

# 基于 J2EE 的 Ajax 宝典

## 目录

### 第 1 部分 初识 Ajax..... 1

#### 第 1 章 Ajax 入门..... 2

##### 1.1 重新思考 Web 应用..... 2

###### 1.1.1 应用系统的发展史..... 2

###### 1.1.2 传统 Web 应用的优势和缺点..... 4

##### 1.2 重新设计 Web 应用..... 5

###### 1.2.1 RIA 应用..... 5

###### 1.2.2 异步发送请求并避免等待..... 7

###### 1.2.3 使用 Ajax..... 7

##### 1.3 Ajax 简介... 8

###### 1.3.1 Ajax 的工作方式..... 8

###### 1.3.2 XMLHttpRequest..... 9

###### 1.3.3 JavaScript 语言..... 10

###### 1.3.4 HTML 页面的 DOM..... 10

###### 1.3.5 数据交换和显示..... 11

##### 1.4 Ajax 的基本特征..... 12

##### 1.5 Ajax 的替代技术..... 13

###### 1.5.1 Sun 的 Java Web Start 技术..... 13

###### 1.5.2 Microsoft 的 ClickOnce 技术..... 13

###### 1.5.3 基于 Flash 的 Flex..... 14

##### 1.6 搭建开发运行环境.... 15

###### 1.6.1 本书的 Ajax 环境..... 15

1.6.2	Windows 下 JDK 的安装.....	15
1.6.3	Linux 下 JDK 的安装.....	19
1.6.4	安装 Tomcat 服务器.....	20
1.6.5	配置 Tomcat 的服务端口.....	22
1.6.6	进入控制台.....	22
1.6.7	部署 Web 应用.....	24
1.6.8	配置 Tomcat 的数据源.....	25
1.6.9	安装 Ant 工具.....	28
1.6.10	安装 Eclipse 开发环境.....	28
1.6.11	在线安装.....	29
1.6.12	手动安装.....	30
1.7	小结.....	31

## 第 2 章 Ajax 初体验..... 32

2.1	Ajax 带来的优势.....	32
2.2	传统的 JSP 聊天室.....	34
2.2.1	实现业务逻辑组件.....	35
2.2.2	实现控制器.....	38
2.2.3	实现视图.....	40
2.2.4	JSP 聊天室的问题.....	41
2.3	Ajax 聊天室.....	42
2.3.1	异步发送请求.....	42
2.3.2	解决多余刷新的问题.....	44
2.3.3	解析服务器响应.....	47
2.3.4	何时发送请求.....	48
2.3.5	Ajax 聊天室的特点.....	52
2.4	传统 Web 应用与 Ajax 的对比.....	52
2.5	小结.....	53

第 2 部分 Ajax 基本技术..... 55

第 3 章 JavaScript 关键语法详解..... 56

3.1 JavaScript 简介..... 56

3.2 数据类型和变量..... 57

3.2.1 定义变量的方式..... 57

3.2.2 类型转换..... 58

3.2.3 变量..... 60

3.3 基本数据类型..... 62

3.3.1 数字类型..... 62

3.3.2 字符串类型.... 67

3.3.3 布尔类型..... 70

3.3.4 undefined 和 null..... 71

3.4 复合类型..... 71

3.4.1 对象..... 72

3.4.2 数组..... 72

3.4.3 函数..... 73

3.5 运算符..... 75

3.5.1 赋值运算符.... 75

3.5.2 算术运算符.... 76

3.5.3 位运算符..... 78

3.5.4 加强的赋值运算符..... 79

3.5.5 比较运算符.... 80

3.5.6 逻辑运算符.... 81

3.5.7 三目运算符.... 82

3.5.8 逗号运算符.... 84

3.5.9 void 运算符.... 84

3.5.10 typeof 运算符..... 85

3.6 语句.....	85
3.6.1 语句块.....	85
3.6.2 空语句.....	86
3.6.3 异常抛出语句.....	86
3.6.4 异常捕捉语句.....	87
3.6.5 with 语句.....	88
3.7 流程控制.....	89
3.7.1 分支.....	89
3.7.2 while 循环.....	91
3.7.3 do while 循环.....	92
3.7.4 for 循环.....	93
3.7.5 for in 循环.....	94
3.7.6 break 和 continue.....	95
3.8 函数.....	97
3.8.1 函数定义.....	98
3.8.2 局部变量和局部函数.....	99
3.8.3 匿名函数.....	100
3.8.4 函数的成员属性和静态属性.....	102
3.8.5 递归函数.....	104
3.9 函数的参数传递.....	106
3.9.1 基本类型参数和复合类型参数.....	106
3.9.2 空参数.....	107
3.9.3 参数类型.....	108
3.10 对象.....	109
3.10.1 面向对象的概念.....	109
3.10.2 JavaScript 中的对象.....	110
3.10.3 继承和 prototype.....	110

3.11 创建对象.....	114
3.11.1 使用 new 关键字创建对象.....	115
3.11.2 使用 Object 直接创建对象.....	115
3.11.3 使用 JSON 语法创建对象.....	118
3.12 小结.....	119
第 4 章 XML 详解.....	120
4.1 XML 概述.....	120
4.1.1 标记语言.....	120
4.1.2 XML 的定义和发展史...	121
4.2 XML 的文档规则....	122
4.2.1 XML 文档分类.....	122
4.2.2 XML 文档的整体结构...	124
4.2.3 XML 声明....	125
4.2.4 XML 元素....	127
4.2.5 字符数据.....	129
4.2.6 注释.....	131
4.2.7 处理指令.....	132
4.3 XML 数据岛.....	133
4.3.1 数据岛概述.....	133
4.3.2 在 HTML 文档中载入 XML 文档....	134
4.3.3 将 XML 数据绑定到表格输出.....	134
4.4 DTD 与 Schema.....	137
4.4.1 内部 DTD.....	137
4.4.2 外部 DTD.....	138
4.4.3 公用 DTD.....	139
4.5 DTD 的语法.....	139
4.5.1 定义元素.....	140

4.5.2	定义子元素.....	142
4.5.3	定义元素属性.....	146
4.5.4	定义实体引用.....	149
4.6	Schema 的语法.....	151
4.6.1	Schema 的基本语法.....	152
4.6.2	定义简单元素.....	152
4.6.3	定义元素的父子结构关系.....	153
4.6.4	引用元素.....	154
4.6.5	Schema 中的内置类型....	156
4.6.6	自定义简单数据类型.....	156
4.6.7	自定义复杂数据类型.....	160
4.6.8	定义空元素.....	163
4.6.9	定义混合内容元素.....	164
4.6.10	定义属性.....	165
4.6.11	元素组与属性组.....	166
4.7	使用 JavaScript 解析 DOM.....	168
4.7.1	DOM 简介....	168
4.7.2	解析 XML 文档.....	169
4.7.3	创建 XML 文档.....	172
4.8	小结.....	173

## 第 5 章 使用 CSS 级联样式单... 174

5.1	样式单概述.....	174
5.2	CSS 样式单的基本使用方式.....	175
5.2.1	引入外部样式文件.....	176
5.2.2	使用内部样式定义.....	177
5.2.3	使用内联样式.....	179
5.3	使用 CSS 属性.....	180

5.3.1	文字相关属性.....	181
5.3.2	整体段落相关属性.....	182
5.3.3	背景相关属性.....	184
5.3.4	表格相关属性.....	186
5.3.5	大小相关属性.....	189
5.3.6	定位相关属性.....	189
5.3.7	边框相关属性.....	191
5.3.8	轮廓相关属性.....	194
5.3.9	常用属性.....	194
5.4	选择器定义.....	196
5.4.1	属性选择器.....	196
5.4.2	ID 选择器.....	198
5.4.3	class 选择.....	199
5.4.4	包含选择器和子元素选择器.....	199
5.4.5	超链接相关选择器.....	200
5.5	在脚本中修改显示样式.....	201
5.5.1	随机改变页面的背景色.....	201
5.5.2	导航菜单效果.....	202
5.5.3	卷帘效果.....	203
5.5.4	动态增加立体效果.....	205
5.6	小结.....	206
第 6 章	DOM 模型详解.....	207
6.1	DOM 模型简介.....	207
6.1.1	DOM 模型的概念.....	207
6.1.2	DOM 模型的作用.....	208
6.2	XML 和 DOM 模型.....	208
6.2.1	XML 和 DOM.....	209

6.2.2	使用 DOM 解析 XML 文档.....	209
6.2.3	使用 SAX 解析 XML 文档.....	212
6.2.4	使用 DOM 解析器创建 XML.....	215
6.3	HTML 文档和 DOM 模型.....	217
6.3.1	HTML 文档简介.....	217
6.3.2	HTML 文档的 DOM 结构.....	218
6.4	DOM 中的 HTML 元素.....	219
6.4.1	HTML 元素之间的继承图.....	219
6.4.2	HTML 元素之间的常见包含关系....	220
6.5	使用 JavaScript 控制 DOM.....	221
6.5.1	创建节点.....	221
6.5.2	修改节点.....	222
6.6	添加节点.....	224
6.6.1	为下拉列表增加选项.....	225
6.6.2	动态生成表格.....	226
6.7	删除节点.....	227
6.7.1	删除下拉列表的选项.....	229
6.7.2	删除表格的行或单元格.....	230
6.8	两个常用范例.....	232
6.8.1	可编辑表格.....	232
6.8.2	导航菜单.....	234
6.9	小结.....	239
第 7 章	事件处理机制.....	240
7.1	基本事件模型.....	240
7.1.1	绑定 HTML 元素属性....	240
7.1.2	绑定对象属性.....	243
7.1.3	使用 addEventListener 方法绑定.....	246



7.2 深入了解事件模型.....	247
7.2.1 事件处理函数的范围.....	247
7.2.2 使用返回值改变默认行为.....	248
7.2.3 JavaScript 中的 MVC.....	249
7.2.4 在代码中触发事件.....	253
7.3 Netscape 4 的事件模型.....	255
7.3.1 事件对象.....	255
7.3.2 事件捕捉.....	257
7.4 Internet Explorer 的事件模型.....	259
7.4.1 在 Internet Explorer 中绑定事件处理器.....	259
7.4.2 使用 script for 绑定.....	259
7.4.3 使用 attachEvent()方法执行绑定.....	260
7.4.4 事件对象.....	260
7.4.5 事件传递.....	264
7.4.6 重定向.....	266
7.5 DOM 级别 2 的事件模型.....	268
7.5.1 绑定事件处理函数.....	269
7.5.2 事件传播和转发.....	270
7.5.3 取消事件的默认行为.....	271
7.5.4 控制事件传播.....	272
7.5.5 转发事件.....	273
7.5.6 DOM 2 中的事件.....	274
7.6 小结.....	275
第 8 章 使用 DHTML 动态操作 HTML 文档.....	276
8.1 DHTML 和 DOM 两种模型.....	276
8.2 使用 DHTML.....	277
8.2.1 DHTML 的包含体系.....	277

8.2.2 使用 Window 对象.....	278
8.3 文档对象.....	284
8.4 表单和表单元素.....	286
8.4.1 表单和表单元素对象.....	286
8.4.2 Form 对象的方法和属性.....	287
8.4.3 Text 和 Textarea.....	288
8.4.4 Radio 和 ChechBox.....	289
8.4.5 Select 和 Option.....	290
8.5 小结.....	294
第 9 章 XMLHttpRequest 对象详解.....	295
9.1 XMLHttpRequest 对象概述.....	295
9.2 XMLHttpRequest 的方法和属性.....	296
9.2.1 XMLHttpRequest 的方法.....	296
9.2.2 XMLHttpRequest 的属性.....	299
9.3 发送请求.....	304
9.3.1 发送简单请求.....	304
9.3.2 发送 GET 请求.....	306
9.3.3 发送 POST 请求.....	310
9.3.4 发送请求时的编码问题.....	311
9.3.5 发送 XML 请求.....	316
9.4 处理服务器响应.....	318
9.4.1 处理的时机.....	318
9.4.2 使用文本响应.....	319
9.4.3 使用 XML 响应.....	319
9.4.4 使用 DOM 模型生成页面.....	321
9.5 XMLHttpRequest 对象的运行周期.....	322
9.6 Ajax 必须解决的问题.....	322

9.6.1 跨浏览器问题.....	323
9.6.2 安全问题.....	324
9.6.3 性能问题.....	326
9.7 小结.....	330
第 3 部分 Ajax 常用框架.....	331
第 10 章 Prototype 框架详解.....	332
10.1 Prototype 的下载和安装.....	332
10.1.1 Prototype 概述.....	332
10.1.2 下载 Prototype.....	333
10.1.3 安装 Prototype.....	333
10.2 使用 Prototype 的基本函数.....	334
10.2.1 使用\$()函数.....	334
10.2.2 使用\$A()函数.....	337
10.2.3 使用\$F()函数.....	339
10.2.4 使用\$H()函数.....	340
10.2.5 使用\$R()函数.....	341
10.2.6 使用 Try.these()函数....	342
10.3 Prototype 的自定义对象和类...	343
10.3.1 使用 Element 对象.....	343
10.3.2 使用 Enumerable 类.....	346
10.3.3 使用 Field 对象操作表单域.....	350
10.3.4 使用 Form 对象操作表单.....	351
10.3.5 使用 Form.Element 对象.....	353
10.3.6 使用 Hash 对象.....	354
10.3.7 使用 Class 对象.....	355
10.3.8 用于操作 HTML 元素的类.....	356
10.3.9 常用的监听器.....	357

10.4	Prototype 的常用扩展.....	359
10.4.1	扩展 Array.....	359
10.4.2	扩展 Document.....	361
10.4.3	扩展 String.....	361
10.4.4	扩展 Event.....	363
10.5	Prototype 为 Ajax 增加的类.....	364
10.5.1	使用 Ajax.Request 类.....	364
10.5.2	使用 Ajax.Responders 对象.....	368
10.5.3	使用 Ajax 对象.....	369
10.5.4	使用 Ajax.Updater 类.....	369
10.5.5	使用 Ajax.PeriodicalUpdater 类.....	374
10.6	小结.....	376
第 11 章	基于 Prototype 的应用：实现自动完成功能.....	377
11.1	应用的基本分析和设计.....	377
11.1.1	数据要求...	377
11.1.2	数据表结构.....	378
11.2	Domain Object 和持久层.....	378
11.2.1	Domain Object.....	378
11.2.2	实现 DAO 组件.....	381
11.3	实现 Service 组件...	385
11.4	使用 Servlet 提供服务器响应.....	388
11.4.1	实现根据前缀查询品牌.....	389
11.4.2	实现根据品牌查询型号.....	390
11.4.3	实现根据型号查询具体描述.....	391
11.5	客户端 HTML 页面实现.....	392
11.6	增加 HTML 页面的事件响应能力.....	394
11.6.1	实现品牌输入框的事件处理函数.....	395

- 11.6.2 实现键盘事件处理函数..... 397
- 11.6.3 根据品牌提示型号..... 399
- 11.6.4 根据型号显示描述..... 401
- 11.6.5 注册 Ajax 事件监听器..... 403

## 11.7 小结..... 404

## 第 12 章 使用 Dojo.... 405

### 12.1 初识 Dojo..... 405

- 12.1.1 Dojo 概述..... 405
- 12.1.2 下载和安装 Dojo..... 406

### 12.2 了解 Dojo 的体系..... 408

- 12.2.1 通用库..... 408
- 12.2.2 Ajax 相关库..... 409
- 12.2.3 页面控件相关库..... 409
- 12.2.4 其他相关库..... 409

### 12.3 使用 Dojo 的通用库..... 410

- 12.3.1 使用 djConfig 对象..... 410
- 12.3.2 do 对象的函数..... 412
- 12.3.3 字符串相关函数..... 414
- 12.3.4 扩展函数... 416
- 12.3.5 DOM 相关函数..... 421

### 12.4 Dojo 的 Ajax 支持..... 425

- 12.4.1 使用 Dojo 开发 Ajax 应用..... 425
- 12.4.2 发送请求的几种形式..... 427
- 12.4.3 使用 queueBind 函数发送请求队列..... 431
- 12.4.4 使用 dojo.io.setIFrameSrc 函数代替 XMLHttpRequest..... 431
- 12.4.5 使用 dojo.io.argsFromMap 函数.... 432

### 12.5 Dojo 的事件机制... 433

12.5.1	简单的 connect 绑定·····	433
12.5.2	使用对象的方法作为事件处理器·····	434
12.5.3	为非 DOM 元素绑定事件处理器·····	435
12.6	基于 connect 的 AOP 实现·····	436
12.6.1	简单的 before 绑定·····	437
12.6.2	简单的 around 绑定·····	437
12.6.3	访问目标方法参数的绑定·····	439
12.6.4	与 around 绑定在一起使用的监听器·····	440
12.7	connect 绑定其他函数·····	441
12.7.1	使用 connectBefore·····	441
12.7.2	使用 connectAround·····	441
12.7.3	使用 connectOnce·····	442
12.7.4	使用 kwConnect·····	442
12.7.5	使用 disconnect·····	443
12.7.6	使用 kwDisconnect·····	443
12.8	基于 Topic 事件机制·····	443
12.9	Dojo 的拖拉功能·····	444
12.9.1	自由拖动·····	445
12.9.2	有相对“坐标”的移动·····	446
12.9.3	带手柄的移动·····	448
12.10	Dojo 的常用 widget·····	449
12.10.1	按钮·····	449
12.10.2	对话框·····	451
12.10.3	日历·····	454
12.10.4	树·····	457
12.10.5	Tab 页·····	459
12.11	Dojo 与 Prototype 的对比·····	461

12.12 小结..... 461

第 13 章 基于 Dojo 的动态树..... 462

13.1 自关联 Domain Object 的设计..... 462

13.2 基于 Hibernate 的 DAO 实现... 464

13.2.1 DAO 组件的接口..... 464

13.2.2 DAO 组件的实现类.... 465

13.3 基于 Spring 的业务逻辑组件... 467

13.3.1 业务逻辑组件的接口..... 467

13.3.2 业务逻辑组件的实现类..... 468

13.3.3 配置 Spring 容器..... 471

13.4 基于 Spring 的集成测试..... 473

13.5 Dojo 树的相关 Widget..... 475

13.5.1 使用 TreeContextMenu..... 475

13.5.2 使用 TreeRPCController..... 478

13.6 Dojo 树的 HTML 页面实现..... 479

13.7 实现服务器响应 Servlet..... 484

13.7.1 实现 getAllRoot..... 484

13.7.2 实现 updateNode..... 486

13.7.3 实现 createChild..... 487

13.7.4 实现 removeNode..... 489

13.7.5 实现 getChildren..... 490

13.8 与其他树的对比.... 491

13.9 小结..... 493

第 14 章 使用 DWR... 494

14.1 DWR 的下载和安装..... 494

14.1.1 DWR 概述..... 494

14.1.2 下载和安装 DWR..... 495

14.2	使用 DWR.....	500
14.2.1	编写处理类.....	500
14.2.2	配置 DWR.....	502
14.3	使用 DWR 的转换器.....	504
14.3.1	基本转换器.....	504
14.3.2	对象转换器.....	504
14.3.3	数组转换器.....	507
14.3.4	集合类型转换器.....	508
14.4	方法声明.....	509
14.5	使用 DWR 的创建器.....	510
14.5.1	创建器的配置.....	510
14.5.2	使用 new 创建器.....	512
14.5.3	使用 none 创建器.....	512
14.5.4	使用 script 创建器.....	513
14.6	调用服务器端的方法.....	514
14.6.1	调用服务器端方法的通用配置.....	514
14.6.2	使用简单回调.....	516
14.6.3	使用 JSON 格式的回调.....	524
14.6.4	将客户端参数传递到回调函数.....	526
14.7	使用 engine.js.....	526
14.7.1	设置调用顺序.....	527
14.7.2	设置全局超时时长.....	527
14.7.3	设置全局 Hook 函数...	527
14.7.4	设置全局处理函数.....	528
14.7.5	设置常用的全局选项.....	528
14.7.6	批处理.....	528
14.8	使用 util.js.....	529
14.8.1	使用\$().....	530
14.8.2	处理列表...	530
14.8.3	处理表格...	535



14.8.4	访问 HTML 元素值····	540
14.8.5	工具函数····	542
14.9	整合第三方 J2EE 框架········	545
14.9.1	访问 Servlet API········	545
14.9.2	整合 Spring········	546
14.9.3	整合 Struts········	549
14.10	异常处理········	551
14.11	反向 Ajax········	553
14.11.1	配置使用反向 Ajax····	553
14.11.2	在 Java 方法中操作 Web 页······	554
14.11.3	在客户端调用反向 Ajax 方法····	556
14.12	小结····	557
第 15 章	基于 DWR 的 Ajax 应用：即时消息系统····	558
15.1	实现 Hibernate 持久	

## 序言

Ajax 技术是 2006 年最热门的技术，无论说它是对传统 Web 技术的改进还是革命，其实质都是：它具有很强的生命力，确实代表了未来 Web 应用开发的趋势。也许 Ajax 并不是唯一的选择，但 Ajax 的功能是大家有目共睹的。

不过，Ajax 绝不是一门独立的技术，笔者在教学过程中曾遇到一个狂热的学员，他说：我就要学习 Ajax 技术，学好 Ajax 技术就不用担心找不到工作了。笔者不禁哑然：这是一个误区，单纯的 Ajax 技术没有任何用处，绝对没有。说穿了，Ajax 仅仅致力于改善用户体验，也就是说，它是表现层的技术，离开了传统 Web 应用的支持，Ajax 技术便会成为“无本之木，无源之水”。

在动手编写本书之前，笔者仔细研究了市面上介绍 Ajax 的图书，有的图书甚至宣称：本书所介绍的内容是平台无关的——这是一个歧义句：Ajax 技术本身是平台无关的，但任何一本书介绍的内容都不可能是平台无关的。因为 Ajax 技术是表现层的，而底层的技术可以采用任何 Web 编程技术：ASP，PHP，.NET，Java 或 Ruby On Rails，因此 Ajax 技术是平台无关的。但一本书的内容不可能是平台无关的，除非每段代码都提供所有语言的实现版本，这不可能。那些宣称内容平台无关的书，一会儿使用 PHP 作为底层实现，一会使用 Java 作为实现，这变成了对读者的一种折磨。

本书并不是一本单纯的 Ajax 入门书籍，也不打算为介绍 Ajax 而介绍 Ajax。除了详细介绍 Ajax 技术的各方面知识之外，本书还介绍了如何让 Ajax 技术与 J2EE 技术完美融合。换句话说，本书介绍的内容更侧重于让 Ajax 技术融入实际应用开发，而不是满足于表面的 Ajax 技巧。本书所示范的应用，底层都按照 J2EE 规范进行实现；而在用户界面上，Ajax 则大展身手，极大地改善了用户体验。

本书的写作风格与笔者在新东方的授课风格一致：平实、浅显，放弃新名词，把所有深奥的内容变得浅显易懂。笔者不喜欢罗列一堆的新名词，抄大段大段的理论来炫耀自己。作为一个有多年编程经验的程序员，笔者相信“代码就是硬道理”，因此本书中涵盖了 Ajax 的绝大部分实际应用场景。希望读者也不要仅仅“看书”，而一定要参照本书进行动手操作，将每个应用都实际做一遍，相信会有很大的提高。

本书优点

### 1. 专业性强

本书介绍的不是单纯的 Ajax 技术，而是详细介绍了如何开发 Ajax 应用，如何将 Ajax 技术和 J2EE 应用整合在一起。本书致力于让 Ajax 技术真正融入实际应用的开发，而不是停留在 Ajax 层，为了 Ajax 而 Ajax。通过认真阅读本书，读者不仅可以掌握如何开发 Ajax 应用，而且可以掌握如何让 Ajax 技术和 J2EE 应用实现完美融合。

## 2. 知识丰富

本书除介绍 Ajax 的各种相关知识（如 JavaScript，DOM 和 XML）之外，还详细介绍了 5 个目前流行的 Ajax 框架：Prototype.js，Dojo，DWR，JSON-RPC-Java 和 AjaxTags，并通过实际示例讲述了如何使用这 5 个框架。

## 3. 经验丰富，针对性强

笔者既担任过软件开发的技术经理，也担任过软件公司的培训导师，还担任过职业培训的专职讲师，熟悉软件开发的难点，同时也了解软件学习过程中的苦楚。因此，本书尽量针对学习过程中容易遇到的难点进行重点讲解，对开发过程中的“陷阱”给出提示。

## 4. 示例丰富，实用性强

本书的各个示例针对不同的知识点设计，尽量不与其他知识点掺杂在一起，以免分散读者的注意力。各知识点的示例重点突出，示范性非常强。另外，本书所有的示例都侧重于介绍 Ajax 在实际项目中的使用，例如第 13 章的全功能 Ajax 树，支持在客户端删除、创建和拖动节点，而且这些操作都将通过 Ajax 请求直接持久化到数据库，非常有实用价值。

### 本书内容

Ajax 的脚本语言 JavaScript，包括 JavaScript 的各种基本知识，还有 JavaScript 的面向对象知识。

XML 语言的相关知识，包括 XML 的文档规则，DTD 和 Schema 的知识，以及如何在 JavaScript 中创建、解析 XML 文档。

CSS 样式单的相关知识，包括 CSS 样式单的选择器，样式单的各种属性，以及如何在脚本中动态修改样式单，从而动态修改 HTML 页面的表现。

DOM 模型的相关知识，包括 HTML 页面和 DOM 模型的关系，以及如何操作 DOM 模型，从而让 HTML 页面内容动态更新。

JavaScript 的事件机制，包括 Internet Explorer 中的事件机制、Netscape 4 中的事件机制和 DOM 2 中的事件机制。

DHTML 模型的相关知识。

XMLHttpRequest 对象的相关知识，包括如何在不同浏览器中创建 XMLHttpRequest 对象，XMLHttpRequest 对象的各种属性、方法，以及如何通过 XMLHttpRequest 对象发送异步请求并获得和处理服务器响应。

Prototype.js 的相关知识，包括 Prototype.js 的扩展和内置函数等。当然，也重点介绍了 Prototype.js 的 Ajax 支持。

Dojo 的相关知识，包括 Dojo 的扩展、内置函数、Dojo 的 Ajax 支持、Dojo 事件机制、Dojo 的拖动功能以及 Dojo 的常用页面控件。

DWR 的各种相关知识，包括 DWR 的创建器、转换器，DWR 框架所生成的 JavaScript 函数库，DWR 和 J2EE 框架的整合，以及 DWR 2.0 的新知识：反向 Ajax。

JSON-RPC-Java 框架的各种相关知识，包括 JSONRPCServlet 和 JSONRPCBridge 等核心类的作用和使用方法，以及如何编写服务器处理类，如何通过 JSONRpcClient 调用服务器的 Java 方法。

AjaxTags 的各种常用标签，以及如何编写 AjaxTags 的各种服务器处理类。

本书介绍的 Ajax 知识虽然也适用于其他 Web 开发平台，但还是希望读者在阅读本书之前具有一定的 J2EE 编程基础；有 J2EE 编程基础的读者可以更好地将 Ajax 技术应用到实际的 J2EE 开发中，切记：仅有 Ajax 技术是做不了任何事情的。

本书由李刚主编，参加本书编写工作的人员还有杨光景、杨毅、李海涛、汪洋、谷文港、陈亮、李守军、江旭初、王坤、赵元、易阳华、孙江苏、姜海森、张明霞、王江、王斌、郭剑云、张大发、刘挺、尹海涛、戴隆忠、李善坡、张磊、唐友生、于兆海、刘洪燕、王悠。

## 第 2 章 Ajax 初体验

- ◆ Ajax 技术带来的改变
- ◆ 体验 Ajax 聊天室的便捷
- ◆ 开发 Ajax 聊天室
- ◆ 异步解析服务器响应
- ◆ Ajax 技术的优势
- ◆ 开发 JSP 聊天室
- ◆ 异步发送请求
- ◆ Ajax 和传统 Web 应用的对比

Ajax 虽然是个很新的名词，但并不是一门全新的技术。正如前文所介绍的，Ajax 所使用的 JavaScript，CSS 和 DOM 对象等早已存在。Ajax 通过这些传统的对象改善用户的交互体验，让用户能异步发送请求：在浏览页面的同时，向服务器发送请求。

JavaScript，CSS 和 DOM 都是相当成熟的技术，它们以前被称为 DHTML，即动态 HTML。DHTML 可以为页面创建交互性很强的页面，但它的致命弱点在于：无法与服务器通信，无法异步发送请求。因此，虽然 DHTML 提供了非常漂亮的用户界面，但频繁的页面刷新限制了它的使用。Ajax 加入了 XMLHttpRequest 对象，这个对象提供了与服务器交互的能力，可以异步发送请求，提供与服务器异步通信的能力，无须独占用户在页面上执行的操作。因而，Ajax 给用户一种全新的体验。

### 2.1 Ajax 带来的优势

Ajax 技术的出现，改变了传统 Web 应用的模式，它既是对传统 Web 应用的完善，也是对传统 Web 应用的革命。Ajax 技术采用异步发送请求的方式代替采用表单提交来更新 Web 页面的方式，从而揭开了无刷新动态更新页面时代的序幕。Ajax 可以成为 Web 应用开发史上的里程碑，因而很多地方将 Ajax 技术称为 Web 2.0 技术（当然，Web 2.0 还包含了其他一些技术）。假设一个关于级联菜单的处理的应用场景，所谓级联菜单就是如图 2.1 所示的菜单。



图 2.1 级联菜单示范

级联菜单右边菜单的菜单项根据左边菜单的改变而改变：随着左边菜单的改变，右边的菜单需要级联改变，如图 2.2 所示，当左边选中“英国”时，右边的下拉菜单也随之改变。



图 2.2 级联菜单的改变

在没有 Ajax 技术之前，因为客户端只能通过提交表单或者采用地址栏传递参数的形式发送参数，不管哪种形式，都将导致页面被重新加载，这不是我们希望看到的结果，因而只能一次性将级联菜单的所有数据全部读取出来并写入数组，然后根据用户的操作，通过 JavaScript 控制显示对应子菜单。

