

```
    session=sessionFactory.openSession（）; //修改数
据增加Employee4
    tx=null; //开始事务
    Employee e4=new Employee（）;
    e4.setName（"Employee4"）;
    e4.setAge（44）;
    //e1.setName（"Employee1 updated! "）;
    try {//开始事务
    tx=session.beginTransaction（）;
    com1=（Company）session.get（Company.class,
pid）;
    com1.setComName（"QQ.com"）; //修改com名字
    com1.getEmployees（）.add（e4）;
    session.update（com1）;
    tx.commit（）;
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）;
    throw e;
    } finally {
    session.close（）;
    }
    session=sessionFactory.openSession（）; //查询数
据
    com1=（Company）session.get（Company.class,
pid）;
    System.out.println（"Company
name: "+com1.getComName（））;
    Iterator
iter=com1.getEmployees（）.iterator（）;
    while（iter.hasNext（））{
    e1=（Employee）iter.next（）;
    System.out.println（"Employee
name: "+e1.getName（））;
    }
    session.close（）;
```

```
    session=sessionFactory.openSession（）；//删除数
据
    tx=null；//开始事务
    try｛
    tx=session.beginTransaction（）；
    com1=（Company）session.get（Company.class,
pid）；
    session.delete（com1）；
    tx.commit（）；
    ｝catch（RuntimeException e）｛
    if（tx! =null）
    tx.rollback（）；
    throw e;
    ｝finally｛
    session.close（）；
    ｝
    e4.setName（"李四"）；//添加被动对象
    e4.setAge（38）；
    session=sessionFactory.openSession（）；
    tx=null；//开始事务
    try｛
    tx=session.beginTransaction（）；
    session.save（e4）；
    tx.commit（）；
    ｝catch（RuntimeException e）｛
    if（tx! =null）
    tx.rollback（）；
    throw e;
    ｝finally｛
    session.close（）；
    ｝
    sessionFactory.close（）；//关闭sessionFactory
    ｝
    ｝
```

运行该测试类，结果如下：

```
    14: 28: 05, 593 DEBUG SQL: 346insert into
ssh.company（com__name）values（? ）
    14: 28: 05, 625 DEBUG SQL: 346insert into
ssh.employee（name, age）values（? , ? ）
    14: 28: 05, 640 DEBUG SQL: 346insert into
ssh.employee（name, age）values（? , ? ）
    14: 28: 05, 640 DEBUG SQL: 346insert into
ssh.company（com__name）values（? ）
    14: 28: 05, 640 DEBUG SQL: 346insert into
ssh.employee（name, age）values（? , ? ）
    14: 28: 05, 671 DEBUG SQL: 346update
ssh.employee set com__id=? where id=?
    14: 28: 05, 671 DEBUG SQL: 346update
ssh.employee set com__id=? where id=?
    14: 28: 05, 671 DEBUG SQL: 346update
ssh.employee set com__id=? where id=?
    14: 28: 05, 781 DEBUG SQL: 346select company0
__.id as id4__0__, company0__.com__name as com2__4__0
__from
    ssh.company company0__where company0__.id=?
    14: 28: 05, 796 DEBUG SQL: 346select employees0
__.com__id as com4__1__, employees0__.id as id1__,
employees0
    __.id as id5__0__, employees0__.name as name5__0
__, employees0__.age as age5__0__from ssh.employee
employees0__
    where employees0__.com__id=?
    14: 28: 05, 796 DEBUG SQL: 346insert into
ssh.employee（name, age）values（? , ? ）
    14: 28: 05, 812 DEBUG SQL: 346update ssh.company
set com__name=? where id=?
    14: 28: 05, 828 DEBUG SQL: 346update
ssh.employee set com__id=? where id=?
```

```
    14: 28: 05, 859 DEBUG SQL: 346select company0
__.id as id4__0__, company0__.com__name as com2__4__0
__from
    ssh.company company0__where company0__.id=?
    Company name: QQ.com
    14: 28: 05, 875 DEBUG SQL: 346select employees0
__.com__id as com4__1__, employees0__.id as id1__,
employees0
    __.id as id5__0__, employees0__.name as name5__0
__, employees0__.age as age5__0__from ssh.employee
employees0__
    where employees0__.com__id=?
    Employee name: Employee2
    Employee name: Employee1
    Employee name: Employee4
    14: 28: 05, 875 DEBUG SQL: 346select company0
__.id as id4__0__, company0__.com__name as com2__4__0
__from
    ssh.company company0__where company0__.id=?
    14: 28: 05, 875 DEBUG SQL: 346select employees0
__.com__id as com4__1__, employees0__.id as id1__,
employees0
    __.id as id5__0__, employees0__.name as name5__0
__, employees0__.age as age5__0__from ssh.employee
employees0__
    where employees0__.com__id=?
    14: 28: 05, 890 DEBUG SQL: 346update
ssh.employee set com__id=null where com__id=?
    14: 28: 05, 890 DEBUG SQL: 346delete from
ssh.employee where id=?
    14: 28: 05, 890 DEBUG SQL: 346delete from
ssh.employee where id=?
    14: 28: 05, 890 DEBUG SQL: 346delete from
ssh.employee where id=?
    14: 28: 05, 890 DEBUG SQL: 346delete from
ssh.company where id=?
```

```
    14: 28: 05, 937 DEBUG SQL: 346insert into
ssh.employee（name, age）values（？，？）
```

源程序解读

（1）公司由于包含多个雇员，所以需要建立Set集合类型变量employees，用来对应“多”的一方。

（2）onetomany属性需要指定class属性值，对应“多”的一方。

（3）“多”的一方不需要配置关系。

15.5　双向一对多（多对一）映射

常见的双向一对多或者是多对一映射是父子关系，如图15.4所示。son表通过非空外键father__id连接father表。

图　15.4　双向一对多（多对一）映射

技术要点

双向的一对多（多对一）映射关系是非常常见的映射，是典型的"父子"关系，在数据库中频繁使用，读者必须掌握双向一对多（多对一）映射的配置和使用。本节代码演示如何配置该关系并给出一个简单的

CRUD示例，来学习manytoone和onetomany的联合使用。

实现代码

建立数据库表，语句如下：

```
CREATE TABLE'father'（
′ id'int（11）NOT NULL auto__increment，
′ name'varchar（20）default NULL，
PRIMARY KEY（′ id'）
）ENGINE=InnoDB DEFAULT CHARSET=gbk
CREATE TABLE'son'（
′ id'int（11）NOT NULL auto__increment，
′ name'varchar（20）default NULL，
′ father__id'int（11）default NULL，
PRIMARY KEY（′ id'）
）ENGINE=InnoDB DEFAULT CHARSET=gbk
```

建立father.java，代码如下：

```
package relation.bidirectional.onemany;
import java.util.HashSet;
import java.util.Set;
public class Father implements
java.io.Serializable {
private Integer id; //主键ID属性变量
private String name; //姓名属性变量
```

```
    private Set sons=new HashSet（）；//Set集合属性变量
    Public Father（）｛//构造器
    }
    public Integer getId（）｛//id属性的Getter（）方法和Setter（）方法
    return this.id;
    }
    public void setId（Integer id）｛
    this.id=id;
    }
    public String getName（）｛//name属性的Getter（）方法和Setter（）方法
    return this.name;
    }
    public void setName（String name）｛
    this.name=name;
    }
    public Set getSons（）｛//sons属性的Getter（）方法和Setter（）方法
    return sons;
    }
    public void setSons（Set sons）｛
    this.sons=sons;
    }
    public void addSon（Son son）｛//增加一个son
    sons.add（son）；
    }
    public void removeSon（Son son）｛//删除一个son
    sons.remove（son）；
    }
    }
```

Son. java代码如下：

```
    package relation.bidirectional.onemany;
    public class Son implements
java.io.Serializable {
    private Integer id; //主键ID属性变量
    private String name; //姓名属性变量
    private Father father; //父亲属性变量
    public Son（） {//构造器
    }
    public Integer getId（） {//id属性的Getter（）方法
和Setter（）方法
    return this.id;
    }
    public void setId（Integer id） {
    this.id=id;
    }
    public String getName（） {//name属性的Getter（）
方法和Setter（）方法
    return this.name;
    }
    public void setName（String name） {
    this.name=name;
    }
    public Father getFather（） {//father属性的
Getter（）方法和Setter（）方法
    return father;
    }
    public void setFather（Father father） {
    this.father=father;
    }
    }
```

映射文件Father.hbm.xml内容如下：

```
    <? xml version="1.0"encoding="utf8"? >
```

```
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
    <class
name="relation.bidirectional.onemany.Father"table=
"father"
    catalog="ssh">
    <id name="id"type="java.lang.Integer">
    <column name="id"/>
    <generator class="native"/>
    </id><! 配置主键映射>
    <property name="name"type="java.lang.String">
    <column name="name"length="20"/>
    </property><! 配置姓名映射>
    <set
name="sons"table="son"cascade="all"inverse="true">
    <key column="father__id"></key>
    <! 配置一对多关系映射>
    <onetomany
class="relation.bidirectional.onemany.Son">
</onetomany>
    </set>
    </class>
    </hibernatemapping>
```

映射文件Son.hbm.xml内容如下：

```
    <? xml version="1.0"encoding="utf8"? >
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
```

```
    <hibernatemapping>
    <class
name="relation.bidirectional.onemany.Son"table="so
n"catalog="ssh">
    <id name="id"type="java.lang.Integer">
    <column name="id"/>
    <generator class="native"/>
    </id><! 配置主键映射>
    <property name="name"type="java.lang.String">
    <column name="name"length="20"/>
    </property><! 配置姓名映射>
    <! 配置多对一关系映射>
    <manytoone name="father"column="father__id"
    cascade="saveupdate"outerjoin="true">
    </manytoone>
    </class>
    </hibernatemapping>
```

建立一个测试类test.java，代码如下：

```
    package relation.bidirectional.onemany;
    import java.util.Iterator;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.Transaction;
    import org.hibernate.cfg.Configuration;
    public class Test {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（）.configure（）;
    //根据Configuration建立SessionFactory
    //SessionFactory用来建立Session
```

```
    SessionFactory
sessionFactory=config.buildSessionFactory（）；
    Father f1=new Father（）；//创建father实例
    f1.setName（"father1"）；//对象赋值
    Son s1=new Son（）；
    s1.setName（"son1"）；
    Son s2=new Son（）；
    s2.setName（"son2"）；//对象赋值
    f1.addSon（s1）；//对象赋值
    f1.addSon（s2）；
    s1.setFather（f1）；
    s2.setFather（f1）；
    Integer pid；//定义主键变量
    Session
session=sessionFactory.openSession（）；//添加数据
    Transaction tx=null;
    try｛//开始事务
    tx=session.beginTransaction（）；
    //创建主键变量
    pid=（Integer）session.save（f1）；
    //session.save（s2）；
    tx.commit（）；
    } catch（RuntimeException e）｛
    if（tx! =null）
    tx.rollback（）；
    throw e;
    } finally｛
    session.close（）；
    }
    Son s3=new Son（）；//修改数据
    s3.setName（"son3"）；
    session=sessionFactory.openSession（）；
    tx=null;
    try｛//开始事务
    tx=session.beginTransaction（）；
    f1=（Father）session.get（Father.class, pid）；//
创建主键变量
```

```
    f1.addSon（s3）;
    s3.setFather（f1）;
    session.update（f1）;
    tx.commit（）;
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）;
    throw e;
    } finally {
    session.close（）;
    }
    Son s4=new Son（）; //反向添加数据
    s4.setName（"son4"）;
    session=sessionFactory.openSession（）;
    tx=null;
    try {//开始事务
    tx=session.beginTransaction（）;
    f1=（Father）session.get（Father.class, pid）;
    f1.setName（"father1 updated"）;
    s4.setFather（f1）;
    session.save（s4）;
    tx.commit（）;
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）;
    throw e;
    } finally {
    session.close（）;
    }
    session=sessionFactory.openSession（）; //查询数
据
    f1=（Father）session.get（Father.class, pid）;
    System.out.println（"father
name: "+f1.getName（））;
    System.out.println（"son: "）;
    Iterator iter=f1.getSons（）.iterator（）;
    while（iter.hasNext（）） {
```

```
    s1=（Son）iter.next（）；
    System.out.println（"Employee
name: "+s1.getName（））；
    }
    session.close（）；
    session=sessionFactory.openSession（）；//删除数
据
    tx=null；//开始事务
    try {
    tx=session.beginTransaction（）；
    f1=（Father）session.get（Father.class, pid）；
    session.delete（f1）；
    tx.commit（）；
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）；
    throw e;
    } finally {
    session.close（）；
    }
    //关闭sessionFactory
    sessionFactory.close（）；
    }
```

运行该测试类，结果如下：

```
    insert into ssh.father（name）values（？）
    15: 04: 40，406 DEBUG SQL: 346insert into
ssh.son（name, father__id）values（？，？）
    15: 04: 40，421 DEBUG SQL: 346insert into
ssh.son（name, father__id）values（？，？）
    15: 04: 40，531 DEBUG SQL: 346select father0__.id
as id7__0__，father0__.name as name7__0__from
ssh.father
```

```
    father0__where father0__.id=?
    15: 04: 40, 546 DEBUG SQL: 346select sons0
__.father__id as father3__1__, sons0__.id as id1__,
sons0__.id as id6__0
    __, sons0__.name as name6__0__, sons0__.father__id
as father3__6__0__from ssh.son sons0__where sons0
__.father__
    id=?
    15: 04: 40, 562 DEBUG SQL: 346insert into
ssh.son（name, father__id）values（? , ? ）
    15: 04: 40, 609 DEBUG SQL: 346select father0__.id
as id7__0__, father0__.name as name7__0__from
ssh.father
    father0__where father0__.id=?
    15: 04: 40, 609 DEBUG SQL: 346insert into
ssh.son（name, father__id）values（? , ? ）
    15: 04: 40, 625 DEBUG SQL: 346update ssh.father
set name=? where id=?
    15: 04: 40, 671 DEBUG SQL: 346select father0__.id
as id7__0__, father0__.name as name7__0__from
ssh.father
    father0__where father0__.id=?
    father name: father1 updated
    son:
    15: 04: 40, 687 DEBUG SQL: 346select sons0
__.father__id as father3__1__, sons0__.id as id1__,
sons0__.id as id6__0
    __, sons0__.name as name6__0__, sons0__.father__id
as father3__6__0__from ssh.son sons0__where sons0
__.father__
    id=?
    Employee name: son4
    Employee name: son1
    Employee name: son3
    Employee name: son2
    15: 04: 40, 703 DEBUG SQL: 346select father0__.id
as id7__0__, father0__.name as name7__0__from
```

```
ssh.father
    father0__where father0__.id=?
    15: 04: 40, 703 DEBUG SQL: 346select sons0
__.father__id as father3__1__, sons0__.id as id1__,
sons0__.id as id6__0
    __, sons0__.name as name6__0__, sons0__.father__id
as father3__6__0__from ssh.son sons0__where sons0
__.father__
    id=?
    15: 04: 40, 718 DEBUG SQL: 346delete from
ssh.son where id=?
    15: 04: 40, 718 DEBUG SQL: 346delete from
ssh.son where id=?
    15: 04: 40, 718 DEBUG SQL: 346delete from
ssh.son where id=?
    15: 04: 40, 718 DEBUG SQL: 346delete from
ssh.son where id=?
    15: 04: 40, 718 DEBUG SQL: 346delete from
ssh.father where id=?
```

源程序解读

（1）father类中必须有集合类变量，对应“多”的一方。

（2）Son类中定义father类型变量，对应“一”的一方。

（3）需要在映射文件中分别配置manytoone和onetomany元素。

（4）维护关系的一方通常由“多”的一方来负责，这样效率比较高。本实例中，father的映射文件中inverse=“true”，即将维护关系交给了son。

（5）从控制台SQL日志可以看到，由于没有在father中配置outerjoin，所以在查询father1数据中，实际执行了两条SQL来获得数据，如果加入了outerjoin=true，则只需要一条SQL即可。

15.6　基于外键的单向一对一映射

基于外键的单向一对一映射，常见的例子是用户（user）和电子邮件（email），每个用户只有一个电子邮件，并通过外键email__id同电子邮件相对应。但是电子邮件可以没有对应的用户，如图15.5所示。

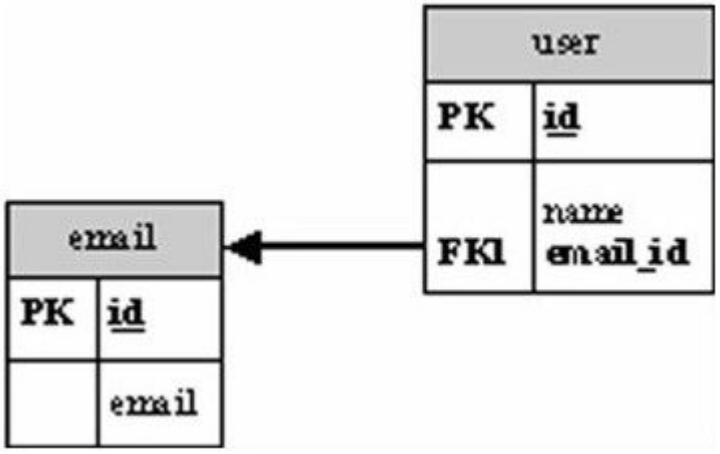

图　15.5　基于外键的单向一对一映射

技术要点

基于外键的单向一对一关系在数据库应用中也是比较重要，但是项目中使用最多的是双向一对一映

射。本节演示如何配置基于外键的单向一对一映射。

使用manytoone配置一对一映射。

理解unique含义。

实现代码

建立数据库表，代码如下：

```
CREATE TABLE'email'（
′ id'int（11）NOT NULL auto__increment，
′ email'varchar（50）default NULL，
PRIMARY KEY（′ id'）
）ENGINE=InnoDB DEFAULT CHARSET=gbk
CREATE TABLE'user'（
′ id'int（11）NOT NULL auto__increment，
′ name'varchar（20）default NULL，
′ email__id'int（11）default NULL，
PRIMARY KEY（′ id'）
）ENGINE=InnoDB DEFAULT CHARSET=gbk
```

建立User.java，代码如下：

```
package relation.unidirectional.onetoone;
public class User implements
java.io.Serializable {
```

```
    private Integer id; //主键ID属性变量
    private String name; //姓名属性变量
    private Email email; //邮件属性变量
    public User（） {//构造器
    }
    public Integer getId（） {//id属性的Getter（）方法
和Setter（）方法
    return this.id;
    }
    public void setId（Integer id） {
    this.id=id;
    }
    public String getName（） {//name属性的Getter（）
方法和Setter（）方法
    return this.name;
    }
    public void setName（String name） {
    this.name=name;
    }
    public Email getEmail（） {//email属性的Getter（）
方法和Setter（）方法
    return email;
    }
    public void setEmail（Email email） {
    this.email=email;
    }
    }
```

建立Email.java，代码如下：

```
    package relation.unidirectional.onetoone;
    public class Email implements
java.io.Serializable {
    private Integer id; //主键ID属性变量
    private String email; //邮件属性变量
    public Email（） {//构造器
    }
    public Integer getId（） {//id属性的Getter（）方法
和Setter（）方法
```

```
    return this.id;
    }
    public void setId（Integer id） {
    this.id=id;
    }
    public String getEmail（） {//email属性的
Getter（）方法和Setter（）方法
    return this.email;
    }
    public void setEmail（String email） {
    this.email=email;
    }
    }
```

映射文件User.hbm.xml内容如下：

```
    <? xml version="1.0"encoding="utf8"? >
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
    <class
name="relation.unidirectional.onetoone.User"table=
"user"catalog="ssh">
    <id name="id"type="java.lang.Integer">
    <column name="id"></column>
    <generator class="native"/>
    </id><! 配置主键映射>
    <property name="name"type="java.lang.String">
    <column name="name"length="20"/>
    </property><! 配置姓名映射>
    <! 配置多对一关联关系>
    <manytoone name="email"column="email__id"
```

```
    class="relation.unidirectional.onetoone.Email"o
uterjoin="true"
    cascade="all"unique="true">
    </manytoone>
    </class>
    </hibernatemapping>
```

映射文件Email.hbm.xml内容如下：

```
    <? xml version="1.0"encoding="utf8"? >
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
    <class
name="relation.unidirectional.onetoone.Email"table
="email"
    catalog="ssh">
    <id name="id"type="java.lang.Integer">
    <column name="id"/>
    <generator class="native"></generator>
    </id><! 配置主键映射>
    <property name="email"type="java.lang.String">
    <column name="email"length="50"/>
    </property><! 配置邮件映射>
    </class>
    </hibernatemapping>
```

增加测试类test.java，代码如下：

```
    package relation.unidirectional.onetoone;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.Transaction;
    import org.hibernate.cfg.Configuration;
    public class Test {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（）.configure（）;
    //根据Configuration建立SessionFactory
    //SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）;
    Email email1=new Email（）; //创建实例
    email1.setEmail（"email1"）;
    Email email2=new Email（）;
    email2.setEmail（"email2"）;
    User u1=new User（）; //创建实例
    u1.setName（"user1"）;
    u1.setEmail（email1）;
    User u2=new User（）;
    u2.setName（"user2"）;
    u2.setEmail（email2）;
    Integer uid; //定义主键变量
    Session
session=sessionFactory.openSession（）; //添加数据
    Transaction tx=null;
    try {
    tx=session.beginTransaction（）;
    uid=（Integer）session.save（u1）; //创建主键变量
    session.save（u2）;
    tx.commit（）;
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）;
    throw e;
```

```
    } finally {
    session.close();
    }
    session=sessionFactory.openSession(); //修改数
据
    tx=null;
    try {
    tx=session.beginTransaction();
    u1=(User) session.get(User.class, uid);
    u1.setName("user1 updated");
    u1.getEmail() .setEmail("email1 updated");
    session.update(u1);
    tx.commit();
    } catch(RuntimeException e) {
    if(tx! =null)
    tx.rollback();
    throw e;
    } finally {
    session.close();
    }
    session=sessionFactory.openSession(); //查询数
据
    u1=(User) session.get(User.class, uid);
    System.out.println("person
name: "+u1.getName());
    System.out.println("Email: "+u1.getEmail() .ge
tEmail());
    session.close();
    session=sessionFactory.openSession(); //删除数
据
    tx=null;
    try {
    tx=session.beginTransaction();
    u1=(User) session.get(User.class, uid);
    session.delete(u1);
    tx.commit();
    } catch(RuntimeException e) {
```

```
    if（tx! =null）
    tx.rollback（）；
    throw e;
    } finally {
    session.close（）；
    }
    session=sessionFactory.openSession（）；//反向增加数据
    tx=null;
    try {
    u1.setName（"反向增加user"）；
    email2.setEmail（"ok@hotmail.com"）；
    tx=session.beginTransaction（）；
    session.save（email2）；
    tx.commit（）；
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）；
    throw e;
    } finally {
    session.close（）；
    }
    //关闭sessionFactory
    sessionFactory.close（）；
    }
    }
```

运行测试类，控制台输出如下内容：

```
    15: 32: 18，265 DEBUG SQL: 346insert into ssh.email（email）values（？）
    15: 32: 18，296 DEBUG SQL: 346insert into ssh.user（name, email__id）values（？，？）
```

```
    15: 32: 18, 296 DEBUG SQL: 346insert into
ssh.email (email) values (? )
    15: 32: 18, 312 DEBUG SQL: 346insert into
ssh.user (name, email__id) values (? , ? )
    15: 32: 18, 453 DEBUG SQL: 346select user0__.id
as id11__1__, user0__.name as name11__1__, user0
__.email__id as
    email3__11__1__, email1__.id as id10__0__, email1
__.email as email10__0__from ssh.user user0__left
outer join
    ssh.email email1__on user0__.email__id=email1
__.id where user0__.id=?
    15: 32: 18, 468 DEBUG SQL: 346update ssh.email
set email=? where id=?
    15: 32: 18, 484 DEBUG SQL: 346update ssh.user
set name=? , email__id=? where id=?
    15: 32: 18, 531 DEBUG SQL: 346select user0__.id
as id11__1__, user0__.name as name11__1__, user0
__.email__id as
    email3__11__1__, email1__.id as id10__0__, email1
__.email as email10__0__from ssh.user user0__left
outer join
    ssh.email email1__on user0__.email__id=email1
__.id where user0__.id=?
    person name: user1 updated
    Email: email1 updated
    15: 32: 18, 531 DEBUG SQL: 346select user0__.id
as id11__1__, user0__.name as name11__1__, user0
__.email__id as
    email3__11__1__, email1__.id as id10__0__, email1
__.email as email10__0__from ssh.user user0__left
outer join
    ssh.email email1__on user0__.email__id=email1
__.id where user0__.id=?
    15: 32: 18, 546 DEBUG SQL: 346delete from
ssh.user where id=?
```

```
    15: 32: 18，546 DEBUG SQL: 346delete from
ssh.email where id=?
    15: 32: 18，609 DEBUG SQL: 346insert into
ssh.email（email）values（？）
```

源程序解读

（1）由于是单向映射，User中需要建立一个email类型变量则email中不需要。

（2）使用manytoone配置一对一映射，需要定义unique=“true”，即不能有重复的主键，就限定了“多”的一方必须变成“一”，这样实现了一对一映射。

（3）单向的一对一映射，只需要配置一方即可。

15.7　基于外键的双向一对一映射

修改前一节单向映射为双向映射。在保持数据库表结构不变的情况下，增加相应的配置信息，实现双向映射。例如，用户表和电子邮件就是双向一对一的映射关系，即每个用户都有唯一的电子邮件对应，例如常见的电子商务网站淘宝和ebay等，一对一可以通过主键对应，也可以通过外键来对应。

技术要点

在数据库应用项目中，基于外键的双向一对一关系非常常见。本节代码演示如何实现基于外键的双向一对一映射。

onetoone的配置。

propertyref属性。

实现代码

修改Email.java，代码如下：

```
    package relation.bidirectional.onetoone;
    public class Email implements
java.io.Serializable {
    private Integer id; //主键ID属性变量
    private String email; //邮件属性变量
    private User user; //用户属性变量
    public Email（） {//构造器
    }
    public Integer getId（） {//id属性的Getter（）方法
和Setter（）方法
    return this.id;
    }
    public void setId（Integer id） {
    this.id=id;
    }
    public String getEmail（） {//email属性的
Getter（）方法和Setter（）方法
    return this.email;
    }
    public void setEmail（String email） {
    this.email=email;
    }
    public User getUser（） {//user属性的Getter（）方
法和Setter（）方法
    return user;
    }
    public void setUser（User user） {
    this.user=user;
    }
    }
```

映射文件User.hbm.xml内容：

```
    <? xml version="1.0"encoding="utf8"? >
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
    <class
name="relation.bidirectional.onetoone.User"table="
user"catalog="ssh">
    <id name="id"type="java.lang.Integer">
    <column name="id"></column>
    <generator class="native"/>
    </id><! 配置主键映射>
    <property name="name"type="java.lang.String">
    <column name="name"length="20"/>
    </property><! 配置姓名映射>
    <! 配置一对一关联关系>
    <manytoone name="email"column="email__
id"unique="true"
    cascade="all">
    </manytoone>
    </class>
    </hibernatemapping>
```

配置文件Email.hbm.xml内容如下：

```
    <? xml version="1.0"encoding="utf8"? >
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
```

```
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
    <class
name="relation.bidirectional.onetoone.Email"table=
"email"
    catalog="ssh">
    <id name="id"type="java.lang.Integer">
    <column name="id"/>
    <generator class="native"></generator>
    </id><! 配置主键映射>
    <property name="email"type="java.lang.String">
    <column name="email"length="50"/>
    </property><! 配置邮件映射>
    <! 配置一对一关系映射>
    <onetoone
name="user"propertyref="email"cascade="all">
</onetoone>
    </class>
    </hibernatemapping>
```

增加测试类test.java，代码如下：

```
    package relation.bidirectional.onetoone;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.Transaction;
    import org.hibernate.cfg.Configuration;
    public class Test {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（）.configure（）;
    //根据Configuration建立SessionFactory
```

```
    //SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）；
    Email email1=new Email（）；//创建实例
    email1.setEmail（"email1"）；
    Email email2=new Email（）；//建立对象
    email2.setEmail（"email2"）；
    User u1=new User（）；//对象赋值
    u1.setName（"user1"）；
    u1.setEmail（email1）；
    User u2=new User（）；//建立对象
    u2.setName（"user2"）；
    u2.setEmail（email2）；
    email1.setUser（u1）；//对象赋值
    email2.setUser（u2）；
    Integer uid；//定义主键变量
    Session
session=sessionFactory.openSession（）；//添加数据
    Transaction tx=null；//开始事务
    try {
    tx=session.beginTransaction（）；
    uid=（Integer）session.save（u1）；//创建主键变量
    session.save（u2）；
    tx.commit（）；
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）；
    throw e；
    } finally {
    session.close（）；
    }
    session=sessionFactory.openSession（）；//修改数
据
    tx=null；//开始事务
    try {
    tx=session.beginTransaction（）；
    u1=（User）session.get（User.class, uid）；
```

```
    u1.setName（"pla"）;
    u1.getEmail（）.setEmail（"yansz@126.com"）;
    session.update（u1）;
    tx.commit（）;
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）;
    throw e;
    } finally {
    session.close（）;
    }
    session=sessionFactory.openSession（）; //查询数
据
    u1=（User）session.get（User.class, uid）;
    System.out.println（"person
name: "+u1.getName（））;
    System.out.println（"Email: "+u1.getEmail（）.ge
tEmail（））;
    session.close（）;
    session=sessionFactory.openSession（）; //删除数
据
    tx=null;
    try {//开始事务
    tx=session.beginTransaction（）;
    u1=（User）session.get（User.class, uid）;
    session.delete（u1）;
    tx.commit（）;
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）;
    throw e;
    } finally {
    session.close（）;
    }
    Email email3=new Email（）; //反向增加数据
    email3.setEmail（"email3"）;
    User u3=new User（）;
```

```
u3.setName（"user3"）;
u3.setEmail（email3）;
email3.setUser（u3）;
session=sessionFactory.openSession（）;
tx=null; //开始事务
try {
tx=session.beginTransaction（）;
session.save（email3）;
tx.commit（）;
} catch（RuntimeException e） {
if（tx! =null）
tx.rollback（）;
throw e;
} finally {
session.close（）;
}
sessionFactory.close（）; //关闭sessionFactory
}
}
```

运行测试类，结果如下：

```
15: 58: 07, 187 DEBUG SQL: 346insert into
ssh.email（email）values（? ）
15: 58: 07, 218 DEBUG SQL: 346insert into
ssh.user（name, email__id）values（? , ? ）
15: 58: 07, 218 DEBUG SQL: 346insert into
ssh.email（email）values（? ）
15: 58: 07, 234 DEBUG SQL: 346insert into
ssh.user（name, email__id）values（? , ? ）
15: 58: 07, 328 DEBUG SQL: 346select user0__.id
as id14__0__, user0__.name as name14__0__, user0
__.email__id as
```

```
    email3__14__0__from ssh.user user0__where user0
__.id=?
    15: 58: 07, 375 DEBUG SQL: 346select email0__.id
as id13__1__, email0__.email as email13__1__, user1
__.id as
    id14__0__, user1__.name as name14__0__, user1
__.email__id as email3__14__0__from ssh.email email0
__left outer join
    ssh.user user1__on email0__.id=user1__.email__id
where email0__.id=?
    15: 58: 07, 375 DEBUG SQL: 346select user0__.id
as id14__0__, user0__.name as name14__0__, user0
__.email__id as
    email3__14__0__from ssh.user user0__where user0
__.email__id=?
    15: 58: 07, 390 DEBUG SQL: 346update ssh.user
set name=? , email__id=? where id=?
    15: 58: 07, 406 DEBUG SQL: 346update ssh.email
set email=? where id=?
    15: 58: 07, 453 DEBUG SQL: 346select user0__.id
as id14__0__, user0__.name as name14__0__, user0
__.email__id as
    email3__14__0__from ssh.user user0__where user0
__.id=?
    person name: pla
    15: 58: 07, 468 DEBUG SQL: 346select email0__.id
as id13__1__, email0__.email as email13__1__, user1
__.id as
    id14__0__, user1__.name as name14__0__, user1
__.email__id as email3__14__0__from ssh.email email0
__left outer join
    ssh.user user1__on email0__.id=user1__.email__id
where email0__.id=?
    15: 58: 07, 468 DEBUG SQL: 346select user0__.id
as id14__0__, user0__.name as name14__0__, user0
__.email__id as
```

```
    email3__14__0__from ssh.user user0__where user0
__.email__id=?
    Email: yansz@126.com
    15: 58: 07, 468 DEBUG SQL: 346select user0__.id
as id14__0__, user0__.name as name14__0__, user0
__.email__id as
    email3__14__0__from ssh.user user0__where user0
__.id=?
    15: 58: 07, 484 DEBUG SQL: 346select email0__.id
as id13__1__, email0__.email as email13__1__, user1
__.id as
    id14__0__, user1__.name as name14__0__, user1
__.email__id as email3__14__0__from ssh.email email0
__left outer join
    ssh.user user1__on email0__.id=user1__.email__id
where email0__.id=?
    15: 58: 07, 484 DEBUG SQL: 346select user0__.id
as id14__0__, user0__.name as name14__0__, user0
__.email__id as
    email3__14__0__from ssh.user user0__where user0
__.email__id=?
    15: 58: 07, 484 DEBUG SQL: 346delete from
ssh.user where id=?
    15: 58: 07, 500 DEBUG SQL: 346delete from
ssh.email where id=?
    15: 58: 07, 593 DEBUG SQL: 346insert into
ssh.email（email）values（? ）
    15: 58: 07, 593 DEBUG SQL: 346insert into
ssh.user（name, email__id）values（? , ? ）
```

源程序解读

（1）双方都需要有对方类型的变量属性。

（2）Email配置文件使用了onetoone配置一对一映射。

（3）onetoone配置有如下属性。

name：属性的名字。

class（可选，默认是通过反射得到的属性类型）：被关联的类的名字。

cascade（级联，可选）：表明操作是否从父对象级联到被关联的对象。

constrained（约束，可选）：表明该类对应的表对应的数据库表，和被关联的对象所对应的数据库表之间，通过一个外键引用对主键进行约束。这个选项影响save（）和delete（）在级联执行时的先后顺序（也在schema export tool中被使用）。

outerjoin（外连接，可选，默认为自动）：当设置hibernate.use__outer__join时，对这个关联允许外连接抓取。

propertyref（可选）：指定关联类的一个属性，这个属性将会和本外键相对应。如果没有指定，会使用对方关联类的主键。

access（可选，默认是property）：Hibernate用来访问属性的策略。

（4）本实例propertyref="email"，即关联到User类中的email变量。

15.8　基于主键的单向一对一映射

一对一映射，前面介绍的是基于外键的，还有一种根据主键的一对一映射，即两个表的主键值相同。单向一对一的情况不多，例如用户和电子邮件之间的关系，假定没有用户必须有唯一的电子邮件与之对应，而电子邮件可以没有对应的用户，即对应关系是单向的，如图15.6所示。

