

Java Web 框架基础及实例

《Java Web 开发教程——入门与提高篇（JSP+Servlet）》之附赠资料

李绪成

2009.4

本文档内容可能存在错误和不当之处，有问题请联系 lixucheng@dl.cn

21 世纪高等学校计算机教育实用规划教材

Java Web 开发教程

——入门与提高篇（JSP+Servlet）

(ISBN 978-7-302-19177-3)

李绪成 闫海珍 主编

清华大学出版社

北京

目录

第一部分：框架基础.....	5
第 1 章 Web应用分层.....	6
1.1 单层应用	6
1.2 增加业务处理层	8
1.3 增加控制器	9
1.4 增加值对象	11
1.5 增加DAO层.....	13
1.6 增加接口并采用注入	14
1.7 使用JPA	16
1.8 Web层框架概述	17
第 2 章 Struts 2.....	18
2.1 概述	18
2.2 开发人员的主要任务	23
2.3 实例	34
第 3 章 JSF技术.....	39
3.1 概述	39
3.2 开发人员的主要工作	42
3.3 实例	48
第 4 章 简单自定义Web层框架.....	53
4.1 解析路径	54
4.2 创建JavaBean的实例	56
4.3 从视图向JavaBean传值	56
4.4 为JavaBean传递其它信息	57
4.5 调用JavaBean方法	57
4.6 从JavaBean向视图传值	58
4.7 响应	59
4.8 小结	59
第 5 章 Java持久性技术.....	61
5.1 持久性的概念	61
5.2 持久性研究的主要内容	61
5.3 持久性实现的方式	63
第 6 章 iBATIS技术.....	65
6.1 概述	65
6.2 开发人员的主要任务	65
6.3 实例	67
第 7 章 Hibernate技术.....	75
7.1 概述	75

7.2 开发人员的工作	76
7.3 实例	79
第 8 章 JPA 技术	84
8.1 概述	84
8.2 开发人员的工作	84
8.3 实例	87
第 9 章 简单自定义持久层框架	93
9.1 连接数据库	95
9.2 编写元注释类	96
9.3 获取映射信息	97
9.4 构造SQL语句	98
9.5 执行SQL语句	100
9.6 处理结果集	100
9.7 自定义框架小结	101
第 2 部分 实例	102
第 10 章 JavaMail	103
10.1 E-mail体系结构	103
10.2 JavaMail API	104
10.3 WebLogic中邮件会话的配置	106
10.4 邮件发送示例程序	107
10.5 邮件接收示例程序	113
第 11 章 办公用品申请管理系统	118
11.1 功能描述	118
11.2 涉及的文件及关系	119
11.3 控制器文件	119
11.4 数据库访问Bean	123
11.5 业务类	125
11.6 界面	130
11.7 配置文件	133
第 12 章 自动信息收集系统	135
12.1 需求描述	135
12.2 设计	136
12.3 核心代码	146

第一部分：框架基础

主要内容：

第 1 章 Web 应用分层

第 2 章 Struts 2

第 3 章 JSF 技术

第 4 章 简单自定义 Web 层框架

第 5 章 Java 持久性技术

第 6 章 iBATIS 技术

第 7 章 Hibernate 技术

第 8 章 JPA 技术

第 9 章 简单自定义持久层框架

第 1 章 Web 应用分层

目标：

- 了解常见的 Web 应用程序的层以及他们之间的关系。

主要内容：

- 单层应用；
- 业务处理层（模型层）；
- 控制层，连接视图层与模型层；
- 值对象（VO），在各层之间传递值；
- 数据访问层（DAO）；
- 接口层；
- 使用 JPA。

1.1 单层应用

全部采用 JSP 文件，并且在同一个文件中出现输入输出、处理和控制。

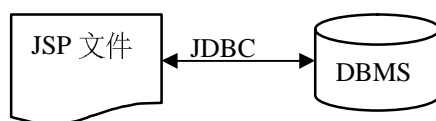


图 1.1 单层应用

输入界面 login.jsp

```
<% @ page contentType="text/html;charset=gb2312"%>
<script language="JavaScript">
    function isValid(form)
    {
        // 得到用户输入的信息
        username = form.username.value;
        userpass = form.userpass.value;

        // 判断用户名长度
        if(!minLength(username,6))
        {
            alert("用户名长度小于 6 位！");
            form.username.focus();
            return false;
        }
        if(!maxLength(username,8))
        {
```

```
        alert("用户名长度大于 8 位! ");
        form.username.focus();
        return false;
    }

    // 判断口令长度
    if(!minLength(userpass,6))
    {
        alert("口令长度小于 6 位! ");
        form.userpass.focus();
        return false;
    }
    if(!maxLength(userpass,8))
    {
        alert("口令长度大于 8 位! ");
        form.userpass.focus();
        return false;
    }

    return true;
}
// 验证是否满足最小长度
function minLength(str,length)
{
    if(str.length>=length)
        return true;
    else
        return false;
}
// 判断是否满足最大长度
function maxLength(str,length)
{
    if(str.length<=length)
        return true;
    else
        return false;
}
}
</script>
<html>
    <head>
        <title>用户登陆</title>
    </head>
```

```

<body>
  <h2>用户登录</h2>
  <form name="form1" action="process.jsp" method="post"
    onsubmit="return isValidate(form1)">
    用户名: <input type="text" name="username"> <br>
    口令: <input type="password" name="userpass"><br>
    <input type="reset" value="重置">
    <input type="submit" value="提交"><br>
  </form>
</body>
</html>

```

处理文件: process.jsp

主要内容: 接收用户输入的用户名和口令, 连接数据库, 判断用户是否存在, 如果存在显示登录成功信息, 如果登录失败, 显示失败信息。

```

<% @ page contentType="text/html;charset=gb2312"%>
<% @ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<c:if test="${param.username=="zhangsan" && param.userpass=="wangwu"}">
  <h2>您好, 欢迎登录网上书店! </h2>
</c:if>
<c:if test="${param.username!="zhangsan" || param.userpass!="wangwu"}">
  <h2>用户名或者口令不正确, 请<a href="login.jsp">重新登录! </a></h2>
</c:if>

```

1.2 增加业务处理层

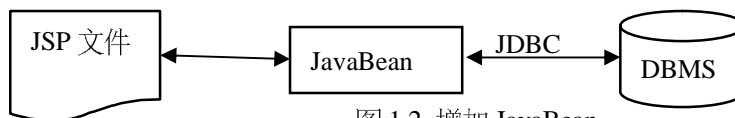


图 1.2 增加 JavaBean

添加一个 UserService 专门负责用户登录信息的验证。添加一个验证的方法, 方法中的代码用于连接数据库进行判断。

```

package service;

public class UserService {
  private String username;
  private String userpass;
  public String getUsername() {
    return username;
  }
  public void setUsername(String username) {
    this.username = username;
  }
  public String getUserpass() {

```



```

        return userpass;
    }
    public void setUserpass(String userpass) {
        this.userpass = userpass;
    }
    public boolean getLogin(){
        return username.equals("zhangsan")&&userpass.equals("wangwu");
    }
}

```

输入界面不变。

输出界面 `process.jsp`: 接收用户输入信息, 调用 `UserService` 的方法, 根据执行的结果分别显示成功或者失败的信息。

```

<% @ page contentType="text/html;charset=gb2312"%>
<% @ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<jsp:useBean id="user" class="service.UserService"/>
<jsp:setProperty property="*" name="user"/>

<c:if test="${user.login}">
    <h2>${sessionScope.username}您好, 欢迎登录网上书店! </h2>
</c:if>
<c:if test="${not user.login}">
    <h2>用户名或者口令不正确, 请<a href="login.jsp">ssss 重新登录! </a></h2>
</c:if>

```

1.3 增加控制器

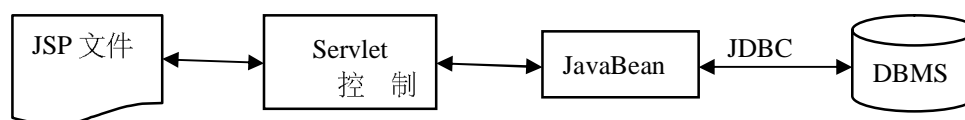


图 1.3 增加 Servlet 控制器

`UserService` 不变。

输入界面不变。

登录成功和登录失败分别使用一个界面表示。

登录成功的界面:

```

<% @ page contentType="text/html;charset=gb2312"%>
<html>
    <head>
        <title>登录成功</title>
    </head>
    <body>
        <h2>您好, 欢迎登录网上书店! </h2>
    </body>

```

```
</html>
```

登录失败的界面：

```
<% @ page contentType="text/html;charset=gb2312"%>
<html>
    <head>
        <title>登录失败</title>
    </head>
    <body>
        <h2>用户名或者口令不正确，请<a href="login.jsp">重新登录！ </a></h2>
    </body>
</html>
```

增加控制器，使用 **Servlet** 实现。主要代码：获取用户名和口令，调用 **UserService**，根据调用的结果选择界面对用户响应。

```
package controller;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import service.UserService;

public class LoginServlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        // 获取用户输入信息
        String username = request.getParameter("username");
        String userpass = request.getParameter("userpass");

        // 创建业务模型对象
        UserService user = new UserService();
        // 初始化
        user.setUsername(username);
        user.setUserpass(userpass);
        // 调用业务方法
        boolean login = user.getLogin();

        // 根据业务方法的执行结果选择界面对用户响应
```

```
String forward;
if(login)
    forward = "success.jsp";
else
    forward = "failure.jsp";
    // 获取 Dispatcher 对象
    RequestDispatcher dispatcher = request.getRequestDispatcher(forward);
    // 完成跳转
    dispatcher.forward(request,response);

}

public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    doGet(request,response);
}

}
```

控制器需要在 web.xml 中配置，代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
    <servlet>
        <description>This is the description of my J2EE component</description>
        <display-name>This is the display name of my J2EE component</display-name>
        <servlet-name>LoginServlet</servlet-name>
        <servlet-class>controller.LoginServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>LoginServlet</servlet-name>
        <url-pattern>/login</url-pattern>
    </servlet-mapping>
</web-app>
```

1.4 增加值对象

增加值对象，在各层之间传递信息。

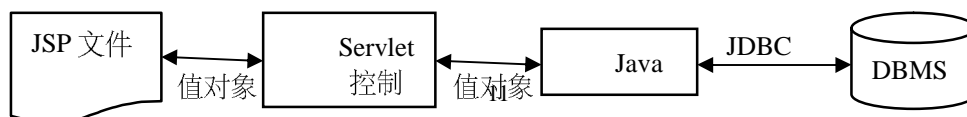


图 1.4 增加值对象

使用 User 类表示用户信息。通常表示称为 DTO（数据传输对象）。

```
package vo;

public class User {
    private String username;
    private String userpass;
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getUserpass() {
        return userpass;
    }
    public void setUserpass(String userpass) {
        this.userpass = userpass;
    }
}
```

修改 Servlet，先把用户名和口令构造成为 User 对象，把 User 对象作为参数调用 UserService。

```
// 获取用户输入信息
String username = request.getParameter("username");
String userpass = request.getParameter("userpass");

// 应用值
User user = new User();
// 初始化
user.setUsername(username);
user.setUserpass(userpass);

// 创建业务模型对象
UserService service = new UserService();
// 调用业务方法
boolean login = service.getLogin(user);

// 根据业务方法的执行结果选择界面对用户响应
String forward;
if(login)
    forward = "success.jsp";
```

```

else
    forward = "failure.jsp";
    // 获取 Dispatcher 对象
    RequestDispatcher dispatcher = request.getRequestDispatcher(forward);
    // 完成跳转
    dispatcher.forward(request,response);

```

修改 UserService，把参数修改为 User 对象。

```

package service;

import vo.User;

public class UserService {
    public boolean getLogin(User user){
        return user.getUsername().equals("zhangsan")
            &&user.getUserpass().equals("wangwu");
    }
}

```

输入和输出界面 JSP 文件不用动。

在 Struts1.2 中，专门使用 ActionForm 在传递视图层和控制层之间传递值。

1.5 增加 DAO 层

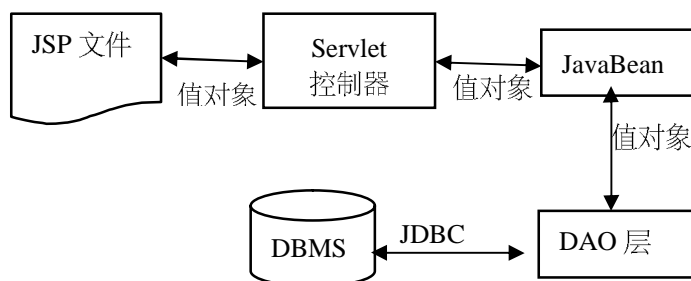


图 1.5 增加 DAO

把数据访问层单独提取出来，形成 DAO 层（下面的代码是简单的模拟，实际的代码要访问数据库）。

```

package dao;

import vo.User;

public class UserDao {
    public User findUserById(String username){
        User user = null;
        if(username.equals("zhangsan")){
            user = new User();
            user.setUsername(username);

```

```

        user.setUserpass("wangwu");
    }
    return user;
}
}

```

增加 UserDao 类，把 UserService 类调用 UserDao 的方法进行验证。

```

package service;

import vo.User;
import dao.UserDao;

public class UserService {
    public boolean getLogin(User user){
        // 创建 DAO 对象
        UserDao dao = new UserDao();
        // 根据用户名查询用户，假设用户名为主键
        User temp = dao.findUserById(user.getUsername());
        // 如果是否存在用户以及口令是否正确
        return temp==null?false:temp.getUserpass().equals(user.getUserpass());
    }
}

```

修改 UserService 中的代码，调用 UserDao 即可。

主要代码：创建 UserDao 对象，调用 UserDao 的方法。

其他文件不变

1.6 增加接口并采用注入

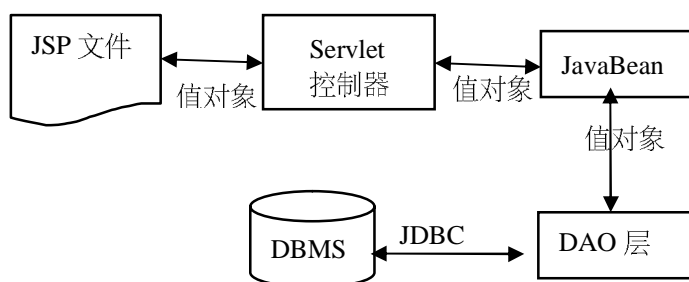


图 1.6 增加接口采用注入

把 UserDao 改写成接口，代码如下：

```

package dao;

import vo.User;

public interface UserDao {

```

```
public User findUserById(String username);
}
使用 SampleUserDao 模拟实现 dao 层，代码如下：
package dao;

import vo.User;

public class SampleUserDao implements UserDao{
public User findUserById(String username){
    System.out.println("SampleUserDao 中实现 dao");
    User user = null;
    if(username.equals("zhangsan")){
        user = new User();
        user.setUsername(username);
        user.setUserpass("wangwu");
    }
    return user;
}
}
```

修改 UserSerice：在 UserService 中增加成员变量，类型为 UserDao 对象。增加对 UserDao 对象进行操作的 set 方法和 get 方法。

```
package service;

import vo.User;
import dao.UserDao;

public class UserService {
private UserDao dao;

public UserDao getDao() {
    return dao;
}

public void setDao(UserDao dao) {
    this.dao = dao;
}

public boolean getLogin(User user){
    // 根据用户名查询用户，假设用户名为主键
    User temp = dao.findUserById(user.getUsername());
    // 如果是否存在用户以及口令是否正确
    return temp==null?false:temp.getUserpass().equals(user.getUserpass());
}
```

```

}
}

```

修改 Servlet: 在 Servlet 中创建 UserService 的对象之后, 再创建 UserDao 对象, 然后调用 UserService 的 setUserdao 方法, 然后再调用 check 方法。

```

// 获取用户输入信息
String username = request.getParameter("username");
String userpass = request.getParameter("userpass");

// 应用值
User user = new User();
// 初始化
user.setUsername(username);
user.setUserpass(userpass);

UserDao dao = new SampleUserDao();
// 创建业务模型对象
UserService service = new UserService();
// 初始化 dao
service.setDao(dao);
// 调用业务方法
boolean login = service.getLogin(user);

// 根据业务方法的执行结果选择界面对用户响应
String forward;
if(login)
    forward = "success.jsp";
else
    forward = "failure.jsp";
// 获取 Dispatcher 对象
RequestDispatcher dispatcher = request.getRequestDispatcher(forward);
// 完成跳转
dispatcher.forward(request,response);

```

其他文件不改。

1.7 使用 JPA

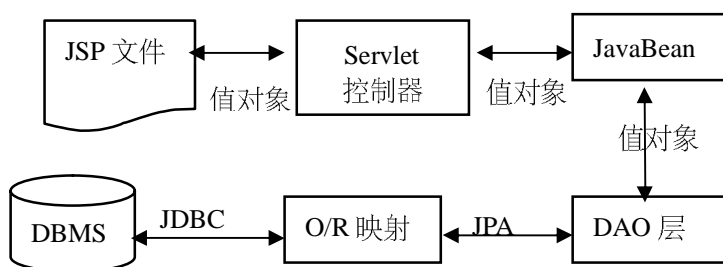


图 1.7 增加 JPA 技术

重新实现 JPADao。关于 JPA 参考本书 22.4 部分。

1.8 Web 层框架概述

框架的目的是减少重复工作，降低开发人员的工作量，提高开发效率，所以框架主要解决一些公共问题。

在 Java Web 应用开发中，比较共性的问题有：

- 使用 JSP 文件显示信息，显示信息通常使用 EL 表达式语言，有时候为了方便信息的显示，很多框架会提供自定义标签；
- 使用 Servlet 进行控制，Servlet 完成的主要任务包括接收信息、信息验证、信息转换、调用 JavaBean、对用户响应，因为执行过程基本不变化，所以通常都会完成一个统一的控制器，然后编写辅助类来完成不能统一处理的功能；
- 各层之间值的传递，各层之间值的传递包括视图层与控制器之间的传值以及控制层与模型层之间的传值，后者通常是通过方法调用完成的，前者通常采用专门的处理机制，例如在 Struts1 中，采用 ActionForm 来完成传值；
- 文件之间的跳转关系，为了能够灵活处理文件之间的跳转关系，通常都采用配置文件的方式来描述。

通过以上的分析，可以看出一般框架应该具有的功能如下：

- 编写自定义标签库，方便用户的输入和输出；
- 编写中心控制器，负责总协调；
- 提供传值的机制，在各层之间传值；
- 提供配置文件来描述文件之间的跳转关系。

现在比较流行的 Web 层框架有：

- Struts1；
- Struts2；
- Tapestry；
- JSF；
- WebWork。

第 2 章 Struts 2

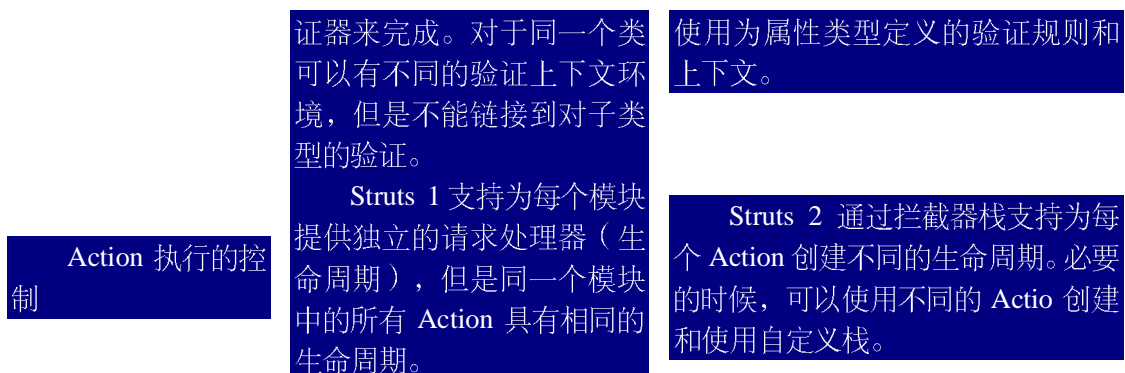
2.1 概述

Struts 现在分两个版本：Struts 1.X 和 Struts 2.X。Struts 1.X 已经有很多年了，可以说非常流行，但是因为其他框架的快速发展以及自身存在的问题，Struts 2 诞生了，Struts 2 与 Struts 1 的区别非常大，实际上 Struts 2 的核心思想是基于另外一个非常成功的 Web 框架 WebWork。两者的区别如表 20.1 所示。下面主要针对 Struts 2 进行介绍。

表 2.1 Struts1 和 Struts2 的比较

Feature	Struts 1	Struts 2
Action 类	在 Struts 1 中要求 Action 类继承抽象的基类。在 Struts 1 中一个普遍存在的问题就是面向抽象类编程，而不是面向接口编程。	Struts 2 中的 Action 可以实现一个 Action 接口，同时可以实现其他的接口，这样可以使用户有选择性地使用其它自定义的服务。Struts 2 提供了基础类 ActionSupport，该类实现了一些通用的接口。Action 接口不是必须的。任何具有 execute 方法的 POJO 对象都可以用作 Struts 2 的 Action 对象。
线程模型	Struts 1 的 Actions 是单例的，因为只有一个类的实例来处理所有对这个 Action 的请求，所以必须是线程安全的。单例策略对 Struts 1 的 Action 的能够完成的功能有很大限制，有些功能需要额外的努力才能完成。Action 资源必须是线程安全的或者 synchronized	Struts 2 的 Action 对象是为每个请求实例化的，因此没有线程安全的问题。（在实践中，Servlet 容器会为每个请求生成多个 throw-away 对象，增加的对象不会对性能产生太大影响或者对垃圾回收产生影响）
Servlet 依赖	Struts 1 的 Action 依赖 Servlet API，因为当调用 Action 的 execute 方法时需要传参数 HttpServletRequest 和 HttpServletResponse。	Struts 2 的 Action 与容器不是紧密结合在一起的。多数情况下，servlet 上下文被表示为 Map 对象，允许对 Action 进行独立的测试。如果需要，Struts 2 的 Action 仍然可以访问原始的 request 和 response 对象。然而，其它框架元素可以减少或者消除对 HttpServletRequest 和

		HttpServletRequest 对象进行直接访问的必要。
可测试性	测试 Struts 1 Action 的一个主要障碍就是 execute 方法使用了 Servlet API。1 个第三方扩展 Struts TestCase，为 Struts 1 提供了一组模拟（mock）对象。	Struts 2 的 Action 可以通过实例化、设置属性和调用方法进行测试。依赖注入支持使测试更简单。
获取输入	Struts 1 使用 ActionForm 对象来获取输入。像 Action 一样，所有的 ActionForm 必须继承一个基类。因为其它的 JavaBean 不能用作 ActionForm，开发人员经常需要创建多余的类来获取输入。可以使用动态 Form 来替换传统的 ActionForm 类，但是开发人员同样可能需要重新描述已有的 JavaBean。	Struts 2 使用 Action 的属性作为输入属性，不用创建第二个输入对象。输入属性可以是复杂的对象类型，还可以有自己的属性。可以在页面中通过 taglib 访问 Action 属性。Struts 2 也支持 ActionForm 模式，以及 POJO 表单对象和 POJO Action。复杂对象类型，包括业务或者域对象，都可以作为输入/输出对象。模型驱动的特性简化了标签库对 POJO 输入对象的引用。
表达式语言	Struts 1 集成了 JSTL，所以可以使用 JSTL 的 EL 语言，EL 提供了基本的对象结构遍历（object graph traversal），但是集合以及索引属性支持比较弱。	Struts 2 可以使用 JSTL，同时 Struts 还支持另外一种功能更强大、使用更灵活的表达式语言，这种语言是 Object Graph Notation Language，简称 OGNL。
值与视图的绑定	Struts 1 使用了标准的 JSP 机制把对象与要访问的页面上下文绑定。	Struts 2 使用了一种 ValueStack 技术，这样标签库不用把视图与要呈现的对象类型关联就可以访问值。ValueStack 策略允许重用涉及多个类型的视图，这些类型可能有相同的属性名，但是属性类型不同。
类型转换	Struts 1 的 ActionForm 属性通常都是字符串类型。Struts 1 使用 Commons-Beanutils 进行类型转换。转换器是针对每个类的，而不能为每个实例配置。	Struts 2 使用 OGNL 进行类型转换，框架包含了常用对象类型和基本数据类型的转换器。
验证	Struts 1 支持手动验证，通过 ActionForm 的 validate 方法或者通过继承通用的验证	Struts 2 支持通过验证方法进行手工验证和 XWork 验证框架。Xwork 验证框架支持对子属性的链接验证，



注：来自 Struts 的官方网站：<http://struts.apache.org/2.0.11.2/docs/comparing-struts-1-and-2.html>
Struts 2 结构图如图 2.1（原图来自 Struts 2 文档）所示：

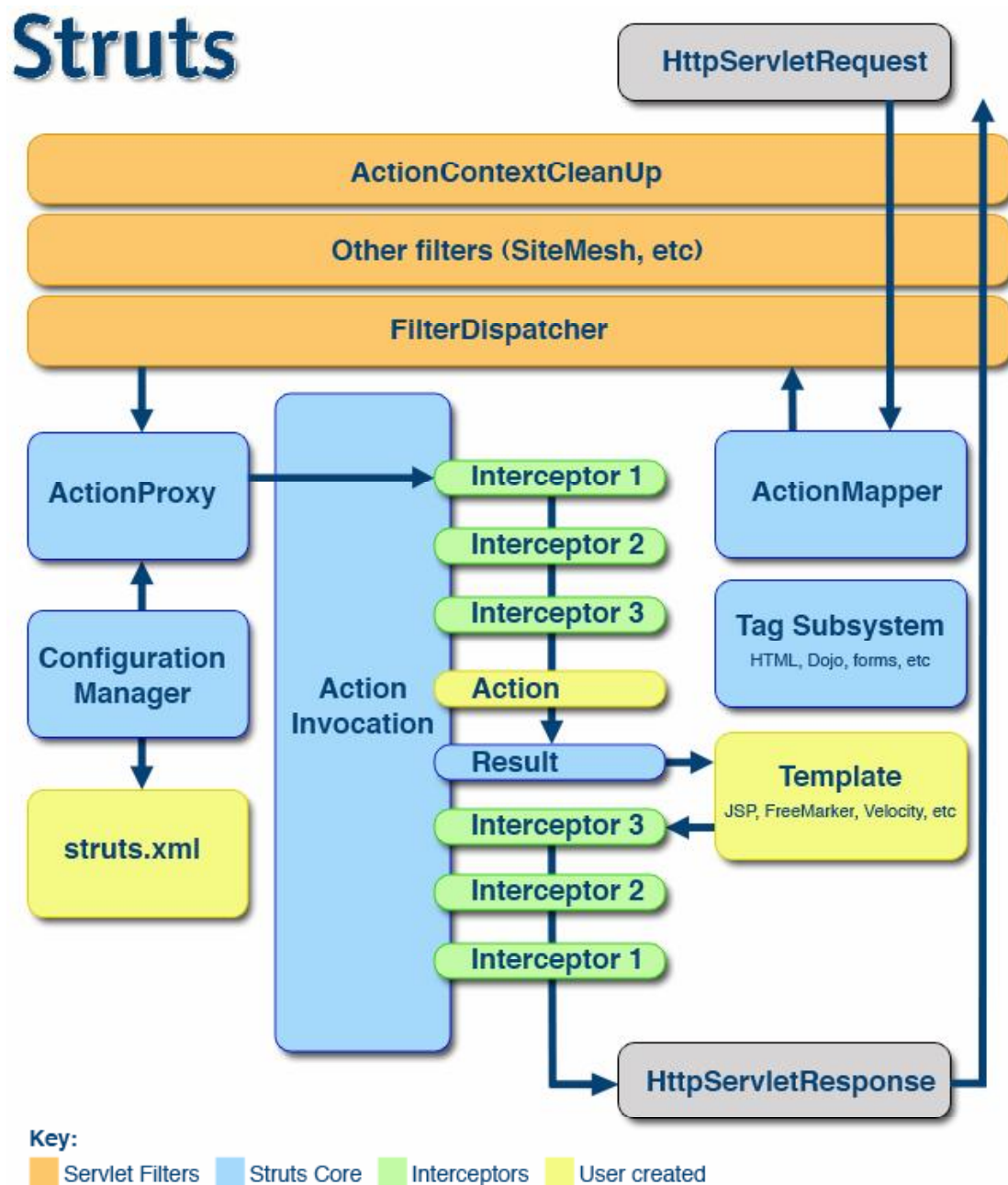


图 2.1 Struts2 结构图

在处理一个请求的时候，主要使用 3 个类：Action、Interceptor 和 Result
处理流程：

- ◆ 请求到达服务器之后，首先经过一系列过滤器，有的是可选的，最主要的过滤器是 **FilterDispatcher**。所有的请求都会提交给它处理，该过滤器是在 **web.xml** 中配置的。配置代码如下：