上面的方法解决了操作响应速度（无须操作时向服务器发送数据）、无须重载页面等问题，但引发了更严重的问题，考虑如下情形：左边的菜单中有 200 个国家，每个国家又对应 200 个城市，采用上面方法将一次从数据库加载 40 200 项数据，并将这些数据保存在数组中——如果更大呢？结果将更加糟糕。也许用户仅需要查找美国对应的城市，即其他国家的城市都无须加载，这样，一次加载方式将读取大量冗余数据，既增加了服务器负载，也浪费了网络带宽，更浪费了客户机的内存（浏览器 JavaScript 必须定义大数组存放数据）。如果遇到菜单有很多级、每一级菜单有上百个子菜单，这种资源的浪费将以几何级数增长。

如果换成 Ajax 方案，将可以完全避免这些问题：页面无须一次加载所有的子菜单，页面加载时只加载最左边的菜单。当用户单击了左边国家的下拉菜单后，异步向服务器发送请求，从服务器获取该菜单的子菜单；当用户单击第二级菜单时，再次异步向服务器发送请求，从服务器获取该菜单的全部子菜单，依此类推。通过这种方法，可避免一次加载全部菜单项，从而提供更好的性能。

Ajax 应用特别适用于交互较多、频繁读数据、数据分类良好的 Web 应用。大致上，使用 Ajax 技术有如下优势：

减轻客户端的内存消耗。Ajax 的根本理念是“按需取数据”，所以最大可能地减少了冗余请求，避免客户端内存加载大量冗余数据。

无刷新更新页面。通过异步发送请求，避免频繁刷新页面，从而减少用户的等待时间，提供给用户一种连续的体验。

Ajax 技术可以将传统的服务器工作转嫁到客户端（例如购物车的状态），从而减轻服务器和带宽的负担，节约空间和带宽租用成本。

Ajax 基于标准化技术，几乎所有浏览器都支持这种技术，无须下载插件或虚拟机程序。

“按需取数据”的模式降低了数据的实际读取量，在传统的 Web 应用里，服务器的每次响应都是一个完整的页面；而在基于 Ajax 技术的 Web 应用里，服务器的响应只是必须更新的数据。

如果服务器响应数据过大，那么传统 Web 应用将出现重新加载的白屏；由于 Ajax 采用异步的方式发送请求，页面的更新由 JavaScript 操作 DOM 完成，因此，在读取数据的过程中，浏览器中也不会出现白屏，而是原来的页面状态（甚至可以使用 LOADING 提示框让用户了解读取状态）。数据接收完成后，页面才开始更新部分内容，这种更新是瞬间完成的，用户根本无法感受到更新。



企业通过使用 Ajax 可以增强网站的功能，改善用户体验。用户可以通过滚动屏幕浏览大量的信息，可以更方便地将物品拖入在线购物车或者在线配置产品，而这些都无须刷新页面。事实上，相当多的企业都在考虑使用 Ajax 技术来改善用户的体验。

关于 Ajax 技术对传统 Web 应用的改善，下面将介绍一个最简单的应用来示范 Ajax 的用法和优势。下面的聊天室应用将分别采用传统 Web 方式和 Ajax 技术实现，读者既可以感受到 Ajax 应用开发的大概过程，也可以作为浏览者感受到 Ajax 的改善。

## 2.2 传统的 JSP 聊天室

JSP 聊天室需要实现的功能有两个：第一个功能是对用户的管理，包括用户登录、用户注册等。第二个功能是用户的聊天，聊天功能需要保证最近聊天信息不丢失。用户信息、聊天信息都可以通过数据保存，为了简单起见，用户信息用 Properties 文件保存，不对聊天信息进行持久化。

本 JSP 聊天室一样遵守 MVC 的开发模式。客户端向控制器发送请求，控制器负责拦截用户请求，调用 Model 处理用户请求，控制器根据 Model 的处理结果，决定向用户呈现怎样的界面。JSP 聊天室的业务逻辑非常简单，包含 3 个简单用例。

用户注册：向用户、密码的持久化信息中增加一条记录。

用户登录：判断用户输入的用户名、密码是否正确，正确则允许登录聊天，否则拒绝聊天。

用户聊天：发送消息让所有的用户看到。

聊天室的组件结构图如图 2.3 所示。



图 2.3 JSP 聊天室系统的组件结构图

### 2.2.1 实现业务逻辑组件

系统没有采用数据库存放用户信息，而是使用 **Properties** 文件存放用户名、密码。所有的用户登录验证、新用户注册都需要通过 **Properties** 文件校验。业务逻辑组件提供了如下方法用于加载属性文件：

```
private Properties loadUser() throws IOException
{
    //判断 userList 是否为空，如果为空，则重新加载属性文件
    if (userList == null)
    {
        //首先检查保存用户名、密码的属性文件是否存在
        File f = new File("userFile.properties");
        //如果文件不存在，则创建文件
        if (!f.exists())
            f.createNewFile();
        //创建新的 Properties 实例，该实例保存了用户名-密码对
        userList = new Properties();
        //从属性文件中加载所有的用户名、密码
        userList.load(new FileInputStream(f));
    }
    //返回保存用户名-密码对的 Properties 实例
    return userList;
}
```

这个方法是个基础方法，用于加载所有用户名、密码，**userList** 是当前系统的所有用户名、密码列表，以业务逻辑组件的实例属性的形式存在。



当然，如果系统的注册用户非常多，则属性文件会非常大，`userList` 也会非常大，可能导致系统的性能下降，这时采用数据来保存用户名和密码更合适。本范例仅为了演示传统 JSP 聊天室和 Ajax 聊天室的对比，因此此处没有使用数据库。

除此之外，还有对应的方法用于将 `userList` 保存到 `Properties` 文件，每次用户注册成功后都应该将新注册的用户保存到 `Properties` 文件。保存 `userList` 的方法如下：

```
private boolean saveUserList() throws IOException
{
    //如果用户列表为空，则无法保存用户
    if (userList == null)
    {
        return false;
    }
    //调用 Properties 类的 store 方法将用户列表保存到文件输出流
    userList.store(new FileOutputStream("userFile.properties"), "userList");
    return true;
}
```

上面两个方法都是系统用于持久化的方法，只是此处的持久化无须访问数据库，而是将持久化信息以文件的形式存放。业务逻辑对象必须向控制器提供的方法有如下 4 个。

`public boolean validLogin(String user, String pass)`: 用于判断用户名、密码是否可以成功登录。

`public boolean addUser(String name, String pass)`: 用户注册时，向 `Properties` 文件中增加记录。

`public String getMsg()`: 用于获取聊天信息。

`public void addMsg(String user, String msg)`: 用户增加聊天信息，聊天信息是瞬态信息，系统没有对聊天信息进行持久化，但每个用户的发言应该被增加到聊天信息里。

业务逻辑组件 `ChatService` 的代码如下：

```
//JSP 聊天室所使用的业务逻辑组件

public class ChatService
{
    //将该组件写成单态模式，将 ChatService 实例保存成静态属性
    private static ChatService cs;

    //本系统的用户列表
    private Properties userList;
```

//系统保存的瞬态对象：聊天信息。每个用户的发言对应 List 的一个元素

```
private LinkedList<String> chatMsg;
```

```
private ChatService()
```

```
{
```

```
}
```

//单态模式的实例访问点，通过该方法返回 ChatService 类的一个实例

```
public static ChatService instance()
```

```
{
```

```
    //保证当 ChatService 对象不为空时，才重新创建实例
```

```
    if (cs == null)
```

```
    {
```

```
        cs = new ChatService();
```

```
    }
```

```
    //返回 ChatService 实例
```

```
    return cs;
```

```
}
```

//验证用户名、密码是否可以成功登录

```
public boolean validLogin(String user, String pass)
```

```
    throws IOException
```

```
{
```

```
    //如果以 user 为 key 获取 value 为空，则表明系统用户不存在，验证失败
```

```
    if (loadUser().getProperty(user) == null)
```

```
    {
```

```
        return false;
```

```
    }
```

```
    //如果以 user 为 key 获取 value，与当前的密码匹配，则验证成功
```

```
    if (loadUser().getProperty(user).equals(pass))
```

```
{
    return true;
}

return false;
}

//新增用户

public boolean addUser(String name, String pass)
    throws Exception
{
    //如果用户 List 为空
    if (userList == null)
    {
        //从 Properties 文件中加载用户 List
        userList = loadUser();
    }

    //添加用户之前，首先判断系统中是否已经包含同名的用户
    if (userList.containsKey(name))
    {
        //如果系统中已包含同名用户
        throw new Exception("用户名已经存在，请重新选择用户名");
    }

    //将用户名、密码放入 Properties 文件保存
    userList.setProperty(name , pass);

    saveUserList();

    return true;
}

//获取所有的聊天信息
```

```
public String getMsg()
{
    //如果保存聊天信息的 List 为空
    if (chatMsg == null)
    {
        //创建新的聊天信息 List
        chatMsg = new LinkedList<String>();
        //返回空，即返回空的聊天信息
        return "";
    }
    String result = "";
    //当聊天信息 List 不为空时，则遍历集合并将集合的元素添加到聊天信息中
    for (String tmp : chatMsg)
    {
        result += tmp + "\n";
    }
    return result;
}
//向瞬态的聊天信息中加入用户的聊天信息
public void addMsg(String user, String msg)
{
    //如果聊天信息为空
    if (chatMsg == null)
    {
        //创建一个新的聊天信息集合
        chatMsg = new LinkedList<String>();
    }
}
```

```
        if (chatMsg.size() > 40)
        {
            chatMsg.removeFirst();
        }

        chatMsg.add(user + "说: " + msg);
    }

    //////////////////////////////////////
    //      下面是系统的工具方法
    //////////////////////////////////////

    //从 Properties 文件中加载用户列表

    private Properties loadUser() throws IOException
    {
        //判断 userList 是否为空，如果为空，则重新加载属性文件

        if (userList == null)
        {
            //首先检查保存用户名、密码的属性文件是否存在

            File f = new File("userFile.properties");

            //如果文件不存在，则创建文件

            if (!f.exists())

                f.createNewFile();

            //创建新的 Properties 实例，该实例保存了用户名、密码对

            userList = new Properties();

            //从属性文件中加载所有的用户名、密码

            userList.load(new FileInputStream(f));
        }

        //返回保存用户名、密码对的 Properties 实例

        return userList;
    }
}
```

```
}  
  
//将 userList 保存到 Properties 文件中  
  
private boolean saveUserList()throws IOException  
{  
    //如果用户列表为空，则无法保存用户  
  
    if (userList == null)  
    {  
        return false;  
    }  
  
    //调用 Properties 类的 store 方法将用户列表保存到文件输出流  
  
    userList.store(new FileOutputStream("userFile.properties"), "userList");  
  
    return true;  
}  
}
```

### 2.2.2 实现控制器

系统的控制器使用 Servlet 充当，Servlet 负责拦截用户请求，调用 ChatService 对象处理用户请求，根据处理结果将请求转发到合适的页面显示。本系统包含 3 个用例：用户注册、用户登录和用户聊天。3 个用例分别对应 3 次请求，系统为每个请求配置一个控制器。控制器的运行结构大致相似，下面以注册所用的控制器为例进行讲解。

```
//控制器 Servlet 继承 HttpServlet  
public class RegServlet extends HttpServlet  
{  
    //Servlet 使用 service 方法响应用户请求  
    public void service(HttpServletRequest request, HttpServletResponse response)  
        throws IOException, ServletException  
    {  
        //获取请求参数：用户名和密码  
        String name = request.getParameter("name");  
        String pass = request.getParameter("pass");  
        //执行服务器端数据校验，用户名、密码不能为空  
        if (name == null || pass == null)
```

```
{
    request.setAttribute("tip", "用户名和密码都不能为空");
    forward("/reg.jsp", request, response);
}
//开始调用业务逻辑对象的方法
try
{
    //调用业务逻辑对象的方法
    if (ChatService.instance().addUser(name, pass))
    {
        //如果注册成功, 则将提示信息存入系统的 tip 属性
        request.setAttribute("tip", "注册成功, 请登录系统");
        //将请求转发到 reg.jsp 页面显示
        forward("/reg.jsp", request, response);
    }
    else
    {
        //如果不能正常注册, 则将提示信息放入 request 属性
        request.setAttribute("tip", "无法正常注册, 请重试");
        forward("/reg.jsp", request, response);
    }
}
//如果捕捉到异常, 例如注册的用户名已经存在
catch (Exception e)
{
    //将异常信息设置成 request 属性
    request.setAttribute("tip", e.getMessage());
    forward("/reg.jsp", request, response);
}
}
//工具方法, 用于转发用户请求
private void forward(String url, HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException
{
    ServletContext sc = getServletConfig().getServletContext();
    RequestDispatcher rd = sc.getRequestDispatcher(url);
    rd.forward(request, response);
}
}
```

其余两个控制器 ChatServlet 和 LoginServlet 与此类似, ChatServlet 调用 addMsg 和 getMsg 方法用于添加聊天信息和显示聊天信息。LoginServlet 则调用 validLogin 和 getMsg 方法用于验证登录和显示聊天信息。

两个控制器调用 `getMsg` 方法获取聊天记录后，将聊天记录设置在 `request` 的 `msg` 属性里。JSP 页面则可以通过简单的表达式语法输出聊天信息：

```
${requestScope.msg}
```

当然，为了使用该 `Servlet`，还需在 `web.xml` 文件中配置它。配置该 `Servlet` 的代码如下：

```
<!-- 配置 Servlet --->

<servlet>

    <!-- 确定 Servlet 的名字 -->

    <servlet-name>reg</servlet-name>

    <!-- 确定 Servlet 的实现类 -->

    <servlet-class>lee.RegServlet</servlet-class>

</servlet>

<servlet-mapping>

    <!-- 确定 Servlet 的名字 -->

    <servlet-name>reg</servlet-name>

    <!-- 确定 Servlet 映射的 URL -->

    <url-pattern>/reg.do</url-pattern>

</servlet-mapping>
```

经过上面的配置，`RegServlet` 对应的 URL 地址为 `reg.do`，注册表单的 `action` 属性可设置为 `reg.do`。当提交表单时，`lee.RegServlet` 将调用业务逻辑组件，然后再处理用户请求。

### 2.2.3 实现视图

JSP 页面就是聊天室的视图。视图负责收集用户请求信息，向服务器发送请求，并显示请求信息。视图还负责完成基本的客户端数据校验。

本聊天室有 3 个视图，分别对应 3 个用户界面，包括用户登录、用户注册和用户聊天。其中，用户登录和用户注册两个视图非常相似，两个视图都负责收集用户名和密码，并将该用户名和密码发送到服务器请求处理，只是请求处理的逻辑有所区别，因此两个视图的请求控制器不同。图 2.4 是输入用户名、密码不匹配后的登录界面。

上面的视图除了包含基本的登录表单之外，还包含如下 JavaScript 代码（对表单域完成客户端数据校验）：



```
<script>

//自定义用户函数，该函数用于完成基本的客户端数据校验

function check()

{

    //通过 getElementById 方法获取文档中的用户名文本框对象

    var name = document.getElementById("name");

    //通过 getElementById 方法获取文档中的密码文本框对象

    var pass = document.getElementById("pass");

    var errStr = "";

    //当用户名为空时

    if (name.value == "" || name.value == null)

    {

        //添加错误提示字符串

        errStr += "用户名不能为空\n";

    }

    //当密码为空时

    if (pass.value == "" || pass.value == null)

    {

        //添加错误提示字符串

        errStr += "密码不能为空\n";

    }

    //如果错误提示字符串为空，表明用户名、密码都已经输入

    if (errStr == "" || errStr == null)

    {

        return true;

    }

    //否则弹出错误提示
```

```

    alert(errStr);

    //拒绝提交表单

    return false;

}

//关联表单提交与数据校验函数

document.getElementById("loginForm").onsubmit = check;

</script>

```



图 2.4 用户名和密码不匹配的登录界面

聊天界面则由一个文本域和一个文本框组成，文本框负责收集用户输入的聊天信息，文本域负责显示当前所有用户的聊天信息。

#### 2.2.4 JSP 聊天室的问题

在经典的 B/S 结构应用里，都采用请求/响应的模式：客户端向服务器发送请求，而服务器则生成对客户端的响应。在这种结构模式里，服务器不会主动生成对客户端的响应，因而如果客户端不发送请求，则即使当前的聊天信息发生改变，用户依然看不到其他用户的聊天信息。

聊天所使用的 JSP 页面经常需要刷新，通过前文的介绍我们知道：用户发送请求，请求被控制器截获，控制器处理完用户请求后，将请求转发到 JSP 页面显示处理结果。关键的问题在于：每次用户发送请求后，必须等待服务器响应，如果服务器响应很慢，客户端浏览器将一直等待，什么事情也做不了。如果客户端想再次发送请求，

则完全不可能，因为服务器没有生成响应，即客户端眼前的界面是一片空白。在聊天过程中，客户端不断下载服务器的聊天页面，每次提交聊天信息，都需要下载一次。

服务器每次生成响应都是一个完整页面。在实际应用中，完整页面包含的内容相当多，少则几百行，多则几千行、上万行。也许这个页面与客户端已经加载的页面大同小异，除了少量的数字需要改变，页面的其他修饰、效果、图片等都无须更新，但客户端必须重新下载这些已经下载过的资源。大量重复下载相同资源，严重占用了客户的宝贵网络带宽，也导致客户端的响应变慢。总体而言，上面的 JSP 聊天室有如下问题：

JSP 页面无法异步发送请求，用户请求与服务器响应严格交替：用户请求→服务器响应。如果用户没有发送请求，则服务器不会生成响应；如果服务器响应没有完成，则用户无法再次发送请求。

服务器的响应总是完整 JSP 页面，大量下载重复资源。

## 2.3 Ajax 聊天室

针对 JSP 聊天室存在的问题，Ajax 聊天室做出了相应的改进。正如前面提到的：Ajax 并不是取代 B/S 结构的应用，而是更好地完善了传统的 Web 应用。

针对 JSP 存在的两个问题，Ajax 都有非常好的解决方案：Ajax 使用 XMLHttpRequest 异步发送请求，Ajax 的服务器响应的仅是必需的数据，而不再是整个页面，必需的数据通过 JavaScript 在视图中显示。使用 Ajax 可提高页面的复用：浏览器从服务器下载一个页面后，不是一旦提交就丢弃该页面，立即进入下个页面——这种代价相当大，用户需要频繁下载完整页面；使用 Ajax，则可以长时间地使用同一个页面，客户端可以很好地复用已下载的页面。

### 2.3.1 异步发送请求

异步发送请求是 Ajax 最核心的内容，Ajax 中的第一个字母就是 Asynchronous（异步）的缩写，这也正说明了 Ajax 的核心。Ajax 使用 XMLHttpRequest 对象异步发送请求。在某种程度上，XMLHttpRequest 对象就是 Ajax 的核心，也是 Ajax 技术中唯一的新概念。Ajax 正是 XMLHttpRequest 这个新对象结合 JavaScript，DOM 和 CSS 后组成的新技术。

与 JavaScript 相似的语言还有 JScript 和 ECMAScript。它们的核心语法相似，作用也相似，只是在适应的浏览器以及各自的特性上存在小小的区别。XMLHttpRequest 在不同浏览器中的实现也不相同，因而创建 XMLHttpRequest 对象的方法也存在区别。

关于 XMLHttpRequest 更详细的信息，请参看第 9 章。为了使用 XMLHttpRequest 对象，必须先创建 XMLHttpRequest 对象。创建该对象的代码如下：

```
//创建 XMLHttpRequest 对象

function createXMLHttpRequest()
{
    //对于 Mozilla 浏览器

    if(window.XMLHttpRequest)
    {
        //直接使用 XMLHttpRequest 函数来创建 XMLHttpRequest 对象

        XMLHttpRequest = new XMLHttpRequest();
    }
    //对于 IE 浏览器

    else if (window.ActiveXObject)
    {
        try
        {
            //使用 ActiveXObject 函数创建浏览器

            XMLHttpRequest = new ActiveXObject("Msxml2.XMLHTTP");
        }
        catch (e)
        {
            //如果出现异常，再次尝试以如下方式创建 XMLHttpRequest 对象

            try
            {
                XMLHttpRequest = new ActiveXObject("Microsoft.XMLHTTP");
            }
            catch (e)
            {
                //如果出现异常，再次尝试以如下方式创建 XMLHttpRequest 对象
            }
        }
    }
}
```

```
{  
  
}  
  
}  
  
}  
  
}
```

前面已经讲过，XMLHttpRequest 在不同浏览器中的实现机制不同，因而在不同的浏览器中创建 XMLHttpRequest 对象的方式也不相同。虽然上面的代码尽量兼顾不同浏览器的实现，但不排除有些浏览器不支持上面的创建方法。

一旦 XMLHttpRequest 对象创建成功，系统便可以使用 XMLHttpRequest 发送请求。XMLHttpRequest 请求与传统请求不同，传统上发送请求需要提交表单或者加载新的地址，而 XMLHttpRequest 发送请求则完全通过 JavaScript 代码完成，从而避免了页面的刷新——这也是异步发送请求的核心。

XMLHttpRequest 对象包含 send 方法，用于发送请求。在发送请求之前，应先与请求的 URL 取得连接，XMLHttpRequest 通过 open 方法打开与请求 URL 的连接。下面是使用 XMLHttpRequest 发送请求的 JavaScript 代码：

```
function sendRequest()  
{  
    //input 是个全局变量，对应于聊天信息的输入文本框  
    //调用聊天信息输入文本框的 value 属性，获取文本框的内容  
    var chatMsg = input.value;  
    //完成 XMLHttpRequest 对象的初始化  
    createXMLHttpRequest();  
    //定义请求的 URL 变量  
    var url = "chat.do";  
    //通过 open 方法取得与服务器的连接  
    //本系统发送 POST 请求  
    XMLHttpRequest.open("POST", url, true);  
    //发送 POST 请求时，应该增加该文件头  
    XMLHttpRequest.setRequestHeader("Content-Type","application/x-www-form-
```

```
urlencoded");  
  
//指定 XMLHttpRequest 状态改变时的处理函数  
XMLHttpRequest.onreadystatechange = processResponse;  
  
//发送请求后，将聊天信息的输入文本框清空  
input.value="";  
  
//发送请求，send 的参数包含许多 key-value 对  
  
//即以“请求参数名=请求参数值”的形式发送请求参数  
XMLHttpRequest.send("chatMsg=" + chatMsg); // 发送请求  
}
```

上面的代码使用 `open` 方法打开与服务器 URL 的连接。因为本系统采用 POST 发送请求参数，因此在请求里增加了 `Content-Type` 请求头，并将该请求头的值设为“`application/x-www-form-urlencoded`”，这是为了保证对请求参数采用合适的格式进行发送。下面是使用 `XMLHttpRequest` 发送请求的步骤：

- step 1** 使用 `open` 方法连接服务器 URL。
- step 2** 调用 `setRequestHeader` 方法为请求设置合适的请求头。根据不同的请求，可能需要设置不同的请求头。
- step 3** 使用回调函数。所谓回调函数，就是用于检测 `XMLHttpRequest` 状态的函数，当 `XMLHttpRequest` 状态发生改变时，该函数将自动执行。
- step 4** 执行 `send` 方法发送请求。

### 2.3.2 解决多余刷新的问题

多余刷新在本聊天室的副作用还不是十分明显，因为本系统的界面修饰相当简陋，没有多余的图片等页面资源。即使对于如此简陋的界面，一样可以对比两种模式下数据的流量。

对于上面的 JSP 聊天室，控制器处理用户请求后，转发到另一个 JSP 页面来显示处理结果；而对于 Ajax 的应用，控制器可以不再转发请求，因为仅需要生成较少数据的响应，控制器可以自己生成响应数据，此时服务器生成的不再是页面内容，而仅是必需的数据。

Ajax 主要用于改善用户体验，是一种表现层技术，并不会影响到底层的技术。对于 J2EE 应用而言，使用 Ajax 并不需要对中间层的任何组件做任何修改，更不需要对底层的 DAO 对象、Domain Object 进行修改。使用 A

jax 和使用 Hibernate, IBAITIS 或者 Spring 等框架没有任何冲突, 结合 Ajax 技术后的 J2EE 应用将更加完美, 带给用户更好的体验。Ajax 也可以与 Struts, WebWork 和 JSF 等框架结合使用。事实上, Struts 和 JSF 将在未来的版本中提供对 Ajax 更好的支持。

对于本系统而言, 系统的业务逻辑组件 ChatService 没有任何改变, 此处不再赘述。控制器 ChatServlet 则有了简单的改变: 对于 Ajax 系统而言, 服务器响应的不再是整个页面内容, 而仅是必需的数据, ChatServlet 不能将请求转发到 chat.jsp 页面。此处, ChatServlet 有两个选择:

直接生成简单的响应数据。

转向一个简单的 JSP 页面, 使用 JSP 页面生成简单的响应。

本节将给出两种实现方式, 用户可根据自己的需求进行选择。

### 2.3.2.1 直接使用控制器生成响应数据

在这种模式下, Servlet 直接通过 response 获取页面输出流, 通过输出流生成字符响应。在这种方式下, 无须转发请求, 系统处理更加简单:

```
//聊天使用的 Servlet, 继承 HttpServlet
public class ChatServlet extends HttpServlet
{
    //Servlet 所使用的服务响应方法
    public void service(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException
    {
        //设置编码方式, XMLHttpRequest 对象总采用 UTF-8 方式发送请求
        request.setCharacterEncoding("UTF-8");
        //获取请求参数: 聊天信息
        String msg = request.getParameter("chatMsg");
        //如果聊天信息不为空
        if ( msg != null && !msg.equals(""))
        {
            //通过 session 获取当前的聊天用户
            String user = (String)request.getSession(true).getAttribute("user");

            //将聊天信息添加到系统当前的聊天记录中
            ChatService.instance().addMsg(user, msg);
        }
        //设置中文输出流
        response.setContentType("text/html;charset=GBK");
        //获取页面输出流
        PrintWriter out = response.getWriter();
        //将当前系统的聊天记录输出到页面
```

```
        out.println(ChatService.instance().getMsg());
    }
}
```

该 Servlet 是一个非常简单的 Servlet，获取请求参数，调用 ChatService 对象的业务方法，输出所有的聊天记录，但请注意该 Servlet 与完全生成视图的 Servlet 的区别：该 Servlet 没有生成任何 HTML 标签，没有生成任何页面效果。在这种情况下，也可以使用 Servlet 来生成客户响应。上面的代码有两个值得注意的地方：

Ajax 使用 XMLHttpRequest 发送请求，XMLHttpRequest 发送请求时，所有参数都以 UTF-8 编码方式发送，因此 request 的 setCharacterEncoding 方法设置解码方式。通过设置 UTF-8 的解码方式，才可以正确获取所有的请求参数。

生成响应时，一定要使用 response 的 setContentType 方法设置页面内容和编码方式。尤其值得注意的是：不能仅使用 response.setHeader("Charset","GBK") 语句。仅使用该语句，系统采用 GBK 编码，但并没有确保页面是普通的 HTML 页面。因为本书是在简体中文 Windows 环境下开发本系统的，故采用 GBK 编码方式。

对于上面的控制器而言，虽然生成了表现层内容，但并未完整地生成 JSP 页面，而是返回了模型数据，因而无须使用额外的 JSP 页面。



因为该响应数据是普通文本数据，而且相当简单，因而可以直接使用控制器生成客户端响应。但如果需要生成的响应非常复杂，即响应生成的内容量大，而且具有丰富的表现格式，则应该考虑将请求转发到 JSP 页面，让 JSP 页面负责生成响应。对于是否需要由 JSP 生成响应，不可一概而论，而应取决于响应的数据量以及表现格式。

### 2.3.2.2 控制器转发到简单 JSP 页面生成响应

对于当前范例，这种做法是多此一举，控制器将请求转发到另外的 JSP 页面，而 JSP 页面仅负责输出聊天信息，下面是这种用法下的控制器代码：

```
public class ChatServlet extends HttpServlet
{
    public void service(HttpServletRequest request, HttpServletResponse
        response) throws IOException, ServletException
    {
        //设置解码格式
        request.setCharacterEncoding("UTF-8");
        //读取用户发送的聊天信息
```



```
String msg = request.getParameter("chatMsg");
//如果发送的信息不为空
if ( msg != null && !msg.equals(""))
{
    String user = (String)request.getSession(true).getAttribute("user");
    ChatService.instance().addMsg(user, msg);
}
//将聊天记录设置成 request 属性
request.setAttribute("chatList", ChatService.instance().getMsg());
//转发请求
forward("/chatreply.jsp", request, response);
}
//用于控制 forward 请求的私有函数
private void forward(String url, HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException
{
    ServletContext sc = getServletConfig().getServletContext();
    RequestDispatcher rd = sc.getRequestDispatcher(url);
    rd.forward(request,response);
}
}
```

控制器将聊天信息设置成 request 属性，然后在 JSP 页面中输出该聊天信息。JSP 代码如下：

```
<%@ page contentType="text/html;charset=GBK" errorPage="error.jsp"%>

//输出当前的聊天信息

${requestScope.chatList}
```

这个 JSP 页面的作用也相当有限，仅完成简单的输出，因此使用 JSP 页面并不是十分必要。

### 2.3.3 解析服务器响应

服务器响应生成简单的文本，而 `XMLHttpRequest` 包含一个属性：`responseText`，该属性对应服务器响应生成的文本。在解析服务器响应之前，必须先判断服务器响应是否完成以及响应是否正确，例如，生成 404 等错误响应是没有意义的。为了判断服务器响应是否完成，响应是否正确，`XMLHttpRequest` 同样提供了两个属性。  
**readyState**：判断服务器响应的状态，其中 4 表明响应完成。