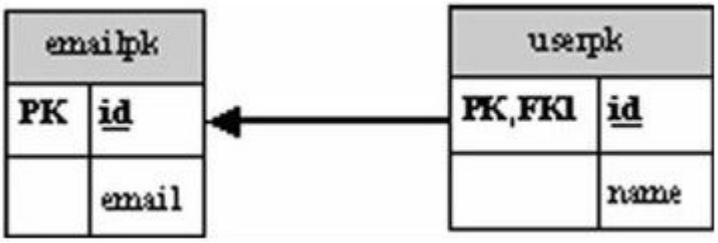

图　15.6　基于主键的一对一映射

技术要点

基于主键的一对一映射特点是两个表都以主键作为关联字段，这同基于外键的一对一映射有很大的不

同，即关联字段都是各自的主键。本节代码演示如何配置基于主键的一对一映射，理解onetoone的使用。

实现代码

建立相关表格，语句如下：

```
CREATE TABLE'userpk'（
′ id'int（11）NOT NULL auto__increment，
′ name'varchar（20）default NULL，
PRIMARY KEY（′ id'）
）ENGINE=InnoDB DEFAULT CHARSET=gbk
CREATE TABLE'emailpk'（
′ id'int（11）NOT NULL auto__increment，
′ email'varchar（20）default NULL，
PRIMARY KEY（′ id'）
）ENGINE=InnoDB DEFAULT CHARSET=gbk
```

建立控制方Emailpk.java，代码如下：

```
package relation.unidirectional.onetoonepk;
public class Emailpk implements
java.io.Serializable {
private Integer id; //主键ID属性变量
private String email; //邮件属性变量
private Userpk userpk; //用户主键类属性变量
public Emailpk（） {//构造器
}
```

```
    public Integer getId（） {//id属性的Getter（）方法
和Setter（）方法
    return this.id;
    }
    public void setId（Integer id） {
    this.id=id;
    }
    public String getEmail（） {//email属性的
Getter（）方法和Setter（）方法
    return this.email;
    }
    public void setEmail（String email） {
    this.email=email;
    }
    public Userpk getUserpk（） {//userpk属性的
Getter（）方法和Setter（）方法
    return userpk;
    }
    public void setUserpk（Userpk userpk） {
    this.userpk=userpk;
    }
    }
```

建立被控制方Userpk.java，代码如下：

```
    package relation.unidirectional.onetoonepk;
    public class Userpk implements
java.io.Serializable {
    private Integer id; //主键属性变量
    private String name; //姓名属性变量
    private Emailpk emailpk; //邮件类主键变量
    public Userpk（） {//构造器
    }
```

```
    public Integer getId（） {//id属性的Getter（）方法和Setter（）方法
    return this.id;
    }
    public void setId（Integer id） {
    this.id=id;
    }
    public String getName（） {//name属性的Getter（）方法和Setter（）方法
    return this.name;
    }
    public void setName（String name） {
    this.name=name;
    }
    public Emailpk getEmailpk（） {//emailpk属性的Getter（）方法和Setter（）方法
    return emailpk;
    }
    public void setEmailpk（Emailpk emailpk） {
    this.emailpk=emailpk;
    }
    }
```

映射文件Emailpk.hbm.xml内容如下：

```
    ＜? xml version="1.0"encoding="utf8"? ＞
    ＜! DOCTYPE hibernatemapping PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemapping3.0.dtd"＞
    ＜hibernatemapping＞
    ＜class name="relation.unidirectional.onetoonepk.Emailpk"table="emailpk"
```

```
    catalog="ssh">
    <id name="id"type="java.lang.Integer">
    <column name="id"/>
    <generator class="foreign">
    <param name="property">userpk</param>
    </generator>
    </id><!配置主键映射，设置主键关联>
    <property name="email"type="java.lang.String">
    <column name="email"length="20"/>
    </property>
    <!配置一对一关系映射>
    <onetoone name="userpk"
    class="relation.unidirectional.onetoonepk.Userp
k"constrained="true"
    cascade="all"/>
    </class>
    </hibernatemapping>
```

映射文件Userpk.hbm.xml内容如下：

```
    <?xml version="1.0"encoding="utf8"? >
    <!DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
    <class
name="relation.unidirectional.onetoonepk.Userpk"ta
ble="userpk"catalog="ssh">
    <id name="id"type="java.lang.Integer">
    <column name="id"/>
    <generator class="native"/>
    </id><!配置主键映射>
    <property name="name"type="java.lang.String">
```

```
    <column name="name"length="20"/>
    </property><! 配置姓名映射>
    </class>
    </hibernatemapping>
```

将映射文件添加到Hibernate配置文件，建立测试类test.java，代码如下：

```
    package relation.unidirectional.onetoonepk;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.Transaction;
    import org.hibernate.cfg.Configuration;
    public class Test {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（）.configure（）;
    //根据Configuration建立SessionFactory
    //SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）;
    Emailpk email1=new Emailpk（）; //创建实例
    email1.setEmail（"email1"）;
    Emailpk email2=new Emailpk（）; //建立对象
    email2.setEmail（"email2"）;
    Userpk u1=new Userpk（）; //对象赋值
    u1.setName（"user1"）;
    Userpk u2=new Userpk（）; //对象赋值
    u2.setName（"user2"）;
    u1.setEmailpk（email1）; //赋值
    u2.setEmailpk（email2）;
    email1.setUserpk（u1）;
```

```
    email2.setUserpk（u2）;
    Integer uid; //定义主键变量
    Session
session=sessionFactory.openSession（）; //添加数据
    Transaction tx=null;
    try {//开始事务
    tx=session.beginTransaction（）;
    uid=（Integer）session.save（email1）; //创建主键
变量
    session.save（email2）;
    tx.commit（）;
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）;
    throw e;
    } finally {
    session.close（）;
    }
    session=sessionFactory.openSession（）; //修改数
据
    tx=null;
    try {//开始事务
    tx=session.beginTransaction（）;
    email1=（Emailpk）session.get（Emailpk.class,
uid）;
    email1.setEmail（"email1 update"）;
    email1.getUserpk（）.setName（"user1 updated"）;
    session.update（email1）;
    tx.commit（）;
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）;
    throw e;
    } finally {
    session.close（）;
    }
```

```
    session=sessionFactory.openSession（）；//查询数
据
    email1=（Emailpk）session.get（Emailpk.class,
uid）；
    System.out.println（"Email: "+email1.getEmail（
））；
    System.out.println（"person
name: "+email1.getUserpk（）.getName（））；
    session.close（）；
    session=sessionFactory.openSession（）；//删除数
据
    tx=null；//开始事务
    try {
    tx=session.beginTransaction（）；
    email1=（Emailpk）session.get（Emailpk.class,
uid）；
    session.delete（email1）；
    tx.commit（）；
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）；
    throw e;
    } finally {
    session.close（）；
    }
    session=sessionFactory.openSession（）；//反向增
加数据
    tx=null；//开始事务
    try {
    tx=session.beginTransaction（）；
    session.save（u2）；
    tx.commit（）；
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）；
    throw e;
    } finally {
```

```
    session.close（）；
    }
    //关闭sessionFactory
    sessionFactory.close（）；
    }
    }
```

运行测试类，结果如下：

```
    08: 45: 25, 437 DEBUG SQL: 346insert into
ssh.userpk（name）values（？）
    08: 45: 25, 484 DEBUG SQL: 346insert into
ssh.emailpk（email, id）values（？, ？）
    08: 45: 25, 484 DEBUG SQL: 346insert into
ssh.userpk（name）values（？）
    08: 45: 25, 500 DEBUG SQL: 346insert into
ssh.emailpk（email, id）values（？, ？）
    08: 45: 25, 734 DEBUG SQL: 346select emailpk0
__.id as id8__0__, emailpk0__.email as email8__0__
from
    ssh. emailpk emailpk0__where emailpk0__.id=?
    08: 45: 25, 765 DEBUG SQL: 346select userpk0__.id
as id9__0__, userpk0__.name as name9__0__from
ssh.userpk
    userpk0__where userpk0__. id=?
    08: 45: 25, 781 DEBUG SQL: 346update ssh.emailpk
set email=? where id=?
    08: 45: 25, 781 DEBUG SQL: 346update ssh.userpk
set name=? where id=?
    08: 45: 25, 828 DEBUG SQL: 346select emailpk0
__.id as id8__0__, emailpk0__.email as email8__0__
from
    ssh.emailpk emailpk0__where emailpk0__.id=?
    Email: email1 update
```

```
    08: 45: 25, 828 DEBUG SQL: 346select userpk0__.id
as id9__0__, userpk0__.name as name9__0__from
ssh.userpk
    userpk0__where userpk0__.id=?
    person name: user1 updated
    08: 45: 25, 828 DEBUG SQL: 346select emailpk0
__.id as id8__0__, emailpk0__.email as email8__0__
from
    ssh.emailpk emailpk0__where emailpk0__.id=?
    08: 45: 25, 843 DEBUG SQL: 346select userpk0__.id
as id9__0__, userpk0__.name as name9__0__from
ssh.userpk
    userpk0__where userpk0__.id=?
    08: 45: 25, 843 DEBUG SQL: 346delete from
ssh.emailpk where id=?
    08: 45: 25, 843 DEBUG SQL: 346delete from
ssh.userpk where id=?
    08: 45: 25, 875 DEBUG SQL: 346insert into
ssh.userpk（name）values（? ）
```

源程序解读

（1）使用onetoone配置基于主键的一对一映射。

（2）generator class=“foreign”意味着Emailpk使用外部主键，＜param name=“property”＞userpk＜/param＞配置了Emailpk使用Userpk的主键

作为自己的主键，这样就实现了基于主键的一对一映射。

（3）constrained=“true”说明userpk的主键值存在一个约束，即emailpk使用了userpk的主键。

15.9　基于主键的双向一对一映射

前面介绍了基于主键的单向一对一映射，本节介绍基于主键的双向一对一映射，仍然使用前面的数据库表和相关类。例如用户和电子邮件之间的关系，每个用户有唯一的电子邮件，每个电子邮件有唯一的用户，假如这种对应是通过各自的主键实现的，那么就是基于主键的双向一对一映射关系。

技术要点

前面小结介绍了基于主键的单向一对一映射，本节介绍基于主键的双向一对一映射，使读者理解如何实现基于主键的双向一对一映射配置。

实现代码

映射文件Emailpk.hbm.xml内容如下：

```
    <? xml version="1.0"encoding="utf8"? >
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
    <class name="relation.bidirectional.onetoone__
pk.Emailpk"table="emailpk"catalog="ssh">
    <id name="id"type="java.lang.Integer">
    <column name="id"/>
    <generator class="foreign">
    <param name="property">userpk</param>
</generator>
    </id><! 配置主键映射>
    <property name="email"type="java.lang.String">
    <column name="email"length="20"/>
    </property>
    <! 配置一对一关系映射>
    <onetoone name="userpk"
    class="relation.bidirectional.onetoone__
pk.Userpk"
    constrained="true"/>
    </class>
    </hibernatemapping>
```

映射文件Userpk.hbm.xml内容如下：

```
    <? xml version="1.0"encoding="utf8"? >
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
```

```
    <class name="relation.bidirectional.onetoone__
pk.Userpk"table="userpk"
    catalog="ssh">
    <id name="id"type="java.lang.Integer">
    <column name="id"/>
    <generator class="native"/>
    </id><! 配置主键映射>
    <property name="name"type="java.lang.String">
    <column name="name"length="20"/>
    </property>
    <! 配置一对一关系映射>
    <onetoone
name="emailpk"class="relation.bidirectional.onetoo
ne__pk.Emailpk"
    cascade="all"/>
    </class>
    </hibernatemapping>
```

将上面的映射文件替换到Hibernate配置文件中，增加测试类test.java，代码如下：

```
    package relation.bidirectional.onetoone__pk;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.Transaction;
    import org.hibernate.cfg.Configuration;
    public class Test {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（）.configure（）;
    //根据Configuration建立SessionFactory
    //SessionFactory用来建立Session
```

```
    SessionFactory
sessionFactory=config.buildSessionFactory（）；
    Emailpk email1=new Emailpk（）；//相互设置属性
    Userpk u1=new Userpk（）；
    u1.setName（"user1"）；
    u1.setEmailpk（email1）；
    email1.setEmail（"email2"）；
    email1.setUserpk（u1）；
    Emailpk email2=new Emailpk（）；
    Userpk u2=new Userpk（）；
    u2.setName（"user2"）；
    u2.setEmailpk（email2）；
    email2.setEmail（"email2"）；
    email2.setUserpk（u2）；
    Integer uid；//定义主键变量
    Session
session=sessionFactory.openSession（）；//添加数据
    Transaction tx=null;
    try {
    tx=session.beginTransaction（）；
    uid=（Integer）session.save（u1）；//创建主键变量
    session.save（u2）；
    tx.commit（）；
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）；
    throw e;
    } finally {
    session.close（）；
    }
    session=sessionFactory.openSession（）；//修改数
据
    tx=null;
    try {
    tx=session.beginTransaction（）；
    email1=（Emailpk）session.get（Emailpk.class,
uid）；
```

```
    email1.setEmail（"email1 update"）;
    email1.getUserpk（）.setName（"user1 updated"）;
    session.update（email1）;
    tx.commit（）;
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）;
    throw e;
    } finally {
    session.close（）;
    }
    session=sessionFactory.openSession（）; //查询数
据
    u1=（Userpk）session.get（Userpk.class, uid）;
    System.out.println（"username: "+u1.getName（）
+"email: "+u1.getEmailpk（）.getEmail（））;
    session.close（）;
    session=sessionFactory.openSession（）; //删除数
据
    tx=null;
    try {
    tx=session.beginTransaction（）;
    //创建主键变量
    session.delete（u1）;
    tx.commit（）;
    } catch（RuntimeException e） {
    if（tx! =null）
    tx.rollback（）;
    throw e;
    } finally {
    session.close（）;
    }
    Emailpk email3=new Emailpk（）; //反向添加数据
    email3.setEmail（"3.com"）;
    Userpk u3=new Userpk（）;
    u3.setName（"u3"）;
    u3.setEmailpk（email3）;
```

```
email3.setUserpk（u3）;
session=sessionFactory.openSession（）;
tx=null;
try {
tx=session.beginTransaction（）;
session.save（email3）;
tx.commit（）;
} catch（RuntimeException e） {
if（tx! =null）
tx.rollback（）;
throw e;
} finally {
session.close（）;
}
sessionFactory.close（）; //关闭sessionFactory
}
}
```

运行测试类，结果如下：

```
    09: 12: 36, 296 DEBUG SQL: 346insert into
ssh.userpk（name）values（? ）
    09: 12: 36, 343 DEBUG SQL: 346insert into
ssh.emailpk（email, id）values（? , ? ）
    09: 12: 36, 343 DEBUG SQL: 346insert into
ssh.userpk（name）values（? ）
    09: 12: 36, 359 DEBUG SQL: 346insert into
ssh.emailpk（email, id）values（? , ? ）
    09: 12: 36, 531 DEBUG SQL: 346select emailpk0
__.id as id13__0__, emailpk0__.email as email13__0__
from
    ssh.emailpk emailpk0__where emailpk0__.id=?
    09: 12: 36, 578 DEBUG SQL: 346select userpk0__.id
as id14__1__, userpk0__.name as name14__1__, emailpk1
```

```
__.id
    as id13__0__, emailpk1__.email as email13__0__
from ssh.userpk userpk0__left outer join
ssh.emailpk emailpk1__on
    userpk0__.id=emailpk1__.id where userpk0__.id=?
    09: 12: 36, 578 DEBUG SQL: 346update ssh.emailpk
set email=? where id=?
    09: 12: 36, 593 DEBUG SQL: 346update ssh.userpk
set name=? where id=?
    09: 12: 36, 656 DEBUG SQL: 346select userpk0__.id
as id14__1__, userpk0__.name as name14__1__, emailpk1
__.id
    as id13__0__, emailpk1__.email as email13__0__
from ssh.userpk userpk0__left outer join
ssh.emailpk emailpk1__on
    userpk0__.id=emailpk1__.id where userpk0__.id=?
    username: user1 updated email: email1 update
    09: 12: 36, 671 DEBUG SQL: 346delete from
ssh.emailpk where id=?
    09: 12: 36, 671 DEBUG SQL: 346delete from
ssh.userpk where id=?
    09: 12: 36, 703 DEBUG SQL: 346insert into
ssh.userpk（name）values（? ）
    09: 12: 36, 718 DEBUG SQL: 346insert into
ssh.emailpk（email, id）values（? , ? ）
```

可见，实现了基于主键的双向一对一映射。

源程序解读

（1）双方都使用了onetoone配置双向的一对一映射。

（2）注意inverse与cascade的关系，这两个属性本身互不影响，但起的作用有些类似，都能引发对关系表的更新。

inverse只对set+onetomany（或manytomany）有效，对manytoone、onetoone无效。cascade对关系标记都有效。

inverse对集合对象整体起作用，cascade对集合对象中的一个一个元素起作用，如果集合为空，那么cascade不会引发关联操作。

起作用的时机不同：cascade在对主控方操作时，级联发生。inverse在flush时（commit会自动执行flush），对session中的所有set、Hibernate判断每

个set是否有变化，对有变化的set执行相应的sql，执行之前，会有个判断：if（inverse==true）return;可以看出cascade在先，inverse在后。

（3）读者注意同基于外键双向一对一的映射区别。

第16章　Criteria条件查询

Hibernate作为一个重要的数据库/实体映射工具，支持多种方式的查询，本章介绍HibernateAPI支持的Criteria条件查询。包括在Criteria中使用SQL语句、使用DetchedCriteria查询等非常通用的技术。

16.1　简单的Criteria查询

Criteria是Hibernate提供的一组API，这种查询方式把查询条件封装为一个Criteria对象。下面介绍一个简单的示例，就是查询一个用户表的相关信息。

技术要点

Hibernate提供了Criteria条件查询，这是Hibernate基本的查询方式之一，Criteria条件查询提

供了程序员通过类的方法来设定查询条件，同传统的SQL语句有所区别，但是掌握起来并不复杂。本节代码演示如何在Hibernate中使用Criteria条件查询。

熟悉如何创建Criteria对象。

使用Criteria对象返回数据库表记录。

实现代码

首先建立一个对应的用户表，语句如下：

```
CREATE TABLE'q__user'（
′ id'int（11）NOT NULL auto__increment，
′ name'varchar（20）NOT NULL，
′ age'int（5）NOT NULL，
′ email'varchar（100）default NULL，
PRIMARY KEY（′ id'）
）ENGINE=InnoDB DEFAULT CHARSET=gbk
```

建立角色的实体类QUser.java，代码如下：

```
package criteria;
```

```
    public class QUser implements
java.io.Serializable {
    private Integer id; //主键ID属性变量
    private String name; //姓名主键变量
    private Integer age; //年龄属性变量
    private String email; //邮件属性变量
    public QUser（） {//构造器
    }
    public Integer getId（） {//id属性的Getter（）方法
和Setter（）方法
    return this.id;
    }
    public void setId（Integer id） {
    this.id=id;
    }
    public String getName（） {//name属性的Getter（）
方法和Setter（）方法
    return this.name;
    }
    public void setName（String name） {
    this.name=name;
    }
    public Integer getAge（） {//age属性的Getter（）方
法和Setter（）方法
    return this.age;
    }
    public void setAge（Integer age） {
    this.age=age;
    }
    public String getEmail（） {//email属性的
Getter（）方法和Setter（）方法
    return this.email;
    }
    public void setEmail（String email） {
    this.email=email;
    }
    }
```

建立映射文件QUser.hbm.xml，内容如下：

```
    <? xml version="1.0"encoding="utf8"? >
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
    <class name="criteria.QUser"table="q__
user"catalog="ssh">
    <id name="id"type="java.lang.Integer">
    <column name="id"/>
    <generator class="native"/>
    </id><! 配置主键映射>
    <property name="name"type="java.lang.String">
    <column name="name"length="20"notnull="true"/>
    </property><! 配置姓名映射>
    <property name="age"type="java.lang.Integer">
    <column name="age"notnull="true"/>
    </property><! 配置年龄映射>
    <property name="email"type="java.lang.String">
    <column name="email"length="100"/>
    </property><! 配置邮件映射>
    </class>
    </hibernatemapping>
```

将该映射文件加入到Hibernate配置文件中，建立测试类Test__Basic.java，代码如下：

```
    package criteria;
```

```
    import java.util.Iterator;
    import java.util.List;
    import org.hibernate.Criteria;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.cfg.Configuration;
    public class Test__Basic {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（） .configure（）；
    //根据config建立SessionFactory
    //SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）；
    //建立Session，相当于建立JDBC的Connection
    Session session=sessionFactory.openSession（）；
    Criteria
criteria=session.createCriteria（QUser.class）；//创
建Criteria实例
    List users=criteria.list（）；//查询所有的q__user
数据表记录
    System.out.println（"id\t name/age"）；
    Iterator iterator=users.iterator（）；//循环访问
List元素
    while（iterator.hasNext（）） {
    QUser user=（QUser）iterator.next（）；
    System.out.println（user.getId（）+"\
t"+user.getName（）+"/"
    +user.getAge（））；
    }
    session.close（）；
    sessionFactory.close（）；
    }
    }
```

运行该测试类，结果如下：

```
    select this__.id as id0__0__, this__.name as
name0__0__, this__.age as age0__0__, this__.email as
email0__0__from ssh.q
    __user this__
    idname/age
    1张三/22
    2 pla/33
    3李四/18
    4闫术卓/30
    5杨强/18
```

源程序解读

（1）首先使用session的createCriteria（QUser.class）建立一个Criteria对象，返回q__user表中所有记录。

（2）使用criteria.list（）方法，获得记录列表。

（3）打印出列表内容。

16.2　设定Criteria查询条件

前一节介绍了简单的Criteria使用，本节介绍如何使用Criteria对象，设定各种条件，获得相应的数据。本节的示例介绍如何设定简单的Criteria查询条件。

技术要点

前面小节介绍了如何使用Criteria来获得数据，但是返回的是全部的记录，而没有加入查询条件，本节在熟悉使用Criteria的条件下，添加相应的查询条件，获得用户期待的结果，使读者能够熟悉如何在Criteria对象中加入相应的条件。

实现代码

仍然使用上节中的数据库表、实体类和映射文件，增加一个测试类Test__Restrictions.java，代码如下：

```
    package criteria;
    import java.util.Iterator;
    import java.util.List;
    import org.hibernate.Criteria;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.cfg.Configuration;
    import org.hibernate.criterion.Restrictions;
    public class Test__Restrictions {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（） .configure（）；
    //根据config建立SessionFactory
    //SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）；
    //建立Session，相当于建立JDBC的Connection
    Session session=sessionFactory.openSession（）；
    Criteria
criteria=session.createCriteria（QUser.class）；//创建Criteria实例
    //1增加查询条件：20至40岁之间（不含）
    criteria.add（Restrictions.gt（"age", new
Integer（20）））；
    criteria.add（Restrictions.lt（"age", new
Integer（40）））；
    List users=criteria.list（）；//查询所有的q__user数据表记录
```

```
    System.out.println（"id\t name/age"）;
    Iterator iterator=users.iterator（）; //循环访问
List元素
    while（iterator.hasNext（）） {
    QUser user=（QUser）iterator.next（）;
    System.out.println（user.getId（）+"\
t"+user.getName（）+"/"
    +user.getAge（））;
    }
    //2增加查询条件：小于20或者是空值
    criteria=session.createCriteria（QUser.class）;
    criteria.add（Restrictions.or（Restrictions.lt（"
age", new Integer（20））,
    Restrictions.isNull（"age"）））;
    users=criteria.list（）; //查询所有的q__user数据表记
录
    System.out.println（"id\t name/age"）;
    iterator=users.iterator（）; //循环访问List元素
    while（iterator.hasNext（）） {
    QUser user=（QUser）iterator.next（）;
    System.out.println（user.getId（）+"\
t"+user.getName（）+"/"
    +user.getAge（））;
    }
    session.close（）;
    sessionFactory.close（）;
    }
    }
```

运行该测试类，结果如下：

```
    08: 46: 15，875 DEBUG SQL: 346select this__.id as
id0__0__, this__.name as name0__0__, this__.age as
age0__0__,
```

```
    this__.email as email0__0__from ssh.q__user this
__where this__.age>? and this__.age<?
    idname/age
    1张三/22
    2 pla/33
    4闫术卓/30
    08: 46: 16，031 DEBUG SQL: 346select this__.id as
id0__0__，this__.name as name0__0__，this__.age as
age0__0__，
    this__.email as email0__0__from ssh.q__user this
__where（this__.age<? or this__.age is null）
    3李四/18
    id name/age
    5杨强/18
```

源程序解读

（1）使用Criteria对象的add（）方法添加查询条件。

（2）Criteria对象的常用查询方法如表16.1所示。

表 16.1　Criteria 对象的常用查询方法

方　法	说　明
Restrictions.eq( )	equal，=
Restrictions.allEq( )	参数为 Map 对象，使用 key/value 进行多个等于的对比，相当于多个 Restrictions.eq( )的效果
Restrictions.gt( )	greater-than，>
Restrictions.lt( )	less-than，<
Restrictions.le( )	less-equal，<=
Restrictions.between( )	对应 SQL 的 between 子句
Restrictions.like( )	对应 SQL 的 like 子句
Restrictions.in( )	对应 SQL 的 in 子句
Restrictions.and( )	and 关系
Restrictions.or( )	or 关系
Restrictions.isNull( )	判断属性是否为空，为空返回 true，否则返回 false
Restrictions.isNotNull( )	与 Restrictions.isNull( )相反
Order.asc( )	根据传入的字段进行升序排序
Order.desc( )	根据传入的字段进行降序排序
MatchMode.EXACT	字符串精确匹配，相当于“like 'value'”
MatchMode.ANYWHERE	字符串在中间位置，相当于“like '%value%'”
MatchMode.START	字符串在最前面的位置，相当于“like 'value%'”
MatchMode.END	字符串在最后面的位置，相当于“like '%value'”

（3）示例中，第一个查询条件为“20至40岁之间（不含）”，可以使用gt（）和lt（）方法设定条件。

（4）第二个查询条件中的“或者”可以使用Restrictions.or（）方法设定。

（5）Restrictions类提供了查询限制机制。它提供了许多方法，以实现查询限制。

16.3 Criteria中使用SQL语句

除了使用Criteria对象的常用查询方法外，还可以使用SQL语句来设定查询条件，本节介绍如何在Criteria中使用SQL语句。

技术要点

Hibernate支持在Criteria中使用SQL语句，这样可以提高程序员在查询数据时的灵活性，对于数据库应用开发非常有用。本节介绍如何在Criteria中使用SQL语句。

实现代码

增加一个测试类Test__Sql.java，代码如下：

```
package criteria;
import java.util.Iterator;
import java.util.List;
```

```
    import org.hibernate.Criteria;
    import org.hibernate.Hibernate;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.cfg.Configuration;
    import org.hibernate.criterion.Restrictions;
    import org.hibernate.type.Type;
    public class Test__Sql {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（） .configure（）；
    //根据config建立SessionFactory
    //SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）；
    //建立Session，相当于建立JDBC的Connection
    Session session=sessionFactory.openSession（）；
    Criteria
criteria=session.createCriteria（QUser.class）；//创
建Criteria实例
    //1模糊查询
    criteria.add（Restrictions.sqlRestriction（"{al
ias} .name LIKE（？）", "闫%",
    Hibernate.STRING））；
    List users=criteria.list（）；//查询所有q__user数
据表记录
    System.out.println（"id\t name/age"）；
    Iterator iterator=users.iterator（）；
    while（iterator.hasNext（）） {//循环访问List元素
    QUser user=（QUser）iterator.next（）；
    System.out.println（user.getId（）+"\
t"+user.getName（）+"/"
    +user.getAge（））；
    }
    //2增加查询条件：小于20或者是空值
    criteria=session.createCriteria（QUser.class）；
```

```
    Integer[]ages= {new Integer（20）, new
Integer（40）}; //定义查询参数
    Type[]types= {Hibernate.INTEGER,
Hibernate.INTEGER}; //定义查询参数类型
    criteria.add（Restrictions.sqlRestriction（//设
定SQL查询参数
    " {alias} .age BETWEEN（？）AND（？）", ages,
types））;
    users=criteria.list（）;
    System.out.println（"id\t name/age"）;
    iterator=users.iterator（）; //循环访问List元素
    while（iterator.hasNext（））{
    QUser user=（QUser）iterator.next（）;
    System.out.println（user.getId（）+"\
t"+user.getName（）+"/"
    +user.getAge（））;
    }
    session.close（）;
    sessionFactory.close（）;
    }
    }
```

运行该测试类，结果如下：

```
    09: 03: 03, 781 DEBUG SQL: 346select this__.id as
id0__0__, this__.name as name0__0__, this__.age as
age0__0__,
    this__.email as email0__0__from ssh.q__user this
__where this__.name LIKE（？）
    idname/age
    4闫术卓/30
    09: 03: 03, 906 DEBUG SQL: 346select this__.id as
id0__0__, this__.name as name0__0__, this__.age as
age0__0__,
```

```
    this__.email as email0__0__from ssh.q__user this
__where this__.age BETWEEN（？）AND（？）
    1张三/22
    id name/age
    2 pla/33
    4闫术卓/30
```

源程序解读

（1）可以在Restrictions.sqlRestriction（）方法中设定SQL查询条件，并使用add（）方法加入到Criteria对象中。

（2）在SQL语句中，不用再写“where”。

（3）{alias}代表了对应实体类的名称，本示例中为QUser。

（4）语句中的“？”将会被实际参数代替，例如“{alias}.name LIKE（？）″,″闫%″,Hibernate.STRING”会被编译为“select*from q__user where name like″闫%″”。

16.4　复杂的Criteria查询

本节介绍如何使用Criteria对象进行统计、分组、计算和排序等查询。技术点并不是主要的，主要就是一种复杂查询理念。读者一定要仔细阅读后面的源程序解读。

技术要点

在数据库应用项目中，经常用到复杂的查询和计算，例如进行统计、分组、计算和排序等操作，Criteria对象支持复杂的查询和计算，读者通过本节可以熟悉如何在Criteria对象中实现统计、分组、计算和排序。本节示例展示如何在Criteria中加入复杂的查询条件。

实现代码

增加一个测试类Test__all.java，代码如下：

```
    package criteria;
    import java.util.Iterator;
    import java.util.List;
    import org.hibernate.Criteria;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.cfg.Configuration;
    import org.hibernate.criterion.Order;
    import org.hibernate.criterion.ProjectionList;
    import org.hibernate.criterion.Projections;
    public class Test__all {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（）.configure（）；
    //根据config建立SessionFactory
    //SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）；
    //建立Session，相当于建立JDBC的Connection
    Session session=sessionFactory.openSession（）；
    Criteria
criteria=session.createCriteria（QUser.class）；//创
建Criteria实例
    //1排序倒序使用desc（）
    criteria.addOrder（Order.asc（"age"））；
    List users=criteria.list（）；//查询所有的q__user
数据表记录
    System.out.println（"id\t name/age"）；
    Iterator iterator=users.iterator（）；
    while（iterator.hasNext（）） {//循环访问List元素
    QUser user=（QUser）iterator.next（）；
```

```
    System.out.println（user.getId（）+"\
t"+user.getName（）+"/"
    +user.getAge（））；
    }
    //2指定记录范围：起始记录和记录数，可以用来分页
    criteria=session.createCriteria（QUser.class）；
    criteria.setFirstResult（2）；
    criteria.setMaxResults（5）；
    users=criteria.list（）；
    System.out.println（"id\t name/age"）；
    iterator=users.iterator（）；
    while（iterator.hasNext（））｛//循环访问List元素
    QUser user=（QUser）iterator.next（）；
    System.out.println（user.getId（）+"\
t"+user.getName（）+"/"
    +user.getAge（））；
    }
    //3统计、分组
    ProjectionList
projectionList=Projections.projectionList（）；//设
定查询条件
    projectionList.add（Projections.groupProperty（"
age"））；
    projectionList.add（Projections.rowCount（））；
    criteria=session.createCriteria（QUser.class）；
    criteria.setProjection（projectionList）；
    criteria.addOrder（Order.desc（"age"））；//排序
    users=criteria.list（）；
    iterator=users.iterator（）；
    while（iterator.hasNext（））｛
    Object[]o=（Object[]）iterator.next（）；
    System.out.println（o[0]+"\t"+o[1]）；
    }
    session.close（）；
    sessionFactory.close（）；
    }
    }
```

运行该测试类，结果如下：

```
    09: 15: 55, 625 DEBUG SQL: 346select this__.id as
id0__0__, this__.name as name0__0__, this__.age as
age0__0__,
    this__.email as email0__0__from ssh.q__user this
__order by this__.age asc
    idname/age
    3李四/18
    5杨强/18
    1张三/22
    4闫术卓/30
    2 pla/33
    09: 15: 55, 828 DEBUG SQL: 346select this__.id as
id0__0__, this__.name as name0__0__, this__.age as
age0__0__,
    this__.email as email0__0__from ssh.q__user this
__limit? , ?
    3李四/18
    4闫术卓/30
    id name/age
    5杨强/18
    09: 15: 55, 843 DEBUG SQL: 346select this__.age
as y0__, count（）as y1__from ssh.q__user this__group
by
    this__.age order by this__.age desc
    33  1
    30  1
    22  1
    18  2
```

源程序解读

（1）asc（）方法是顺序排序，desc（）方法是倒序排序。

（2）可以使用setFirstResult（）方法和setMaxResults（）方法设定查询的起始记录和最大记录数量，提供了一个简单的分页处理功能。

（3）Projections类主要用于帮助Criteria接口完成数据的分组查询和统计功能，例如下面代码：

```
List cats=session.createCriteria（Cat.class）
.setProjection（Projections.projectionList（）
.add（Projections.rowCount（））
.add（Projections.avg（"weight"））
.add（Projections.max（"weight"））
.add（Projections.min（"weight"））
.add（Projections.groupProperty（"color"））
）.addOrder（Order.asc（"color"））.list（）;
```

相当于下面的SQL语句：

```
select color, count（）, avg（weight）,
max（weight）, min（weight）, min（weight）from cat
group by color order
by color asc;
```

16.5 使用DetchedCriteria查询

从前面的介绍中可以看到，Criteria对象与Session是绑定的，其生命周期跟随着Session结束而结束，使用Criteria时进行查询时，每次都要于执行时动态建立对象，并加入各种查询条件，随着Session的回收，Criteria也跟着回收。

为了能够重复使用Criteria对象，在Hibernate 3中新增了DetchedCriteria，可以先建立DetchedCriteria对象，并加入各种查询条件，并于需要查询时再与Session绑定，获得一个绑定Session的Criteria对象。

技术要点

Criteria对象的作用范围是Session，是同Session绑定在一起的，DetchedCriteria对象同Session对象是分开的，这就是两者之间的差异。本节将展示这种差异，并介绍如何使用DetchedCriteria对象。