```
<filter>
  <filter-name>struts2</filter-name>
  <filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
```

```

</filter>

<filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

```

- ◆ **FilterDispatcher** 过滤器接收到请求之后调用 **ActionMapper** 查看是否需要调用 **Action**。**ActionMapper** 提供了 **HttpRequest** 与 **Action** 调用请求之间的映射关系，可以决定当前请求是否需要调用 **Action**。如果 **ActionMapper** 返回的信息表明需要调用 **Action**。**FilterDispatcher** 过滤器把控制前交给 **ActionProxy**；
- ◆ **ActionProxy** 调用配置文件管理器 **ConfigurationManager**，该管理器从 **struts.xml** 配置文件中获取配置信息，获取的信息主要包括当前请求对应哪个 **Action**（对用户的请求进行处理），对应哪些 **Result**（决定了如何对用户响应），有时候还涉及拦截器。然后根据这些信息创建 **ActionInvocation** 对象，该对象负责具体的调用过程。**struts.xml** 是用户需要提供的最主要的配置文件。下面是一个 **struts.xml** 配置文件的部分内容。

```

<struts>
    <package name="default" extends="struts-default">

        <action name="Logon" class="mailreader2.Logon">
            <result name="input">/pages/Logon.jsp</result>
            <result name="cancel" type="redirectAction">Welcome</result>
            <result type="redirectAction">MainMenu</result>
            <result name="expired" type="chain">ChangePassword</result>
        </action>

        <action name="Logoff" class="mailreader2.Logoff">
            <result type="redirectAction">Welcome</result>
        </action>

    </package>
</struts>

```

- ◆ **ActionInvocation** 对象按照顺序执行当前请求所对应的拦截器，拦截器能够对请求进行预处理，例如验证、文件上传等，并能够对响应内容进行再处理。通常拦截器是由系统提供的，如果需要，编程人员只需要进行配置即可。在调用 **Action** 的方法之前，会调用拦截器的预处理方法；
- ◆ **ActionInvocation** 对象调用拦截器的预处理方法之后会调用 **Action** 的 **execute** 方法，**Action** 中的代码主要由编程人员根据功能进行编写的，通常从数据库检索信息或者向数据库存储信息。**Action** 的方法返回一个字符串。下面是一个简单的 **Action** 例子。

```

package simple;
import java.util.Map;
import javax.servlet.http.HttpSession;

```

```
import com.opensymphony.webwork.ServletActionContext;
import com.opensymphony.xwork.ActionSupport;

public class LogoutAction extends ActionSupport {

    public String execute() throws Exception {
        Map session = ActionContext.getContext().getSession();
        session.remove("logged");
        session.remove("context");
        return SUCCESS;
    }

}
```

- ◆ ActionInvocation 对象根据 Action 方法的返回结果以及 struts 配置文件生成 Result 对象。Result 对象选择一个模板文件来响应用户,模板文件可以是 JSP、FreeMarker 和 Velocity。
- ◆ 容器加载并执行模板文件,使用在 Action 中获取的信息对模版中的变量进行赋值,也可能从资源文件或者其他内部对象中获取信息。最终向浏览器呈现的是 HTML、PDF 或者其他内容。
- ◆ 模板文件执行的结果会经过拦截器进行再处理,最后通过过滤器返回给客户端。

在该结构图中,既包含了 Struts 框架提供的基础接口,也包括了用户要编写的文件。其中,ActionMapper、ActionProxy、ConfigurationManager、ActionInvocation 和 Result 是框架提供的核心类。过滤器和拦截器是框架提供的,用户可以根据需要进行配置,当然也可以编写自己的过滤器和拦截器。用户需要编写的文件是 struts.xml、Action 和模板文件,这些也是用户在使用 Struts 2 框架时需要做的工作。

2.2 开发人员的主要任务

框架为开发人员提供了大量的辅助类,用户在使用框架开发的时候只需要编写很少文件。在使用 Struts 2 开发的时候,首先应该把环境搭建起来,然后使用 Struts 2 提供的标签开发界面,然后编写 Action 类,最后进行配置。

环境搭建

在进行具体的开发之前,需要先搭建环境。包括如下过程:

- ◆ 创建 Web 工程;
- ◆ 加载 Struts 2 的核心类库,核心类库包括 commons-logging-1.0.4.jar、freemarker-2.3.8.jar、ognl-2.6.11.jar、struts2-core-2.0.11.2.jar 和 xwork-2.0.5.jar,把这些类库放到 Web 工程的 WEB-INF/lib 下面;
- ◆ 配置 web.xml,主要配置 Struts 中心控制器 FilterDispatcher,下面是 1 个例子。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_9" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
```

```
<display-name>Struts Blank</display-name>

<filter>
    <filter-name>struts2</filter-name>
    <filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
</filter>

<filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

<welcome-file-list>
    <welcome-file>index.html</welcome-file>
</welcome-file-list>

</web-app>
```

- ◆ 创建 `struts.xml` 配置文件，与类文件放在一起，空白的 `struts` 文件如下所示。在使用 `Struts 2` 进行开发所有的配置基本上都在这个文件中完成。也可以根据需要创建多个配置文件，然后在这个配置文件中使⤵用`<include file="example.xml"/>`进行包含。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>

    <constant name="struts.enable.DynamicMethodInvocation" value="false" />
    <constant name="struts.devMode" value="false" />

    <include file="example.xml"/>

    <!-- Add packages here -->

</struts>
```

环境搭建完之后，在具体开发过程中主要完成 3 个方面的工作：

- ◆ 制作模板文件，可以使用 `JSP`、`FreeMarker` 或者 `Velocity` 等；
- ◆ 编写 `Action`，基本上每个动作对应 1 个 `Action`；
- ◆ 配置，主要在 `struts.xml` 中进行配置。

下面分别介绍。

制作模板文件

模版文件的主要作用是接收用户输入的信息，并向用户展示信息。`Struts` 提供了多个标签库

来简化页面的代码量，使用标签之后页面也更容易维护。下面是一段标签：

```
<s:actionerror/>
<s:form action="Profile_update" validate="true">
  <s:textfield label="Username" name="username"/>
  <s:password label="Password" name="password"/>
  <s:password label="(Repeat) Password" name="password2"/>
  <s:textfield label="Full Name" name="fullName"/>
  <s:textfield label="From Address" name="fromAddress"/>
  <s:textfield label="Reply To Address" name="replyToAddress"/>
  <s:submit value="Save" name="Save"/>
  <s:submit action="Register_cancel" value="Cancel" name="Cancel"
    onclick="form.onsubmit=null"/>
</s:form>
```

Struts 2 中提供了两类通用标签和 3 类界面标签：

- ◆ 控制标签
- ◆ 数据标签
- ◆ Form 标签
- ◆ Non-Form 用户接口标签
- ◆ Ajax 标签

下面对这些类型的标签进行介绍。

控制标签及其用法如表 2.2 所示。

表 2.2 控制标签

标签名	描述	例子
if	与 Java 中的 if 基本相同	<pre><s:if test="%{false}"> <div>Will Not Be Executed</div> </s:if> <s:elseif test="%{true}"> <div>Will Be Executed</div> </s:elseif> <s:else> <div>Will Not Be Executed</div> </s:else> <s:append var="myAppendIterator"> <s:param value="%{myList1}" /> <s:param value="%{myList2}" /> <s:param value="%{myList3}" /> </s:append> <s:iterator value="%{#myAppendIterator}"> <s:property /> </s:iterator> <s:generator val="%{'aaa,bbb,ccc,ddd,eee'}"> <s:iterator> <s:property />
</pre>
else if	与 Java 中的 else if 基本相同	
else	与 Java 中的 else 基本相同	
append	按照顺序把多个迭代器的元素组合到一个迭代器中，保持原来的顺序不变。	
generator	根据 val 属性的给定的值生成迭代器对象。	

iterator	对迭代器或者集合进行遍历，类似于 Java 中的 for-each 循环。	<pre> </s:iterator> </s:generator> <s:iterator value="#it.days" status="rowstatus"> <tr> <s:if test="#rowstatus.odd == true"> <td style="background: grey"><s:property/></td> </s:if> <s:else> <td><s:property/></td> </s:else> </tr> </s:iterator> <s:merge var="myMergedIterator1"> <s:param value="%{myList1}" /> <s:param value="%{myList2}" /> <s:param value="%{myList3}" /> </s:merge> <s:iterator value="%{#myMergedIterator1}"> <s:property /> </s:iterator> <s:sort var="mySortedList" comparator="myComparator" source="myList" /> <s:subset var="mySubset" source="myList" count="13" start="3" /> </pre>
merge	把多个迭代器的元素合并到一个迭代器中，合并后的顺序为 1.1, 2.1, 3.1, 1.2, 1.3..., 1.1 表示第 1 个迭代器的第 1 个元素。	
sort	对 List 进行排序。	
subset	获取集合的子集。	

数据标签及其用法如表 2.3 所示。

表 2.3 数据标签

标签名	描述	例子
a	生成 HTML 的<a>	<pre> <s:a href="%{testUrlId}"><img src="<s:url value="/images/delete.gif"/>" border="none"/></s:a> </pre>
action	在 JSP 页面中直接调用 Action	<pre> <s:action name="actionTagAction" executeResult="true" /> </pre>
bean	实例化 JavaBean 对象	<pre> <s:bean name="org.apache.struts2.example.counter.SimpleCounter" var="counter"> <s:param name="foo" value="BAR" /> The value of foot is : <s:property value="foo"/>
 </s:bean> <s:date name="person.birthday" format="dd/MM/yyyy" /> </pre>
date	创建 Date 对象	
debug		

il8n	得到 ResourceBundle 对象。	<pre><s:il8n name="myCustomBundle"> </s:il8n></pre>
include	包含 1 个 JSP 或者 Servlet 的输出。	<pre><s:include value="myJsp.jsp"> <s:param name="param1" value="value2" /> </s:include></pre>
param	为其他标签提供参数	参考上面的例子
property push	获取属性值 把值保存起来使用	参考 bean 标签的例子 <pre><s:push value="user"> <s:property value="firstName" /> <s:property value="lastName" /> </s:push></pre>
set	把某个值保存到某个作用范围的变量中。	<pre><s:set name="personName" value="person.name"/> Hello, <s:property value="#personName"/>. How are you?</pre>
text	呈现 il8n 的文本消息	<pre><s:il8n name="struts.action.test.il8n.Shop"> <s:text name="main.title"/> </s:il8n></pre>
url	用于生成 URL	<pre><s:url value="editGadget.action"> <s:param name="id" value="%{selected}" /> </s:url></pre>

Form 标签及其用法如表 2.4 所示。

表 2.4 Form 标签

标签名	描述	例子
checkbox	生成复选框	<pre><s:checkbox label="checkbox test" name="checkboxField1" value="aBoolean" fieldValue="true"/></pre>
checkboxlist	生成多个复选框	<pre><s:checkboxlist name="foo" list="bar"/></pre>
comboBox	输入框与下拉框的组合。	<pre><s:comboBox label="My Favourite Fruit" name="myFavouriteFruit" list="{ 'apple','banana','grape','pear' }" headerKey="-1" headerValue="--- Please Select ---" emptyOption="true" value="banana" /></pre>
doubleselect	生成联动菜单	<pre><s:doubleselect label="doubleselect test1" name="menu" list="{ 'fruit','other' }" doubleName="dishes" doubleList="top == 'fruit' ? { 'apple', 'orange' } : { 'monkey', 'chicken' }" /></pre>

head	生成 HTML 的 head 部分。	<pre><head> <title>My page</title> <s:head/> </head></pre>
file	生成文件输入框	<pre><s:file name="anUploadFile" accept="text/*" /></pre>
form	生成 form 表单	<pre><p/> <s:form ... /> <p/></pre>
hidden	生成隐藏域	<pre><s:hidden name="foo" value="bar" /></pre>
label	生成标签	<pre><s:label key="userName" /></pre>
optiontransferselect	生成两个列表框，可以通过中间的按钮把左边的选项移动到右边，也可以把右边的选项移动到左边。	<pre><s:optiontransferselect label="Favourite Cartoons Characters" name="leftSideCartoonCharacters" list="{ 'Popeye', 'He-Man', 'Spiderman' }" doubleName="rightSideCartoonCharacters" doubleList="{ 'Superman', 'Mickey Mouse', 'Donald Duck' }" /></pre>
optgroup	在 select 中提供选项	<pre><s:select label="My Selection" name="mySelection" value="% { 'POPEYE' }" list="% {#{ 'SUPERMAN': 'Superman', 'SPIDERMAN': 'spiderman' } }"> <s:optgroup label="Adult" list="% {#{ 'SOUTH_PARK': 'South Park' } }" /> <s:optgroup label="Japanese" list="% {#{ 'POKEMON': 'pokemon', 'DIGIMON': 'digimon', 'SAILORMOON': 'Sailormoon' } }" /> </s:select></pre>
password	密码输入框	<pre><s:password label="% { text('password') }" name="password" size="10" maxlength="15" /></pre>
radio	单选按钮	<pre><s:radio label="Gender" name="male" list="#genders.genders"/></pre>
reset	重置按钮	<pre><s:reset value="Reset" /></pre>

submit	提交按钮	<code><s:submit value="OK" /></code>
textarea	生成文本域	<code><s:textarea label="Comments" name="comments" cols="30", rows="8"/></code>
textfield	生成输入框	<code><s:textfield key="user" /></code>
token	阻止表单重复提交	<code><s:textfield key="user" /></code>
updownselect	创建元素能够上下移动的列表框	<code><s:updownselect list="#{'england':'England', 'america':'America', 'germany':'Germany'}" name="prioritisedFavouriteCountries" headerKey="-1" headerValue="--- Please Order Them Accordingly ---" emptyOption="true" /></code>

non-form UI 标签及其用法如表 2.5 所示。

表 2.5 non-form 标签

标签名	描述	例子
actionerror	呈现错误信息	<code><s:actionerror /></code>
actionmessage	呈现提示信息	<code><s:actionmessage /></code>
component	创建自定义组件	<code><s:component template="/my/custom/component.vm"/></code>
div	生成 HTML <code><div></code>	
fielderror	输出关于输入元素的错误信息	<code><s:fielderror> <s:param>field1</s:param> <s:param>field2</s:param> </s:fielderror> <s:form > </s:form></code>

Ajax 标签包括 a、autocompleter、bind、datetimepicker、div、head、submit、tabbedpanel、textarea、tree、treenode 等。具体用法参考 Struts 2 帮助文档。

编写 Action

针对每个功能可以编写 1 个 Action，也可以多个功能共享 1 个 Action。Action 完成的主要功能包括：

- ◆ 获取用户的输入信息，这个获取的过程是由框架完成的，但是用户需要在 Action 中定义与用户输入表单元素名字相同的成员变量，关键是要提供对成员变量赋值的 set 方法，这样框架在获取用户输入信息之后会调用 set 方法把值赋给 Action 的成员变量。

- ◆ 根据用户的请求信息，调用完成业务逻辑的 **JavaBean**。如果希望要把某些执行结果传递给模板文件（**JSP**、**FreeMarker** 和 **Velocity** 等），需要在 **Action** 中定义成员变量来表示这些结果，最关键的是要定义 **get** 方法，这样在执行模版文件的时候会通过 **get** 方法来获取这些信息。
- ◆ 根据执行的结果，返回 1 个字符串，这个字符串决定了使用什么模板对用户进行响应。下面是 1 个简单的例子。

```
public class LoginAction extends ActionSupport {  
  
    private String userId;  
    private String passwd;  
    // 对 userId 和 passwd 操作的 setter 和 getter 方法  
    public String execute() throws Exception {  
        if ("admin".equals(userId) && "password".equals(passwd)) {  
            Map session = ActionContext.getContext().getSession();  
            session.put("logged", "true");  
            session.put("context", new Date());  
            return SUCCESS;  
        }  
        return ERROR;  
    }  
}
```

注意：并不是必须继承 **ActionSupport**，主要提供 **execute** 方法即可。

配置

通过配置文件 **Struts.xml** 对 **Web** 应用的流程进行管理，包括 **Action** 映射和 **Result** 处理，前者把请求与 **Action** 关联起来，后者把 **Action** 执行的结果与响应界面关联起来。下面是一段配置。下面是一个简单的例子。

```
<struts>  
    <package name="default" extends="struts-default">  
  
        <action name="Logon" class="mailreader2.Logon">  
            <result name="input">/pages/Logon.jsp</result>  
            <result name="cancel" type="redirectAction">Welcome</result>  
            <result type="redirectAction">MainMenu</result>  
            <result name="expired" type="chain">ChangePassword</result>  
        </action>  
  
        <action name="Logoff" class="mailreader2.Logoff">  
            <result type="redirectAction">Welcome</result>  
        </action>  
  
    </package>  
</struts>
```

Struts 2 中完成的主要配置如表 2.6 所示。

表 2.6 Struts 2 的主要配置信息

配置元素	例子
JavaBean	<pre><bean type="com.opensymphony.xwork2.ObjectFactory" name="myfactory" class="com.company.myapp.MyObjectFactory" /> <constant name="struts.devMode" value="true" /></pre>
常量包命名空间	<pre><package name="employee" extends="struts-default" namespace="/employee"> ... </package> <include file="Home.xml"/></pre>
包含拦截器	<pre><interceptors> <interceptor name="security" class="com.company.security.SecurityInterceptor"/> <interceptor-stack name="secureStack"> <interceptor-ref name="security"/> <interceptor-ref name="defaultStack"/> </interceptor-stack> </interceptors></pre>
引用拦截器	<pre><action name="VelocityCounter" class="org.apache.struts2.example.counter.SimpleCounter"> <result name="success">...</result> <interceptor-ref name="defaultComponentStack"/> </action></pre>
Action Result	<pre>全局 Result: <global-results> <result name="error">/Error.jsp</result> <result name="invalid.token">/Error.jsp</result> </global-results></pre>
异常配置	<pre>在 Action 中使用: <exception-mapping exception="com.company.SecurityException" result="login"/> 全局: <global-exception-mappings> <exception-mapping exception="java.sql.SQLException" result="SQLException"/> <exception-mapping exception="java.lang.Exception" result="Exception"/> </global-exception-mappings></pre>

Struts 2 提供了大量的拦截器，用户可以根据需要调用。

Struts 2 的配置文件 struts.xml 的 DTD 定义如下。

```
<!--
```

Struts configuration DTD.

Use the following DOCTYPE

```
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">
-->

<!ELEMENT struts (package|include|bean|constant)*>

<!ELEMENT package (result-types?, interceptors?, default-interceptor-ref?, default-action-ref?,
default-class-ref?, global-results?, global-exception-mappings?, action*)>
<!ATTLIST package
    name CDATA #REQUIRED
    extends CDATA #IMPLIED
    namespace CDATA #IMPLIED
    abstract CDATA #IMPLIED
    externalReferenceResolver NMTOKEN #IMPLIED
>

<!ELEMENT result-types (result-type+)>

<!ELEMENT result-type (param*)>
<!ATTLIST result-type
    name CDATA #REQUIRED
    class CDATA #REQUIRED
    default (true|false) "false"
>

<!ELEMENT interceptors (interceptor|interceptor-stack)+>

<!ELEMENT interceptor (param*)>
<!ATTLIST interceptor
    name CDATA #REQUIRED
    class CDATA #REQUIRED
>

<!ELEMENT interceptor-stack (interceptor-ref*)>
<!ATTLIST interceptor-stack
    name CDATA #REQUIRED
>

<!ELEMENT interceptor-ref (param*)>
```



```
<!--ATTLIST interceptor-ref
      name CDATA #REQUIRED
-->

<!--ELEMENT default-interceptor-ref (param*)-->
<!--ATTLIST default-interceptor-ref
      name CDATA #REQUIRED
-->

<!--ELEMENT default-action-ref (param*)-->
<!--ATTLIST default-action-ref
      name CDATA #REQUIRED
-->

<!--ELEMENT default-class-ref (param*)-->
<!--ATTLIST default-class-ref
      class CDATA #REQUIRED
-->

<!--ELEMENT global-results (result+)-->

<!--ELEMENT global-exception-mappings (exception-mapping+)-->

<!--ELEMENT action (param|result|interceptor-ref|exception-mapping)*-->
<!--ATTLIST action
      name CDATA #REQUIRED
      class CDATA #IMPLIED
      method CDATA #IMPLIED
      converter CDATA #IMPLIED
-->

<!--ELEMENT param (#PCDATA)-->
<!--ATTLIST param
      name CDATA #REQUIRED
-->

<!--ELEMENT result (#PCDATA|param)*-->
<!--ATTLIST result
      name CDATA #IMPLIED
      type CDATA #IMPLIED
-->

<!--ELEMENT exception-mapping (#PCDATA|param)*-->
```

```
<!--ATTLIST exception-mapping
      name CDATA #IMPLIED
      exception CDATA #REQUIRED
      result CDATA #REQUIRED
-->

<!--ELEMENT include (#PCDATA)-->
<!--ATTLIST include
      file CDATA #REQUIRED
-->

<!--ELEMENT bean (#PCDATA)-->
<!--ATTLIST bean
      type CDATA #IMPLIED
      name CDATA #IMPLIED
      class CDATA #REQUIRED
      scope CDATA #IMPLIED
      static CDATA #IMPLIED
      optional CDATA #IMPLIED
-->

<!--ELEMENT constant (#PCDATA)-->
<!--ATTLIST constant
      name CDATA #REQUIRED
      value CDATA #REQUIRED
-->
```

2.3 实例

功能：登录。

涉及的文件有：

- Login.jsp，用于输入登录信息；
- welcome.jsp，登录之后的欢迎界面；
- loginCheck.jsp，判断用户是否登录；
- LoginAction.java，完成登录业务处理，正常情况下会调用其他业务逻辑 JavaBean 来完成；
- LogoutAction.java，完成退出业务处理；
- struts.xml，应用的配置文件。

下面分别介绍。

Login.jsp

源文件：Login.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head><body>
<form action="login.action" method="post">
User id<input type="text" name="userId" /> <br/>
Password <input type="password" name="passwd" /> <br />
<input type="submit" value="Login"/>
</form>
</body>
</html>
```

/pages/welcome.jsp

```
<% @ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<% @ taglib prefix="ww" uri="/webwork" %>
<jsp:include page="WEB-INF/inc/loginCheck.jsp" />
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Welcome</title>
</head>

<body>Welcome, you have logined. <br />
The attribute of 'context' in session is
<ww:property value="#session.context" />
<br /><br /><br />
<a xhref="<%= request.getContextPath() %>/logout.action">Logout</a>
<br />
<a xhref="<%= request.getContextPath() %>/logout2.action">Logout2</a>
</body>
</html>
```

/WEB-INF/inc/loginCheck.jsp

```
<% @ taglib="/webwork" prefix="ww" %>
<ww:if test="#session.login != 'true'">
<jsp:forward page="<%= request.getContextPath() %>/login.jsp" />
</ww:if>
```

simple.LoginAction.java

```
package simple;
```

```
import java.util.Date;import java.util.Map;

import javax.servlet.http.HttpSession;

import com.opensymphony.webwork.ServletActionContext;
import com.opensymphony.xwork.ActionSupport;

public class LoginAction extends ActionSupport {

    private String userId;
    private String passwd;

    public String execute() throws Exception {
        if ("admin".equals(userId) && "password".equals(passwd)) {
            // HttpSession session = ServletActionContext.getRequest().getSession();
            // session.setAttribute("logged","true");
            // session.setAttribute("context", new Date());
            // Better is using ActionContext
            Map session = ActionContext.getContext().getSession();
            session.put("logged","true");
            session.put("context", new Date());
            return SUCCESS;
        }
        return ERROR;
    }

    public String logout() throws Exception {
        // HttpSession session = ServletActionContext.getRequest().getSession();
        // session.removeAttribute("logged");
        // session.removeAttribute("context");
        Map session = ActionContext.getContext().getSession();
        session.remove("logged");
        session.remove("context");
        return SUCCESS;
    }

    public String getPasswd() {
        return passwd;
    }

    public void setPasswd(String passwd) {
        this.passwd = passwd;
    }
}
```

```
public String getUserId() {  
    return userId;  
}  
  
public void setUserId(String userId) {  
    this.userId = userId;  
}  
}
```

simple.LogoutAction.java

```
package simple;  
import java.util.Map;  
import javax.servlet.http.HttpSession;  
  
import com.opensymphony.webwork.ServletActionContext;  
import com.opensymphony.xwork.ActionSupport;  
  
public class LogoutAction extends ActionSupport {  
  
    public String execute() throws Exception {  
        Map session = ActionContext.getContext().getSession();  
        session.remove("logged");  
        session.remove("context");  
        return SUCCESS;  
    }  
  
}
```

/WEB-INF/classes/xwork.xml

```
<!DOCTYPE    xwork    PUBLIC    "-//OpenSymphony    Group//XWork    1.1.1//EN"  
"http://www.opensymphony.com/xwork/xwork-1.1.1.dtd">  
  
<xwork>  
    <include file="webwork-default.xml"/>  
  
    <package name="default" extends="webwork-default">  
        <!-- Add your actions here -->  
        <action name="login" class="simple.LoginAction" >  
            <result name="success" type="dispatcher">/pages/welcome.jsp</result>  
            <result name="error" type="redirect">/login.jsp</result>  
        </action>
```

```
<action name="logout2" class="simple.LoginAction" method="logout" >
    <result name="success" type="redirect">/login.jsp</result>
</action>

<action name="logout" class="simple.LogoutAction" >
    <result name="success" type="redirect">/login.jsp</result>
</action>
</package>
</xwork>
```

第 3 章 JSF 技术

JSF 是一种用于构建基于 Java 的 Web 应用程序的服务器端用户接口 (UI) 组件框架。它提供了一种以组件为中心来开发 Java Web 用户界面的方法,从而简化了开发。应用开发人员和 Web 设计人员将发现 JSF 开发可以简单到只需将 UI 组件拖放到页面上,而系统开发人员将发现丰富而强健的 JSF API 为他们提供了无与伦比的功能和编程灵活性。JSF 还将结构良好的模型-视图-控制器 (MVC) 设计模式集成到它的体系结构中,确保了应用程序具有更高的可维护性。最后,由于 JSF 是通过 JCP 开发的一种 Java 标准,因此开发工具提供商完全能够为 JavaServer Faces 提供易于使用而又高效的可视化开发环境。

3.1 概述

JSF 定义了一组 UI 组件,以及一组标准的 API。所有 UI 组件都可以直接用在网页里,而且大部分组件几乎都是 HTML form 系列标记的翻版。API 可用来扩充原有的标准组件,也可以开发全新的组件。连接在组件上的验证器不仅能够检验用户输入的数据,而且能自动将输入数据传递给应用对象。每当用户做点击链接或按下按钮等操作时便会触发事件处理器,而事件处理器可以改变其他组件的状态,或是运行某段后台程序。借助一个可插入的导航处理器,事件处理器可以控制接下来要显示哪一个网页。

将 JSF 技术用于 JSP 页面中,可以大大减少用户界面中的程序代码。JSF UI 组件声明它们可能发生什么事件(诸如数值改变、按下按钮等),并且可以配置处理该事件的事件监听器。同一种 JSF UI 组件,只要搭配不同的 **renderer**,便可呈现不同的外观(按钮或链接,或以其他标记语言表示)。此外,JSF 也定义了辅助性的工具对象,包括能够验证输入数据有效性的验证器 (**validator**),以及能够转换数据类型的转换器 (**converter**)。因此,使用 JSF 技术,开发者可使用熟悉的 GUI 框架来开发 Web 应用系统,并且开发出来的系统又易于维护。

JSF 页面的生命周期有点像 JSP 页面:客户端对页面发出一个 HTTP 请求,服务器将页面翻译成 HTML 进行响应。但是 JSF 生命周期又不同于 JSP 生命周期。为了支持复杂的 UI 组件模型,JSF 生命周期划分为多个阶段。这种模型需要按照一定的顺序对组件数据进行转换和验证、处理组件事件并将组件数据保存到 Bean 中。

JSF 页面是用 UI 组件树表示的,称为视图。在生命周期中,当用户关注的状态从前一页面提交后,JSF 实现必须构建视图。当客户端提交一个页面的时候,JSF 实现执行多个任务。例如,对视图组件中输入的数据进行验证,并将输入的数据转换成服务器端指定的类型。

JSF 将所有这些任务作为生命周期中的一系列步骤来执行。在生命周期中执行哪一步,取决于请求是否来自 JSF 应用以及响应是否在 JSF 的生命周期的呈现阶段生成。

图 20.2 演示了 JSF 请求-处理生命周期的步骤。

请求处理生命周期处理两种请求: **initial** 请求和 **postback** 请求。当一个用户对页面发出 **initial** 请求的时候,是第一次请求该页面。当用户执行 **postback** 请求的时候,提交的表单所在页面曾经由于 **initial** 请求而被装载到浏览器。当生命周期处理 **initial** 请求的时候,仅仅执行重建视图和呈现响应阶段,因为用户没有输入或动作需要处理。相反,当生命周期处理 **postback** 请求的时候,会执行所有阶段。