**status**：判断服务器响应对应的状态码，其中 200 表明响应正常，而 404 表明资源丢失，500 表明内部错误等。

关于 `XMLHttpRequest` 的详细介绍可以参考第 9 章。

判断完响应状态后，可以使用 `responseText` 方法获取服务器响应文本，并将该文本输出到页面显示。下面是解析、处理服务器响应的代码：

//用于处理服务器响应的程序

```
function processResponse()
{
    //如果服务器响应已经完成
    if (XMLHttpRequest.readyState == 4)
    {
        // 判断对象状态，如果服务器生成了正常响应
        if (XMLHttpRequest.status == 200)
        {
            //信息已经成功返回，开始处理信息
            //将聊天文本域的内容设置成聊天信息
            document.getElementById("chatArea").value = XMLHttpRequest.responseText;
        }
        else
        {
            //页面不正常

            window.alert("您所请求的页面有异常。");
        }
    }
}
```

此时，浏览器的页面通过 JavaScript 与服务器的通信基本完成。客户端通过 sendRequest 函数向服务器发送请求，服务器通过 ChatServlet 处理用户请求，处理完用户请求后，有两种做法：Servlet 直接生成响应，或者将请求转发到 JSP 页面生成响应。客户端通过 processResponse 处理服务器响应。

### 2.3.4 何时发送请求

虽然定义了发送请求的方法，但没有定义何时发送请求。根据聊天室的特点，请求应该是需要定时发送的，因为即使本人没有参与聊天，他也希望看到其他人的聊天记录，但该请求与前面介绍的请求存在少许差别，因为这种定时发送的请求无须读取聊天记录，无须发送聊天信息。下面是这种定时发送请求的代码：

```
//定义定时发送的请求
function sendEmptyRequest()
{
    //创建 XMLHttpRequest 对象
    createXMLHttpRequest();
    //定义服务器响应的 URL
```

```
var url = "chat.do";
//建立与服务器的连接
XMLHttpRequest.open("POST", url, true);
//设置发送请求的参数格式
XMLHttpRequest.setRequestHeader("Content-Type","application/x-www-form-
    urlencoded");
//指定响应处理函数
XMLHttpRequest.onreadystatechange = processResponse;
//发送请求
XMLHttpRequest.send(null);
setTimeout("sendEmptyRequest()", 800);
}
```

注意，sendEmptyRequest 函数在最后调用了 setTimeout("sendEmptyRequest()", 800)，setTimeout 是 JavaScript 的计时器，该代码表示系统将在 0.8s 后再次执行 sendEmptyRequest 函数。因此，该函数一旦开始执行就不会停止：因为每次函数执行结束后，都将在 0.8s 后再次调用该函数。自动发送的请求应在进入聊天室后立即发送，因此将该函数定义在页面加载时触发，也就是指定在 body load 时触发即可。

除此之外，还需要获取用户聊天信息，需发送带参数的请求。这种请求应该定义在单击“提交”按钮或在聊天文本框中按下回车键时发送。要实现按下回车键后发送请求很简单：只需要为该键定义 onclick 事件即可。如需在文本框中按下回车键时发送请求，则应为聊天文本框指定键盘处理函数，该函数监控文本框的所有键盘事件。键盘处理函数的代码如下：

```
//键盘处理函数

function enterHandler(event)
{
    //定义键盘中发出事件的键

    var keyCode = event.keyCode ? event.keyCode : event.which ? event.which :
        event.charCode;

    //回车键的代码为 13,如果按下了回车键

    if (keyCode == 13)
    {
        sendRequest();
    }
}
```

整个聊天 HTML 页面的代码如下：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=GBK">
<title>聊天页面</title>
</head>
<!-- body 的 load 事件中发送定时发送的请求 -->
<body onload = "sendEmptyRequest()">
<table width="780" border="1" align="center">
<tr>
<td><p align="center">聊天页面</p>
<!-- 下面定义聊天使用的文本域，该文本域用于显示当前聊天信息 -->
<p align="center">
<textarea name="chatArea" cols="100" rows="30" readOnly></textarea>
</p>
<div align="center">
<!-- 下面是输入聊天信息所使用的文本，并为 onKeyPress 事件指定了监听函数 -->
<input name="chatMsg" type="text" size="90" onKeyPress="enterHandler(event);">
<!-- 下面是输入聊天信息的文本框，并为 onclick 事件指定了监听函数 -->
<input type="button" name="button" value="提交" onclick="sendRequest();">
</div>
<p>&nbsp;</p>
</td>
</tr>
</table>
<script>
//将输入文本框定义成 input 变量
var input = document.getElementById("chatMsg");
//将焦点定位在聊天输入框内
input.focus();
//系统使用的 XMLHttpRequest 对象
var XMLHttpRequest;
//创建 XMLHttpRequest 对象
function createXMLHttpRequest()
{
//对于 Mozilla 浏览器
if(window.XMLHttpRequest)
{
//直接使用 XMLHttpRequest 函数来创建 XMLHttpRequest 对象
XMLHttpRequest = new XMLHttpRequest();
}
}
```

```
//对于 IE 浏览器
else if (window.ActiveXObject)
{
    try
    {
        //使用 ActiveXObject 函数创建浏览器
        XMLHttpReq = new ActiveXObject("Msxml2.XMLHTTP");
    }
    catch (e)
    {
        //如果出现异常，再次尝试以如下方式创建 XMLHttpRequest 对象
        try
        {
            XMLHttpReq = new ActiveXObject("Microsoft.XMLHTTP");
        }
        catch (e)
        {
        }
    }
}
}

function sendRequest()
{
    //input 是个全局变量，对应聊天信息的输入文本框
    //调用聊天信息输入文本框的 value 属性获取文本框的内容
    var chatMsg = input.value;
    //完成 XMLHttpRequest 对象的初始化
    createXMLHttpRequest();
    //定义请求的 URL 变量
    var url = "chat.do";
    //通过 open 方法取得与服务器的连接
    //本系统发送 POST 请求
    XMLHttpReq.open("POST", url, true);
    //发送 POST 请求时应该增加该文件头
    XMLHttpReq.setRequestHeader("Content-Type", "application/x-www-
        form-urlencoded");
    //指定 XMLHttpRequest 状态改变时的处理函数
    XMLHttpReq.onreadystatechange = processResponse;
    //发送请求后，将聊天信息的输入文本框清空
    input.value="";
    //发送请求，send 的参数包含许多的 key-value 对
    //即以“请求参数名=请求参数值”的形式发送请求参数
    XMLHttpReq.send("chatMsg=" + chatMsg); // 发送请求
}
```

```
//定义定时发送的请求
function sendEmptyRequest()
{
    //创建 XMLHttpRequest 对象
    createXMLHttpRequest();
    //定义服务器响应的 URL
    var url = "chat.do";
    //建立与服务器的连接
    XMLHttpRequest.open("POST", url, true);
    //设置发送请求的参数格式
    XMLHttpRequest.setRequestHeader("Content-Type","application/x-www-
        form-urlencoded");
    //指定响应处理函数
    XMLHttpRequest.onreadystatechange = processResponse;
    //发送请求
    XMLHttpRequest.send(null);
    setTimeout("sendEmptyRequest()", 800);
}
//用于处理服务器响应的程序
function processResponse()
{
    //如果服务器响应已经完成
    if(XMLHttpRequest.readyState == 4)
    {
        //判断对象状态，如果服务器生成了正常响应
        if (XMLHttpRequest.status == 200)
        {
            //信息已经成功返回，开始处理信息
            //将聊天文本域的内容设置成聊天信息
            document.getElementById("chatArea").value = XMLHttpRequest.responseText;
        }
        else
        {
            //页面不正常
            window.alert("您所请求的页面有异常。");
        }
    }
}
//键盘处理函数
function enterHandler(event)
{
    //定义键盘中发出事件的键
    var keyCode=event.keyCode?event.keyCode:event.which?event.which:
        event.charCode;
```

```
//回车键的代码为 13，如果按下了回车键
if (keyCode == 13)
{
    sendRequest();
}
}
</script>
</body>
</html>
```

通过上面的设置，基于 Ajax 的聊天室已基本完成。Ajax 聊天室的客户端请求在后台异步发送，客户端读取服务器响应也通过 JavaScript 完成。整个过程不会阻塞用户的聊天，即使服务器的响应变慢，客户端依然可发送请求或者查看原有的聊天记录，无须等待下载页面。图 2.5 显示了该聊天页面的运行效果。

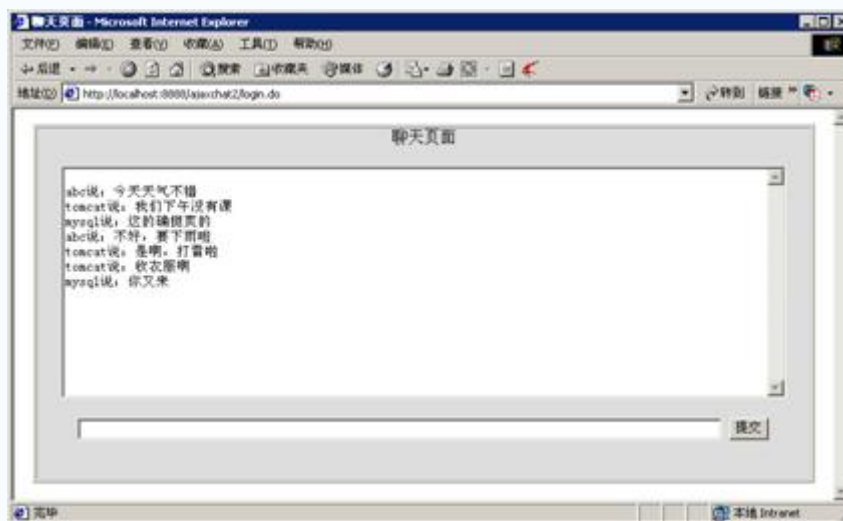


图 2.5 聊天页面的运行效果

### 2.3.5 Ajax 聊天室的特点

虽然 JSP 聊天室和 Ajax 聊天室的外观差不多，但用户聊天时可以体会到区别，Ajax 聊天室不会重复下载页面，因而不会看到不停下载新页面。正如图 2.5 所示，不管什么时候，页面的左下脚都显示“完毕”提示。

相对于传统的 JSP 聊天室而言，Ajax 聊天室的速度更快，响应更加流畅。对于复杂页面，Ajax 的优势将更加明显：Ajax 聊天室只需从服务器获取必须更新的聊天记录，而无须下载整个页面。

Ajax 聊天室的最大特点是页面无须刷新，用户感觉不到页面的下载。使用 Ajax 聊天室时，用户感觉到仿佛在使用普通的 Socket 聊天室，因为聊天室的页面无须刷新，但用户的聊天信息实时更新。这一切都依赖于 Ajax 的异步发送请求和动态更新页面。



## 2.4 传统 Web 应用与 Ajax 的对比

Ajax 技术就是所谓的 Web 2.0 技术的重要组成部分，Ajax 技术既是对传统 Web 技术的革命，也是对传统 Web 技术的一种改良和发展。引入 Ajax 技术后，不仅改进了 Web 应用的性能，也改善了用户的体验。下面就从几个方面谈谈传统 Web 应用与 Ajax 之间的不同。

**用户体验方面：**这是 Ajax 技术最大的改善之处，对于传统的 Web 应用，用户只能发送独占式请求，一旦请求发送出去，页面就处于等待状态，等待服务器响应完成，在服务器响应完成之前，客户端的浏览器只能是一片空白；而 Ajax 技术则完全不同，它允许采用异步的方法发送请求，请求的发送完全不会阻塞当前的浏览器线程，浏览器可以继续下一步操作，例如继续浏览，甚至再次发送异步请求。对于用户的体验而言，Ajax 提供了一种重大的改善，它让用户不会处于等待状态，用户会感觉自己一直与应用处于交互状态。

**响应速度：**就响应速度而言，一般人会认为 Ajax 应用的速度比传统 Web 应用要快，实际上这种说法并不完全正确。正如前面见到的，基于 Ajax 的应用需要大量增加 JavaScript 代码，大量增加 JavaScript 代码后的 Web 页面在第一次加载时速度将比传统 Web 页还慢（因为必须下载大量的 JavaScript 代码）。一旦进入该页后，响应速度便会明显提高，因为无须频繁地在各页面之间跳转，从服务器获得的仅是必须改变的数据，因此减少了冗余数据的下载，从而大幅度提高响应速度。有的人说，Ajax 包含的大量 JavaScript 代码会占用用户的大量带宽，这是相当错误的说法，Ajax 应用让页面一次下载，但可以多次重复使用。表面上看，一次下载的 JavaScript 代码量虽然增大，但从长时间来看，传统 Web 应用需要多次下载 Web 页面，需要的带宽更大。

**应用架构：**传统 Web 应用主要由 3 层组成，而增加 Ajax 技术的 Web 应用将在传统的 Web 应用上额外增加一个 Ajax 引擎，其实质就是一层 JavaScript 代码。这些 JavaScript 代码可以在客户端保存用户状态而无须使用 session，能将控制器的部分功能转移到客户端页面，但这必然会导致安全性等方面的问题，需要开发者认真对待。

**开发的代码量：**Ajax 技术的大部分功能都依赖于 JavaScript 语言实现，大量的 JavaScript 代码严重降低了程序员的开发速度。JavaScript 本身不是面向对象的编程语言，这严重限制了 JavaScript 代码的可重用性等。JavaScript 代码并没有一个完善的调试工具，这无形中也加重了程序员的负担。也有人说 Ajax 技术是通过折磨程序员来取悦用户的技术。

**服务器的负担：**传统的看法是 Ajax 技术降低了服务器的负担，因为服务器只需要生成客户端必须更新的数据。这种说法在某些场合下也许正确，但实际的情形是：大量使用 Ajax 技术的 Web 应用将导致服务器的负担大大加重，而绝不是减轻。因为 Ajax 技术往往比传统 Web 应用需要发送更多的请求，例如对于一个自动完成的输入框，传统 Web 应用无须发送任何请求，等待用户输入即可；而 Ajax 技术的情形则是：用户每输入一个字符，应用都将向服务器发送一次请求。



Ajax 技术是一种非常优秀的技术，但应该理性对待，绝不能在整个应用中盲目增加大量的 Ajax 交互。

## 2.5 小结

本章通过级联菜单应用的对比，介绍了采用 Ajax 技术带来的改进。本章还重点对比了聊天室项目，该项目既有采用传统 Web 技术开发的案例，也有采用 Ajax 技术开发的案例。通过两个案例的对比，希望读者能够体会到 Ajax 带来的技术革命：Ajax 带来的不仅有应用性能上的提高、对服务器负载的降低，还有对用户的体贴。

从下一章开始，将带着读者深入学习 Ajax 的各种相关知识，包括 JavaScript, XML, DOM 和 CSS 样式单等。当然，这些知识很多都是“古老”的知识，如果读者对某一章的内容已经非常熟悉，则可跳过该章，直接进入下一章。

## 第 17 章 基于 JSON-RPC-Java 的 Ajax 应用：在线相册

- ◆实现应用 Hibernate 持久层
- ◆以 Hibernate 框架为基础实现 DAO 组件
- ◆实现 JSON-RPC-Java 所需的服务器处理类
- ◆处理用户登录
- ◆处理用户注册
- ◆处理相片的上传
- ◆获得用户相片

本章将以一个基本的 Ajax 应用——在线相册为例，介绍如何使用 JSON-RPC-Java 开发 Ajax 应用，本应用的持久层采用 Hibernate 框架实现，但没有整合 Spring 应用。这是因为 JSON-RPC-Java 暂时还不能直接调用 Spring 容器中的 Bean，这一点可能也会限制 JSON-RPC-Java 框架的广泛使用。

JSON-RPC-Java 能将普通的 Java 对象的方法直接暴露给 JavaScript 客户端，因此本应用将使用一个业务逻辑组件来封装业务逻辑方法，而这些业务逻辑方法将被暴露给远程 JavaScript 客户端。

### 17.1 实现 Hibernate 持久层

正如本书前面所介绍的，本书总是力求将 Ajax 技术融入传统的 J2EE 应用中，通过 Ajax 技术改善传统的 J2EE 应用。因此，本书不会考虑使用 JDBC 那种原始的持久化技术。采用 Hibernate 的 ORM 框架来完成持久化工作是目前最流行而且最实用的持久化策略。

#### 17.1.1 设计 Hibernate 的持久化类

本应用需要两个表，这两个表分别用于存放用户信息和相片信息。用户表里主要保存了用户的用户名和密码等信息。对于一个实际使用的在线相册系统，可能还需要一些用户的详细资料、注册时间和最后访问时间等信息，但本应用中不打算保存这些详细信息，这对于应用的实现没有丝毫影响。

本应用的相片表里则需要保存相片的标题、相片对应的文件名以及该相片的属主。因此，用户表和相片表有主从表的约束关联，一个用户可对应于多张相片。相片所对应的持久化类如下：

```
public class Photo
{
    //相片的标识属性
    private int id;
```

```
//相片的标题

private String title;

//相片的文件名

    private String fileName;

//相片的属主

    private User user;

//标识属性对应的 setter 方法

public void setId(int id)

{

    this.id = id;

}

//省略其他属性的 setter 方法

.....

//相片属主对应的 setter 方法

public void setUser(User user)

{

    this.user = user;

}

//标识属性的 getter 方法

public int getId()

{

    return (this.id);

}

//省略其他属性的 getter 方法

.....

//相片属主的 getter 方法

public User getUser()
```

```
{  
    return (this.user);  
}  
}
```

上面的 **Photo** 类中包含了一个 **User** 类型的属性，该属性指向另一个持久化类 **User**。从相片到用户是多对一的关联，因此每个 **Photo** 都可以访问对应的 **User** 实例。上面的持久化类省略了其他普通属性的 **setter** 和 **getter** 方法。如果实际使用，应该补充 **title** 和 **fileName** 两个属性的 **setter** 和 **getter** 方法。

相册用户对应的持久化类的代码如下：

```
public class User  
{  
    //用户的标识属性  
    private int id;  
    //用户名  
    private String name;  
    //用户密码  
    private String pass;  
    //该用户对应的相片  
    private Set<Photo> photos = new HashSet<Photo>();  
    //标识属性的 setter 方法  
    public void setId(int id)  
    {  
        this.id = id;  
    }  
    //关联相片的 setter 方法  
    public void setPhotos(Set<Photo> photos)  
    {  
        this.photos = photos;  
    }  
}
```

```
}

//此处省略其他属性的 setter 方法

.....

//标识属性的 getter 方法

public int getId()

{

    return (this.id);

}

//关联属性：Photo 的 getter 方法

public Set<Photo> getPhotos(){

    return (this.photos);

}

//此处省略其他属性的 getter 方法

.....

}
```

因为一个用户可以对应多个相片实例，因此用户持久化类里增加了一个 Set 属性，该属性用于保存当前用户所有相片的引用。上面持久化类的代码省略了普通属性的 setter 和 getter 方法。

一旦完成了上面持久化类的定义，增加 Hibernate 配置文件和映射文件后，便可以进行持久层访问了。

### 17.1.2 完成配置文件和映射文件

Hibernate 进行持久化操作需要两种 XML 文件：一种是进行数据访问的配置文件，用于指定 Hibernate 的全局属性，例如连接数据库所用的驱动、URL、用户名和密码等；另一种 XML 文件是 Hibernate 的映射文件，用于定义持久化类和数据表之间的映射关系，只有定义了这种映射后，Hibernate 才能以面向对象的方式操作关系数据库。

用户和相片之间的关联是一个典型的 1：N 关联，对于 1：N 关联，采用双向关联性能更好，最好做成反向的双向关联，即 1 的一端只能访问 N 的一端，但不能控制关联关系。

下面是 Photo 持久化类的映射文件，这个映射文件增加 many-to-one 引用关联类：

```
<?xml version="1.0" encoding="GBK"?>

<!-- Hibernate 映射文件的根元素 -->

<!DOCTYPE hibernate-mapping PUBLIC

    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"

    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="org.yeeke.model">

<!-- 每个 class 元素映射一个持久化类 -->

<class name="Photo" table="photo_table">

    <!-- 定义标识属性 -->

    <id name="id">

        <!-- 指定主键生成器策略 -->

        <generator class="identity"/>

    </id>

    <!-- 定义普通属性映射 -->

    <property name="title"/>

    <property name="fileName"/>

    <!-- 定义 N:1 关联属性，必须使用 column 指定外键列的列名-->

    <many-to-one name="user" column="owner_id" not-null="true"/>

</class>

</hibernate-mapping>
```

下面是 User 类的持久化映射文件，因为一个 User 对应于多张相片，因此增加一个 Set 元素，Set 元素对应于关联类 Photo。映射文件的代码如下：

```
<?xml version="1.0" encoding="GBK"?>

<!-- Hibernate 映射文件的根元素 -->

<!DOCTYPE hibernate-mapping PUBLIC

    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"

    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
```

```
<hibernate-mapping package="org.yeeke.model">

<!-- 每个 class 元素映射一个持久化类 -->

<class name="User" table="user_table">

    <!-- 映射标识属性 -->

    <id name="id">

        <!-- 指定主键生成器策略 -->

        <generator class="identity"/>

    </id>

    <!-- 下面映射了两个普通属性 -->

    <property name="name" unique="true"/>

    <property name="pass"/>

    <!-- set 映射 1:N 关联，inverse="true" 此端无法控制关联 -->

    <set name="photos" inverse="true">

        <!-- 指定外键列的列名 -->

        <key column="owner_id"/>

        <one-to-many class="Photo"/>

    </set>

</class>

</hibernate-mapping>
```

当进行双向的 1 : N 关联时，两边都应该指定外键列的列名。因为两边指定的外键列实际上是同一列，因此两边指定的列名应该相同。



对于双向的 1 : N 关联，两边指定的外键列的列名应该相同。

上面的映射文件仅定义了 Java 类和数据表之间的对应关系，但连接数据库的属性，例如数据库驱动、数据库服务的 URL、数据库用户名和密码等信息，依然没有配置，这些通用的配置信息是在另一个配置文件中指定的。

Hibernate 还需要一个公用的配置文件，该文件的代码如下：

```
<?xml version="1.0" encoding="GBK"?>

<!-- Hibernate 配置文件的 DTD 等信息 -->

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<!-- Hibernate 配置文件的根元素 -->

<hibernate-configuration>

<!-- 配置 Hibernate 的 SessionFactory 的属性 -->

<session-factory>

    <!-- 指定所连接数据库的方言 -->

    <property name="dialect">org.hibernate.dialect.MySQLDialect

    </property>

    <!-- 指定连接数据库所用的驱动 -->

    <property name="connection.driver_class">com.mysql.jdbc.Driver

    </property>

    <!-- 指定所连接数据库服务的 URL-->

    <property name="connection.url">jdbc:mysql://localhost:3306/album

    </property>

    <!-- 指定登录数据库的用户名-->

    <property name="connection.username">root</property>

    <!-- 指定登录数据库的密码-->

    <property name="connection.password">32147</property>

    <!-- 下面的配置信息用于配置 C3P0 连接池 -->

    <property name="c3p0.max_size">5</property>
```



```
<property name="c3p0.min_size">1</property>

<property name="c3p0.timeout">5000</property>

<property name="c3p0.max_statements">50</property>

<property name="c3p0.idle_test_period">2000</property>

<property name="c3p0.acquire_increment">1</property>

<property name="c3p0.validate">false</property>

<!-- 配置 Hibernate 的其他属性 -->

<property name="show_sql">false</property>

<property name="hbm2ddl.auto">update</property>

<!-- 列出所有的持久化映射文件 -->

<mapping resource="User.hbm.xml"/>

<mapping resource="Photo.hbm.xml"/>

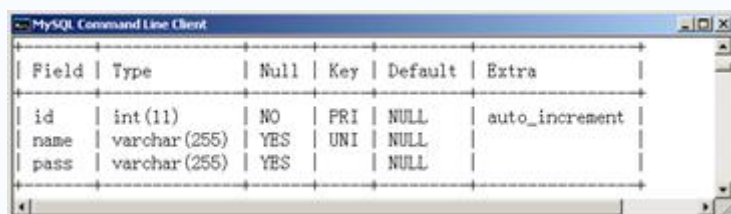
</session-factory>

</hibernate-configuration>
```

上面的配置文件配置了一个 `session-factory` 元素，在该元素里详细配置了连接数据库所需的各种信息。`SessionFactory` 实际上是 `Hibernate` 进行持久化访问的根本，它是数据库编译后的内存镜像。通常，一个数据库对应一个 `SessionFactory`。

### 17.1.3 数据库的设计

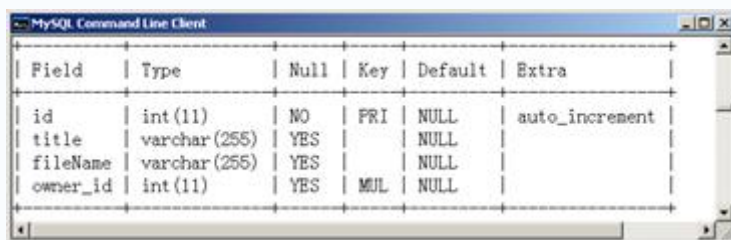
根据 `Hibernate` 持久化类以及对应的映射文件，基本上可以得出该应用的数据表结构：该应用的数据库需要两个数据表，分别是用户表和相片表。其中，用户表用于保存相册的每个用户，相片表则用于保存每张相片。每条相片记录都需要保留一个用户外键，用于定义两个表之间的主外键约束关系。用户表的表结构如图 17.1 所示。



Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
name	varchar(255)	YES	UNI	NULL	
pass	varchar(255)	YES		NULL	

图 17.1 用户表的表结构

这个表仅保存用户的基本信息、用户主键、用户名和密码。系统不允许有同名用户，因此还为用户名增加了非空约束。相片表的表结构如图 17.2 所示。



Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
title	varchar(255)	YES		NULL	
fileName	varchar(255)	YES		NULL	
owner_id	int(11)	YES	MUL	NULL	

图 17.2 相片表的表结构

相片表里增加了 owner\_id 列，该列参照用户表的 id 列，从而定义用户表和相片表的主外键约束。为了保证删除用户之后，该用户对应的相片也被全部删除，外键约束增加了级联删除的定义。

## 17.2 实现 DAO 组件

本应用的持久层访问依然依赖于 DAO 组件，DAO 组件提供了数据库访问能力，主要是对各自数据表的 CRUD 方法。

DAO 组件建立在 Hibernate 框架基础之上，因此 DAO 的每个 CRUD 方法都需要提供 Hibernate 的 Session 参数。同时，由于 JSON-RPC-Java 框架要求处理类都实现 java.io.Serializable 接口，包括处理类的依赖类。因此，这些 DAO 组件的实现类都实现了 java.io.Serializable 接口。DAO 接口里是一些方法的声明。

### 17.2.1 DAO 接口定义

DAO 接口仅定义 DAO 组件应该包含哪些方法，而不会对这些方法提供实现。使用 DAO 接口的主要目的是为了实现更好的解耦。用户 DAO 接口的代码如下：

```
public interface UserDao
{
    /**
     * 保存用户
     * @param sess 保存用户所用的 Session
     * @param p 需要保存的用户
     */
}
```

```
void save(Session sess, User p);

/**
 * 删除用户
 * @param sess 删除用户所用的 Session
 * @param p 需要删除的用户
 */

void delete(Session sess, User p);

/**
 * 删除用户
 * @param sess 删除用户所用的 Session
 * @param p 需要删除的用户 ID
 */

void delete(Session sess, int id);

/**
 * 修改用户
 * @param sess 修改用户所用的 Session
 * @param p 需要修改的用户
 */

void update(Session sess, User p);

/**
 * 根据用户名查找用户
 * @param sess 查找用户所用的 Session
 * @param name 需要查找的用户的用户名
 * @return 查找到的用户
 */

User findByName(Session sess,String name);

}
```

相片 DAO 接口的代码如下：

```
public interface PhotoDao
{
    //每页显示的相片数
    public final static int PAGE_SIZE = 8;

    /**
     * 保存相片
     * @param sess 修改相片所用的 Session
     * @param p 需要保存的相片
     */
    void save(Session sess, Photo p);

    /**
     * 删除相片
     * @param sess 删除相片所用的 Session
     * @param p 需要删除的相片
     */
    void delete(Session sess, Photo p);

    /**
     * 删除相片
     * @param sess 删除相片所用的 Session
     * @param p 需要删除的相片 ID
     */
    void delete(Session sess, int id);

    /**
     * 修改相片
     * @param sess 修改相片所用的 Session
     * @param p 需要修改的相片
     */
}
```

```
*/  
  
void update(Session sess, Photo p);  
  
/**  
 * 查询指定用户、指定页的相片  
 * @param sess 查询相片所用的 Session  
 * @param user 需要查询相片的用户  
 * @param pageNo 需要查询的指定页  
 * @return 查询到的相片  
 */  
  
List findByUser(Session sess, User user, int pageNo);  
  
}
```

这些接口定义了 DAO 组件应该实现的方法，但没有给出具体实现，具体的实现依赖于 DAO 接口的实现类。

### 17.2.2 完成 DAO 组件的实现类

DAO 组件的实现依赖于 Hibernate ORM 框架，借助于每个方法声明里的 Session 实例，可以很方便地实现这些 DAO 方法。因为应用的事务需要在业务逻辑组件中控制，因此 Hibernate Session 在业务逻辑组件里获得，然后传给 DAO 组件。

DAO 组件的实现类需要 Hibernate Session，并不负责 Session 的实例化，DAO 实例的 Session 实例是从业务逻辑组件中传过来的。下面是 UserDao 实现类的代码：

```
//UserDao 的实现类，实现 UserDao 接口和 Serializable  
  
public class UserDaoImpl implements UserDao, Serializable  
{  
    //通过 Session 保存指定 User  
  
    public void save(Session sess, User u)  
    {  
        sess.save(u);  
    }  
}
```

//通过 Session 删除指定 User

```
public void delete(Session sess, User u)
{
    sess.delete(u);
}
```

//通过 Session 根据指定 ID 删除指定 User

```
public void delete(Session sess, int id)
{
    sess.delete(sess.get(User.class, new Integer(id)));
}
```

//通过 Session 更新指定的 User

```
public void update(Session sess, User u)
{
    sess.update(u);
}
```

//通过 Session 来查找指定用户名对应的 User

//因为 User 表中的 name 列有唯一约束，因此通过 name 可查找唯一记录

```
public User findByName(Session sess, String name)
{
    return (User)sess.createQuery("from User u where u.name = :name")
        .setString("name", name)
        .uniqueResult();
}
}
```

下面是 PhotoDao 实现类的代码：