实现代码

增加一个测试类Test__DetchedCriteria.java，代码如下：

```
    package criteria;
    import java.util.Iterator;
    import java.util.List;
    import org.hibernate.Criteria;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.cfg.Configuration;
    import
org.hibernate.criterion.DetachedCriteria;
    import org.hibernate.criterion.Restrictions;
    public class Test__DetchedCriteria {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（） .configure（）;
```

```
    //根据config建立SessionFactory
    //SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）；
    //建立Session，相当于建立JDBC的Connection
    DetachedCriteria
detchedCriteria=DetachedCriteria//建立
DetachedCriteria实例
    .forClass（QUser.class）；
    detchedCriteria.add（Restrictions.ge（"age"，new
Integer（25）））；//加入查询条件
    Session
session=sessionFactory.openSession（）；//第一个
session绑定
    //绑定Session，返回一个Criteria实例
    Criteria
criteria=detchedCriteria.getExecutableCriteria（ses
sion）；
    List users=criteria.list（）；
    Iterator iterator=users.iterator（）；
    System.out.println（"id\t name/age"）；
    while（iterator.hasNext（）） {
    QUser ur=（QUser）iterator.next（）；
    System.out.println（ur.getId（）+"\
t"+ur.getName（）+"/"
    +ur.getAge（））；
    }
    session.close（）；
    sessionFactory.close（）；
    //第二个session绑定
    detchedCriteria.add（Restrictions.le（"age"，new
Integer（32）））；
    session=sessionFactory.openSession（）；
    //绑定Session，返回一个Criteria实例
    criteria=detchedCriteria.getExecutableCriteria
（session）；
    users=criteria.list（）；
```

```
    iterator=users.iterator（）；
    System.out.println（"id\t name/age"）；
    while（iterator.hasNext（）） {
    QUser ur=（QUser）iterator.next（）；
    System.out.println（ur.getId（）+"\
t"+ur.getName（）+"/"
    +ur.getAge（））；
    }
    session.close（）；
    sessionFactory.close（）；
    }
    }
```

源程序解读

（1）可以在未绑定Session之前建立DetchedCriteria对象，并使用add（）方法加入查询条件。

（2）建立Session后，通过getExecutableCriteria（）方法获得Criteria实例。

（3）在Session关闭后，DetchedCriteria对象仍然存在，可以继续访问和操作，本示例中在第1个

Session关闭后，增加了一个查询条件，然后同第2个Session绑定，获得新的Criteria实例。

第17章　HQL查询

Hibernate配备了一种非常强大的查询语言HQL（Hibernate Query Language），这种语言看上去很像SQL。但是不要被语法结构上的相似所迷惑，HQL被设计为完全面向对象的查询，它可以理解如继承、多态和关联之类的概念。

17.1　简单的HQL查询

HQL是Hibernate所推荐使用的查询方式，HQL同标准的SQL非常相似，但是有很多区别，本节通过示例演示如何在Hibernate中使用HQL查询语句。首先建立两个示例数据库表student和ex__result，用来存放学生和考试成绩数据，其结构如图17.1所示。本例就介绍如何浏览学生表，并打印出学生的年龄。

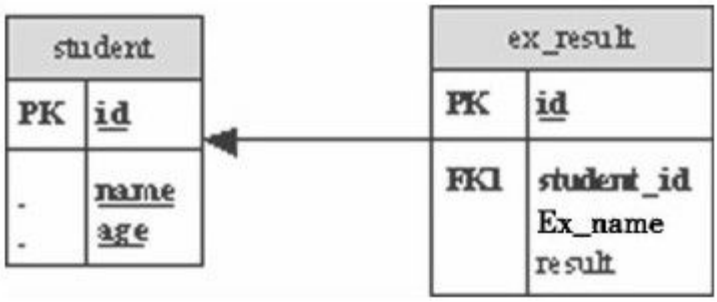

图　17.1　HQL示例表关系

注意：本章示例均不用配置2个表之间的关系，图17.1只是表示它们之间的逻辑关系，并不需要在映射文件中配置。

技术要点

HQL是Hibernate的常用查询方式，也是Hibernate的核心部分之一，读者必须熟练掌握HQL。本例帮助读者熟悉简单的HQL查询语句。

实现代码

建立学生的实体类Student.java，代码如下：

```
    ackage hql;
    public class Student implements
java.io.Serializable {
    private Integer id; //主键ID属性变量
    private String name; //姓名属性变量
    private Integer age; //年龄属性变量
    public Student（） {//构造器
    }
    public Integer getId（） {//id属性的Getter（）方法
和Setter（）方法
    return this.id;
    }
    public void setId（Integer id） {
    this.id=id;
    }
    public String getName（） {//name属性的Getter（）
方法和Setter（）方法
    return this.name;
    }
    public void setName（String name） {
    this.name=name;
    }
    public Integer getAge（） {//age属性的Getter（）方
法和Setter（）方法
    return this.age;
    }
    public void setAge（Integer age） {
    this.age=age;
    }
    }
```

建立学生成绩的实体类ExResult.java，代码如下：

```
    package hql;
    public class ExResult implements
java.io.Serializable {
    private Integer id; //主键属性变量
    private Integer studentId; //学生ID属性变量
    private String exName; //考试名称属性变量
    private Double result; //考试成绩属性变量
    public ExResult() {//构造器
    }
    public Integer getId() {//id属性的Getter()方法
和Setter()方法
    return this.id;
    }
    public void setId(Integer id) {
    this.id=id;
    }
    public Integer getStudentId() {//studentid属性
的Getter()方法和Setter()方法
    return this.studentId;
    }
    public void setStudentId(Integer studentId) {
    this.studentId=studentId;
    }
    public String getExName() {//exname属性的
Getter()方法和Setter()方法
    return this.exName;
    }
    public void setExName(String exName) {
    this.exName=exName;
    }
    public Double getResult() {//result属性的
Getter()方法和Setter()方法
    return this.result;
    }
    public void setResult(Double result) {
    this.result=result;
```

```
    }
    }
```

建立映射文件Student.hbm.xml，内容如下：

```
    <? xml version="1.0"encoding="utf8"? >
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
    <class
name="hql.Student"table="student"catalog="ssh">
    <id name="id"type="java.lang.Integer">
    <column name="id"/>
    <generator class="native"/>
    </id><! 配置主键映射>
    <property name="name"type="java.lang.String">
    <column name="name"length="20"notnull="true"/>
    </property><! 配置姓名映射>
    <property name="age"type="java.lang.Integer">
    <column name="age"notnull="true"/>
    </property><! 配置年龄映射>
    </class>
    </hibernatemapping>
```

建立映射文件ExResult.hbm.xml，内容如下：

```
    <? xml version="1.0"encoding="utf8"? >
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
```

```
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
    <class name="hql.ExResult"table="ex__
result"catalog="ssh">
    <id name="id"type="java.lang.Integer">
    <column name="id"/>
    <generator class="native"/>
    </id><! 配置主键映射>
    <property
name="studentId"type="java.lang.Integer">
    <column name="student__id"notnull="true"/>
    </property><! 配置学生id映射>
    <property
name="exName"type="java.lang.String">
    <column name="Ex__
name"length="20"notnull="true"/>
    </property><! 配置考试名称映射>
    <property
name="result"type="java.lang.Double">
    <column
name="result"precision="22"scale="0"notnull="true"
/>
    </property><! 配置考试成绩映射>
    </class>
    </hibernatemapping>
```

将该映射文件加入到Hibernate配置文件中，建立测试类Test__HQL__Basic.java，代码如下：

```
    package hql;
    import java.util.Iterator;
    import java.util.List;
```

```
    import org.hibernate.Query;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.cfg.Configuration;
    public class Test__HQL__Basic {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（） .configure（）；
    //根据config建立SessionFactory
    //SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）；
    //建立Session，相当于建立JDBC的Connection
    Session session=sessionFactory.openSession（）；
    Query query=session//1 HQL基本查询
    .createQuery（"select user.name, user.age from
Student as user"）；
    List names=query.list（）；
    Iterator iterator=names.iterator（）；
    while（iterator.hasNext（）） {
    Object[]obj=（Object[]）iterator.next（）；
    System.out.println（obj[0]+"\t"+obj[1]）；
    }
    //2查询记录总数
    query=session.createQuery（"select count（） from
Student as user"）；
    names=query.list（）；
    iterator=names.iterator（）；
    while（iterator.hasNext（）） {
    System.out.println（iterator.next（））；
    }
    query=session//3计算最大年龄
    .createQuery（"select max（user.age）from
Student as user"）；
    names=query.list（）；
    iterator=names.iterator（）；
```

```
while（iterator.hasNext（））{
System.out.println（iterator.next（））;
}
session.close（）;
sessionFactory.close（）;
}
}
```

运行该测试类，结果如下：

```
09: 57: 00, 421 DEBUG QueryTranslatorImpl:
206HQL: select user.name, user.age from hql.Student
as user
09: 57: 00, 421 DEBUG QueryTranslatorImpl:
207SQL: select student0__.name as col__0__0__,
09: 57: 00, 437 DEBUG SQL: 346select student0
__.name as col__0__0__, student0__.age as col__1__0__
from
ssh.student student0__
张三  14
李四  16
王小慧  17
杨强  12
……
09: 57: 00, 500 DEBUG QueryTranslatorImpl:
206HQL: select count（）from hql.Student as user
09: 57: 00, 500 DEBUG QueryTranslatorImpl:
207SQL: select count（）as col__0__0__from
ssh.student
student0__
09: 57: 00, 500 DEBUG SQL: 346select count（）as
col__0__0__from ssh.student student0__
4
……
```

```
    09: 57: 00，515 DEBUG QueryTranslatorImpl:
206HQL: select max（user.age）from hql.Student as
user
    09: 57: 00，515 DEBUG QueryTranslatorImpl:
207SQL: select max（student0__.age）as col__0__0__
from
    ssh.student student0__
    09: 57: 00，515 DEBUG SQL: 346select
max（student0__.age）as col__0__0__from ssh.student
student0__
    17
    ……
```

源程序解读

（1）HQL查询语句中，大小写是不敏感的，但是其中的对象类的名称和属性是大小写敏感的。例如：

“sElect cat. name from Cat as cat”和“select cat.name from Cat as cat”是一样的；“sElect cat.name from CAT as cat”和“select cat.name from Cat as cat”是不一样的。

（2）from后面的是对象的类名称，而不是对应的数据库表名称，初学者一定要注意。

（3）条件语句同SQL类似。

（4）使用session.createQuery（）方法进行查询。

17.2　复杂的HQL查询

本节介绍如何在HQL中进行复杂条件和分组查询。前面虽然创建了一个学生表和成绩表，但只通过HQL查询了学生表的内容，那如何通过两个表关联，来找到学生的成绩呢？

技术要点

前面小节了解了简单的HQL查询，本节介绍复杂的HQL查询，例如复杂条件或者是分组统计（通过两个表关联来实现）。对HQL查询语句的掌握，直接影响程序员使用Hibernate的能力。本例首先设计了多个条件，然后设计两个数据库的关联，如满足年龄大于15的学生的某门课的成绩，条件描述有点复杂。

实现代码

仍然使用前面的数据库表和配置文件，增加一个测试类Test__HQL.java，代码如下：

```
    p a c k a g e h q l;
    import java.util.Iterator;
    import java.util.List;
    import org.hibernate.Query;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.cfg.Configuration;
    public class Test__HQL {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（） .configure（）；
    //根据config建立SessionFactory
    //SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）；
    //建立Session，相当于建立JDBC的Connection
    Session session=sessionFactory.openSession（）；
    //1查询年龄大于15的学生的数学成绩
    Query query=session
    .createQuery（"select student.name,
student.age, res.result from Student as student,
ExRe
    sult as res where student.id=res.studentId and
student.age＞15 and res.exName=′数学′ "）；
    List res=query.list（）；
    Iterator iterator=res.iterator（）；
    while（iterator.hasNext（）） {
    Object[]obj=（Object[]）iterator.next（）；
    System.out.println（obj[0]+"\t"+obj[1]+"\
t"+obj[2]）；
```

```
    }
    //2按照数学、物理成绩合计进行排序
    String hql="select stu.name, stu.age,
sum（res.result）from Student as stu, ExResult as
res where
    stu.id=res.studentIdgroup by stu.name having
sum（res.result）＞180 order by sum（res.result）
desc";
    query=session.createQuery（hql）;
    res=query.list（）;
    iterator=res.iterator（）;
    while（iterator.hasNext（）） {
    Object[]obj=（Object[]）iterator.next（）;
    System.out.println（obj[0]+"＼t"+obj[1]+"＼
t"+obj[2]）;
    }
    session.close（）;
    sessionFactory.close（）;
    }
    }
```

运行该测试类，结果如下：

```
    10: 09: 13, 421 DEBUG QueryTranslatorImpl:
206HQL: select student.name, student.age,
res.result from
    hql.Student as student, hql.ExResult as res
where student.id=res.studentId and student.age＞15
and
    res.exName=′数学′
    10: 09: 13, 421 DEBUG QueryTranslatorImpl:
207SQL: select student0__.name as col__0__0__,
student0__.age as
```

```
    col__1__0__, exresult1__.result as col__2__0__from
ssh.student student0__, ssh.ex__result exresult1__
where student0__
    .id=exresult1__.student__id and student0__.age>
15 and exresult1__.Ex__name=′数学′
    10: 09: 13, 421 DEBUG ErrorCounter:
68throwQueryException（）: no errors
    10: 09: 13, 437 DEBUG SQL: 346select student0
__.name as col__0__0__, student0__.age as col__1__0
__, exresult1__
    .result as col__2__0__from ssh.student student0
__, ssh.ex__result exresult1__where student0
__.id=exresult1__
    .student__id and student0__.age>15 and
exresult1__.Ex__name=′数学′
    李四  16  67.5
    王小慧17  96.0
    ……
    10: 09: 13, 562 DEBUG QueryTranslatorImpl:
206HQL: select stu.name, stu.age, sum（res.result）
from
    hql.Student as stu, hql.ExResult as res where
stu.id=res.studentIdgroup by stu.name having sum
    （res.result）>180 order by sum（res.result）
desc
    10: 09: 13, 562 DEBUG QueryTranslatorImpl:
207SQL: select student0__.name as col__0__0__,
student0__.age as
    col__1__0__, sum（exresult1__.result）as col__2__0
__from ssh.student student0__, ssh.ex__result
exresult1__where
    student0__.id=exresult1__.student__id group by
student0__.name having sum（exresult1__.result）>
180 order by
    sum（exresult1__.result）desc
    10: 09: 13, 562 DEBUG ErrorCounter:
68throwQueryException（）: no errors
```

```
    10: 09: 13，562 DEBUG SQL: 346select student0
__.name as col__0__0__，student0__.age as col__1__0
__，sum（exre
    sult1__.result）as col__2__0__from ssh.student
student0__，ssh.ex__result exresult1__where student0
__.id=exresult1
    __.student__id group by student0__.name having
sum（exresult1__.result）>180 order by
sum（exresult1__
    .result）desc
    杨强12  195.2
    王小慧17  194.0
    ……
```

源程序解读

（1）使用Log4j日志，可以清楚地看到HQL查询语句如何转换为标准的SQL语句，这对于查找错误至关重要。

（2）下面给出几个常见的HQL查询示例：

在where子句上进行表达式计算作为条件，例如：

```
    Query query=session.createQuery（"from User
user where（user.age/10=3）"）；
```

可以在where子句上使用and、or等逻辑符号，例如：

```
Query query=session.createQuery（"from User user where（user.age＞20）and（user.name='caterpillar'）"）；
```

可以使用“is not null”与“is null”测试字段值是否为空值，例如：

```
Query query=session.createQuery（"from User user where user.name is not null"）；
```

使用“between”可以测试字段值是否在指定的范围之内，例如：

```
Query query=session.createQuery（"from User user where user.age between 20 and 30"）；
```

可以使用“in”或“not in”来测试字段值是否在指定的集合中，例如：

```
```

```
Query query=session.createQuery（"from User user where user.name in（′ caterpillar', ′ momor'）"）；
```

使用“like”或“not like”可以进行模糊条件查询，例如：

```
Query query=session.createQuery（"from User user where user.name like'cater%′ "）；
```

可以使用“order by”对查询结果进行排序：

```
Query query=session.createQuery（"from User user order by user.age"）；
```

可使用“desc”反排序：

```
Query query=session.createQuery（"from User user order by user.age desc"）；
```

可同时指定两个以上的排序方式，例如先按照“age”反序排列，如果“age”相同，则按照“name”顺序排列：

```
Query query=session.createQuery（"from User user order by user.age desc, user.name"）;
```

可以配合GROUP BY子句，自动将指定的字段依相同的内容群组，例如依字段“sex”分组并作平均：

```
Query query=session.createQuery（"select user.sex, avg（user.age）from User user group by user.sex"）;
```

17.3 HQL更新、删除操作

在Hibernate 2中，HQL只能用来查询，不能用于更新和产出数据，更新和删除需要使用Session的update（）、saveOrUpdate（）和delete（）等方法，在Hibernate 3中，HQL新增了update和delete语句，可以直接使用HQL进行更新或删除操作。

技术要点

Hibernate 3版本中，除了使用HQL进行数据库查询，还提供了HQL对数据库进行更新、删除操作的支持。这是Hibernate 3及其以后版本的新特性，读者有必要熟练掌握。本节就通过实际代码演示更新和删除操作。

实现代码

仍然使用前面的数据库表和配置文件，增加1个测试类Test__Update.java，代码如下：

```
    package hql;
    import java.util.Iterator;
    import java.util.List;
    import org.hibernate.Query;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.Transaction;
    import org.hibernate.cfg.Configuration;
    public class Test__Update {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（） .configure（）；
    //根据config建立SessionFactory
    //SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）；
    //建立Session，相当于建立JDBC的Connection
    Session session=sessionFactory.openSession（）；
    Transaction tx=session.beginTransaction（）；
    Query query=session.createQuery（"update
Student set name=′张三update'where name=′张三
    ′ "）；//修改"张三"的名字
    query.executeUpdate（）；
    tx.commit（）；
    //显示修改结果
    query=session.createQuery（"select user.name,
user.age from Student as user"）；
    List names=query.list（）；
    Iterator iterator=names.iterator（）；
    while（iterator.hasNext（） ） {
```

```
Object[]obj=（Object[]）iterator.next（）；
System.out.println（obj[0]+"\t"+obj[1]）；
}
session.close（）；
sessionFactory.close（）；
}
}
```

运行该测试类，结果如下：

```
10：26：05，703 DEBUG SQL：346update ssh.student
set name=′张三update'where name=′张三′
……
10：26：05，828 DEBUG QueryTranslatorImpl：
206HQL：select user.name，user.age from hql.Student
as user
10：26：05，828 DEBUG QueryTranslatorImpl：
207SQL：select student0__.name as col__0__0__，
student0__.age as
col__1__0__from ssh.student student0__
10：26：05，843 DEBUG ErrorCounter：
68throwQueryException（）：no errors
10：26：05，843 DEBUG SQL：346select student0
__.name as col__0__0__，student0__.age as col__1__0__
from
ssh.student student0__
张三update14
李四  16
王小慧17
杨强  12
```

源程序解读

（1）本节示例只给出了update操作，与delete操作基本相同。

（2）使用HQL进行数据库更新和删除操作时，建议打开事务管理Transaction，使用executeUpdate（）方法执行SQL语句后，提交事务。

（3）注意Hibernate 2不支持更新、删除的HQL操作。

17.4　在XML中定义HQL

有数据库项目开发经验的程序员都会知道，由于业务逻辑的改变，SQL查询语句会频繁发生变化，前面介绍的HQL查询，都是将查询语句采用“硬编码”的方式编写在Java程序文件中，降低了可维护性。Hibernate支持在XML文件中定义HQL查询语句，这样极大地提高了重用性和可维护性。

技术要点

许多有经验的程序员都会发现，传统的将查询语句书写在程序代码中，也就是所谓的“硬编码”，会给项目系统以后的运行维护带来很多困难，因为频繁的查询条件变化将导致频繁的重新编译和部署。

Hibernate提供了在XML文件中定义HQL查询语句的功能，这样修改查询语句的条件无需修改程序代码，可以直接修改映射文件中的查询条件，不用重新编译和部署，本节介绍相关知识。

实现代码

仍然使用前面的数据库表，在Student.hbm.xml映射文件中增加query配置元素，指定查询语句。修改后的Student.hbm.xml映射文件内容如下：

```
    <? xml version="1.0"encoding="utf8"? >
    <! DOCTYPE hibernatemapping
PUBLIC"//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http: //hibernate.sourceforge.net/hibernatemap
ping3.0.dtd">
    <hibernatemapping>
    <class
name="hql.Student"table="student"catalog="ssh">
    <id name="id"type="java.lang.Integer">
    <column name="id"/>
    <generator class="native"/>
    </id><! 配置主键映射>
    <property name="name"type="java.lang.String">
    <column name="name"length="20"notnull="true"/>
    </property><! 配置姓名映射>
```

```
    <property name="age"type="java.lang.Integer">
    <column name="age"notnull="true"/>
    </property><! 配置年龄映射>
    </class>
    <! 定义查询语句>
    <query name="student__query">
    <! [CDATA[select user.name, user.age from
Student as user where user.age>: age]]></
    query>
    </hibernatemapping>
```

增加一个测试类Test__xml.java，代码如下：

```
    package hql;
    import java.util.Iterator;
    import java.util.List;
    import org.hibernate.Query;
    import org.hibernate.Session;
    import org.hibernate.SessionFactory;
    import org.hibernate.cfg.Configuration;
    public class Test__xml {
    public static void main（String[]args） {
    //Configuration管理Hibernate配置
    Configuration config=new
Configuration（）.configure（）;
    //根据config建立SessionFactory
    //SessionFactory用来建立Session
    SessionFactory
sessionFactory=config.buildSessionFactory（）;
    //建立Session，相当于建立JDBC的Connection
    Session session=sessionFactory.openSession（）;
    //调用映射文件中的查询语句
    Query query=session.getNamedQuery（"student__
query"）;
```

```
query.setInteger（"age"，14）；//定义参数值
List names=query.list（）；
Iterator iterator=names.iterator（）；
while（iterator.hasNext（）） {
Object[]obj=（Object[]）iterator.next（）；
System.out.println（obj[0]+"\t"+obj[1]）；
}
session.close（）；
sessionFactory.close（）；
}
}
```

运行该测试类，结果如下：

```
10: 38: 43，578 DEBUG QueryTranslatorImpl:
206HQL:
select user.name, user.age from hql.Student as
user where user.age>: age
10: 38: 43，578 DEBUG QueryTranslatorImpl:
207SQL: select student0__.name as col__0__0__,
student0__.age as
col__1__0__from ssh.student student0__where
student0__.age>?
10: 38: 43，578 DEBUG ErrorCounter:
68throwQueryException（）: no errors
10: 38: 43，687 DEBUG SQL: 346select student0
__.name as col__0__0__, student0__.age as col__1__0__
from
ssh.student student0__where student0__.age>?
李四 16
王小慧17
```

源程序解读

（1）在配置文件中，使用query定义查询语句，其中name属性是该查询语句的标示，Java程序中使用“session.getNamedQuery（″student__query″）”方法获得该查询语句。

（2）HQL查询语句中的变量使用“：”表示，这样可以在Java程序中进行变量赋值。

（3）建议在XML文件中设定查询语句以方便维护。

第18章　Spring入门

本章介绍时下最流行的J2EE轻量级开发框架Spring。本章针对Spring的整体技术架构做一简单扼要的介绍。希望没有学习过Spring的读者，能对Spring有一个清晰明了的整体概念。本章最后介绍了Spring框架的下载，并通过一个案例介绍了Srping框架的基本应用。

18.1　Spring历史发展过程

Spring最早的形成其实是源于一个人和他写的一本书，此人的名字叫Rod Johnson。在2000年作为一个金融技术顾问，他受邀为伦敦的金融界提供独立咨询业务。当时的他为了让为金融界使用的应用程序更好地和J2EE平台上各种不同的组件进行无缝结合，自己

亲自编写了一套技术代码框架。以当今的眼光来看这件事情其实也是很普通的一件事情。毕竟就在我们中国也有很多杰出的J2EE技术开发者，写出了很多可以独立使用在各种行业（不仅仅局限在金融行业）的技术代码框架。

18.1.1 Spring为什么越来越流行

不过Rod Johnson写的技术代码框架，不仅是可以让用户使用那么简单。在实现卓越的易用性和性能稳定性方面，Rod Johnson花了相当大的心血在他自己编写的这套技术代码框架中，结果就铸就了这套技术代码框架。更何况在当时，Sun公司的J2EE组件EJB是金融行业使用最多、最流行的一个技术组件。而Rod Johnson的这套框架却可以作为EJB的替代品，能更好地开发出为行业用户使用的应用程序。这样的举动在

当时Java业界中被视为里程碑，是富有历史意义的举动。

在做完这份工作后，Rod Johnson趁热打铁写了一本名为《Java企业应用设计与开发的专家一对一》的书。在书中他进一步拓展和修改了他的代码，用以更好地向大众声明，如何使用他的框架与其他所有J2EE的技术组件互相配合，从而产生更便捷、性能更可靠的、能更好地为行业用户服务的应用程序。当时他把自己这套技术代码框架命名为Spring，于是Spring诞生了。

在当时，Sun公司开发的Java Servlet API和EJB是最流行的技术组件。两者已经在Java业界获得了广泛的共识，Spring的出现被Sun公司和其支持者视为一种技术上的严峻挑战。但Spring框架的优势是基于最优方法，并适用于各种应用类型的开发框架。Rod

Johnson在他写的书中所提到的，种种对J2EE应用程序开发的改良和优化理念，也引起了很多Java技术开发者的兴趣。书发表后，基于读者的要求，源代码在开源使用协议下得以提供。

一群Java技术爱好者或开发者自发组成了团队，2003年2月在Sourceforge上构建了一个项目。在Spring框架上工作了一年之后，该团队在2004年3月发布了第一个版本（1.0）。这个版本之后，Spring框架在Java业界开始变得非常流行，而且由于它完整和详细的技术文档和参考文献更是吸引了新的一群开发者为此而疯狂，所以可以说2004年是Spring元年，之前的几年只是Spring的十月怀胎期。

18.1.2 Spring框架的核心

Spring在2004年也受到了很多责备和怀疑，不过这反而让它成为热烈争论的主题，同时更进一步扩大了它的知名度。Spring初现时，许多开发人员把它看作是远离传统编程模式的一步，特别是对远离像EJB这样的技术而言尤其如此。它的一个重要设计目标就是更好地与已存在的J2EE标准和商用工具整合。

Spring框架核心是由两大部分组成：

第一部分是反向控制（IOC），不过在如今的Spring官方网站上普遍使用的叫法是依赖注入（DI）。

第二部分AOP（面向方面的编程）则是Spring一种新的技术开发理念。也正是因为Spring框架非常高的

采用率，让AOP在Java业界也广受众多开发者的欢迎。

2005年，Spring因具有里程碑意义的新的版本的推出，更多功能的添加让Spring进一步在Java业界受到推广。并且Spring官方论坛对广大用户而言也已经成为最重要的信息和帮助的源泉。18.2节将详细说明Spring的特点和组成Spring的各个技术部分细节。

18.2 Spring的技术知识介绍

Spring其实也是一个很独立的技术开发框架。原则上在开发一个Web项目时，可以使用Spring框架来完成，无需使用其他J2EE技术组件或者是其他轻量级的J2EE技术框架。但是Spring也并不是这么的“排外”，它在和其他技术的整合应用上有着很大的优势。几乎可以说在现有J2EE技术中，没有任何技术能像它那样将其他技术与其整合的如此天衣无缝，因此它在一部分开发者中也得到了一个名为“粘合剂”的绰号。

本节将对Spring的所有组成部分的技术知识，做一个简单介绍。首先希望读者能对Spring有一个全面的了解，如图18.1所示列出了Spring组成部分的整体架构。

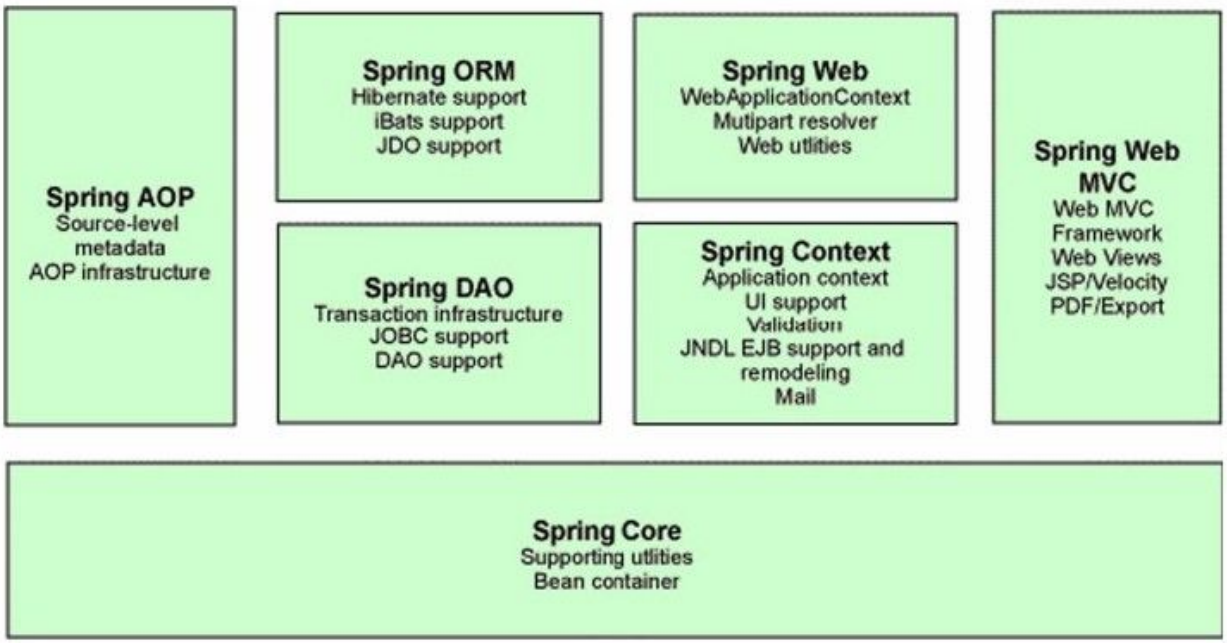

图　18.1　Spring整体结构图

组成Spring框架的这些部分（或模块）都可以单独存在，也可以与其他一个或多个模块组合起来共同实现。每个模块的功能如下。

核心容器：它提供Spring框架的基本功能。主要组件是BeanFactory，它是工厂模式的实现。BeanFactory使用控制反转（IOC）模式，将应用程序的配置和依赖性规范与实际的应用程序代码分开。

Spring上下文：Spring上下文（Context）其实是一个配置文件，用于向Spring框架提供上下文信息。Spring上下文包括企业服务，如JNDI、EJB、电子邮件、国际化、校验和调度功能。

Spring AOP：通过配置管理特性，Spring AOP模块直接将面向方面的编程功能，集成到了Spring框架中。所以，可以很方便地使Spring框架管理的任何对象支持AOP。Spring AOP模块为基于Spring的应用程序中的对象提供了事务管理服务。通过使用Spring AOP，不用依赖EJB组件，就可以将声明性事务管理集成到应用程序中。它也是Spring所具有的独特组件，同时也是最核心的组成部分之一。

Spring DAO：JDBC DAO抽象层提供了有意义的异常层次结构，可用该结构来管理异常处理和不同数据库供应商抛出的错误消息。异常层次结构简化了错误处理，并且很大程度地降低了需要编写的大量异常代

码（如打开和关闭数据库连接的代码）。Spring DAO的面向JDBC的异常也遵从通用的DAO异常层次结构。

Spring ORM：Spring框架插入了若干个ORM框架（如Hibernate、iBats）。所有这些ORM也都遵从Spring的通用事务和DAO异常层次结构。

Spring Web模块：Web上下文模块建立在应用程序上下文模块之上，为基于Web的应用程序提供了上下文。所以，Spring框架支持与以Struts为首的各种MVC框架的集成。Web模块还简化了处理多部分请求，以及将请求参数绑定到域对象的工作。

Spring MVC框架：MVC框架是一个功能齐全的构建Web应用程序的MVC实现。通过策略接口，MVC框架变成为高度可配置的，容纳了大量视图技术，其中包括JSP、Velocity等。

Spring框架的功能可以用在任何J2EE服务器中，大多数功能也适用于不受管理的环境。它的核心理念是：支持不绑定到特定J2EE服务的可重用业务和数据访问对象。因此这样的对象就可以在不同J2EE环境（Web或EJB）、独立应用程序、测试环境之间进行重用。

下面针对这些具体技术细节，做一下初步概念的介绍，希望初学者或入门者能对Spring有一个完整的认识。

18.2.1 Spring核心容器

介绍Spring核心容器，一定要先向读者介绍依赖注入（或称之为控制反转）这个概念。

依赖注入（控制反转）概念的主要内容是指：只描述程序中对象的被创建方式但不显式的创建对象。

在以XML语言描述的配置文件中，声明Web系统中哪个组件需要哪一种服务时，不是在程序中让对象和服务直接连接。具体负责连接工作的是Web容器（Spring中就是指IOC容器）。

在开发中具体实现的方式有三种：

每个服务实现专门的接口，对象通过接口来提供这些服务。由对象可知道具体的服务和对象之间的依赖关系。

通过JavaBean的属性来实现依赖关系（如每一个JavaBean对象的setter方法）。

利用对象中的构造方法来实现依赖关系，不显式地公开具体依赖关系。

在Spring中主要是采用后两种方式来实现依赖注入。具体的三种方式实现代码也可参考

http：//martinfowler.com/articles/injection.html，在该文中详细介绍了这三种方式的实现原理。

Spring核心部分通过使用实现依赖注入的org.springframework.beans这个包，和JavaBean一起使用来构成Spring框架的基础。之前所介绍的其他模块，都是由该模块作为实现其功能的最底层基础技术。

18.2.2 Spring上下文

Spring上下文在Spring中对应辅助程序包是org.springframework.context包，它提供了管理操作JavaBean的功能，其实也可以将它理解为Web系统中一个对象注册表的角色。

使用它在Spring中就不需要额外使用其他接口和对象，而且它也提供了国际化、事件处理和查询相关JavaBean的依赖关系等功能。如果开发者要进行Web系统开发，它是最好的轻量级解决方案。因为开发者只需要在配置文件中，用声明式方式配置管理对象和对象之间的依赖关系就可，减少了使用Spring所需的代码编写量。org.springframework.context包中也有已经开发好的API，能自动把它加载到开发者的Web应用程序中去。

18.2.3 Spring AOP解疑

AOP（面向方面编程）已经成为Java业界一个很热门的话题。如果读者具有一定的开发经验，会常常在开发中使用日志记录功能。假如在一个方法中记录该方法何时开始、何时结束，通常都是把这样的日志记录功能划分为一个独立的对象。在方法开始和结束时，调用该对象中记录开始和结束这两个方法。如果使用AOP，就只需在声明日志上记录对象的这两个方法应该在需要调用它们的方法开始、结束时候调用即可，这样就无需在方法的开始和结束时，用代码显式调用日志记录对象的这两个方法了，也进一步缩减了代码编写量。

注意：AOP和OOP（面向对象编程）不是矛盾的双方。反而AOP是OOP的一个有效的补充。只用AOP不用

OOP编程绝对是天方夜谭。因为AOP本身就是建立在OOP的基础上。但是在使用OOP编程时对于上述这种日志记录问题，使用AOP会使开发者的编程工作更有效率，也减少了代码编写的工作量。

AOP有两种不同类型：静态和动态AOP。Spring AOP是属于动态AOP的范畴。因此也可以理解成Spring AOP是AOP的子集，并且是动态AOP的一个很好的例子。

Spring AOP有自己独特的概念和术语。因此一般初学者学习起来会很费劲。现将这些概念和术语罗列在下，并逐个解释。

连接点（Jointpoint）：如果开发者调用一个方法，该方法执行，则类和对象初始化都可以称之为连接点。说的直白些，连接点就是程序执行过程中一个特定点。它是Spring AOP核心概念之一。使用它的主