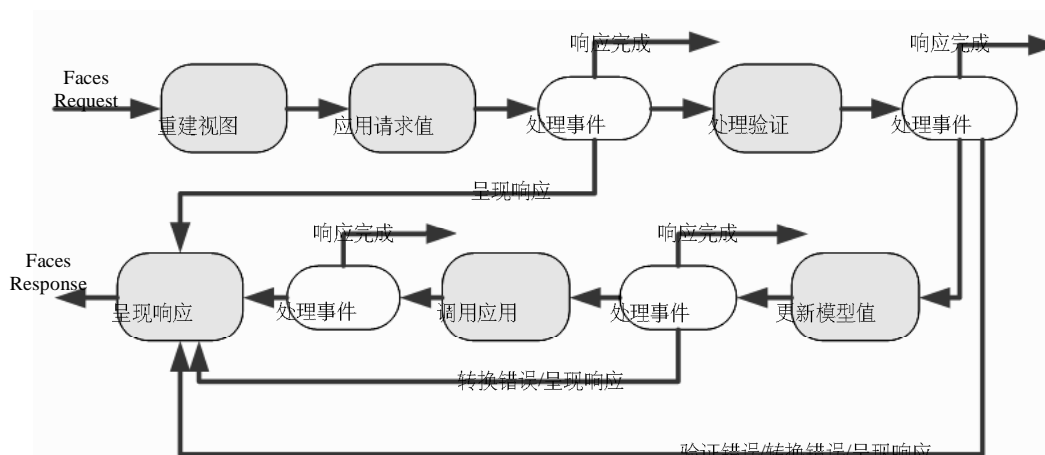


图 3.1 JSF 标准请求-处理生命周期

JSF 的标准请求处理生命周期执行以下六个阶段：

1. 重建视图阶段（restore view）
2. 应用请求值阶段（apply request values）
3. 处理验证阶段（process validations）
4. 更新模型值阶段（update model values）
5. 调用应用阶段（invoke application）
6. 呈现响应阶段（render response）

下面以登录功能为例分别介绍这六个阶段。例子的界面如图 21.3 所示。

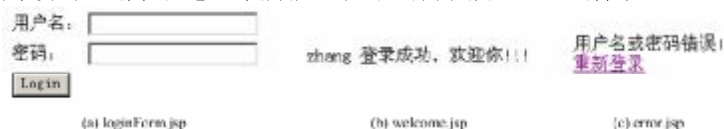


图 3.2 运行效果

重建视图阶段

当对一个 JSF 页面发出请求的时候，例如点击超链接或者按钮的时候，JSF 实现开始重建视图阶段。

在这个阶段，JSF 实现构建页面视图组件，将事件处理器和验证器绑定到视图组件上，并在 `FacesContext` 实例中保存视图。这个实例包含处理一个请求需要的所有信息。应用中所有的组件标签、事件处理器、转换器和验证器都可以访问 `FacesContext` 实例。

如果对页面的请求是一个初始请求，则 JSF 实现在这个阶段将创建一个空视图，生命周期直接跳到呈现响应阶段。当页面在再次返回（`postback`）的时候会进行处理，空视图会被填充。

如果对页面的请求是一个 `postback`，则对应这个页面的视图已经存在。在这个阶段，JSF 实现利用保存在客户端或者服务器端的状态信息重建视图。

login 例子中 `loginForm.jsp` 页面的视图树根是 `UIView` 组件，`form` 是它的孩子，其余 JSF UI 组件是 `form` 的孩子。

应用请求值阶段

在组件树重建以后，树中的每个组件都使用 `decode` 方法从请求参数中提取新值。然后，值被存储在本地组件中。如果值转换失败，则会产生与组件相关的错误消息并放入消息队列中。这些消息在呈现响应阶段显示出来，同时显示的还有处理验证阶段生成的所有验证信息。

在 loginForm.jsp 页面上的 username 组件中，值是由用户输入到文本域中的。因为绑定到组件上的对象是 LoginBean 的成员 username，而它是 String 类型，所以不需要执行转换。但是如果它在辅助 Bean 中对应的成员是 int 类型，则 JSF 实现会将 String 转换为 Integer。

如果在这个阶段有事件放入队列，JSF 实现将把这些事件广播给所有感兴趣的监听器。

如果页面上的某些组件将 immediate 属性设置为 true，那么验证、转换和与组件相关的事件将在这个阶段处理。在这种情况下，如果应用需要重定向到一个不同的 Web 应用资源或者生成一个不包含任何 JSF 组件的响应，它可以调用 FacesContext.responseComplete。

在这个阶段结束的时候，组件的值被更新，消息和事件放入队列。

处理验证阶段

在这个阶段，JSF 实现处理组件上注册的所有验证器。它检查指定验证规则的组件属性，并将这些规则与为组件存储的本地值进行比较。

如果本地值无效，JSF 实现将在 FacesContext 实例上添加一个错误消息，生命周期直接跳到呈现响应阶段，在页面再次呈现的时候就会显示错误消息。如果有来自应用请求值阶段的转换错误，这些错误消息也会显示出来。

如果事件在这个阶段放入队列，则 JSF 实现也会把这些事件广播给所有感兴趣的监听器。

在 loginForm.jsp 页面中，JSF 实现处理两个验证器。一个是 username 和 password 上注册的 required 验证器，要求这两个输入域不能为空。如果任何一个为空，则显示错误信息。另一个是 password 上注册的检查长度的验证器，它检查用户输入到文本域中的数据长度是否大于等于 6 个字符。如果数据无效或者在应用请求值阶段出现转换错误，处理过程跳到呈现响应阶段。在这个阶段，loginForm.jsp 页面再次呈现，并将验证和转换错误信息显示在与 message 标签相关的组件中（参见图 20.4 和图 20.5）。

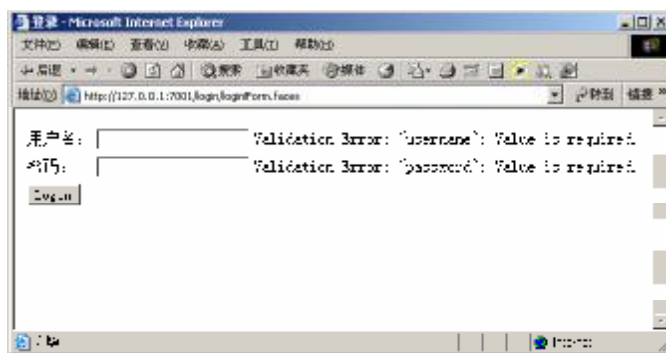


图 3.3 如果用户名或密码输入为空，则显示错误信息

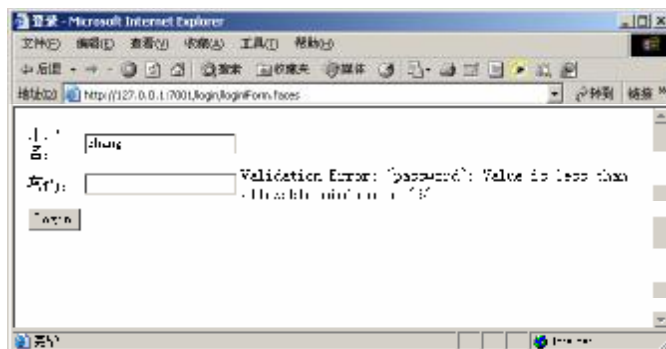


图 3.4 如果密码长度不是 6 位，则显示错误信息

更新模型值阶段

在 JSF 实现确定数据有效之后，它遍历组件树并将对应的服务器端对象属性设置为组件的本地值。JSF 实现只更新由输入组件的 `value` 属性指向的 `Bean` 属性。如果本地数据不能转换为 `Bean` 属性指定的类型，则生命周期会直接跳到呈现响应阶段，并在页面呈现的时候显示错误信息。这与发生验证错误的情况类似。

如果事件在这个阶段放入队列，JSF 实现也会把这些事件广播给所有感兴趣的监听器。

在这个阶段，`LoginBean` 的 `username` 属性被设置为 `username` 组件的本地值，`password` 属性被设置为 `password` 组件的本地值。

调用应用阶段

在这个阶段，JSF 实现处理任何应用级的事件，例如提交一个表单或者链接到其他页面。

如果被处理的视图是根据来自前面请求的状态信息重建的，而且如果组件触发一个事件，则这些事件将被广播到所有感兴趣的监听器。

`login` 例子中的 `loginForm.jsp` 页面有一个与 `UICommand` 组件相关的应用级的事件。当处理这个事件的时候，一个默认的 `ActionListener` 实现从组件的 `action` 属性中检索结果 `success`。监听器把结果传给默认的 `NavigationHandler`。将结果与应用的应用配置资源文件中定义的相应导航规则相匹配，从而决定接下来要显示哪一个页面。如果用户名和密码正确，则转向 `welcome.jsp`，否则转向 `error.jsp`。接着 JSF 实现将响应视图设置为这个新页面。最后，JSF 实现将控制转交给呈现响应阶段。

呈现响应阶段

在这个阶段，如果应用使用 JSP 页面，那么 JSF 实现将呈现页面的权限委派给 JSP 容器。如果这是一个初始请求，当 JSP 容器执行页面的时候，页面上表示的组件将添加到组件树中。如果这不是一个初始请求，则组件已经添加到树中，因此不需要再次添加。在任何一种情况下，当 JSP 容器遍历页面中的标签时，组件都将呈现其自身。

如果请求是一个 `postback`，而且在应用请求值阶段或者更新模型值阶段已经遇到错误，那么在这个阶段将呈现原来的页面。如果页面包含 `message` 或 `messages` 标签，那么队列中的所有错误信息都将显示在页面上。

在视图中的内容呈现以后，响应的状态被保存下来，以便后续的请求能够访问它，并且在重建视图阶段同样能访问它。在 `login` 例子中，如果对 `loginForm.jsp` 页面的请求是初始请求，那么表示这个页面的视图被构建，并且在重建视图阶段保存在 `FacesContext` 中，然后在这个阶段呈现。如果对页面的请求是 `postback`（例如当用户输入一些无效的数据并点击 `Submit` 时），树在重建视图阶段重建，并继续请求处理生命周期的所有阶段。

3.2 开发人员的主要工作

与使用 `Strut 2` 进行开发要完成的工作基本相同，只是在细节上不同。首先也是搭建环境，然后开始具体的开发过程。在具体开发过程中，包括 3 个方面：

- 使用 JSF 提供的组件进行界面的编写，包括为组件添加事件处理、添加验证等；
- 编写辅助 `Bean`，完成验证、验证、事件处理等功能；
- 配置导航和辅助 `Bean`，导航也就是页面的跳转关系。

环境搭建

环境搭建的基本过程如下：

- 创建工程，与创建其他 Web 应用的过程相同；
- 添加类库支持，与 Struts 不同，JSF 本身是一种标准，有很多实现。因为在 Java EE5 中把 JSF 技术作为规范的一部分，这样实现 Java EE 5 的应用服务器都会提供 JSF 实现。这些类库通常包括 jsf-api.jar 和 jsf-impl.jar，把这些类库放到 Web 工程的 WEB-INF/lib 下面；
- 配置 web.xml 文件，在 JSF 中是通过 FacesServlet 中心控制器来完成整个应用的控制的，所有请求都提交给中心控制器，需要在 web.xml 中进行配置，下面是一个配置的例子；

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    version="2.5">
    <context-param>
        <param-name>javax.faces.CONFIG_FILES</param-name>
        <param-value>/WEB-INF/faces-config.xml</param-value>
    </context-param>
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.faces</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

- 创建 JSF 的配置文件，在 web.xml 中的 <context-param> 中声明了一个 javax.faces.CONFIG_FILES 变量，表示 JSF 的配置文件，按照上面的配置，该配置文件位于 WEB-INF 下，名字为 faces-config.xml。初始的配置文件的内容如下：

```
<?xml version='1.0' encoding='UTF-8'?>

<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
    version="1.2">
</faces-config>
```

这样环境就搭建好了，接下来就可以进行应用的开发了。

创建页面

JSF 页面的开发主要是在页面中加入各种组件，1 个典型的 JSF 界面的代码通常是这样的：

```
<%@ taglib uri = "http://java.sun.com/jsf/html" prefix = "h" %>
<%@ taglib uri = "http://java.sun.com/jsf/core" prefix = "f" %>
<f:view>
<h:form>
其他 JSF 标签或核心标签
包含一个或多个用于提交表单的按钮或超链接
</h:form>
</f:view>
```

JSF 的界面标签主要包括两个：html 组件标签和核心标签。html 组件标签如表 3.2 所示，核心标签如表 3.1 所示，核心标签通常与组件标签一起使用。对于 html 组件标签，在服务器端会有相应的组件，这样可以根据页面在服务器端重建视图。

表 3.1 JSF core 标签库中的标签

标签类型	标签	功能
事件处理标签	actionListener	在父组件上注册一个 action listener
	PhaseListener	在 UIViewRoot 组件上注册一个 PhaseListener 实例
	setPropertyActionListener	注册一个特殊的事件监听器，其唯一目的是在表单提交的时候将数值放入辅助 bean 中
	valueChangeListener	在父组件上注册一个 value-change listener
属性标签	attribute	在父组件上添加可配置的属性
数据转换标签	Converter	在父组件上注册一个任意的转换器
	convertDateTime	在父组件上注册一个 DateTime 转换器
	convertNumber	在父组件上注册一个 Number 转换器
Facet 标签	facet	表示一个嵌入的组件，它与自身的封装标签有特殊的关系
本地化标签	loadBundle	指定一个 ResourceBundle，类型为 Map
参数替换标签	param	在 MessageFormat 实例中替换参数，并在 URL 中添加查询字符串名称-值对
表示列表项的标签	selectItem	表示 UISelectOne 或者 UISelectMany 组件中的列表项中的一项
	selectItems	表示 UISelectOne 或者 UISelectMany 组件中的一组项目
容器标签	subview	包含页面中的所有 JSF 标签，该页面包含在另一个包含 JSF 标签的 JSP 页面中
验证标签	validateDoubleRange	在组件上注册一个 DoubleRange Validator
	validateLength	在组件上注册一个 Length Validator
	validateLongRange	在组件上注册一个 LongRangeValidator
	validator	在组件上注册一个自定义验证器
输出标签	verbatim	生成一个 UIOutput 组件，该组件从 verbatim 标签体中获取内容
form 标签的容器	view	包括页面上的所有 JSF 标签

表 3.2 JSF 标准 HTML 标签库中定义的组件标签及其呈现方式

标签	功能	呈现的 HTML	外观
column	表示在 UIData 组件中的一列数据	在 HTML table 中的一列数据	table 中的一

			列
commandButton	将一个表单提交给应用	HTML <input type = <i>type</i> >元素, 其中 <i>type</i> 值可以是 submit, reset 或者 image	按钮
commandLink	链接到另一个页面或者同一页面的某个位置	HTML <a href>元素	超链接
dataTable	用来封装一组数据	HTML <table>元素	可以动态修改的表格
form	表示一个输入表单, 表单中接收数据的输入标签将随同表单一起被提交	HTML <form>元素	无显示
graphicImage	显示一幅图像	HTML 元素	一幅图像
inputHidden	允许页面制作人员在页面中包含一个隐藏变量	HTML<input type = hidden>元素	无显示
inputSecret	允许用户输入一个字符串而在文本域中并不显示实际的字符串	HTML <input type = password> 元素	一个文本域, 显示一行字符但不是实际输入的字符串
inputText	允许用户输入一个字符串	HTML <input type = text>元素	一个文本域
inputTextarea	允许用户输入一个多行字符串	HTML <textarea>元素	一个多行文本域
message	显示一个本地化的消息	HTML 元素, 如果使用了样式	一个文本字符串
messages	显示多个本地化的消息	一组 HTML 元素, 如果使用了样式	一个文本字符串
outputFormat	显示一个本地化的消息	纯文本	纯文本
outputLabel	作为一个 label 为一个特定的输入域显示一个嵌入的组件	HTML <label>元素	纯文本
outputLink	链接到另一个页面或者同一页面的某个位置, 但不产生动作事件	HTML <a>元素	超链接
outputText	显示一行文本	纯文本	纯文本
panelGrid	显示一个 table	带有<tr>和<td>的 HTML <table>元素	表格
panelGroup	将多个组件分为同一个父亲下的一组		表格中的行
selectBooleanCheckbox	允许用户修改 Boolean 选择的值	HTML<input type = checkbox>元素	复选框
selectItem	表示 UISelectOne 组件中列表项中的一项	HTML<option>元素	无显示
selectItems	表示 UISelectOne 组件中的多项	一组 HTML<option>元素	无显示
selectManyCheckbox	显示一组复选框, 用户可以从中选择多个值	checkbox 类型的一组 HTML<input>元素	一组复选框
selectManyListbox	允许用户从一组列表中选择多项, 所有选择项立刻显示	HTML<select>元素	列表框
selectManyMenu	允许用户从下拉菜单的一组项目中选择多项	HTML<select>元素	下拉式组合框
selectOneListbox	允许用户从一组列表中选择一项, 所有选择项立刻显示	HTML<select>元素	列表框
selectOneMenu	允许用户从下拉菜单的一组项目中选择一项	HTML<select>元素	下拉式组合框
selectOneRadio	允许用户从一组项目中选择一项	HTML<input type = radio>元素	一组单选按钮

下面是一个使用 JSF 组件创建登录界面的例子:

```
<f:view>
  <h:form>
    <h:panelGrid columns = "3">
      <h:outputLabel for = "username" value = "用户名: " />
```

```

        <h:inputText id = "username" value = "#{loginBean.username}" required="true" />
        <h:message for = "username" />
        <h:outputLabel for = "password" value = "密码： " />
        <h:inputSecret id = "password" value="#{loginBean.password}" required="true" >
            <f:validateLength minimum = "6" />
        </h:inputSecret>
        <h:message for = "password" />
    </h:panelGrid>
    <h:panelGrid>
        <h:panelGroup>
            <h:commandButton value = "Login" action = "#{loginBean.login}" />
        </h:panelGroup>
    </h:panelGrid>
</h:form>
</f:view>

```

在表单元素中可以使用统一表达式语言“#{ }”来建立界面组件与模型中属性的关联关系。例如，下面就是建立输入框与模型属性关联的例子，这样输入框 `username` 的值就会与 `loginBean` 的 `username` 属性关联起来。

```

<h:inputText id = "username" value = "#{loginBean.username}" required="true" />

```

可以为组件添加验证，可以使用标准验证器，例如上面代码中的 `required="true"`，以及 `<f:validateLength minimum = "6" />`，也可以使用自定义验证器。系统提供的验证器如表 218 所示：

表 20.8 Validator 类

验证器类	标签	功能
DoubleRangeValidator	validateDoubleRange	检查组件的本地值是否在特定的范围内。这个值必须是浮点数或者可以转换成浮点数
LengthValidator	validateLength	检查组件的本地值的长度是否在特定范围内。这个值必须是 <code>java.lang.String</code>
LongRangeValidator	validateLongRange	检查组件的本地值的大小是否在特定范围内。这个值必须是数值类型或者可以转换成 <code>long</code> 的 <code>String</code>

可以为组件添加转换器，如果组件与模型的属性关联，则系统自动完成转换。如果使用转换器，则可以使用标准的转换器，也可以使用自定义转换器，下面的例子使用的是标准转换器。标准转换器实现位于 `javax.faces.convert` 包中，包括：

- `BigDecimalConverter`
- `BigIntegerConverter`
- `BooleanConverter`
- `ByteConverter`
- `CharacterConverter`
- `DateTimeConverter`
- `DoubleConverter`
- `FloatConverter`
- `IntegerConverter`
- `LongConverter`

- NumberConverter
- ShortConverter

```
<h:inputText value = "#{LoginBean.Age}" />
<f:converter converterId = "Integer" />
</h:inputText>
```

还可以为组件添加事件监听器来监听组件的事件。JSF 技术支持三种类型的事件：value-change 事件、action 事件和 data-model 事件。

编写 JavaBean

在 JSP 技术中，JavaBean 完成的主要任务包括：

- 接收用户通过界面输入的信息，在 JavaBean 中提供属性以及 set 方法；
- 提供界面要显示的动态信息，在 JavaBean 中提供属性及 get 方法；
- 处理用户的事件，可以做为 JavaBean 的方法，也可以编写单独的事件监听类；
- 完成自定义验证器的验证代码，提供验证的方法；
- 完成自定义转换器的转换代码，提供转换的方法。

配置

页面之间的跳转关系以及编写好的由容器管理的 JavaBean 需要在 faces-config.xml 中配置。跳转关系包括从哪个页面跳转到哪个页面以及跳转的条件。下面是一个跳转的例子：

```
<navigation-rule>
  <from-view-id>/loginForm.jsp</from-view-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/welcome.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>failure</from-outcome>
    <to-view-id>/error.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

<navigation-rule>表示一个跳转的配置，<from-view-id>表示输入界面。<navigation-case>表示跳转的情况，<from-outcome>表示 action 事件处理一个可能的结果，action 事件处理的结果如果和<from-outcome>中的字符串匹配，则将跳转到<to-view-id>所指向的输出界面。

配置 JavaBean 的主要目的是让容器来管理这些 JavaBean 对象，包括 JavaBean 对象的创建和方法的调用等。下面是一个配置的例子。

```
<managed-bean>
  <managed-bean-name>loginBean</managed-bean-name>
  <managed-bean-class>backing.LoginBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>uv</property-name>
    <value>#{userValidator}</value>
  </managed-property>
```

```
</managed-bean>
```

<managed-bean>表示一个被管理的 JavaBean, <managed-bean-name>表示 JavaBean 对象名字, <managed-bean-class>表示被管理的 JavaBean 的 Bean 类, <managed-bean-scope>表示 JavaBean 的作用范围, <managed-property>表示要处理的 JavaBean 的属性, <property-name>表示属性的名字, <value>表示对这个属性赋什么值, 这里应用了另外一个被管理的 JavaBean。

3.3 实例

实例包括的主要文件如下:

- loginForm.jsp, 用于用户名和口令的界面
- welcome.jsp, 登录成功的界面
- error.jsp, 登录失败的界面
- backing.LoginBean, 表示登录信息
- validator.UserValidator, 对用户名和口令进行验证
- web.xml, web 应用的配置文件
- faces-config.xml, JSF 应用的配置文件, 配置两个 JavaBean 文件和页面之间的跳转关系。

loginForm.jsp

```
<%@ page language = "java" pageEncoding = "gb2312"%>
<%@ taglib uri = "http://java.sun.com/jsf/html" prefix = "h" %>
<%@ taglib uri = "http://java.sun.com/jsf/core" prefix = "f" %>

<html>
<head>
    <title>登录</title>
</head>

<body>
<f:view>
    <h:form>
        <h:panelGrid columns = "3">
            <h:outputLabel for = "username" value = "用户名: " />
            <h:inputText id = "username" value = "#{loginBean.username}" required = "true"
        />

            <h:message for = "username" />
            <h:outputLabel for = "password" value = "密码: " />
            <h:inputSecret id = "password" value = "#{loginBean.password}" required = "true"
        />

            <f:validateLength minimum = "6" />
            <h:inputSecret>
            <h:message for = "password" />
        </h:panelGrid>
        <h:panelGrid>
            <h:panelGroup>
```



```
                <h:commandButton value = "Login" action = "#{loginBean.login}" />
            </h:panelGroup>
        </h:panelGrid>
    </h:form>
</f:view>
</body>
</html>
```

welcome.jsp

```
<%@ page language = "java" pageEncoding = "gb2312"%>
<%@ taglib uri = "http://java.sun.com/jsf/html" prefix = "h" %>
<%@ taglib uri = "http://java.sun.com/jsf/core" prefix = "f" %>
<html>
    <head>
        <title>登录成功!!!</title>
    </head>
    <body>
        <f:view>
            <h:outputText value = "#{loginBean.username}" />
            登录成功，欢迎你!!!<br>
        </f:view>
    </body>
</html>
```

error.jsp

```
<%@ page language = "java" pageEncoding = "gb2312"%>
<%@ taglib uri = "http://java.sun.com/jsf/html" prefix = "h" %>
<%@ taglib uri = "http://java.sun.com/jsf/core" prefix = "f" %>
<html>
    <head>
        <title>Error!!!</title>
    </head>
    <body>
        <f:view>
            用户名或密码错误! <BR>
            <h:outputLink id = "relogin" value = "loginForm.faces" rendered = "true" >
                <h:outputLabel for = "relogin" value = "重新登录" />
            </h:outputLink>
        </f:view>
    </body>
</html>
```

LoginBean.java

```
package backing;

import validator.UserValidator;

public class LoginBean {
    private String username;
    private String password;
    private UserValidator uv;

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public UserValidator getUv() {
        return uv;
    }

    public void setUv(UserValidator uv) {
        this.uv = uv;
    }

    //用户登录过程
    public String login() {
        return uv.validator(username, password);
    }
}
```

UserValidator.java

```
package validator;
```

```
public class UserValidator {
// 这是验证用户名和密码的过程
public String validator(String username, String password) {
    if ((username == null) || (username.length() < 1))
        return "failure";
    if ((password == null) || (password.length() < 1))
        return "failure";
    if ((username.equals("zhang")) && (password.equals("123456")))
        return "success";
    else
        return "failure";
}
}
```

faces-config.xml

```
<?xml version='1.0' encoding='UTF-8'?>

<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig\_1\_2.xsd"
    version="1.2">

    <managed-bean>
        <managed-bean-name>userValidator</managed-bean-name>
        <managed-bean-class>validator.UserValidator</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>
    <managed-bean>
        <managed-bean-name>loginBean</managed-bean-name>
        <managed-bean-class>backing.LoginBean</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
        <managed-property>
            <property-name>uv</property-name>
            <value>#{userValidator}</value>
        </managed-property>
    </managed-bean>

    <navigation-rule>
        <from-view-id>/loginForm.jsp</from-view-id>
        <navigation-case>
            <from-outcome>success</from-outcome>
            <to-view-id>/welcome.jsp</to-view-id>
        </navigation-case>
    </navigation-rule>
</faces-config>
```

```
        </navigation-case>
        <navigation-case>
            <from-outcome>failure</from-outcome>
            <to-view-id>/error.jsp</to-view-id>
        </navigation-case>
    </navigation-rule>
</faces-config>
```

第 4 章 简单自定义 Web 层框架

为了让读者了解 Web 层框架的实现机制，本章实现了一个简单的 Web 层框架。使用该框架用户能够完成的功能如下：

- 把用户的请求直接提交给 **JavaBean** 的某个方法处理，用户需要在界面的请求中写出业务类的名字，以及业务方法的名字，如果不写方法名字，默认的调用 **JavaBean** 的 **action** 方法。另外需要在配置文件中设置默认的包名，如果不在默认包中，可以在请求中加入包的信息。这个例子中，类的完整名字为：**test.action.UserAction**。这时候需要在 **package.properties** 中定义：**base=test.action**，需要在请求中写 **/action/userAction**，其中 **action** 是固定的，**userAction** 表示类，把类名的首字母写成小写了。在类中定义了默认的 **action** 业务方法，所以在请求中不用写方法名。也可以指定方法名，采用下面的方式：**userAction!methodName**，其中 **methodName** 就是方法名。如果包名不仅仅是 **test.action**，例如是 **test.action.m1**，则需要在请求中写出 **m1** 部分，这时候请求路径中需要写成 **/action/m1.userAction**。

界面中请求的代码：

```
<form name="form1" action="<%=base%>/action/userAction" method="post" >
```

配置文件 **package.properties** 中信息：

```
base=test.action
```

类头部分：

```
package test.action;
```

```
public class UserAction {
```

- 把界面表单输入的信息传递给业务处理的 **JavaBean**，用户需要做的是在 **JavaBean** 中提供与表单元素名字相同的属性，并提供 **set** 方法。例如下面的代码：

界面代码：

```
用户名: <input type="text" name="username">
```

JavaBean 中代码：

```
private String username;
```

```
public void setUsername(String username) {  
    this.username = username;  
}
```

- 把用户业务 **JavaBean** 中的信息保存到 **request** 中，这样在界面可以通过 **EL** 显示。用户需要在 **JavaBean** 中提供 **get** 方法。下面的代码展示了用法。

JavaBean 中代码

```
public String getUsername() {  
    return username;  
}
```

界面中代码：

用户名: \${username}

- 为 JavaBean 提供 request 中的请求信息，用户需要在 JavaBean 中提供 setRequest 方法，例如下面的代码：

```
private HttpServletRequest request;

public void setRequest(HttpServletRequest request){
    this.request = request;
}
```

- 可以访问控制器中的其它信息，用户需要在 JavaBean 中提供 setServlet 方法，例如下面的代码：