```
public class PhotoDaoImpl implements PhotoDao, Serializable
{
```

//通过 Session 保存指定的 Photo

```
public void save(Session sess, Photo p)
{
    sess.save(p);
}
```

//通过 Session 删除指定的 Photo

```
public void delete(Session sess, Photo p)
{
    sess.delete(p);
}
```

//通过 Session 删除指定 ID 对应的 Photo

```
public void delete(Session sess, int id)
{
    sess.delete(sess.get(Photo.class, new Integer(id)));
}
```

//通过 Session 更新指定的 Photo

```
public void update(Session sess, Photo p)
{
    sess.update(p);
}
```

//使用 Session，根据用户对应的全部 Photo 提供分页

```
public List findByUser(Session sess, User user, int pageNo)
{
    return sess.createQuery("from Photo p where p.user = :user")
        .setEntity("user", user)
        .setMaxResults(PAGE_SIZE)
        .setFirstResult(PAGE_SIZE * (pageNo - 1))
```

```
.list();  
  
}  
  
}
```

上面的 PhotoDao 组件中存在一个 findByUser(Session sess, User user, int pageNo)方法,该方法查找指定用户、指定页码的所有 Photo,该 DAO 方法的实现依赖于 Hibernate 提供的两个分页方法。

setMaxResult(int PageSize): 该方法用于设置本次选择最多选出的记录数。

setFirstResult(int startResult): 该方法用于设置选出第一条记录的位置。

对于普通的 DAO 组件,并不需要实现 java.io.Serializable 接口,但因为这两个 DAO 组件都是服务器处理类的依赖类,因此应该实现 java.io.Serializable 接口。

此处存在一个问题:为什么不在 DAO 对象中打开 Session,而在业务逻辑组件中打开 Session,然后传入 DAO 组件呢?这主要是从事务控制考虑,使用 Hibernate 进行数据库访问时,事务是通过 Session 开始的,如果 Session 在 DAO 方法中打开,则事务控制降低到 DAO 层次,对单个的 DAO 方法增加事务控制是没有意义的(考虑这样一个场景:从 A 账户向 B 账户转账,A 账户的余额减少是一个 DAO 操作,如果事务控制在 DAO 层次上,A 账户的余额减少是具有原子性的,但可能 A 账户的余额减少了,但 B 账户的余额没有增加,这将导致数据的不一致)。事务应该对整个业务逻辑操作起作用,也就是说,事务应该在整個业务逻辑层次上,绝不能降低到 DAO 层次上。

一旦完全实现了 DAO 组件,便可以此 DAO 组件为基础,来实现系统的服务器处理类。

## 17.3 实现服务器处理类

因为 JSON-RPC-Java 暂时不支持与 Spring 的整合,因此本应用没有使用 Spring 容器来管理业务逻辑组件。本应用中的业务逻辑组件是一个普通的 POJO,这个组件类负责处理所有的 Ajax 请求,实际上,JSON-RPC-Java 会将该类的方法暴露给客户端 JavaScript 使用。考虑本应用的实际情况,客户端 JavaScript 代码需要访问如下几个方法。

处理用户登录。根据用户名和密码验证用户登录是否成功。

注册用户。增加一个新的系统用户。

增加相片。为特定的用户增加对应的相片。

通过用户获得指定页的所有相片。

验证某个用户名是否可用。

实际上,这 5 个方法将被暴露给客户端 JavaScript,JavaScript 调用 5 个方法完成系统的功能。下面是服务器处理类的代码:



```
public class AlbumService implements Serializable
{
    private UserDao ud = null;
    private PhotoDao pd = null;
    //业务逻辑组件的构造器，构造业务逻辑组件时，完成 DAO 组件的实例化
    public AlbumService()
    {
        ud = new UserDaoImpl();
        pd = new PhotoDaoImpl();
    }
    /**
     * 验证用户登录是否成功，如果登录成功，则将用户名添加到 Session 中
     * @param user 登录用的用户名
     * @param pass 登录用的密码
     * @param request 直接使用的 HttpServletRequest
     */
    public boolean login(String user, String pass, HttpServletRequest request)
    {
        //使用工具类打开 Session
        Session s = HibernateUtil.currentSession();
        //开始事务
        Transaction tx = s.beginTransaction();
        //根据用户名查找用户
        User u = ud.findByName(s, user);
        //如果指定用户名的用户且密码与用户输入的密码符合
        if (u != null && u.getPass().equals(pass))
        {
            //登录成功，将用户名放入 Session 中
            request.getSession().setAttribute("user", user);
            //提交事务
            tx.commit();
            //关闭 Session
            HibernateUtil.closeSession();
            return true;
        }
        //回滚事务
        tx.rollback();
        HibernateUtil.closeSession();
        return false;
    }
    /**
     * 添加新用户
     * @param user 添加的用户名
     * @param pass 用户对应的密码
     */
}
```

```
* @param request 对于该 Web 应用的 HttpServletRequest 对象
* @return 添加用户的结果，成功返回 true，否则返回 false
*/
public boolean addUser(String user, String pass, HttpServletRequest request)
{
    try
    {
        Session s = HibernateUtil.currentSession();
        Transaction tx = s.beginTransaction();
        //创建一个新用户
        User u = new User();
        //设置新用户的属性
        u.setName(user);
        u.setPass(pass);
        //保存用户
        ud.save(s, u);
        tx.commit();
        //关闭 Session
        HibernateUtil.closeSession();
        request.getSession().setAttribute("user", user);
        return true;
    }
    catch(Exception e)
    {
        e.printStackTrace();
        return false;
    }
}

/**
* 添加相片
* @param user 添加相片的用户
* @param title 添加相片的标题
* @param fileName 添加相片的文件名
* @return 添加相片的结果，成功返回 true，否则返回 false
*/
public boolean addPhoto(String user, String title, String fileName)
{
    Session s = HibernateUtil.currentSession();
    Transaction tx = s.beginTransaction();
    //创建一个新 Photo 实例
    Photo p = new Photo();
    //设置 Photo 的属性
    p.setTitle(title);
    p.setFileName(fileName);
```

```
        p.setUser(ud.findByName(s, user));
//保存 Photo
        pd.save(s, p);
        tx.commit();
//关闭 Session
        HibernateUtil.closeSession();
        return true;
    }
}
/**
 * 根据用户获得该用户的所有相片
 * @param user 当前用户
 * @param pageNo 页码
 * @return 返回当前用户指定页的相片
 */
public List<PhotoHolder> getPhotoByUser(String user, int pageNo)
{
    Session s = HibernateUtil.currentSession();
    Transaction tx = s.beginTransaction();
//调用 DAO 方法返回指定用户、指定页的所有 Photo
    List pl = pd.findByUser(s, ud.findByName(s, user), pageNo);
//将查询结果封装成集合后返回
    List<PhotoHolder> result = new ArrayList<PhotoHolder>();
//遍历查询结果
    for (Object o : pl )
    {
        Photo p = (Photo)o;
        result.add(new PhotoHolder(p.getTitle(), p.getFileName()));
    }
    tx.commit();
    HibernateUtil.closeSession();
    return result;
}
/**
 * 验证用户名是否可用，即数据库里是否已经存在该用户名
 * @param user 需要校验的用户名
 * @return 校验该用户名后返回的字符串，该字符串会直接发送给浏览者
 */
public String valid(String user)
{
    Session s = HibernateUtil.currentSession();
    Transaction tx = s.beginTransaction();
//根据用户名查找
    User u = ud.findByName(s, user);
    if (u != null)
```

```
{
    tx.commit();
    HibernateUtil.closeSession();
    return "该用户名已经被使用了，请选择新用户名";
}
tx.rollback();
HibernateUtil.closeSession();
return "这个用户名还没有被使用，您可以使用该用户名";
}
}
```

服务器处理类在返回 `Photo` 时，并未直接返回 `Photo` 持久化类实例。根据 J2EE 规范，处于底层的 PO 实例不应该传到表现层。为了将相应的数据传到表现层，系统提供了一个简单的 `JavaBean`，这个 `JavaBean` 是一个 VO（值对象），这个 VO 封装了 `Photo` 里的基本信息。

上面的服务器处理类有两个值得注意的地方：

第一个地方是 `login` 方法声明里多出一个 `HttpServletRequest`，服务器处理类是一个 POJO，并没有与任何 `Servlet API` 耦合，此处额外多出一个 `HttpServletRequest` 确实很奇怪。要知道，这个方法将被暴露给客户端 `JavaScript` 调用，但实际上 `Java` 方法绝不可能直接暴露到 `JavaScript` 客户端。这种暴露是一个假象，是以 `JSON-RPC-Java` 框架作为中间桥梁的——关键就在这里，当客户端 `JavaScript` 代码调用该方法时，并不需要提供 `HttpServletRequest` 参数，该参数由 `JSON-RPC-Java` 框架负责提供。与之类似的是，`addUser` 方法中也有这个参数，效果类似。

第二个地方是 `HibernateUtil` 工具类，因为 `Hibernate` 的 `Session` 是一个线程不安全对象，因此将 `Session` 绑定到请求线程是一个不错的做法，`HibernateUtil` 正好提供了这个功能。下面是 `HibernateUtil` 的代码：

```
public class HibernateUtil
{
    //SessionFactory 是 Hibernate 持久化访问的基础对象，它是数据库编译后的内存镜像
    public static final SessionFactory sessionFactory;

    //静态初始化块，用于初始化 SessionFactory
    static
    {
        try
        {
```

```
//采用默认的 hibernate.cfg.xml 来启动一个 Configuration 的实例

Configuration configuration=new Configuration().configure();

//由 Configuration 的实例来创建一个 SessionFactory 实例

    sessionFactory = configuration.buildSessionFactory();

}

catch (Throwable ex)

{

    System.err.println("Initial SessionFactory creation failed." + ex);

    throw new ExceptionInInitializerError(ex);

}

}

//ThreadLocal 是隔离多个线程的数据共享，不存在多个线程之间共享资源,因此不再需要

//对线程同步

public static final ThreadLocal local = new ThreadLocal();

//获取当前线程的 Session

    public static Session currentSession() throws HibernateException

    {

        Session s = (Session)local.get();

        //如果该线程还没有 Session，则创建一个新的 Session

        if (s == null)

        {

            s = sessionFactory.openSession();

            //将获得的 Session 变量存储在 ThreadLocal 变量 local 里

            local.set(s);

        }

        return s;

    }

}
```

```
//关闭 Session
```

```
public static void closeSession() throws HibernateException {  
    Session s = (Session) local.get();  
    if (s != null)  
        s.close();  
    local.set(null);  
}  
}
```

通过这个工具类，可以保证每个请求线程只有一个 Session 对象，并且该 Session 对象被绑定到当前线程，从而避免多线程操作共享数据引发异常。

到目前为止，已经完成了服务器处理类的实现，以及处理类所依赖的 DAO 组件的实现。下面介绍如何在客户端通过 JavaScript 调用这个处理类的方法。

## 17.4 实现客户端调用

与 DWR 类似的是，JSON-RPC-Java 可将 Java 方法暴露给 JavaScript 客户端，允许从 JavaScript 客户端直接调用远程 Java 方法。暴露都是通过将 Java 对象注册成客户端的 JavaScript 对象来实现的，只是注册过程略有区别。一旦将远程对象转换成对应的 JavaScript 对象，便可以通过回调函数调用服务器的 Java 方法。

### 17.4.1 暴露 Java 对象

JSON-RPC-Java 暴露远程 Java 对象依赖于 JSONRPCBridge 和 JSONRPCServlet。其中，JSONRPCServlet 负责拦截用户请求，JSONRPCBridge 负责将拦截到的用户请求转发给用户的处理类。JSONRPCServlet 是一个有特殊功能的拦截器，JSONRPCBridge 是客户端 JavaScript 对象和 Java 对象之间的桥梁。

为了暴露服务器端的 Java 对象，应该在 web.xml 中定义 JSONRPCServlet，该 Servlet 负责拦截用户请求，即应该在 web.xml 文件增加如下的配置片段：

```
<!-- 在 web.xml 定义 JSONRPCServlet -->  
  
<servlet>  
  
<!-- 指定核心 Servlet 的名字 -->
```

```
<servlet-name>jsonServlet</servlet-name>

<!-- 指定核心 Servlet 的实现类 -->

    <servlet-class>com.metaparadigm.jsonrpc.JSONRPCServlet</servlet-class>

</servlet>

<servlet-mapping>

<!-- 指定核心 Servlet 的名字 -->

<servlet-name>jsonServlet</servlet-name>

<!-- 指定核心 Servlet 映射的 URL -->

    <url-pattern>/myjson</url-pattern>

</servlet-mapping>
```

一旦完成了上面的映射，便表示所有向/myjson 地址的请求都将发给 JSONRPCServlet 处理。因此，JSON-RPC-Java 框架中的另一个核心类 JSONRpcClient 应以此 URL 创建实例。为了创建该实例，在该应用的 JavaScript 代码库中增加了如下函数：

```
//页面加载时自动执行

function onLoad()
{
//创建 JavaScript 对象：JSONRpcClient 对象

    jsonrpc = new JSONRpcClient("myjson");

//另一个自动执行的函数

onLoadHandler();

}
```

JSONRpcClient 是 JSON-RPC-Java 框架的另一个核心类，它用于创建一个 JavaScript 对象，该对象里有所有被导出到客户端 JavaScript 的对象。因此，该对象应该在页面 load 时执行，该函数将被指定给页面的 onLoad 事件。上面函数中的 onLoadHandler( )是另一个函数，该函数负责每隔 0.5s 发送一次请求，判断对应用户是否有新相片。

完成了 JSONRpcClient 的初始化后，还必须将远程 Java 对象注册在该对象下。为了注册该对象，在 JSP 页面中使用如下代码片段：

```
<!-- 初始化 JSONRPCBridge 对象 -->

<jsp:useBean id="JSONRPCBridge" scope="session" class="com.metaparadigm.
jsonrpc.JSONRPCBridge" />

<!-- 初始化 AlbumService 对象 -->

<jsp:useBean id="as" scope="session" class="org.yeekeu.service.AlbumService"/>

<%

//注册 AlbumService 对象，将其转换为一个客户端 JavaScript 的一个 as 对象

JSONRPCBridge.registerObject("as", as);

%>
```

前面已经介绍过了，JSONRPCBridge 是 Java 对象和 JavaScript 对象之间的桥梁，它的主要作用就是注册 Java 对象，并将其转换为一个 JavaScript 对象——这并不是多么值得推崇的方式，DWR 可以在 XML 文件中定义这种注册，因而更加灵活。

完成这种注册后，AlbumService 对象将被暴露成一个 JavaScript 对象，这个对象作为 JSONRpcClient 对象的一个属性引用。因此，为了调用 AlbumService 对象的方法，可以通过如下代码来引用 AlbumService 对象：

```
//jsonrpc 是 JSONRpcClient 对象的名字，而 as 是 AlbumService 的对象

jsonrpc.as
```

### 17.4.2 处理用户登录

在 JSP 页面中提供了两个文本框，用于浏览者输入用户名和密码。一旦用户单击“登录”按钮，便通过 JavaScript 发送登录请求，这种登录请求通过调用 AlbumService 对象的 login 方法实现。下面是处理用户登录的客户端 JavaScript 函数：

```
//验证登录

function login()
{
//获得 user 输入框的值

var user = document.getElementById("user").value;

//获得 pass 输入框的值
```



```
var pass = document.getElementById("pass").value;

//进行基本校验，如果用户名为空或者密码为空

if (user == null || user == "" || pass == null || pass == "")
{
    //弹出警告框，让用户输入用户名和密码

    alert("必须先输入用户名和密码");

    return false;
}
else
{
    //如果用户已经输入了用户名和密码，则调用远程 Java 方法

    jsonrpc.as.login(logincb, user, pass);

    //将用户名赋为当前用户名

    curUser = user;
}
}
```

在上面的登录函数中有一个全局变量 `curUser`，该变量用于保存当前登录系统的用户。当调用 `AlbumService` 对象的 `login` 方法时，增加了一个参数 `logincb`。`AlbumService` 对象的 `login` 方法只有两个参数：`user` 和 `pass`。此处的 `logincb` 就是回调函数，用于执行异步调用，`logincb` 回调函数将在服务器响应完成后自动运行。下面是 `logincb` 回调函数的代码：

```
//////////登录的回调函数//////////

function logincb(result, exception)
{
    //服务器响应出错

    if(result == null && exception != null)
    {
        //提示用户服务器响应出错
    }
}
```

```
        alert(exception.message);
    }

    //服务器响应完成

    if (result != null && exception == null)
    {
        //如果登录成功

        if (result == true)
        {
            //显示用户登录成功

            alert("您已经登录成功");

            //隐藏页面中登录框

            document.getElementById("noLogin").style.display = "none";

            //显示页面中用户的控制面板

            document.getElementById("hadLogin").style.display = "";

            jsonrpc.as.getPhotoByUser(getPhotocb , curUser , curPage);
        }
        else
        {
            //登录失败，将当前用户赋空

            curUser = null;

            //弹出提示：用户名和密码不匹配

            alert("您输入的用户名和密码不匹配，请重新输入");
        }
    }
}
```

在上面的回调函数中，当用户登录成功后，再次调用了 jsonrpc.as 的另一个方法 getPhotoByUser，该方法用于请求当前用户，指定页面的相片。调用该函数的 curPage 参数是一个全局参数，该参数用于保存当前用户浏览的页码。

与所有异步调用相同，调用 getPhotoByUser 函数的第一个参数是一个回调函数 getPhotocb，该函数用于将服务器响应的全部相片显示在页面中。下面是该回调函数的代码：

```
//////////取得相片的回调函数//////////  
function getPhotocb(result, exception)  
{  
    //如果服务器处理出错  
    if(result == null && exception != null)  
    {  
        //使用提示框告诉用户处理出错  
        alert(exception.message);  
    }  
    //如果服务器处理成功  
    if (result != null && exception == null)  
    {  
        //获取页面中显示相片列表的元素  
        var list = document.getElementById("list");  
        //清空该相片列表  
        list.innerHTML = "";  
        //遍历服务器返回的相片列表  
        for (var item in result.list )  
        {  
            //拼接相片列表中的内容  
            list.innerHTML += "<div align='center'><a href='#' onClick='show(\"" +  
                result.list[item].fileName + "\" );'>" + result.list[item].title  
                + "</a></div>"  
        }  
    }  
}
```

在上面的整个过程中，完成了用户的登录。一旦用户登录，用户的登录面板便立即隐藏，使用用户的控制面板代替，并在相片列表框中列出当前用户的所有相片，而且相片列表框中的每个相片名都有对应的 JavaScript 处理，如果单击相片名，将显示出对应的相片。图 17.3 显示了未登录本系统时的界面。



图 17.3 未登录时的系统界面

正如图 17.3 所显示的，当未登录系统时，界面的最上方是用户的登录面板，用于收集用户名和密码，并处理用户登录；界面的左方是一个空的相片列表——因为还没有登录。如果用户正确输入用户名和密码并单击“登录”按钮，将看到如图 17.4 所示的提示框。



图 17.4 登录成功

一旦用户登录成功，系统便自动加载当前用户的所有相片，相片名将在左边的相片列表框中显示出来。如果单击左边相片列表框中的任一相片，将显示该相片。为了显示相片，还必须为每个相片标题增加一个 JavaScript 函数，该函数的代码如下：

```
///显示相片

function show(fileName)
{
    //在 show 图片元素中显示相片

    document.getElementById("show").src = "uploadfiles/" + fileName;
}
```

图 17.5 显示了登录成功的界面。



图 17.5 登录成功的界面

图 17.5 的上面部分已经更换成用户的控制面板，而不再是登录面板。左边的相片列表框中列出了当前用户、当前页的全部相片。为了保证每次用户添加相片成功，页面提供了一个周期性执行的函数，该函数负责获得用户的相片列表。

### 17.4.3 周期性地获得用户相片列表

获得用户的相片列表是通过 AlbumService 的 getPhotoByUser 方法完成的，只要使用定时器周期性地调用该方法即可。下面是周期性地调用该方法的 JavaScript 函数：

```
//自动请求当前用户的相片

function onLoadHandler()
{
    //只有当前用户不为空，才能获得用户的相片列表
    if ( curUser != null)
    {
        //调用远程 Java 方法，其中 getPhotocb 是回调函数
    }
}
```

```
jsonrpc.as.getPhotoByUser(getPhotocb, curUser, curPage);  
  
}  
  
//指定调用频率，0.5s 后将再次调用该函数  
  
setTimeout("onLoadHandler()", 500);  
  
}
```

该函数使用的回调函数 `getPhotocb` 也是负责将服务器响应的相片列表在页面中显示出来，因此使用的还是前面已经介绍过的回调函数，该函数保证页面可以自动加载用户上传的相片。

#### 17.4.4 处理用户注册

当进入系统时，系统显示的页面提供了一个登录面板，在登录面板中提供了一个注册面板的链接，一旦单击该链接，登录面板将切换到注册面板。将登录面板切换到注册面板的函数如下：

```
//切换登录到注册  
  
function changeRegist()  
{  
  
    //隐藏登录面板  
  
    document.getElementById("loginDiv").style.display="none";  
  
    //显示注册面板  
  
    document.getElementById("registDiv").style.display="";  
  
}
```

登录面板与注册面板的界面大同小异，只是提供了一个“注册”按钮，当单击“注册”按钮时，可发送注册申请。在注册用户之前，建议浏览者先验证用户名是否已经存在，因为本系统要求用户名不重复。因此，注册面板还提供了一个链接，该链接可用于验证用户名是否已经存在。验证用户名是否已经存在的 JavaScript 代码如下：

```
//验证用户名是否可用  
  
function valid()  
{  
  
    //获得用户文本框的值  
  
    var user = document.getElementById("user").value;
```

```
//如果还没有输入用户名

if (user == null || user == "")
{
    //弹出警告框

    alert("您还没有输入用户名");

    return false;
}
else
{
    //验证用户名是否可用

    jsonrpc.as.valid(validcb, user);
}
}
```

验证用户名是否可用使用了 AlbumService 的 valid 方法，该方法的返回值是对用户的提示文本。因此，调用该方法的回调函数的代码非常简单，直接输出服务器提示即可：

```
//验证用户名是否可用的回调函数

function validcb(result, exception)
{
    //如果服务器响应出错

    if(result == null && exception != null)
    {
        //输出出错提示

        alert(exception.message);
    }

    //如果服务器响应完成

    if (result != null && exception == null)
    {

```

```
//输出服务器响应的提示  
  
alert(result);  
  
}  
  
}
```

借助于 AlbumService 的 valid 方法，可以验证用户名在数据库中是否已经存在。如果用户名已经存在，则提示用户更换另一个用户名，否则提示用户该用户名可用。

校验完用户名是否可用后，如果用户名可用，则发送注册请求。注册请求通过调用 AlbumService 的 addUser 方法完成。下面是注册用户的 JavaScript 函数：

```
//////////处理注册//////////  
  
function regist()  
{  
    //获取页面中的 user 和 pass 值  
  
    var user = document.getElementById("user").value;  
    var pass = document.getElementById("pass").value;  
  
    //如果没有输入用户名，或没有输入密码  
  
    if (user == null || user == "" || pass == null || pass == "")  
    {  
        //弹出警告框：必须输入用户名和密码  
  
        alert("必须先输入用户名和密码");  
  
        return false;  
    }  
  
    else  
    {  
        //注册新用户，其中 registcb 是回调函数  
  
        jsonrpc.as.addUser(registcb, user, pass);  
  
        curUser = user;  
    }  
}
```



}

调用 AlbumService 的 addUser 方法后，服务器会返回一个注册成功与否的提示，注册成功返回 true，否则返回 false，下面是该回调函数的代码：

```
//////////处理注册的回调函数//////////
function registcb(result, exception)
{
    //如果服务器响应失败
    if(result == null && exception != null)
    {
        //输出服务器的异常信息
        alert(exception.message);
    }
    //如果服务器响应成功
    if (result != null && exception == null)
    {
        //如果服务器响应返回 true，即注册成功
        if (result == true)
        {
            alert("您已经注册成功");
            //启动请求用户所有相片的周期运行函数
            onLoadHandler();
            //隐藏注册面板
            document.getElementById("noLogin").style.display = "none";
            //显示用户的控制面板
            document.getElementById("hadLogin").style.display = "";
        }
        else
        {
            //如果注册失败，将当前用户赋空
            curUser = null;
            //弹出注册失败的提示
            alert("注册失败，请重试");
        }
    }
}
```

上面的 JavaScript 函数在用户注册成功时会向系统发送周期性请求，这种请求用于自动加载用户对应的全部相片。当然，一个用户新注册时没有任何相片，但可以在当前页面添加相片，一旦添加了相片，这个周期性执行的函数将可以获得该用户对应的全部相片。图 17.6 显示了用户注册成功的提示框。



17.6 注册成功

### 17.4.5 处理上传

很多人以为，Ajax 技术可以很方便地实现无刷新的文件上传。实际上，这存在一个障碍：根据安全性需要，JavaScript 代码不能访问客户端文件系统（否则，这种行为将可能导致 JavaScript 非法操作客户端用户文件）。当然，借助 Internet Explorer 里的 FSO（File System Object，文件系统对象）可以访问浏览者的文件系统，但这终究不是一个好主意：局限性太大，这种访问必须得到用户的同意。如果 JavaScript 不能访问用户文件系统，那么 XMLHttpRequest 的请求参数就无法获得文件的内容信息，只能获得文件的文件名。

XMLHttpRequest 只能将需要上传的文件名发送到服务器，但服务器获得需要上传的文件名没有任何意义，因为服务器依然不能访问客户端的文件系统。这就是通过 Ajax 技术实现无刷新文件上传的障碍，在现阶段，使用 XMLHttpRequest 实现无刷新的文件上传是不大可能的。

为了解决文件上传问题，本系统采用打开一个 JavaScript 窗口的方式，当上传成功后，该窗口自动关闭。因为打开新窗口等同于打开了另一个浏览器，因此无法获取当前浏览器中的 JavaScript 变量，这也是需要在 AlbumService 中将当前用户添加为 session 属性的原因。打开新窗口的 JavaScript 函数如下：

```
//打开上传窗口

function openUpload()
{
    //打开一个新的上传窗口
    win = window.open("upload.jsp","console","width=400,height=250");
    //将焦点移动到新窗口
    win.focus();
}
```

打开新的 JavaScript 窗口的 JSP 页面的代码如下：

```
<%@ page contentType="text/html; charset=gb2312"%>

<%
```

```
//从 Request 中读取 result 信息，result 是上传结果

String result = (String)request.getAttribute("result");

//如果上传结果不为空且其值为 success，则表明上传成功

if (result != null && result.equals("success"))

{

%>

<script>

alert("上传成功！");

window.close();

</script>

<%}%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"

"http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=gb2312">

<title>上传图片</title>

<link href="css.css" rel="stylesheet" type="text/css">

</head>

<body>

<form action="upload" method="post" enctype="multipart/form-data" name="form1">

<table width="400" height="142" border="0" >

<caption>上传图片</caption>

<tr>

<td height="25">图片标题: </td>

<td><input name="title" type="text"></td>

</tr>
```

```
<tr>

    <td height="25">浏览图片: </td>

    <td><input name="file" type="file"></td>

</tr>

<tr>

    <td colspan="2" align="center"><input type="submit" value="上传">

    &nbsp;&nbsp;&nbsp;<input type="reset" value="重设"></td>

</tr>

</table>

</form>

</body>

</html>
```

当用户单击上传图片链接后，将打开如图 17.7 所示的窗口。



图 17.7 上传图片

上面的页面提供了一个文本框，该文本框用于收集相片标题。除此之外，还提供了一个文件浏览域，这个文件浏览域用于浏览上传文件。一旦单击“上传”按钮，将向 `upload` 处发送上传图片的请求。

`upload` 是系统的一个 Servlet，该 Servlet 负责处理文件的上传，以及调用 `AlbumService` 的方法添加相片。上传还使用了另一个开源项目 `commons-fileupload`，因此应该将 `commons-fileupload.jar` 文件添加到应用的 `WEB-INF/lib` 路径下。下面是处理上传的 Servlet 的代码：

```
public class UpLoadServlet extends HttpServlet
{
```

//响应用户请求的方法，该方法既可以响应 POST 请求，也可以响应 GET 请求

```
public void service(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
{
    //用于遍历用户请求项的遍历器
    Iterator iter = null;
    String title = null;
    try
    {
        //这是 common-fileupload 的一个类，负责上传的工厂类
        FileItemFactory factory = new DiskFileItemFactory();
        ServletFileUpload upload = new ServletFileUpload(factory);
        //解析请求中的所有项
        List items = upload.parseRequest(request);
        iter = items.iterator();
    }
    catch (FileUploadException fue)
    {
        fue.printStackTrace();
    }
    //遍历请求中的每一项
    while (iter.hasNext())
    {
        FileItem item = (FileItem) iter.next();
        //如果该项是一个基本表单域，不是文件上传域
        if (item.isFormField()) {
            String name = item.getFieldName();
```

```
//如果该表单域的 name 为 title

    if (name.equals("title"))
    {
        //获得上传相片的标题

        title = item.getString("GBK");
    }
}

//处理文件上传域

else
{
    //获得 session 中的 user 属性

    String user = (String)request.getSession().getAttribute("user");

    //相片上传后在服务器上的文件名

    String serverFileName = null;

    //如果用户为空，则直接返回

    if (user == null || user.equals(""))
    {
        return;
    }

    //返回文件名

    String fileName = item.getName();

    String appden = fileName.substring(fileName.lastIndexOf("."));

    //返回文件类型

    String contentType = item.getContentType();

    //只准上传 jpg 和 gif 文件

    if (contentType.equals("image/jpeg") ||

        contentType.equals("image/gif"))
```

```
{  
//从上传文件域中获得输入流  
  
    InputStream input = item.getInputStream();  
//使用当前时间作为上传的文件名  
  
    serverFileName = String.valueOf(System.currentTimeMillis());  
//以上传文件打开一个文件输出流  
  
    FileOutputStream output = new FileOutputStream  
    (getServletContext().getRealPath("/")  
        + "uploadfiles\\" + serverFileName + appden);  
  
    byte[] buffer = new byte[1024];  
  
    int len = 0;  
//将上传文件写到服务器端  
  
    while((len = input.read(buffer)) > 0 )  
    {  
        output.write(buffer, 0, len);  
    }  
  
    input.close();  
    output.close();  
}  
  
//从 session 中获得 AlbumService 实例  
  
    AlbumService as = (AlbumService)request.getSession().  
    getAttribute("as");  
  
    if (as == null)  
    {  
        return;  
    }  
  
//在数据库中增加添加相片的记录
```

```

        as.addPhoto(user, title, serverFileName + appden);

        //转发请求

        ServletContext sc = getServletConfig().getServletContext();

        //在 request 中增加上传图片的结果

        request.setAttribute("result", "success");

        //转回 upload.jsp 页面

        RequestDispatcher rd = sc.getRequestDispatcher("/upload.jsp");

        rd.forward(request,response);

    }

}

}

}

```

当该 Servlet 处理上传成功后，会自动跳转到浏览上传的 upload.jsp 页面，并设置 result 属性，upload.jsp 页面会根据该 result 属性自动关闭。如果上传图片成功，我们将看到如图 17.8 所示的提示框。



17.8 上传成功

如果单击图 17.8 中的“确定”按钮，那么上传页面将自动关闭，系统自动回到主页面。回到主页面后，可以看到刚刚添加的相片已经被列在左边的相片列表框中。单击任一相片标题，可以看到相片在右边显示出来。

## 17.5 客户端 JSP 页面

客户端 JSP 页面是整个应用操作的核心，大部分操作都在该页面上完成。该页面负责收集用户的登录、注册信息，显示用户的相片列表，显示相应的相片，并激发用户请求。正如前面所见到的，因为该页面并没有额外的修饰，因此非常简单。该页面的代码如下：

```

<%@ page contentType="text/html; charset=GBK"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"

```



```
"http://www.w3.org/TR/html4/loose.dtd">

<jsp:useBean id="JSONRPCBridge" scope="session" class="com.metaparadigm.
jsonrpc.JSONRPCBridge" />

<jsp:useBean id="as" scope="session" class="org.yeeke.service.AlbumService"/>

<%
//将 AlbumService 暴露成客户端 JavaScript 的 as 对象
JSONRPCBridge.registerObject("as", as);
%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=gb2312">

<title>在线相册</title>

<script type="text/javascript" src="jsonrpc.js"></script>

<script type="text/javascript" src="album.js"></script>

<link href="css.css" rel="stylesheet" type="text/css">

</head>

<!-- 当页面加载时, 创建 JSONRpcClient 对象 -->

<body onLoad="onLoad();">

<table width="780" height="500" border="0" align="center" cellpadding="1" bgcolor="#000000">

<caption>在线相册</caption>

<tr bgcolor="#FFFFFF">

<td height="60" colspan="2">

<div id="noLogin">

<table width="100%" border="0">

<tr>

<td width="9%">用户名: </td>
```

设" $\lambda$

```

        增加相片</a>

</div>

</td>

</tr>

<tr bgcolor="#FFFFFF">

<td width="120" height="440" width="20%" valign="top">

<div align="center"><h3>相片列表</h3></div>

<div id="list"></div><hr>

<div align="center"><a href="#" onClick="getPhoto(-1);">上一页</a>&nbsp;

<a href="#" onClick="getPhoto(1);">下一页</a></div>

</td>

<td width="660" align="center" valign="middle">

<img id="show" src="" width="640" height="430"></img></td>

</tr>

</table>

<div align="center">

    All Rights Reserved.<br>

    版权所有 Copyright@2006 Yeeku.H.Lee <BR>

    如有任何问题和建议, <A href="mailto:kongyeeku@163.com">请 E-mail to me</A>! <BR>

    建议您使用 1024*768 分辨率, IE 5.0 以上版本使用本系统!