要目的是告诉开发者在程序的哪个地方，可以通过AOP加入注入日志记录这样新的代码逻辑。

通知（Advice）：在指定的连接点执行的代码就是Advice。它有很多类型，最基本的有在连接点之前执行的前置通知，在连接点之后执行的后置通知。

切入点（Pointcut）：某一个通知可以在多个连接点执行，那么这些执行相同通知的连接点的集合就是切入点。这样开发者就可以自己控制程序中什么部分的代码需要调用执行什么通知。

方面（Aspect）：通知加切入点就是方面。所以方面不但定义了代码中包含的各种各样的逻辑（切入点），也定义了在何时执行这些逻辑（通知）。

编织（Weaving）：就笔者的理解这应该是一个动词，表明是执行一个动作。而这个动作在Spring AOP

中就是将方面加入代码的过程。对于Spring AOP这样的动态AOP而言，该动作是在程序运行时执行的。

目标（Target）：目标另外有一个名字为被通知对象。这个名字比目标更好理解。望文生义就是一个执行过程中有AOP作用的对象。

引入（Introduction）：此术语也可称为介入。它通过增加新的方法和属性，来改变一个对象的代码结构。甚至可以让被改变的对象从没有实现任何接口到变成实现开发者想让它实现的接口。

在我们编写的Struts 2部分也说到了代理模式（并有代码示例，具体翻阅Struts 2部分），而代理模式也是AOP的一个基础组成部分。在Spring中AOP的代理有两种不同情况：一是JDK动态代理，还有一个是CGLIB代理。其中前者是使用Spring开发Web系统中常

用的代理模式。但就执行性能来说CGLIB代理更佳，而且它还可以代理类。JDK动态代理则只能代理接口。

18.2.4 Spring DAO说明

该组成部分主要将J2EE中的JDBC重新做了封装和处理，将原有DAO设计中发生变化的部分分离或者封装起来。让开发者使用Spring DAO时，无需关心使用特定数据库的细节。

做过开发的读者都知道，进行数据库操作的开发流程无非是先取得数据源，取得数据库连接，进行数据库的增删改查操作，进行数据库事务管理，如果有发生数据库操作异常则处理这些异常。对于不同的业务逻辑和数据库来说，整体的数据库操作开发流程都是如此，只是少部分不同。因此Spring DAO运用模板模式，将不同的细节部分委托DAO支持对象来处理。

对于异常也是都使用Spring定义的异常结构，而不是处理与JDBC相关的异常处理体系，这样就让系统

运行时处理异常情况，不是耦合于JDBC技术。

18.2.5 Spring ORM介绍

该部分其实和Spring DAO相类似，只不过此部分通过支持大多数ORM技术来进行数据库访问等操作。比如本书介绍的Hibernate就是一种ORM技术。Spring ORM在开发中最常用的应用就是和Hibernate一起整合使用。

18.2.6 Spring Web模块

Spring Web模块是建立在Spring上下文的基础上，主要应用在Web系统的开发中。本书中也详细讲述了很多MVC结构的框架。该部分就是让Spring和众多MVC框架相结合来完成开发工作。最常用的MVC框架是Struts，如果读者此时还对Struts不熟悉，请参考前面的案例和介绍。

18.2.7 Spring MVC框架

Spring MVC框架是Spring框架中自己独特的MVC框架。它和Struts 2等MVC框架一样也是使用Model 2模式建立的，并且它也提供了控制器实现，HTTP请求参数转发等功能。不过在实际开发工作中它使用的并不是很多，因此本书从实用角度出发也就不做过多的介绍了。

18.3　使用Spring的基础示例

Spring最新版本是2.5.5，可以在http：//www.springframework.org/download下载到。下载有两个压缩包：springframework2.5.5.zip和springframework2.5.5dependencies.zip（其包含了众多支持spring的J2EE组件包和其他开源技术包）。

（1）下载完成后解压springframework2.5.5.zip压缩包，在dist文件夹下可看到Spring.jar包，显示界面如图18.2所示。

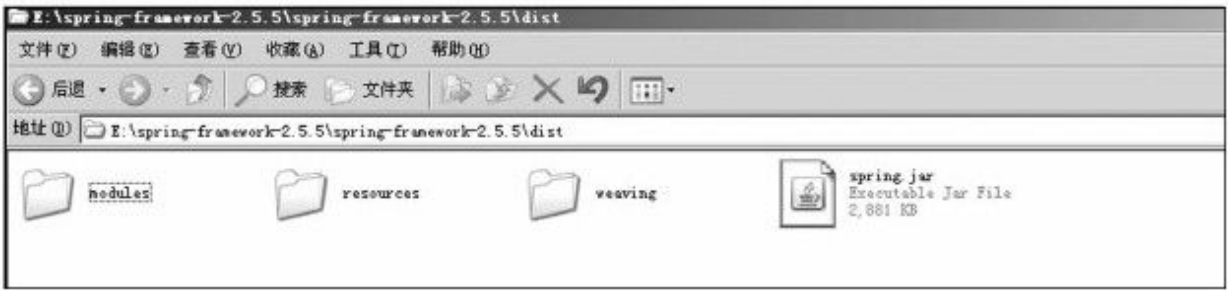

图　18.2　Spring文档结构图

（2）在modules文件夹中，包括了Spring各个组成部分的jar包。这样开发者就可以选择自己Web系统开发中需要的jar包，文档结构图如图18.3所示。

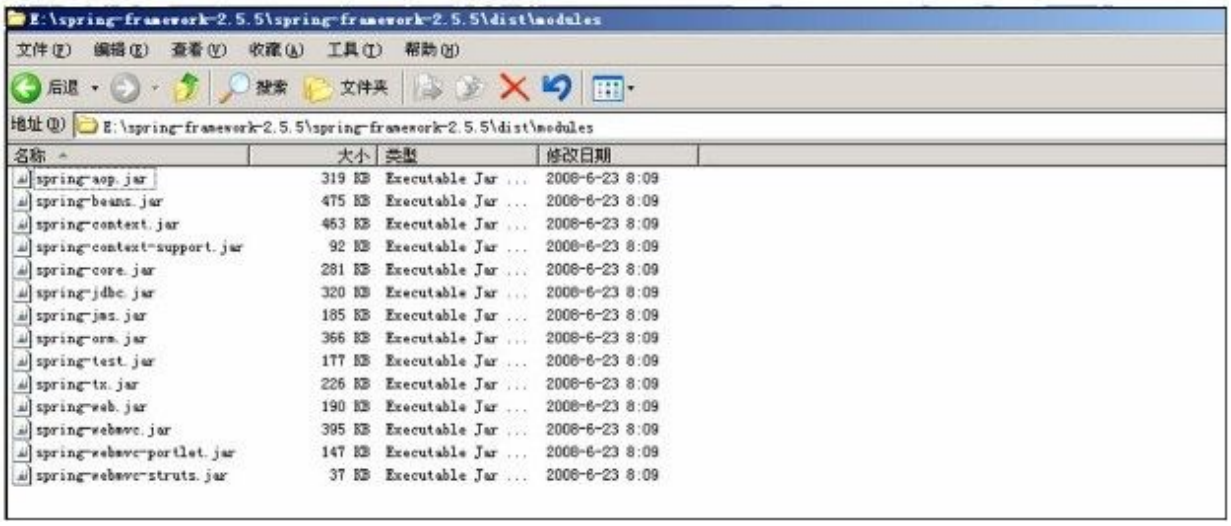

图　18.3　Spring各部分jar包图

熟悉完Spring的文件架构后，下面给出使用Spring示例的详细步骤。

（1）打开MyEclipse，在其中新建一个webproject。然后将使用Spring开发的几个jar包导入，如图18.4所示。

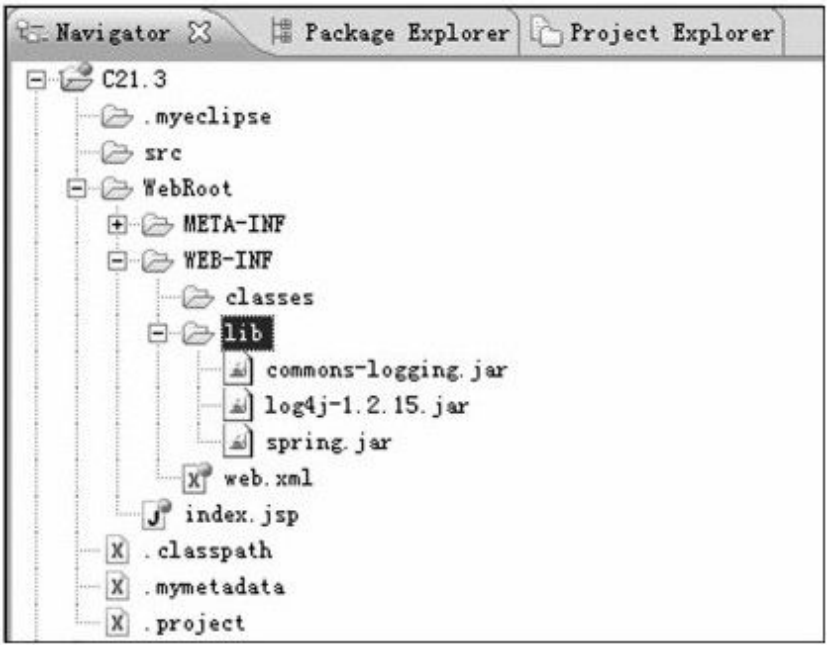

图　18.4　lib文件夹下使用Spring的jar包

其中spring.jar可在图18.2中的文件夹中找到。另外commonslogging.jar和log4j1.2.15.jar包，可在之前所说要下载的springframework2.5.5dependencies.zip压缩包中找到（两个jar包具体的文件路径，读者可用搜索文件功能找到）。

（2）为了在Web项目中使用Spring，在web.xml文件中加入以下代码：

```
    <!……文件名：
web.xml……>
    <contextparam>
    <paramname>contextConfigLocation</paramname>
    <paramvalue
>/WEBINF/classes/applicationContext.xml
</paramvalue>
    </contextparam>
    <listener>
    <listenerclass>
org.springframework.web.util.Log4jConfigListener
</listenerclass>
    </listener>
    <listener>
    <listenerclass>
org.springframework.web.context.ContextLoaderListen
er</listenerclass>
    </listener>
```

（3）代码中的applicationContext.xml是Spring的配置文件，也就是所说的Spring上下文设置的那个配置文件。该文件设置的所有bean都处于IOC容器中，代码如下：

```
    <!……文件名：
applicationContext.xml……>
    <? xml version="18.0"encoding="UTF8"? >
    <! DOCTYPE beans PUBLIC"//SPRING//DTD BEAN//EN"
    "http: //www.springframework.org/dtd/springbeans
.dtd">
```

```
<beans>
<! 具体的bean定义>
<bean id="greetingService"
class="spring.service.GreetingServiceImpl">
<! 具体的bean中属性定义>
<property name="name">
<value>Frank</value><! 具体的bean中属性值>
</property>
</bean>
</beans>
```

在上面所示代码中，每个bean的id是它们在这个容器中的唯一标识，class表示对应的类。property就是依赖注入的关键，这里通过get、set方法将所需的依赖注入到对象中。在下面的代码中我们给greetingService中的name赋值为Frank。

在该项目中笔者使用接口编程的概念编写了一个接口和实现该接口的类代码。具体代码内容如下：

```
<! ……文件名：IGreetingService.Java……>
package spring.service;
public interface IGreetingService {
public String greet（）; //声明接口方法，为实现该接口的具体类来实现方法
}
```

```
    <！……文件名：
GreetingServiceImpl.Java……>
    package spring.service;
    public class GreetingServiceImpl implements
IGreetingService {
    private String name; //配置文件中定义的name属性
    public String getName（） {//定义name属性的get方法
    return name;
    }
    //使用setter方式进行依赖注入
    public void setName（String name） {
    this.name=name;
    }
    //实现接口中greet方法具体操作，在这里是打印name属性并致
问候
    public String greet（） {
    return name+", Welcome to Spring! ";
    }
    }
```

在这两个类文件代码，使用了依赖注入中setter注入方式将属性name注入了GreetingServiceImpl类中，这样在applicationContext.xml文件中定义的name值内容就无需在代码中显式声明，而只是在xml文件中定义，也就实现了依赖注入这个Spring的基础概念。

下面编写了一个测试对象，用来测试方法greet（）是否真的能打印出xml配置文件中定义的那个name值内容“Frank”。具体测试类代码如下：

```
    <！……文件名：SpringTest.Java……>
    package spring.test;
    import org.springframework.context.support.ClassPathXmlApplicationContext;
    import spring.service.IGreetingService;
    public class SpringTest {
    //测试方法
    public static void main（String[]args） {
    //初始化IOC容器
    //ClassPathXmlApplicationContext为Spring中实现IOC容器初始化功能的自带类
    ClassPathXmlApplicationContext ctx=
    new ClassPathXmlApplicationContext（"applicationContext.xml"）；
    //利用getBean方法得到配置文件中定义的greetingService类，并使用接口封装它的实现
    IGreetingService gs=（IGreetingService） ctx.getBean（"greetingService"）；
    //打印greet方法返回内容
    System.out.println（gs.greet（））；
    }
    }
```

单击运行按钮，在控制台会显示如下信息：

```
Frank, Welcome to Spring!
```

由上可见，xml配置文件中定义的那个name值内容“Frank”，的确是被依赖注入进代码，然后被由程序中设置的打印方法将其值内容打印出来。

以上仅是Spring最基础的应用示例。详细的Spring各组成部分的示例和技术细节上的说明、介绍，可参看后面的章节。

第19章　为什么要使用控制反转

在具体介绍Spring的核心容器IOC（控制反转）和它的实现策略DI（依赖注入）之前，首先介绍一下出现这些编程方法的原因。IOC代表的是一种思想，也是一种开发模式，是解决调用者和被调用者之间的一种关系。由于它不是什么具体的方法，所以理解起来有点困难。要理解IOC的具体内涵，最简单的方式就是看它的实例。

在软件设计方法和设计模式的发展史上，在IOC和DI出现之前产生了三种类的调用方法：自己创建、工厂模式、外部注入。本节代码通过演示三种方法的具体实现来说明它们之间的具体优劣。

19.1　new——自己创建

首先介绍一下最常用的类的调用方法，即使用new关键字来实现类的调用。这种方式只是为了学习下面的知识做铺垫，形成对比，从而看出其他方式的优点。下面的案例是使用new关键字来实现一个学生类，并创建一个学生对象。

技术要点

当使用new——自己创建方式来实现类的调用，无法更换被调用者，并且要负责被调用者的整个生命周期。

实现代码

首先介绍代码的运行背景，存在两个对象：调用者学生Student和被调用者培训班Train，要设计的代码功能就是学生接受培训。在包com.cjg.spring.dao中设计接口Train，代码如下：

```
package com.cjg.spring.dao;
public interface Train {
public void acceptTrain（）; //学生接受培训
}
```

建立被调用实体类ComputerTrain.java，代码如下：

```
package com.cjg.spring.dao;
public class ComputerTrain implements Train {
//实现Train接口的方法
public void acceptTrain（） {
System.out.println（"……
ComputerTrain.acceptTrain（）……"）；打印提示信息
}
}
```

建立被调用实体类MusicTrain.java，代码如下：

```
package com.cjg.spring.dao;
public class MusicTrain implements Train {
//实现Train接口的方法
public void acceptTrain（） {
System.out.println（"……
MusicTrain.acceptTrain（）……"）；打印提示信息
}
}
```

在包com.cjg.spring.manage中建立调用实体类NewStudent.java，代码如下：

```
package com.cjg.spring.manage;
import com.cjg.spring.dao.ComputerTrain;
import com.cjg.spring.dao.MusicTrain;
public class NewStudent {
//接受计算机培训
public void acceptComputTrain() {
ComputerTrain Train=new ComputerTrain(); //创建一个计算机学员对象
Train.acceptTrain();
}
//接受音乐培训
public void acceptMusicTrain() {
MusicTrain Train=new MusicTrain(); //创建一个音乐学员对象
Train.acceptTrain();
}
}
```

在包com.cjg.spring.client中建立客户端NewClinet.java，代码如下：

```
package com.cjg.spring.client;
import com.cjg.spring.manage.Student;
public class NewClinet {
public static void main(String[]args) {
NewStudent student=new NewStudent(); //创建学生对象
```

```
student.acceptComputTrain（）; //接受计算机培训
student.acceptMusicTrain（）; //接受音乐培训
}
}
```

运行该测试类NewClient，结果如下：

```
……ComputerTrain.acceptTrain（）……
……MusicTrain.acceptTrain（）……
```

源程序解读

（1）Student要培训ComputerTrain，就要定义acceptComputTrain（），并自己创建ComputerTrain的对象。

（2）Student要培训MusicTrainTrain，就要定义acceptMusicTrain（），并自己创建MusicTrain的对象。

（3）建立一个测试客户端NewClinet.java来创建一个NewStudent对象，然后调用

acceptComputTrain（）和acceptMusicTrain（）来实现计算机培训和音乐培训。

注意：对象student通过new的方式创建ComputerTrain的对象和MusicTrain的对象，并负责这两个对象的一切。

上述代码的简单图示如图19.1所示。

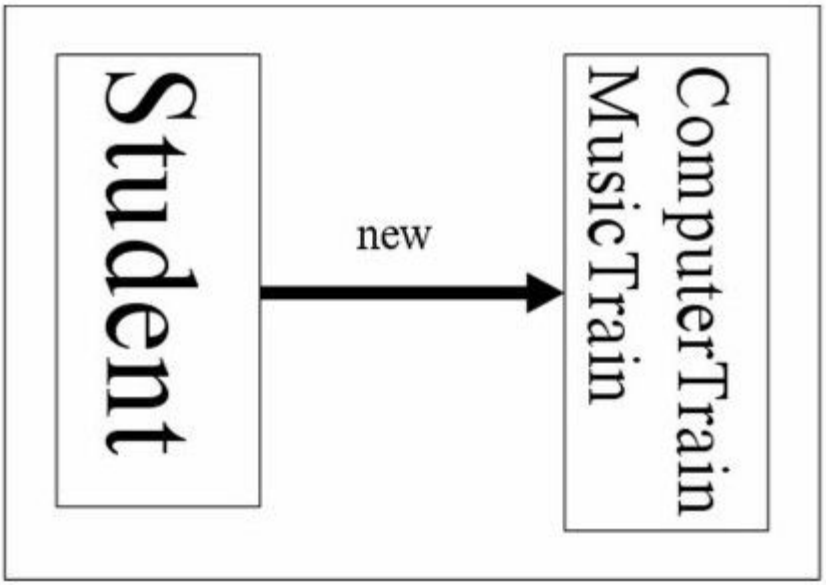

图　19.1　自己创建方式

19.2 get——工厂模式

从上一节的实例看出，使用new自己创建时，每次调用都需要自己来创建对象，这样需要创建的对象就会很多，造成管理上的不便。为了解决new——自己创建方式的弊端，出现了另外一种模式即get——工厂模式。使用该模式可以创建一个“工厂”来统一管理。

技术要点

当使用get——工厂模式来实现类调用的好处就是，实现了对象的统一创建，调用者无须关心对象的过程，只管从工厂中取得即可。

实现代码

首先介绍代码的运行背景，存在两个对象：调用者学生Student和被调用者培训班Train，要设计的代

码功能就是学生接受培训。首先在包

com.cjg.spring.dao中建立一个培训工厂

TrainFactory，代码如下：

```
    package com.cjg.spring.dao;
    public class TrainFactory {
    //培训工厂
    public static Train getComputerTrain() {
    Train train=new ComputerTrain(); //创建
ComputerTrain对象
    return train; //返回train对象
    }
    public static Train getMusicTrain() {
    Train train=new MusicTrain(); //创建MusicTrain
对象
    return train; //返回train对象
    }
    }
```

在包com.cjg.spring.manage中建立调用实体类

GetStudent，代码如下：

```
    package com.cjg.spring.manage;
    import com.cjg.spring.dao.Train;
    import com.cjg.spring.dao.TrainFactory;
    public class GetStudent {
    //接受计算机培训
    public void acceptComputTrain() {
```

```
    Train
train=TrainFactory.getComputerTrain（）；//调用工厂方
法
    train.acceptTrain（）；//获取信息
    }
    //接受音乐培训
    public void acceptMusicTrain（） {
    Train train=TrainFactory.getMusicTrain（）；//调
用工厂方法
    train.acceptTrain（）；//获取信息
    }
    }
```

在包com.cjg.spring.client中建立客户端GetClinet.java，代码如下：

```
    package com.cjg.spring.client;
    import com.cjg.spring.manage.GetStudent;
    public class GetClient
     {
    public static void main（String[]args）
     {
    GetStudent student=new GetStudent（）；//建立调用
者student
    student.acceptComputerTrain（）；//student接受计
算机培训
    student.acceptMusicTrain（）；//student接受音乐培
训
    }
    }
```

运行该测试类，结果如下：

```
……ComputerTrain.acceptTrain（）……
……MusicTrain.acceptTrain（）……
```

源程序解读

（1）在包com.cjg.spring.dao中建立一个培训工厂类TrainFactory，然后为该类添加两个函数getComputerTrain（）和getMusicTrain（），用于创建ComputerTrain和MusicTrain的对象。

（2）在包com.cjg.spring.manage中建立调用实体类GetStudent，然后在该类的两个函数acceptComputerTrain（）和acceptMusicTrain（）利用工厂类取得计算机培训和音乐培训对象。

（3）建立一个测试客户端GetClinet.java来创建一个student对象，然后调用

acceptComputerTrain（）和acceptMusicTrain（）来实现计算机培训和音乐培训。

注意：工厂模式虽然比自己创建多了一个工厂类，但是却把自己创建中NewStudent类中创建对象的代码提取到了工厂类，工厂模式中的GetStudent可以直接从工厂类中取得要创建的对象。

上述代码的简单图示如图19.2所示。

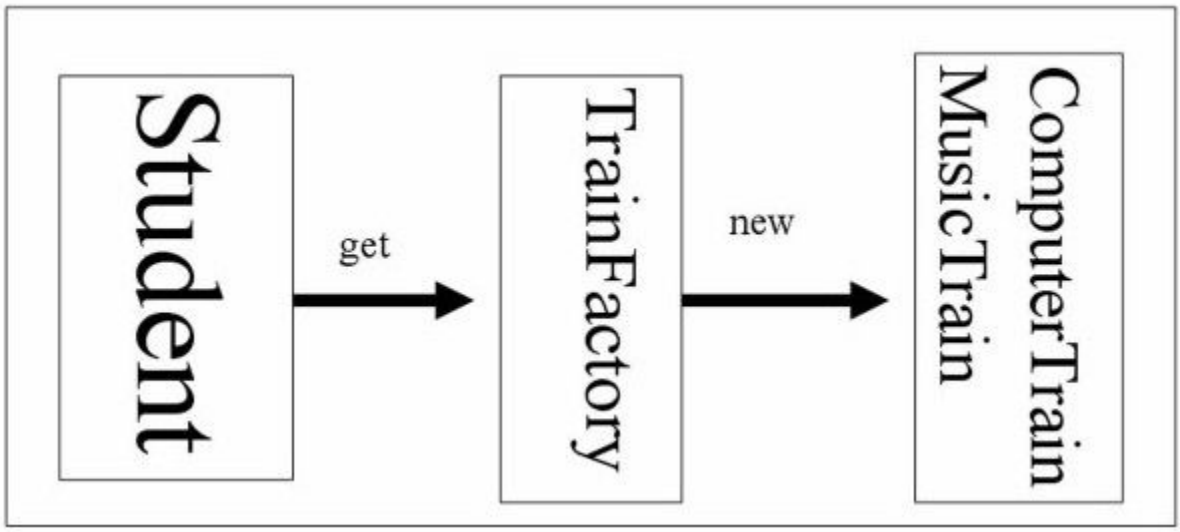

图　19.2　工厂模式

19.3 set——外部注入

在前面两节的实例中，第一个实例使用控制反转需要创建对象，第二个实例使用控制反转需要创建“工厂”，他们都具有依赖性。为了取消这种依赖性，也就是不需要创建任何东西就能使用控制反转，从而出现了set外部注入。

技术要点

当使用set——外部注入方式来实现类调用的好处就是没有依赖性。不用再像get工厂模式一样需要创建一个类。

实现代码

首先介绍代码的运行背景，存在两个对象：调用者学生Student和被调用者培训班Train，要设计的代

码功能就是学生接受培训。首先在包

com.cjg.spring.manage中建立一个调用实体类

SetStudent，代码如下：

```
package com.cjg.spring.manage;
import com.cjg.spring.dao.Train;
import com.cjg.spring.dao.TrainFactory;
//创建SetStudent实体类
public class SetStudent {
//定义以Train对象为参数的方法
public void acceptTrain（Train train）
 {
train.acceptTrain（）; //使用Train对象调用方法
}
}
```

在包com.cjg.spring.client中建立客户端

GetClinet.java，代码如下：

```
package com.cjg.spring.client;
import com.cjg.spring.dao.ComputerTrain;
import com.cjg.spring.dao.MusicTrain;
import com.cjg.spring.dao.Train;
import com.cjg.spring.manage.SetStudent;
public class SetClient
 {
public static void main（String[]args）
 {
```

```
    Train computerTrain=new ComputerTrain（）；//创建计算机对象
    Train musicTrain=new MusicTrain（）；//创建音乐对象
    SetStudent student=new SetStudent（）；//获取一个学员对象
    student.acceptTrain（computerTrain）；//以ComputerTrain对象调用
    student.acceptTrain（musicTrain）；//以musicTrain对象调用
    }
    }
```

运行该测试类，结果如下：

```
……ComputerTrain.acceptTrain（）……
……MusicTrain.acceptTrain（）……
```

源程序解读

（1）在包com.cjg.spring.manage中建立调用实体类SetStudent，其完全简化了调用实体类的方法，acceptTrain（train）的方法不再依赖于某一个特定的Train，而是使用了接口Train。

（2）建立一个测试客户端SetClinet.java，然后在该代码中创建两个Train的实现对象computerTrain和musicTrain，接着再创建一个SetStudent类对象student，最后，把对象computerTrain和musicTrain传入到对象student的方法acceptTrain（）。

上述代码的简单图示如图19.3所示。

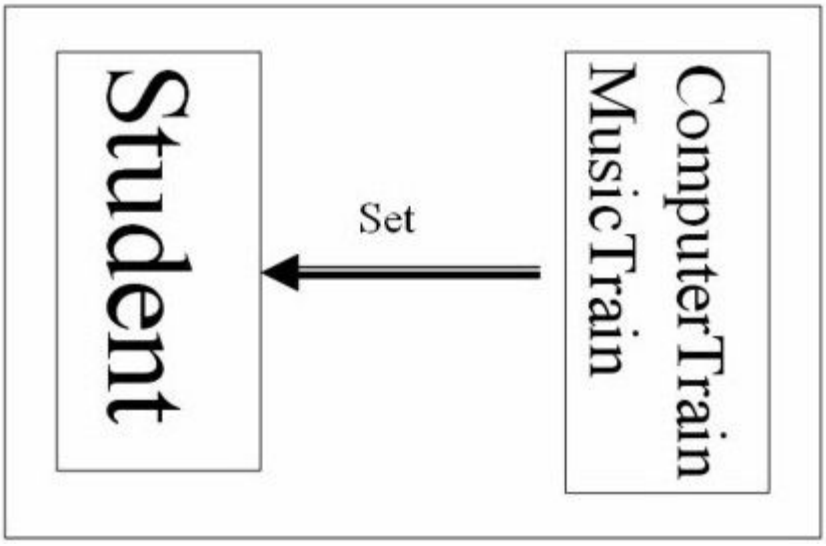

图　19.3　外部注入方式

注意：自己创建的方式依赖于被调用者对象，工厂模式的方式依赖于工厂，这两种方式都存在依赖

性。外部注入方式却完全抛开了依赖关系，实现了自由的外部注入。通过SetStudent类中的acceptTrain（）使用了接口类作为参数，使得该类的对象student完全解脱了与某一种Train对象的依赖关系。

第20章　IOC容器的反射机制和装载机制

本章介绍Spring的核心容器IOC（控制反转），IOC就是由容器来控制业务对象之间的依赖关系，而非传统方式中的由new关键字来实现。IOC的本质就是控制权由应用代码转到了外部容器，控制权的转移就是所谓的反转。

IOC中最基本的Java技术手段就是“反射”编程，“反射”即根据给出的类名来生成对象的方法。这种“反射”的编程方式可以让对象在生成时才决定要生成哪一种对象。下面就通过“反射”编程来了解一下IOC的运行机制。IOC容器利用Java反射机制创建类、调用方法等，它是如何把它们转载起来呢？本章将详细介绍。

20.1　操作构造函数

本节介绍如何利用反射机制来实现操作构造函数。构造函数是创建对象的方式，在下面的实例中同样还是通过对学生类和培训班类的操作来说明。同样，使用构造函数实现反射机制也是一个基础，为了和后面学习的内容形成对照。

技术要点

获取构造函数方法使用。

构造函数的使用。

实现代码

首先介绍代码的运行背景，存在两个对象：调用者学生Student和被调用者培训班Train，要设计的代码功能就是学生接受培训。在包

com.cjg.reflection.train中设计一个培训机构类TestTrain，代码如下：

```
package com.cjg.reflection.train;
public class TestTrain {
private String name; //新报名学生的姓名
private int age; //新报名学生的年龄
private static int total=20; //培训机构的总人数
public TestTrain（） {//定义空构造器
super（）; //调用父构造器
total++; //培训机构总人数加1
}
public TestTrain（String name, int age） {//定义一个有参构造器
super（）; //调用父构造器
this.name=name;
this.age=age;
total++; //总人数加1
}
public String getName（） {//定义学生姓名的get方法和set方法
return name;
}
public void setName（String name） {
this.name=name;
}
public int getAge（） {//定义学生年龄的get方法和set方法
return age;
}
public void setAge（int age） {
this.age=age;
}
```

```
    public static int getTotal（） {//定义学生总数的
get方法和set方法
    return total;
    }
    public static void setTotal（int total） {
    TestTrain.total=total;
    }
    public void setAll（String name, int age） {//显
示学生的信息
    this.name=name;
    this.age=age;
    }
    public static void showTotal（） {//显示培训机构的
招收学生的总人数
    System.out.println（"total="+total）;
    }
    public String toString（） {
    return"Name: "+name+"＼tAge: "+age; //新报名学生的
信息
    }
    }
```

在包com.cjg.reflection.refexample中，设计利用构造函数来建立实例的类Constractreflection，代码如下：

```
    package com.cjg.reflection.refexample;
    import com.cjg.reflection.train.TestTrain;
    import java.lang.reflect.Constructor;
    import java.lang.reflect.Field;
    import java.lang.reflect.Method;
    import java.lang.reflect.Modifier;
```

```
    import java.util.Scanner;
    public class Constractreflection {
    public static void main（String[]args） {
    Class c;
    try {
    c=Class.forName（"com.cjg.reflection.train.Test
Train"）;
    //调用有参数的构造方法
    Class[]argTypes= {String.class, int.class};
    Constructor
constructor=c.getConstructor（argTypes）;
    Object obj=constructor.newInstance（"张三",
20）;
    System.out.println（obj）;
    constructor=c.getConstructor（）; //调用无参构造方
法1
    obj=constructor.newInstance（）;
    System.out.println（obj）;
    obj=c.newInstance（）; //调用无参构造方法2
    System.out.println（obj）;
    } catch（Exception e） {
    e.printStackTrace（）;
    }
    }
    }
```

运行该类，结果如下：

```
    Name: 张三Age: 20
    Name: nul Age: 0
    Name: nul Age: 0
```

源程序解读

（1）首先该段代码的第一句就是获取所要操作类的对象，为什么要写成上面的形式呢？这是因为假如写成下面的形式：

```
    Class
c=Class.forName（"com.cjg.reflection.train.TestTrai
n"）;
```

当使用这种形式时，装入类涉及的所有工作都必须在内部进行。如果写成下面的形式：

```
    Class c=null;
    try {
    c=Class.forName（name）;
    } catch（ClassNotFoundException ex） {
    }
```

当在运行时需要从某些外部源读取类名时，就需要使用上面的形式。同时还有一个好处是如果已经装

入类，就会得到现有类的信息；如果类未被装入，就会抛出异常。

（2）如何使用类构造函数，java.lang.Class提供了4种独立的方法：

Constructor[]getConstructors（）；

Constructor getConstructor（Class[]params）；

Constructor getDeclaredConstructor（Class[]params）；

Constructor[]getDeclaredConstructors（）。

上面的代码通过使用第二种方法来实现调用有参数和无参数的构造方法，同时通过Constructor类的newInstance（）也能实现调用无参数的构造方法。

20.2 get——工厂模式

本节介绍如何利用反射机制来实现操作类的字段。通过第19章学习的get工厂模式来实现上一个实例中操作类字段的功能。通过工厂模式将处理事情的责任从应用程序转移到框架，从而简化开发。

技术要点

获取字段方法使用。

字段的使用。

实现代码

首先介绍代码的运行背景，存在两个对象：调用者学生Student和被调用者培训班Train，要设计的代码功能就是学生接受培训。在包

com.cjg.reflection.refexample中，设计利用构造函数来建立实例的类Fieldreflection，代码如下：

```
    package com.cjg.reflection.refexample;
    import java.lang.reflect.Constructor;
    import java.lang.reflect.Field;
    import com.cjg.reflection.train.TestTrain;
    public class Fieldreflection {
    public static void main（String[]args）throws
NoSuchFieldException, IllegalAccessException,
Class
    NotFoundException {
    TestTrain test=new TestTrain（"张三", 20）; //新加
入的学生
    Fieldreflection t=new Fieldreflection（）; //创建
Fieldreflection对象
    System.out.println（"……"）;
    t.mf1（test, "name", "李四"）; //插入名称信息
    t.mf1（test, "age", 30）; //插入年龄信息
    System.out.println（"……"）;
    t.mf2（"com.cjg.reflection.train.TestTrain", "to
tal", 60）; //插入总人数信息
    }
    //直接操作对象字段
    public void mf1（Object o, String fieldName,
Object newValue）throws NoSuchFieldException,
IllegalAc
    cessException {
    Class c=o.getClass（）;
    Field f=c.getField（fieldName）; //获取文件名称
    Object fv=f.get（o）; //获取Object类型信息
    System.out.print（"修改前: "+fieldName+"="+fv）;
    f.set（o, newValue）; //对信息进行修改
```

```
    System.out.println（"\t修改
后："+fieldName+"="+f.get（o））；
    }
    //直接操作类字段
    public void mf2（String className, String
fieldName, Object newValue）throws
ClassNotFoundException，
    NoSuchFieldException, IllegalAccessException {
    Class c=Class.forName（className）；
    Field f=c.getField（fieldName）；//获取文件名称
    Object fv=f.get（c）；//获取Object类型信息
    System.out.print（"修改前："+fieldName+"="+fv）；
    f.set（c, newValue）；//对信息进行修改
    System.out.println（"\t修改
后："+fieldName+"="+f.get（c））；
    }
    }
```

运行该类，结果如下：

```
……
修改前：name=张三修改后：name=李四
修改前：age=20修改后：age=30
……
修改前：total=20修改后：total=60
```

源程序解读

在上面的例子中通过两种方式来设置对象实例的字段值。在mf1方法中通过o参数指定了一个对象实例（实际上，该对象为TestTrain类的对象实例），并通过Object类的getClass方法获得了Class对象。而在mf2方法中使用了类的全名作为第一个参数，并在mf2方法中通过Class的forName方法获得与该类名对应的Class对象。

mf1方法和mf2方法在为字段赋值时只有获得Class对象的方式不同，后面的代码基本类似。在得到lass对象后，通过getField方法获得指定字段的Field对象，并通过Field的get方法获得未修改之前的字段值，并通过Field的set方法修改这个字段值。