```
private HttpServlet servlet;

public void setServlet(HttpServlet servlet){
    this.servlet = servlet;
}
```

- 系统根据用户在业务方法中返回的文件名进行跳转，例如要跳转到 success.jsp，则返回 success.jsp，如果想重定向到某个文件，在文件名前面加上“redirect:”，这样控制器就会使用 response.sendRedirect 完成重定向。下面的代码就是 UserAction 中的 action 方法。

```
public String action(){
    if(username!=null    &&    username.equals("zhangsan")    &&    password!=null    &&
password.equals("lisi")){
        return "success.jsp";
    }else{
        return "error.jsp";
    }
}
```

框架完成的主要功能包括：

- 解析请求路径，解析请求中包含的类的信息、包的信息和方法信息；
- 生成业务 JavaBean 的实例；
- 为 JavaBean 传值；
- 为 JavaBean 传递 request 以及当前 Servlet 对象的引用；
- 调用 JavaBean 的业务方法；
- 把 JavaBean 中的某些值保存到 request 中；
- 完成响应。

下面分别介绍。

4.1 解析路径

通过路径解析，能够从路径中获取类的名字、包的名字和方法的名字。

```
/*
 * 获取请求的目标文件
 */
```

```
private String getServletPath() {

    // 获取应用名字的长度，应用名字通过 getContextPath 得到，格式为：/test
    int length = request.getContextPath().length();

    // 获取请求资源的 URI
    String uri = request.getRequestURI();

    // 返回应用之外的 URI 部分,+8 是为了去除“/action/”
    return uri.substring(length + 8);
}
/*
 * 查找请求中的方法的名字，如果请求中没有方法，则使用默认的方法 action
 */
private String findMethod(String servletPath) {
    int index = servletPath.lastIndexOf("!");
    if (index == -1)
        return "action";
    else
        return servletPath.substring(index + 1);
}
/*
 * 查找 action 的名字
 */
private String findClass(String servletPath) {

    // 取出 action 和 method
    String temp;
    int index = servletPath.lastIndexOf(".");
    if (index > -1) {
        temp = servletPath.substring(index + 1);
    } else {
        temp = servletPath;
    }

    // 取出 action
    index = temp.lastIndexOf("!");
    if (index > -1) {
        temp = temp.substring(0, index);
    }

    // 把首字母换成大写的
    char first = temp.charAt(0);
```

```

        first = Character.toUpperCase(first);

        return first + temp.substring(1);
    }
    /*
     * 得到 action 的包信息
     */
    private String findPackage(String servletPath) {
        // 要得到的包名
        String packageName = "";

        // 包含包信息的属性文件
        ResourceBundle packages = ResourceBundle.getBundle("package");

        // 包的根路径
        String basePackage = packages.getString("base");

        // 查找请求路径中是否有包信息
        int index = servletPath.lastIndexOf(".");

        if (index > 0) {
            // 请求信息中的包信息
            String packageInfo = servletPath.substring(0, index);
            packageName = "." + packages.getString(packageInfo);
        }
        return basePackage + packageName;
    }
}

```

4.2 创建 JavaBean 的实例

通过上面的解析能够得到 **JavaBean** 的类名，然后通过反射机制创建 **JavaBean** 的实例。下面的代码用于实例化 **JavaBean**。

```
action = Class.forName(actionClass).newInstance();
```

4.3 从视图向 JavaBean 传值

从视图向 **JavaBean** 传值，首先通过 **request** 的 **getParameterNames** 得到所有请求信息的名字，然后根据这个名字使用 **request** 的 **getParameter** 或者 **getParameterValues** 获取值，然后调用 **JavaBean** 的方法对属性赋值，须需要使用反射机制。下面的代码完成从表单元素中获取信息并对属性赋值。代码中的 **BeanUtils** 是 **Apache** 提供的工具类。

```

/*
 * 使用 request 中的请求值对 action 的属性赋值
 */
public void setProperties(Object actionInstance)

```



```
        throws IllegalAccessException, InvocationTargetException {

    // 获取请求信息
    Map parameters = request.getParameterMap();

    // 赋值
    BeanUtils.populate(actionInstance, parameters);

}
```

4.4 为 JavaBean 传递其它信息

为了让 JavaBean 的业务方法能够访问 Servlet 中的信息, 需要把 request 以及当前 Servlet 的引用传递到 JavaBean 中, JavaBean 需要创建 setRequest 和 setServlet 方法来接收这些信息, 如果不需要访问这些对象, 则可以不用实现这些方法。下面的代码完成了这个过程。

```
/*
 * 把 Servlet、request 传递给 action, 这样 action 可以获取更多的信息
 */
private void processServletInfo(Object action) {
    // 处理 setServlet 方法
    try {
        // 得到方法名字
        Method method = action.getClass().getMethod("setServlet",
            HttpServletRequest.class);

        // 赋值
        method.invoke(action, this);
    } catch (Exception e) { }

    // 处理 setRequest 方法
    try {
        // 得到方法名字
        Method method = action.getClass().getMethod("setRequest",
            HttpServletRequestRequest.class);

        // 赋值
        method.invoke(action, request);
    } catch (Exception e) { }

}
```

4.5 调用 JavaBean 方法

调用 JavaBean 的方法, 需要使用 JavaBean 实例以及方法的名字, 然后使用反射机制进行调用。

下面的代码是调用业务方法的代码。

```
/*
 * 调用业务方法
 * @param action, 业务 Bean
 * @param methodName, 业务方法的名字
 */
public String invokeBusiness(Object action, String methodName) {
    if (methodName == null) {
        methodName = "action";
    }
    try {
        // 得到业务方法
        Method method = action.getClass().getMethod(methodName);

        // 调用业务方法
        String result = (String) method.invoke(action);

        // 返回执行的结果, jsp 文件或者其他的 action
        return result;
    } catch (Exception e) {
        System.out.println(e.toString());
    }
    return null;
}
```

4.6 从 JavaBean 向视图传值

正常情况下, JavaBean 的执行结果要传到视图层, 通常使用 request 对象保存, 然后在视图层使用 EL 显示。

本框架为了方便用户传值, 会把 JavaBean 的所有 get 方法的返回结果保存到 request, 这样用户要传递什么信息, 只需要编写 get 方法即可。控制器中的处理代码如下:

```
/*
 * 把 action 中的属性存储到 request 中, 这样在界面就可以访问了
 */
public void saveResult(Object action) {
    // 得到 action 的所有方法
    Method methods[] = action.getClass().getMethods();

    /*
     * 遍历所有方法, 根据所有 get 方法来获取所有的返回值
     */
    if (methods != null) {
        // 遍历
```

```
        for (Method me : methods) {
            // 判断是不是 get 方法
            if (me.getName().startsWith("get")) {
                try {

                    // 把属性值保存到 request 中
                    request.setAttribute(propertyName(me.getName()),
                                           me.invoke(action));

                } catch (Exception e) {
                    System.out.println(e.toString());
                }
            }
        }
    }
}
```

4.7 响应

根据业务方法返回的结果，使用 `response.sendRedirect` 方法或者 `RequestDispatcher` 的 `forward` 方法对用户进行响应。下面的代码完成对用户的响应。

```
/*
 * 完成跳转，如果要转向的目标是以"redirect:"开始的，
 * 则使用 response.sendRedirect 重定向
 * 否则，使用 RequestDispatcher 的 forward 方法转发请求
 */
private void doForward(HttpServletRequest request,
                        HttpServletResponse response, String forward)
    throws ServletException, IOException {
    int redirect = forward.indexOf("redirect:");

    if (redirect > -1)
        response.sendRedirect("/" + forward.substring(9));
    else {
        RequestDispatcher rd = request.getRequestDispatcher("/" +
forward);

        rd.forward(request, response);
    }
}
```

4.8 小结

本节介绍的框架仅仅完成了一些功能，与实际应用还有一段距离。框架在如下几个方面可以

改进：

- 增加验证框架；
- 增加类型转换；
- 定义方便界面使用的标签；
- 国际化支持；
- 异常和日志的处理。

第 5 章 Java 持久性技术

5.1 持久性的概念

文件

计算机中的文件用于存储各种各样的信息，要存储文本信息可以使用 DOC 文件或者 TXT 文件等，要存储图片可以使用 JPG 文件或者 GIF 文件等，要存储视频文件可以使用 DAT 文件等。使用文件存储信息的好处在于文件可以长期保存，即使关闭电脑，信息也不会丢失。

应用程序

在使用计算机的时候，会有很多程序在运行。而程序在运行的时候会生成很多信息，这些信息是程序在运行过程中所需要的。例如，两个人在玩网络游戏的时候，需要把一个人的信息传递给另一个人，同样需要把对方的信息传递回来，程序会记录双方在玩游戏过程中的数据。但是当游戏结束的时候，这些信息就没有了，除非把它们存储在文件中。

持久性

如果不想让程序运行过程中的信息丢失，就必须想办法保存这些信息。这些信息可以存储在文件中，也可以存储在数据库中（实际上，数据库也是文件，只是封装了对文件的操作，便于用户对文件进行管理）。

把程序中的状态信息进行保存供以后使用，称为持久化。

把程序中的信息存储到文件中的方式通常适用于信息量比较少的情况，可以直接通过文件操作来完成。如果存储在 XML 文件中，也可以通过对 XML 文件进行操作的相关接口来实现。

如果信息量比较大，通常使用数据库的方式来存储信息，并且可以使用数据库管理系统强大的管理功能。通常所说的持久性主要指的是把信息存储到数据库这种方式。

5.2 持久性研究的主要内容

从大的方面来讲，持久性主要研究如何把应用程序中的信息写到数据库中，并且如何把数据库中的信息导入到应用程序中。通常涉及以下的几个方面。

数据库信息配置

所有的持久性实现方式都必须明确 Java 应用程序和什么样的数据库进行映射。访问数据库通常都是采用 JDBC 的方式。所以在配置数据库信息的时候，通常需要提供的配置信息包括：

- 数据库的位置，包括数据库所在的服务器以及 DBMS 所使用的端口。
- 数据库的名字，通常在一个 DBMS 系统中会有多个数据库，因而需要指定当前应用所需要的数据库。
- 连接数据库所需要的用户名和口令，通常对数据库的访问都需要提供用户名和口令，这样才能保证数据库的安全性。
- 连接数据库所使用的驱动程序，需要知道数据库驱动程序的完整名字。此外，需要先提供 JDBC 驱动程序。如果没有驱动程序，就没有办法完成 O/R（对象/关系）映射。

为了使系统具有可移植性，不会把这些信息写在程序里。如果写在程序中，以后当数据库的信息发生变化的时候，就需要改动程序，因而维护很不方便。所以，通常都把这些信息写在配置文件中。

无论学习什么样的 O/R 映射工具，都需要明确下面的三件事情：

- 准备 JDBC 驱动程序
- 明确这些配置信息
- 明确配置文件的位置及使用方式

对象数据的加载

数据的加载指的是把数据库中的信息加载到应用程序中，通常需要明确数据来自哪个表或者哪几个表，以及把数据加载到哪个对象或者哪几个对象。其中，涉及数据的查询和对象信息的更新。包括的工作是：

- 明确数据来源表的名字，因为数据库中会有很多表。
- 要查询的字段名，因为并不是所有字段都是需要查询的。
- 查询表中的哪条记录，因为表中可能会有很多条记录。
- 把查询的结果赋给对象的属性。

实体数据的更新

实体数据的更新指的是把对象变化后的信息更新到数据库中。主要是对数据库中数据的更新。包括的工作如下：

- 明确所更新的数据库中的表的名字。
- 如何执行更新，也就是使用什么样的方法来完成更新。
- 什么时候进行更新，假设改变了一个对象的多个属性，明确是在每个属性发生变化之后都更新数据库，还是在执行完所有的属性改变之后再更新数据库。

数据的查询

对数据的使用主要是查询，可以根据各种条件进行查询。通常是根据用户的查询条件构造查询语句，然后执行这些查询语句，再对查询的结果进行处理。所以在学习 O/R 映射的时候，需要关注几个方面：

- 如何处理查询条件
- 如何构造查询语句
- 在什么地方以及如何执行查询语句
- 查询的结果如何处理；查询的结果可能是一个也可能是多个，如果是一个就直接修改对象的信息，如果是多个则需要分别处理。

关系的管理

在数据库中可能会存在大量的表，表之间可能会存在各种关系，就是通常所说的实体关系。在应用程序中可能会存在大量的对象，这些对象之间也可能存在各种关系，表现为对象之间的关系。要进行 O/R 映射，必须关心如何实现实体之间的关系与对象之间的关系同步。

批量处理

对数据库的操作经常会涉及多个对象或者多个数据库记录。在研究 O/R 映射的时候，需要关心能否同时删除多个对象，或者同时更新多个对象。包括：

- 如何构造这些批量更新或者批量删除的语句
- 如何执行这些语句

安全问题

任何应用都会有安全性问题，对于持久性来说需要对数据库进行操作。不安全的操作可能会带来灾难性的后果，所以对于持久性的安全问题必须进行处理。

事务问题

与安全问题相同，事务问题对于持久性来说也是非常重要的。对于数据库的操作，事务处理是不能少的，而持久性处理的问题多数与数据库相关，所以事务是必须要加以研究的问题。

5.3 持久性实现的方式

持久性实现的方式很多，可以分为下面几类。

直接通过 JDBC 实现

JDBC 提供了应用程序与数据库之间交互的统一接口，无论访问什么样的数据库都可以采用相同的方式。不同的数据库管理系统提供者都会提供自己的 JDBC 驱动程序，而应用程序可以使用不同的 JDBC 驱动程序与不同的数据库系统进行交互。

使用 JDBC API 来完成持久性非常方便，但是在使用 JDBC 完成持久性的过程中会编写很多重复的代码。例如，不管进行什么样的操作都需要先连接数据库，建立语句对象，在执行完对数据库的操作之后需要关闭这些对象。

通过 JDO 进行操作

JDO 相对于 JDBC 的优势为：

首先，使用 JDO 存储和检索对象状态的代码要比 JDBC 少。

其次，JDO 减少了把对象模型映射到关系模型所需要的技巧。

最后，只有在构造查询的时候才需要考虑对象模型。

使用标准标签库中的 SQL 标签

在 JSP 的标准标签库中提供了一组 SQL 标签，通过这组 SQL 标签可以方便地完成 Web 应用对于数据库的操作，使用起来比较方便。

但是同样也存在其缺点：不够灵活，并且只能在 Java Web 应用中使用。

通过 O/R 映射工具实现

流行的 O/R 映射工具很多，包括 Oracle 的 TopLink，Apache 的 Torque 和 ObjectRelational-Bridge，Hibernate 的 Hibernate。

这些工具中的一些非常成功，而现在 Hibernate 的使用非常广泛。这些工具的好处是不再需要编写与数据库进行交互的代码，只需要编写对实体进行操作的查询语句即可，实体中的数据与数据库中数据的同步由 O/R 映射工具完成，降低了程序员的负担。如果能够掌握，编写程序时的效率会非常高。

但是使用这些工具也有自身的问题，由于需要进行配置，因而对程序员的要求比较高。

EJB 2.X 实体 Bean

EJB 2.X 版本中，提供了实体 Bean，是对数据库访问的封装，并能够完成 O/R 的映射。根据

对数据库访问的代码由谁编写可以把实体 Bean 分为 BMP（Bean-managed Persistence）实体 Bean 和 CMP（Container Managed Persistence）实体 Bean。BMP 实体 Bean 由 Bean 提供者编写对数据库的访问代码，CMP 实体 Bean 由容器完成对数据库的访问，但是需要用户进行配置。

Java 持久性 API

EJB 2.X 实体 Bean 的功能非常强大，但是因为 EJB 2.X 非常复杂，因而不容易使用，也不容易学习，并且是重量级的。与此同时，各种 O/R 映射工具不断出现，并且使用比较简单，性能方面也有一定的优势。这样就造成实体 Bean 在实际应用中使用得比较少。

对 EJB 2.X 进行修改势在必行，经过这几年的发展，在今年（2006 年）5 月推出了 EJB 3 的最终版本。在 EJB 3 中，EJB 的类型只有有状态会话 Bean、无状态会话 Bean 和消息驱动 Bean，不再包含实体 Bean。并且将 O/R 映射独立出来，形成 Java 持久性 API（Java Persistence API），简称 JPA。

虽然 EJB 3 中推出了持久性 API，但是为了能够让以前根据 EJB 2.X 规范编写的应用可以在 EJB 3 容器中运行，EJB 3 规范要求服务器提供者实现的产品必须能够支持根据 EJB 2.X 规范编写的实体 Bean。

本书主要介绍 iBATIS、Hibernate 和 JPA 技术。多个技术中使用的相同的表，表定义语句如下：

```
CREATE TABLE ACCOUNT(  
  ACC_ID INT AUTO_INCREMENT PRIMARY KEY,  
  ACC_FIRST_NAME VARCHAR(20),  
  ACC_LAST_NAME VARCHAR(20),  
  ACC_EMAIL VARCHAR(30)  
);
```


第 6 章 iBATIS 技术

6.1 概述

iBATIS 数据映射(Data Mapper)框架使得在 Java 和 .NET 应用中使用数据库变得简单。iBATIS 使用 XML 描述符把对象与存储过程或者 SQL 语句联系起来。简单是 iBATIS 数据映射框架相对于其他对象关系映射工具最大的优势。

要使用 iBATIS 数据映射框架，只需要有对象、XML 和 SQL 等知识就可以，不需要学习其他陌生的东西。使用 iBATIS 数据映射工具，你可以充分发挥 SQL 和存储过程的功能。

6.2 开发人员的主要任务

使用 iBATIS 数据映射工具完成持久性，开发人员需要完成的工作如下：

- ◆ 搭建环境；
- ◆ 创建 SqlMapConfig.xml 文件；
- ◆ 针对每个要操作的对象，创建模型类；
- ◆ 针对每个模型类，创建一个 XML 文档；
- ◆ 在数据访问层使用 iBATIS 提供的接口完成数据库的操作。

下面以更新功能为例介绍使用 iBATIS 技术开发人员需要完成的工作。

环境搭建

获取 iBATIS 实现类，最新版本可以从 <http://ibatis.apache.org/> 下载。

要访问数据库，需要获取相应的 JDBC 驱动程序类。这里还使用 MySQL 数据库。

把实现类库和驱动程序压缩包放在工程的 WEB-INF 下面的 lib 下面即可。

创建 SqlMapConfig.xml 文件

SqlMapConfig.xml 文件，该文件主要描述要访问的数据库的基本信息以及有哪些具体的描述 SQL 语句的配置文件。

根元素为<sqlMapConfig>，主要包括数据源信息的声明以及其它 SQL 映射文件的声明。

数据源信息的声明使用<transactionManager>元素声明，在其内部包含<dataSource>子元素。需要声明的信息包括驱动程序、URL、数据库用户名和口令。下面的例子展示了如何配置。

```
<transactionManager type="JDBC" commitRequired="false">
  <dataSource type="SIMPLE">
    <property name="JDBC.Driver" value="com.mysql.jdbc.Driver"/>
    <property name="JDBC.ConnectionURL"
      value="jdbc:mysql://127.0.0.1:3306/JSPBOOK"/>
    <property name="JDBC.Username" value="root"/>
    <property name="JDBC.Password" value="root"/>
  </dataSource>
</transactionManager>
```

其他的 SQL 映射文件的声明采用<sqlMap>元素，通常在完成其他的 SQL 映射文件之后来修改 SqlMapConfig.xml 文件，下面的代码展示了如何声明。

```
<sqlMap resource="Account.xml"/>
```

编写模型类

iBATIS 数据映射工具主要的工作就是把查询的结果封装成对象，而从对象中获取需要的信息来执行 SQL 语句，这个过程中需要使用用户提供的模型类。模型类就是普通的 JavaBean，模型类中主要包含属性以及对属性进行操作的方法。属性应该包含要访问的数据库表的所有列。

下面是 Account 类的部分代码：

```
package com.mydomain.domain;

public class Account {

    private int id;
    ... // 省略其他属性

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
    ... // 省略其他方法
}
```

创建 SQL 映射文件

SQL 映射文件采用 XML 的格式，主要用于描述对模型类进行操作的主要 SQL 语句以及数据库表中字段与模型类属性之间的对应关系。

SQL 映射文件的根元素是<sqlMap>元素，主要包括如下子元素：

- <typeAlias>子元素，为类指定 1 个别名，为了便于文档中其他地方引用；
- <resultMap>子元素，描述类的属性与查询语句中的列之间的对应关系；
- <update>子元素，完成更新功能；
- <select>子元素，完成查询功能；
- <insert>子元素，完成添加功能；
- <delete>子元素，完成删除功能。

要完成更新功能，编写<update>子元素，id 属性指出名字，访问的时候据此访问，parameterClass 指出参数的类型（可以使用别名）。在<update>的标签体中编写 update 语句，如果需要使用变量，使用两个“#”引起来，例如“#firstName#”。这个 SQL 语句被执行的时候会使用所提供作为参数的对象的相应属性来替代。例如，要执行这个 SQL 语句，会传递一个 Account 对象作为参数，该对象的 firstName 属性的值将会替换 SQL 语句中的变量“#firstName#”。

下面的代码以更新功能为例展示了 SQL 映射文件的内容。

```
<?xml version="1.0" encoding="UTF-8" ?>
```

```

<!DOCTYPE sqlMap
  PUBLIC "-//ibatis.apache.org/DTD SQL Map 2.0//EN"
  "http://ibatis.apache.org/dtd/sql-map-2.dtd">

<sqlMap namespace="Account">

  <!-- 使用类别名可以避免每次都输入完整的类名。 -->
  <typeAlias alias="Account" type="com.mydomain.domain.Account"/>

  <!-- 更新实例，使用 Account 作为参数类。 -->
  <update id="updateAccount" parameterClass="Account">
    update ACCOUNT set
      ACC_FIRST_NAME = #firstName#,
      ACC_LAST_NAME = #lastName#,
      ACC_EMAIL = #emailAddress#
    where
      ACC_ID = #id#
  </update>
  <!--省略了其他的 sql 映射 -->
</sqlMap>

```

注意：编写好的 SQL 映射文件需要在 SqlMapperConfig.xml 进行声明。

使用 iBATIS 完成数据库操作

使用 iBATIS 完成数据库操作需要如下过程：

- 加载配置文件 sqlMapConfig.xml 并生成 SqlMapClient 对象。

```

SqlMapClient sqlMapper;
Reader reader = Resources.getResourceAsReader("SqlMapConfig.xml");
sqlMapper = SqlMapClientBuilder.buildSqlMapClient(reader);

```

- 根据要执行的 SQL 创建参数对象，要执行更新语句需要创建 Account 对象，代码如下：

```

Account account = new Account();
account.setId(3);
account.setFirstName("wu");
account.setLastName("Wang");
account.setEmailAddress("lisi@dl.cn");

```

- 执行相应的 SQL 语句，调用 SqlMapClient 的 update 方法，第 1 个参数是在 SQL 映射文件中定义的 id。

```
sqlMapper.update("updateAccount", account);
```

6.3 实例

本实例完成对帐户信息的增删改查功能，涉及的文件如下：

- SqlMapConfig.xml，映射配置文件；

- Account.xml, SQL 映射文件;
- Account.java, 模型类;
- AccountDAO, 封装了增删改查方法;
- Test, 对各功能进行测试。

模型类

源文件: Account.java

```
package com.mydomain.domain;

public class Account {

    private int id;
    private String firstName;
    private String lastName;
    private String emailAddress;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getEmailAddress() {
        return emailAddress;
    }
}
```

```
public void setEmailAddress(String emailAddress) {
    this.emailAddress = emailAddress;
}

public String toString() {
    return "编号: " + id + ",姓名: " + lastName + firstName + ",Email:"
        + emailAddress;
}
}
```

SQL 映射文件

所有的 SQL 语句使用 XML 文档描述。

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE sqlMap
    PUBLIC "-//ibatis.apache.org/DTD SQL Map 2.0//EN"
    "http://ibatis.apache.org/dtd/sql-map-2.dtd">

<sqlMap namespace="Account">

    <!-- 使用类别名可以避免每次都输入完整的类名。 -->
    <typeAlias alias="Account" type="com.mydomain.domain.Account"/>

    <!-- 用于描述从查询中返回的列与类属性之间的映射关系，如果列
        名或者别名与属性匹配，也可以不要 resultMap。 -->
    <resultMap id="AccountResult" class="Account">
        <result property="id" column="ACC_ID"/>
        <result property="firstName" column="ACC_FIRST_NAME"/>
        <result property="lastName" column="ACC_LAST_NAME"/>
        <result property="emailAddress" column="ACC_EMAIL"/>
    </resultMap>

    <!-- 使用 Account 类的 resultMap 来完成无参数的查询。 -->
    <select id="selectAllAccounts" resultMap="AccountResult">
        select * from ACCOUNT
    </select>

    <!-- 一个简单的不使用 resultMap 的查询例子，注意 select 中的别名与结果类型
        Account 中的属性一致。 -->
    <select id="selectAccountById" parameterClass="int" resultClass="Account">
        select
            ACC_ID as id,
```

```
        ACC_FIRST_NAME as firstName,
        ACC_LAST_NAME as lastName,
        ACC_EMAIL as emailAddress
    from ACCOUNT
    where ACC_ID = #id#
</select>

<!-- 添加实例，使用 Account 作为参数类。 -->
<insert id="insertAccount" parameterClass="Account">
    insert into ACCOUNT (
        ACC_ID,
        ACC_FIRST_NAME,
        ACC_LAST_NAME,
        ACC_EMAIL
    )values (
        #id#, #firstName#, #lastName#, #emailAddress#
    )
</insert>

<!-- 更新实例，使用 Account 作为参数类。 -->
<update id="updateAccount" parameterClass="Account">
    update ACCOUNT set
        ACC_FIRST_NAME = #firstName#,
        ACC_LAST_NAME = #lastName#,
        ACC_EMAIL = #emailAddress#
    where
        ACC_ID = #id#
</update>

<!-- 删除实例，使用 int 作为参数类型。 -->
<delete id="deleteAccount" parameterClass="int">
    delete from ACCOUNT where ACC_ID = #id#
</delete>

</sqlMap>
```

数据库信息相关的配置文件：

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE sqlMapConfig
    PUBLIC "-//ibatis.apache.org//DTD SQL Map Config 2.0//EN"
    "http://ibatis.apache.org/dtd/sql-map-config-2.dtd>
```

```
<sqlMapConfig>

<!-- 配置一个内置的事务管理器，如果使用应用服务器，可能需要使用它提供的
      事务管理器和被事务管理器管理的数据源。 -->
<transactionManager type="JDBC" commitRequired="false">
    <dataSource type="SIMPLE">
        <property name="JDBC.Driver" value="com.mysql.jdbc.Driver"/>
        <property name="JDBC.ConnectionURL"
            value="jdbc:mysql://127.0.0.1:3306/JSPBOOK"/>
        <property name="JDBC.Username" value="root"/>
        <property name="JDBC.Password" value="root"/>
    </dataSource>
</transactionManager>

<!-- 列出 SQL 映射 XML 文件。 -->
<sqlMap resource="Account.xml"/>
<!-- 如果还有其他的...
<sqlMap resource="Order.xml"/>
<sqlMap resource="Documents.xml"/>
-->

</sqlMapConfig>
```

DAO 层

```
package com.mydomain.data;

import com.ibatis.sqlmap.client.SqlMapClient;
import com.ibatis.sqlmap.client.SqlMapClientBuilder;
import com.ibatis.common.resources.Resources;
import com.mydomain.domain.Account;

import java.io.Reader;
import java.io.IOException;
import java.util.List;
import java.sql.SQLException;

/**
 * 封装了对 Account 进行操作的常用方法
 */
public class AccountDAO {

    /**
     * SqlMapClient 实例是线程安全的，因为只需要一次。这种情况下，使用 static 实例。
```

```
    */
    private static SqlMapClient sqlMapper;

    /**
     * 初始化 sqlMapper。
     */
    static {
        try {
            Reader reader = Resources.getResourceAsReader("SqlMapConfig.xml");
            sqlMapper = SqlMapClientBuilder.buildSqlMapClient(reader);
            reader.close();
        } catch (IOException e) {
            throw new RuntimeException(
                "Something bad happened while building the SqlMapClient instance."
                + e, e);
        }
    }

    /**
     * 查询所有帐户
     */
    public List selectAllAccounts() throws SQLException {
        return sqlMapper.queryForList("selectAllAccounts");
    }

    /**
     * 根据主键查询帐户
     */
    public Account selectAccountById(int id) throws SQLException {
        return (Account) sqlMapper.queryForObject("selectAccountById", id);
    }

    /**
     * 添加新帐户
     */
    public void insertAccount(Account account) throws SQLException {
        sqlMapper.insert("insertAccount", account);
    }

    /**
     * 更新帐户信息
     */
    public void updateAccount(Account account) throws SQLException {
```



```
        sqlMapper.update("updateAccount", account);
    }

    /*
     * 根据主键删除帐户
     */
    public static void deleteAccount(int id) throws SQLException {
        sqlMapper.delete("deleteAccount", id);
    }
}
```

测试