</div>

</body>

</html>

```

可能有读者会发现, 该页面需要激发 JavaScript 函数的地方都是直接将该事件与其对应的处理器写在页面中, 这可能并不符合显示逻辑和控制逻辑分离的规则。因为 JSON-RPC-Java 没有提供任何 JavaScript 辅助库, 这非常不方便, 所以没有将这些从 JSP 页面中分离出来。

当然，JSON-RPC-Java 框架完全可以与 Prototype.js 或者 Dojo 等 JavaScript 函数库整合使用。如果在本应用中引入 Prototype.js 或 Dojo 库，将可以做得更加完美——读者可以自行进行这项工作。

## 17.6 小结

本章通过一个实例——在线相册详细介绍了如何使用 JSON-RPC-Java 开发 Ajax 应用。本应用的持久层采用 Hibernate 框架，一样使用了 DAO 模式进行持久层访问。业务逻辑组件（也就是服务器处理类）基于 DAO 组件实现，并通过 JSONRPCBridge 暴露成一个 JavaScript 对象，从而可以在客户端调用该方法。本章还示范了如何通过 commons-fileupload 项目实现文件上传。

下一章将介绍一个非常简单的、完全封装的 Ajax 框架——AjaxTags，通过该框架的帮助，甚至无须任何 Ajax 知识，也可开发出非常专业的 Ajax 应用。借助于 AjaxTags 的帮助，完全可以以 JSP Tags 的方式来开发 Ajax 应用。

## 第 18 章 使用 AjaxTags 简化开发

- ◆了解 AjaxTags
- ◆为 AjaxTags 实现服务器响应
- ◆使用 AjaxTags 的标签
- ◆AjaxTags 的适用场景
- ◆AjaxTags 的下载和安装
- ◆几种服务器的处理类
- ◆了解 AjaxTags 的缺陷

AjaxTags 是由一系列的 JSP 标签组成的，这些标签用以简化 Ajax 应用的开发。对很多开发者而言，开发 Ajax 应用时总是从创建 XMLHttpRequest 对象开始——这是相当糟糕的行为，你在重复发明轮子，实际上，那个轮子已经有人帮你做好了！通过前面的介绍，我们知道有两种选择：一种是使用 JavaScript 函数库来简化 Ajax 应用的开发，例如 Prototype.js 和 Dojo 等；另外一种是基于 RPC 的 Ajax 应用，例如 DWR 和 JSON-RPC-Java 等。

AjaxTags 则提供了更高层次的简化，它以 Prototype.js 和其扩展项目 Scriptaculous.js 为基础，将常用的 Ajax 应用封装成简单的标签。通过使用 AjaxTags，J2EE 程序开发者可以像使用普通标签一样来获得 Ajax 功能。虽然 AjaxTags 的灵活性相对较差，但对于大部分 J2EE 应用而言，常用的 Ajax 功能（如自动完成、级联菜单等）AjaxTags 都有现成的实现，直接使用就可以了，所以我们没有理由拒绝 AjaxTags。

### 18.1 AjaxTags 的下载和安装

AjaxTags 是属于 sourceforge 的一个开源项目，它以 Prototype.js 及相关项目为基础，提供了一套简单的 JSP 标签库（早期的 AjaxTags 还需要 Struts 的支持，但今天的 AjaxTags 已经不再依赖于 Struts 了），通过这些标签库可以非常简单地开发出 Ajax 应用。

#### 18.1.1 AjaxTags 概述

AjaxTags 是一套简单的 JSP 标签，这套标签以一些现有的开源项目为基础，AjaxTags 最核心的依赖是 Prototype.js 库及其相关项目 Scriptaculous.js。当然，JSTL 和 jakarta-commons 的一些包也是 AjaxTags 所必需的。除此之外，早期的 AjaxTags 还依赖于 Struts 框架，如果需要用 AjaxTags 的 displayTag 标签，则还需要 display Tag 框架。

AjaxTags 致力于解决 J2EE 应用开发者的软肋：厌倦了烦琐的 JavaScript 代码，以直接而烦琐的操作来操作 DOM 结构，从而提供 HTML 页面的动态显示。AjaxTags 完全不需要开发者写一行 JavaScript 代码，它以一种“非

常 Java 的方式”来开发 Ajax 应用，因此它专属于 J2EE 应用，Ajax 技术本身不局限于任何语言，但 AjaxTags 只能用于 Java 平台。

只要开发者能使用简单的 JSP 标签，就可以使用 AjaxTags。它提供了一组简单的标签，这些标签封装了常用的 AjaxTags 功能。

### 18.1.2 下载和安装 AjaxTags

AjaxTags 是属于 sourceforge 的一个开源项目，因此可以登录官方站点 <http://ajaxtags.sourceforge.net> 下载 AjaxTags 的最新版。目前，AjaxTags 的最新版本是 1.2 beta3，这是一个测试版，可能存在一定风险，但 AjaxTags 的产品版实在太老了，因此推荐使用 AjaxTags 1.2 beta3 版。

在浏览器的地址栏中输入 [http://sourceforge.net/project/showfiles.php?group\\_id=140499](http://sourceforge.net/project/showfiles.php?group_id=140499)，即可下载到 AjaxTags 的 1.2 beta3 版本，将下载到的压缩文件解压缩，可看到如下文件结构。

css: 该路径下包含了使用 AjaxTags 必需的 CSS 样式单，这些样式单文件管理 Ajax 页面最通用的显示效果。当然，用户可以开发自己的 CSS 样式单文件。

dist: 该路径下包含了 AjaxTags 编译生成的 JAR 文件。

docs: 该路径下存放 AjaxTags 的各种文档，但没有包含 API 文档（AjaxTags 的 API 文档需要另行下载）。

images: 该路径下包含使用 AjaxTags 必需的图片文件，这些图片文件用于生成 Ajax 页面中最通用的效果。当然，用户也可以使用自己喜欢的图片。

js: 该路径下包含了使用 AjaxTags 必需的 JavaScript 库文件，例如 prototype.js 等文件都可以在该路径下找到。

lib: 该路径下存放了编译和运行 AjaxTags 所依赖的第三方类库，用户可根据需要选择使用。

授权等其他相关文档。

要在一个 Web 应用中使用 AjaxTags，只需要将其 JAR 文件复制到 Web 应用的 WEB-INF/lib 下就可以了。因为 AjaxTags 是一个专注表现层的技术，因此，如果需要在一个 Web 应用中使用 AjaxTags，则需要更多的东西。要在一个 Web 应用中使用 AjaxTags，可以按如下步骤进行操作：

**step 1** 将 AjaxTags 项目解压缩路径下的 dist/ajaxtags-1.2-beta3.jar 复制到 Web 应用的 WEB-INF/lib 下。

**step 2** 因为 AjaxTags 还需要第三方类库的支持，所以至少需要将 JSTL 项目的 JAR 文件和 jakarta-commons 下必需的 JAR 文件复制到 WEB-INF/lib 路径下。

**step 3**

因为需要使用JSP标签，所以对于使用JSP 1.1的用户，应该将ajaxtags.tld文件（该文件位于ajaxtags-1.2-beta3.jar文件的META-INF文件夹下）复制到WEB-INF路径下，并在web.xml文件中定义标签库的引用，即在web.xml文件中增加如下一段：

```
<taglib>

<!-- 定义标签库的 URI -->

    <uri>/tags/ajax</uri>

<!-- 定义标签库 TLD 文件的物理位置-->

    <location>/WEB-INF/ajaxtags.tld</location>

</taglib>
```

上面的配置文件是属于JSP自定义标签库的内容，请参考JSP自定义标签库的相关资料。上面的配置意味着在JSP页面中应使用如下代码来导入AjaxTags标签库：

```
<%@ taglib uri="/tags/ajax" prefix="ajax" %>
```

实际上，对于使用JSP 2.0的用户，根本不需要在web.xml文件中定义标签库，而可以直接在JSP页面中使用标签库的绝对地址，即通过如下方式引用AjaxTags的标签库：

```
<%@ taglib uri="http://ajaxtags.org/tags/ajax" prefix="ajax" %>
```

**step 4**

将AjaxTags项目解压缩路径下的js整个路径复制到Web应用的根路径下。当然，也可以放在Web应用的任意路径下，只要在JSP页面中能引用到这些JavaScript文件即可。AjaxTags通常依赖于如下4个关键的JavaScript库：

```
Prototype-1.4.0.js
Scriptaculous.js
Overlibmws.js
AjaxTags.js
```

因此，在每个JSP页面中都应该导入如下4行代码：

```
<script type="text/javascript" src="js/prototype-1.4.0.js"></script>

<script type="text/javascript" src="js/scriptaculous.js"></script>

<script type="text/javascript" src="js/overlibmws.js"></script>

<script type="text/javascript" src="js/ajaxtags.js"></script>
```

**step 5**

将AjaxTags项目解压缩路径下的images文件夹复制到Web应用的根路径下。当然，也可以放在Web应用的任意路径下，只要在JSP页面中能引用到该路径下的图片文件即可。如果程序开发者决定使用自己的图片，则可以省略该步骤，但开发者应该自己提供相应的图片，例如自动完成应用中使用的转动图片。

**step 6**

将AjaxTags项目解压缩路径下的css整个路径复制到Web应用的根路径下。当然，也可以放在Web应用的任意路径下，理由与images相似。同样，开发者也可以选择放弃AjaxTags默认的CSS样式，而使用自己的CSS样式。通常，我们可以使用如下格式来导入CSS样式：

```
<link type="text/css" rel="stylesheet" href="css/ajaxtags-sample.css" />
```

经过上面的步骤，我们就可以在一个 JSP 页面中使用 AjaxTags 标签了，这个标签可以极大地简化 Ajax 应用的开发。因为我们需要使用标签库，因此这些 Ajax 页面不能是普通的 HTML 页面，而必须是 JSP 页面。



使用 AjaxTags 的页面不能是静态的 HTML 页面，而必须是 JSP 页面。

当然，即使有 AjaxTags 的帮助，服务器的响应还是必须由程序员提供。下面，我们介绍如何开发服务器响应类。

## 18.2 AjaxTags 入门

前面我们已经说过了：使用 Ajax 是相当简单的事情，完全不需要烦琐地创建 XMLHttpRequest 对象来发送 Ajax 请求。使用 AjaxTags 甚至不需要编写回调函数，不需要解析服务器响应，不需要程序员手动更新 HTML 页面显示，整个过程完全由 AjaxTags 完成，应用开发者只需要编写服务器响应即可。服务器响应就是 Ajax 请求的处理类。

### 18.2.1 编写处理类

这里说的处理类并不一定是一个完整的 Java 类，它可以是一个 JSP 页面，也可以是一个配置在 Web 应用中的 Servlet 或者 Struts 的 Action，甚至是一个非 Java 的服务器组件，只要它能响应用户的请求即可。当然，因为



AjaxTags 是一种高度封装的 Ajax 框架，因此处理类的返回值不能像之前那样随心所欲，而必须满足某种格式。

服务器处理类的返回值必须满足如下 XML 文件格式：

```
<!-- XML 文件的声明部分 -->

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

<!-- AjaxTags 服务器响应的根元素 -->

<ajax-response>

  <!-- AjaxTags 服务器响应的内容必须在 response 里 -->

  <response>

    <!-- 下面的 item 节点分别用于不同的选择 -->

    <item>

      <name>Record 1</name>

      <value>1</value>

    </item>

    <item>

      <name>Record 2</name>

      <value>2</value>

    </item>

    <item>

      <name>Record 3</name>

      <value>3</value>

    </item>

  </response>

</ajax-response>
```

当然，也可以使用普通文本响应，使用普通文本响应则应该生成如下格式的响应：

```
#普通文本响应的示范

#每项响应对应一行，每行的前面部分是 name，后面是 value

#中间以英文逗号隔开
```

Record 1,1

Record 2,2

Record 3,3

下面介绍的是一个简单应用，以自动完成作为示范建立一个对应的处理类，处理类以 JSP 来代替。下面是自动完成的处理 JSP 页面代码。这是一个简单的级联菜单应用，用户一旦选中第一个下拉列表框的某个选项值，下一个下拉列表框将随之变化。处理类由一个 JSP 页面充当，该页面负责生成一个 XML 响应，而 XML 响应则符合上面的格式。下面是该处理器页面的代码：

```
<%@ page contentType="text/html; charset=GBK" language="java" %>

<%@ page import="java.util.*"%>

<%

//获取请求参数

int country = Integer.parseInt(request.getParameter("country"));

//设置响应的页面头

response.setContentType("text/xml; charset=UTF-8");

//控制响应不会在客户端缓存

response.setHeader("Cache-Control", "no-cache");

//用于控制服务器的响应

List<String> cityList = new ArrayList<String>();

//根据请求参数 country 来控制服务器响应

switch(country)

{

//对于选择下拉框的“中国”选项

case 1:

    cityList.add("广州");

    cityList.add("深圳");

    cityList.add("上海");

    break;
```

```
//对于选择下拉框的“美国”选项

case 2:

    cityList.add("华盛顿");

    cityList.add("纽约");

    cityList.add("加州");

    break;

//对于选择下拉框的“日本”选项

case 3:

    cityList.add("东京");

    cityList.add("大阪");

    cityList.add("福冈");

    break;

}

%>

<ajax-response>

    <response>

        <%

            //遍历集合，依次将城市添加到服务器响应的 item 项里

            for(String city : cityList)

            {

                %>

                <item>

                    <name><%=city%></name>

                    <value><%=city%></value>

                </item>

            }%>

        </response>
```

```
</ajax-response>
```

该页面根据请求参数，依次将 3 个城市添加到 cityList 集合里，然后通过如下代码表示生成的页面是一个 XML 文件：

```
response.setContentType("text/xml; charset=UTF-8");
```

一旦生成了 XML 响应，就可以在客户端 JSP 页面使用标签来生成 Ajax 响应了。

## 18.2.2 使用标签

在客户端页面使用 AjaxTags 标签是非常简单而且“非常 Java”的，几乎感觉不到使用了 Ajax 功能，但该页面已经具有了 Ajax 能力。在 JSP 页面使用 AjaxTags 应按如下步骤进行：

**step 1** 在 JSP 页面中使用 taglib 导入 AjaxTags 标签库。

**step 2** 在 JSP 页面中导入 AjaxTags 必需的 JavaScript 库。

**step 3** 使用 AjaxTags 对应的标签。

使用 AjaxTags 的 select 标签的 JSP 页面代码如下：

```
<!-- 导入 AjaxTags 标签库 -->
<%@ taglib uri="http://ajaxtags.org/tags/ajax" prefix="ajax" %>
<!-- 设置页面的内容以及对应的编码集 -->
<%@ page contentType="text/html; charset=GBK"%>
<html>
<head>
  <title>第一个 AjaxTags 应用</title>
  <!-- 在 JSP 页面中引入必需的 JavaScript 库 -->
  <script type="text/javascript" src="js/prototype-1.4.0.js"></script>
  <script type="text/javascript" src="js/scriptaculous.js"></script>
  <script type="text/javascript" src="js/overlibmws.js"></script>
  <script type="text/javascript" src="js/ajaxtags.js"></script>
  <!-- 在 JSP 页面中引入必需的 CSS 样式单 -->
  <link type="text/css" rel="stylesheet" href="css/ajaxtags-sample.css" />
</head>
<body>
  国家:
  <!-- 激发 Ajax 的源 select 元素 -->
  <select id="country" name="country">
    <option value="">选择一个国家</option>
    <option value="1">中国</option>
    <option value="2">美国</option>
```

```
<option value="3">日本</option>
</select>
城市:
<!-- 显示 Ajax 响应的 select 元素 -->
<select id="city" name="city">
  <option value="">城市列表</option>
</select>
<!-- 使用 AjaxTags 标签 -->
<ajax:select
  baseUrl="res.jsp"
  source="country"
  target="city"
  parameters="country={country}" />
</body>
</html>
```

在上面的 JSP 页面中，除了引入几个 JavaScript 代码库外，完全不需要任何 JavaScript 代码，丝毫感受不到使用 Ajax 的痕迹，但因为使用了 AjaxTags 的 select 标签（该标签用于生成级联菜单的 Ajax 应用），该页面也具有了级联菜单的功能。图 18.1 显示了使用 AjaxTags 后该页面的级联菜单效果。



图 18.1 使用 AjaxTags 后级联菜单的效果

通过在 JSP 页面中使用 AjaxTags 标签，可以看到 AjaxTags 标签使 Ajax 应用非常简单。



因为有些浏览器在处理以 GET 方式发送的请求时存在一些问题，因此这里修改了 ajaxtags.js 文件中的请求发送方式。关于请求的发送方式，笔者更倾向于使用 POST 请求，而 AjaxTags 默认的请求发送方式是 GET。只需打开 ajaxtags.js 文件，将文件中的 method: 'get' 替换成 method: 'post'，即可改变 AjaxTags 的请求发送方式。

## 18.3 处理类的几种形式

前面已经介绍过了，AjaxTags 的处理类并不一定是一个真正的 Java 类，它可以有很多形式，甚至于可以是一个非 Java 文件，唯一的要求是该处理类能返回一个满足格式的 XML 文件。除此之外，AjaxTags 还提供了几个特殊的类，这些特殊的类可以简化处理类的实现。

下面依次介绍处理类的几种形式，介绍这几种形式时都以前面介绍的级联菜单应用为基础，应用使用 AjaxTags 标签的 JSP 页面没有太大改变，仅改变了 ajax:select 标签的 baseUrl 属性——这是因为采用不同处理类时，处理类在服务器端配置的 URL 不同。

### 18.3.1 使用普通 Servlet 生成响应

使用普通 Servlet 生成响应与前面介绍的 JSP 响应是类似的，因为 JSP 的实质就是 Servlet，一旦完成了 Servlet 的配置，便可以将 baseUrl 指定为该 Servlet 的 URL 地址，将请求向该 Servlet 发送，该 Servlet 负责生成 XML 响应，从而完成整个 Ajax 交互。负责响应的 Servlet 的源代码如下：

```
//普通 Servlet，继承 HttpServlet

public class SelectServlet extends javax.servlet.http.HttpServlet
{
    //普通 Servlet 的服务响应方法

    public void service(HttpServletRequest request, HttpServletResponse response)
        throws ServletException,IOException
    {
        //获得请求参数，即国家值

        int country = Integer.parseInt(request.getParameter("country"));

        //用于控制服务器的响应

        List<String> cityList = new ArrayList<String>();

        switch(country)
        {
            //对于选择下拉框的"中国"选项
```

```
case 1:

    cityList.add("广州");

    cityList.add("深圳");

    cityList.add("上海");

    break;

//对于选择下拉框的"美国"选项

case 2:

    cityList.add("华盛顿");

    cityList.add("纽约");

    cityList.add("加州");

    break;

//对于选择下拉框的"日本"选项

case 3:

    cityList.add("东京");

    cityList.add("大阪");

    cityList.add("福岡");

    break;

}

//该类用于辅助生成 XML 响应

AjaxXmlBuilder builder = new AjaxXmlBuilder();

for (String city : cityList )

{

    builder = builder.addItem(city, city);

}

System.out.println(builder);

//设置响应的页面头

response.setContentType("text/xml; charset=UTF-8");
```

```
        PrintWriter out = response.getWriter();

        out.println(builder.toString());

    }

}
```

该 Servlet 的代码与 JSP 代码非常类似，它们都是先获取请求参数 `country`，并根据请求参数值的不同，生成包含 3 个城市的 List 对象。值得注意的是，Servlet 的代码并不需要像在 JSP 页面里一样自己来控制输出 `<ajax-response>`，`<response>`，`<name>`和`<value>`等标签，这一系列的过程是通过 `AjaxTags` 提供的一个辅助类完成的，这个辅助类为 `AjaxXmlBuilder`。借助于 `AjaxXmlBuilder` 的帮助，程序开发者可以更简单地生成 `AjaxTags` 所需格式的响应。关于 `AjaxXmlBuilder` 的内容，将在下一部分介绍。

编译该 Servlet，在 `web.xml` 文件中增加如下配置，之后该 Servlet 就能处理客户端请求了。在 `web.xml` 文件中增加的配置片段如下：

```
<!-- 配置 Servlet -->

<servlet>

    <!-- 配置该 Servlet 的名字 -->

    <servlet-name>select</servlet-name>

    <!-- 配置该 Servlet 的实现类 -->

    <servlet-class>lee.SelectServlet</servlet-class>

    <!-- 配置该 Servlet 随应用启动而自启动 -->

    <load-on-startup>2</load-on-startup>

</servlet>

<!-- 配置 Servlet 映射的 URL -->

<servlet-mapping>

    <!-- 配置该 Servlet 的名字 -->

    <servlet-name>select</servlet-name>

    <!-- 配置该 Servlet 映射的 URL 地址 -->

    <url-pattern>/select</url-pattern>

</servlet-mapping>
```



一旦完成了 Servlet 的映射，该 Servlet 就能在 select 的 URL 处提供响应了。因此，只需要在使用 AjaxTags 标签的页面中通过如下代码来使用标签：

```
<ajax:select  
    baseUrl="select"  
    source="country"  
    target="city"  
    parameters="country={country}" />
```

与之前直接使用 JSP 作为响应对比，修改了 ajax:select 标签的 baseUrl 属性，baseUrl 属性是 Ajax 请求发送的服务器 URL，此处改为 select，即表示向 SelectServlet 发送请求。AjaxTags 标签各属性的详细内容将在后面介绍。

### 18.3.2 使用 AjaxXmlBuilder 辅助类

AjaxXmlBuilder 是一个工具类，它可以包含一些工具方法，这些工具方法根据字符串或者集合来生成满足 AjaxTags 要求的响应。借助于 AjaxXmlBuilder 的帮助，程序开发者无须手动输出 <ajax-response>，<response>，<name>和<value>等标签，只需调用 AjaxXmlBuilder 的响应方法，它就会自动生成这些标签，并将字符串或集合里的对象的值增加到该响应里。AjaxXmlBuilder 对象主要包含如下两个方法。

**public AjaxXmlBuilder addItem(java.lang.String name, java.lang.String value):** 每调用一次，为响应增加一个 item 节点。其中，第一个参数是 item 节点下 name 节点的值，第二个参数是 item 节点下 value 节点的值。  
**public addItems(java.util.Collection collection, String nameProperty, String valueProperty):** 该方法编辑集合 collection 的值，collection 里的元素是对象，该对象必须包含 nameProperty 和 valueProperty 两个属性；该方法增加 collection 的长度个 item 节点，每个 item 节点的值就是集合元素的 nameProperty 属性的值，而 value 节点的值就是 valueProperty 属性的值。

通过这两个方法，可以非常便捷地生成 AjaxTags 所需格式的响应。看下面的简单代码：

```
public class Test  
{  
    public static void main(String[] args)  
    {  
        //创建一个默认的 AjaxXmlBuilder 实例  
        AjaxXmlBuilder builder = new AjaxXmlBuilder();
```

```
//采用循环依次为响应添加 5 个 item 节点

    for (int i = 0; i < 5; i++ )
    {
        builder.addItem("name 值" + i, "value 值" + i);
    }

//打印出 builder 本身所生成的 XML 响应

    System.out.println(builder.toString());

}
```

该文件将生成一个满足 AjaxTags 响应格式的 XML 文件，该文件的 ajax-response 和 response 节点都会默认包含，该文件包含了 5 个 item 节点。下面是该程序的打印结果：

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

<ajax-response>

<response>

<item>

    <name>name 值 0</name>

    <value>value 值 0</value>

</item>

<item>

    <name>name 值 1</name>

    <value>value 值 1</value>

</item>

<item>

    <name>name 值 2</name>

    <value>value 值 2</value>

</item>

<item>
```

```
<name>name 值 3</name>

<value>value 值 3</value>

</item>

<item>

    <name>name 值 4</name>

    <value>value 值 4</value>

</item>

</response>

</ajax-response>
```

上面的 XML 文件是使用 `AjaxXmlBuilder` 的 `addItem` 生成的。除此之外，还可以使用 `addItems` 方法，该方法会自动遍历集合，并将集合元素的指定属性分别添加为 `item` 节点下 `name` 节点的值和 `value` 节点的值。看下面的示例程序：

```
public class Test2
{
    public static void main(String[] args)throws Exception
    {
        //构造一个集合对象
        List<Person> pl = new ArrayList<Person>();
        //向集合添加 3 个元素，每个元素都是 Person 实例
        pl.add(new Person("Jack", "男"));
        pl.add(new Person("Rose", "女"));
        pl.add(new Person("小强", "男"));
        AjaxXmlBuilder builder = new AjaxXmlBuilder();
        //为生成的 XML 响应增加 3 个 item 节点
        builder.addItems(pl, "name", "gender");
        //输出生成的 XML 响应
        System.out.println(builder.toString());
    }
}
```

```
}
```

```
}
```

该代码使用 `addItems` 方法为 XML 响应添加了 3 个 `item` 节点。每个节点都包含 `name` 和 `value` 子节点，`name` 节点的值集合元素的指定 `name` 属性，而 `value` 节点则为集合元素的 `gender` 属性——前提是集合元素有 `name` 属性和 `gender` 属性。集合元素是 `Person` 实例，`Person` 实例应该包括这两个属性，下面是 `Person` 类的代码：

```
public class Person
{
    //Person 类包含 name 和 gender 属性

    private String name;

    private String gender;

    //无参数的构造器

    public Person()
    {
    }

    //有参数的构造器

    public Person(String name, String gender)
    {
        this.name = name;

        this.gender = gender;
    }

    //name 和 gender 属性的 setter 方法

    public void setName(String name)
    {
        this.name = name;
    }

    public void setGender(String gender)
    {
```

```
        this.gender = gender;
    }

    //name 和 gender 属性的 getter 方法
    public String getName()
    {
        return (this.name);
    }

    public String getGender()
    {
        return (this.gender);
    }
}
```

运行上面的代码，将生成如下输出：

```
<?xml version="1.0" encoding="UTF-8" ?>
<ajax-response>
<response>
<item>
    <name><![CDATA[Jack]]></name>
    <value><![CDATA[男]]></value>
</item>
<item>
    <name><![CDATA[Rose]]></name>
    <value><![CDATA[女]]></value>
</item>
<item>
    <name><![CDATA[小强]]></name>
    <value><![CDATA[男]]></value>
```

```
</item>

</response>

</ajax-response>
```