20.3　操作类的方法

本节介绍如何利用反射机制来实现操作类的方法。本节中来介绍一个通过反射机制原理来操作学生和培训班类中方法的实例。操作类的方法是和操作构造函数很相似的。

技术要点

获取方法使用。

方法的使用。

实现代码

首先介绍代码的运行背景，存在两个对象：调用者学生Student和被调用者培训班Train，要设计的代码功能就是学生接受培训。在包

com.cjg.reflection.refexample中，设计利用构造函数来建立实例的类Methodreflection，代码如下：

```
    package com.cjg.reflection.refexample;
    import
java.lang.reflect.InvocationTargetException;
    import java.lang.reflect.Method;
    import com.cjg.reflection.train.TestTrain;
    public class Methodreflection {
    public static void main（String[]args）throws
NoSuchMethodException,
    IllegalAccessException,
InvocationTargetException,
IllegalArgumentException,
    ClassNotFoundException {
    TestTrain test=new TestTrain（"张三", 20）; //新加
入的学生
    Methodreflection t=new Methodreflection（）; //
创建Methodreflection对象
    System.out.println（"……"）;
    Class[]argTypes= {String.class, int.class} ;
    Object[]test1=new Object[] {"王五", 99} ;
    t.mf1（test, "setAll", argTypes, test1）;
    System.out.println（test）;
    System.out.println（"……"）;
    t.mf2（"com.cjg.reflection.train.TestTrain", "sh
owTotal", null, null）;
    }
    //调用对象成员方法
    public void mf1（Object o, String methodName,
Class[]argTypes, Object[]args）throws NoSuchMetho
    dException, IllegalAccessException,
InvocationTargetException {
    Class c=o.getClass（）; //获得类信息
```

```
    Method m=c.getMethod（methodName, argTypes）; //获得类方法信息
    Object result=m.invoke（o, args）; //获得方法的返回结果
    System.out.println（"result: "+result）;
    }
    //调用类成员方法
    public void mf2（String className, String methodName, Class[]argTypes, Object[]args）throws Class
    NotFoundException, NoSuchMethodException, IllegalArgumentException, IllegalAccessException, Invocation
    TargetException {
    Class c=Class.forName（className）; //获得类信息
    Method m=c.getMethod（methodName, argTypes）; //获得类方法信息
    Object result=m.invoke（null, args）; //获得方法的返回结果
    System.out.println（"result: "+result）;
    }
    }
```

运行该类，结果如下：

```
……
result: null
用户名：王五  密码：99
……
total=88
result: null
```

源程序解读

（1）首先讲解一下调用对象成员方法

mf1（Object o, String methodName, Class[]argTypes, Object[]args），该方法的四个参数的意思：o代表得是类、methodName代表的是类中方法、argTypes代表的是类中方法的参数列表、args代表的是实际参数。

该方法的第一句：Class c=o.getClass（），获取类名o的对象；Method m=c.getMethod（methodName, argTypes），类o调用通过方法名methodName和参数列表argTypes决定的方法；Object result=m.invoke（o, args），返回方法的结果。

（2）接着讲解一下类成员方法mf2（String o, String methodName, Class[]argTypes,

Object[]args），该方法与mf1方法的区别就在于第一个参数的类型的不同。主要的区别就在于静态方法可以被类直接调用，而普通方法必须被类的对象来调用。该方法的流程跟mf1一样。如何获得类方法信息，java.lang.Class提供了4种的方法：

Method getMethod（String name, Class[]params）；

Method[]getMethod（）；

Method getDeclaredMethod（String name, Class[]params）；

Method[]getDeclaredMethod（）。

上面的两个方法都通过使用第二种方法即通过使用特定的参数类型来获得命名的方法，最后通过invoke（）来获得方法返回对象。

实际上IOC容器就好比一个大工厂，只不过这个大工厂要生产的对象都在XML文件中给出定义，然后利用Java的“反射”编程，根据XML中给出的类名生成相应的对象，实现具体的功能。

本章通过三个具体的实例来讲解了反射中重要类、方法和实现机制，为后面IOC容器的讲解做了铺垫。

20.4　IOC容器装载机制

在Spring的场境中，只需要编写调用者和被调用者类的代码同时把所有的Bean放在XML配制文件中即可，而利用上一小节中的Java反射机制创建类，调用方法的过程和什么时间调用什么方法都是由IOC容器来帮助实现。

技术要点

高级抽象接口BeanFactory，实现了工厂设计模式，可以通过名称创建和检索对象即管理对象之间的关系。作为IOC容器的核心接口，负责容纳XML文件所描述的Bean，并对Bean进行管理。如果使用接口BeanFactory，要经过三个步骤：配置XML数据，实例化容器，调用BeanFactory的方法。

工厂设计模式的BeanFactory接口。

IOC容器装载的三个步骤。

实现代码

首先介绍代码的运行背景，存在两个对象：调用者学生Student和被调用者培训班Train，要设计的代码功能就是学生接受培训。在包com.cjg.spring.dao中，设计一个培训机构接口Train，代码如下：

```
package com.cjg.spring.dao;
public interface Train {
public void acceptTrain（）; //学生接受培训
}
```

在包com.cjg.spring.dao中，设计一个计算机培训机构类ComputerTrain，代码如下：

```
package com.cjg.spring.dao;
public class ComputerTrain implements Train {
public void acceptTrain（） {//实现接口的方法
```

```
    System.out.println（"……
ComputerTrain.acceptTrain（）……"）；
    }
    }
```

在包com.cjg.spring.dao中，设计一个音乐培训机构类ComputerTrain，代码如下：

```
    package com.cjg.spring.dao;
    public class MusicTrain implements Train {
    public void acceptTrain（） {//接受音乐培训
    System.out.println（"……
MusicTrain.acceptTrain（）……"）；
    }
    }
```

接着在包com.cjg.spring.manage中，设计一个学生类NewStudent.java，代码如下：

```
    package com.cjg.spring.manage;
    import com.cjg.spring.dao.Train;
    public class NewStudent {
    private Train AcceptTrain;
    public NewStudent（Train Train） {//构造函数
    this.AcceptTrain=Train;
    }
    public void acceptTrain（） {//定义接受培训方法
    this.AcceptTrain.acceptTrain（）；
    }
```

```
    }
```

最后在包com.cjg.spring.test中，设计一个测试类Client.java，代码如下：

```
    package com.cjg.spring.client;
    import
org.springframework.beans.factory.BeanFactory;
    import
org.springframework.context.ApplicationContext;
    import
org.springframework.context.support.ClassPathXmlAp
plicationContext;
    import
org.springframework.context.support.FileSystemXmlA
pplicationContext;
    import com.cjg.spring.dao.ComputerTrain;
    import com.cjg.spring.dao.Train;
    import com.cjg.spring.manage.NewStudent;
    public class AcceptStudent {
    public static void main（String[]args） {
    //装载配置文件，并创建管理工厂
    BeanFactory factory=new
ClassPathXmlApplicationContext（"applicationContext
.xml"）;
    //读取工厂管理对象
    NewStudent newStudent=（NewStudent）
factory.getBean（"NewStudent"）;
    newStudent.acceptTrain（）; //调用对象的方法
    }
    }
```

运行该类，结果如下：

```
……ComputerTrain.acceptTrain（）……
```

源程序解读

（1）在最后两个步骤中间，还有一个重要的文件（applicationContext.xml）要配置。其部分内容如下：

```
    <！创建对象ComputerTrain>
    <bean
id="ComputerTrain"class="com.cjg.spring.dao.Comput
erTrain"/>
    <！创建对象MusicTrain>
    <bean
id="MusicTrain"class="com.cjg.spring.dao.MusicTrai
n"/>
    <！创建对象NewStudent>
    <bean
id="NewStudent"class="com.cjg.spring.manage.NewStu
dent">
    <！创建依赖关系>
    <constructorarg ref="ComputerTrain"/>
```

在上面的配置文件中，＜bean/＞标记用来存储三个对象：ComputerTrain、MusicTrain、NewStudent，而＜constructorarg/＞标记则用来创建三个对象间的依赖关系。这些标记间的属性在以后章节介绍。

（2）在AcceptStudent.java文件中，有一个非常重要的接口BeanFactory。该接口在包org.springframework.beans.factory中，可以实现各种不同的存储方法。同时还要注意XML文件中定义的Bean是被消极加载的，即在得到接口BeanFactory后需要Bean之前，Bean本身不会被初始化，如果想要检索到所需的Bean，就需要调用参数是所需要的Bean的getBean（）方法。例如：

```
   Train train=（Train）
factory.getBean（"ComputerTrain"）
```

查看API文档，可以发现接口BeanFactory提供了一些非常有用的方法来进行Bean编辑，它们分别如下所示。

boolean containsBean（String）：如果包含指定名称String的Bean定义，则返回true；否则相反。

Object getBean（String）：返回给定名称注册String的Bean实例。如果没有找到指定的Bean，将会抛出一个异常。

Object getBean（String, Class）：返回给定名称注册String的Bean实例，并转换为给定类型class。如果转换失败，将会抛出一个异常。

Class getType（String name）：返回给定名称注册Bean的Class。如果没有找到指定的Bean实例，将会抛出一个异常。

boolean isSingleton（String）：如果给定名称的Bean定义是singleton模式，则返回true；否则相反。

String[]getAliases（String）：返回给定Bean名称的所有别名。

（3）得到所需要的Bean后，接着就可以调用该Bean的方法：

```
train.acceptTrain（）
```

（4）上面的三个步骤也就是使用接口BeanFactory的三个步骤：配置XML数据、实例化容器、调用BeanFactory提供的方法。在实例化容器这一步骤中可以用三种方式来实现。

方法一：

```
```

```
    Resource resource=new
FileSystemResource（"xxx.xml"）;
    BeanFactory factory=new
XmlBeanFactory（resource）;
```

方法二：

```
    ClassPathResource resource=new
ClassPathResource（"xxx.xml"）;
    BeanFactory factory=new
XmlBeanFactory（resource）;
```

方法三：

```
    ApplicationContext context=new
ClassPathXmlApplicationContext（new String[]
（"xxx.xml", "xxx
    part1.xml"））;
    BeanFactory factory=（BeanFactory）context;
```

注意：三种方法的环境，编写的XML文件为xxx.xml，并且放在默认的目录下。同时还要注意第三种方法可以用来加载多个XML文件配置。

（5）ApplicationContext类是BeanFactory接口的扩展，其覆盖了BeanFactory的所有功能并提供了更多自己的功能，所以API文档建议优先使用ApplicationContext类。查看API文档，可以发现该类的getMessage方法提供了处理消息的功能，该方法有三种重载形式。

String getMessage（Strin code, Object[]args, String default, Locale loc）：用来从资源文件loc中获取消息。如果在loc中没有找到消息，则使用默认的消息。

String getMessage（Strin code, Object[]args, Locale loc）：与上一个方法基本相同，区别是少了一个用于指定默认值的参数default。如果没有找到消息，就会抛出一个异常。

String getMessage（MessageSourceResolvable, Locale locale）：可以指定MessageSourceResolvable实现。

IOC容器装载的三个步骤中，通常用简单直观的XML来作为配置数据，用类ApplicationContext来实例化应用中的各种对象。

第21章　DI注入方式

DI策略是实现IOC容器最合适的策略，该策略让容器全权负责依赖，受控对象只要暴露属性和带参数的构造函数，在初始化对象时就可以设置对象间的依赖关系。依赖关系的确定不依赖于特定的API和接口等，而是语言本身就可以实现。DI的意思就是依赖先剥离，然后在适当时再注射进入。DI类型分别有接口注入、构造注入和设置注入，本章通过具体的实例一一介绍这些类型，同时还要注意各种类型是如何实现注入的。

21.1　设值注入

在IOC容器中，一般都会使用构造注入和设值注入方式，因为接口注入方式会具有侵入性，所以接口注入方式一般不用。如果想实现设值注入方式，只要在

对象中提供JavaBean标准的属性即为调用者类添加setter方法就完全可以了。设值注入模式是实际开发中应用最多的。

技术要点

当采用设值注入方式来实现依赖关系时，该种依赖关系是通过受控对象的属性来表达自己所依赖的对象和所需配置的值，IOC容器就是通过调用类的setter方法，将所需要的依赖关系注入其中。

XML文件中＜Bean＞标记及其子元素的配置。

设值注入方式。

实现代码

首先介绍代码的运行背景，存在两个对象：调用者学生Student和被调用者培训班Train，要设计的代

码功能就是学生接受培训。在包com.cjg.spring.dao中，设计一个培训机构类Train，代码如下：

```
    package com.cjg.spring.dao;
    public class Train {
    private String trainname; //设计培训机构名称字段
    public String getTrainname() {//定义培训机构名称字段的get方法
    return trainname;
    }
    public void setTrainname(String trainname) {//定义培训机构名称字段的set方法
    this.trainname=trainname;
    }
    }
```

接着在包com.cjg.spring.manage中，设计一个学生类NewStudent.java，代码如下：

```
    package com.cjg.spring.manage;
    import com.cjg.spring.dao.Train;
    public class NewStudent {
    private String name; //学生名字字段
    private Train trainName; //培训机构名称字段
    public String getName() {//定义学生名字字段的get方法
    return name;
    }
    public void setName(String name) {//定义学生名字字段的set方法
```

```
    this.name=name;
    }
    public Train getTrainName（） {//定义培训机构名称字段的get方法
    return trainName;
    }
    public void setTrainName（Train trainName） {//定义培训机构名称字段的set方法
    this.trainName=trainName;
    }
    }
```

最后在包com.cjg.spring.test中，设计一个测试类Client.java，代码如下：

```
    package com.cjg.spring.test;
    import
org.springframework.context.ApplicationContext;
    import
org.springframework.context.support.FileSystemXmlApplicationContext;
    import com.cjg.spring.manage.NewStudent;
    public class Client {
    public static void main（String[]args） {
    //读取配置文件
    ApplicationContext context=
    new
FileSystemXmlApplicationContext（"src/applicationContext.xml"）;
    //得到所需要的Bean
    NewStudent student=（NewStudent）
context.getBean（"newstudent"）;
    //调用Bean中的方法
```

```
    System.out.println（student.getName（））；
    System.out.println（student.getTrainName（）.getT
rainname（））；
    }
    }
```

运行该类，结果如下：

```
    cjg
    计算机培训
```

源程序解读

（1）在最后两个步骤中间，还有一个重要的文件（applicationContext.xml）要配置。其具体内容如下：

```
    <? xml version="1.0"encoding="UTF8"? >
    <beans>
    <! 创建对象train>
    <bean
id="train"class="com.cjg.spring.dao.Train"abstract=
"false"
    singleton="true"lazyinit="default"autowire="defa
ult"
    dependencycheck="default">
    <property name="trainname"value="计算机培训"/>
    </bean>
```

```
    <! 创建对象newstudent>
    <bean
id="newstudent"class="com.cjg.spring.manage.NewStud
ent"abstract="false"
    singleton="true"lazyinit="default"autowire="defa
ult"
    dependencycheck="default">
    <property name="name">
    <value>cjg</value>
    </property>
    <! 创建依赖关系>
    <property name="trainName">
    <ref bean="train"/>
    </property>
    </bean>
    </beans>
```

在上面的配置文件中，<bean/>标记用来存储两个对象：train和newstudent对象。现在讲解一下<bean>标记的属性，如表21.1所示。

表 21.1　<bean>标记的属性

属 性 名	是否可选	作　用
id	必选	用来命名 Bean
class	必选	需要实例化的类
scope	可选	Bean 的作用域
lazy-ini	可选	是否在启动时启动 bean
autowire	可选	自动装载的方式

各属性的详细解释和用法如下。

id属性：Bean的命名，其采用标准的Java命名规定，即小写字母开头，如myName等。同时要注意在IOC容器总id必须是唯一的。如下例所示。

```
    <bean id="train"……/>
```

class属性：指定用来实例化的类名。如下例所示。

```
    <bean
id="train"class="com.cjg.spring.manage.NewStudent"……/>
```

其他的三个属性后面章节会介绍。

要进行Bean注入就需要给＜bean＞设置被注入的对象，即给＜bean＞配置子元素。注入方式为设值注入时，是通过＜property＞元素来实现。在＜property＞元素中属性name表示要设值的属性名称，其值类型可以用三种方式表示：

用value属性直接设置。

用＜value＞子元素来设置。

用＜ref＞子元素指向另一个＜bean＞。

如下例所示：

```
<property name="trainname"value="计算机培训"/>
```

设置培训机构类Train中的trainname字段的值为计算机培训。

```
<property name="name">
<value>cjg</value>
</property>
```

设置学生类NewStudent中的name字段的值为cjg。

```
<property name="trainName">
<ref bean="train"/>
</property>
```

把学生类NewStudent中的trainName字段的值转向Bean对象train。

（2）在设值注入中，必须要为调用者对象学生NewStudent设置字段并同时添加getter/setter函数，才能实现设值注入。例如NewStudent.java文件中的代码：

```
    private String name; //学生名字字段
    private Train trainName; //培训机构名字字段
    public String getName（） {//定义学生名字字段的get方法
    return name;
    }
    public void setName（String name） {//定义学生名字字段的set方法
    this.name=name;
    }
    public Train getTrainName（） {//定义培训机构名字字段的get方法
    return trainName;
    }
    public void setTrainName（Train trainName） {//定义培训机构名字字段的set方法
    this.trainName=trainName;
    }
```

（3）通过上面两步的设置后，就可以用测试类Client.java文件中的方式来构建NewStudent对象student，Train对象就会被Spring的IOC容器创建并利用设值注入方式注入到NewStudent类中。这样就可以通过对象student的getTrainName（）属性获得对象Train，然后就可以调用对象Train的方法。

```
    ApplicationContext context=new
FileSystemXmlApplicationContext（"src/applicationCo
ntext.xml"）；
    NewStudent student=（NewStudent）
context.getBean（"newstudent"）；
    System.out.println（student.getName（））；
    System.out.println（student.getTrainName（）.getT
rainname（））；
```

21.2　构造注入

本节介绍如何使用构造方式来实现注入。在本节中将介绍一个通过构造方式来对学生类和培训班实现注入的实例。通过使用构造注入，容器可以通过调用类的构造方法，将其依赖关系注入其中。

技术要点

采用构造注入方式来实现依赖关系时，该种依赖关系是通过类构造函数建立的，IOC容器就是通过调用类的构造方法，将所需要的依赖关系注入其中。

构造注入方式的XML文件配置。

＜constructorarg＞标记的使用。

实现代码

首先介绍代码的运行背景，存在两个对象：调用者学生Student和被调用者培训班Train，要设计的代码功能就是学生接受培训。在包com.cjg.spring.dao中，设计一个培训机构类Train，代码如下：

```
    package com.cjg.spring.dao;
    public class Train {
    private String trainname; //定义一个培训机构名称字段
    public String getTrainname() {//定义培训机构名称字段的get方法
    return trainname;
    }
    public void setTrainname(String trainname) {//定义培训机构名称字段的set方法
    this.trainname=trainname;
    }
    }
```

接着在包com.cjg.spring.manage中，设计一个学生类NewStudent.java，代码如下：

```
    package com.cjg.spring.manage;
    import com.cjg.spring.dao.Train;
    public class NewStudent {
    private String name; //定义一个学生名字字段
    private Train trainname; //定义一个培训机构名称字段
```

```
    //拥有一个参数的构造函数
    public NewStudent（Train trainname） {
    this.trainname=trainname;
    }
    public String getName（） {//定义学生名字字段的get
方法
    return name;
    }
    public void setName（String name） {//定义学生名字
字段的set方法
    this.name=name;
    }
    public Train getTrainname（） {//定义培训机构名称的
get方法
    return trainname;
    }
    public void setTrainname（Train trainname） {//
定义培训机构名称的set方法
    this.trainname=trainname;
    }
    }
```

最后在包com.cjg.spring.test中，设计一个测试类Client.java，代码如下：

```
    package com.cjg.spring.test;
    import
org.springframework.context.ApplicationContext;
    import
org.springframework.context.support.ClassPathXmlAp
plicationContext;
    import com.cjg.spring.manage.NewStudent;
    public class Client {
```

```
    public static void main（String[]args） {
    //读取配置文件
    ApplicationContext context=
    new
ClassPathXmlApplicationContext（"applicationContext
.xml"）;
    //获取所需要的Bean
    NewStudent student=（NewStudent）
context.getBean（"newstudent"）;
    //调用Bean中的方法
    System.out.println（student.getName（））;
    System.out.println（student.getTrainname（）.get
Trainname（））;
    }
    }
```

运行该类，结果如下：

```
    cjg
    计算机培训
```

源程序解读

（1）在最后两个步骤中间，还有一个重要的文件（applicationContext.xml）要配置。其具体内容如下：

```
    <? xml version="1.0"encoding="UTF8"? >
    <beans>
    <! 创建对象train>
    <bean
id="train"class="com.cjg.spring.dao.Train"abstract
="false"
    singleton="true"lazyinit="default"autowire="def
ault"
    dependencycheck="default">
    <! 配置培训班名称变量>
    <property name="trainname">
    <value>计算机培训</value><! 配置培训班名称变量值
>
    </property>
    </bean>
    <! 创建对象newstudent>
    <bean
id="newstudent"class="com.cjg.spring.manage.NewStu
dent"abstract="false"
    singleton="true"lazyinit="default"autowire="con
structor"
    dependencycheck="default">
    <! 配置学生名称变量>
    <property name="name">
    <value>cjg</value><! 配置学生名称变量值>
    </property>
    </bean>
    </beans>
```

在上面的配置文件中，<bean/>标记用来存储两个对象：Train和NewStudent，而对象Train利用构造函数注入到对象NewStudent中。

```
    <bean
id="newstudent"class="com.cjg.spring.manage.NewStu
dent"abstract="false"
    singleton="true"lazyinit="default"autowire="con
structor"
    dependencycheck="default">
```

在上面的<bean>的属性中比设值注入的方式多了一个autowire="constructor"属性。这就决定了使用构造注入。该属性的具体介绍在后面的章节。

（2）在<bean>中也可以使用子元素<constructorarg>来实现构造注入，其有三种方式：输入参数的类型用type表示，输入参数的值用value表示。

指向其他的Bean可以用ref属性。

指向其他的Bean可以用<ref>子元素。

上面id=“newstudent”可以这样配置，如下所示：

```
    <bean
id="newstudent"class="com.cjg.spring.manage.NewStu
dent"abstract="false"
    singleton="true"lazyinit="default"autowire="def
ault"
    dependencycheck="default">
    <constructorarg ref="train"/>
    <property name="name">
    <value>cjg</value><! 配置学生名称变量值>
    </property>
    </bean>
```

同时也可以这样配置，如下所示：

```
    <bean
id="newstudent"class="com.cjg.spring.manage.NewStu
dent"abstract="false"
    singleton="true"lazyinit="default"autowire="def
ault"
    dependencycheck="default">
    <constructorarg>
    <ref bean="train"/>
    </constructorarg>
    <property name="name">
    <value>cjg</value><! 配置学生名称变量值>
    </property>
    </bean>
```

两种配置实现的效果是一样的。假如构造函数的类型都是简单的类型，如下所示：

```
    public class NewStudent {
    private String name; //定义字符类型的name字段
    private int num; //定义int类型的num字段
    public NewStudent（String name, int num）{//构造方法
    this.name=name;
    this.num=num;
    }
    }
```

那么其配置文件可以如下所示：

```
    ＜bean id="newstudent"class="com.cjg.spring.manage.NewStudent"abstract="false"
    singleton="true"lazyinit="default"autowire="default"
    dependencycheck="default"＞
    ＜constructorarg type="String"value="cjg"/＞＜!配置学生名称变量值＞
    ＜constructorarg type="int"value="20"/＞＜! 配置学生年龄变量值＞
    ＜/bean＞
```

除了采用命名参数外，还可以通过index属性指定构造函数参数出现的顺序。上面的配置文件也可以如下所示：

```
    <bean
id="newstudent"class="com.cjg.spring.manage.NewStu
dent"abstract="false"
    singleton="true"lazyinit="default"autowire="def
ault"
    dependencycheck="default">
    <constructorarg index="0"value="cjg"/><! 配置
第一个变量值>
    <constructorarg index="1"value="20"/><! 配置第
二个变量值>
    </bean>
```

注意：index属性值从0开始。

（3）通过对applicationContext.xml文件设置，就可以用测试类Client.java文件中的方式来构建NewStudent对象student, Train对象就会被Spring的IOC容器创建并利用构造注入方式注入到NewStudent类中。这样就可以通过对象student的getTrainName（）属性获得对象Train，然后就可以调用对象Train的方法。

21.3　集合类型注入

函数的字段类型如果是普通类型如Int、String等时，前面两节中配置文件出现的标签就可以实现值注入；如果为集合时，在XML文件如何实现对它们的注值呢？

技术要点

在XML文件中，可以通过＜list/＞、＜set/＞、＜list/＞和＜map/＞标签来注入设置与集合类型相对应的List、Set、数组和Map集合的值。

List集合类型。

Set集合类型。

数组集合类型。

Map集合类型。

实现代码

首先在包com.cjg.spring中，设计一个类ComAttribute，代码如下：

```
    package com.cjg.spring;
    import java.util.List;
    import java.util.Map;
    import java.util.Set;
    public class ComAttribute {
    private List listValue;
    private Set setValue; //定义一个Set集合类型字段
    private String[]arrayValue; //定义一个数组类型字段
    public List getListValue() {//定义listValue的get方法
    return listValue;
    }
    public void setListValue(List listValue) {//定义listValue的set方法
    this.listValue=listValue;
    }
    public Set getSetValue() {//定义setValue的get方法
    return setValue;
    }
    public void setSetValue(Set setValue) {//定义setValue的set方法
    this.setValue=setValue;
    }
```

```
    public String[]getArrayValue（）｛//定义
arrayValue的get方法
    return arrayValue;
    }
    public void setArrayValue（String[]arrayValue）
｛//定义arrayValue的set方法
    this.arrayValue=arrayValue;
    }
    public Map getMapValue（）｛//定义mapValue的get方
法
    return mapValue;
    }
    public void setMapValue（Map mapValue）｛//定义
mapValue的set方法
    this.mapValue=mapValue;
    }
    }
    private Map mapValue; //定义一个Map集合类型字段
```

接着在文件test中的包com.cjg.test中，设计一个测试类TestComAttribute.java，代码如下：

```
    package com.cjg.test;
    import
org.springframework.beans.factory.BeanFactory;
    import
org.springframework.context.support.ClassPathXmlAp
plicationContext;
    import com.cjg.spring.ComAttribute;
    import junit.framework.TestCase;
    public class TestComAttribute extends
TestCase {
    private BeanFactory factory;
```

```
    //重写TestCase父类方法
    protected void setUp（）throws Exception {
    factory=new
ClassPathXmlApplicationContext（"applicationContext
ComAttribute.xml"）;
    }
    public void testInjection1（） {
    //调用工厂Bean
    ComAttribute comAttribute=（ComAttribute）
factory.getBean（"comAttribute"）;
    //调用方法调用后值
    System.out.println（"comAttribute.listValue="+c
omAttribute.getListValue（））;
    System.out.println（"comAttribute.setValue="+co
mAttribute.getSetValue（））;
    System.out.println（"comAttribute.arrayValue="+
comAttribute.getArrayValue（））;
    System.out.println（"comAttribute.mapValue="+co
mAttribute.getMapValue（））;
    }
    }
```

运行该类，结果如下：

```
    comAttribute.listValue=[list__1, list__2]
    comAttribute.setValue=[set__1, set__2]
    comAttribute.arrayValue=[Ljava.lang.String;
@1c74f37
    comAttribute.mapValue= {key1=value1,
key2=value2}
```

源程序解读

（1）在最后两个步骤中间，还有一个重要的文件（applicationContext.xml）要配置。其部分内容如下：

```
    <bean
id="comAttribute"class="com.cjg.spring.ComAttribut
e">
    <property name="listValue">
    <list>
    <value>list__1</value><! 设置list类型的第一个值
>
    <value>list__2</value><! 设置list类型的第二个值
>
    </list>
    </property>
    <property name="setValue">
    <set>
    <value>set__1</value><! 设置set类型的第一个值>
    <value>set__2</value><! 设置set类型的第二个值>
    </set>
    </property>
    <property name="arrayValue">
    <list>
    <value>array__1</value><! 设置array类型的第一个
值>
    <value>array__2</value><! 设置array类型的第二个
值>
    </list>
    </property>
    <property name="mapValue">
    <map>
    <entry key="key1"value="value1"/><! 设置map类型
的第一个值>
```

```
    <entry key="key2"value="value2"/><! 设置map类型的第二个值>
    </map>
    </property>
    </bean>
    </beans>
```

在上面的配置文件中，<bean/>标记用来存储一个对象：comAttribute，而<property>及其子标记则用注入各种集合的值。

（2）查看applicationContext.xml文件，可以看出通过<list>标记及其子标记<value>来实现对List集合值的注入；通过<set>标记及其子标记<value>来实现对Set集合值的注入；通过<list>标记及其子标记<value>来实现对数组集合值的注入；通过<map>标记及其子标记<entry>来实现对Map集合值的注入。

注意：标记<map>中子标记<entry>中key和value的值不能是bean、list、set、map和value元

素；同时标记＜set＞中子标记＜value＞的值也不能是上面的元素。

21.4　自定义类型注入

函数的字段类型如果是日期型或自定义类型时，如果把它们当做普通类型即用普通的方式注入值时，就会出现类型转换的错误，那么在XML文件中如何实现对它们的注值呢？

技术要点

当函数的字段类型是int型等普通类型或集合类型时，Spring已经内置了属性编辑器，用来把配置文件中的字符串转换成相应的类型为相对应的类型注入值。同时，程序员也可以自己编写属性编辑器，来实现把字符型变量转换成日期型或自定义类型。

为什么要用属性编辑器。

使用属性编辑器。

实现代码

首先在包com.cjg.spring中，设计一个培训机构类AttributeEditor，代码如下：

```
package com.cjg.spring;
import java.util.Date;
public class AttributeEditor {
private Date dateValue; //定义一个日期型字段
public Date getDateValue（） {//定义dateValue的get方法
return dateValue;
}
public void setDateValue（Date dateValue） {//定义dateValue的set方法
this.dateValue=dateValue;
}
}
```

在包com.cjg.spring中，设计一个测试类UtilDatePropertyEditor.java，代码如下：

```
package com.cjg.spring;
import java.beans.PropertyEditorSupport;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
```

```
    public class UtilDatePropertyEditor extends
PropertyEditorSupport {
    private String format="yyyyMMdd"; //定义固定字符串
格式
    //重写父类方法
    public void setAsText（String text）throws
IllegalArgumentException {
    System.out.println（"UtilDatePropertyEditor.sav
eAsText（）text="+text）;
    SimpleDateFormat sdf=new
SimpleDateFormat（format）; //定义类对象
    try {
    Date d=sdf.parse（text）; //获取时间信息
    this.setValue（d）;
    } catch（ParseException e） {
    e.printStackTrace（）;
    }
    }
    public void setFormat（String format） {
    this.format=format;
    }
    }
```

接着在文件test中的包com.cjg.test中，设计一个测试类TestAttrEditor.java，代码如下：

```
    package com.cjg.test;
    import
org.springframework.beans.factory.BeanFactory;
    import
org.springframework.context.support.ClassPathXmlAp
plicationContext;
    import com.cjg.spring.AttributeEditor;
```

```
    import junit.framework.TestCase;
    public class TestAttrEditor extends TestCase {
    private BeanFactory factory;
    //重写父类方法
    protected void setUp（）throws Exception {
    factory=new
ClassPathXmlApplicationContext（"applicationContext
.xml"）;
    }
    public void testInjection1（） {
    AttributeEditor
attributeEditor=（AttributeEditor）
factory.getBean（"attributeEditor"）;
    System.out.println（"bean1.dateValue="+attribut
eEditor.getDateValue（））;
    }
    }
```

运行该类，结果如下：

```
    UtilDatePropertyEditor.saveAsText（）
text=20080815
    bean1.dateValue=Fri Aug 15 00: 00: 00 CST 2008
```

源程序解读

（1）在最后两个步骤中间，还有一个重要的文件（applicationContext.xml）要配置。其具体内容如

下：

```
    <bean id="customEditorConfigurer"
    class="org.springframework.beans.factory.config
.CustomEditorCon
    figurer">
    <property name="customEditors">
    <map>
    <entry key="java.util.Date">
    <bean
class="com.cjg.spring.UtilDatePropertyEditor">
    <! 设置时间格式>
    <property name="format"value="yyyyMMdd"/>
    </bean>
    </entry>
    </map>
    </property>
    </bean>
    <! 创建对象attributeEditor>
    <bean
id="attributeEditor"class="com.cjg.spring.Attribut
eEditor">
    <property name="dateValue">
    <value>20080815</value><! 设置变量值>
    </property>
    </bean>
    </beans>
```

在上面的配置文件中，<bean/>标记用来存储两个对象：customEditorConfigurer和

attributeEditor，同时对象customEditorConfigurer用来把自定义的属性编辑器注入到IOC容器中。

（2）属性编辑器代码

UtilDatePropertyEdito.java中，首先要继承类PropertyEditorSupport，该类在包java.bean.PropertyEditorSupport中。接着要覆盖类PropertyEditorSupport中的setAsText方法，该方法中就是实现类型转换的具体代码。使用日期类型转换类SimpleDateFormat来实现转换成特定的自定义格式formant类型。然后通过类SimpleDateFormat的方法parse把传进来的参数实现转换。最后因为setAsText方法没有返回值，所以把SimpleDateFormat方法的返回值传入到PropertyEditorSupport方法的setValue方法。

（3）在第2步中编写完属性编辑器后，就要把该属性编辑器注入到IOC容器中。如下面代码：

```
    <bean
id="customEditorConfigurer"class="org.springframew
ork.beans.factory.
    config.CustomEditorConfigurer">
    <property name="customEditors">
    <map>
    <entry key="java.util.Date">
    <bean
class="com.cjg.spring.UtilDatePropertyEditor">
    <! 设置时间格式>
    <property name="format"value="yyyyMMdd"/>
    </bean>
    </entry>
    </map>
    </property>
    </bean>
```

首先要把自己编写的属性编辑器注入到org.springframework.beans.factory.config.CustomEditorConfigurer中，查看API文档，可以得到如下的定义：

```
    Public class CustomEditorConfigurer（） {
```

```
    Private Map customEditors; //定义customEditors字段
    Pubic void setCustomEditors（Map customEditors）{//定义customEditors字段的set方法
    this.customEditors=customEditors;
    }
    Pubic void getCustomEditors（）{//定义customEditors字段的get方法
    return customEditors;
    }
    }
```

从上面的两段代码可以看出，属性编辑器是被注入到CustomEditorConfigurer方法的customEditors中，同时要注意customEditors是Map集合。

（4）通过第2步编写属性编辑器和第3步的注入属性编辑器，在测试文件TestAttrEditor中就可以用属性编辑器来转换格式了。

第22章　如何合理地编写配置文件

通过上一章的学习可以知道，在Spring中只要是对象一般都要注入到IOC容器中。当需要注入的Bean不多时，对XML文件的编写还可以接受，可是当注入的Bean太多时，就会出现问题。如：XML太大不好查阅读，反复注入公共属性等。本章将介绍如何合理地编写注入到文件（配置文件）中。