```
package com.mydomain.data;

import java.util.List;

import com.mydomain.domain.Account;

public class Test {
    public static void main(String args[]) {
        try {
            AccountDAO example = new AccountDAO();

            /*
             * 所有帐户
             */
            // List<Account> list = example.selectAllAccounts();
            // System.out.println(list.size());
            /*
             * 添加功能
             */
            // Account account = new Account();
            // account.setFirstName("si");
            // account.setLastName("li");
            // account.setEmailAddress("lisi@dl.cn");
            // example.insertAccount(account);
            /*
             * 删除功能
             */
            // example.deleteAccount(5);
            /*
             * 修改功能
             */
        }
    }
}
```

```
        */
        // Account account = new Account();
        // account.setId(3);
        // account.setFirstName("wu");
        // account.setLastName("Wang");
        // account.setEmailAddress("lisi@dl.cn");
        // example.updateAccount(account);
        /*
        * 根据主键查找
        */
        // Account account = example.selectAccountById(3);
        // System.out.println(account.toString());
    } catch (Exception e) {
        System.out.println(e.toString());
    }
}
}
```

第 7 章 Hibernate 技术

7.1 概述

相对于传统的使用 SQL 和 JDBC API 进行的手工编程，Hibernate 的目标是替用户完成 95% 的数据持久性相关的编程任务，从而大大减轻开发人员的工作，提高开发效率。

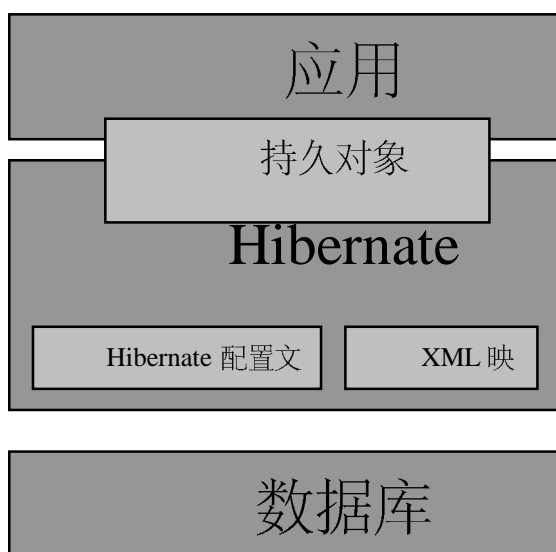


图 7.1 Hibernate 顶层架构图

Hibernate 的顶层架构如图 7.1 所示。Hibernate 位于应用与数据库之间，完成持久对象与数据库表记录之间的映射。映射功能的完成主要使用的是 Hibernate 提供的 API。要使用 Hibernate 提供的 API，用户需要提供 Hibernate 配置文件来描述数据库的相关信息，需要提供映射文件来描述类和表之间的对应关系。

Hibernate 非常复杂并且支持多种工作方式。下面给出两种极端，最简单的情况和最复杂的情况，分别如图 7.2 和图 7.3 所示。

在图 7.2 所示的结构中，应用自己来提供 JDBC 连接，并且自己来管理事务等。在图 7.3 所示的结构中，Hibernate 把 JDBC/JTA 等操作从应用中分离出来，然后由 Hibernate 来管理这些细节。

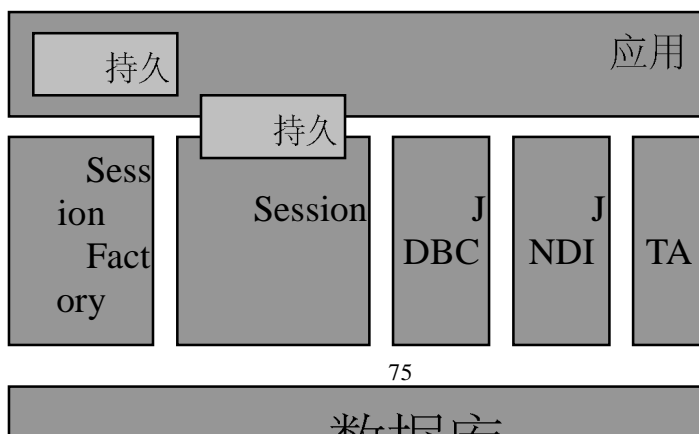


图 7.2 最简单结构图

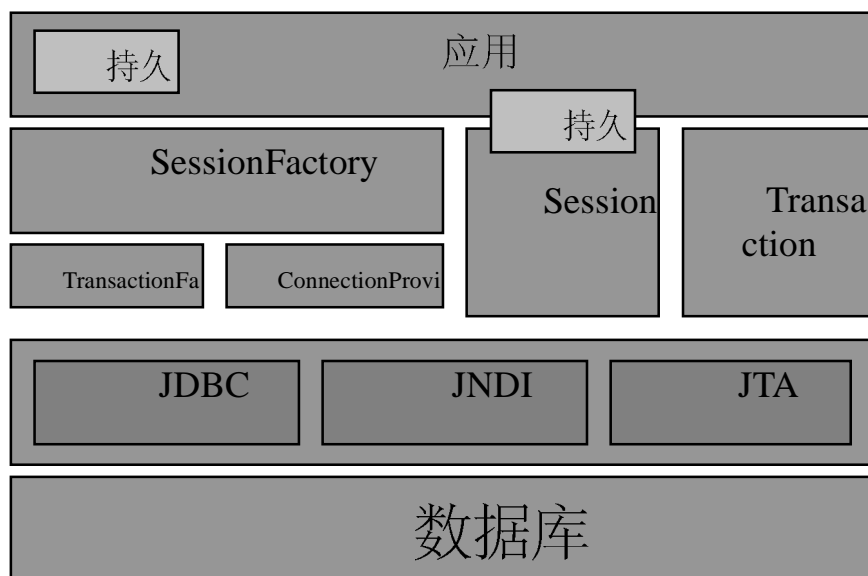


图 7.3 最复杂结构图

下面对图中描述的主要 API（也是 Hibernate 的主要 API）进行简单的介绍：

- SessionFactory（org.hibernate.SessionFactory），用于保存编译后的数据库映射信息，也就是把 Hibernate 配置文件中的映射信息包存在 SesssionFactory 中。
- Session（org.hibernate.Session），相对来说生命周期比较短，表示应用和持久存储之间的一次会话。封装了 JDBC 对象，能够生成 Transaction 对象，保存了持久对象。
- Transaction（org.hibernate.Transaction），应用用于指定原子操作序列的单线程的、生命周期较短的对象，使用它把应用从 JDBC、JTA 和 CORBA 等事务中解放出来，会话可能会跨越多个事务。

在应用中，主要使用这些 API 完成对实体的管理。

7.2 开发人员的工作

使用 HibernateO/R 映射工具完成持久性，开发人员需要完成的工作如下：

- ◆ 搭建环境；
- ◆ 创建模型类，与 iBATIS 中模型类的功能类似；
- ◆ 创建 Hibernate 配置文件，描述数据库的相关信息；
- ◆ 创建 Hibernate 映射文件，用于描述类的属性与表的列的对应关系；
- ◆ 在数据访问层使用 Hibernate 提供的接口完成数据库的操作。

下面以更新功能为例介绍使用 Hibernate 技术人员需要完成的工作。

搭建环境

Hibernate 最新版本以及相关文档可以从 <http://www.hibernate.org/> 下载。

创建模型类

模型类就是普通的 **JavaBean**，在 **Hibernate** 技术中模型类需要提供如下内容：

- 实现一个默认的构造方法，这样的话 **Hibernate** 就可以使用 **Constructor.newInstance()** 来实例化它们。
- 定义与持久化操作相关的属性以及对属性进行操作的 **setter** 方法和 **getter** 方法。
- 如果是集合类型的属性，它的类型必须定义为集合的接口。例如：**List**、**Set**。
- 定义关系属性以及对关系属性进行操作的 **setter** 方法和 **getter** 方法。
- 提供一个标识属性（**identifier property**）。如果没有该属性，一些功能不起作用，比如：级联更新（**Cascaded updates**）**Session.saveOrUpdate()**。
与 **iBATIS** 中的模型类基本相同。

创建 Hibernate 配置文件

Hibernate 配置文件主要描述要访问的数据库的信息以及应用所涉及的 **Hibernate** 映射文件，**Hibernate** 映射文件在后面介绍。配置文件可以采用 **XML** 文件的形式也可以采用属性文件的形式。

配置文件中需要描述的数据库相关信息包括：

- 数据源

或者

- **JDBC** 驱动程序；
- **URL**；
- 用户名；
- 口令。

与数据库相关的配置信息是最基本的信息，还可以配置很多信息。下面给出了 1 个采用 **XML** 文件格式的配置文件。

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <!-- 数据库连接相关信息设置 -->
        <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="connection.url">jdbc:mysql://127.0.0.1:3306/JSPBOOK</property>
        <property name="connection.username">root</property>
        <property name="connection.password">root</property>

        <!-- 数据库连接池 -->
        <property name="connection.pool_size">1</property>

        <!-- SQL dialect -->
        <property name="dialect">org.hibernate.dialect.HSQLDialect</property>
```

```
<!-- 启用 Hibernate 的自动会话上下文管理-->
<property name="current_session_context_class">thread</property>

<!-- 关闭两层缓存 -->
<property
name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>

<!-- 把所有执行过的 SQL 输出到标注输出设备上 -->
<property name="show_sql">true</property>

<!-- 在启动的时候删除和重新创建数据库 schema -->
<property name="hbm2ddl.auto">create</property>

<!-- 类与表的关系映射文件 -->
<mapping resource="com/mydomain/data/Account.hbm.xml"/>

</session-factory>

</hibernate-configuration>
```

创建 Hibernate 映射文件

Hibernate 映射文件主要用于描述模型类与数据库表的列之间的对应关系以及实体之间的关系。Hibernate 映射文件的基本格式：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
...
</hibernate-mapping>
```

在<hibernate-mapping>中定义映射关系，每个类的映射关系可以使用一个<class>元素，下面是一个映射例子：

```
<class name="com.mydomain.domain.Account" table="ACCOUNT">
    <id name="id" column="ACC_ID">
        <generator class="native"/>
    </id>
    <property name="firstName" column="ACC_FIRST_NAME"/>
    <property name="lastName" column="ACC_LAST_NAME"/>
    <property name="emailAddress" column="ACC_EMAIL"/>
</class>
```

<class>元素中的 name 表示完整的类名，table 表示数据库表的名字。子元素<id>表示主键属

性, name 表示属性名, column 表示列名, 其子元素<generator>表示主键生成策略。子元素<property>表示每个属性与数据库表中列的对应关系。

与 iBATIS 技术不同, 在 Hibernate 映射文件中看不到 SQL 语句, 大部分的 SQL 语句都会有系统生成。

编写应用

在应用使用 Hibernate 技术的主要过程包括:

- 加载配置文件, 可以使用下面的代码:


```
new Configuration().configure()
```
- 得到 SessionFactory, 使用下面的代码:


```
SessionFactory factory = new Configuration().configure().buildSessionFactory();
```
- 得到 Session:


```
Session session = factory.getCurrentSession();
```
- 操作对象
 - 开始事务:


```
session.beginTransaction();
```
 - 执行操作, 常用操作的执行如下:
 - ◆ 保存对象: session.save(o);
 - ◆ 更新对象: session.update(o);
 - ◆ 删除对象: session.delete(o);
 - ◆ 根据主键查找: session.load(类型, 主键)
 - 提交事务:


```
session.getTransaction().commit();
```

7.3 实例

Account 类与 iBATIS 中所使用的相同。

表与类对应关系设置:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

    <class name="com.mydomain.domain.Account" table="ACCOUNT">
        <id name="id" column="ACC_ID">
            <generator class="native"/>
        </id>
        <property name="firstName" column="ACC_FIRST_NAME"/>
        <property name="lastName" column="ACC_LAST_NAME"/>
        <property name="emailAddress" column="ACC_EMAIL"/>
    </class>
```

```
</hibernate-mapping>
```

数据库信息配置：

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <!-- 数据库连接相关信息设置 -->
        <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="connection.url">jdbc:mysql://127.0.0.1:3306/JSPBOOK</property>
        <property name="connection.username">root</property>
        <property name="connection.password">root</property>

        <!-- 数据库连接池 -->
        <property name="connection.pool_size">1</property>

        <!-- SQL dialect -->
        <property name="dialect">org.hibernate.dialect.HSQLDialect</property>

        <!-- 启用 Hibernate 的自动会话上下文管理-->
        <property name="current_session_context_class">thread</property>

        <!-- 关闭两层缓存 -->
        <property
name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>

        <!-- 把所有执行过的 SQL 输出到标注输出设备上 -->
        <property name="show_sql">true</property>

        <!-- 在启动的时候删除和重新创建数据库 schema -->
        <property name="hbm2ddl.auto">create</property>

        <!-- 类与表的关系映射文件 -->
        <mapping resource="com/mydomain/data/Account.hbm.xml"/>

    </session-factory>

</hibernate-configuration>
```


DAO 层。

```
package com.mydomain.data;

import java.sql.SQLException;
import java.util.List;

import org.hibernate.Session;

import com.mydomain.domain.Account;
import com.mydomain.util.HibernateUtil;

/**
 * 封装了对 Account 进行操作的常用方法
 */
public class AccountDAO {

    /**
     * 查询所有帐户
     */
    public List selectAllAccounts() throws SQLException {

        Session session = HibernateUtil.getSessionFactory().getCurrentSession();

        session.beginTransaction();

        List result = session.createQuery("from Account").list();

        session.getTransaction().commit();

        return result;

    }

    /**
     * 根据主键查询帐户
     */
    public Account selectAccountById(int id) throws SQLException {

        Session session = HibernateUtil.getSessionFactory().getCurrentSession();

        Account account = (Account) session.load(Account.class, Integer
            .valueOf(id));

    }

}
```

```
        return account;
    }

    /*
     * 添加帐户
     */
    public void insertAccount(Account account) throws SQLException {
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();

        session.beginTransaction();

        session.save(account);

        session.getTransaction().commit();
    }

    /*
     * 更新帐户
     */
    public void updateAccount(Account account) throws SQLException {
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();

        session.beginTransaction();

        session.update(account);

        session.getTransaction().commit();
    }

    /*
     * 删除帐户
     */
    public void deleteAccount(int id) throws SQLException {

        Session session = HibernateUtil.getSessionFactory().getCurrentSession();

        session.beginTransaction();

        Account account = (Account) session.load(Account.class, Integer
            .valueOf(id));

        session.delete(account);
    }
}
```

```
        session.getTransaction().commit();
    }

}
```

测试类也基本相同。

使用 **Hibernate** 能够减少用户编写的 **SQL** 语句以及重复的 **JDBC** 代码。但是相对于直接使用 **JDBC** 完成数据库操作来说，使用 **Hibernate** 技术需要编写多个配置文件，并且需要掌握相关 **API** 的使用，对用户来说也是一种新的压力。这个用户不用担心，多数集成开发环境中都提供了很少的支持，能够根据表结构生成实体类文件，以及映射文件，用户需要做的就是使用 **API** 进行基本的操作。

第 8 章 JPA 技术

8.1 概述

JPA 是 Java Persistence API 的缩写，是在 EJB 3 中增加的 Java 持久性解决方案。JPA 本身是一组 API，持久性的实现者通常是 O/R 映射工具，现在用的比较多的是 Hibernate 和 Toplink。

JPA 中通过配置文件 persistence.xml 来描述数据库的相关信息，类与数据库表的对应关系在实体类中通过元注释声明。JPA 中提供了一组 API 来完成对实体的相关操作。

persistence.xml 需要配置的信息主要包括：驱动程序、URL、用户名和口令。

JPA 提供的元注释主要用于实体类，主要包括：

- Entity，声明某个类为实体类；
- Table，声明与哪个表对应；
- Id，声明主键字段；
- Column，声明与表的什么字段对应；
- OneToMany、ManyToOne、ManyToMany 和 ManyToOne，用于声明实体之间的关系。

JPA 中提供的主要类和接口包括：

- EntityManager，通过该接口完成对实体的各种操作，包括实体的添加、修改、删除、各种查询；
- EntityManagerFactory，通过该接口对 EntityManager 对象进行管理；
- EntityTransaction，用于管理实体操作相关的事务；
- Query，完成各种查询。
 - Persistence，用于获取 EntityManagerFactory 对象。

8.2 开发人员的工作

开发人员使用 JPA 进行持久性操作需要做的工作包括：

- 在配置文件 persistence.xml 中配置数据库相关的信息；
- 创建实体类，在实体类上通过元注释声明实体-表对应关系、属性-列对应关系以及实体之间的关联关系。
- 使用 JPA 提供的 API 完成对实体相关的操作。

编写配置文件

配置文件 persistence.xml 主要用于配置数据库相关信息以及管理的实体类。配置文件的基本格式如下面的代码所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">
```

```

<persistence-unit name="JPAPU" transaction-type="RESOURCE_LOCAL">
  <provider>org.hibernate.ejb.HibernatePersistence</provider>
  <class>com.mydomain.domain.Account</class>
  <properties>
    <property name="hibernate.connection.driver_class"
      value="com.mysql.jdbc.Driver" />
    <property name="hibernate.connection.url"
      value="jdbc:mysql://127.0.0.1:3306/JSPBOOK" />
    <property name="hibernate.connection.username" value="root" />
    <property name="hibernate.connection.password" value="root" />
  </properties>
</persistence-unit>

</persistence>

```

创建实体类

创建实体类，在实体类中主要完成的工作包括：

使用 `@Entity` 声明当前类为实体类；

使用 `@Table(name="表名")` 声明当前实体类与哪个表对应，如果表名和类名相同，则可以省略该元注释，例如：

```

@Entity
@Table(name="CUST", schema="RECORDS")
public class Customer { ... }

```

使用 `@Id` 声明某个属性为主键字段，可以把元注释声明在属性上，也可以将元注释声明在属性对应的 `get` 方法上，例如：

```

@Id
public Long getId() { return id; }

```

使用 `@Column(name="列名")` 声明属性与字段的对应关系，可以把元注释声明在属性上，也可以将元注释声明在属性对应的 `get` 方法。

```

@Column(name="DESC", nullable=false, length=512)
public String getDescription() { return description; }

```

如果在实体类中要声明与其它实体之间的关系，可以使用 `OneToMany`、`ManyToOne`、`ManyToMany` 等，例如顾客和订单项之间的一对多可以通过下面的代码声明：

在顾客类中：

```

@OneToMany(cascade=ALL, mappedBy="customer")
public Set getOrders() { return orders; }

```

在订单类中：

```

@ManyToOne
@JoinColumn(name="CUST_ID", nullable=false)
public Customer getCustomer() { return customer; }

```

另外 JPA 还提供里一些其它的元注释，例如可以使用 `PrePersist` 元注释和 `PostPersist` 声明在

保存对象之前和之后要执行的方法，可以使用 **PreRemove** 和 **PostRemove** 声明在删除对象之前和之后要执行的方法。

对于用户来说，编写这样复杂的实体类比较麻烦，但是现在的集成开发环境都能够提供很好的支持，可以根据表结构生成实体类，对于用户来说，只需要编写对实体进行操作的代码就可以了。

编写应用

在编写完实体类之后，就可以使用 **JPA** 编写数据访问层的代码了。使用 **JPA** 完成操作的主要过程包括：

- 得到实体管理器工厂，可以通过注入的方式，也可以通过 **Persistence** 的方法；

通过 **Persistence** 的方法获取实体管理器工厂：

```
EntityManagerFactory factory = Persistence.createEntityManagerFactory("PUName");
```

通过注入方式获取实体管理器工厂：

```
@PersistenceUnit
```

```
private EntityManagerFactory factory;
```

- 得到实体管理器，可以通过实体管理器工厂得到实体管理器，也可以通过注入的方式得到实体管理器；

通过实体管理器工厂得到实体管理器：

```
EntityManager em = factory.getManager();
```

通过注入的方式得到实体管理器：

```
@PersistenceContext
```

```
EntityManager em;
```

- 通过实体管理器对实体进行操作，包括增删改查等操作。增删改操作的代码比较固定，而查询的时候需要创建 **Query** 对象，创建这个对象的过程把需要执行的查询语句作为参数，而查询语句使用的是 **JPA QL**。注意：在使用增删改的时候需要使用事务处理。

添加对象：

```
em.persist(o);
```

删除对象：

```
em.remove(o);
```

更新对象：

```
o.setXX(xx);
```

执行查询：

```
String sql = "select u from user u";
```

```
Query q = manager.createQuery(sql);
```

// 查询语句中的 user 不是表名，而是抽象模式名。

根据主键查询：

```
Teacher t = em.find(Teacher.class, "0011");
```

- 得到结果，如果执行 **SELECT** 语句，需要通过 **Query** 对象的方法得到查询结果，而查询结果是实体的对象或者是实体的对象的集合。

得到集合：

```
List list = q.getResultList();
```

得到对象：

```
Teacher t = q.getSingleResult();
```

8.3 实例

本实例完成的功能与前面介绍的 **Hibernate** 实例完成的功能相同。包括的文件主要有：

- persistence.xml 文件，描述数据库相关信息；
- Account 类，是实体类，使用多个元注释来描述与列的对应关系；
- 数据访问层，完成对实体的基本操作。

下面分别进行介绍：

persistence.xml 文件

该文件的代码参见 8.2 小节。

Account 类

Account 类是实体类，代码如下；

```
package com.mydomain.domain;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

/**
 * Account entity.
 *
 * @author MyEclipse Persistence Tools
 */
@Entity
@Table(name = "account", catalog = "jspbook", uniqueConstraints = {})
public class Account implements java.io.Serializable {

    // 属性

    private Integer accId;
    private String accFirstName;
    private String accLastName;
    private String accEmail;

    // 构造方法

    /** 默认构造方法 */
    public Account() {
    }
}
```

```
/** 带有一个参数的构造方法 */
public Account(Integer accId) {
    this.accId = accId;
}

/** 对所有成员进行初始化的构造方法 */
public Account(Integer accId, String accFirstName, String accLastName,
    String accEmail) {
    this.accId = accId;
    this.accFirstName = accFirstName;
    this.accLastName = accLastName;
    this.accEmail = accEmail;
}

// 属性访问器
@GeneratedValue
@Id
@Column(name = "ACC_ID", unique = true, nullable = false, insertable = true, updatable = true)
public Integer getId() {
    return this.accId;
}

public void setId(Integer accId) {
    this.accId = accId;
}

@Column(name = "ACC_FIRST_NAME", unique = false, nullable = true, insertable = true,
updatable = true, length = 20)
public String getFirstName() {
    return this.accFirstName;
}

public void setFirstName(String accFirstName) {
    this.accFirstName = accFirstName;
}

@Column(name = "ACC_LAST_NAME", unique = false, nullable = true, insertable = true,
updatable = true, length = 20)
public String getLastName() {
    return this.accLastName;
}
```



```
public void setLastName(String accLastName) {
    this.accLastName = accLastName;
}

@Column(name = "ACC_EMAIL", unique = false, nullable = true, insertable = true, updatable =
true, length = 30)
public String getEmailAddress() {
    return this.accEmail;
}

public void setEmailAddress(String accEmail) {
    this.accEmail = accEmail;
}

public String toString() {
    return "编号: " + getId() + ",姓名: " + getLastName() + getFirstName()
        + ",Email:" + getEmailAddress();
}
}
```

DAO 层

完成对实体的基本操作。代码如下：

```
package com.mydomain.data;

import java.sql.SQLException;
import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;

import com.mydomain.domain.Account;

/**
 * 封装了对 Account 进行操作的常用方法
 */
public class AccountDAO {

    /**
     * 得到实体管理器
     */
    public EntityManager getEntityManager() {
```

```
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("JPAPU");

        EntityManager em = emf.createEntityManager();

        return em;
    }

    /*
     * 查询所有帐户
     */
    public List selectAllAccounts() throws SQLException {

        EntityManager em = getEntityManager();

        List result = em.createQuery("select o from Account o").getResultList();

        em.close();

        return result;
    }

    /*
     * 根据主键查询帐户
     */
    public Account selectAccountById(int id) throws SQLException {

        EntityManager em = getEntityManager();

        Account account = em.find(Account.class, new Integer(id));

        em.close();

        return account;
    }

    /*
     * 添加帐户
     */
    public void insertAccount(Account account) throws SQLException {
```

```
EntityManager em = getEntityManager();

EntityTransaction transaction = em.getTransaction();

transaction.begin();

em.persist(account);

transaction.commit();

em.close();
}

/*
 * 更新帐户
 */
public void updateAccount(Account account) throws SQLException {

    EntityManager em = getEntityManager();

    EntityTransaction transaction = em.getTransaction();

    transaction.begin();

    em.merge(account);

    transaction.commit();

    em.clear();
}

/*
 * 删除帐户
 */
public void deleteAccount(int id) throws SQLException {

    EntityManager em = getEntityManager();

    EntityTransaction transaction = em.getTransaction();

    transaction.begin();

    Account account = em.find(Account.class, new Integer(id));
```

```
        em.remove(account);

        transaction.commit();

        em.close();
    }

}
```

第 9 章 简单自定义持久层框架

为了让读者了解持久层框架的实现机制，本节实现了一个简单的持久层框架，能够完成对象的增删改查操作。使用这个框架完成对象的增删改查，用户需要完成的工作包括：

- 提供一个 `jdbc.properties` 文件，包括要连接的数据库的基本信息，下面展示了基本用法：

```
driverclass=com.mysql.jdbc.Driver
url=jdbc:mysql://127.0.0.1:3306/jspbook?useUnicode=true&characterEncoding=gbk&autoReconnect=true
username=root
password=root
```

- 提供实体类，与普通 Java 类基本相同，但是需要采用元注释声明类对应的数据库表，主键列以及属性与列的对应关系下面的代码展示了用法，与 JPA 中的实体类非常相似：

```
package com.my.myjpa.test.bean;

import com.my.myjpa.anoation.Column;
import com.my.myjpa.anoation.Id;
import com.my.myjpa.anoation.Table;

@Table(name = "usertable")
public class UserBean {
    @Id
    @Column(name = "userid")
    private String userid;

    @Column(name = "username")
    private String username;

    @Column(name = "usertype")
    private String usertype;

    @Column(name = "department")
    private String department;

    @Column(name = "user_en")
    private String user_en;

    @Column(name = "pass")
    private String pass;
```

```
// 省略了 set 方法和 get 方法
public String toString() {
    return new StringBuffer().append("username:").append(username).append(
        ",userpass:").append(pass).append(",userid:").append(userid)
        .append(",user_en:").append(user_en).toString();
}
}
```

- 使用 Session 类完成对象的操作，下面的代码展示了基本用法：

保存对象信息到数据库：

```
UserBean user = new UserBean();
Session session = new Session();
user.setUserid("777");
user.setUsername("李绪成");
user.setDepartment("jisuanjixi");
user.setPass("dddddd");
user.setUser_en("lixucheng");
user.setUserType("普通用户");
session.persist(user);
```

修改信息：

```
UserBean user = new UserBean();
Session session = new Session();
user.setUserid("777");
user.setUsername("李绪成");
user.setDepartment("jisuanjixi");
user.setPass("dddddd");
user.setUser_en("lixucheng");
user.setUserType("普通用户");
session.update(user);
```

删除信息：

```
UserBean user = new UserBean();
Session session = new Session();
user.setUserid("777");
session.delete(user);
```

根据主键查询对象：

```
UserBean user = new UserBean();
Session session = new Session();
user = (UserBean) session.findById(UserBean.class, "00001");
System.out.println(user);
```

查找所有对象：

```
UserBean user = new UserBean();
Session session = new Session();
List<UserBean> list = session.findAll(UserBean.class);
for (UserBean temp : list) {
```

```
System.out.println(temp);  
}
```

下面主要介绍框架是如何提供对这些操作的支持的。为了提供对这些操作的支持，框架主要完成如下工作：

- 根据用户通过 `jdbc.properties` 属性文件提供的信息建立与数据库的连接；
- 提供了 `Table`、`Id` 和 `Column` 等元注释供用户在实体类中使用；
- 根据用户在实体类中提供的元注释，获取实体类与数据库表的对应关系、主键属性与主键列的对应关系以及列与属性的对应关系，这些信息称为映射信息；
- 根据用户要完成的操作以及通过解析得到的映射构造 `sql` 语句；
- 执行 `SQL` 语句；
- 根据映射关系对结果集进行处理，转换成对象或者对象的列表。