通过使用 `AjaxXmlBuilder` 的 `builder.addItem(pl, "name", "gender")` 方法, 该 `builder` 将自动遍历 `pl` 集合中的元素, 每个元素对应一个 `item` 节点。集合元素的 `name` 属性为每个 `item` 节点的 `name` 节点值, `gender` 属性为每个 `item` 节点的 `value` 节点值。

### 18.3.3 使用 `BaseAjaxAction` 生成响应

这是一种更早期的用法, 这种用法提供了一个 `BaseAjaxAction` 辅助类来负责生成响应。这个类继承了 `Struts` 的 `Action`。因此, 如果需要使用这种响应, 则应该将 `Struts` 所需要的 `JAR` 文件复制到 `Web` 应用中。

当然, 使用这种响应有一个最大的好处: 如果 `Web` 应用的表现层已经使用了 `Struts` 作为 `MVC` 框架, 那么该应用与 `AjaxTags` 的整合将变得更加容易。如果 `Web` 应用本身不使用 `Struts` 作为 `MVC` 框架, 那么这种响应不是一种理想选择。

`BaseAjaxAction` 是一个抽象类, 但该类已经实现了 `Struts Action` 的 `execute(ActionMapping mapping, ActionForm form, HttpServletRequest request, HttpServletResponse response)` 方法, `AjaxTags` 通过实现该方法来提供响应, 具体响应则通过回调 `getXmlContent(ActionMapping mapping, ActionForm form, HttpServletRequest request, HttpServletResponse response)` 方法来实现, 该方法是一个抽象方法, 必须由程序员实现。

程序员实现 `getXmlContent(ActionMapping mapping, ActionForm form, HttpServletRequest request, HttpServletResponse response)` 方法, 该方法返回一个满足 `AjaxTags` 要求的 `XML` 字符串, `BaseAjaxAction` 根据该方法的返回值来生成响应。下面是一个 `BaseAjaxAction` 实现类的示范, 该示范也能提供上面级联菜单的响应。

```
//使用 BaseAjaxAction 响应, 应该集成 BaseAjaxAction 类

public class SelectAction extends BaseAjaxAction
{
    //实现抽象方法

    public String getXmlContent(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response) throws Exception
    {
```

```
//获取请求参数

int country = Integer.parseInt(request.getParameter("country"));

List<String> cityList = new ArrayList<String>();

//对于不同的请求参数，向集合中添加不同的城市字符串

switch(country)

{

    //对于选择下拉框的"中国"选项

    case 1:

        cityList.add("广州");

        cityList.add("深圳");

        cityList.add("上海");

        break;

    //对于选择下拉框的"美国"选项

    case 2:

        cityList.add("华盛顿");

        cityList.add("纽约");

        cityList.add("加州");

        break;

    //对于选择下拉框的"日本"选项

    case 3:

        cityList.add("东京");

        cityList.add("大阪");

        cityList.add("福冈");

        break;

}

//该类用于辅助生成 XML 响应

AjaxXmlBuilder builder = new AjaxXmlBuilder();
```

```
        for (String city : cityList )
        {
            builder = builder.addItem(city, city);
        }

        //返回生成的 XML 字符串

        return builder.toString();
    }
}
```

一旦实现了 `getXmlContent` 方法，`BaseAjaxAction` 便可以根据该方法的返回值来生成 Ajax 响应。

值得注意的是，这个响应是一个 `Action`，并不是普通的 `Servlet`。因此，它不可以直接作为 `Servlet` 来提供响应，而必须借助于 `Struts` 框架。为了在 Web 应用中使用 `Struts` 框架，应该修改 `web.xml` 文件，在文件中载入 `Struts` 框架。修改后的 `web.xml` 文件的代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app 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" version="2.4">
<!-- 配置 Struts 的核心 Servlet -->
<servlet>
    <!-- 指定 Servlet 的名字 -->
    <servlet-name>action</servlet-name>
    <!-- 指定 Servlet 的实现类 -->
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<!-- 配置 Servlet 的映射 -->
<servlet-mapping>
    <!-- 指定 Servlet 的名字 -->
    <servlet-name>action</servlet-name>
    <!-- 指定 Servlet 的映射的 URL -->
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
</web-app>
```

经过上面的配置，所有匹配 `*.do` 模式的请求都将自动转向 `Struts` 框架，由 `ActionServlet` 负责处理。`ActionServlet` 根据请求将请求转发到对应的 `Action` 处理。



一旦需要使用 Struts 框架，则少不了 Struts 的配置文件，本示例应用中仅使用了一个 Action，负责生成客户端响应。因此，struts-config.xml 文件中仅需要配置一个 Action，具体的配置文件如下：

```
<?xml version="1.0" encoding="iso-8859-1"?>

<!-- Struts 配置文件的 DTD 信息 -->

<!DOCTYPE struts-config PUBLIC

    "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"

    "http://struts.apache.org/dtds/struts-config_1_2.dtd">

<struts-config>

<action-mappings>

    <!-- Struts 配置文件的 Action 配置 -->

        <action path="/select" type="lee.SelectAction"/>

    </action-mappings>

</struts-config>
```



与传统的 Struts 的 Action 配置不同，因为 AjaxTags 的标签 action 无须转发，因此无须配置 forward 页面。

该配置文件配置了一个 /select 的 Action，该 Action 能处理从 ActionServlet 转发来的请求。为了保证用户请求进入 Struts 框架的处理，用户请求必须使用 .do 结尾。页面中使用该 Action，必须修改请求对应的 URL。下面的代码片段是在页面中使用 AjaxTags 的 BaseAjaxAction 响应：

```
<!-- 使用 AjaxTags 的 select 标签，baseUrl 指定为 Struts 的 Action -->

<ajax:select

    baseUrl="select.do"

    source="country"

    target="city"

    parameters="country={country}" />
```

通过这种方式，AjaxTags 可以与 Struts 以无缝的方式整合起来。

### 18.3.4 使用 BaseAjaxServlet 生成响应

这种方式是对使用 Servlet 作为响应的一种改进，也是 AjaxTags 脱离 Struts、成为独立框架的一个重大突破。通过使用 BaseAjaxServlet 作为响应，服务器的响应无须使用编写复杂的 Servlet 输出，也无须依赖于 Struts 框架。

BaseAjaxServlet 是 HttpServlet 的一个子类，该类可以直接配置在 web.xml 文件中来响应客户端请求，无须任何框架的支持。BaseAjaxServlet 也是一个抽象类，但它已经重写了 HttpServlet 的 doPost(HttpServletRequest request, HttpServletResponse response)和 doGet(HttpServletRequest request, HttpServletResponse response)方法，程序员无须实现这两个方法，这两个方法用于对客户端提供响应。与 BaseAjaxAction 类似的是，BaseAjaxServlet 的子类也必须实现一个回调方法，即 getXmlContent(HttpServletRequest request, HttpServletResponse response)，该方法返回一个 XML 字符串，该字符串被作为 Ajax 请求的响应。

```
//用于响应 Ajax 请求的 Servlet，继承 BaseAjaxServlet
public class SelectServlet extends BaseAjaxServlet
{
    //重写 getXmlContent 方法，该方法的返回值将作为 Ajax 请求的响应
    public String getXmlContent(HttpServletRequest request, HttpServletResponse
        response) throws Exception
    {
        //获取请求参数
        int country = Integer.parseInt(request.getParameter("country"));

        //初始化需要的集合
        List<String> cityList = new ArrayList<String>();

        //针对不同的请求参数，将不同的城市添加到集合中
        switch(country)
        {
            //对于选择下拉框的"中国"选项
            case 1:
                cityList.add("广州");
```

```
        cityList.add("深圳");

        cityList.add("上海");

        break;

//对于选择下拉框的"美国"选项
case 2:

    cityList.add("华盛顿");

    cityList.add("纽约");

    cityList.add("加洲");

    break;

//对于选择下拉框的"日本"选项
case 3:

    cityList.add("东京");

    cityList.add("大阪");

    cityList.add("福冈");

    break;

}

//该类用于辅助生成 XML 响应

AjaxXmlBuilder builder = new AjaxXmlBuilder();

//遍历集合，将集合中的元素添加为 item 节点的子节点 name 和 value 的值

for (String city : cityList )

{

    builder = builder.addItem(city, city);

}

//返回一个 XML 字符串

return builder.toString();

}

}
```

这个 Servlet 没有重写 `doGet` 和 `doPost` 方法，这也正是 `BaseAjaxServlet` 的用处，避免了编写烦琐的 `doGet` 和 `doPost` 方法，而且无须手动获取输出流，只要返回一个 XML 字符串即可，而 `BaseAjaxServlet` 则将 `getXmlContent` 方法的返回值作为响应。

在 `web.xml` 文件中配置该 Servlet 与配置普通 Servlet 并没有区别，下面是在 `web.xml` 文件中配置该 Servlet 的代码片段：

```
<!-- 配置 Servlet -->

<servlet>

<!-- 配置该 Servlet 的名字 -->

<servlet-name>select</servlet-name>

    <!-- 配置该 Servlet 的实现类 -->

<servlet-class>lee.SelectServlet</servlet-class>

</servlet>

<!-- 配置 Servlet 映射的 URL -->

<servlet-mapping>

<!-- 配置该 Servlet 的名字 -->

<servlet-name>select</servlet-name>

<!-- 配置该 Servlet 映射的 URL 地址 -->

<url-pattern>/select</url-pattern>

</servlet-mapping>
```

上面的配置文件表示，该 Servlet 负责响应 URL 为 `/select` 的请求。因此，在 JSP 页面中使用 Ajax 标签的代码片段与使用普通 Servlet 作为响应的标签并没有区别，代码片段如下：

```
<!-- 使用 Ajax 的 select 标签 -->

<ajax:select

    baseUrl="select"

    source="country"

    target="city"

    parameters="country={country}" />
```

### 18.3.5 使用非 Java 响应

前面已经介绍过了，AjaxTags 标签并不一定要求服务器采用 Servlet 或者 JSP 作为响应，甚至允许使用非 Java 响应，例如直接使用静态 XML 文件作为响应。虽然这种场景的应用不是特别广泛，但也不是完全没有用处，下面介绍使用静态 XML 文件作为响应的情形。静态 XML 文件的源代码如下：

```
<?xml version="1.0" encoding="GBK"?>

<ajax-response>

  <response>

    <item>

      <name>广州</name>

      <value>广州</value>

    </item>

    <item>

      <name>深圳</name>

      <value>深圳</value>

    </item>

    <item>

      <name>上海</name>

      <value>上海</value>

    </item>

  </response>

</ajax-response>
```

读者可能已经看出来了：这个静态 XML 文件是当 country 请求参数为 1 时 Servlet 的响应。如果使用这个静态 XML 文件作为响应，则不管在客户端选择哪个国家，都将输出中国的 3 个城市。这并不符合实际情况，这里仅用于测试程序的功能。

假设该文件的文件名为 res.xml，并将该文件放置在与 select.jsp 页面相同的路径下，然后在 select.jsp 页面中通过如下标签来完成 Ajax 交互：

```
<!-- 使用 AjaxTags 的标签 -->
```

```
<ajax:select
    baseUrl="res.xml"
    source="country"
    target="city"
    parameters="country={country}" />
```

Ajax select 标签的 baseUrl 改为 res.xml，表示直接向静态的 XML 文件请求。不管请求参数是什么，请求总是得到相同的响应。图 18.2 显示了这种情况。



图 18.2 使用静态 XML 文件作为响应

## 18.4 AjaxTags 常用标签的使用

AjaxTags 提供的标签并不是特别多，但这些标签都是特别常用的，例如自动完成、级联菜单等。使用 AjaxTags 的标签才能真正让我们从烦琐的 JavaScript 处理中抽身而出，以“非常 Java”的方式优雅地完成 Ajax 应用。

### 18.4.1 使用自动完成标签

自动完成标签的功能类似于 Internet Explorer 中文本框具有的功能，系统可以根据用户的输入提示自动完成选择。

假设输入城市，如果用户输入“广”，系统会读取数据库，对比数据库的记录，找出所有以“广”开头的城市，例如“广岛”和“广州”，从而提供给用户选择。这种自动完成的 Ajax 应用在第 11 章已经介绍过了，但那个自动完成应用大约有 200 多行的 JavaScript 代码，下面以 AjaxTags 来完成这个 Ajax 应用。自动完成功能使用 ajax:autocomplete，这个标签有如下几个属性。

**var:** 这是一个可选属性，该属性定义了 autocomplete 标签创建的 JavaScript 对象名。通常无须指定该属性。  
**attachTo:** 这是一个可选属性，该属性定义了 var 对应的自动完成对象将应用到的对象。

**baseUrl:** 这是一个必需属性，该属性定义了 Ajax 请求发送的 URL。该属性指定的 URL 将返回一个典型的 AjaxTags 所需要的 XML 响应，响应的每个 item 节点的 name 值就是自动完成的提供给用户选择的一项。该属性支持表达式语言。

**source:** 这是一个必需属性，该属性指定的 HTML 元素的值一旦发生改变，就将发送 Ajax 请求。通常，该属性的文本将作为自动完成的前缀部分，即 source 元素指定的 HTML 元素里的文本将被作为请求参数，伴随着 Ajax 请求一同发送，一旦该请求参数发送到服务器的 baseUrl，baseUrl 处的服务器响应将返回一个满足条件的 XML 响应。当然，也可以指定其他请求参数。

**target:** 这是一个必需属性，该属性指定一个文本框，该文本框将显示自动完成的选择项对应的 value。如果用户无须使用额外的文本框来显示 value，则可将该参数设置为与 source 相同。

**parameters:** 这是一个必需属性，该属性指定了 Ajax 请求的请求参数。

**className:** 这是一个必需属性，该属性指定了自动完成所提供的下拉框的 CSS 样式单的名字。通常，系统提供该 CSS 样式单，但用户也可以自定义自己的 CSS 样式单。

**indicator:** 这是一个可选属性，该属性指定一个 HTML 元素，该元素在 Ajax 请求开始时出现，随着 Ajax 交互完成而隐藏。该元素可以通知用户 Ajax 交互的进度。

**minimumCharacters:** 这是一个可选属性，该属性指定自动完成最少所需的字符数。假设 source 指定一个文本框，如果 minimumCharacters 的属性值为 2，则至少要求 source 指定的文本框提供两个字符的输入，系统才会提供自动完成功能。

**appendSeparator:** 这是一个可选属性，一旦设置了该属性，target 属性指定的文本框的值就不会被覆盖，而是在后面添加上自动完成的 value 节点的值，添加时将以 appendSeparator 属性指定的字符串作为分隔符。

**preFunction:** 这是一个可选属性，该属性指定了 Ajax 交互之前自动执行的函数。

**postFunction:** 这是一个可选属性，该属性指定了 Ajax 交互完成后自动执行的函数。

**errorFunction:** 这是一个可选属性，该属性指定服务器响应出错时执行的函数。

**parser:** 这是一个可选属性，该属性指定一个服务器响应的解析器，通常无须指定该解析器，除非用户需要自己完成特别的工作。默认的解析器是 ResponseHtmlParser。

为了更好地演示 Ajax 应用，首先引入了一个 JavaBean，这个 JavaBean 表示一本书，书里包含书名和出版社两个属性。下面是 Book 类的代码：

```
//普通的值对象（VO）

public class Book implements Serializable
{
    //Book 类包含的两个属性

    private String name;

    private String publisher;

    //Book 类的构造器

    public Book()
    {
    }
}
```

```
//Book 类的构造器，构造时传入两个参数

public Book(String name, String publisher) {

    this.name = name;

    this.publisher = publisher;

}

//name 属性的 setter 方法

public void setName(String name) {

    this.name = name;

}

//publisher 属性的 setter 方法

public void setPublisher(String publisher) {

    this.publisher = publisher;

}

//name 属性的 getter 方法

public String getName() {

    return (this.name);

}

//publisher 属性的 getter 方法

public String getPublisher() {

    return (this.publisher);

}

/**
 * 重写的 toString 方法
 * @see java.lang.Object#toString()
 */

public String toString()

{
```



```
return new ToStringBuilder(this).append("书名: ", name).append("出  
    版社: ", publisher).toString();  
}  
}
```

该 Book 类就是一个普通的值对象（VO），它包装了在页面中显示的 Java 实例。

为了简单起见，我们不再使用复杂的持久层组件，即不再使用数据库来保存数据，不使用 DAO 组件来完成数据库访问，而是将所有的数据以静态属性的方式保存在业务逻辑组件中，业务逻辑组件不再需要依赖持久层组件，而是直接访问静态属性，从而提供业务逻辑的实现。考虑到后面的 AjaxTags 标签，此处的 BookService 组件包含了多个业务逻辑方法。

下面是 BookService 类的源代码：

```
public class BookService  
{  
    //以此静态属性代替数据库保存的数据  
    static final List<Book> books = new ArrayList<Book>();  
    //添加数据  
    static  
    {  
        //添加电子工业出版社出版的书籍  
        books.add(new Book("Spring2.0 宝典", "电子工业出版社"));  
        books.add(new Book("Java 入门与精通", "电子工业出版社"));  
        books.add(new Book("Spring 实战", "电子工业出版社"));  
        books.add(new Book("Hibernate 入门与精通", "电子工业出版社"));  
        books.add(new Book("Java 网络编程", "电子工业出版社"));  
        //添加清华大学出版社出版的书籍  
        books.add(new Book("软件工程导论", "清华大学出版社"));  
        books.add(new Book("Java 教程", "清华大学出版社"));  
        books.add(new Book("Hibernate 持久化", "清华大学出版社"));  
        books.add(new Book("Java 动画设计", "清华大学出版社"));  
        //添加机械工业出版社出版的书籍  
        books.add(new Book("软件工程的艺术", "机械工业出版社"));  
        books.add(new Book("Java 高级程序设计", "机械工业出版社"));  
        books.add(new Book("Spring 项目指南", "机械工业出版社"));  
        books.add(new Book("Java 项目指南", "机械工业出版社"));  
    }  
    //构造器  
    public BookService()  
    {  
    }  
}
```

```
/**
 * 根据出版社查询所出版的书
 * @param publisher 出版社
 * @return 该出版社所出的全部书籍
 */
public List getBooksByPublisher(String publisher)
{
    //作为结果返回的集合对象
    List<Book> result = new ArrayList<Book>();
    //遍历所有的书，从中选出 publisher 出版社的书
    for ( Book book : books)
    {
        if (book.getPublisher().equalsIgnoreCase(publisher))
        {
            result.add(book);
        }
    }
    return result;
}

/**
 * 根据书名前缀返回以该前缀开始的全部书籍
 * @param prefix 书名前缀
 * @return 所有以 prefix 开头的书籍
 */
public List<Book> getBooksByPrefix(String prefix)
{
    //作为结果返回的集合对象
    List<Book> result = new ArrayList<Book>();
    //遍历所有的书，从中选出所有以 prefix 开头的书籍
    for (Book book : books)
    {
        if (book.getName().toLowerCase().startsWith(prefix.toLowerCase()))
        {
            result.add(book);
        }
    }
    return result;
}

/**
 * 返回全部书籍
 * @return 所有书籍
 */
public List<Book> getAllBooks()
{

```

```
        return books;
    }
}
```

这个 BookService 业务逻辑组件将作为本节所有示例程序的业务逻辑组件。本示例程序在添加书籍实例时，并未考虑书籍的真实性，只是随便添加，作为应用示例使用，请读者不要误会。本业务逻辑组件将所有的持久层数据作为组件属性保存，并未真正从数据库读取。下面是自动完成的 Servlet，该 Servlet 基于 BaseAjaxServlet 完成，其代码如下：

```
//自动完成的 Servlet，继承 BaseAjaxServlet

public class AutocompleteServlet extends BaseAjaxServlet
{
    //重写 getXmlContent 方法，该方法的返回值作为 Ajax 请求的响应

    public String getXmlContent(HttpServletRequest request, HttpServletResponse
        response) throws Exception
    {
        //获得请求参数

        String prefix = request.getParameter("prefix");

        System.out.println("xxx" + prefix);

        //创建业务逻辑组件的实例

        BookService service = new BookService();

        //返回以特定前缀开始的所有书籍

        List list = service.getBooksByPrefix(prefix);

        //借助于 AjaxXmlBuilder 返回 XML 字符串

        AjaxXmlBuilder builder = new AjaxXmlBuilder();

        try
        {
            builder = builder.addItem(list, "name", "publisher");

            return builder.toString();
        }
    }
}
```

```
        catch (Throwable e)
        {
            e.printStackTrace();
        }

        return null;
    }
}
```

该 Servlet 根据发送的请求参数调用业务逻辑组件方法，从所有的书籍中选择出所有书名以 **prefix** 开头的书籍。将该 Servlet 配置在 web.xml 文件中，该 Servlet 即能对 Ajax 请求提供响应。在 web.xml 文件中增加如下片段来完成 Servlet 的配置：

```
<!-- 配置 Servlet -->

<servlet>

<!-- Servlet 的名字 -->

<servlet-name>autocomplete</servlet-name>

<!-- Servlet 的实现类 -->

<servlet-class>lee.AutoCompleteServlet</servlet-class>

</servlet>

<!-- 配置 Servlet 的映射 -->

<servlet-mapping>

<!-- Servlet 的名字 -->

<servlet-name>autocomplete</servlet-name>

<!-- Servlet 映射的 URL -->

<url-pattern>/autocomplete</url-pattern>

</servlet-mapping>
```

一旦完成了该 Servlet 的映射，即可在页面中使用 **autocomplete** 标签，该标签提供了自动完成功能。下面是在页面中使用自动完成标签的代码片段：

```
<ajax:autocomplete
```

```
//根据书名文本框的值改变来发送 Ajax 请求
```

```
source="name"
```

```
//将 value 节点的值输入 publisher 文本框
```

```
target="publisher"
```

```
//Ajax 请求的发送地址
```

```
baseUrl="autocomplete"
```

```
//自动完成的提示框的 CSS 样式单
```

```
className="autocomplete"
```

```
//请求参数，其中{name}是表达式，输出 name 表单域的值
```

```
parameters="prefix={name}"
```

```
//当 Ajax 交互时显示 indicator
```

```
indicator="indicator"
```

```
//至少需要两个字符才发送 Ajax 请求
```

```
minimumCharacters="2"
```

```
/>
```

图 18.3 显示了自动完成的示范效果。

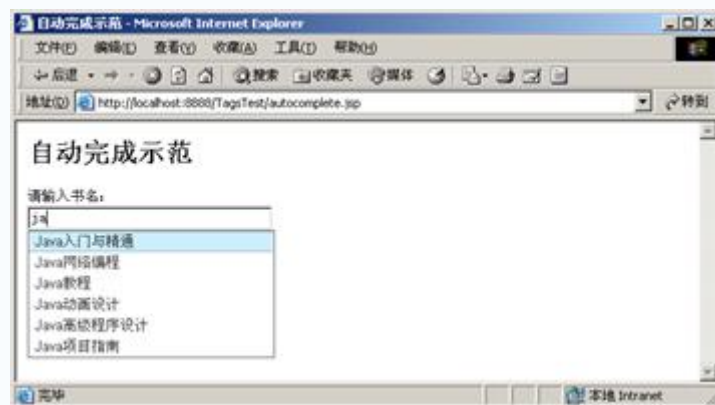


图 18.3 自动完成示范

一旦用户选择了相应的书籍，该书籍的出版社将自动填写在 publisher 文本框内。

### 18.4.2 使用 area 标签

该标签允许在页面中开辟出一个单独区域，而该区域的超链接请求等不会刷新整个页面，而是仅刷新部分内容。该标签有如下几个属性。

**id:** 这是一个必需属性，该属性指定 area 的 ID，用于唯一标识该 area，该属性值将成为 area 对应的 DIV 元素的 ID 属性值。

**ajaxFlag:** 这是一个可选属性，该属性指定该页面的其他部分是否忽略 Ajax 请求。

**style:** 这是一个可选属性，用于指定内联的 CSS 样式。

**styleClass:** 这是一个可选属性，用于指定 CSS 样式单名。

**ajaxAnchors:** 这是一个可选属性。

下面是一个使用 area 标签的示例代码片段：

```
<!-- 定义一个页面范围内的 Java 实例 -->

<jsp:useBean id="now" class="java.util.Date"/>

<!-- 使用表达式输出页面内的 Java 实例 -->

日期: ${now}<p>

<!-- 使用 ajax:area 标签 -->

<ajax:area id="myarea" style="background:#eeeeee; width:300px; height:80px;"
  ajaxAnchors="true" ajaxFlag="myarea">
  简单的页面 area
<br/>
<a href="pagearea.jsp">单击此处</a>
<br/>
<!-- 使用表达式输出页面内的 Java 实例 -->

日期: ${now}

</ajax:area>
```

页面中分别有两次输出当前时间的代码，有两次输出 now 变量的代码。因为后一个日期的输出放在 ajax:area 标签内，单击“单击此处”超链接时，页面刷新，但不会刷新整个页面，仅刷新 ajax:area 标签内的部分内容。如果单击了“单击此处”超链接，那么可看到第二个日期发生了改变，但第一个日期不会被刷新。图 18.4 显示了 ajax:area 的局部刷新效果。



图 18.4 ajax:area 的局部刷新

### 18.4.3 使用 anchors 标签

这是一个超链接标签，但该超链接标签与普通的超链接不同，该超链接标签仅刷新页面的局部——刷新 ajax:area 标签指定的页面部分。因此，该标签必须与 ajax:area 一起使用。该标签有如下两个属性。

**target:** 这是一个必需属性，该属性指定刷新的 ajax:area 元素的 ID 属性。

**ajaxFlag:** 这是一个可选属性，该属性指定页面的其他部分是否忽略 Ajax 请求。

使用 ajax:anchors 标签，可以通过页面中的超链接而不是 ajax:area 中的超链接来控制局部刷新。

```
<jsp:useBean id="now" class="java.util.Date"/>
```

```
日期: ${now}<p>
```

```
<ajax:anchors target="myarea" ajaxFlag="myarea"><a href="anchors.jsp">
```

```
单击此处</a></ajax:anchors>
```

```
<ajax:area id="myarea" style="background:#eeeeee; width:300px; height:80px;"
```

```
ajaxAnchors="true" ajaxFlag="myarea">
```

```
简单的页面 area
```

```
<br/>
```

```
<a href="anchors.jsp">单击此处</a>
```

```
<br/>
```

```
日期: ${now}
```

```
</ajax:area>
```

在上面的页面中有两个超链接，一个是页面范围的超链接，一个是在 ajax:area 范围内的超链接。两个超链接都可以控制页面的局部刷新。

#### 18.4.4 使用 callout 标签

该标签是一个服务器提示功能，这个功能以前通常在客户端完成，当用户的鼠标移动到某个产品上面时，该产品上将出现一个提示框，但这个提示框的信息往往是写在客户端的。通过使用 callout 标签，可以让服务器响应作为提示框的信息。从 AjaxTags 1.2 以后，该功能的实现依赖于 OverLIBMWS JavaScript 库，因此应将其 JavaScript 库复制到对应的 Web 应用中。该标签支持如下几个属性。

**var:** 这是一个可选属性，该属性定义了 autocomplete 标签创建的 JavaScript 对象名。通常无须指定该属性。

**attachTo:** 这是一个可选属性，该属性定义了 var 对应的自动完成对象将应用到的对象。

**baseUrl:** 这是一个必需属性，该属性定义了 Ajax 请求发送的 URL。该属性指定的 URL 将返回一个典型的 AjaxTags 所需要的 XML 响应，响应的每个 item 节点的 name 值就是自动完成功能提供给用户选择的一项。该属性支持表达式语言。

**source:** 该属性指定哪个 HTML 元素将触发服务器提示框，即指定哪个 HTML 元素触发 Ajax 请求。必须指定 source 和 sourceClass 两个属性之一。

**sourceClass:** 该属性指定一类 HTML 元素将触发服务器提示框，即指定哪些 HTML 元素是该 CSS 样式单，这些 HTML 元素都可以触发 Ajax 请求。必须指定 source 和 sourceClass 两个属性之一。

**parameters:** 伴随 Ajax 请求发送的请求参数。该属性的值支持一个特殊的变量 ajaxParameter，该变量代表发送请求的内容。

**title:** 这是一个可选属性，该属性指定信息提示框的标题。

**overlib:** 这是一个可选属性，该属性指定 OverLib 库的各种选项。

**preFunction:** 这是一个可选属性，该属性指定了 Ajax 交互之前自动执行的函数。

**postFunction:** 这是一个可选属性，该属性指定了 Ajax 交互完成后自动执行的函数。

**errorFunction:** 这是一个可选属性，该属性指定服务器响应出错时执行的函数。

callout 标签所需要的 XML 响应只需要一个 item 节点，该节点的 name 节点值将作为提示的标题显示，而 value 节点值将作为提示的内容显示。下面是示例应用的 Servlet:

```
//作为 AjaxTags 响应的 Servlet

public class CalloutServlet extends BaseAjaxServlet
{
    //重写 getXmlContent 方法

    public String getXmlContent(HttpServletRequest request, HttpServletResponse
        response) throws Exception
    {
        //Ajax 请求总以 utf-8 编码集发送
        request.setCharacterEncoding("UTF-8");

        //获取请求参数
```



```
String param = request.getParameter("book");

System.out.println(param);

//第一个参数是 name 节点的值，作为提示的标题

//第二个参数是 value 节点的值，作为提示的内容

AjaxXmlBuilder builder = new AjaxXmlBuilder().addItemAsCDATA(

    "提示标题",

    "<p>关于书籍:<b>" + param + "</b>的信息如下: <br>" +

    "服务器的提示信息 </p>");

return builder.toString();

}

}
```

将该 Servlet 配置在应用中，为了配置该 Servlet，在 web.xml 文件中增加如下片段：

```
<!-- 配置 Servlet -->

<servlet>

<!-- Servlet 的名字 -->

<servlet-name>callout</servlet-name>

<!-- Servlet 的实现类 -->

<servlet-class>lee. CalloutServlet</servlet-class>

</servlet>

<!-- 配置 Servlet 的映射 -->

<servlet-mapping>

<!-- Servlet 的名字 -->

<servlet-name> callout </servlet-name>

<!-- Servlet 映射的 URL -->

<url-pattern>/ callout </url-pattern>

</servlet-mapping>
```

该 Servlet 即可响应用户的 Ajax 请求，下面是页面中使用 callout 标签的代码片段：

```
<div style="font-size: 90%; width: 650px; border: 1px dashed #999; padding: 10px">