22.1　文件的分割和提取公共属性

如果Bean可以归为几类，那么就可以把一个配置文件分成几个配置文件，每类Bean注入到一个文件中，这样便于Bean的查找和管理。如果每个Bean中都有一些相同的属性，那么在注入时，是不是每注入一个Bean共同的属性就得注入一次呢？

技术要点

当XML文件太大时，可以把一个XML文件分成两个或者更多的XML文件，这就是XML文件的分割；当好多Bean类中有公共属性时，可以把这些公共属性提取出来，只注入一次。

XML文件的分割。

公共属性的提取。

实现代码

首先在包com.cjg.spring中，设计一个培训机构类Train1.java，代码如下：

```
package com.cjg.spring;
public class Train1 {
private Train2 train2; //定义培训机构2字段
private Train3 train3; //定义培训机构3字段
private Train4 train4; //定义培训机构4字段
public Train2 getTrain2（） {//定义Train2的get方法
return train2;
}
```

```
    public void setTrain2（Train2 train2）{//定义
Train2的set方法
    this.train2=train2;
    }
    public Train3 getTrain3（）{//定义Train3的get方法
    return train3;
    }
    public void setTrain3（Train3 train3）{//定义
Train3的set方法
    this.train3=train3;
    }
    public Train4 getTrain4（）{//定义Train4的get方法
    return train4;
    }
    public void setTrain4（Train4 train4）{//定义
Train4的set方法
    this.train4=train4;
    }
    }
```

在包com.cjg.test中，设计一个培训机构类Train2.java，代码如下：

```
    package com.cjg.spring;
    public class Train2 {
    private int number; //定义一个人数字段
    private String name; //定义一个名字字段
    private String add; //定义一个地址字段
    public String getName（）{//定义名字字段的get方法
    return name;
    }
    public void setName（String name）{//定义名字字段
的set方法
```

```
    this.name=name;
    }
    public int getNumber（） {//定义人数字段的get方法
    return number;
    }
    public void setNumber（int number） {//定义人数字段
的set方法
    this.number=number;
    }
    public String getAdd（） {//定义地址字段的get方法
    return add;
    }
    public void setAdd（String add） {//定义地址字段的
set方法
    this.add=add;
    }
    }
```

在包com.cjg.test中，设计一个培训机构类Train3.java，代码如下：

```
    package com.cjg.spring;
    public class Train3 {
    private int nmuber; //定义一个人数字段
    private String name; //定义一个名字字段
    public int getNmuber（） {//定义人数字段的get方法
    return nmuber;
    }
    public void setNmuber（int nmuber） {//定义人数字段
的set方法
    this.nmuber=nmuber;
    }
    public String getName（） {//定义名字字段的get方法
```

```
    return name;
    }
    public void setName（String name）{//定义名字字段
的set方法
    this.name=name;
    }
    }
```

在包com.cjg.test中，设计一个培训机构类Train4.java，代码如下：

```
    package com.cjg.spring;
    public class Train4 {
    private int nmuber; //定义一个人数字段
    public int getNmuber（）{//定义人数字段的get方法
    return nmuber;
    }
    public void setNmuber（int nmuber）{//定义人数字段
的set方法
    this.nmuber=nmuber;
    }
    }
```

接着在文件test中的包com.cjg.spring中，设计一个测试类Client.java，代码如下：

```
    package com.cjg.spring;
    import
org.springframework.beans.factory.BeanFactory;
```

```
    import
org.springframework.context.support.ClassPathXmlApp
licationContext;
    import com.cjg.spring.Bean2;
    import junit.framework.TestCase;
    public class Client extends TestCase {
    private BeanFactory factory; //定义私有Bean工厂对象
    //重写父类方法
    protected void setUp() throws Exception {
    factory=new
ClassPathXmlApplicationContext("applicationContext
.xml");
    }
    public void test2() {
    Train1 bean1=(Train1)
factory.getBean("train1"); //得到Train1对象bean1
    //把各个属性都输出到屏幕上
    System.out.println(bean1.getTrain2().getAdd(
));
    System.out.println(bean1.getTrain2().getName(
));
    System.out.println(bean1.getTrain2().getNumber
());
    System.out.println(bean1.getTrain3().getName(
));
    System.out.println(bean1.getTrain3().getNmuber
());
    System.out.println(bean1.getTrain4().getNmuber
());
    }
    }
```

运行该类，结果如下：

```
    2号
```

```
cjg
1000
cjg
1000
1000
```

源程序解读

（1）还需要配置一个重要的文件（applicationContext.xml）。其具体内容如下：

```
    <bean
id="train1"class="com.cjg.spring.Train1">
    <property name="train2"ref="train2"/><! 设置
train2变量值>
    <property name="train3"ref="train3"/><! 设置
train3变量值>
    <property name="train4"ref="train4"/><! 设置
train4变量值>
    </bean>
    <bean
id="train2"class="com.cjg.spring.Train2">
    <property name="number"value="1000"/><! 设置
number变量值为1000>
    <property name="name"value="cjg"><! 设置name变
量值为cjg>
    <property name="add"value="2号"/><! 设置add变量
值为2号>
    </bean>
    <bean
id="train3"class="com.cjg.spring.Train3">
```

```
    <property name="number"value="1000"/><! 设置
number变量值为1000>
    <property name="name"value="cjg"><! 设置name变
量值为cjg>
    </bean>
    <bean
id="train4"class="com.cjg.spring.Train4">
    <property name="number"value="1000"/><! 设置
number变量值为1000>
    </bean>
    </beans>
```

在上面的配置文件中，<bean/>标记用来存储4个对象：train1、train2、train3和train4。同时注意train1中的属性分别注入到对象train2、train3和train4中。

（2）当需要把上面的文件分割成两个文件时，文件名为applicationContextbean1.xml的代码如下所示：

```
    <bean
id="train1"class="com.cjg.spring.Train1">
    <property name="train2"ref="train2"/><! 设置
train2变量值>
    <property name="train3"ref="train3"/><! 设置
train3变量值>
```

```
    <property name="train4"ref="train4"/><! 设置
train4变量值>
    </bean>
    </beans>
```

文件名为applicationContextotherbean.xml的代码如下所示：

```
    <bean
id="train2"class="com.cjg.spring.Train2">
    <property name="number"value="1000"/><! 设置
number变量值为1000>
    <property name="name"value="cjg"><! 设置name变
量值为cjg>
    <property name="add"value="2号"/><! 设置add变量
值为2号>
    </bean>
    <bean
id="train3"class="com.cjg.spring.Train3">
    <property name="number"value="1000"/><! 设置
number变量值为1000>
    <property name="name"value="cjg"><! 设置name变
量值为cjg>
    </bean>
    <bean
id="train4"class="com.cjg.spring.Train4">
    <property name="number"value="1000"/><! 设置
number变量值为1000>
    </bean>
    </beans>
```

当配置文件改成上面的两个文件时，只要在测试文件测试类Client.java中，将这句代码：

```
    protected void setUp（）throws Exception｛
    factory=new
ClassPathXmlApplicationContext（"applicationContext
.xml"）；
    ｝
```

改变为：

```
    protected void setUp（）throws Exception｛
    factory=new
ClassPathXmlApplicationContext（"applicationContext
    .xml"）；
    ｝
```

这样，两个配置文件都会被读入程序代码中，注意两个配置文件中的＜benad＞标记的id属性是唯一的。

（3）查看Train2、Train3和Train4对象，它们中有共同属性。可以把共同的属性提出来，实现共同属

性只需注入一次，而不需要在每个对象中都注入。

在（2）中的文件applicationContextbean1.xml不需要变化，把文件applicationContextotherbean.xml改写成如下所示：

```
    <bean id="beanAbstract"abstract="true">
    <property name="number"value="1000"/><! 设置number变量值为1000>
    </bean
    <bean id="train2"class="com.cjg.spring.train2"parent="beanAbstract">
    <property name="name"value="cjg"><! 设置name变量值为cjg>
    <property name="add"value="2号"/><! 设置add变量值为2号>
    </bean>
    <bean id="train3"class="com.cjg.spring.train3"parent="beanAbstract">
    <property name="name"value="cjg"><! 设置name变量值为cjg>
    </bean>
    <bean id="train4"class="com.cjg.spring.train4"parent="beanAbstract">
    </bean>
```

在上面的配制文件applicationContextotherbean.xml中，出现了＜bean＞标记中的两个新属性，如表22.1所示。

表 22.1　＜bean＞标记属性

属性名	作用	是否可选
abstract	标记不完整的 bean，即抽象的 bean	否
parent	继承 Bean	否

当公共属性被抽取出后，可以注入到一个抽象的bean中，这个bean只是作为一个模板，不需要自己的实例，所以不需要class属性。例如：

```
    ＜bean id="beanAbstract"abstract="true"＞
    ＜property name="number"value="1000"/＞＜! 设置number变量值为1000＞
    ＜/bean
```

在＜bean＞的众多属性中，可以用属性parenet来指定当前Bean的继承类。继承后，当前的Bean除了可以有选择地增加新的值外，还可以继承父类Bean中的属性值和方法。例如：

```
    <bean
id="train2"class="com.cjg.spring.train2"parent="bea
nAbstract">
    <property name="name"value="cjg"><! 设置name变
量值为cjg>
    <property name="add"value="2号"/><! 设置add变量
值为2号>
    </bean>
```

上面的对象train2，除了继承对象beanAbstract的属性number的值外，自己还有选择地增加了两个属性name和add。

（4）通过上面提到的两种方法，在一定程度上可以改变配置文件的阅读和大小。不管采用任何一种方法，最后都会得到正确的结果。

22.2　根据名字自动装配的配置文件

本节接着上一节的内容来讲解如何实现简化XML配置文件的配置，特别是在开发阶段的XML配置文件配置。本节将来介绍一个根据名字自动装配配置文件的实例。

技术要点

当采用构造注入方式来实现依赖关系时，该种依赖关系是通过类构造函数建立的，IOC容器就是通过调用类的构造方法，将所需要的依赖关系注入其中。Spring IOC容器可以自动协调Bean之间的依赖关系。通过对＜bean＞标记的属性autowire的配置，可以根据名字或类型来实现自动装配。

根据名字自动装配的属性值byName。

根据类型自动装配的属性值byType。

实现代码

首先在包com.cjg.spring中，设计一个培训机构类Train1.java，代码如下：

```
    package com.cjg.spring;
    public class Train1 {
    private Train2 train2; //定义培训机构2字段
    private Train3 train3; //定义培训机构3字段
    private Train4 train4; //定义培训机构4字段
    public Train2 getTrain2（） {//定义Train2的get方法
    return train2;
    }
    public void setTrain2（Train2 train2） {//定义
Train2的set方法
    this.train2=train2;
    }
    public Train3 getTrain3（） {//定义Train3的get方法
    return train3;
    }
    public void setTrain3（Train3 train3） {//定义
Train3的set方法
    this.train3=train3;
    }
    public Train4 getTrain4（） {//定义Train4的get方法
    return train4;
    }
    public void setTrain4（Train4 train4） {//定义
Train4的set方法
```

```
    this.train4=train4;
    }
    }
```

在包com.cjg.test中，设计一个培训机构类Train2.java，代码如下：

```
    package com.cjg.spring;
    public class Train2 {
    private int number; //定义一个人数字段
    private String name; //定义一个名字字段
    private String add; //定义一个地址字段
    public String getName（） {//定义名字字段的get方法
    return name;
    }
    public void setName（String name） {//定义名字字段的set方法
    this.name=name;
    }
    public int getNumber（） {//定义人数字段的get方法
    return number;
    }
    public void setNumber（int number） {//定义人数字段的set方法
    this.number=number;
    }
    public String getAdd（） {//定义地址字段的get方法
    return add;
    }
    public void setAdd（String add） {//定义地址字段的set方法
    this.add=add;
    }
```

```
    }
```

在包com.cjg.test中，设计一个培训机构类Train3.java，代码如下：

```
    package com.cjg.spring;
    public class Train3 {
    private int nmuber; //定义一个人数字段
    private String name; //定义一个名字字段
    public int getNmuber（） {//定义人数字段的get方法
    return nmuber;
    }
    public void setNmuber（int nmuber） {//定义人数字段的set方法
    this.nmuber=nmuber;
    }
    public String getName（） {//定义名字字段的get方法
    return name;
    }
    public void setName（String name） {//定义名字字段的set方法
    this.name=name;
    }
    }
```

在包com.cjg.test中，设计一个培训机构类Train4.java，代码如下：

```
    package com.cjg.spring;
```

```
public class Train4 {
private int nmuber; //定义一个人数字段
public int getNmuber（） {//定义人数字段的get方法
return nmuber;
}
public void setNmuber（int nmuber） {//定义人数字段的set方法
this.nmuber=nmuber;
}
}
```

接着在文件test中的包com.cjg.spring中，设计一个测试类Client.java，代码如下：

```
package com.cjg.spring;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.cjg.spring.Bean2;
import junit.framework.TestCase;
public class Client extends TestCase {
private BeanFactory factory; //定义私有的工厂Bean对象
//重写父类方法
protected void setUp（）throws Exception {
factory=new ClassPathXmlApplicationContext（"applicationContext.xml"）;
}
public void test2（） {
//得到Train1对象bean1
```

```
    Train1 bean1=（Train1）
factory.getBean（"train1"）;
    //把各个属性都输出到屏幕上
    System.out.println（bean1.getTrain2（）.getAdd（
））;
    System.out.println（bean1.getTrain2（）.getName（
））;
    System.out.println（bean1.getTrain2（）.getNumber
（））;
    System.out.println（bean1.getTrain3（）.getName（
））;
    System.out.println（bean1.getTrain3（）.getNmuber
（））;
    System.out.println（bean1.getTrain4（）.getNmuber
（））;
    }
    }
```

运行该类，结果如下：

```
    2号
    cjg
    1000
    cjg
    1000
    1000
```

源程序解读

（1）在最后两个步骤中间，还有一个重要的文件（applicationContext.xml）要配置。其具体内容如下：

```
    <bean
id="train1"class="com.cjg.spring.Train1">
    <property name="train2"ref="train2"/><! 设置
train2变量值>
    <property name="train3"ref="train3"/><! 设置
train3变量值>
    <property name="train4"ref="train4"/><! 设置
train4变量值>
    </bean>
    <bean
id="train2"class="com.cjg.spring.Train2">
    <property name="number"value="1000"/><! 设置
number变量值为1000>
    <property name="name"value="cjg"><! 设置name变
量值为cjg>
    <property name="add"value="2号"/><! 设置add变量
值为2号>
    </bean>
    <bean
id="train3"class="com.cjg.spring.Train3">
    <property name="number"value="1000"/><! 设置
number变量值为1000>
    <property name="name"value="cjg"><! 设置name变
量值为cjg>
    </bean>
    <bean
id="train4"class="com.cjg.spring.Train4">
    <property name="number"value="1000"/><! 设置
number变量值为1000>
```

```
    </bean>
    </beans>
```

如果使用根据名字自动转载机制，可以对id属性值为train1的<bean>标签的配置修改成如下所示：

```
    <bean
id="train1"class="com.cjg.spring.Train1">
```

然后在标签<beans>中，把属性defaultautowire的值设置为“byName”。

注意：当<bean>标记的属性autowire的设置只是针对当前的XML配置文件，而不是整个项目中的XML配置文件。

把上面的配置文件修改后，针对带有命名为train2属性的对象，IOC容器会自动到XML配置文件中找到id为train2的配置，然后根据该配置的class属性

实例化即创建一个对象，最后自动传入到对象的属性上，同理train3和train4也一样。

（2）如果接着（1）修改配置文件，把autowire的设置改为“byType”。运行测试文件，还是会得到相同的结果。这是为什么呢？

当autowire属性值设置为byType时，查看train1文件，可以发现其属性有train2类型，Spring IOC容器就会自动到XML配置文件中，找＜bean＞标记中class属性值为train2的配置，然后实例化该Bean。在查找的过程中，忽略了＜bean＞标记中id属性值。

注意：在开发阶段中，为了实现快速开发，可以使用该种技术，但是在实施和维护阶段却不建议使用，因为会影响代码的阅读。

（3）查看API可以得到对autowire属性值的说明，如表22.2所示。

表 22.2　autowire 属性值

模式	说　明
byName	IOC 容器会自动根据名字查找与属性名字完全一致的 Bean，并将其与属性自动装配，即根据属性名自动装配
byType	IOC 容器会自动根据类型查找与属性类型完全一致的 Bean，并将其与属性自动装配，即根据类型自动装配

22.3 Bean的作用范围

本节讲解Bean的作用域，什么是Bean的作用域呢？就是当多次从IOC容器中采用getBean方法把注入的Bean拿出来时，到底是一个实例还是多个实例。

技术要点

在Spring中是通过XML配置文件中的＜bean＞标记的属性scope来决定作用域，当该属性值为singleton时，返回相同的实例；如果为prototype时，则返回不同的实例。

返回相同实例的属性值：singleton。

返回不同实例的属性值：prototype。

实现代码

首先在包com.cjg.spring中，设计一个培训机构类Train1.java，代码如下：

```
    package com.cjg.spring;
    public class Train1 {
    }
```

接着在包com.cjg.test中，设计一个测试类Client.java，代码如下：

```
    package com.cjg.test;
    import
org.springframework.beans.factory.BeanFactory;
    import
org.springframework.context.support.ClassPathXmlAp
plicationContext;
    import com.cjg.spring.Train1;
    import junit.framework.TestCase;
    public class Client extends TestCase {
    private BeanFactory factory; //定义私有工程Bean对
象
    //重写父类方法
    protected void setUp（）throws Exception {
    factory=new
ClassPathXmlApplicationContext（"applicationContext
.xml"）;
    }
    public void testScope1（） {
    Train1 train11=（Train1）
factory.getBean（"train1"）; //获取Bean中train1值
```

```
    Train1 train12=（Train1）
factory.getBean（"train1"）; //获取Bean中train2值
    if（train111==train12）{//如果两个值相等
    System.out.println（"同一个"）;
    } else {
    System.out.println（"不是同一个"）; //如果两个值不等
    }
    }
    }
    }
```

运行该类，结果如下：

```
    2号
    cjg
    1000
    cjg
    1000
    1000
```

源程序解读

（1）如果配置文件applicationContext.xml具体内容如下：

```
    <bean
id="train1"class="com.bjsxt.spring.Train1"scope="s
ingleton"/>
    </beans>
```

运行测试文件，得到的Bean对象在内存中都是指向了同一个对象，即只初始化一次，所以他俩train11和train12是同一个对象。

运行该测试类，结果如下：

```
同一个
```

（2）如果配置文件applicationContext.xml具体内容如下：

```
    <bean
id="bean1"class="com.bjsxt.spring.Bean1"scope="pro
totype"/>
    </beans>
```

运行测试文件，每使用一次getBean方法就会创建一个Bean对象，所以train11和train12不是同一个对象。

运行该测试类，结果如下：

```
不是同一个
```

第23章　使用AOP

在Spring还没有出现前，大多数程序员都使用EJB来声明式事务。即用EJB的SessionBean来控制哪些方法拥有事务，执行到方法时自动开启事务，当方法成功执行后，会自动提交；当该方法执行失败时，会自动回滚。

Spring出现后，完全可以利用Spring的AOP来代替EJB来声明式事务。最重要的是Spring的AOP拥有声明式服务，即不用修改代码就可以拥有这种能力。EJB事务只能应用到自己的组件上，而如果作用到类等普通Java对象上是做不到。而Spring的AOP却可以做到。本章还是先从AOP基本原理来讲解。

23.1　静态代理

学习AOP最重要的是要学会AOP的一整套理论和概念，如果想学好就必须从低层实现来学习。代理的技术是AOP理论的重要部分。Spring的AOP默认采用动态代理机制，所以就必须从代理学起。

技术要点

在Java中，代理分为静态代理和动态代理两种，本小节就开始讲解静态代理。

为什么要使用静态代理。

实现静态代理。

实现代码

首先介绍代码的运行背景，培训机构在接受学生报名时，就要把学生的信息保存起来。在保存学生信

息时就要检查只有交过钱的学生的信息才能被保存起来。

在包com.cjg.spring中设计一个管理学生的接口NewStudentManager，代码如下：

```
    package com.cjg.spring;
    public interface NewStudentManager {
    public void addStudent（String Studentname, String password）; //增加学生方法
    public void deleteStudent（int id）; //删除学生方法
    public void modifyStudent（int id, String Studentname, String password）;
    //修改学生信息方法
    public String findStudentById（int id）; //查找学生方法
    }
```

接着在包com.cjg.spring中建立实现接口的实体类NewStudentManagerAdd.java，代码如下：

```
    package com.cjg.spring;
    public class NewStudentManagerAdd implements NewStudentManager {
    public void addStudent（String Studentname, String password） {//增加一个学生
```

```
    System.out.println（"……
NewStudentManagerAdd.addStudent（）……"）;
    }
    public void deleteStudent（int id）{//删除学生
    System.out.println（"……
NewStudentManagerAdd.deleteStudent（）……"）;
    }
    public String findStudentById（int id）{//查找学
生
    System.out.println（"……
NewStudentManagerAdd.findStudentById（）……"）;
    return null;
    }
    //修改学生的信息
    public void modifyStudent（int id, String
Studentname, String password）{
    System.out.println（"……
NewStudentManagerAdd.modifyStudent（）……"）;
    }
    }
```

在包com.cjg.spring中建立代理类NewStudentManagerAdd.java，代码如下：

```
    package com.cjg.spring;
    public class NewStudentManagerAddProxy
implements NewStudentManager{
    private NewStudentManager newStudentManager;
    public
NewStudentManagerAddProxy（NewStudentManager
newStudentManager）{
    this.newStudentManager=newStudentManager;
    }
```

```
    public void addStudent（String Studentname,
String password）{//增加学生
    check（）;
    this.newStudentManager.addStudent（Studentname,
password）;
    }
    public void deleteStudent（int id）{//删除学生
    check（）;
    this.newStudentManager.deleteStudent（id）;
    }
    public String findStudentById（int id）{//查找学
生
    return null;
    }
    //修改学生的信息
    public void modifyStudent（int id, String
Studentname, String password）{
    }
    private void check（）{//检验的方法
    System.out.println（"……check（）……"）;
    }
    }
```

在包com.cjg.test中建立客户端Client.java，代码如下：

```
    package com.cjg.test;
    import com.cjg.spring.NewStudentManagerAdd;
    import
com.cjg.spring.NewStudentManagerAddProxy;
    public class Client {
    public static void main（String[]args）{
```

```
    NewStudentManagerAddProxy
newStudentManagerAddProxy=
    new NewStudentManagerAddProxy（new
NewStudentManagerAdd（））；
    newStudentManagerAddProxy.addStudent（"cjg", "12
3"）；//增加一个学生
    newStudentManagerAddProxy.deleteStudent（1）；//
删除一个学生
    }
    }
```

运行该测试类Client，结果如下：

```
    ……check（）……
    ……NewStudentManagerAdd.addStudent（）……
    ……check（）……
    ……NewStudentManagerAdd.deleteStudent（）……
```

源程序解读

（1）首先设计一个管理学生的接口NewStudentManager和实现该接口的类NewStudentManagerAdd。在类NewStudentManagerAdd中分别实现了增加学生、删除学生、查找学生和修改学生信息的方法。

（2）如果没有编写代理类，就在测试类Client中直接调用类NewStudentManagerAdd，如下所示：

```
    package com.cjg.test;
    import com.cjg.spring.NewStudentManagerAdd;
    public class Client {
    public static void main（String[]args） {
    NewStudentManagerAdd newStudentManagerAdd=new
NewStudentManagerAdd（）;
    newStudentManagerAdd.addStudent（"cjg", "123"）
; //添加一个学生信息
    }
    }
```

运行结果如下：

```
    ……NewStudentManagerAdd.addStudent（）……
```

这显然与要求不符合，因为在增加学生的同时要检验该学生是否交过钱，所以只有通过检查的学生，其信息才能被保存起来。

（3）为了弥补（2）中的不足，一般把NewStudentManagerAdd.java文件修改如下：

```
    package com.cjg.spring;
    public class NewStudentManagerAdd {
    public void addStudent（String Studentname,
String password） {//增加学生信息
    check（）;
    addStudent（Studentname, password）;
    }
    public void deleteStudent（int id） {//删除学生信
息
    check（）;
    deleteStudent（id）;
    }
    public String findStudentById（int id） {//查找学
生信息
    return null;
    }
    //修改学生信息
    public void modifyStudent（int id, String
Studentname, String password） {
    }
    private void check（） {//检验方法
    System.out.println（"……check（）……"）;
    }
    }
```

在NewStudentManagerAdd.java文件中编写一个检验方法Check，然后在其他的每一个方法中都调用该方法。虽然，该方法也可以实现要求的功能，但是破坏了类的封装性。

（4）为了弥补（3）中的不足，就需要建立一个代理类NewStudentManagerAdd.java文件，该类与需要代理的类NewStudentManagerAdd有共同的接口NewStudentManager，并且需要获取NewStudentManagerAdd对象，从而实现对对象NewStudentManagerAdd的控制，就是在使用每个方法时都需要检验。

（5）在测试文件中，通过对代理类的调用，可以实现相应的功能。

注意：虽然静态代理完全可以实现要求的功能，但是检查方法Check（）分散在需要检查方法的各个方法中，如果需要检查的方法有成百上千的话，显然静态代理这种方式不可取。如何解决这种问题了，请看23.2节。

23.2　动态代理

在上一小节的应用背景中，检验学生信息与保存学生信息是完全不相关的业务，对于保存学生信息来讲，把检验学生信息加入就可以实现该功能，把其拿走就不能实现该功能，对保存学生信息不能什么修改，这就是所谓的横切性问题。在保存学生信息功能中，需要加入检验方法就是所谓的横切性关注点。

技术要点

Sping中的AOP默认采用动态代理机制，动态代理的原理就是把横切性关注点提取出来放在一个类里，在运行时根据实际情况加入到所需的方法中。

实现动态代理。

AOP中的术语。

实现代码

首先介绍代码的运行背景，培训机构在接受学生报名时，就要把学生的信息保存起来。在保存学生信息时就要检查只有交过钱的学生的信息才能被保存起来。在包com.cjg.spring中设计一个管理学生的接口NewStudentManager，代码如下：

```
    package com.cjg.spring;
    public interface NewStudentManager {
    public void addStudent（String Studentname,
String password）; //增加学生
    public void deleteStudent（int id）; //删除学生
    public void modifyStudent（int id, String
Studentname, String password）; //修改学生信息
    public String findStudentById（int id）; //查找学
生
    }
```

接着在包com.cjg.spring中建立实现接口的实体类NewStudentManagerAdd.java，代码如下：

```
    package com.cjg.spring;
    public class NewStudentManagerAdd implements
NewStudentManager {
```

```
    //增加一个学生
    public void addStudent（String Studentname,
String password） {
    System.out.println（"……
NewStudentManagerAdd.addStudent（）……"）;
    }
    public void deleteStudent（int id） {//删除学生
    System.out.println（"……
NewStudentManagerAdd.deleteStudent（）……"）;
    }
    public String findStudentById（int id） {//查找学
生
    System.out.println（"……
NewStudentManagerAdd.findStudentById（）……"）;
    return null;
    }
    //修改学生的信息
    public void modifyStudent（int id, String
Studentname, String password） {
    System.out.println（"……
NewStudentManagerAdd.modifyStudent（）……"）;
    }
    }
```

在包com.cjg.check中建立代理类Check.java，代码如下：

```
    package com.cjg.check;
    import java.lang.reflect.InvocationHandler;
    import java.lang.reflect.Method;
    import java.lang.reflect.Proxy;
    public class Check implements
InvocationHandler {
```

```
    private Object targetObject; //定义一个Object对象
    public Object newProxy（Object targetObject） {
    this.targetObject=targetObject;
    return
Proxy.newProxyInstance（targetObject.getClass（） .g
etClassLoader（）,
    targetObject.getClass（） .getInterfaces（）,
this）;
    }
    //定义判断方法
    public Object invoke（Object proxy, Method
method, Object[]args）
    throws Throwable {
    check（）;
    Object ret=null;
    try {
    ret=method.invoke（this.targetObject, args）;
    } catch（Exception e） {
    e.printStackTrace（）;
    throw new java.lang.RuntimeException（e）;
    }
    return ret;
    }
    private void check（） {//定义验证方法
    System.out.println（"……check（） ……"）;
    }
    }
```

在包com.cjg.test中建立客户端Client.java，代码如下：

```
    package com.cjg.test;
    import com.cjg.check.SecurityCheck;
```

```
    import com.cjg.spring.NewStudentManager;
    import com.cjg.spring.NewStudentManagerAdd;
    public class Client {
    public static void main（String[]args） {
    Check check=new Check（）; //创建Check对象
    NewStudentManager newStudentManager=
     （NewStudentManager）Check.newProxy（new
NewStudentManagerAdd（））;
    newStudentManager.addStudent（"cjg", "123"）; //
添加一个学生对象
    }
    }
```

运行该测试类Client，结果如下：

```
    ……checky（）……
    ……NewStudentManagerAdd.addStudent（）……
```

源程序解读

（1）首先设计一个管理学生的接口NewStudentManager和实现该接口的类NewStudentManagerAdd。在类NewStudentManagerAdd中分别实现了增加学生、删除学生、查找学生和修改学生信息的方法。

（2）Check. java文件动态创建代理，类Check首先要实现InvocationHandler接口，该接口是代理实例的调用处理程序实现的接口。查看API可以得到该接口的定义：

```
public interface InvocationHandler（） {}
```

其次要为某个类创建代理类，就必须得到该类的引用，只有这样才能动态为它创建代理类。targetObject就是对对象NewStudentManagerAdd的引用。

（3）方法newProxy，就是为传进来的目标对象动态地创建代理类实例。在该方法中，利用Proxy类的newProxyInstance方法来创建代理对象，该方法返回一个指定接口的代理类实例，查看API可以得到该方法的定义：

```
    public static Object
newProxyInstance（ClassLoader loader,
    Class＜? ＞[]interfaces,
    InvocationHandler h）
    throws IllegalArgumentException
```

该接口可以将方法调用指派到指定的调用处理程序中，参数

targetObject.getClass（）.getClassLoader（）：定义代理类的类加载器，其值为

targetObject.getClass（）.getClassLoader（）；

targetObject.getClass（）.getInterfaces（）：代理类要实现的接口，其值为

targetObject.getClass（）.getInterfaces（）；指派方法调用的调用处理程序：其值为this。最后通过关键字return返回创建的代理类对象。

（4）当调用目标对象动态的创建代理类实例的方法时，默认会先调用invoke方法，查看API可以得到该方法的定义：

```
Object invoke（Object proxy，
Method method，
Object[]args）
throws Throwable
```

该方法为实现接口InvocationHandler中的方法，作用是在代理实例proxy上使用带有参数args的处理方法method调用并返回结果。其中有3个参数：参数obj：代理实例，其值为Proxy类型对象（方法newProxy返回的对象）；参数method：代理实例上的处理方法，其值必须是method对象；参数args：方法调用的参数，其值为args。

（5）为了实现具体的功能，可以在方法invoke中具体编写，先调用检验方法Check（），接着再动态地调用目标对象的方法，这时就要用到method.invoke方法，该方法作用是返回使用参数args在obj上指派该对象所表示方法的结果，查看API可以得到该方法的定义：

```
public Object invoke（Object obj,
Object……args）
throws IllegalAccessException,
IllegalArgumentException,
InvocationTargetException
```

该方法的两个参数值分别为：目标对象targetObject和参数列表args。最后返回该方法的结果ref。

（6）在测试文件Client.java中，首先得到Check对象Check，然后通过Check的方法newProxy，得到类NewStudentManager的代理类newStudentManager，最后调用newStudentManager的addStudent方法。

下面根据上述项目来讲解AOP中的术语，这些术语间的关系如图23.1所示。从上到下方法调用一个应用背景即目标对象保存学生信息，加入了一个学生信息检查，这个检查学生信息就叫横切性关注点。把检查学生信息的方法模块化放在一个类中，这个类就叫切

面。检查学生信息方法的具体实现就叫advice，其分为before advice、after advice等，这些分类是以advice最终应用到目标对象保存学生信息中方法的前面还是后面而划分。

同时还有一个表示过滤条件术语Pointcut，例如保存学生信息中所有以add开头的方法。把advice根据自己的属性加入到目标对象保存学生信息中符合Pointcut的方法的过程叫做植入（Weave），而符合Pointcut的方法就叫连接点（Joinpoint）。由于Sping的AOP采用动态代理机制，所以在具体应用中会为目标对象保存学生信息动态生成一个目标代理（Target Object Proxy），而动态生成目标对象代理方法的过程就叫Introduction。

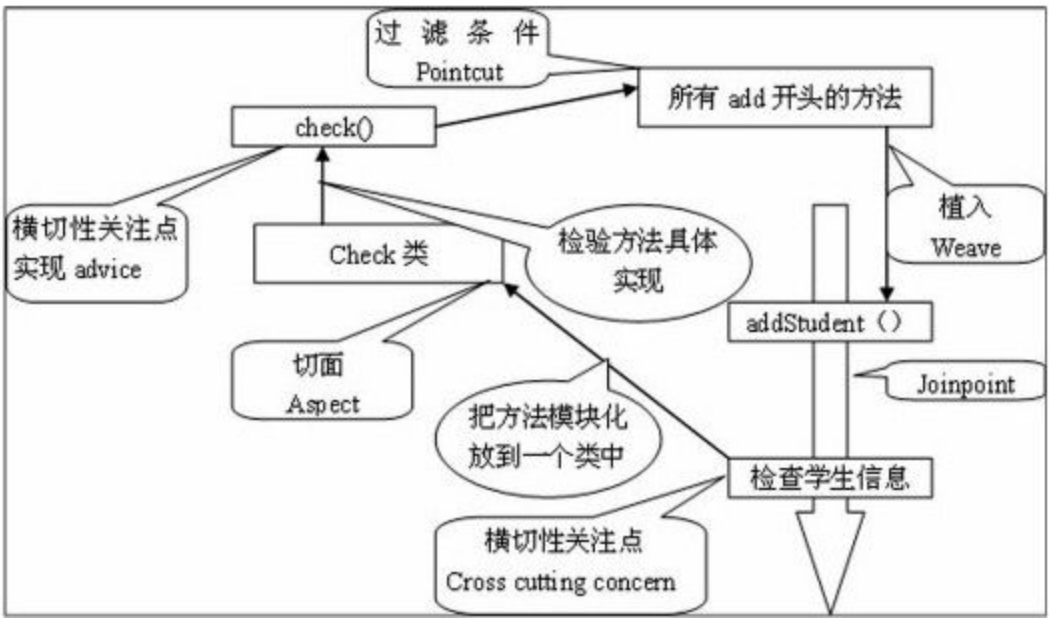

图　23.1　AOP术语

第24章　在Spring环境中实现AOP

在Spring环境中，可以通过两种方式来实现AOP，即注解方式（Annotation）和配置文件方式，本章将详细介绍这两种方式。当Spring的IOC为目标对象创建出动态代理类时，开发人员能获取动态代理类的参数吗？同时还要注意一下，在编写目标对象时是否必须继承接口？这些问题在本章中都会得到解决。

24.1　采用Annotation方式实现AOP

这一节通过Spring的AOP来实现上两节中的应用。在Spring中可以通过注解（Annotation）方式来实现AOP。在本节中就来讲解一个采用Annotation方式实现AOP的案例。通过注解的形式可以无缝链接AOP和IOC等知识。