下面分别介绍。

9.1 连接数据库

首先从用户提供的 `jdbc.properties` 属性文件中获取驱动程序、URL、用户名和口令等信息，然后根据这些信息建立与数据库的连接。下面是连接数据库连的代码。

```
package com.my.myjpa.core;  
// 省略了 import 语句  
  
public class ConnectionManager {  
    private static Connection con;  
  
    static{  
        try{  
            con = createConnection();  
        }catch(ClassNotFoundException e1){  
            System.out.println("找不到驱动程序: "+e1.toString());  
        }catch(IOException e2){  
            System.out.println("找不到配置文件 JDBC.properties");  
        }catch(SQLException e3){  
            System.out.println("连接数据库出错! ");  
        }  
    }  
  
    private static Connection createConnection()  
        throws ClassNotFoundException, SQLException, IOException{  
        // 读取资源文件  
        ResourceBundle jdbcInfo = ResourceBundle.getBundle("JDBC");  
  
        // 获取资源文件中信息  
        String driver = jdbcInfo.getString("driverclass");  
        String url = jdbcInfo.getString("url");
```

```
String username = jdbcInfo.getString("username");
String password = jdbcInfo.getString("password");

Class.forName(driver);
Connection con
    = DriverManager.getConnection(url,username,password);
return con;
}

public static Connection getConnection(){
    return con;
}

protected void finalize(){
    con = null;
}
}
```

9.2 编写元注释类

为了让用户在实体类中声明与数据库的对应关系，框架提供了元注释：

- Table，描述与哪个表对应；
- Id，定义在属性上，表示当前属性是主键属性；
- Column，定义在属性上，描述当前属性与哪一列对应。

Table 元注释的定义如下：

```
package com.my.myjpa.anoation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface Table {
    String name();
}
```

Id 元注释的定义如下：

```
package com.my.myjpa.anoation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
```



```
import java.lang.annotation.Target;

// 注释会在运行的时候起作用
@Retention(RetentionPolicy.RUNTIME)
// 注释可以用在类上，可以用在方法上
@Target({ElementType.FIELD})
public @interface Id{
}
```

Column 元注释的定义如下：

```
package com.my.myjpa.anoation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD})
public @interface Column {
    public String name();
}
```

9.3 获取映射信息

需要根据用户提供的实体类来获取映射信息，主要使用反射机制来完成。要获取的映射信息是通过 **Table**、**Id** 和 **Column** 元注释声明在类中。**Table** 声明在类上，**Id** 和 **Column** 声明在属性上。所以需要分别获取类的元注释和属性的元注释。

类的元注释通过类的 **getAnnotations** 来获取，获取之后需要找出哪个元注释是 **Table**，然后从中取出表名。

要获取属性的元注释，需要先得到所有的属性，然后获取每个属性的元注释，从中找出 **Id** 元注释和 **Column** 元注释，从而得到主键和属性-列对应关系。

下面的代码完成的功能就是根据类解析映射信息。

```
public void parseAnnotation(Class c) {
    // 得到类的元注释
    Annotation[] as = c.getAnnotations();
    if (as != null) {
        for (Annotation a : as) {
            if (a instanceof Table) {
                Table t = (Table) a;
                tablename = t.name();
            }
        }
    }
}
```

```

// 得到所有属性
Field fields[] = c.getDeclaredFields();
if (fields != null) {
    for (Field f : fields) {
        // 得到每个属性的元注释
        Annotation[] fas = f.getAnnotations();
        // 判断
        for (Annotation fa : fas) {
            // 记录主键字段
            if (fa instanceof Id) {
                id = f.getName();
            }

            // 记录列与属性的对应关系，key 表示属性名，value 表示列名
            if (fa instanceof Column) {
                columns.put(f.getName(), ((Column) fa).name());
                properties.put(((Column) fa).name(), f.getName());
            }
        }
    }
}
}

```

9.4 构造 SQL 语句

本实例中完成的框架完成的 SQL 语句包括：

- insert 语句；
- delete 语句；
- update 语句；
- select 语句，包括根据主键查询对象和查询所有对象两个功能；

下面以 update 语句为例介绍 SQL 语句的构造。update 语句包括 3 个组成部分：update 部分，set 部分和 where 条件。update 部分需要用到前面解析出来的表名，set 部分需要用到解析出来的属性-列对应关系，where 条件需要用到解析出来的主键。下面给出了得到 update 语句的方法，其它方法参见本书所带光盘。

```

/*
 * 构造update语句
 */
private String getUpdateSql(Object o) throws NoSuchMethodException,
        IllegalArgumentException, IllegalAccessException,
        InvocationTargetException {
    // 列名部分
    StringBuffer columnStr = new StringBuffer();
}

```

```

// 所有属性的名字
Set propertyNames = columns.keySet();

// 表示属性值
params = new ArrayList();

// 把所有参数放到params
// 构造sql语句中的列名部分
for (Object temp : propertyNames) {
    if (((String) temp).equals(id))
        continue;
    // 得到列名, 然后添加到查询语句中
    columnStr.append(columns.get(temp));
    // 列名之间使用逗号
    columnStr.append("=?, ");
    // 把属性值添加到参数列表中
    params.add(getPropertyValue(o, (String) temp));
}

// 构造sql语句
StringBuffer sb = new StringBuffer();
sb.append("update ");
sb.append(tablename);
sb.append(" set ");
sb.append(columnStr);

// 删除最后一个", "
sb.deleteCharAt(sb.length() - 1);

sb.append("where ");
sb.append(id);
sb.append("=?");

params.add(getPropertyValue(o, id));

return sb.toString();
}

```

为了对SQL语句中的变量赋值,需要使用反射机制获取对象的值,然后通过PreparedStatement语句对象的相应的set方法赋值。获取对象值的代码如下:

```

// 根据属性确定方法, 例如 username, 方法名为 getUsername
Method method = o.getClass().getMethod(
    "get" + (char) (propertyName.charAt(0) - 32)

```

```
        + propertyName.substring(1));

// 调用 get 方法
return method.invoke(o);
```

9.5 执行 SQL 语句

使用 `PreparedStatement` 语句对象的 `executeUpdate` 方法执行没有结果集的 SQL 语句，使用 `PreparedStatement` 语句对象的 `executeQuery` 方法执行有结果集返回的 SQL 语句。

```
// 执行有结果集的 sql 语句
ResultSet rs = pstmt.executeQuery();
// 执行没有结果集的 sql 语句
pstmt.executeUpdate();
```

9.6 处理结果集

在查询到结果集之后需要把结果集转换成对象，如果是一条记录，结果集可以转换成对象，如果结果集有多条记录，需要把结果集转换成对象的集合，通常采用 `ArrayList` 类型。不管是把一条记录转换成一个对象，还是把多条记录转换成对象的集合，都需要针对每条记录创建一个对象，然后把记录的每一列赋值给该对象的相应属性。下面的代码把结果集转换成对象链表。

```
public List getListFromResult(ResultSet rs, Class c) throws Exception {
    ArrayList list = new ArrayList();
    Object o = null;

    // 获取元数据
    ResultSetMetaData rsmt = rs.getMetaData();

    // 得到列数
    int columns = rsmt.getColumnCount();

    while (rs.next()) {
        o = c.newInstance();
        // 获取每一列然后处理
        for (int i = 0; i < columns; i++) {
            // 得到列名
            String tempColumnName = rsmt.getColumnName(i + 1);

            // 得到列的值
            Object tempValue = rs.getObject(tempColumnName);

            // 赋值
            setPropertyValue(o, (String) properties.get(tempColumnName),
                             tempValue);
        }
    }
}
```

```
        list.add(o);
    }
    return list;
}
```

代码中调用的 `setProperty` 方法采用反射机制对对象属性赋值。该方法的代码如下：

```
private void setProperty(Object o, String propertyName, Object value)
    throws SecurityException, NoSuchMethodException,
        IllegalArgumentException, IllegalAccessException,
        InvocationTargetException {
    // 得到 set 方法名
    String methodName = new StringBuffer("set").append(
        Character.toUpperCase(propertyName.charAt(0))).append(
            propertyName.substring(1)).toString();

    // 得到赋值方法
    Method method = o.getClass().getMethod(methodName, value.getClass());

    // 赋值
    method.invoke(o, value);
}
```

上面给出了框架的部分代码，框架的完整代码参见本书所附光盘。

9.7 自定义框架小结

本节介绍的框架仅仅完成了基本的增删改查功能，与实际应用还有一段距离，在如下几个方面需要改进：

- 增加复杂的查询；
- 增加关系的操作；
- 增加事务处理；
- 增加更加灵活的查询结果形式。

第 2 部分 实例

主要内容：

第 10 章 JavaMail

第 11 章 办公用品申请管理系统

第 12 章 自动信息收集系统

第 10 章 JavaMail

在 Java EE 应用程序中，经常需要发送 E-mail。Java EE 框架为应用提供了 JavaMail 接口，通过 JavaMail 相关的接口可以读取邮件服务器的邮件，并且可以完成邮件的发送过程。

本章的主要内容包括：

- E-mail 体系结构
- JavaMail API
- 如何使用 JavaMail API 发送邮件
- 如何使用 JavaMail API 接收邮件

10.1 E-mail 体系结构

10.1.1 什么是 E-mail

E-mail 是用户间或应用程序间交换信息的 Internet 标准。每个用户都有自己的网上邮箱，发送方把信息发送到自己的网上信箱，并声明信息的接收方；该信箱所在的“邮局”会把信息发送到接收方的“邮局”，接收方从接收方的“邮局”中自己的信箱获取信息。这样就完成了信息从发送方到接收方的传递。所以，要发送或者接收邮件首先应该有自己的邮箱。

E-mail 消息可以包含普通文本，也可以包含更为复杂的多媒体数据类型和图像声音等。这样，用户就可以交换各种各样的信息。

每个 E-mail 消息的头信息中都包含消息的发出者、发送的目的地和其他相关信息。

10.1.2 E-mail 体系结构

要完成消息的交互，需要几方面的支持：邮件发送客户端程序、邮件接收客户端程序、邮件发送服务器和邮件接收服务器，此外，还需要相关的通信协议。

邮件发送客户端程序和邮件接收客户端程序可以是相同的，例如经常使用的微软的 Outlook 既可以发送邮件，也可以接收邮件。

邮件发送服务器和邮件接收服务器也可以是相同的服务器。在与邮件服务器交互的过程中，主要完成两个动作，把邮件发送到邮件服务器，以及从邮件服务器读取邮件。所以，主要使用两类协议，分别进行邮件的发送和接收。

邮件从发送方到接收方的传递过程（参见图 10.1）如下：

- （1）邮件发送方通过邮件发送客户端把邮件发送到发送方的邮件服务器，在发送的过程中需要用到 SMTP 协议。
- （2）发送方的邮件服务器把邮件发送到接收方的邮件服务器，使用的协议也是 SMTP。
- （3）邮件接收方从接收方邮件服务器接收邮件，使用 POP3 协议或者 IMAP 协议。

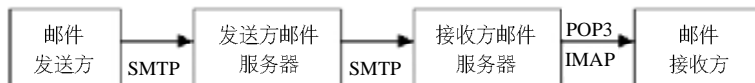


图 10.1 邮件从发送方到接收方的传递过程

10.1.3 E-mail 相关的协议

- 简单邮件传输协议（SMTP）

在邮件发送方把邮件发送到发送方邮件服务器的时候，需要用到简单邮件传输协议（Simple Mail Transport Protocol, 简称 SMTP），发送方邮件服务器把邮件发送到接收方邮件服务器的时候，也要用到 SMTP。SMTP 是应用程序与邮件服务器通信并发送邮件的 Internet 标准，SMTP 通信基于 TCP 协议端口 25 之上。

- 邮箱协议（POP3）

邮件接收方从接收方邮件服务器接收邮件的时候，需要使用检索协议，可以使用 POP3 或者 IMAP。POP3 是 Post Office Protocol 的简称，是用于接收方从邮件服务器上检索 E-mail 消息的协议，工作在 TCP 协议端口 110 之上。

- Internet 消息访问协议（IMAP）

IMAP 是 Internet Message Access Protocol 的简称，与 POP3 基本相同。完成的主要功能是从邮件服务器接收邮件，使用的端口是 103。

10.1.4 什么是 Java Mail

在 Java EE 应用中，经常需要通过 E-mail 与用户进行交互，主要是与邮件服务器的交互，例如向邮件服务器发送邮件，或者从邮件服务器接收邮件。如果是用户之间直接进行交互可以使用邮件客户端程序，例如微软的 Outlook。如果想在应用程序中发送邮件，就不能直接使用通用的客户端了，需要编写自己专门的邮件发送和接收代码，并且需要与邮件服务器进行交互。可以通过 Socket 编程，使用相关的协议完成，但是这个过程非常复杂。而 Java Mail 提供了比较便利的解决方案。

JavaMail 是 Java EE 中的标准 API，是对邮件服务器访问过程的封装。使用 JavaMail API 则不需要编写与邮件服务器交互的详细过程，只需要调用相关接口即可。在接口中封装了与邮件服务器交互的详细过程。

本章的主要内容是介绍如何通过 JavaMail API 完成邮件的发送和接收。

10.2 JavaMail API

JavaMail API 主要包括四个部分：Session，Message，Transport 和 InternetAddress。下面分别进行介绍。

10.2.1 Session

Session 定义了全局的和每个用户的与邮件相关的属性，这些属性详细说明了客户机和服务器如何交流信息。Session 中定义的属性如下：

- mail.store.protocol：确定检索邮件所使用的协议。可以是 IMAP，也可以是 POP3。
- mail.transport.protocol：确定发送邮件所使用的协议，可以是 SMTP。
- mail.host：确定邮件服务器的主机名。
- mail.user：确定检索邮件或者发送邮件的用户名。
- mail.protocol.host：确定具体的发送邮件服务器或者接收邮件服务器。有时候发送邮件服务器和接收邮件服务器使用的主机不同，这时候需要详细指定各个协议使用的主机，如果不指定，则使用 mail.host 所确定的主机。
- mail.protocol.user：为登录特定邮件服务器所使用的用户名，如果没有指定，使用 mail.user 所指定的用户。

- **mail.from**: 为邮件指定默认的回复地址, 如果没有指定, 使用 **mail.user** 所指定的用户。

10.2.2 Message

Message 表示单个邮件消息, 其属性包括消息类型、地址信息和所定义的目录结构。但是 **Message** 类是一个抽象类, 必须实现它的一个子类。通常使用 **MimeMessage**, 它是 **Message** 类的一个派生类。

Message 类的主要方法有两部分, 第一部分主要在发送邮件的时候使用, 用于设置邮件的相关信息, 包括邮件的发送者、接收者、主题和发送时间等。这些方法如下:

- **setFrom()**, 用于设置邮件的发送者, 值从 “**mail.user**” 属性中获取, 如果这个值是默认的, 则使用系统属性 “**user.name**”。
- **setFrom (Address address)**, 与上一个方法的作用相同, 值是通过参数确定的, 是 **Address** 的对象, 通常使用其实现者 **InternetAddress** 的对象作为参数。
- **addFrom (Address[] addresses)**, 在已有的邮件发送者中添加其他的邮件发送者, 参数是要添加的邮件发送者的地址。
- **setSubject (String subject)**, 用于设置邮件的标题。
- **setContent (String contentType)**, 用于设置邮件的内容类型。
- **setSentDate (java.util.Date date)**, 用于设置邮件发送的时间。
- **setRecipient (Message.RecipientType type, Address address)**, 用于设置邮件的接收者。有两个参数, 第一个参数是接收者的类型, 第二个参数是接收者。接收者类型可以是 **Message.RecipientType.TO**, **Message.RecipientType.CC** 和 **Message.RecipientType.BCC**, **TO** 表示主要接收人, **CC** 表示抄送人, **BCC** 表示秘密抄送人。接收者与发送者一样, 通常使用 **InternetAddress** 的对象。
- **addRecipient (Message.RecipientType type, Address address)**, 用于添加邮件的接收者, 其参数与 **setRecipient** 方法的基本相同。
- **setRecipients (Message.RecipientType type, Address[] addresses)**, 作用和 **setRecipient** 基本相同, 区别在于该方法可以同时设置多个邮件的接收者。
- **addRecipients (Message.RecipientType type, Address[] addresses)**, 用于添加邮件接收者, 可以同时添加多个接收者。
- **setReplyTo (Address[] addresses)**, 设置邮件的回复地址, 参数用于确定要回复的地址。
- **setText (String text)**, 用于设置邮件的文本, 同时还将邮件的内容类型设置为 **text/plain**。如果邮件的内容类型不是文本的, 则需要通过 **setContent** 方法来设置内容类型。

第二部分用于获取邮件的相关信息, 在接收邮件的时候使用:

- **Flags getFlags()**, 用于获取与邮件相关的标记属性。
- **Folder getFolder()**, 用于获取该邮件所在的文件夹。
- **Address getFrom()**, 用于获取邮件的发送者。
- **int getMessageNumber()**, 用于获取邮件的编号, 该编号是邮件系统设置的。
- **Address[] getAllRecipients()**, 获取邮件的所有接收者。
- **java.util.Date getReceivedDate()**, 用于获取邮件的接收时间。
- **Address[] getRecipients(Message.RecipientType type)**, 用于获取指定接收类型的接收者, 参数用于确定接收者的类型。
- **Address[] getReplyTo()**, 用于获取邮件的回复者, 也就是邮件要给哪些人回复。
- **java.util.Date getSentDate()**, 用于获取邮件的发送时间。

- `java.lang.String getSubject()`，用于获取邮件的主题。

10.2.3 Transport

`Transport` 是一个抽象类，用于邮件的发送，主要的方法有：

- `public static void send(Message msg) throws MessagingException`，用于发送邮件，参数就是要发送的邮件本身，该方法是一个静态方法，不需要实例化对象，可以直接使用。
- `public static void send(Message msg, Address[] addresses) throws MessagingException`，也是用于发送邮件，有两个参数，第一个参数是要发送的邮件本身，第二个参数是邮件发送的目的地。该方法会忽略在邮件中设置的接收者。

10.2.4 InternetAddress

`InternetAddress` 把用户的 E-mail 地址映射为 `Internet` 地址。得到的邮件发送者和接收者通常都是字符串，但是在 `Message` 中确定邮件的接收者和发送者，以及在发送邮件时候使用的都是 `Address` 的对象。`InternetAddress` 是 `Address` 的派生类，可以把字符串转换成 `InternetAddress` 类的对象。

构造函数如下：

`InternetAddress()`，无参数的默认构造函数。

`InternetAddress(java.lang.String address)`，把一个字符串构造成一个 `InternetAddress`，用得比较多。

`InternetAddress(java.lang.String address, java.lang.String personal)`，使用字符串和个人姓名构造一个 `InternetAddress`。

`InternetAddress(java.lang.String address, java.lang.String personal, java.lang.String charset)`，使用字符串和个人姓名构造一个 `InternetAddress` 对象，名字的编码方式是第三个参数确定的。

要设置该对象所表示的 `Internet` 地址，可以通过下面的方法：

`void setAddress(java.lang.String address)`，参数确定了地址。

如果想把 `Internet` 地址转换成字符串，则使用下面的方法：

`java.lang.String toString()`

10.3 WebLogic 中邮件会话的配置

在介绍邮件发送和邮件接收程序之前，需要在 `WebLogic` 中配置邮件会话的相关信息。需要配置所使用的发送邮件服务器和接收邮件服务器。在 `WebLogic` 中的配置过程如下：

- (1) 进入到 `WebLogic Server` 的控制台；
- (2) 在控制台的左下方选择【Domain Structure】→【Service】→【Mail Sessions】；
- (3) 在控制台的左上方点击【Lock & Edit】；
- (3) 在界面的右边点击【New】；
- (4) 在接下来的过程中需要分别输入下面的信息：
 - 会话的名字 (Name): `MailSession`。
 - 会话的 JNDI 名字 (JNDIName): `MailSession`；会话的 JNDI 名字可以和会话的名字相同。
 - 会话相关的属性：

```
mail.pop3.host = 218.25.154.4 （应该写你所使用的邮件服务器的 IP 地址）
mail.transport.protocol = smtp
```

```
mail.user = lixucheng
mail.smtp.host = 218.25.154.6
mail.store.protocol = pop3
```

其中 mail.pop3.host 是所使用的接收邮件服务器, mail.transport.protocol 是发送邮件所使用的协议, mail.user 是发送邮件和接收邮件时候的用户名, mail.smtp.host 是所使用的发送邮件服务器, mail.store.protocol 是检索邮件所使用的协议。

输入完这些信息之后, 点击【Save】, 创建该邮件会话。

- (5) 把邮件会话部署到相应的服务器上: 选择【Targets】页面, 从服务器列表中选择相应的服务器, 然后点击【Save】, 就完成邮件会话的配置了。要让之前的配置起作用, 需要点击左上角的【Activate Changes】(参见图 10.2 和图 10.3)。

10.4 邮件发送示例程序

发送邮件的基本过程如下:

- (1) 得到会话对象
- (2) 构造邮件对象
- (3) 发送邮件

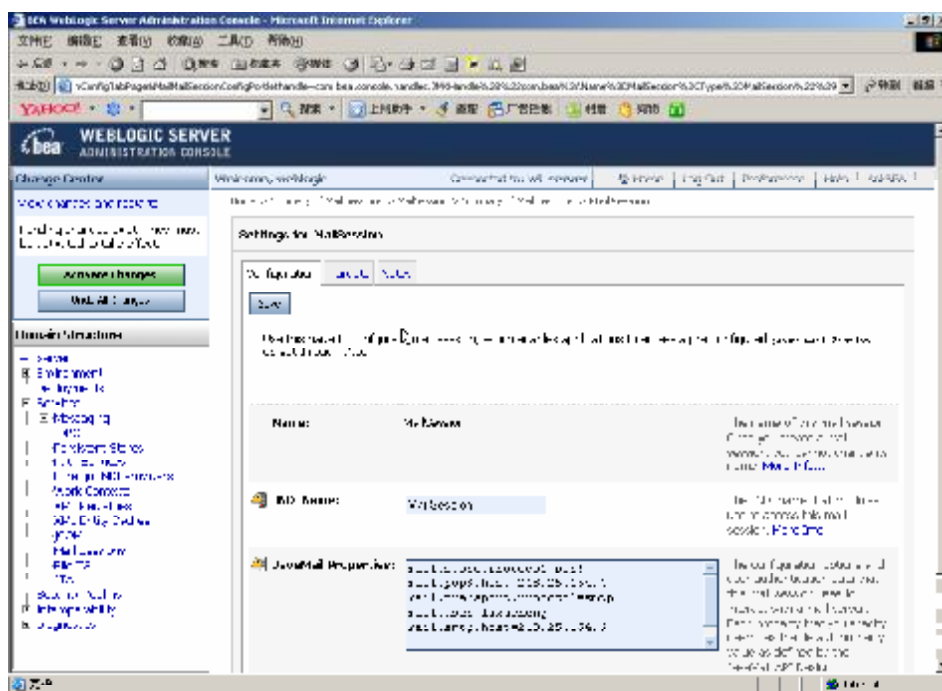


图 10.2 邮件会话的配置

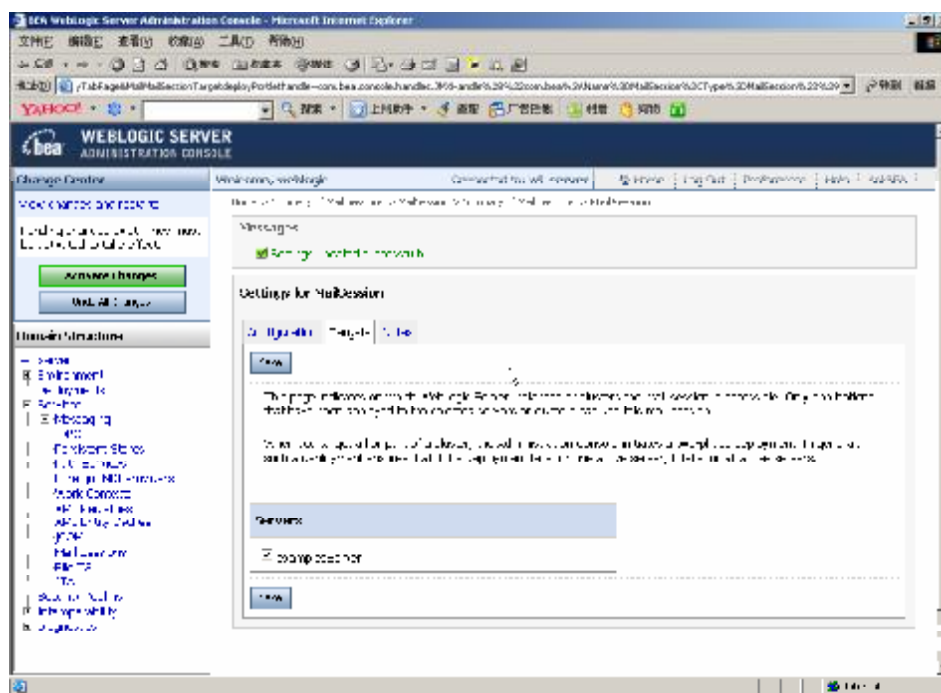


图 10.3 把会话部署到相应的服务器上

10.4.1 得到会话对象

这里使用前面在 WebLogic 中配置好的邮件会话，所以首先要获取这个会话。下面的代码完成获取邮件会话的功能：

```
//获取上下文环境
Context ctx = new InitialContext();
//从 JNDI 中查找会话 MailSession
Session mailSession = (Session) ctx.lookup("MailSession");
```

其中，MailSession 是我们在前面配置的邮件会话中的 JNDI 名字。

如果不使用配置好的邮件会话，可以通过创建 Properties 对象设置相关会话属性，然后，通过 Session.getInstance(properties, null) 创建会话对象。

10.4.2 构造邮件对象

发送一封邮件通常需要确定邮件发送者、邮件接收者、邮件的主题和邮件的内容，有时候还需要发送附件。这里先不考虑附件。其他的条件通过 JSP 界面接收，下面是构造邮件的代码：

```
//获取相关参数
String to = request.getParameter("to");
String subject = request.getParameter("subject");
String from = request.getParameter("from");
String message = request.getParameter("message");
to = new String(to.getBytes("8859_1"));
subject = new String(subject.getBytes("8859_1"));
from = new String(from.getBytes("8859_1"));
//创建消息对象
Message msg = new MimeMessage(mailSession);
```

```
//把邮件地址映射到 Internet 地址上
InternetAddress dest = new InternetAddress(to);

//设置消息内容
msg.setFrom(new InternetAddress(from));
msg.setSubject(subject);
msg.setRecipient(Message.RecipientType.TO, dest);
msg.setContent(message, "text/plain");
```

首先,通过 request 对象的 getParameter 方法获取邮件接收者、邮件发送者、邮件主题和邮件内容。然后,创建 Message 的对象。最后,通过 Message 的 setXXX 方法设置邮件的相关信息。

10.4.3 发送邮件

通过 Transport 的 send(Message msg)方法发送构建好的消息。

```
Transport.send(msg);
```

10.4.4 完整的代码

该实例的代码分为两部分,第一部分是发送邮件的界面,第二部分是发送邮件的处理代码。完整的代码如下:

```
<%@ page import = "java.util.*,
                javax.mail.*,
                javax.mail.internet.*,
                javax.naming.*"
%>
<%@ page contentType = "text/html;charset = gb2312"%>
<html>
<head>
<title>Mail Sender JSP</title>
</head>
<body>

<%

if (request.getMethod().equals("POST")) {

try {

    //获取相关参数
    String to = request.getParameter("to");
    String subject = request.getParameter("subject");
    String from = request.getParameter("from");
    String message = request.getParameter("message");
    to = new String(to.getBytes("8859_1"));
    subject = new String(subject.getBytes("8859_1"));
    from = new String(from.getBytes("8859_1"));
    message = new String(message.getBytes("8859_1"));
    //获取上下文环境
    Context ctx = new InitialContext();

    //从 JNDI 中查找会话 MailSession
```

```
Session mailSession = (Session) ctx.lookup("MailSession");

//创建消息对象
Message msg = new MimeMessage(mailSession);

//把邮件地址映射到 Internet 地址上
InternetAddress dest = new InternetAddress(to);

//设置消息内容
msg.setFrom(new InternetAddress(from));
msg.setSubject(subject);
msg.setRecipient(Message.RecipientType.TO, dest);
msg.setContent(message, "text/plain");

//通过 Transport 类发送消息
Transport.send(msg);

out.println("<h2>到 " + to + "的邮件发送成功!<h2>");

}
catch (Exception e) {

    out.println(e);

}
} else {

%>

<h1>发送邮件!</h1>
<form method = "post" action = "mailsender.jsp">

To :<input type = "text" name = "to" size = 16><p>

From :<input type = "text" name = "from" size = 16><p>

Subject :<input type = "text" name = "subject" size = 16><p>

Message :<input type = "text" name = "message" size = 16>
<p>
<input type = "submit" value = "Submit" name = "Command">

<%
}
%>
</body>
</html>
```