技术要点

在使用Annotation方式时，最重要就是要学会实现注解的那些类。

实现代码

首先介绍代码的运行背景，培训机构在接受学生报名时，就要把学生的信息保存起来。在保存学生信息时就要检查只有交过钱的学生的信息才能被保存起来。

在包com.cjg.spring中设计一个管理学生的接口NewStudentManager，代码如下：

```
    package com.cjg.spring;
    public interface NewStudentManager {
    public void addStudent（String Studentname,
String password）; //增加学生
    public void deleteStudent（int id）; //删除学生
    public void modifyStudent（int id, String
Studentname, String password）; //修改学生信息
    public String findStudentById（int id）; //查找学
生
```

```
    }
```

接着在包com.cjg.spring中建立实现接口的实体类NewStudentManagerAdd.java，代码如下：

```
    package com.cjg.spring;
    public class NewStudentManagerAdd implements
NewStudentManager {
    //增加一个学生
    public void addStudent（String Studentname,
String password） {
    System.out.println（"……
NewStudentManagerAdd.addStudent（）……"）;
    }
    public void deleteStudent（int id） {//删除学生
    System.out.println（"……
NewStudentManagerAdd.deleteStudent（）……"）;
    }
    public String findStudentById（int id） {//查找学
生
    System.out.println（"……
NewStudentManagerAdd.findStudentById（）……"）;
    return null;
    }
    //修改学生的信息
    public void modifyStudent（int id, String
Studentname, String password） {
    System.out.println（"……
NewStudentManagerAdd.modifyStudent（）……"）;
    }
    }
```

在包com.cjg.check中建立代理类Check.java，代码如下：

```
package com.cjg.check;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
@Aspect
public class Check {
@Pointcut("execution(add(..)))")
private void allAddMethod() {
};
@Before("allAddMethod()")
private void check() {
System.out.println("……check()……");
}
}
```

在包com.cjg.test中建立客户端Client.java，代码如下：

```
package com.cjg.test;
import
org.springframework.beans.factory.BeanFactory;
import
org.springframework.context.support.ClassPathXmlAp
plicationContext;
```

```
    import com.cjg.check.Check;
    import com.cjg.spring.NewStudentManager;
    import com.cjg.spring.NewStudentManagerAdd;
    public class Client {
    public static void main（String[]args） {
    BeanFactory factory=new
ClassPathXmlApplicationContext（"applicationContext
.xml"）;
    NewStudentManagerAdd newStudentManagerAdd=
    （NewStudentManagerAdd）
factory.getBean（"newStudentManagerAdd"）;
    //添加一个学生信息
    newStudentManagerAdd.addStudent（"cjg", "123"）
;
    }
    }
```

运行该测试类Client，结果如下：

```
    ……checky（）……
    ……NewStudentManagerAdd.addStudent（）……
```

源程序解读

（1）该项目的配置文件applicationContext.xml的具体代码如下：

```
    ＜AOP: aspectjautoproxy/＞
    ＜bean id="Check"class="com.cjg.check.Check"/＞
```

```
    <bean
id="newStudentManagerAdd"class="com.cjg.spring.New
StudentManagerAdd"/>
    </beans>
```

（2）在Check.java文件中首先要让Spring的IOC知道那个类是切面（Aspect），可以用"@Aspect"定义该类是切面。接着用@Pointcut（）来定义过滤条件（Pointcut），在方法@Pointcut中可以编写表达式用来描述一些方法，例如：

```
    @Pointcut（"execution（add（..））||
execution（del（..））")
```

上面代码“*add（..）”中，第一个“*”表示不管是否有返回值，第二个“*”是和add合并在一起表示以add开头的方法名，“（..）”表示不管是一个参数还是两个参数或没参数，综合上面的意思，所有以add开头的方法，不管其是否有返回值参数多少，都可完成切面的添加操作。

注意：被定义为Pointcut的方法allAddMethod只是一个标识，其不能有参数和返回值。该方法的名字同时也是Pointcut的名称。还要定义横切性关注点（advice），“@Before（"allAddMethod（）"）”表示别标识为advice的方法Check（）的执行是在所有以add开头的方法的前面。

（3）对类NewStudentManagerAdd和Check在applicationContext.xml中进行配置，其中代码如下：

```
<AOP: aspectjautoproxy/>
```

表示对Annotation方式编程支持。

（4）Client. java文件跟前几章的测试文件一样，只是注意下面一句代码生成的是代理类。

```
NewStudentManagerAdd
newStudentManagerAdd=（NewStudentManagerAdd）
```

```
factory.getBean（"newStu
    dentManagerAdd"）；
```

注意：在使用Annotation方式编写支持AOP的程序时，定义为Pointcut的方法是不被执行的，它存在的目的仅仅是为了重用切入点。

24.2　采用配置文件方式实现AOP

在上一小节中，通过Annotation方式来实现AOP，不仅需要把切面等在类中标示出来，最后还得把类注入到Spring的IOC中。这样感觉有点麻烦，所以本小节将讲解如何只用配置文件来实现Spring的AOP。

技术要点

当采用配置文件的方式来实现AOP时，就要对配置所需要的标签熟悉。

＜AOP：config＞标签。

实现代码

首先介绍代码的运行背景，培训机构在接受学生报名时，就要把学生的信息保存起来。在保存学生信

息时就要检查只有交过钱的学生的信息才能被保存起来。在包com.cjg.spring中设计一个管理学生的接口NewStudentManager，代码如下：

```
    package com.cjg.spring;
    public interface NewStudentManager {
    public void addStudent（String Studentname, String password）; //增加学生
    public void deleteStudent（int id）; //删除学生
    public void modifyStudent（int id, String Studentname, String password）; //修改学生信息
    public String findStudentById（int id）; //查找学生
    }
```

接着在包com.cjg.spring中建立实现接口的实体类NewStudentManagerAdd.java，代码如下：

```
    package com.cjg.spring;
    public class NewStudentManagerAdd implements NewStudentManager {
    //增加一个学生
    public void addStudent（String Studentname, String password） {
    System.out.println（"…… NewStudentManagerAdd.addStudent（）……"）;
    }
    public void deleteStudent（int id） {//删除学生
```

```
    System.out.println（"……
NewStudentManagerAdd.deleteStudent（）……"）；
    }
    public String findStudentById（int id）｛//查找学
生
    System.out.println（"……
NewStudentManagerAdd.findStudentById（）……"）；
    return null;
    }
    //修改学生的信息
    public void modifyStudent（int id, String
Studentname, String password）｛
    System.out.println（"……
NewStudentManagerAdd.modifyStudent（）……"）；
    }
    }
```

在包com.cjg.check中建立代理类Check.java，代码如下：

```
    package com.cjg.check;
    public class Check｛//创建代理类
    private void check（）｛
    System.out.println（"……check（）……"）；
    }
    }
```

在包com.cjg.test中建立客户端Client.java，代码如下：

```
    package com.cjg.test;
    import
org.springframework.beans.factory.BeanFactory;
    import
org.springframework.context.support.ClassPathXmlAp
plicationContext;
    import com.cjg.check.Check;
    import com.cjg.spring.NewStudentManager;
    import com.cjg.spring.NewStudentManagerAdd;
    public class Client {
    public static void main（String[]args） {
    BeanFactory factory=new
ClassPathXmlApplicationContext（"applicationContext
.xml"）;
    //获取工厂Bean对象
    NewStudentManagerAdd newStudentManagerAdd=
    （NewStudentManagerAdd）
factory.getBean（"newStudentManagerAdd"）;
    //添加一个学生信息
    newStudentManagerAdd.addStudent（"cjg", "123"）
;
    }
    }
```

运行该测试类Client，结果如下：

```
    ……checky（）……
    ……NewStudentManagerAdd.addStudent（）……
```

源程序解读

（1）该项目的配置文件applicationContext.xml的具体代码如下：

```
    <bean id="Check"class="com.cjg.check.Check"/>
    <bean
id="newStudentManagerAdd"class="com.cjg.spring.New
StudentManagerAdd"/>
    <AOP: config>
    <AOP: aspect id="check"ref="Check">
    <AOP: pointcut
id="allAddMethod"expression="execution（com.cjg.spr
ing.
    NewStudentManagerAdd.add（..））"/>
    <AOP: before
method="check"pointcutref="allAddMethod"/>
    </AOP: aspect>
    </AOP: config>
    </beans>
```

（2）applicationContext. xml配置文件必须让IOC容器知道哪个类是切面等，这些都是在标签＜AOP：config＞中配置的。

＜AOP：aspect＞表示切面，＜AOP：pointcut＞表示过滤条件，其属性expression用来表示表达式，

而表达式的编写与上一节介绍的一样。＜AOP：before＞标签代表横切性关注点。

（3）其他文件跟上一节一样。

24.3　获取参数

通过Annotation方式和配置文件的方式都可以实现Spring的AOP，但是一些代理类需要传递参数。在使用这些方式编写时，如何能够获的代理类的参数，就成为学习重点。

技术要点

在Spring中可以使用JoinPoint参数来获取代理类的参数。

JoinPoint参数。

实现代码

首先介绍代码的运行背景，培训机构在接受学生报名时，就要把学生的信息保存起来。在保存学生信

息时就要检查只有交过钱的学生的信息才能被保存起来。

在包com.cjg.spring中设计一个管理学生的接口NewStudentManager，代码如下：

```
    package com.cjg.spring;
    public interface NewStudentManager {
    public void addStudent（String Studentname, String password）; //增加学生
    public void deleteStudent（int id）; //删除学生
    public void modifyStudent（int id, String Studentname, String password）; //修改学生信息
    public String findStudentById（int id）; //查找学生
    }
```

接着在包com.cjg.spring中建立实现接口的实体类NewStudentManagerAdd.java，代码如下：

```
    package com.cjg.spring;
    public class NewStudentManagerAdd implements NewStudentManager {
    //增加一个学生
    public void addStudent（String Studentname, String password） {
```

```
    System.out.println（"……
NewStudentManagerAdd.addStudent（）……"）;
    }
    public void deleteStudent（int id）{//删除学生
    System.out.println（"……
NewStudentManagerAdd.deleteStudent（）……"）;
    }
    public String findStudentById（int id）{//查找学
生
    System.out.println（"……
NewStudentManagerAdd.findStudentById（）……"）;
    return null;
    }
    //修改学生的信息
    public void modifyStudent（int id, String
Studentname, String password）{
    System.out.println（"……
NewStudentManagerAdd.modifyStudent（）……"）;
    }
    }
```

在包com.cjg.check中建立代理类Check.java，代码如下：

```
    package com.cjg.check;
    import org.aspectj.lang.JoinPoint;
    public class Check {
    private void check（JoinPoint joinPoint）{
    Object[]args=joinPoint.getArgs（）; //获取输入内容
    for（int i=0; i＜args.length; i++）{//对输入内容循
环
    System.out.println（args[i]）;
    }
```

```
    //显示获取信息
    System.out.println（joinPoint.getSignature（）.g
etName（））；
    System.out.println（"……check（）……"）；
    }
    }
```

在包com.cjg.test中建立测试类Client.java，代码如下：

```
    package com.cjg.test;
    import
org.springframework.beans.factory.BeanFactory;
    import
org.springframework.context.support.ClassPathXmlAp
plicationContext;
    import com.cjg.spring.NewStudentManager;
    public class Client {
    public static void main（String[]args） {
    BeanFactory factory=new
ClassPathXmlApplicationContext（"applicationContext
.xml"）；
    //获取工厂Bean对象
    NewStudentManager userManager=
    （NewStudentManager）
factory.getBean（"newStudentManagerAdd"）；
    //添加学生信息
    userManager.addStudent（"cjg", "123"）；
    userManager.deleteStudent（1）；
    }
    }
```

运行该测试类Client，结果如下：

```
cjg
123
addStudent
……check（）……
……NewStudentManagerAdd.addStudent（）……
……NewStudentManagerAdd.deleteStudent（）……
```

源程序解读

（1）该项目的配置文件applicationContext.xml的具体代码如下：

```
    <bean id="Check"class="com.cjg.check.Check"/>
    <bean
id="newStudentManagerAdd"class="com.cjg.spring.New
StudentManagerAdd"/>
    <AOP: config>
    <AOP: aspect id="check"ref="Check">
    <AOP: pointcut
id="allAddMethod"expression="execution（com.cjg.spr
ing.
    NewStudentManagerAdd.add（..））"/>
    <AOP: before
method="check"pointcutref="allAddMethod"/>
    </AOP: aspect>
    </AOP: config>
    </beans>
```

（2）在测试文件Client.java中，当运行到下面代码时，由于是从Spring的IOC容器中读取出，所以userManager类是一个代理类。

```
    NewStudentManager
userManager=（NewStudentManager）
factory.getBean（"newStudentManagerAdd"）;
```

如何获取代理类userManager中的参数？即当运行代码

```
    userManager.addStudent（"cjg", "123"）;
```

在Check.java文件中，可以获取传过来的两个参数的值。这时就需要在Check.java文件中的方法check中添加参数JoinPoint类型参数joinPoint。然后在方法体中，通过joinPoint的方法getArgs（）就可以获取所有参数值，只要遍历一边就可以全部打印出来。

同时通过joinPoint.getSignature（）.getName（）也可以得到方法名。

（3）在Check.java文件即AOP的Advice中添加一个JoinPoint参数，由spring自动生成代理类的参数值就会由spring自动传入，从而就可以从JoinPoint中取得参数值、方法名等。

24.4　使用CGLIB库

在Spring中实现AOP时，切面类默认情况下不用实现接口；但对于目标对象，在默认情况下必须实现接口，如果没有实现接口就必须引入CGLIB库。

技术要点

如果目标对象实现了接口，默认情况下会采用JDK的动态代理来实现AOP，这一节以前的AOP事例都是这种类型；当目标对象实现了接口，也可以强制使用CGLIB实现AOP；如果目标对象没有实现接口，就必须使用CGLIB实现AOP。

实现代码

首先介绍代码的运行背景，培训机构在接受学生报名时，就要把学生的信息保存起来。在保存学生信

息时就要检查只有交过钱的学生的信息才能被保存起来。在包com.cjg.spring中设计一个管理学生的接口NewStudentManager，代码如下：

```
    package com.cjg.spring;
    public interface NewStudentManager {
    public void addStudent（String Studentname, String password）; //增加学生
    public void deleteStudent（int id）; //删除学生
    public void modifyStudent（int id, String Studentname, String password）; //修改学生信息
    public String findStudentById（int id）; //查找学生
    }
```

接着在包com.cjg.spring中建立实现接口的实体类NewStudentManagerAdd.java，代码如下：

```
    package com.cjg.spring;
    public class NewStudentManagerAdd implements NewStudentManager {
    //增加一个学生
    public void addStudent（String Studentname, String password） {
    System.out.println（"…… NewStudentManagerAdd.addStudent（）……"）;
    }
    public void deleteStudent（int id） {//删除学生
```

```
    System.out.println（"……
NewStudentManagerAdd.deleteStudent（）……"）；
    }
    public String findStudentById（int id）｛//查找学
生
    System.out.println（"……
NewStudentManagerAdd.findStudentById（）……"）；
    return null;
    }
    //修改学生的信息
    public void modifyStudent（int id, String
Studentname, String password）｛
    System.out.println（"……
NewStudentManagerAdd.modifyStudent（）……"）；
    }
    }
```

在包com.cjg.check中建立检验类Check.java，代码如下：

```
    package com.cjg.check;
    import org.aspectj.lang.JoinPoint;
    public class Check｛
    private void check（JoinPoint joinPoint）｛
    Object[]args=joinPoint.getArgs（）；//获取输入信息
    for（int i=0；i＜args.length；i++）｛//对输入信息循
环
    System.out.println（args[i]）；
    }
    //显示获取信息
    System.out.println（joinPoint.getSignature（）.g
etName（））；
    System.out.println（"……check（）……"）；
```

```
    }
    }
```

在包com.cjg.test中建立客户端Client.java，代码如下：

```
    package com.cjg.test;
    import
org.springframework.beans.factory.BeanFactory;
    import
org.springframework.context.support.ClassPathXmlAp
plicationContext;
    import com.cjg.spring.NewStudentManager;
    public class Client {
    public static void main（String[]args） {
    BeanFactory factory=new
ClassPathXmlApplicationContext（"applicationContext
.xml"）;
    //获取工厂Bean对象
    NewStudentManager userManager=
    （NewStudentManager）
factory.getBean（"newStudentManagerAdd"）;
    //添加学生信息
    userManager.addStudent（"cjg", "123"）;
    userManager.deleteStudent（1）;
    }
    }
```

运行该测试类Client，结果如下：

```
cjg
123
addStudent
……check（）…….
……NewStudentManagerAdd.addStudent（）……
……NewStudentManagerAdd.deleteStudent（）……
```

源程序解读

（1）当为项目提供了CGLIB库后，然后修改配置文件applicationContext.xml中的代码如下：

```
    <AOP: aspectjautoproxy proxytargetclass="true"/
>
    <bean id="Check"class="com.cjg.check.Check"/>
    <bean
id="newStudentManagerAdd"class="com.cjg.spring.New
StudentManagerAdd"/>
    <AOP: config>
    <AOP: aspect id="check"ref="Check">
    <AOP: pointcut
id="allAddMethod"expression="execution（com.cjg.spr
ing.
    NewStudentManagerAdd.add（..））"/>
    <AOP: before
method="check"pointcutref="allAddMethod"/>
    </AOP: aspect>
    </AOP: config>
    </beans>
```

这时就会在测试文件中强制目标对象使用CGLIB库来实现代理类。如果在配置文件中去掉这一＜AOP：aspectjautoproxy proxytargetclass=″true″/＞句代码，在测试文件时就会使用默认的JDK动态代理来实现代理类。

（2）如果项目在没有引入CGLIB库的情况下去掉目标对象NewStudentManagerAdd的接口类NewStudentManager，然后修改测试文件Client.java中的代码如下：

```
    package com.cjg.test;
    import
org.springframework.beans.factory.BeanFactory;
    import
org.springframework.context.support.ClassPathXmlAp
plicationContext;
    import com.cjg.spring.NewStudentManager;
    public class Client {
    public static void main（String[]args） {
    BeanFactory factory=new
ClassPathXmlApplicationContext（"applicationContext
.xml"）;
    //获取工厂Bean对象
    NewStudentManagerAdd userManager=
```

```
    （NewStudentManagerAdd）
factory.getBean（"newStudentManagerAdd"）;
    //添加学生信息
    userManager.addStudent（"cjg", "123"）;
    userManager.deleteStudent（1）;
    }
    }
```

当运行该测试文件时，编译工具会报错。这是因为如果目标对象没有实现接口，必须使用CGLIB库。

第25章　Spring与Hibernate结合

对于Hibernate来说，事务是一个非常重要的概念，事务就是处理一批事情，如果成功，则都成功，如果失败，则都失败。那么，在Spring中是如何配合Hibernate来实现事务编程呢？本章介绍两种实现事务的方法。一种是使用编程方式，一种是使用声明式。

25.1　使用编程方式实现事务

为了彻底理解Spring如何实现事务，因此本小节用编程方式实现事务，使读者能有一个清晰的认识。本节就来开发一个使用编程方式实现事务的案例。通过本节和下一节进行对比来体会实现事务的不同。

技术要点

为了讲清楚Spring对事务的实现，本小节采用编程式方式来实现事务。

getCurrentSession（）与openSession（）的区别。

对getCurrentSession（）方法配置。

实现代码

首先介绍代码的运行背景，培训机构在接受学生报名时，就要把学生的信息保存起来。但在保存学生信息时，需要检查只有交过钱的学生的信息才能被保存起来。在包com.cjg.student.model中设计一个学生报名情况信息类Inform.java，代码如下：

```
package com.cjg.student.model;
import java.util.Date;
public class Inform {
private String id; //招收学生的id号
private String type; //招收学生的类型
private String detail; //招收学生的健康情况
```

```
    private Date time; //招收学生的时间
    public int getId（） {//定义id号的get方法
    return id;
    }
    public void setId（int id） {//定义id号的set方法
    this.id=id;
    }
    public String getType（） {//定义类型号的get方法
    return type;
    }
    public void setType（String type） {//定义类型号的
set方法
    this.type=type;
    }
    public String getDetail（） {//定义健康情况的get方
法
    return detail;
    }
    public void setDetail（String detail） {//定义健
康情况的set方法
    this.detail=detail;
    }
    public Date getTime（） {//定义时间的get方法
    return time;
    }
    public void setTime（Date time） {//定义时间的set
方法
    this.time=time;
    }
    }
```

编写类Inform中的映射文件Inform.hbm.xml。

```
    <? xml version="1.0"? >
```

```
    <Hibernatemapping
package="com.cjg.student.model">
    <class name="Inform"table="t_inform">
    <id name="id">
    <generator class="native"/>
    </id>
    <property name="type"/>
    <property name="detail"/>
    <property name="time"/>
    </class>
    </Hibernatemapping>
```

接着在包com.cjg.student.model中设计一个学生资料的类Student，代码如下：

```
    package com.cjg.student.model;
    public class Student {
    private int id; //定义id字段
    private String name; //定义名称字段
    public int getId() {//定义id字段的get方法
    return id;
    }
    public void setId(int id) {//定义id字段的set方法
    this.id=id;
    }
    public String getName() {//定义名称字段的get方法
    return name;
    }
    public void setName(String name) {//定义名称字段
的set方法
    this.name=name;
    }
    }
```

编写类Student中的映射文件Student.hbm.xml。

```
    <? xml version="1.0"? >
    <Hibernatemapping
package="com.cjg.student.model">
    <class name="Student"table="t__student">
    <id name="id">
    <generator class="native"/>
    </id>
    <property name="name"/>
    </class>
    </Hibernatemapping>
```

在包com.cjg.student.manager中建立代理类InformManager.java，代码如下：

```
    package com.cjg.student.manager;
    import com.cjg.student.model.Inform;
    public interface InformManager {
    public void addInform（Inform log）;
    }
```

在包com.cjg.student.util中建立Session封装类ExportDB.java，代码如下：

```
package com.cjg.student.util;
import org.Hibernate.cfg.Configuration;
import
org.Hibernate.tool.hbm2ddl.SchemaExport;
public class ExportDB {
public static void main（String[]args） {
//读取Hibernate.cfg.xml文件
Configuration cfg=new
Configuration（） .configure（）;
SchemaExport export=new SchemaExport（cfg）; //
获取SchemaExport对象
export.create（true, true）;
}
}
```

在包com.cjg.student.util中建立工具类HibernateUtils.java，代码如下：

```
package com.cjg.student.util;
import org.Hibernate.Session;
import org.Hibernate.SessionFactory;
import org.Hibernate.cfg.Configuration;
public class HibernateUtils {
private static SessionFactory factory; //定义工厂
对象
static {
try {
Configuration cfg=new
Configuration（） .configure（）;
factory=cfg.buildSessionFactory（）;
} catch（Exception e） {
e.printStackTrace（）;
}
```

```
    }
    public static SessionFactory
getSessionFactory（） {
    return factory;
    }
    //建立会话
    public static Session getSession（） {
    return factory.openSession（）;
    }
    //关闭会话
    public static void closeSession（Session
session） {
    if（session! =null） {
    if（session.isOpen（）） {
    session.close（）;
    }
    }
    }
    }
```

在包com.cjg.student.manager中建立管理学生报名情况接口InformManager.java，代码如下：

```
    package com.cjg.student.manager;
    import com.cjg.student.model.Inform;
    public interface InformManager {
    public void addInform（Inform inform）;
    }
```

在包com.cjg.student.manager中建立管理学生报名情况类InformManagerAdd.java，代码如下：

```
    package com.cjg.student.manager;
    import com.cjg.student.model.Inform;
    import com.cjg.student.util.HibernateUtils;
    public class InformManagerAdd implements
InformManager {
    public void addInform（Inform inform） {
    //从当前线程中获取session，并把对象inform保存到对象
session中
    HibernateUtils.getSessionFactory（）.getCurrentS
ession（）.save（inform）;
    }
    }
```

在包com.cjg.student.manager中建立管理学生资料的接口StudentManager.java，代码如下：

```
    package com.cjg.student.manager;
    import com.cjg.student.model.Student;
    public interface StudentManager {
    public void addStudent（Student student）;
    }
```

在包com.cjg.student.manager中建立管理学生资料的类StudentManagerAdd.java，代码如下：

```
    package com.cjg.student.manager;
    import java.util.Date;
    import org.Hibernate.Session;
    import com.cjg.student.model.Inform;
    import com.cjg.student.model.Student;
    import com.cjg.student.util.HibernateUtils;
    public class StudentManagerAdd implements
StudentManager {
    public void addStudent（Student student） {
    Session session=null;
    try {
    //通过工具类HibernateUtils的方法
getSessionFactory（）获取session
    session=HibernateUtils.getSessionFactory（）.get
CurrentSession（）;
    session.beginTransaction（）; //手动启动事务
    session.save（student）; //保存用户到当前的session
    Inform inform=new Inform（）; //建立一个学生报名情
况信息对象
    inform.setType（"新报名"）;
    inform.setDetail（"身体健康"）;
inform.setTime（new Date（））;
    //把对象inform通过管理对象logManager保存起来
    InformManager logManager=new
InformManagerAdd（）;
    logManager.addInform（inform）;
    session.getTransaction（）.commit（）; //手动提交
事务
    } catch（Exception e） {
    e.printStackTrace（）;
```

```
    session.getTransaction（）.rollback（）; //手动启动事务回滚
    }
    }
    }
```

在包com.cjg.student.client中建立测试类Client.java，代码如下：

```
    package com.cjg.student.client;
    import com.cjg.student.manager.StudentManager;
    import com.cjg.student.manager.StudentManagerAdd;
    import com.cjg.student.model.Student;
    public class Client {
    public static void main（String[]args） {
    //生成学生资料类
    Student studnet=new Student（）;
    studnet.setName（"cjg"）;
    //通过对象studentManager把学生资料对象student保存起来
    StudentManager studentManager=new StudentManagerAdd（）;
    studentManager.addStudent（studnet）;
    }
    }
```

运行ExportDB.java文件，结果如下：

```
    drop table if exists t__inform
```

```
    04: 22: 55, 062 WARN JDBCExceptionReporter:
48SQL Warning: 1051, SQLState: 42S02
    04: 22: 55, 062 WARN JDBCExceptionReporter:
49Unknown table't__inform'
    drop table if exists t__student
    04: 22: 55, 062 WARN JDBCExceptionReporter:
48SQL Warning: 1051, SQLState: 42S02
    04: 22: 55, 062 WARN JDBCExceptionReporter:
49Unknown table't__inform'
    04: 22: 55, 062 WARN JDBCExceptionReporter:
48SQL Warning: 1051, SQLState: 42S02
    04: 22: 55, 062 WARN JDBCExceptionReporter:
49Unknown table't__student'
    create table t__inform（id integer not null auto
__increment, type varchar（255）, detail
varchar（255）, time da
    tetime, primary key（id））
    04: 22: 55, 093 WARN JDBCExceptionReporter:
48SQL Warning: 1051, SQLState: 42S02
    04: 22: 55, 093 WARN JDBCExceptionReporter:
49Unknown table't__inform'
    04: 22: 55, 093 WARN JDBCExceptionReporter:
48SQL Warning: 1051, SQLState: 42S02
    04: 22: 55, 093 WARN JDBCExceptionReporter:
49Unknown table't__student'
    create table t__student（id integer not null
auto__increment, name varchar（255）, primary
key（id））
    04: 22: 55, 109 WARN JDBCExceptionReporter:
48SQL Warning: 1051, SQLState: 42S02
    04: 22: 55, 109 WARN JDBCExceptionReporter:
49Unknown table't__inform'
    04: 22: 55, 109 WARN JDBCExceptionReporter:
48SQL Warning: 1051, SQLState: 42S02
    04: 22: 55, 109 WARN JDBCExceptionReporter:
49Unknown table't__student'
```

运行Client.java文件，结果如下：

```
    Hibernate: insert into t__student（name）
values（? ）
    Hibernate: insert into t__inform（type, detail,
time）values（? , ? , ? ）
```

（1）在编写测试类之前，还有一个重要的文件（Hibernate.cfg.xml）要配置，具体内容如下：

```
    <Hibernateconfiguration>
    <sessionfactory>
    <property name="Hibernate.connection.url">
    jdbc: mysql: //localhost/Hibernate__1</property
>
    <property name="Hibernate.connection.driver__
class">
    com.mysql.jdbc.Driver</property>
    <! 定义用户名>
    <property
name="Hibernate.connection.username">root
</property>
    <! 定义密码>
    <property
name="Hibernate.connection.password">123
</property>
    <property name="Hibernate.dialect">
org.Hibernate.dialect.MySQLDialect</property>
    <property name="Hibernate.show__sql">true
</property>
```

```
    <property name="Hibernate.current__session__
context__class">thread</property>
    <mapping
resource="com/cjg/student/model/Student.hbm.xml"/>
    <mapping
resource="com/cjg/student/model/Inform.hbm.xml"/>
    </sessionfactory>
    </Hibernateconfiguration>
```

（2）从文件Inform.java中到文件HibernateUtils.java的建立，因为与事务没什么关系，只是创建了运行的背景，所以就不具体解释。

（3）在文件类StudentManagerAd.java中，如果不使用getCurrentSession（）方法得到session，则文件如下：

```
    public class StudentManagerAdd implements
StudentManager {
    public void addStudent（Student student） {
    Session session=null;
    try {
    session=HibernateUtils.getSession（）; //获取
session
    session.beginTransaction（）; //手动启动事务
    session.getTransaction（）.commit（）; //手动提交
事务
    } catch（Exception e） {
```

```
    e.printStackTrace（）;
    session.getTransaction（）.rollback（）; //手动启动事务回滚动
    } finally {
    HibernateUtils.closeSession（session）; //手动关闭事务
    }
    }
    }
```

从上面的代码可以看出当采用getCurrentSession（）创建的session在commit或rollback时，会自动关闭，而采用openSession（）创建的session必须手动关闭。这只是形式上的不同，其功能上则是采用getCurrentSession（）创建的session会绑定到当前线程中，而采用openSession（）创建的session则不会绑定到当前线程中。

上面运用程序的背景是在保存学生资料的同时必须保存学生报名信息，即必须把对象inform和student

保存到同一个session。所以在管理学生报名资料的类InformManagerAdd中，编写如下代码：

```
    public void addInform（Inform inform） {
    //从当前线程中获取session，并把对象inform保存到对象session中
    HibernateUtils.getSessionFactory（） .getCurrentSession（） .save（inform）；
    }
```

以保证在文件StudentManagerAdd.java中，对象student和inform保存到同一个session中。

（4）如果在StudentManagerAdd.java中不使用getCurrentSession（）方法，那么Hibernate.cfg.xml配置文件必须写成如下：

```
    ＜Hibernateconfiguration＞
    ＜sessionfactory＞
    ……
    ＜property name="Hibernate.current__session__context__class"＞jta＜/property＞
    ……
    ＜/sessionfactory＞
    ＜/Hibernateconfiguration＞
```

（5）最后，在测试文件Client.java中，首先创建一个学生资料类student，接着再用管理类studentManager把其保存起来。在运行该程序时，首先在MySql中创建一个数据库Hibernate__1，接着运行文件ExportDB.java创建表结构，最后运行Client.java文件把数据保存到数据库中。

25.2　实现声明式事务

上一小节通过手动编程方式实现了一个事务，那么本小节把代码修改一下，通过Spring来实现事务。在Spring中主要是通过AOP来实现事务，从而实现声明式事务。

技术要点

在Spring中的AOP中存在一种声明式编程方式，所以Spring中的事务不用创建只需要声明一下就可以，即所谓的声明式事物。

声明式事务的编写。

了解事物的传播特性。

实现代码

首先介绍代码的运行背景，培训机构在接受学生报名时，就要把学生的信息保存起来。在保存学生信息时，需要检查只有交过钱的学生的信息才能被保存起来。首先在包com.cjg.student.model中设计一个学生报名情况信息类Inform.java，代码如下：

```
package com.cjg.student.model;
import java.util.Date;
public class Inform {
private String id; //招收学生的id号
private String type; //招收学生的类型
private String detail; //招收学生的健康情况
private Date time; //招收学生的时间
public int getId() {//定义id号的get方法
return id;
}
public void setId(int id) {//定义id号的set方法
this.id=id;
}
public String getType() {//定义类型号的get方法
return type;
}
public void setType(String type) {//定义类型号的set方法
this.type=type;
}
public String getDetail() {//定义健康情况的get方法
return detail;
}
public void setDetail(String detail) {//定义健康情况的set方法
```

```
    this.detail=detail;
    }
    public Date getTime（）｛//定义时间的get方法
    return time;
    }
    public void setTime（Date time）｛//定义时间的set方法
    this.time=time;
    }
    }
```

编写类Inform中的映射文件Inform.hbm.xml。

```
    ＜Hibernatemapping package="com.cjg.student.model"＞
    ＜class name="Inform"table="t__inform"＞
    ＜id name="id"＞
    ＜generator class="native"/＞
    ＜/id＞
    ＜property name="type"/＞
    ＜property name="detail"/＞
    ＜property name="time"/＞
    ＜/class＞
    ＜/Hibernatemapping＞
```

接着在包com.cjg.student.model中设计一个学生资料的类Student，代码如下：

```
    package com.cjg.student.model;
    public class Student｛
```

```
    private int id; //定义id字段
    private String name; //定义名称字段
    public int getId（） {//定义id字段的get方法
    return id;
    }
    public void setId（int id） {//定义id字段的set方法
    this.id=id;
    }
    public String getName（） {//定义名称字段的get方法
    return name;
    }
    public void setName（String name） {//定义名称字段
的set方法
    this.name=name;
    }
    }
```

编写类Student中的映射文件Student.hbm.xml。

```
    <? xml version="1.0"? >
    <Hibernatemapping
package="com.cjg.student.model">
    <class name="Student"table="t__student">
    <id name="id">
    <generator class="native"/>
    </id>
    <property name="name"/>
    </class>
    </Hibernatemapping>
```

在包com.cjg.student.manager中建立管理学生报名情况接口InformManager.java，代码如下：

```
package com.cjg.student.manager;
import com.cjg.student.model.Inform;
public interface InformManager {
public void addInform（Inform inform）;
}
```

在包com.cjg.student.manager中建立管理学生报名情况类InformManagerAdd.java，代码如下：

```
package com.cjg.student.manager;
import
org.springframework.orm.Hibernate3.support.HibernateDaoSupport;
import com.cjg.student.model.Inform;
import com.cjg.student.util.HibernateUtils;
public class InformManagerAdd extends
HibernateDaoSupport implements InformManager {
public void addInform（Inform imform） {
this.getHibernateTemplate（）.save（imform）;
}
}
```

在包com.cjg.student.manager中建立管理学生资料的接口StudentManager.java，代码如下：

```
    package com.cjg.student.manager;
    import com.cjg.student.model.Student;
    public interface StudentManager {
    public void addStudent（Student student）throws
Exception;
    }
```