10.4.5 程序的运行结果

直接访问 mailsender.jsp 文件时候的结果，这时候使用的请求方式是 get，所以显示邮件发送的界面，参见图 10.4。

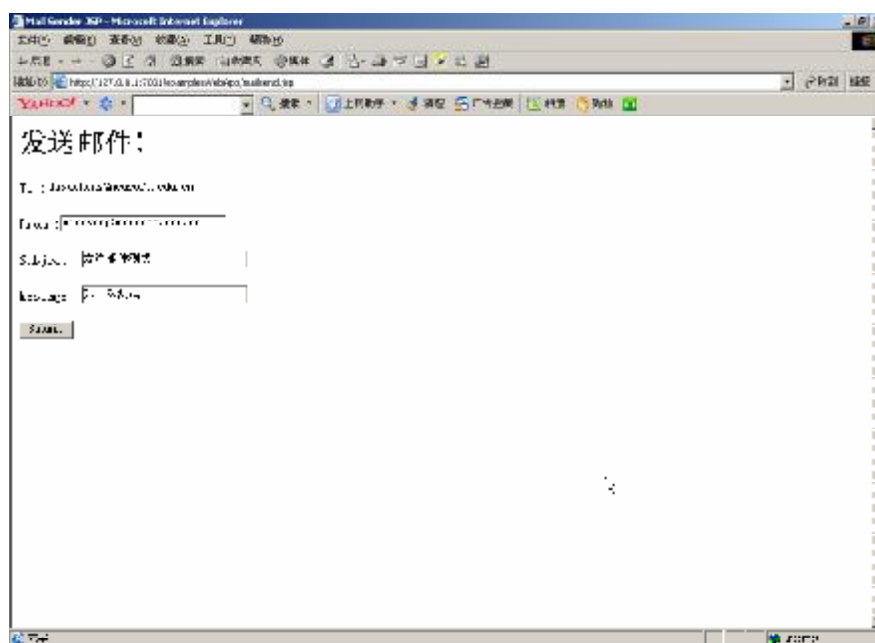


图 10.4 邮件发送的界面

图 10.5 显示了邮件发送成功的界面。填写完邮件的信息之后，提交给服务器，这时候的请求方式是 Post。首先获取用户输入的与邮件相关的信息，然后把邮件发送出去。

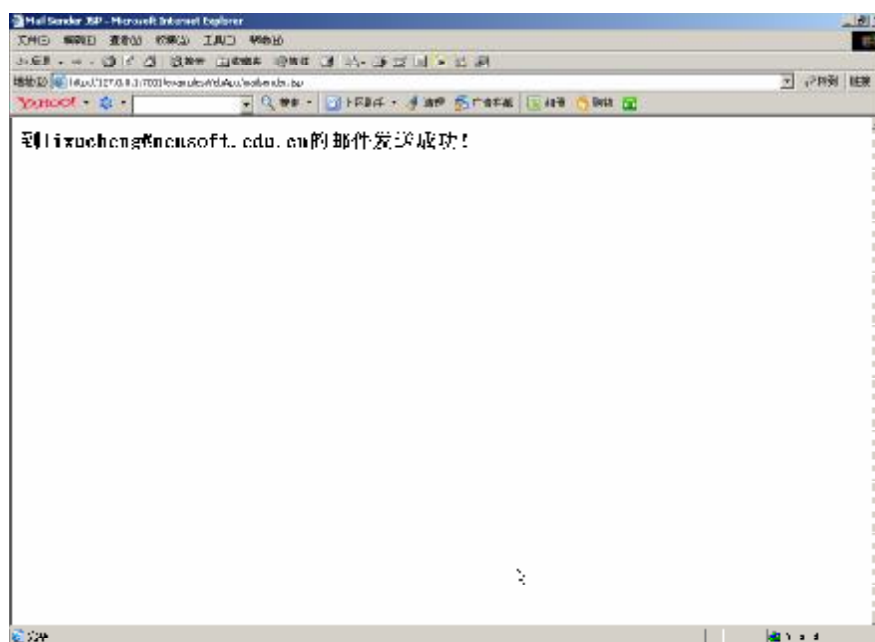


图 10.5 邮件发送成功的界面

10.4.6 发送 HTML 格式的邮件

HTML 格式的邮件发送过程与前面文本类型邮件的发送过程基本相同，不同的是需要读取邮件的 HTML 文件，同时，需要设置邮件内容的格式。

先获取邮件的内容：

```
String content = "";
String file = "source.htm";
//要发送的html文件
String line = "";
FileReader f = new FileReader(file);
BufferedReader b = new BufferedReader(f);
While((line = b.readLine()) != null)
    content += line;
```

然后设置邮件的内容，把邮件的内容添加到邮件对象中：

```
message.setContent(content, "text/html");
```

10.4.7 发送带附件的邮件

我们经常需要发送带附件的邮件。带附件的邮件发送过程与前面介绍的普通邮件发送过程基本相同，不同的是邮件本身的构造比较麻烦。下面我们介绍如何发送带附件的邮件，有些代码和前面是相同的，所以这里只介绍与前面不同的代码。

创建一个带附件的邮件的过程如下：

- 创建 **BodyPart** 对象，该对象表示邮件的主体或者邮件的附件。
- 把所有的 **BodyPart** 对象添加到 **MimeMultipart** 对象中。
- 把 **MimeMultipart** 对象添加到 **MimeMessage** 对象中。

(1) BodyPart 对象的创建

BodyPart 是抽象类，具体使用的是 **MimeBodyPart** 类的对象，而 **MimeBodyPart** 是 **BodyPart** 类的派生类。组成邮件的各部分都是 **MimeBodyPart** 的对象。

首先，创建邮件的主体部分：

```
BodyPart messagebody = new MimeBodyPart();
//创建 BodyPart 类的对象
messagebody.setText("带附件的邮件，注意查看附件!");
//设置邮件的内容
```

这样就创建了邮件的第一部分，也就是邮件的主体部分，下面创建邮件的附件部分。其中 **filename** 表示附件的名字。

```
BodyPart attachment = new MimeBodyPart();
//创建 BodyPart 对象，用于表示邮件的附件
String filename = "filename";
//要发送的邮件附件的名字
DataSource ds = new FileDataSource(filename);
//创建数据源，数据源指向附件文件
attachment.setDataHandler(new DataHandler(ds));
//把附件 BodyPart 对象指向 ds
attachment.setFileName(filename);
//设置附件的文件名
```


这样把邮件的主体部分和邮件的附件部分全部创建完了。

（2）把 Body 对象添加到 MimeMultipart 对象中

邮件的各个组成部分不能独立添加到 `MimeMessage` 对象中，可以添加到其中的只能是 `Multipart` 的对象，而组成邮件的各个部分可以分别添加到 `Multipart` 对象中。下列代码完成的功能是把前面创建好的邮件的两部分添加到 `Multipart` 对象中。

```
Multipart multipart = new MimeMultipart();
//创建 MimeMultipart 对象
multipart.addBodyPart(messagebody);
//把 messagebody 添加到 multipart 对象中
multipart.addBodyPart(attachment);
//把附件添加到 multipart 中
```

（3）把 MimeMultipart 对象添加到 MimeMessage 中

把 `MimeMultipart` 对象添加到 `MimeMessage` 对象中的方法非常简单，与前面介绍的基本相同。

```
message.setContent(multipart);
```

把 `MimeMultipart` 对象添加到 `message` 中之后，邮件的构造就完成了。邮件的发送过程与前面介绍的简单邮件的发送过程相同。

10.5 邮件接收示例程序

邮件接收的基本过程如下：

- 获得邮件会话；
- 创建 `Store` 对象；
- 连接到邮件服务器；
- 得到默认的文件夹；
- 得到所要操作的文件夹；
- 打开文件夹，可以获取与文件夹相关的信息；
- 获取文件夹中的所有邮件；
- 获取邮件相关的信息。

10.5.1 获得邮件会话

与发送邮件相同，接收邮件也需要获取相关的邮件会话。可以使用在 `WebLogic` 中配置好的邮件会话，也可以通过 `Properties` 对象保存邮件会话相关的信息，然后通过该 `Properties` 对象创建邮件会话。下面的代码使用的是 `WebLogic` 中配置好的邮件会话。在运行程序之前，需要保证配置好邮件会话。

```
Context ctx = new InitialContext();
//创建上下文环
Session mailsession = (Session)ctx.lookup("MailSession");
//得到邮件会话
```

10.5.2 创建 Store 对象

要检索邮件服务器上的邮件，需要连接到邮件服务器，与邮件服务器的连接可以通过 `Store` 对象来完成。`Store` 对象是通过 `Session` 的 `getStore` 方法创建的。

```
Store store = mailsession.getStore();
//创建存储对象
```

10.5.3 连接到邮件服务器

连接到邮件服务器，需要知道邮件服务器的地址、连接邮件服务器的用户名以及该用户的口令。连接过程是通过 `Store` 对象的 `connect` 方法完成的。该方法需要三个参数，第一个参数为邮件服务器的地址、域名或者 IP 地址，第二个参数是用户名，第三个参数是口令。

```
store.connect("218.25.154.4","lixucheng","123456");
//连接到邮件服务器
```

10.5.4 得到默认的文件夹

连接到邮件服务器之后，可以得到一个默认的文件夹，通过它可以获得其他的文件夹。默认文件夹是通过 `Store` 对象的 `getDefaultFolder` 方法得到的。文件夹是邮件的存储地方。

```
Folder defaultFolder = store.getDefaultFolder();
//得到默认的文件夹
```

10.5.5 得到所要操作的文件夹

要访问邮件，需要得到邮件所在的文件夹，可以通过默认文件夹得到该文件夹，`defaultFolder` 的 `list` 方法可以得到所有的文件夹。

```
Folder[] allfolder = defaultFolder.list();
```

也可以通过 `Store` 对象的 `getFolder` 方法得到想要的文件夹。该方法需要一个参数，用于指定要打开的文件夹的名字。如果想得到 `INBOX` 文件夹，可以通过下面的方法。

```
Folder folder = store.getFolder("INBOX");
```

10.5.6 打开文件夹，可以获取与文件夹相关的信息

可以通过 `Folder` 对象的 `open` 方法打开文件夹。该方法需要一个参数，指定了文件夹的打开方式。文件夹的打开方式有两种：`READ_ONLY` 和 `READ_WRITE`。如果只是检索邮件信息而不改变其状态或内容，则应该使用 `READ_ONLY` 打开方式。下面的代码是分别打开邮箱中的文件夹，并获得文件夹的相关信息。

```
for(int i = 0;i<allfolder.length;i++)
{
    allfolder[i].open(Folder.READ_ONLY);
    out.println(allfolder[i].getName());
    out.println("\t"+allfolder[i].getMessageCount());
    out.println("\t"+allfolder[i].getNewMessageCount());
    out.println("\t"+allfolder[i].getUnreadMessageCount());
    out.println("<br>");
    allfolder[i].close(false);
}
```

其中，`getName()` 方法用于获取文件夹的名字，`getMessageCount` 得到文件夹中邮件的数量，`getNewMessageCount` 得到文件夹中新邮件的数量，`getUnreadMessageCount` 得到文件夹中未读邮件的数量。读取文件夹的信息之后，需要关闭文件夹。

10.5.7 获取文件夹中的所有邮件

要获取文件夹中的邮件，首先要打开文件夹。获取邮件是通过文件夹的 `getMessages` 方法来完成。

```
defaultFolder = allfolder[0];
```

```
defaultFolder.open(Folder.READ_ONLY);
//打开该文件夹
Message[] messages = defaultFolder.getMessages();
//得到邮件信息
```

10.5.8 获取邮件相关的信息

得到邮件对象 **Message** 之后，就可以获取邮件相关的信息以及邮件的内容了。下面的代码输出了邮件的相关信息：

```
Message message = messages[j];
out.println("<tr><td>" + message.getFrom()[0].toString() + "</td>");
out.println("<td>" + message.getSubject() + "</td>");
out.println("<td>" + message.getSentDate().toLocaleString() + "</td>");
out.println("<td>" + message.getSize() + "</td></tr>");
out.println("<td>" + (String)message.getContent() + "</td></tr>");
```

其中，**getFrom()** 得到邮件的发送者，**getSubject** 得到邮件的主题，**getSentDate** 得到邮件的发送时间，**getSize** 得到邮件的大小，**getContent** 得到邮件的内容。

10.5.9 邮件的状态

邮件可以处于不同的状态，使用系统标记来标识邮件的状态，可以用 **Flags.Flag** 类中的静态成员变量来标记。共有七种：**ANSWERED**，**DELETED**，**DRAFT**，**FLAGGED**，**RECENT**，**SEEN** 和 **USER**。

ANSWERED 表示该邮件已经被回复，当客户端对该信息回复后可以把该邮件标记为 **ANSWERED**。

DELETED 表示该邮件已经被删除，对某个文件夹的删除操作将删除该文件夹中所有标记为 **DELETED** 的邮件。

DRAFT 表示该邮件是草稿，这样就不会被发送。

FLAGGED 标记没有定义明确的语义，客户端可以根据自己的需要来使用该标记。

RECENT 表示该邮件是最近一段时间的，是上次文件夹打开之后到达的邮件。

SEEN 表示该邮件被查看过，当该邮件的内容以某种方式被用户得到后该邮件标记为 **SEEN**，一般情况下调用 **Message** 的 **getInputStream** 和 **getContent** 方法会使得该标识得到设置。

USER 是一个特殊的标志，表示文件夹支持用户自定义的标志。

getFlags() 可以得到邮件的标记，该方法返回的是 **Flags** 对象，要获得所有的标记，通过 **Flags** 对象的 **getSystemFlags** 方法获得具体标记的集合，而具体标记是 **Flags.Flag** 的对象。

isSet 方法用于判断该邮件是否被设置了某个标记，而参数是某个给定的标志。如果邮件被设置了该标记，返回值是 **true**；如果邮件没有被设置该标记，返回值为 **false**。该方法通常用于判断邮件的状态。

setFlag 用于设置邮件的状态，共有两个参数。第一个参数是所设置的标识的类型，第二个参数是值，取值为 **true** 或者 **false**。

setFlags 方法与 **setFlag** 方法的作用大致相同，不同之处在于可以同时设置多个标识。

10.5.10 接收邮件的完整代码

接收邮件的完整代码如下：

```
<%@ page import = "java.util.*,
    javax.mail.*,
    javax.mail.internet.*,"
```

```

        javax.naming.*"
    %>
    <%@ page contentType = "text/html;charset = gb2312"%>
    <!doctype html public "-//w3c/dtd HTML 4.0//en">
    <html>
    <head>
    <title>Mail Sender JSP</title>
    </head>
    <body>
    <%
    try
    {
        Context ctx = new InitialContext();
        //创建上下文环
        Session mailsession = (Session)ctx.lookup("MailSession");
        //得到邮件会话
        Store store = mailsession.getStore();
        //创建存储对象
        store.connect("218.25.154.4", "lixucheng", "123456");
        //连接到邮件服务器
        Folder defaultFolder = store.getDefaultFolder();
        //得到默认的文件夹
        Folder[] allfolder = defaultFolder.list();
        for(int i = 0; i < allfolder.length; i++)
        {
            allfolder[i].open(Folder.READ_ONLY);
            out.println(allfolder[i].getName());
            out.println("\t" + allfolder[i].getMessageCount());
            out.println("\t" + allfolder[i].getNewMessageCount());
            out.println("\t" + allfolder[i].getUnreadMessageCount());
            out.println("<br>");
            allfolder[i].close(false);
        }
        defaultFolder = allfolder[0];
        defaultFolder.open(Folder.READ_ONLY);
        //打开默认文件夹
        Message[] messages = defaultFolder.getMessages();
        //得到邮件信息
        out.println("邮件的数量: " + messages.length);
        if(messages.length > 0)
        {
            out.println("<table>");
            out.println("<tr><td>发送者</td><td>主题</td><td>接收时间</td><td>大
小</td><td>内容</td></hr>");
            for(int j = 0; j < messages.length; j++)
            {
                Message message = messages[j];
                out.println("<tr><td>" + message.getFrom()[0].toString() + "</td>");
                out.println("<td>" + message.getSubject() + "</td>");
                out.println("<td>" + message.getSentDate().toLocaleString() + "</td>");
                out.println("<td>" + message.getSize() + "</td></tr>");
            }
        }
    }
    catch (Exception e)
    {
        out.println("邮件发送失败: " + e.getMessage());
    }
    %>
    </body>
    </html>

```

```
out.println("<td>"+(String)message.getContent()+"</td></tr>");
    }
    out.println("</table>");
}
defaultFolder.close(false);
store.close();
}catch(Exception e)
{
    out.println(e.toString());
    e.printStackTrace();
}

%>
</body>
</html>
```

第 11 章 办公用品申请管理系统

11.1 功能描述

登录之后能够查看可以申请的物品，用户可以选择要申请的物品，另外用户可以看到自己已经选择的物品。

登录成功之后的界面如下：

可申请物品

选择	序号	物品名	规格	价格	数量
<input type="checkbox"/>	1	笔记本	个	3.0	<input type="text" value="1"/>
<input type="checkbox"/>	2	告示贴	个	4.0	<input type="text" value="1"/>
<input type="checkbox"/>	3	双面胶	个	2.5	<input type="text" value="1"/>
<input type="button" value="确定"/>					<input type="button" value="重置"/>

已申请物品

序号	物品名	规格	价格	数量
1	告示贴	个	4.0	7
2	双面胶	个	2.5	3
3	笔记本	个	3.0	5

界面中上半部分是可以申请的物品，下面是已经申请的物品。用户要申请什么物品，只要选中物品前面的复选框，然后修改后面的数量，点击“确定”即可。看起来这个例子比较简单，但是涉及的知识点比较多。

系统用到的表结构如下：

```
DROP TABLE IF EXISTS `thingsforwork`;
CREATE TABLE `thingsforwork` (
  `id` varchar(10) NOT NULL,
  `name` varchar(20) default NULL,
  `spec` varchar(20) default NULL,
  `price` float default NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB;
```

```
DROP TABLE IF EXISTS `usertable`;
CREATE TABLE `usertable` (
```

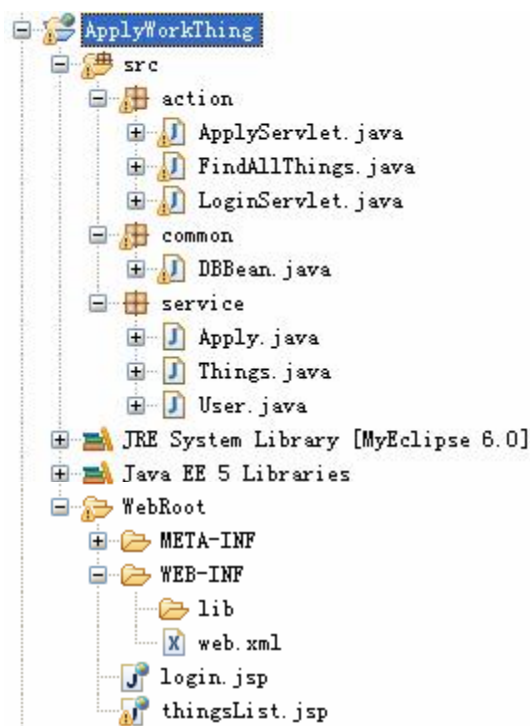
```

`userid` varchar(20) NOT NULL,
`username` varchar(20) default NULL,
`usertype` varchar(10) default NULL,
`department` varchar(20) default NULL,
`user_en` varchar(20) default NULL,
`pass` varchar(20) default NULL,
PRIMARY KEY (`userid`)
) ENGINE=InnoDB;
DROP TABLE IF EXISTS `apply`;
CREATE TABLE `apply` (
  `userid` varchar(20) default NULL,
  `id` varchar(10) default NULL,
  `quantity` int(11) default NULL,
  `applydate` date default NULL,
  `state` varchar(10) default NULL
) ENGINE=InnoDB;

```

11.2 涉及的文件及关系

工程文件结构如下；



11.3 控制器文件

11.3.1 ApplyServlet.java

```
package action;
```

```
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

import service.Apply;
import service.Things;
import service.User;

public class ApplyServlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        // 获取申请信息
        ArrayList<Apply> list=new ArrayList<Apply>();
        Things things = new Things();
        HttpSession session = request.getSession(true);
        User user = (User)session.getAttribute("user");

        // 得到申请的物品的 ID
        String[] ids = request.getParameterValues("applylist");
        if(ids!=null){
            for(String id:ids){
                Apply tempApply = new Apply();
                Things tempThings = things.findById(id);
                int quantity = 0;
                try{
                    // 得到申请的每种物品的数量
                    quantity = Integer.parseInt(request.getParameter(id));
                    tempApply.setQuantity(quantity);
                    tempApply.setThings(tempThings);
                    tempApply.setUser(user);
                }catch(Exception e){e.printStackTrace();}
                list.add(tempApply);
            }
        }
    }
}
```



```
// 处理申请
new Apply().process(list);

response.sendRedirect("thingsList");
// 定义跳转文件
// RequestDispatcher rd=request.getRequestDispatcher("thingsList");
// 完成重定向
// rd.forward(request,response);
}

public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    doGet(request,response);
}

}
```

11.3.2 FindAllThings.java

```
package action;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

import service.Apply;
import service.Things;
import service.User;

public class FindAllThings extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        // 获取用户信息
        HttpSession session = request.getSession();
        User user = (User)session.getAttribute("user");
```

```
// 创建模型对象
Things thing = new Things();

// 调用业务方法获取物品信息
ArrayList<Things> list = thing.getAll();

ArrayList<Apply> applies = Apply.findAllApplyByUser(user.getName_en(),"0");

// 存储信息
request.setAttribute("thinglist",list);
request.setAttribute("applies",applies);

// 获取 Dispatcher 对象
RequestDispatcher dispatcher = request.getRequestDispatcher("thingsList.jsp");

// 完成跳转
dispatcher.forward(request,response);
}

public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    doGet(request,response);
}

}
```

11.3.3 LoginServlet.java

```
package action;

import javax.servlet.*;
import javax.servlet.http.*;

import service.User;

import java.io.*;

public class LoginServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request,HttpServletResponse response)
        throws IOException,ServletException    {

        // 获取用户提交的信息
```

```
String userid = request.getParameter("userid");
String userpass = request.getParameter("userpass");

// 创建 JavaBean 对象
User ub = new User();

// 转向的文件
String forward=null;

User user = ub.findById(userid);

// 无效
if(user == null || user.getPass().equals(userpass)){
    forward="login.jsp";
}
// 有效
else{
    HttpSession session = request.getSession(true);
    session.setAttribute("user",user);
    forward="thingsList";
}

// 定义跳转文件
RequestDispatcher rd=request.getRequestDispatcher(forward);
// 完成重定向
rd.forward(request,response);
}

public void doPost(HttpServletRequest request,HttpServletResponse response)
    throws IOException,ServletException {
    doGet(request,response);
}
}
```

11.4 数据库访问 Bean

11.4.1 DBBean.java

```
package common;

import java.sql.*;

public class DBBean {
    private Connection con = null;
    private Statement stmt = null;
    private ResultSet rs;

    public DBBean() {
```

```
String className = "com.mysql.jdbc.Driver";
String url = "jdbc:mysql://127.0.0.1:3306/jspbook";
String username = "root";
String userpass = "root";

ResultSet rs = null;
try {
    Class.forName(className);
    con = DriverManager.getConnection(url, username, userpass);

} catch (Exception e) {
    System.out.println(e.toString());
}

// 执行有结果集返回的sql语句
public ResultSet executeQuery(String sql) throws Exception {
    if (con == null)
        throw new Exception("没有连接对象可用");
    // 创建语句对象
    stmt = con.createStatement();
    rs = stmt.executeQuery(sql);
    return rs;
}

// 执行更新语句
public int executeUpdate(String sql) throws Exception {
    if (con == null)
        throw new Exception("没有连接可用");
    // 创建语句对象
    stmt = con.createStatement();
    // 执行sql语句
    return stmt.executeUpdate(sql);
}

public void close() {
    try {
        if (rs != null)
            rs.close();
    } catch (Exception e) {
    }
    try {
        stmt.close();
    } catch (Exception e) {
    }
    try {
        con.close();
    } catch (Exception e) {
    }
}
}
```

11.5 业务类

11.5.1 Apply.java

```
package service;

import java.sql.ResultSet;
import java.util.ArrayList;

import common.DBBean;

public class Apply {
    private Things things;
    private User user;
    private int quantity;

    public Things getThings() {
        return things;
    }

    public void setThings(Things things) {
        this.things = things;
    }

    public User getUser() {
        return user;
    }

    public void setUser(User user) {
        this.user = user;
    }

    public int getQuantity() {
        return quantity;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }

    public void process(ArrayList<Apply> list) {
        DBBean db = new DBBean();
        try {
            for (Apply temp : list) {
```

```
String sql;
sql = "select * from apply where userid="
      + temp.getUser().getName_en() + " and id="
      + temp.getThings().getId() + " and state='0'";
ResultSet rs = db.executeQuery(sql);
if (rs.next())
    sql = "update apply set quantity="
          + (rs.getInt(3) + temp.getQuantity())
          + " where userid=" + temp.getUser().getName_en()
          + " and id=" + temp.getThings().getId()
          + " and state='0'";
else
    sql = "insert into apply values("
          + temp.getUser().getName_en() + ", "
          + temp.getThings().getId() + ", "
          + temp.getQuantity() + ",now(),'0')";
db.executeUpdate(sql);
}
} catch (Exception e) {
    e.printStackTrace();
}
}

public static ArrayList<Apply> findAllApplyByUser(String user, String state) {
    DBBean db = new DBBean();
    ArrayList<Apply> applys = new ArrayList<Apply>();
    try {
        String sql = "select * from apply where userid=" + user
            + " and state=" + state + "";
        ResultSet rs = db.executeQuery(sql);
        while (rs.next()) {
            Apply temp = new Apply();
            temp.setUser(User.findById(user));
            temp.setQuantity(rs.getInt(3));
            temp.setThings(new Things().findById(rs.getString(2)));
            applys.add(temp);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    return applys;
}
}
```

11.5.2 Things.java

```
package service;
```

```
import java.sql.ResultSet;
```

```
import java.util.ArrayList;
```

```
import common.DBBean;
```

```
public class Things {  
    private String id;  
    private String name;  
    private String spec;  
    private float price;  
    private int quantity;  
    public String getId() {  
        return id;  
    }  
    public void setId(String id) {  
        this.id = id;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public String getSpec() {  
        return spec;  
    }  
    public void setSpec(String spec) {  
        this.spec = spec;  
    }  
    public float getPrice() {  
        return price;  
    }  
    public void setPrice(float price) {  
        this.price = price;  
    }  
    public Things findById(String id){  
        DBBean db = new DBBean();  
        try {  
            ResultSet rs = db.executeQuery("select * from thingsforwork where id='"+id+"'");  
            if(rs.next()){
```

```
        Things things = new Things();
        things.setId(rs.getString(1));
        things.setName(new String(rs.getString(2).getBytes("8859_1")));
        things.setSpec(new String(rs.getString(3).getBytes("8859_1")));
        things.setPrice(rs.getFloat(4));
        return things;
    }
} catch (Exception e) {
    e.printStackTrace();
}
return null;
}

public ArrayList<Things> getAll(){
    ArrayList<Things> list = new ArrayList<Things>();
    DBBean db = new DBBean();
    try {
        ResultSet rs = db.executeQuery("select * from thingsforwork");
        while(rs.next()){
            Things temp = new Things();
            temp.setId(rs.getString(1));
            temp.setName(new String(rs.getString(2).getBytes("8859_1")));
            temp.setSpec(new String(rs.getString(3).getBytes("8859_1")));
            temp.setPrice(rs.getFloat(4));
            list.add(temp);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    return list;
}

public int getQuantity() {
    return quantity;
}

public void setQuantity(int quantity) {
    this.quantity = quantity;
}
}
```

11.5.3 User.java

```
package service;
```

```
import java.sql.ResultSet;
```



```
import common.DBBean;

public class User {
    private String userid;
    private String username;
    private String pass;
    private String name_en;
    private String usertype;
    private String department;
    public String getUserid() {
        return userid;
    }
    public void setUserid(String userid) {
        this.userid = userid;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPass() {
        return pass;
    }
    public void setPass(String pass) {
        this.pass = pass;
    }
    public String getName_en() {
        return name_en;
    }
    public void setName_en(String name_en) {
        this.name_en = name_en;
    }
    public String getUstertype() {
        return usertype;
    }
    public void setUstertype(String usertype) {
        this.usertype = usertype;
    }
    public String getDepartment() {
        return department;
    }
    public void setDepartment(String department) {
```

```
        this.department = department;
    }

    // 登录
    public boolean check(String user_en,String pass){
        boolean validate = false;
        String sql = "select * from usertable where user_en='"+user_en+"' and pass='"+pass+"'";
        DBBean db = new DBBean();
        try{
            ResultSet rs = db.executeQuery(sql);
            if(rs.next())
                validate = true;
        }catch(Exception e){System.out.println(e.toString());}
        return validate;
    }

    public static User findById(String user_en){
        String sql = "select * from usertable where user_en='"+user_en+"'";
        DBBean db = new DBBean();
        try{
            ResultSet rs = db.executeQuery(sql);
            if(rs.next()){
                User user = new User();
                user.setUserid(rs.getString(1));
                user.setName_en(user_en);
                user.setUsername(rs.getString(2));
                user.setUserType(rs.getString(3));
                user.setDepartment(rs.getString(4));
                user.setPass(rs.getString(5));
                return user;
            }
        }catch(Exception e){System.out.println(e.toString());}
        return null;
    }
}
```