在包com.cjg.student.manager中建立管理学生资料的类StudentManagerAdd.java，代码如下：

```
    package com.cjg.student.manager;
    import org.Hibernate.Session;
    import
org.springframework.orm.Hibernate3.support.Hibernat
eDaoSupport;
    import com.cjg.student.model.Inform;
    import com.cjg.student.model.Student;
    import com.cjg.student.util.HibernateUtils;
    public class StudentManagerAdd extends
HibernateDaoSupport implements
    StudentManager {
    //实现对informManagerAdd的注入
    private InformManagerAdd informManagerAdd;
    public void addStudent（Student student）throws
Exception {
    //把学生资料保存到Seeion中
    this.getHibernateTemplate（）.save（student）;
    Inform inform=new Inform（）;
    inform.setType（"新报名学生"）; //保存学生类型
    inform.setDetail（"身体健康"）; //保存学生健康状况
    inform.setTime（new Date（））; //保存时间
    informManagerAdd.addInform（inform）;
    throw new Exception（）;
```

```
    }
    public void
setInformManagerAdd（InformManagerAdd
informManagerAdd） {
    this.informManagerAdd=informManagerAdd;
    }
    }
    import java.util.Date;
```

在包com.cjg.student.client中建立测试类Client.java，代码如下：

```
    package com.cjg.student.client;
    import
org.springframework.beans.factory.BeanFactory;
    import
org.springframework.context.support.ClassPathXmlApp
licationContext;
    import com.cjg.student.manager.StudentManager;
    import com.cjg.student.model.Student;
    public class Client {
    public static void main（String[]args） {
    Student student=new Student（）; //创建一个学生对象
    student.setName（"cjg"）; //设置学生名称
    //创建工厂Bean对象
    BeanFactory factory=
    new
ClassPathXmlApplicationContext（"applicationContext
.xml"）;
    （StudentManager）
factory.getBean（"studentManager"）;
    StudentManager studentManager=
    try {
```

```
    studentManager.addStudent（student）;
    } catch（Exception e） {
    e.printStackTrace（）;
    }
    }
    }
```

运行ExportDB.java文件，结果如下：

```
    drop table if exists t__inform
    06: 57: 05, 546  WARN JDBCExceptionReporter:
48SQL Warning: 1051, SQLState: 42S02
    06: 57: 05, 562  WARN JDBCExceptionReporter:
49Unknown table't__inform'
    drop table if exists t__student
    06: 57: 05, 562  WARN JDBCExceptionReporter:
48SQL Warning: 1051, SQLState: 42S02
    06: 57: 05, 562  WARN JDBCExceptionReporter:
49Unknown table't__inform'
    06: 57: 05, 562  WARN JDBCExceptionReporter:
48SQL Warning: 1051, SQLState: 42S02
    06: 57: 05, 562  WARN JDBCExceptionReporter:
49Unknown table't__student'
    create table t__inform（id integer not null auto
__increment, type varchar（255）, detail
varchar（255）, time da
    tetime, primary key（id））
    06: 57: 05, 671  WARN JDBCExceptionReporter:
48SQL Warning: 1051, SQLState: 42S02
    06: 57: 05, 671  WARN JDBCExceptionReporter:
49Unknown table't__inform'
    06: 57: 05, 671  WARN JDBCExceptionReporter:
48SQL Warning: 1051, SQLState: 42S02
    06: 57: 05, 671  WARN JDBCExceptionReporter:
49Unknown table't__student'
```

```
    create table t__student（id integer not null
auto__increment, name varchar（255）, primary
key（id））
    06: 57: 05, 718  WARN JDBCExceptionReporter:
48SQL Warning: 1051, SQLState: 42S02
    06: 57: 05, 718  WARN JDBCExceptionReporter:
49Unknown table't__inform'
    06: 57: 05, 718  WARN JDBCExceptionReporter:
48SQL Warning: 1051, SQLState: 42S02
    06: 57: 05, 718  WARN JDBCExceptionReporter:
49Unknown table't__student'
```

运行Client.java文件，结果如下：

```
    Hibernate: insert into t__student（name）
values（？）
    Hibernate: insert into t__inform（type, detail,
time）
```

（1）在编写测试类之前，还要配置Sping中的配置文件applicationContextcommon.xml，其代码如下：

```
    <！配置sessionFactory>
    <bean
id="sessionFactory"class="org.springframework.orm.H
ibernate3.LocalSess
    ionFactoryBean">
    <property name="configLocation">
```

```
    <value>classpath: Hibernate.cfg.xml</value>
    </property>
    </bean>
    <! 配置事务管理器>
    <bean
id="transactionManager"class="org.springframework.o
rm.Hibernate3.Hiber
    nateTransactionManager">
    <property name="sessionFactory">
    <ref bean="sessionFactory"/>
    </property>
    </bean>
    <! 配置事务的传播特性>
    <tx: advice
id="txAdvice"transactionmanager="transactionManager
">
    <tx: attributes>
    <tx: method name="add"propagation="REQUIRED"/>
    <tx: method name="del"propagation="REQUIRED"/>
    <tx: method
name="modify"propagation="REQUIRED"/>
    <tx: method name=""readonly="true"/>
    </tx: attributes>
    </tx: advice>
    <! 哪些类的哪些方法参与事务>
    <aop: config>
    <aop: pointcut
id="allManagerMethod"expression="execution（com.cjg
.student.
    manager..（..））"/>
    <aop: advisor
pointcutref="allManagerMethod"adviceref="txAdvice"/
>
    </aop: config>
    </beans>
```

在Hibernate中配置文件Hibernate.cfg.xml是一个很重要的文件，根据该配置文件可以创建sessionFactory对象，所以首先要把其注入到IOC中，代码如下：

```
    <bean
id="sessionFactory"class="org.springframework.orm.H
ibernate3.LocalSess
    ionFactoryBean">
    <property name="configLocation">
    <value>classpath: Hibernate.cfg.xml</value>
    </property>
    </bean>
```

上述代码是把Hibernate.cfg.xml文件注入到Spring类LocalSessionFactoryBean的属性configLocation中，查看API，可以得到如下定义：

```
    public class LocalSessionFactoryBean extends
AbstractSessionFactoryBean {
    ……
    private Resource[]configLocations;
    ……
    public void setConfigLocation（Resource
configLocation） {
```

```
    this.configLocations=new Resource[]
 {configLocation} ;
    }
    public void
setConfigLocations（Resource[]configLocations） {
    this.configLocations=configLocations;
    }
    ……
    }
```

通过得到sessionFactory就可以得到事务。那么如何管理事物呢？这时就必须注入一个事物管理器：

```
    <bean
id="transactionManager"class="org.springframework.o
rm.Hibernate3.Hiber
    nateTransactionManager">
    <property name="sessionFactory">
    <ref bean="sessionFactory"/><！注入事物管理器>
    </property>
    </bean>
```

上述代码是通过Spring类HibernateTransactionManager中的类来实现的。查看API，可以得到如下定义：

```
    public class HibernateTransactionManager extends
AbstractPlatformTransactionManager
```

```
    implements BeanFactoryAware, InitializingBean {
    ……
    private SessionFactory sessionFactory;
    ……
    public void setSessionFactory（SessionFactory
sessionFactory） {
    this.sessionFactory=sessionFactory;
    }
    public SessionFactory getSessionFactory（） {
    return sessionFactory;
    }
    ……
    }
```

接着配置事务的传播特性，事务的传播特性决定了事务是否创建。其具体的值如表25.1所示。

表 25.1　事务的传播性

属　性　值	T1（transaction）	T2（transaction）	意　　义
Required	无 T1	T2 T2	如果存在一个事务，则支持当前事务。如果没有事务则开启
RequiredNew	无 T1	T2 T2	总是开启一个新的事务。如果一个事务已经存在，则将这个存在的事务挂起
Support	无 T1	无 T1	如果存在一个事务，则支持当前事务。如果没有事务，则非事务的执行
Mandatory	无 T1	抛出异常 T1	如果已经存在一个事务，则支持当前事务。如果没有一个活动的事务，则抛出异常
NoSupport	无 T1	无 无	总是非事务地执行，并挂起任何存在的事务
Never	无 T1	无 抛出异常	总是非事务地执行，如果存在一个活动事务，则抛出异常

其具体配置如下所示：

```
    <tx: advice
id="txAdvice"transactionmanager="transactionManager
">
    <tx: attributes>
    <tx: method name="add"propagation="REQUIRED"/>
    <tx: method name=""readonly="true"/>
    </tx: attributes>
    </tx: advice>
```

上述代码标签<tx：advice>有一个属性transactionmanager用来表示事物管理器，所以其值为transactionManager。接着在其的子标签<tx：method>中配置各种方法的传播特性，即以add开头的方法，其事务属性为REQUIRED；剩下的方法，把其属性设置为只读，属性设置为只读属性后会提高性能，因为更新这个事务后参数就不再做检查了。最后配置参与事务的方法，具体配置如下：

```
    <aop: config>
    <aop: pointcut
id="allManagerMethod"expression="execution（com.cjg
.student.
    manager..（..））"/>
    <aop: advisor
pointcutref="allManagerMethod"adviceref="txAdvice"/
>
```

```
    </aop: config>
```

上述代码标签<aop: config>下的子标签<aop: pointcut>的属性expression配置包com.cjg.student.manager下的所有类所有方法来参与事务。

（2）在文件StudentManagerAdd.java中，必须继承HibernateDaoSupport类，接着通过该类的方法getHibernateTemplate（）得到Session，然后通过方法save（）把对象student保存到Session中。对于对象InformManagerAdd，可以实现注入IOC中。

（3）如何对对象StudentManagerAdd和InformManagerAdd实现注入呢？其代码如下：

```
    <bean
id="studentManager"class="com.cjg.student.manager.S
tudentManagerAdd">
    <! 注入StudentManagerAdd对象>
    <property
name="sessionFactory"ref="sessionFactory"/>
```

```
    <property
name="informManager"ref="informManager"/>
    </bean>
    <! 注入InformManagerAdd对象>
    <bean
id="informManager"class="com.cjg.student.manager.In
formManagerAdd">
    <property
name="sessionFactory"ref="sessionFactory"/>
    </bean>
```

为了能够得到Session，所以必须有一个SessionFactory属性，可是StudentManagerAdd.java文件中根本没有写sessionFactory属性。不要忘记该类还继承了类HibernateDaoSupport，查看API可以得到如下定义：

```
    public abstract class HibernateDaoSupport
extends DaoSupport {
    public final void
setSessionFactory（SessionFactory sessionFactory） {
    ……
    }
    public final SessionFactory
getSessionFactory（） {
    ……
    ）;
    }
```

（4）除了步骤（3）的实现外，还要配置Hibernate.cfg.xml文件，其具体代码如下：

```
    <Hibernateconfiguration>
    <sessionfactory>
    <property name="Hibernate.connection.url">
    jdbc: mysql: //localhost/spring__Hibernate__2
    </property>
    <property name="Hibernate.connection.driver__
class">
    com.mysql.jdbc.Driver
    </property>
    <!—设置用户名>
    <property
name="Hibernate.connection.username">root
</property>
    <!—设置密码>
    <property
name="Hibernate.connection.password">123</property
>
    <property name="Hibernate.dialect">
    org.Hibernate.dialect.MySQLDialect
    </property>
    <property name="Hibernate.show__sql">true
</property>
    <property name="Hibernate.current__session__
context__class">
    thread
    </property>
    <mapping
resource="com/cjg/student/model/Inform.hbm.xml"/>
    <mapping
resource="com/cjg/student/model/Student.hbm.xml"/>
    </sessionfactory>
```

```
</Hibernateconfiguration>
```

第26章　Spring与Struts结合

首先回忆一下Struts的运行过程，浏览器会发出请求，这些请求会被Struts的前端控制器（ActionServlet）拦截，ActionServlet的作用就是截取所有请求，然后根据strutsconfig.xml配置文件分发到相应的业务层控制器（Action）上。Action为业务层控制器，通常每个动作对应一个Action，其会处理从model模型中传过来的封装的信息，然后把处理完的信息交给ActionServlet。由ActionServlet根据处理后的信息转向相应的Web组件上。最后Web组件会生成动态页面发送给浏览器。其流程图如图26.1所示。为了实现Spring与Struts结合，应该在Action上实现两者的结合。

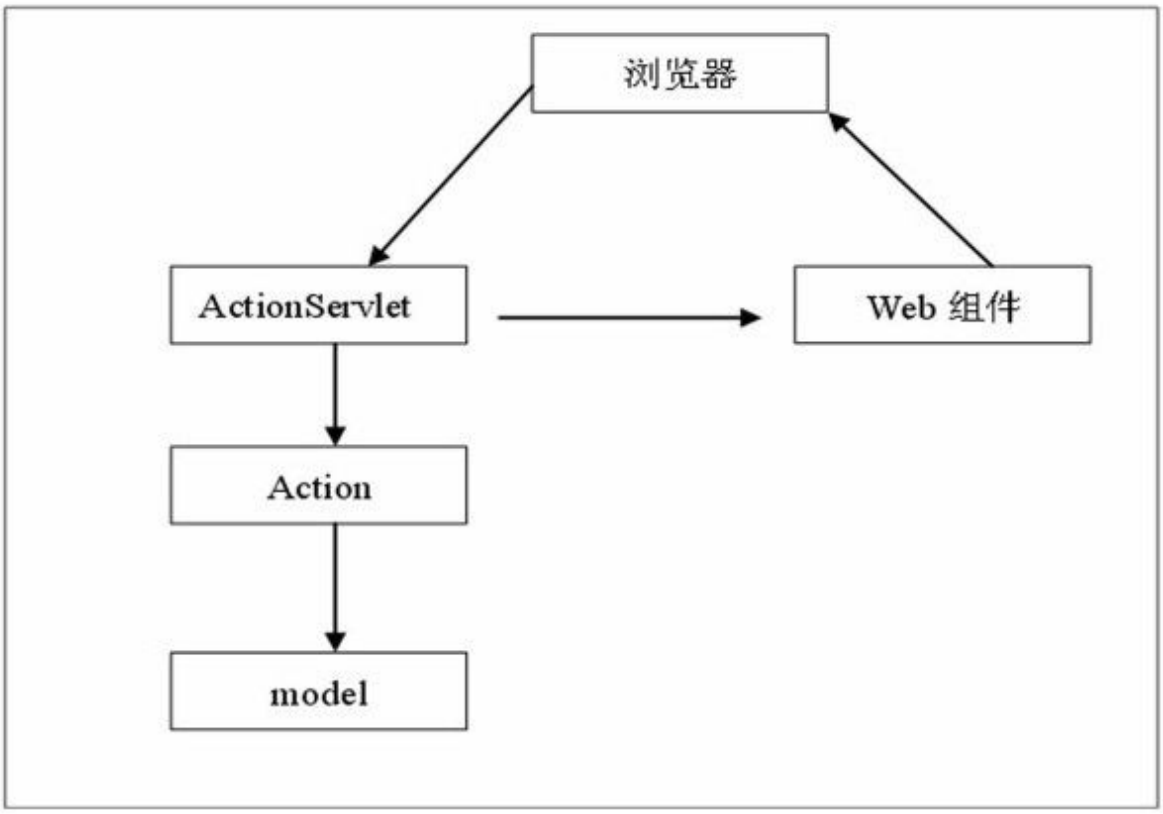

图　26.1　Struts流程

26.1　依赖查找方式实现Spring与Struts结合

为什么叫依赖查找方式，因为在Struts的Action中，是通过依赖查找方式取得BeanFactory。只要获取BeanFactory对象，就可以从IOC容器中获取业务对象

然后调用业务方法，从而实现Spring和Struts的结合。

技术要点

如何实现与Spring结合呢？在Spring中，最重要是要得到BeanFactor，只有得到该对象才能从IOC容器中获取业务对象，并调用业务方法。所以在Action中，实现BeanFactor类是一个不错得选择。

在Action中获取BeanFactor对象。

业务逻辑对象的实现。

实现代码

首先介绍代码的运行背景，培训机构在接受学生报名时，就要把学生的信息保存起来。在保存学生信息时，需要检查只有交过钱的学生的信息才能被保存起来。建立首页index.jsp，代码分别如下：

```
    <%@page language="java"contentType="text/html;
charset=GB18030"
    pageEncoding="GB18030"%>
    <html>
    <head>
    <meta
httpequiv="ContentType"content="text/html;
charset=GB18030">
    <title>spring和struts的集成</title>
    </head>
    <body><a href="logininput.do">登录</a>
    </body>
    </html>
```

登录页面login.jsp，代码如下：

```
    <%@page language="java"contentType="text/html;
charset=GB18030"
    pageEncoding="GB18030"%>
    <html>
    <head>
    <meta
httpequiv="ContentType"content="text/html;
charset=GB18030">
    <title>用户登录</title>
    </head>
    <body>
    <form action="login.do"method="post">
    用户：<input type="text"name="username"><br>
    密码：<input type="password"name="password"><br
>
    <input type="submit"value="登录">
    </form>
    </body>
```

```
    </html>
```

当登录页面login.jsp发出请求后，其请求的信息会被封装到Form中，然后由ActionServlet传给相应的Action，接着在包com.cjg.user.forms中建立LoginActionForm.java实现对信息的封装。

```
    package com.cjg.user.forms;
    import org.apache.struts.action.ActionForm;
    public class LoginActionForm extends
ActionForm {
    private String username; //定义一个用户名字段
    private String password; //定义密码字段
    public String getUsername() {//定义用户名的get方
法
    return username;
    }
    public void setUsername(String username) {//定
义用户名的set方法
    this.username=username;
    }
    public String getPassword() {//定义密码的get方法
    return password;
    }
    public void setPassword(String password) {//定
义密码的set方法
    this.password=password;
    }
    }
```

编写业务层，对LoginActionForm中封装的信息实现相应的功能，在包com.cjg.user.manager中建立接口UserManager，其代码如下：

```
    package com.cjg.user.manager;
    //定义一个被实现的接口
    public interface UserManager {
    //定义一个登录方法
    public void login（String username, String
password）;
    }
```

在包com.cjg.user.manager中建立类UserManagerAdd，其代码如下：

```
    package com.cjg.user.manager;
    //定义类实现上面的接口
    public class UserManagerAdd implements
UserManager {
    public void login（String username, String
password） {//实现接口中的登录方法
    System.out.println（"UserManagerImpl.login（）
username="+username）;
    }
    }
```

在包com.cjg.user.actions中建立类LoginAction.java，实现对请求信息操作，代码如下：

```
    package com.cjg.user.actions;
    import javax.servlet.http.HttpServletRequest;
    import javax.servlet.http.HttpServletResponse;
    import org.apache.struts.action.Action;
    import org.apache.struts.action.ActionForm;
    import org.apache.struts.action.ActionForward;
    import org.apache.struts.action.ActionMapping;
    import
org.springframework.beans.factory.BeanFactory;
    import
org.springframework.context.ApplicationContext;
    import
org.springframework.context.support.ClassPathXmlApp
licationContext;
    import
org.springframework.web.context.support.WebApplicat
ionContextUtils;
    import com.cjg.user.forms.LoginActionForm;
    import com.cjg.user.manager.UserManager;
    import com.cjg.user.manager.UserManagerAdd;
    public class LoginAction extends Action {
    //重写父类方法
    public ActionForward execute（ActionMapping
mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse
response）
    throws Exception {
    LoginActionForm laf=（LoginActionForm）form;
    //获取工厂Bean对象
```

```
    BeanFactory factory=WebApplicationContextUtils
    .getRequiredWebApplicationContext（request.getSe
ssion（）
    .getServletContext（））；
    //获取Bean对象
    UserManager userManager=（UserManager）
factory.getBean（"userManager"）；
    userManager.login（laf.getUsername（），
laf.getPassword（））；
    return mapping.findForward（"success"）；//返回密
码
    }
    }
```

当LoginAction处理完后，就会转向相应的页面，即页面success.jsp，其代码如下：

```
    <%@page language="java"contentType="text/html;
charset=GB18030"
    pageEncoding="GB18030"%>
    <html>
    <head>
    <title>Insert title here</title>
    </head>
    <body>
     {loginForm.username}，用户登录成功!
     {loginForm.password}，是其密码!
    </body>
    </html>
```

把该应用程序配置到服务器后，用IE浏览器来查看index.jsp，运行效果如图26.2所示。单击“登录”超级链接，运行效果如图26.3所示。

图　26.2　index.jsp页面

图　26.3　登录页面

在用户和密码文本框中填写相应的信息，单击“登录”按钮后，运行效果如图26.4所示。

图 26.4 转向页面

源程序解读

（1）在上述代码能运行之前，必须要配置文件strutsconfig.xml，其代码如下：

```
    <? xml version="1.0"encoding="ISO88591"? >
    <strutsconfig>
    <formbeans>
    <formbean
name="loginForm"type="com.cjg.user.forms.LoginActio
nForm"/>
    </formbeans>
    <actionmappings>
    <action path="/logininput"
    forward="/login.jsp"
    ></action>
    <action path="/login"
    type="com.cjg.user.actions.LoginAction"
    name="loginForm"
    scope="request"
```

```
    >
    <forward name="success"path="/success.jsp"/>
<! 跳转登录成功页面>
    </action>
    </actionmappings>
    <messageresources parameter="MessageResources"/
>
    </strutsconfig>
```

上述代码首先配置了ActionFrom，即代码中的loginForm。接着配置Action，在标签<action>中，属性path的值表示如何来访问Action，注意值是以“/”开始。如果在Action中使用了ActionFrom，就要设置属性name的值为loginForm，该ActionFrom的使用范围是request。在其子标签转向标签<forward>中，当返回值为“success”时，转到页面success.jsp上。

（2）把业务层对象配置到spring配置文件applicationContextbeans.xml中，其代码如下：

```
    <bean
id="userManager"class="com.cjg.user.manager.UserMan
```

```
agerAdd"/>
    </beans>
```

（3）在文件LoginAction.java中，如何创建对象BeanFactory呢？当使用如下方式时：

```
    BeanFactory factory=new
ClassPathXmlApplicationContext（"applicationContext
beans.xml"）;
```

每次请求都会创建一个BeanFactory对象，显然这是不符合要求。这时可以使用Spring的listener类，其会读取sping的配置文件，然后根据配置文件创建一个对象BeanFactory，然后把这个对象放在ServletContext中。在使用listener类之前，必须先在配置文件web.xml中实现配置，其代码如下：

```
    <? xml version="1.0"encoding="UTF8"? >
    <welcomefilelist>
    <welcomefile>index.jsp</welcomefile><! 定义欢
迎页面，即登录后第一个页面>
    </welcomefilelist>
    <servlet>
    <servletname>action</servletname>
```

```
    <servletclass>
org.apache.struts.action.ActionServlet
</servletclass>
    <initparam>
    <paramname>config</paramname>
    <paramvalue>/WEBINF/strutsconfig.xml
</paramvalue><! 指定Struts配置文件>
    </initparam>
    <initparam>
    <paramname>debug</paramname>
    <paramvalue>2</paramvalue>
    </initparam>
    <initparam>
    <paramname>detail</paramname>
    <paramvalue>2</paramvalue>
    </initparam>
    <loadonstartup>2</loadonstartup>
    </servlet>
    <! 设置servlet配置文件>
    <servletmapping>
    <servletname>action</servletname>
    <urlpattern>.do</urlpattern>
    </servletmapping>
    <contextparam>
    <paramname>contextConfigLocation</paramname>
    <paramvalue>classpath: applicationContext.xml
</paramvalue>
    </contextparam>
    <listener>
    <listenerclass>
org.springframework.web.context.ContextLoaderListen
er</listenerclass>
    </listener>
    </webapp>
```

在上述代码中，标签＜contextparam＞的作用就是制定配置文件的位置，而标签＜listener＞的作用则是指定Sping中的类来实现配置文件的读取。注意标签＜contextparam＞中子标签＜paramname＞的值不能改变。查看API可以得到如下定义：

```
    public class ContextLoaderListener implements
ServletContextListener {
    //实现ServletContextListener接口中的方法
    protected ContextLoader createContextLoader（） {
    return new ContextLoader（）; //返回ContextLoader
对象
    }
    }
```

在方法ContextLoaderListener中，有一个方法来创建类ContextLoader，查看API可以得到如下定义：

```
    public class ContextLoader {
    public static final String CONFIG__LOCATION__
PARAM="contextConfigLocation";
    }
```

在方法ContextLoaderListener中有一个常量CONFIG__LOCATION__PARAM，其值为contextConfigLocation。

（4）listener类只是把对象放在了对象ServletContext中，如何读取出来呢？Spring提供了一个辅助类WebApplicationContextUtils，通过该类的方法getRequiredWebApplicationContext就可以实现相应功能。在getRequiredWebApplicationContext方法中，需要一个ServletContext类型参数，而request.getSession（）.getServletContext（）就可以得到当前ServletContext。这就是在文件LoginAction.java中出现下面代码的原因。

```
    BeanFactory
factory=WebApplicationContextUtils.getRequiredWebAp
plicationContext（request.getSession
    （）.getServletContext（））；
```

（5）该应用程序所采用的方案虽然实现了Struts和Spring的结合，但是却因为BeanFactory的获取，而存在依赖关系。

26.2 Action注入方式实现Spring与Struts结合

上一小节实现Struts与Spring的开发方案虽然已使用，但是该方案有依赖性。即在Action中直接使用了依赖查找，使得Action离不开BeanFactory对象。由于第一种方案的原理是在Action中取得BeanFactory对象，然后通过BeanFactory获取业务逻辑对象。为了解决第一种方案的弊端，可以把业务对象抽象化，然后通过Spring注入到Action中。

技术要点

这节的方案就是将业务逻辑对象通过Spring注入到Action中，从而避免了在Action类中的直接代码查询。

业务逻辑对象注入。

Action配置。

实现代码

首先介绍代码的运行背景，培训机构在接受学生报名时，就要把学生的信息保存起来。在保存学生信息时，需要检查只有交过钱的学生的信息才能被保存起来。建立首页index.jsp代码如下：

```
    <html>
    <head>
    <meta
httpequiv="ContentType"content="text/html;
charset=GB18030">
    <title>spring和struts的集成</title>
    </head>
    <body><a href="logininput.do">登录</a>
    </body>
    </html>
```

登录页面login.jsp的代码如下：

```
    <html>
```

```
    <head>
    <title>用户登录</title>
    </head>
    <body>
    <form action="login.do"method="post">
    用户：<input type="text"name="username"><br>
    密码：<input type="password"name="password"><br
>
    <input type="submit"value="登录">
    </form>
    </body>
    </html>
```

当登录页面login.jsp发出请求后，其请求的信息会被封装到Form中，然后由ActionServlet传给相应的Action，接着在包com.cjg.user.forms中建立LoginActionForm.java，实现对信息的封装。

```
    package com.cjg.user.forms;
    import org.apache.struts.action.ActionForm;
    public class LoginActionForm extends
ActionForm {
    private String username; //定义用户名字段
    private String password; //定义密码字段
    public String getUsername() {//定义用户名字段的
get方法
    return username;
    }
    public void setUsername(String username) {//定
义用户名字段的set方法
    this.username=username;
```

```
    }
    public String getPassword（）｛定义密码字段的get方法
    return password;
    }
    public void setPassword（String password）｛//定义密码字段的set方法
    this.password=password;
    }
    }
```

编写业务层，对LoginActionForm中封装的信息实现相应的功能，在包com.cjg.user.manager中建立接口UserManager，其代码如下：

```
    package com.cjg.user.manager;
    //定义用户登录接口
    public interface UserManager｛
    //定义登录方法
    public void login（String username, String password）;
    }
```

在包com.cjg.user.manager中建立类UserManagerAdd，其代码如下：

```
    package com.cjg.user.manager;
    //定义登录类实现上面的接口
```

```
    public class UserManagerAdd implements
UserManager {
    //实现接口中的登录方法
    public void login（String username, String
password） {
    System.out.println（"UserManagerImpl.login（）
username="+username）;
    }
    }
```

在包com.cjg.user.actions中建立类LoginAction.java实现对请求信息的操作，其代码如下：

```
    package com.cjg.user.actions;
    import javax.servlet.http.HttpServletRequest;
    import javax.servlet.http.HttpServletResponse;
    import org.apache.struts.action.Action;
    import org.apache.struts.action.ActionForm;
    import org.apache.struts.action.ActionForward;
    import org.apache.struts.action.ActionMapping;
    import com.cjg.user.forms.LoginActionForm;
    import com.cjg.user.manager.UserManager;
    public class LoginAction extends Action {
    private UserManager userManager;
    //重写父类方法
    public ActionForward execute（ActionMapping
mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse
response）
    throws Exception {
    LoginActionForm laf=（LoginActionForm）form;
```

```
    userManager.login（laf.getUsername（），
laf.getPassword（））；//登录
    return mapping.findForward（"success"）；//跳转成
功页面
    }
    public void setUserManager（UserManager
userManager）{
    this.userManager=userManager;
    }
    }
```

当LoginAction处理完后，就会转向相应的页面，即页面success.jsp，其代码如下：

```
    <html>
    <head>
    <meta
httpequiv="ContentType"content="text/html;
charset=GB18030">
    <title>Insert title here</title>
    </head>
    <body>
     {loginForm.username}，用户登录成功!
     {loginForm.password}，是其密码!
    </body>
    </html>
```

把该应用程序配置到服务器后，用IE浏览器来查看index.jsp，运行效果如图26.5所示。单击“登录”

按钮超级链接，运行效果如图26.6所示。在用户和密码文本框中填写相应的信息，单击“登录”按钮后，运行效果如图26.7所示。

图　26.5　index.jsp页面

图　26.6　登录页面

图　26.7　转向页面

源程序解读

（1）由于对象userManager是通过Spring注入到类LoginAction中的，所以必须提供如下代码：

```
    public class LoginAction extends Action {
    private UserManager userManager;
    ……
    public void setUserManager（UserManager
userManager） {
    this.userManager=userManager;
    }
    }
```

（2）如何解决Action中依赖关系呢？可以把Action注入到Spring的IOC中，这样IOC容器就可以创

建和管理Action，从而消除Action中的依赖关系。其配置文件如下：

```
    <bean
name="/login"class="com.cjg.user.actions.LoginActio
n"scope="prototype">
    <property name="userManager"ref="userManager"/
>
    </bean>
    </beans>
```

登录页面发出请求，然后在strutsconfig.xml配置文件中找path属性为/login的标签<action>。接着查看该标签的type属性的值是否实例化，如果没有实例化Action, Struts会自动实例化。所以当配置文件中代码为下面代码时，该配置文件不起任何作用，因为Action是由Struts来实现的。

```
    <bean
id="loginAction"class="com.cjg.user.actions.LoginAc
tion"scope="prototype">
    <property name="userManager"ref="userManager"/
>
    </bean>
```

可以使用Spring中Action代理类ActionProxy来实现Action的注入，当使用ActionProx时，必须对strutsconfig.xml文件实现如下配置：

```
    <strutsconfig>
    <formbeans>
    <formbean
name="loginForm"type="com.cjg.user.forms.LoginActio
nForm"/>
    </formbeans>
    <actionmappings>
    <! 跳转到登录页面>
    <action path="/logininput"
    forward="/login.jsp"
    ></action>
    <action path="/login"
    type="org.springframework.web.struts.DelegatingA
ctionProxy"
    name="loginForm"
    scope="request">
    <! 如果登录成功，跳转到成功页面>
    <forward name="success"path="/success.jsp"/>
    </action>
    </actionmappings>
    <messageresources parameter="MessageResources"/
>
    </strutsconfig>
```

登录页面时发出请求，然后在strutsconfig.xml配置文件中找path属性为/login的标签<action>。

再把请求交给代理类

org.springframework.web.struts.DelegatingActionProxy，在代理类创建BeanFactory对象后，对象BeanFactory就可以根据“name=/login”在Spring配置文件中创建出真正的Action文件。这时Struts的运行流程图如图26.8所示。

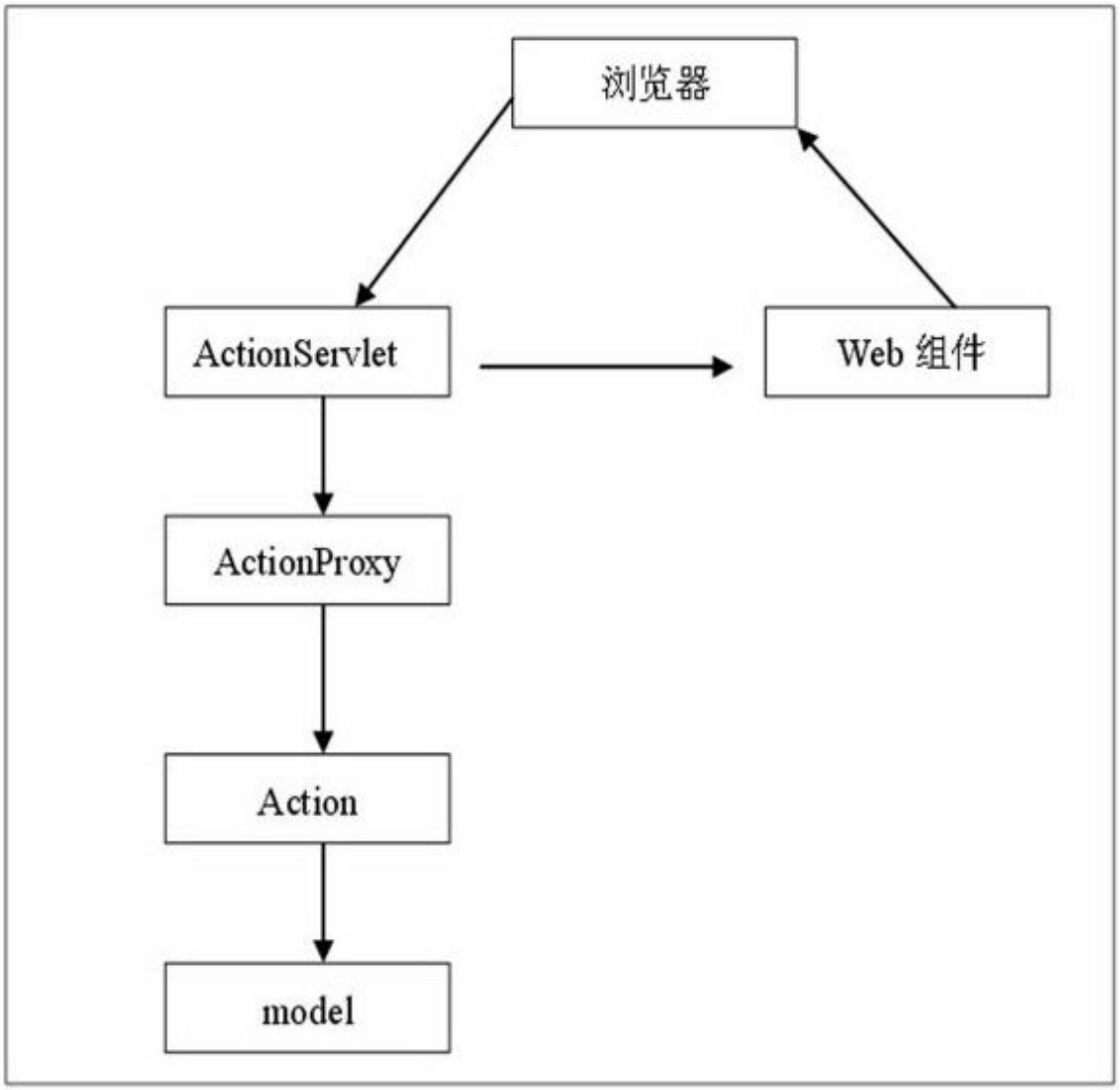

图　26.8　Struts流程图

（3）由于Struts中经常会创建多个Action实例对象，而Spring默认会创建一个实例对象，所以在＜bean＞标签中的Scope属性值为prototype，例如：

```
    <bean
name="/login"class="com.cjg.user.actions.LoginActio
n"scope="prototype">
```

（4）其他内容跟上一节一样，所以就不再讲解。