11.6 界面

11.6.1 login.jsp

```
<%@ page contentType="text/html; charset=gb2312"%>
<html>
    <head>
        <title>用户登陆</title>
    </head>
    <body>
        <h2>用户登录</h2>
```

```

<form name="form1" action="login" method="post">
  用户名: <input type="text" name="userid"> <br>
  口令: <input type="password" name="userpass"><br>
    <input type="reset" value="重置">
    <input type="submit" value="提交"><br>
</form>
</body>
</html>

```

11.6.2 thingsList.jsp

```

<%@ page contentType="text/html;charset=gb2312" isELIgnored="false"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"
prefix="c"%>
可申请物品
<form action="apply" method="post">
  <table align=center>
    <tr>
      <th>
        选择
      </th>
      <th>
        序号
      </th>
      <th>
        物品名
      </th>
      <th>
        规格
      </th>
      <th>
        价格
      </th>
      <th>
        数量
      </th>
    </tr>
    <c:forEach items="${thinglist}" var="thing" varStatus="index">
      <tr>
        <td>
          <input type="checkbox" name="applylist"
value="${thing.id}">
        </td>
        <td>
          ${index.index+1}
        </td>
        <td>
          ${thing.name}
        </td>
        <td>
          ${thing.spec}
        </td>
        <td>
          ${thing.price}
        </td>
      </tr>
    </c:forEach>
  </table>
</form>

```

```

        <td>
            <input type="text" name="${thing.id}" value="1">
        </td>
    </tr>
</c:forEach>
<tr>
    <td>
        <input type="submit" value="确定">
    </td>
    <td></td>
    <td></td>
    <td></td>
    <td></td>
    <td>
        <input type="reset" value="重置">
    </td>
</tr>
</table>
</form>
<c:if test="${not empty applys}">
    <hr>
    已申请物品
    <table align=center>
        <tr>
            <th>
                序号
            </th>
            <th>
                物品名
            </th>
            <th>
                规格
            </th>
            <th>
                价格
            </th>
            <th>
                数量
            </th>
        </tr>
        <c:forEach items="${applys}" var="apply" varStatus="status">
            <tr>
                <td>
                    ${status.index+1}
                </td>
                <td>
                    ${apply.things.name}
                </td>
                <td>
                    ${apply.things.spec}
                </td>
                <td>
                    ${apply.things.price}
                </td>
            </tr>
        </c:forEach>
    </table>
</if>

```

```

        <td>
            ${apply.quantity}
        </td>
    </tr>
</c:forEach>
</table>
</c:if>

```

11.7 配置文件

11.7.1 web.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <servlet>
        <description>This is the description of my J2EE
component</description>
        <display-name>This is the display name of my J2EE
component</display-name>
        <servlet-name>LoginServlet</servlet-name>
        <servlet-class>action.LoginServlet</servlet-class>
    </servlet>
    <servlet>
        <description>This is the description of my J2EE
component</description>
        <display-name>This is the display name of my J2EE
component</display-name>
        <servlet-name>FindAllThings</servlet-name>
        <servlet-class>action.FindAllThings</servlet-class>
    </servlet>
    <servlet>
        <description>This is the description of my J2EE
component</description>
        <display-name>This is the display name of my J2EE
component</display-name>
        <servlet-name>ApplyServlet</servlet-name>
        <servlet-class>action.ApplyServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>LoginServlet</servlet-name>
        <url-pattern>/login</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>FindAllThings</servlet-name>
        <url-pattern>/thingsList</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>ApplyServlet</servlet-name>
        <url-pattern>/apply</url-pattern>

```

```
</servlet-mapping>
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
</web-app>
```

第 12 章 自动信息收集系统

¹自动信息收集系统用于收集Excel形式的信息，管理员上传Excel文档模板，系统根据Excel模板生成Web形式的输入界面，用户通过界面提交信息，这些信息最终会写入Excel文件，管理员可以下载以Excel形式收集好的信息。

12.1 需求描述

功能如图 12.1 所示。

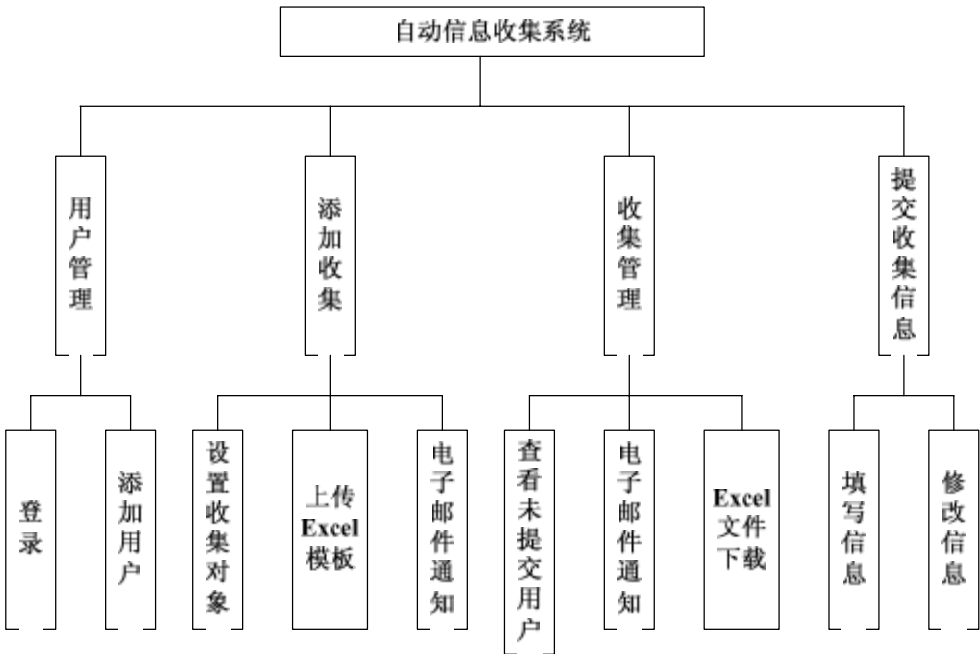


图 12.1 系统功能模块图

具体功能需求描述如下：

（1）用户管理功能

管理员及用户需要登录才能执行相关操作或管理；管理员可以添加用户。

（2）添加收集功能

管理员登录后可以根据需要添加新的收集，添加新的收集时需指定被收集的用户并上传 Excel 模板。上传的 Excel 文件的格式如表 12.1 所示，在 Excel 表格

¹ 本章内容来自 04 级学生郝嘉的论文，已经得到该学生的允许，在此表示感谢。

第一行填入要收集的列名，在应用时管理员可以根据实际的列名替换表中的第一行。确认收集后管理员通过电子邮件通知需要提交收集的用户。

表 12.1 Excel 文件格式样例

编号	姓名	电话	地址

(3) 收集管理功能

管理员可以查看没有提交收集的用户并可以通过电子邮件通知该用户；管理员可以下载 Excel 文件汇总结果。

(4) 提交收集信息功能

用户在查看收集列表的同时点击需要自己完成的收集链接，系统根据 Excel 模板的内容动态生成表单供用户填写。用户还可以对和自己有关且已经填写完的收集进行修改。

12.2 设计

包括数据库设计、功能设计和界面设计。

12.2.1 数据库设计

根据需求调研结果确定本系统的数据库结构，如图 12.2 所示。

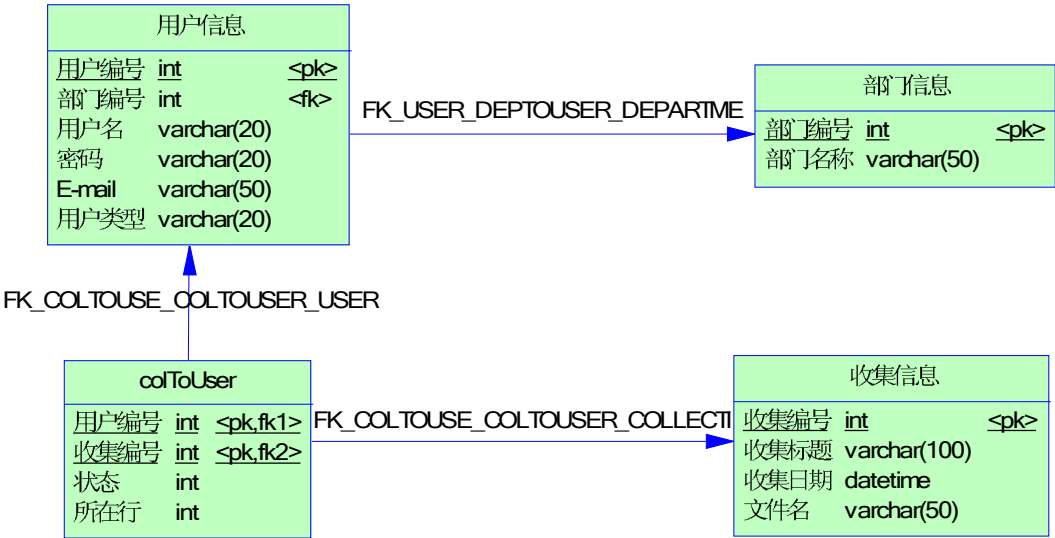


图 12.2 系统数据库结构图

(1) 用户表 (user)，用于存放管理员和用户的信息，如表 12.1 所示。

表 12.1 用户表 (user)

字段名	数据类型	长度	是否主键	是否外键	是否为空	含义
userid	int	4	是	是	否	用户编号
departmentno	int	4	否	否	否	部门编号
username	varchar	20	否	否	否	用户名
password	varchar	20	否	否	否	密码
email	varchar	20	否	否	否	E-mail
usertype	varchar	20	否	否	否	用户类型

(2) 收集表 (collection)，用于存放收集项的信息，如表 12.2 所示。

表 12.2 收集表 (collection)

字段名	数据类型	长度	是否主键	是否外键	是否为空	含义
collectionno	int	4	是	是	否	收集编号
collectiontopic	varchar	100	否	否	否	收集标题
collectiondate	datetime	8	否	否	否	收集日期
filename	varchar	50	否	否	否	文件名

(3) 部门表 (department)，用于存放部门信息，如表 12.3 所示。

表 12.3 部门表 (department)

字段名	数据类型	长度	是否主键	是否外键	是否为空	含义
departmentno	int	4	是	是	否	部门编号
departmentname	varchar	30	否	否	否	部门名称

departmentno	int	4	是	否	否	部门 编号
departmentname	varchar	50	否	否	否	部门 名称

（4）用户收集联系表（colToUser），用于存放每次收集所涉及的用户填写信息的状态，如表 12.4 所示。

表 12.4 用户收集联系表（colToUser）

字段名	数据类型	长度	是否主键	是否外键	是否为空	含义
userno	int	4	是	否	否	用户 编号
collectionno	int	4	是	否	否	收集 编号
status	int	4	否	否	否	状态
inrow	int	4	否	否	否	所在 行

12.2.2 系统的功能结构设计

根据需求调研结果确定本系统主要包括：用户管理，添加收集，收集管理和提交收集信息四大模块。

用户管理功能

（1）登录：管理员和用户需要登录才能执行相关操作或管理。具体流程如图 12.3 所示。

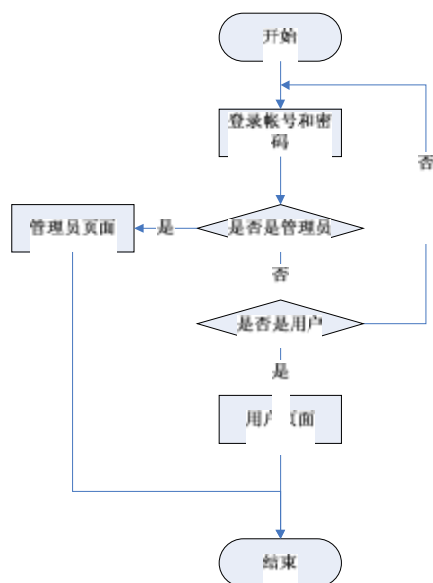


图 12.3 登录流程图

(2) 添加用户：管理员可以添加用户。

添加收集功能

管理员登录后可以根据需要添加新的收集，添加新的收集时需指定标题和被收集的用户。系统使用 SmartUpload 控件上传 Excel 模板，同时将此次收集的编号、标题、日期和上传的 Excel 文件名存入收集表（collection）中，将被指定的用户编号、此次收集的编号、状态（默认为“0”，也就是未填写）和填写内容所在的行数（默认为“0”）存入用户收集联系表（colToUser）中。确认收集后管理员通过电子邮件通知需要提交收集的用户。这里电子邮件的发送是利用 JavaMail 实现的。添加收集操作流程如图 12.4 所示。

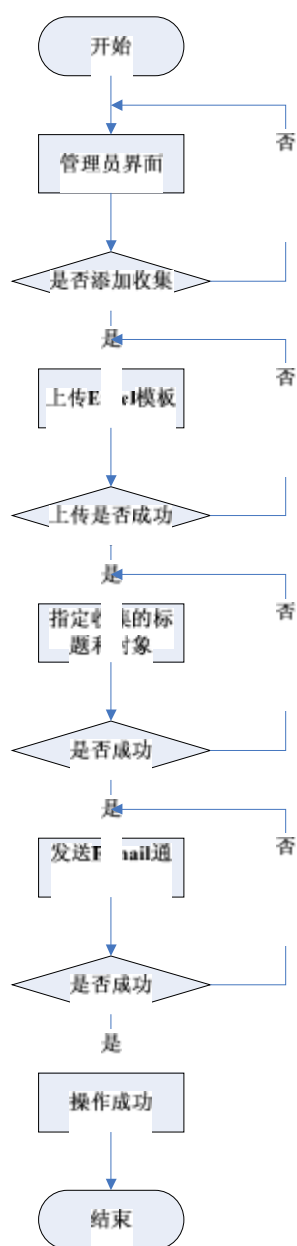


图 12.4 添加收集流程图

收集管理功能

(1) 管理员点击查看没有提交收集用户的链接，系统根据收集编号查询数据库得到与该收集有关且状态为“0”的用户并将这些用户一一列出。

（2）在（1）所得到的未提交信息的用户表中，每个用户都对应着一个发送 E-mail 的链接，管理员点击这些链接就可以通知那些没有提交信息的用户。

（3）管理员点击“查看旧的收集”链接可以得到一个收集列表，这里列出了所有的收集项，每个收集后都有“下载”链接，管理员点击这个链接就可以随时下载已填写的 Excel 文件。

收集管理流程如图 12.5 所示。

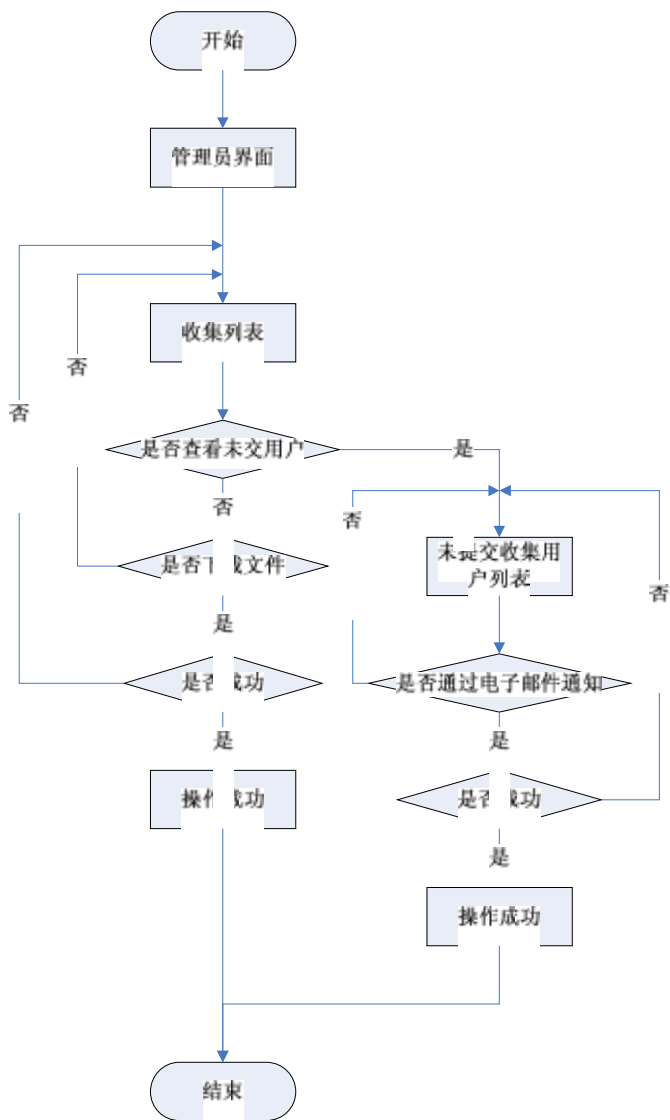


图 12.5 收集管理流程图

提交收集信息功能

在用户查看收集列表时，系统根据用户收集联系表（colToUser）判断出该用户是否已经提交了此次收集，如果是，则在收集标题的后面显示“修改”链接；否则显示“填写”链接。

（1）填写信息：用户填写信息，系统通过利用 Java Excel API 编写好的 **JavaBean** 读取 Excel 文件的行数和列数，并将内容存在一个二维字符串数组中。根据读取的第一行中的列名，生成对应的输入文本框，这样就动态的生成了一张 **HTML** 表单。用户提交的信息将存在一个一维的字符串数组中，然后调用 **JavaBean** 中的写入 Excel 的方法将用户提交的内容保存在 Excel 文件中。

（2）修改信息：用户修改已经提交的信息，系统根据用户收集联系表（colToUser）判断出该用户填写的内容在 Excel 文件中的位置，同时读取填写的内容，同（1）中一样的生成一张已经填写好的 **HTML** 表单，这样用户可以看到自己以前填写过的内容。用户提交后系统用新的内容替换掉旧的内容，这样就完成了修改。

提交收集信息流程如图 12.6 所示。

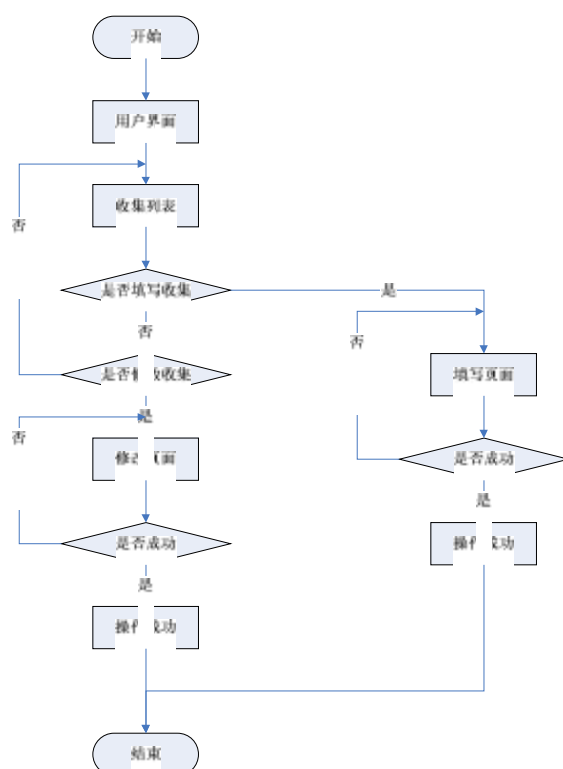


图 12.6 提交收集信息流程图

12.2.3 用户界面设计

系统用户界面设计。

(1) 管理员及用户登录界面，如图 12.7 所示。

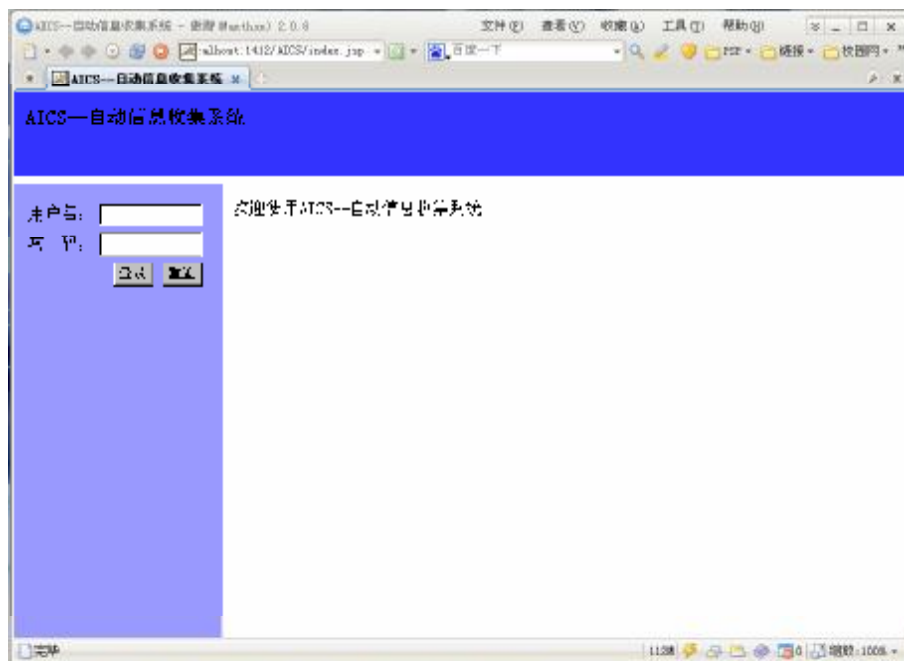


图 12.7 管理员及用户登录界面

(2) Excel 模板上传界面。如图 12.8 所示。

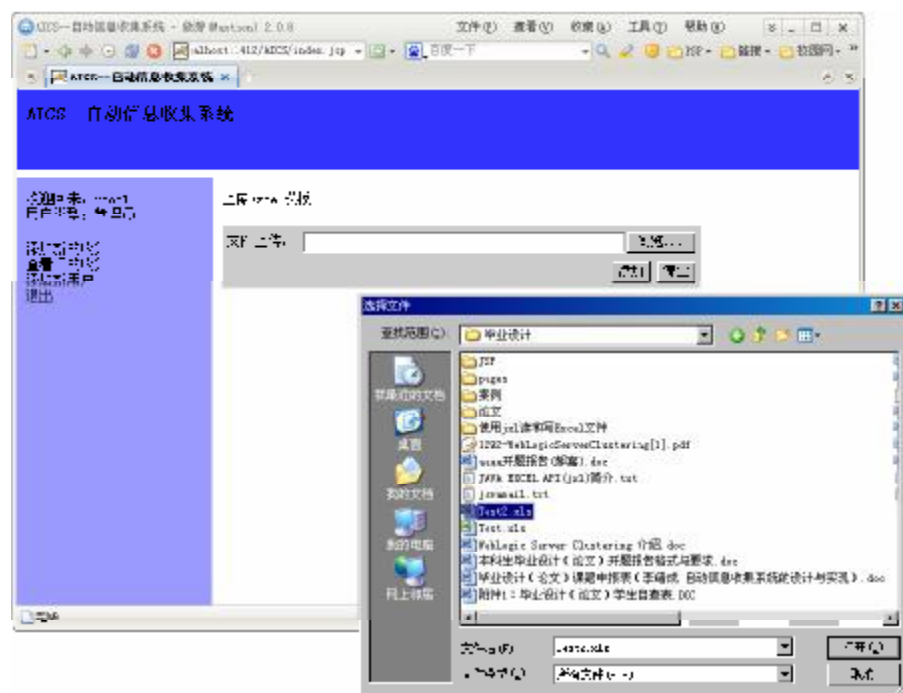


图 12.8 Excel 模板上传界面

(3) 管理员查看收集列表界面。如图 12.9 所示。



图 12.9 管理员查看收集列表界面

(4) 用户在动态生成的 HTML 表单中修改已经填写的收集信息界面。如图 12.10 所示。



图 12.10 用户在动态生成的 HTML 表单中修改已经填写的收集信息界面

12.3 核心代码

12.3.1 Excel 文件读取与写入 JavaBean

```
public String[][] jExcelRead(String fileName,int sheetNumber) {  
    .....  
    try {  
        is=new FileInputStream(path+fileName);//写入到FileInputStream  
        wb=Workbook.getWorkbook(is);//得到工作簿  
        sheet=wb.getSheet(sheetNumber);//得到工作簿的一个工作表  
        rowCount=sheet.getRows();  
        columnCount=sheet.getColumns();  
        content=new String[rowCount][columnCount];//根据行列数创建一个二维数组  
        for(int i=0;i<rowCount;i++) {//循环将内容复制到二维数组中  
            for(int j=0;j<columnCount;j++) {  
                cell=sheet.getCell(j,i);  
                content[i][j]=cell.getContents();  
            }  
        }  
        wb.close();//关闭工作簿  
        is.close();//关闭输入流  
    }  
}
```

```

.....
return content;
}
public void jExcelWrite(String fileName,int sheetNumber,int rowCount,String[] content) {
.....
try {
    is=new FileInputStream(path+fileName);
    wb=Workbook.getWorkbook(is);
    os=new FileOutputStream(path+fileName);//输出到FileOutputStream
    ww=Workbook.createWorkbook(os,wb);//创建可写工作簿
    ws=wb.getSheet(sheetNumber);//得到一个可写工作表
    columnCount=ws.getColumns();
    for(int i=0;i<columnCount;i++) { //循环将数组内容写入到单元格
        Label label=new Label(i,rowCount,content[i]);
        ws.addCell(label);
    }
    ww.write();//写入
    ww.close();//关闭工作簿
    is.close();//关闭输入流
    os.close();//关闭输出流
    }
.....
}

```

12.3.2 根据 Excel 文件内容动态生成 HTML 表单：

```

.....
HttpSession hs=(HttpSession)request.getSession(true);
String filename=request.getParameter("filename");//得到文件名
int collectionno=Integer.parseInt(request.getParameter("collectionno"));//得到收集编号
String fillinType=request.getParameter("fillinType");//得到填写类型
hs.setAttribute("fillinType",fillinType);
JExcelBean jeb=new JExcelBean();
String[][] content=jeb.jExcelRead(filename,0);//得到文件内容
int rowCount=content.length;//得到行数
int columnCount=content[0].length;//得到列数
hs.setAttribute("content",content);
//根据填写类型的不同分别处理
if(fillinType.equals("fillin")) {
    hs.setAttribute("rowCount",rowCount);
}

```

```

if(fillinType.equals("update")) {
    int inrow=0;
    UserInfoBean uib=(UserInfoBean)hs.getAttribute("uib");//得到用户
    int userno=uib.getUserno();//得到用户编号
    try {
        DBBean dbb=new DBBean();
        ResultSet rs=dbb.executeQuery("select inrow from colToUser where
userno="+userno+" and collectionno="+collectionno);//根据用户编号和收集编号得到写入的行
        if(rs.next()) {
            inrow=rs.getInt("inrow");
        }
        hs.setAttribute("rowCount",inrow);
        dbb.close();
    }
    .....
}
.....

```

12. 3. 3 数据库访问 JavaBean:

```

public ResultSet executeQuery(String sql) { //数据库查询
    ResultSet rs=null;
    try {
        conn=DriverManager.getConnection(connStr,username,password);
        stmt=conn.createStatement();
        rs=stmt.executeQuery(sql);
    }
    .....
}
public int executeUpdate(String sql) { //数据库更新
    int result=0;
    try {
        conn=DriverManager.getConnection(connStr,username,password);
        stmt=conn.createStatement();
        result=stmt.executeUpdate(sql);
    }
    .....
}

```

12. 3. 4 导航栏（JSTL 和 EL 的应用）:

```

<c:choose>
    <c:when test="${(sessionScope.uib.usertype) eq '管理员'}">
        欢迎回来, ${sessionScope.uib.username}<br>
        用户类型: ${sessionScope.uib.usertype}<p>

```

```
<a href="addCollection.jsp" target="mainFrame">添加新的收集</a><br>
<a href="ShowCollection" target="mainFrame">查看旧的收集</a><br>
<a href="userRegister.jsp" target="mainFrame">添加新用户</a><br>
<a href="Logout" target=_top>退出</a><br>
</c:when>
<c:when test="${(sessionScope.uib.usertype) eq '用户'}">
欢迎回来，${sessionScope.uib.username}<br>
用户类型：${sessionScope.uib.usertype}<p>
<a href="ShowCollection" target="mainFrame">查看收集列表
  <c:if test="${sessionScope.notFinishedCollection ne '0'}">
    (<font color="red">${sessionScope.notFinishedCollection}</font>)
  </c:if>
</a><br>
  <a href="Logout" target=_top>退出</a><br>
</c:when>
<c:otherwise>
.....
</c:otherwise>
</c:choose>
完整代码参考资源包。
```