  <p>

下面是目前 J2EE 领域内容最丰富的三本书: <p>

<!-- 下面 3 个 HTML 元素的 class 属性为 book -->

<li><a href="javascript:void(0)" class="book">Spring2.0 宝典</a>

<li><a href="javascript:void(0)" class="book">轻量级 J2EE 企业开发实战</a>

<li><a href="javascript:void(0)" class="book">Ajax 开发宝典</a>

  </p>

</div>

<!-- 使用 callout 标签 -->

<ajax:callout

//Ajax 请求发送的 URL

    baseUrl="callout"

//所有 class 属性为 book 的 HTML 元素都将触发 Ajax 请求

    sourceClass="book"

//请求参数

    parameters="book={ajaxParameter}"

    title="书籍详细信息"

/>
```

读者应该看到 parameters 属性的值为 book={ajaxParameter}，其中，ajaxParameter 是一个特殊的变量，这个变量代表任何发送请求的 HTML 元素本身。当“Spring2.0 宝典”发送请求时，该变量的值就是“Spring2.0 宝典”。图 18.5 显示了这种服务器提示的效果。



图 18.5 服务器提示

#### 18.4.5 使用 htmlContent 标签

该标签能将一个 HTML 页面的内容显示在当前页面的某个区域内（通常是一个 DIV 元素）。该标签不再需要 XML 响应，它只需要一个标准的 HTML 响应，这个 HTML 响应将直接输出页面的目标容器。该标签有如下几个属性。

**var:** 这是一个可选属性，该属性定义了 `autocomplete` 标签创建的 JavaScript 对象名。通常无须指定该属性。

**attachTo:** 这是一个可选属性，该属性定义了 `var` 对应的自动完成对象将应用到的对象。

**baseUrl:** 这是一个必需属性，该属性指定了 Ajax 请求发送的 URL。

**source:** 该属性指定哪个 HTML 元素将触发服务器提示框，即指定哪个 HTML 元素触发 Ajax 请求。必须指定 `source` 和 `sourceClass` 两个属性之一。

**sourceClass:** 该属性指定一类 HTML 元素将触发服务器提示框，即指定哪些 HTML 元素是该 CSS 样式单，这些 HTML 元素都可以触发 Ajax 请求。必须指定 `source` 和 `sourceClass` 两个属性之一。

**target:** 这是一个必需属性，该属性指定了 HTML 响应的输出容器。

**parameters:** 这是一个可选属性，如果需要发送请求参数，则需要使用该属性。

**preFunction:** 这是一个可选属性，该属性指定了 Ajax 交互之前自动执行的函数。

**postFunction:** 这是一个可选属性，该属性指定了 Ajax 交互完成后自动执行的函数。

**errorFunction:** 这是一个可选属性，该属性指定服务器响应出错时执行的函数。

下面是提供了 `htmlContent` 响应的 Servlet，该 Servlet 不再生成 XML 响应，而是生成 HTML 响应。

```
public class HtmlContentServlet extends BaseAjaxServlet
{
    //重写 getXmlContent 方法，生成服务器响应

    public String getXmlContent(HttpServletRequest request, HttpServletResponse
```

```
response) throws Exception

{
    //设置服务器解码方式
    request.setCharacterEncoding("UTF-8");
    //获取请求参数

    String publisher = request.getParameter("publisher");
    //创建业务逻辑组件实例

    BookService service = new BookService();
    //根据出版社获取所有的书籍

    List<Book> list = service.getBooksByPublisher(publisher);
    //开始拼接返回的字符串

    StringBuffer html = new StringBuffer();
    html.append("<h3>").append(publisher).append("出版的书籍包括如下</h3>");
    for (Book book: list)
    {
        html.append("<li>").append(book.getName()).append("</li>");
    }
    html.append("</ul>");
    html.append("<br>");
    html.append("最后更新时间:" + new Date());
    return html.toString();
}
}
```

正如在代码中看到的，该 Servlet 不再借助于 AjaxXmlBuilder 类来辅助生成 XML 响应，该 Servlet 不再返回一个 XML 文件，而是返回一个 HTML 文档。htmlContent 会将该响应直接输出在 HTML 文档的目标元素中。将该 Servlet 配置在 Web 应用中，在 web.xml 文件中增加如下片段：

```
<servlet>
```

```
<servlet-name>htmlContent</servlet-name>

<servlet-class>lee.HtmlContentServlet</servlet-class>

</servlet>

<servlet-mapping>

    <servlet-name>htmlContent</servlet-name>

    <url-pattern>/htmlContent</url-pattern>

</servlet-mapping>
```

在页面中使用 ajax:htmlContent 标签，本页面中使用了两种方式来输出 htmlContent 内容，一种是采用超链接，一种是采用下拉框。代码片段如下：

选择出版社查看详细信息: <br>

```
<ul>
```

```
<!-- 下面使用超链接来生成 htmlContent -->
```

```
<li><a href="javascript://nop/" class="publisher">电子工业出版社</a></li>
```

```
<li><a href="javascript://nop/" class="publisher">清华大学出版社</a></li>
```

```
<li><a href="javascript://nop/" class="publisher">机械工业出版社</a></li>
```

```
</ul>
```

```
<!-- 下面使用下拉列表来生成 htmlContent -->
```

```
<p>选择出版社查看详细信息: </p>
```

```
<select id="publishSelector" name="publishSelector">
```

```
    <option value="电子工业出版社">电子工业出版社</option>
```

```
    <option value="清华大学出版社">清华大学出版社</option>
```

```
    <option value="机械工业出版社">机械工业出版社</option>
```

```
</select>
```

```
<>
```

```
<div id="bookDesc" style="position:absolute;left:300px;top:20px;background-color:
#ffffaa"> </div>
```

```
<!-- 第一次使用 htmlContent 标签 -->
```

```
<ajax:htmlContent
    baseUrl="htmlContent"

//所有 class 属性为 publisher 的 HTML 元素都将发送 Ajax 请求

    sourceClass="publisher"

//指定输出 HTML 响应的目标容器

    target="bookDesc"

    parameters="publisher={ajaxParameter}"

/>

<!-- 第二次使用 htmlContent 标签 -->

<ajax:htmlContent
    baseUrl="htmlContent"

//指定 publisherSelector 元素发送 Ajax 请求

    source="publishSelector"

//指定输出 HTML 响应的目标容器

    target="bookDesc"

//发送请求参数

    parameters="publisher={publishSelector}"

/>
```

页面中的超链接和下拉框都可以激发 HTML 内容，一旦用户单击了超链接或者选择了下拉列表，都可以看到该出版社出版的所有图书。图 18.6 显示了 htmlContent 标签的效果。

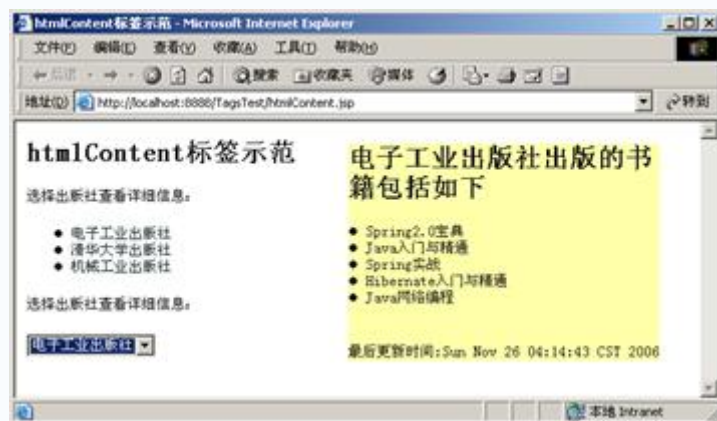


图 18.6 使用 htmlContent 输出 HTML 响应

### 18.4.6 使用 portlet 标签

portlet 标签将在页面上生成一个 Portlet 区域，该区域的内容直接显示服务器端 HTML 响应。类似于 htmlContent 标签，该标签不需要 XML 响应，而是支持 HTML 响应。使用 ajax:portlet 标签还可以定义该 Portlet 的内容是否支持周期性刷新。该标签包含如下几个属性。

**var:** 这是一个可选属性，该属性定义了 autoComplete 标签创建的 JavaScript 对象名。通常无须指定该属性。

**attachTo:** 这是一个可选属性，该属性定义了 var 对应的自动完成对象将应用到的对象。

**baseUrl:** 这是一个必需属性，该属性指定了 Ajax 请求发送的 URL。

**source:** 这是一个必需属性，该属性指定了 Portlet 的 ID 属性值。

**parameters:** 这是一个可选属性，该属性指定发送 Ajax 请求的请求参数。

**classNamePrefix:** 这是一个可选属性，该属性指定了 Portlet 的 Box, Tools, refresh, Size, Close, Title 和 Content 元素的 CSS 样式单。

**title:** 这是一个必需属性，该属性指定 Portlet 的标题。

**imageClose:** 这是一个可选属性，该属性指定关闭按钮的图标。

**imageMaximize:** 这是一个可选属性，该属性指定最大化按钮的图标。

**imageMinimize:** 这是一个可选属性，该属性指定最小化按钮的图标。

**imageRefresh:** 这是一个可选属性，该属性指定刷新按钮的图标。

**refreshPeriod:** 这是一个可选属性，该属性指定 Portlet 的内容刷新频率，即隔多少秒刷新一次。如果没有指定该属性，则 Portlet 的内容不会自动刷新，除非手动刷新。默认情况下，当页面加载时，Portlet 的内容也会刷新，但如果设置 executeOnLoad 为 false，则页面载入时，Portlet 的内容不会刷新，除非手动刷新。

**executeOnLoad:** 这是一个可选属性，该属性指定当页面重载时，是否重新检索 Portlet 里的内容，默认是重新检索。

**expireDays:** 这是一个可选属性，该属性指定 cookie 持久保存的天数。

**expireHours:** 这是一个可选属性，该属性指定 cookie 持久保存的小时数。

**expireMinutes:** 这是一个可选属性，该属性指定 cookie 持久保存的分钟数。

**preFunction:** 这是一个可选属性，该属性指定了 Ajax 交互之前自动执行的函数。

**postFunction:** 这是一个可选属性，该属性指定了 Ajax 交互完成后自动执行的函数。

**errorFunction:** 这是一个可选属性，该属性指定服务器响应出错时执行的函数。

该标签需要的响应完全类似于 htmlContent 标签的响应，此处不再单独为该标签编写服务器处理类，而是直接向 htmlContent 发送 Ajax 请求，因此可以在页面中直接使用 portlet 标签。使用 portlet 标签的代码片段如下：

```
<ajax:portlet
//指定 Portlet 的 ID 属性
    source="tsinghua"
//发送请求的 URL
    baseUrl='htmlContent'
```

```
//Portlet 必需的 CSS 样式

    classNamePrefix="portlet"

//Portlet 的标题

    title="清华大学出版社 Portlet"

//指定几个按钮的图标

    imageClose="img/close.png"

    imageMaximize="img/maximize.png"

    imageMinimize="img/minimize.png"

    imageRefresh="img/refresh.png"

//请求参数

parameters="publisher=清华大学出版社"

//每 5s 刷新一次该 Portlet

    refreshPeriod="5" />

<ajax:portlet

    source="phei"

    baseUrl='htmlContent'

    classNamePrefix="portlet"

    title="电子工业出版社 Portlet"

    imageClose="img/close.png"

    imageMaximize="img/maximize.png"

    imageMinimize="img/minimize.png"

    imageRefresh="img/refresh.png"

parameters="publisher=电子工业出版社"

    refreshPeriod="5" />
```

上面的页面使用了两个 Portlet，页面执行的效果如图 18.7 所示，其中清华大学出版社的 Portlet 已经被最小化了，因此看不到该 Portlet 的内容。





图 18.7 使用 portlet 标签生成 Portlet 效果

#### 18.4.7 使用 select 标签

select 标签就是在 18.2 和 18.3 节中频繁使用的标签，它的主要作用是实现级联下拉框效果。所谓级联下拉框，就是根据第一个下拉框选择的值，动态改变第二个下拉框的选项。select 标签有如下几个属性。

**var:** 这是一个可选属性，该属性定义了 autocomplete 标签创建的 JavaScript 对象名。通常无须指定该属性。

**attachTo:** 这是一个可选属性，该属性定义了 var 对应的自动完成对象将应用到的对象。

**baseUrl:** 这是一个必需属性，该属性指定了 Ajax 请求发送的 URL。

**source:** 这是一个必需属性，该属性指定了第一个下拉框的 ID 属性，当该属性指定的下拉框改变时触发 Ajax 交互。

**target:** 这是一个必需属性，该属性指定了第二个下拉框的 ID 属性，当 Ajax 响应完成后，AjaxTags 会根据响应来自动更新该属性指定的下拉框。

**parameters:** 这是一个可选属性，如果需要发送请求参数，则需要使用该属性。

**eventType:** 这是一个可选属性，该属性指定了源对象上触发请求的事件类型。

**executeOnLoad:** 这是一个可选属性。

**defaultOptions:** 这是一个可选属性，该属性是一系列以逗号隔开的值，这些值将总是作为第二个下拉框的默认选项。

**preFunction:** 这是一个可选属性，该属性指定了 Ajax 交互之前自动执行的函数。

**postFunction:** 这是一个可选属性，该属性指定了 Ajax 交互完成后自动执行的函数。

**errorFunction:** 这是一个可选属性，该属性指定服务器响应出错时执行的函数。

**parser:** 这是一个可选属性，该属性指定一个服务器响应的解析器。通常无须指定该解析器，除非用户需要自己完成特别的工作。默认的解析器是 ResponseHtmlParser。

因为在 18.2 和 18.3 节中已经大量使用了该标签，因此此处不再给出关于它的示范应用。值得注意的是，select 标签只能有一个源下拉框和一个目标下拉框，这往往不能满足实际需要。

例如，对于一个常用场景：第一个下拉框是国家，第二个下拉框是省份，第三个下拉框是城市。每个下拉框的值都应该随前面下拉框值的改变而改变。AjaxTags 的 select 也可以满足这个需要，只要使用两个 select 标签即可。

对于第一个 select 标签，国家下拉框是源下拉框，省份下拉框是目标下拉框；对于第二个 select 标签，省份下拉框是源下拉框，城市下拉框是目标下拉框。



使用 select 标签可以很方便地实现多级联动下拉框。

#### 18.4.8 创建 Tab 页

Tab 页的创建依赖于两个标签：tabPanel 和 tab。tabPanel 是一个整体的 Tab 效果，而每个 tab 则是该 Tab 效果里的每个 tab 页。因此，tabPanel 和 tab 两个标签通常一起使用。tabPanel 标签包含如下几个属性。

**var:** 这是一个可选属性，该属性定义了 autocomplete 标签创建的 JavaScript 对象名。通常无须指定该属性。

**attachTo:** 这是一个可选属性，该属性定义了 var 对应的自动完成对象将应用到的对象。

**panelStyleId:** 这是一个必需属性，指定 Tab 页的 ID 属性值。

**contentStyleId:** 这是一个必需属性，指定 Tab 页面内容的 ID 属性值。

**panelStyleClass:** 这是一个可选属性，该属性指定 Tab 页整体使用的 CSS 样式单。

**contentStyleClass:** 这是一个可选属性，该属性指定 Tab 页内容所使用的 CSS 样式单。

**currentStyleClass:** 这是一个必需属性，该属性指定了激活的 Tab 页所使用的 CSS 样式单。

**preFunction:** 这是一个可选属性，该属性指定了 Ajax 交互之前自动执行的函数。

**postFunction:** 这是一个可选属性，该属性指定了 Ajax 交互完成后自动执行的函数。

**errorFunction:** 这是一个可选属性，该属性指定服务器响应出错时执行的函数。

**parser:** 这是一个可选属性，该属性指定一个服务器响应的解析器。通常无须指定该解析器，除非用户需要自己完成特别的工作。默认的解析器是 ResponseHtmlParser。

tab 标签包含如下几个属性。

**baseUrl:** 这是一个必需属性，该属性指定了 Ajax 请求发送的 URL。

**caption:** 这是一个必需属性，该属性指定了 Tab 页的标题。

**defaultTab:** 这是一个可选属性，该属性指定页面是否作为 Tab 效果的初始页。

parameters: 这是一个可选属性, 如果加载页面时需要发送请求参数, 则需要使用该属性。

值得注意的是, tab 标签的 Ajax 响应无须使用 XML 响应, 而应该使用标准的 HTML 响应, tab 标签将该 HTML 内容直接输出在 Tab 页中。

下面的应用示范一个简单的 Tab 效果, 每个 Tab 页面的 baseUrl 都使用前面 htmlContent 中已经定义了的 htmlContent Servlet。下面是使用 tabPanel 和 tab 标签的代码片段:

```
<!-- 使用 tabPanel 构建整体的 Tab 效果 -->

<ajax:tabPanel

panelStyleId="tabPanel"

contentStyleId="tabContent"

panelStyleClass="tabPanel"

contentStyleClass="tabContent"

currentStyleClass="ajaxCurrentTab">

    <!-- tabPanel 的每个 tab 子标签对应一个 Tab 页 -->

    <ajax:tab caption="电子工业出版社"

baseUrl="htmlContent"

parameters="publisher=电子工业出版社"

defaultTab="true"/>

    <ajax:tab caption="清华大学出版社"

baseUrl="htmlContent"

parameters="publisher=清华大学出版社"/>

    <ajax:tab caption="机械工业出版社"

baseUrl="htmlContent"

parameters="publisher=机械工业出版社"/>

</ajax:tabPanel>
```

上面的代码将 ajax:tab 标签放在 ajax:tabPanel 内, 从而形成一个整体的 Tab 效果, 每个 tab 标签对应一个基本的 Tab 页, 每个 Tab 页所显示的 URL 完全相同, 请求参数不同, 每个 Tab 页的内容也不相同。图 18.8 显示了该 Tab 效果。



图 18.8 Tab 效果

#### 18.4.9 使用 displayTag 标签

这个标签需要依赖于 Apache 组织下的 DisplayTag 项目，AjaxTags 封装了 DisplayTag 项目，但增加了 Ajax 引擎，以便能以 Ajax 的方式来排序、分页。该标签的核心是 DisplayTags 项目，读者应该具有 DisplayTags 的相关知识。AjaxTags 中的 displayTag 标签有如下几个属性。

**id:** 这是一个必需属性，该属性指定了 displayTag 对应的 DIV 元素的 ID 属性值。

**ajaxFlag:** 这是一个可选属性，该属性指定该页面的其他部分是否忽略 Ajax 请求。

**style:** 内联 CSS 样式单属性。

**styleClass:** 这是一个可选属性，该属性指定 displayTag 的 CSS 样式单属性。

**pagelinksClass:** 这是一个可选属性，该属性指定 DisplayTag 的分页导航栏的 CSS 样式。

**tableClass:** 这是一个可选属性，该属性指定 DisplayTag 的 table 元素的 CSS 样式。

**columnClass:** 这是一个可选属性，该属性指定 DisplayTag 里 table 中每列的 CSS 样式。

**baseUrl:** 这是一个可选属性，没有太大的作用。

**postFunction:** 这是一个可选属性，该属性指定了 Ajax 交互完成后自动执行的函数。

为了使用 AjaxTags 的 displayTag 标签，必须先在 Web 应用中安装 DisplayTags 项目。安装 DisplayTags 可以按如下步骤进行：

**step 1** 将 displaytag-{version}.jar 文件复制到 Web 应用的 WEB-INF/lib 下。

**step 2** 将 DisplayTag 所依赖的 JAR 文件复制到 Web 应用的 WEB-INF/lib 下。这个步骤对于 AjaxTags 而言，往往已经完成了，因此无须额外复制。

**step 3** 如果使用 JSP 1.1 或者更早的容器，则应该将 displaytag-{taglibversion}.tld 文件复制到 Web 应用的 WEB-INF/ 路径下，并在 web.xml 文件中配置标签库。配置标签库的代码如下：

```
<taglib>
<!-- 标签库所在的 URI -->
<taglib-uri>http://displaytag.sf.net</taglib-uri>
<!-- 指定标签库 TLD 文件所在的物理路径 -->
    taglib-location>/WEB-INF/displaytag-{taglibversion}.tld</taglib-location>
</taglib>
```

**step 4**

如果需要使用Display的导出选项（这种导出选项非常有用，它可以将表格显示的内容直接导出成XML文档和Excel文档等），则应该在web.xml文件中配置如下：

```
<filter>
    <filter-name>ResponseOverrideFilter</filter-name>
    <filter-class>org.displaytag.filter.ResponseOverrideFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>ResponseOverrideFilter</filter-name>
    <url-pattern>/displaytag.jsp</url-pattern>
</filter-mapping>
```

**step 5**

如果需要自定义DisplayTag显示的某些效果，则还需要增加一个displaytag.properties文件，关于该文件的各种属性以及具体含义可以参考DisplayTag的官方文档。下面是本示例应用增加在WEB-INF/classes路径下的displaytag.properties文件：

```
sort.behavior=list
sort.amount=list
basic.empty.showtable=true
basic.msg.empty_list=找不到满足要求的结果
paging.banner.placement=bottom
paging.banner.some_items_found=查询到{0}条记录，当前显示从{2}到{3}条记录.
export.types=csv excel xml
export.amount=list
export.csv=false
export.excel=true
export.pdf=true
export.xml=false
export.excel.include_header=true
```

因为该文件中包含了中文字符，因此还必须使用 native2ascii 命令将该属性文件转为国际化的属性文件。经过这 5 个步骤，该 Web 应用就可以使用 AjaxTags 的 displayTag 标签了。在页面中使用 displayTag 标签的代码如下：

```
<jsp:useBean id="now" class="java.util.Date"/>
<!-- 直接初始化业务逻辑组件 -->
<jsp:useBean id="service" class="lee.BookService" />
```

<!-- 将 display 标签放在 ajax:displayTag 标签内，以便可以以 Ajax 方式进行排序、分页 -->

<ajax:displayTag id="displayTagFrame" ajaxFlag="displayAjax">

最后更新时间: \${now}

<display:table name="service.allBooks" class="displaytag" pagesize="10"

scope="page" defaultsort="1" defaultorder="descending" export="true"

id="row" excludedParams="ajax">

<!-- 输出业务逻辑组件中的两列 -->

<display:column property="name" title="书名" sortable="true" headerClass="sortable" />

<display:column property="publisher" title="出版社" sortable="true" headerClass="sortable" />

</display:table>

</ajax:displayTag>

在浏览器中浏览该页面，将看到如图 18.9 所示的效果。

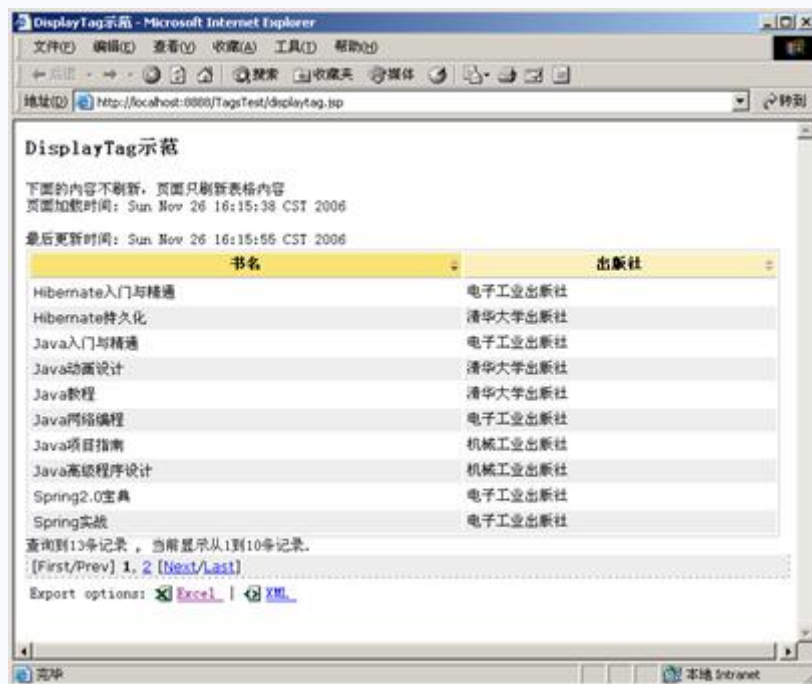
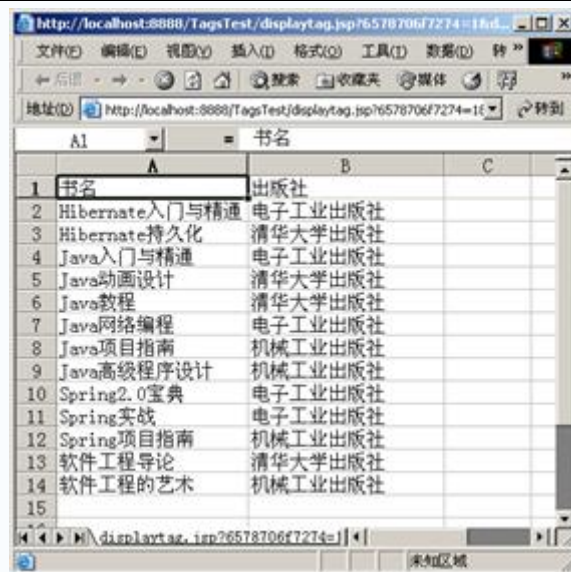


图 18.9 使用 AjaxTags 的 displayTag 标签

可以看到，页面中两次输出的时间并不相同，那是因为这里已经单击了“书名”列，从而按书名排序了，只是这种排序是以 Ajax 方式进行的，因此只刷新表格部分，并未刷新整个页面内容，从而看到两个时间并不相同。

如果单击表格下面的分页导航，则可看到以 Ajax 方式完成分页。如果单击下面的导出 Excel 按钮，将可以看到如图 18.10 所示的界面，这是 DisplayTag 的功能，与 Ajax 的 displayTag 并没有什么关系。





	A1	= 书名
	A	B
1	书名	出版社
2	Hibernate入门与精通	电子工业出版社
3	Hibernate持久化	清华大学出版社
4	Java入门与精通	电子工业出版社
5	Java动画设计	清华大学出版社
6	Java教程	清华大学出版社
7	Java网络编程	电子工业出版社
8	Java项目指南	机械工业出版社
9	Java高级程序设计	机械工业出版社
10	Spring2.0宝典	电子工业出版社
11	Spring实战	电子工业出版社
12	Spring项目指南	机械工业出版社
13	软件工程导论	清华大学出版社
14	软件工程的艺术	机械工业出版社
15		

图 18.10 导出 Excel 文档的效果

#### 18.4.10 使用 updateField 标签

这个标签实现了一种常用的效果：当一个表单域的输入完成后，其他几个表单域的值根据该表单域的值计算得到。在大部分时候，如果这种计算无须服务器数据参与，则可以在客户端通过 JavaScript 计算完成。在某些情况下，例如，商品的折扣是通过后台程序设定的，则需要服务器数据的参与，因此应该使用该标签来完成。updateField 标签有如下几个属性。

**var:** 这是一个可选属性，该属性定义了 autocomplete 标签创建的 JavaScript 对象名。通常无须指定该属性。

**attachTo:** 这是一个可选属性，该属性定义了 var 对应的自动完成对象将应用到的对象。

**baseUrl:** 这是一个必需属性，该属性指定了 Ajax 请求发送的 URL。

**source:** 这是一个必需属性，该属性指定一个表单域，该表单域的值将作为请求参数随 Ajax 请求向服务器发送。

**target:** 这是一个必需属性，该属性的值以逗号隔开，指定了一系列的表单域，Ajax 响应的结果将在这些表单域中输出。

**action:** 这是一个必需属性，该属性指定的 HTML 元素能触发 onclick 事件，该事件将触发 Ajax 交互。

**parameters:** 这是一个可选属性，该属性指定需要发送到服务器端的请求参数。

**eventType:** 这是一个可选属性，该属性指定能触发 Ajax 请求的事件类型。

**preFunction:** 这是一个可选属性，该属性指定了 Ajax 交互之前自动执行的函数。

**postFunction:** 这是一个可选属性，该属性指定了 Ajax 交互完成后自动执行的函数。

**errorFunction:** 这是一个可选属性，该属性指定服务器响应出错时执行的函数。

parser: 这是一个可选属性, 该属性指定一个服务器响应的解析器, 默认采用 ResponseHtmlParser 解析器; 如果使用 XML 响应, 则通常指定为 ResponseXmlParser。

值得注意的是, 该标签的响应一样是一个标准的 AjaxTags 响应, 该响应包含的 item 节点数应与需要动态计算的表单域的数量相等, 而且每个 item 节点的 name 节点值应与目标表单域的 ID 属性相同。

下面的应用示范了通过一个初始价格计算出五星级会员、四星级会员、三星级会员、二星级会员和一星级会员的会员价。计算打折价的 Servlet 的代码如下:

```
public class CalDiscountServlet extends BaseAjaxServlet
{
    //重写 getXmlContent 方法, 该方法返回的 XML 字符串作为 Ajax 请求的响应
    public String getXmlContent(HttpServletRequest request, HttpServletResponse
        response)
    {
        //price 为初始价
        double price = 0;
        //下面 5 个变量分别为不同级别会员的打折价
        double five = 0;
        double four = 0;
        double three = 0;
        double two = 0;
        double one = 0;
        //获取请求参数
        price = Double.parseDouble(request.getParameter("price"));
        //调用服务器计算
        five = price * 0.7;
        four = price * 0.8;
        three = price * 0.85;
        two = price * 0.9;
        one = price * 0.95;
        //构造响应的 XML 字符串, 并返回
        return new AjaxXmlBuilder()
            .addItem("five", Double.toString(five))
            .addItem("four", Double.toString(four))
            .addItem("three", Double.toString(three))
            .addItem("two", Double.toString(two))
            .addItem("one", Double.toString(one))
            .toString();
    }
}
```

上面代码中的 addItem 有两个参数: 第一个参数分别为 five 和 four 等, 这些参数并不是随意填写的, 应与页面中需要通过服务器计算的表单域的 ID 属性相同, 即页面中的 five 表单域的值等于 Double.toString(five)。

页面中使用 updateField 标签来完成该 Ajax 交互, 因为同时有 5 个表单域需要通过计算得到, 因此 target 的值为以逗号隔开的 5 个值。下面是页面中使用 updateField 的代码片段:



```
<ajax:updateField
    baseUrl="calDiscount"
//发送自动计算的源表单域
    source="price"
//下面 5 个表单域将通过计算得到
    target="five,four,three,two,one"
//action 元素触发 Ajax 请求
    action="action"
//发送的请求参数
    parameters="price={price}"
    parser="new ResponseXmlParser()"/>
```

在页面中的初始价格文本框中输入“80”，然后单击“计算”按钮，将出现如图 18.11 所示的界面。



图 18.11 服务器计算表单域的值

## 18.5 关于 AjaxTags 的选择

正如前面介绍的，通过使用 AjaxTags 标签完成一个 Ajax 应用是如此简单，对于常见的 Ajax 应用场景，AjaxTags 都提供了对应的封装，程序开发者只需要使用 JSP 标签即可开发出功能完备的 Ajax 应用。但 AjaxTags 并不是万能的，有些时候，我们必须放弃 AjaxTags，选择更烦琐的开发方式。

### 18.5.1 AjaxTags 的优势和使用场景

AjaxTags 的优势相当明显，当 Ajax 技术面世时，有这样一种说法：Ajax 通过对程序员的折磨来取悦用户。这种说法在某种程度上是对的，但所有的技术都以改善用户感受为最终目标，对于一个程序员而言，能带给用户更好的体验就是最大的成就。

Ajax 技术的烦琐不言而喻，JavaScript 本身不是一门严格的语言，缺乏严格的调试机制，即使在底层所有响应完成后，程序员还必须在表现层完成异常烦琐的 DOM 更新，还必须应用 CSS 样式。如果再加上跨浏览器支持、向后兼容性等一系列的技巧，那么开发一个普通的 Ajax 页面可能需要两天时间，这简直不可想象。

多亏了大量的 JavaScript 库，例如 Prototype.js 和 Dojo 等，但即使使用这些 JavaScript 库，我们依然需要面对很多问题，依然需要动态更新 DOM，依然必须编写大量的 JavaScript 代码。

实际上，大量的 Ajax 应用场景重复出现，级联下拉框、自动完成、页面提示.....每个常用的 Ajax 交互都需要花费大量的时间和精力。

AjaxTags 对这些常用的 Ajax 交互场景提供了封装，程序开发者几乎无须编写任何 JavaScript 代码就可以完成一个专业的 Ajax 应用。特别是对于 J2EE 应用开发者而言，编写一个传统的 Servlet，并将该 Servlet 配置在 Web 应用中，然后在页面中使用 Ajax 标签即可完成一个 Ajax 应用，相当简单。

AjaxTags 最大的优点是简单，J2EE 应用开发者甚至无须了解 Ajax 技术细节，只需要会编写 Servlet，会使用 JSP 标签，就可以开发出专家级的 Ajax 应用，这是 AjaxTags 提供的最大好处。

因为 AjaxTags 简单，所以也可以大大节省开发者的时间。

对于所有能使用 AjaxTags 的情况，推荐优先考虑使用 AjaxTags，因为使用 AjaxTags 既可以节省时间，也可以避免错误。AjaxTags 的更新非常快，经常有新的标签加入，每个 beta 版之间的标签数量、标签的用法也存在差异，希望读者在使用 AjaxTags 时到其官方网站看一看 AjaxTags 的最新版包含了那些简便的标签。

### 18.5.2 AjaxTags 的缺点

AjaxTags 以简单、快捷的特性方便了 J2EE 的 Ajax 开发者，但它也不是万能的，在某些情形下，它依然存在一些小小的缺陷。AjaxTags 大致存在如下缺陷：

AjaxTags 只能在 J2EE 应用环境下使用，不能在其他 Web 编程脚本语言（如 ASP 和 PHP 等）中使用。

AjaxTags 的高度封装虽然简化了 Ajax 的开发，但导致灵活性严重丧失；对于复杂的 Ajax 交互，使用 AjaxTags 完成更加烦琐。

AjaxTags 对 Ajax 交互提供了封装，但不像 Dojo 那样提供了一个调试环境。整个 Ajax 交互不仅对普通浏览者透明，对于应用开发者也是透明的，调试难度相对较大。

虽然存在这些缺点，但 AjaxTags 的简单远远可以掩盖这些缺陷，在能使用 AjaxTags 标签的地方，应该尽量考虑使用 AjaxTags。如果需要对 AjaxTags 进行大量扩展、修改，则应该考虑使用其他技术。毕竟，AjaxTags 与其他 Ajax 技术并不是互斥的，例如 Prototype.js 本身就是 AjaxTags 所依赖的技术；如果有必要，还可以引入 Dojo。

## 18.6 小结

本章从 AjaxTags 的安装开始，介绍了 AjaxTags 处理类的编写和配置，以及 AjaxTags 处理类的几种形式。本章重点介绍了 AjaxTags 的常用标签以及标签里各属性的含义，并且结合示例程序介绍了如何使用这些标签。本章最后还介绍了 AjaxTags 标签的优点、缺点及其适合的应用场景。

至此，本书关于 Ajax 框架的内容就介绍完毕了。至于如何选择这些框架，就是“运用之妙，存乎一心”的事情了，不能强行规定用户应该使用哪个框架，放弃哪个框架。就经验而看，以一个专业的 Ajax 框架为主，以 JavaScript 库为辅是不错的选择。后面的章节将通过几个具体的应用来讲述如何使用这些框架开发真实的 Ajax 应用。

## 第 19 章 Ajax 案例：Blog 系统

- ◆ 系统的持久层设计
- ◆ 实现 DAO 组件
- ◆ 实现业务逻辑组件
- ◆ 使用 Spring 容器管理系统的各个组件
- ◆ 使用 DWR 将 Spring 容器中的 Bean 暴露成 JavaScript 对象
- ◆ 在 JavaScript 中调用 Spring 容器中的 Bean

本章将介绍一个基于 Ajax 技术的 Blog 系统，这个系统允许发表 Blog 文章和评论。本系统并不是一个完善的 Blog 系统，因为它没有进行任何权限控制，就是说任何用户都可以在本系统内发表 Blog 文章。本书将在第 20 章的综合应用中介绍更完善的权限控制，让 Ajax 和 Spring 的 AOP 结合来提供用户的权限控制。

该 Blog 系统的后台系统依然以 Hibernate 作为持久层的 ORM 框架，使用 Spring 作为中间层的 IoC 容器，将各组件以松散耦合的方法组织在一起。DAO 对象的实现采用 Spring 的 DAO 支持。Ajax 引擎使用了 DWR，让 DWR 直接访问 Spring 容器中的业务逻辑组件，DWR 的这个功能非常实用，它可以利用 Spring 容器的强大功能。

### 19.1 实现 Hibernate 持久层

与前面的 J2EE 应用类似，本章同样采用 Hibernate 作为应用的持久层。使用 Hibernate 作为持久层的访问框架，可使用面向对象的方式操作关系数据库。

#### 19.1.1 设计 Hibernate 的持久化类

本应用包含两个实体类：Blog 文章和文章评论，因为本应用没有涉及用户管理，因此不需要用户登录和用户权限检查等功能，无须保存用户实体。如果需要扩展本系统，则可以考虑增加用户实体，并提供相关功能。

本系统设计的两个实体类保存了各自必需的信息，Blog 文章类保存了文章标题和文章内容等，文章评论类则保存了评论者的用户名、评论内容、评论者的电子邮件等信息。

两个实体存在 1 : N 关联，即一篇 Blog 文章对应 N 个评论。Hibernate 以面向对象的方式操作数据库，因此它完全可以理解这种关联。Blog 文章对应的实体类的代码如下：

```
public class Blog
{
    //标识属性，对应 Blog 文章在数据表中的主键
```

```
private int id;

//Blog 文章的标题

    private String title;

//Blog 文章的内容

    private String content;

//Blog 文章的添加时间

private Date addTime;

//每篇 Blog 文章对应多个评论

private Set<Comment> comments = new HashSet<Comment>();

//无参数的构造器

public Blog()

{

}

//带初始化参数的构造器

public Blog(int id, String title, String content, Date addTime) {

    this.id = id;

    this.title = title;

    this.content = content;

    this.addTime = addTime;

}

//标识属性的 setter 方法

public void setId(int id)

{

    this.id = id;

}

//此处省略其他属性的 setter 方法

.....
```

```
//Blog 文章所关联的评论的 setter 方法

public void setComments(Set<Comment> comments)

{

    this.comments = comments;

}

//标识属性的 getter 方法

public int getId()

{

    return (this.id);

}

//此处省略其他属性的 getter 方法

.....

//Blog 文章所关联的评论的 getter 方法

public Set<Comment> getComments()

{

    return (this.comments);

}

}
```

上面的 Blog 类中包含了一个 Set 类型的属性，该属性指向一系列的持久化类：Comment。对于一篇 Blog 文章而言，可以包含多个评论，即存在 1：N 关联。每篇 Blog 文章都可以包含多篇评论，提供了关联映射后，每篇 Blog 文章都可以直接访问与之关联的评论。

因为篇幅关系，上面的持久化类省略了 Blog 文章其他属性的 getter 和 setter 方法，这些 getter 和 setter 方法都非常简单，相信读者可以将这些方法补齐。Blog 文章评论对应的持久化类的代码如下：

```
public class Comment

{

    //Blog 文章的标识属性，对应于数据表中的主键

    private int id;
```

```
//发表评论的用户名
    private String user;

//评论者的电子邮件
    private String email;

//评论者 Blog 的 URL 地址
    private String url;

//评论的内容
    private String content;

//评论时间
    private Date addTime;

//评论所对应的 Blog 文章
    private Blog blog;

//无参数的构造器
    public Comment()
    {
    }

//带初始化参数的构造器
    public Comment(int id, String user, String email, String url,
        String content, Date addTime, Blog blog)
    {
        this.id = id;
        this.user = user;
        this.email = email;
        this.url = url;
        this.content = content;
        this.addTime = addTime;
        this.blog = blog;
    }
```

```
}

//标识属性的 setter 方法

public void setId(int id)

{

    this.id = id;

}

//此处省略其他属性的 setter 方法

.....

//评论所对应的 Blog 文章的 setter 方法

public void setBlog(Blog blog)

{

    this.blog = blog;

}

//标识属性的 getter 方法

public int getId()

{

    return (this.id);

}

//此处省略其他属性的 getter 方法

.....

//评论所对应的 Blog 文章的 getter 方法

public Blog getBlog()

{

    return (this.blog);

}

}
```



评论和 Blog 文章之间存在 N : 1 的关联关系, 每个评论实例都有一篇与之对应的 Blog 文章, 评论可以通过这种关联关系直接访问与之对应的 Blog 文章。在上面的代码中, 评论持久化类里保存了一个 Blog 类型的属性, 该属性就是评论所对应的 Blog 文章。一旦完成了上面持久化类的定义, 增加 Hibernate 配置文件和映射文件后便可进行持久层访问。

### 19.1.2 完成映射文件

仅有上面定义的 POJO 还不足以访问持久层数据库, Hibernate 需要 XML 映射文件, 该映射文件定义了 POJO 的属性和数据表的对应关系。当开发者以面向对象的方式操作 POJO 时, Hibernate 通过这种对应关系将这种面向对象的操作转换为对数据库的操作。因此, 这种 Hibernate 映射文件是非常重要的。

Hibernate 完全可以理解关联映射, 对于 1 : N 关联, 采用双向关联性能更好, 最好做成反向的双向关联, 即 1 的一端只能访问 N 的一端, 但不能控制关联关系。本应用的 Blog 文章和评论之间是典型的 1 : N 关联, 因此采用 1 : N 关联映射。下面是 Comment 持久化类的映射文件, 这个映射文件增加 many-to-one 引用关联类:

```
<?xml version="1.0" encoding="GBK"?>
<!-- 指定 Hibernate 映射文件的 DTD 等信息 -->
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="org.yeeeku.model">
<!-- 每个 class 元素映射一个持久化类 -->
<class name="Comment" table="comment_table">
    <!-- 映射标识属性 -->
    <id name="id">
        <!-- 指定主键生成器策略 -->
        <generator class="identity"/>
    </id>
    <!-- 映射普通属性: 评论者的用户名 -->
    <property name="user"/>
</class>
</hibernate-mapping>
```

```
<!-- 映射普通属性: 评论者的电子邮件 -->
<property name="email"/>
<!-- 映射普通属性: 评论者 Blog 的 URL -->
<property name="url"/>
<!-- 映射普通属性: 评论内容 -->
<property name="content"/>
<!-- 映射普通属性: 评论时间 -->
<property name="addTime"/>
<!-- 映射关联的持久化类: Blog -->
<many-to-one name="blog" column="blog_id" not-null="true"/>
</class>
</hibernate-mapping>
```

上面的映射文件完成了 Blog 文章评论的映射, 在这个映射文件中, blog 属性映射到关联持久化类: Blog 文章。

下面是 Blog 文章的映射文件, 因为每篇 Blog 文章对应多篇评论, 因此在 Blog 文章的映射文件中增加 set 元素, 表明映射的关联属性是系列的评论。

```
<?xml version="1.0" encoding="GBK"?>
<!-- 指定 Hibernate 映射文件的 DTD 等信息 -->
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="org.yee ku.model">
<!-- 每个 class 元素映射一个持久化类 -->
<class name="Blog" table="blog_table">
    <!-- 映射标识属性 -->
    <id name="id">
        <!-- 指定主键生成器策略 -->
```

```
<generator class="identity"/>

</id>

<!-- 映射 Blog 文章的标题 -->

<property name="title"/>

<!-- 映射 Blog 文章的内容 -->

<property name="content" type="text"/>

<!-- 映射 Blog 文章的添加时间 -->

<property name="addTime"/>

<!-- 映射 Blog 文章的 1:N 关联 -->

<set name="comments" inverse="true">

    <!-- 指定外键列的列名 -->

    <key column="blog_id"/>

    <!-- 映射到关联持久化类: Comment -->

    <one-to-many class="Comment"/>

</set>

</class>

</hibernate-mapping>
```

当进行双向的 1 : N 关联时，两边都应该指定外键列的列名。因为两边指定的外键列实际上是同一列，因此两边指定的列名应该相同。对于 1 : N 关联，通常不推荐使用 1 的一端控制关系，因此在上面的映射文件中，为 set 元素增加 inverse="true" 属性。



对于双向的 1 : N 关联，两边指定的外键列的列名应该相同。

完成上面的映射后，Hibernate 便可以理解数据表和 POJO 之间的对应关系了，但 Hibernate 不知道连接哪个数据库以及连接数据库的全局属性——因为没有配置 Hibernate 的 SessionFactory 属性。

本应用将使用 Spring 来管理所有的 DAO 组件,因此将 Hibernate 的 SessionFactory 放在 Spring 容器中管理,即在 Spring 配置文件中增加如下片段:

```
<!-- 配置数据源, 使用 C3P0 数据源 -->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
destroy-method="close">
<!-- 配置数据源连接的数据库 URL -->
<property name="jdbcUrl" value="jdbc:mysql://localhost:3306/blog"/>
<!-- 配置连接数据源所用的驱动 -->
<property name="driverClass" value="com.mysql.jdbc.Driver"/>
<!-- 配置连接数据源所用的用户名 -->
<property name="user" value="root"/>
<!-- 配置连接数据源所用的密码 -->
<property name="password" value="32147"/>
<!-- 配置连接池最大连接数 -->
<property name="maxPoolSize" value="40"/>
<!-- 配置连接池最小连接数 -->
<property name="minPoolSize" value="1"/>
<!-- 配置连接池初始连接数 -->
<property name="initialPoolSize" value="1"/>
<!-- 配置连接池里连接的最大空闲时间 -->
<property name="maxIdleTime" value="20"/>
</bean>
<!--定义了 Hibernate 的 SessionFactory -->
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.
LocalSessionFactoryBean">
<!-- 指定 SessionFactory 所需的数据源 -->
<property name="dataSource" ref="dataSource"/>
<!-- 添加系统所使用的映射文件 -->
<property name="mappingResources">
<list>
<value>Blog.hbm.xml</value>
<value>Comment.hbm.xml</value>
</list>
</property>
<!-- 指定 Hibernate 的属性 -->
<property name="hibernateProperties">
<props>
<!-- 连接数据库所用的方言 -->
<prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
<!-- 是否显示 Hibernate 数据库访问中的 SQL 语句 -->
<prop key="show_sql">true</prop>
<prop key="hibernate.hbm2ddl.auto">update</prop>
<prop key="hibernate.jdbc.batch_size">20</prop>
```

```
</props>
</property>
</bean>
```

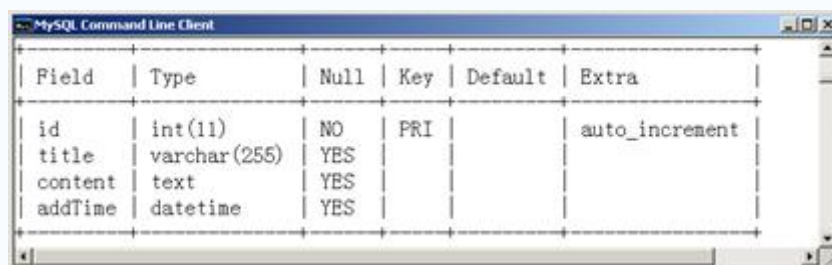
经过上面的配置，Hibernate 可以理解系统需要操作的数据库以及连接数据的全局属性，同时也能理解 POJO 和数据表之间的对应关系。当然，本应用中的 DAO 组件借助于 Spring 的 DAO 支持实现，而不是直接通过 Hibernate 访问完成。

### 19.1.3 数据表的结构

对于采用这种 OOA 和 OOD 思路完成的应用，数据表的结构通常无须开发者手动建立，开发者只要设计对应的 POJO，并定义合适的映射文件，系统就可以自动生成所需的数据表，包括数据表的主、外键约束。

本应用中有两个实体类，两个实体类之间使用无连接表的 1：N 映射策略，系统将根据持久化类生成两个表，分别用于存放 Blog 文章和文章评论。

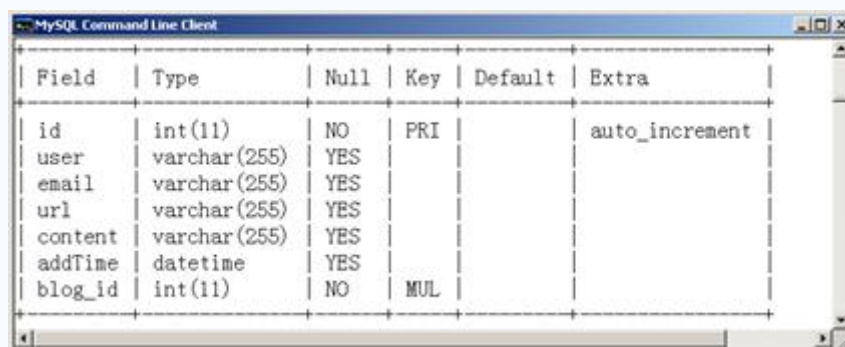
图 19.1 是用于存放 Blog 文章的表结构。



Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI		auto_increment
title	varchar(255)	YES			
content	text	YES			
addTime	datetime	YES			

图 19.1 存放 Blog 文章的表结构

Blog 文章是 1：N 关联中 1 的一端，对于采用主外键约束的关联，该表无须保留外键，外键将保存在 N 的一端。图 19.2 是存放评论的表结构。



Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI		auto_increment
user	varchar(255)	YES			
email	varchar(255)	YES			
url	varchar(255)	YES			
content	varchar(255)	YES			
addTime	datetime	YES			
blog_id	int(11)	NO	MUL		

图 19.2 存放评论的表结构

两个表之间通过主、外键建立约束，将 Comment 表中的 blog\_id 列参照到 Blog 表中的 id 列。

## 19.2 实现 DAO 组件

J2EE 应用的持久层访问都是通过 DAO 组件来完成的，DAO 组件封装了所有的数据库访问，让数据库访问与业务逻辑组件分离，从而将数据库访问逻辑和业务逻辑分离。使用 DAO 模式的原因是为了实现更好的分离。

本系统有两个 DAO 组件，这两个 DAO 组件都借助于 Spring 的 DAO 抽象实现。DAO 接口里提供了各种方法定义，DAO 实现类则对这些方法提供实现。

### 19.2.1 DAO 接口定义

Blog DAO 组件负责 Blog 文章的数据库访问操作，包括 Blog 记录的增加、删除、修改和查询等，下面是 Blog DAO 组件的代码：

```
public interface BlogDao
{
    /**
     * 根据主键加载 Blog
     * @param id 需要加载的 Blog ID
     * @return 加载的 Blog
     */
    Blog get(int id);

    /**
     * 保存 Blog
     * @param b 需要保存的 Blog
     */
    void save(Blog b);

    /**
     * 删除 Blog
     * @param b 需要删除的 Blog
     */
}
```

```
void delete(Blog b);

/**
 * 删除 Blog
 * @param id 需要删除的 Blog ID
 */

void delete(int id);

/**
 * 修改 Blog
 * @param b 需要修改的 Blog
 */

void update(Blog b);

/**
 * 查询指定用户、指定页的 Blog
 * @param pageNo 需要查询的指定页
 * @return 查询到的 Blog 集合
 */

List findAllByPage(int pageNo);

/**
 * 查询最新 Blog 文章的 ID
 * @return 最新 Blog 文章
 */

Blog findLastest();
}
```

正如前面所见的，Blog DAO 接口中定义了基本的 CRUD 操作方法，用于完成 Blog 记录的增加、删除和修改等操作。除此之外，还定义了两个简单的查询方法：findAllByPage 和 findLastest，这两个方法分别用于查询所有的 Blog 文章和最新的 Blog 文章。查询所有的 Blog 文章时，提供了一个页码参数，用于控制分页。Comment DAO 接口的代码如下：

```
public interface CommentDao
{
    /**
     * 根据主键加载评论
     * @param id 需要加载的评论 ID
     * @return 加载的评论
     */
    Comment get(int id);

    /**
     * 保存评论
     * @param c 需要保存的评论
     */
    void save(Comment c);

    /**
     * 删除评论
     * @param c 需要删除的评论
     */
    void delete(Comment c);

    /**
     * 删除评论
     * @param id 需要删除的评论的 ID
     */
    void delete(int id);

    /**
     * 修改评论
     * @param c 需要修改的评论
     */
}
```



```
void update(Comment c);

/**
 * 根据评论名查找评论
 * @param b 评论所对应的 Blog
 * @param pageNo 查找指定页的 Blog
 * @return 查找到的评论集合
 */

List findByBlogByPage(Blog b, int pageNo);
}
```

Comment DAO 接口里一样定义了 Blog 评论的增加、删除和修改等方法，除此之外，还定义了一个查询方法：findByBlogByPage，该方法根据 Blog 文章和页码查询评论。

### 19.2.2 分页实现

本系统的查询方法都使用了分页，Spring 的 HibernateTemplate 并未提供分页的实现，因此，这里扩展了 Spring 的 HibernateDaoSupport 类，扩展后的 HibernateDaoSupport 增加了 3 个分页查询方法，使用这 3 个分页查询方法可以实现分页。关于为什么选择扩展 HibernateDaoSupport，而不是扩展 HibernateTemplate，可以参考第 15 章的介绍。扩展后的 HibernateDaoSupport 的代码如下：

```
public class YeekuHibernateDaoSupport extends HibernateDaoSupport
{
    /**
     * 使用 hql 语句进行分页查询操作
     * @param hql 需要查询的 hql 语句
     * @param offset 第一条记录索引
     * @param pageSize 每页需要显示的记录数
     * @return 当前页的所有记录
     */

    public List findByPage(final String hql,
```

```
        final int offset, final int pageSize)
    {
        List list = getHibernateTemplate().executeFind(new HibernateCallback()
        {
            public Object doInHibernate(Session session)
                throws HibernateException, SQLException
            {
                List result = session.createQuery(hql).
                    setFirstResult(offset).setMaxResults(pageSize).
                    list();
                return result;
            }
        });
        return list;
    }
}

/**
 * 使用 hql 语句进行分页查询操作
 * @param hql 需要查询的 hql 语句
 * @param value 如果 hql 有一个参数需要传入，那么 value 就是传入的参数
 * @param offset 第一条记录索引
 * @param pageSize 每页需要显示的记录数
 * @return 当前页的所有记录
 */
public List findByPage(final String hql, final Object value,
    final int offset, final int pageSize)
{
    List list = getHibernateTemplate().executeFind(new HibernateCallback()
```

```
{
    public Object doInHibernate(Session session)
        throws HibernateException, SQLException
    {
        List result = session.createQuery(hql).
            setParameter(0, value).setFirstResult(offset).
            setMaxResults(pageSize).list();
        return result;
    }
});
return list;
}

/**
 * 使用 hql 语句进行分页查询操作
 * @param hql 需要查询的 hql 语句
 * @param values 如果 hql 有多个参数需要传入，那么 values 就是传入的参数数组
 * @param offset 第一条记录索引
 * @param pageSize 每页需要显示的记录数
 * @return 当前页的所有记录
 */
public List findByPage(final String hql, final Object[] values,
    final int offset, final int pageSize)
{
    List list = getHibernateTemplate().executeFind(new HibernateCallback()
    {
        public Object doInHibernate(Session session)
            throws HibernateException, SQLException
```

```
{  
    Query query = session.createQuery(hql);  
    for (int i = 0; i < values.length; i++)  
    {  
        query.setParameter( i, values[i]);  
    }  
    List result = query.setFirstResult(offset).  
        setMaxResults(pageSize).list();  
    return result;  
}  
});  
return list;  
}  
}
```

上面的 YeekuHibernateDaoSupport 类中提供了 3 个查询方法,这 3 个查询方法分别用于无参数的 HQL 查询、一个参数的 HQL 查询和多个参数的 HQL 查询。借助于这 3 个查询方法,可以实现查询的分页。

### 19.2.3 DAO 组件的实现

DAO 组件的实现类实现了 DAO 接口,继承了上面的 YeekuHibernateDaoSupport 类。YeekuHibernateDaoSupport 是 HibernateDaoSupport 的子类,可以使用 Spring 的 DAO 支持。下面是 BlogDaoHibernate 实现类的代码:

```
public class BlogDaoHibernate extends YeekuHibernateDaoSupport  
implements BlogDao  
{  
    //每页显示的消息数  
    private int pageSize;  
    //每页的消息数通过依赖注入管理
```

```
public void setPageSize(int pageSize)
{
    this.pageSize = pageSize;
}

/**
 * 根据主键加载 Blog
 * @param id 需要加载的 Blog ID
 * @return 加载的 Blog
 */
public Blog get(int id)
{
    return (Blog) getHibernateTemplate().get(Blog.class, new Integer(id));
}

/**
 * 保存 Blog
 * @param b 需要保存的 Blog
 */
public void save(Blog b)
{
    getHibernateTemplate().save(b);
}

/**
 * 删除 Blog
 * @param b 需要删除的 Blog
 */
public void delete(Blog b)
{

```

```
        getHibernateTemplate().delete(b);
    }

    /**
     * 删除 Blog
     * @param id 需要删除的 Blog ID
     */
    public void delete(int id)
    {
        getHibernateTemplate().delete(get(id));
    }

    /**
     * 修改 Blog
     * @param b 需要修改的 Blog
     */
    public void update(Blog b)
    {
        getHibernateTemplate().saveOrUpdate(b);
    }

    /**
     * 查询指定用户、指定页的 Blog
     * @param pageNo 需要查询的指定页
     * @return 查询到的 Blog 集合
     */
    public List findAllByPage(int pageNo)
    {
        int offset = (pageNo - 1) * pageSize;

        return findByPage("from Blog as b order by b.id desc", offset, pageSize);
    }
}
```

```
}

/**
 * 查询最新 Blog 文章的 ID
 * @return 最新 Blog 文章
 */
public Blog findLastest()
{
    List l = getHibernateTemplate().find("from Blog as b where b.id >=
        (select max(b.id) from Blog as b)");
    if (l == null || l.size() == 0 || l.get(0) == null)
    {
        return null;
    }
    return (Blog)l.get(0);
}
}
```

在上面的实现类中，除了实现 DAO 接口中定义的系列方法外，还提供了一个依赖注入方法，这个依赖注入方法用于注入一个 pageSize 变量值，这个变量值就是每页显示的记录数。

CommentDaoHibernate 实现类与之类似，也提供了每页显示记录数的依赖注入方法。下面是该 DAO 组件实现类的代码：

```
public class CommentDaoHibernate extends YeekuHibernateDaoSupport
implements CommentDao
{
    //每页显示的消息数
    private int pageSize;

    //每页的消息数通过依赖注入管理
    public void setPageSize(int pageSize)
```

```
{
    this.pageSize = pageSize;
}

/**
 * 根据主键加载评论
 * @param id 需要加载的评论 ID
 * @return 加载的评论
 */
public Comment get(int id)
{
    return (Comment)getHibernateTemplate().get(Comment.class, new
        Integer(id));
}

/**
 * 保存评论
 * @param c 需要保存的评论
 */
public void save(Comment c)
{
    getHibernateTemplate().save(c);
}

/**
 * 删除评论
 * @param c 需要删除的评论
 */
public void delete(Comment c)
{

```



```
        getHibernateTemplate().delete(c);
    }

    /**
     * 删除评论
     * @param id 需要删除的评论的 ID
     */
    public void delete(int id)
    {
        getHibernateTemplate().delete(get(id));
    }

    /**
     * 修改评论
     * @param c 需要修改的评论
     */
    public void update(Comment c)
    {
        getHibernateTemplate().saveOrUpdate(c);
    }

    /**
     * 根据评论名查找评论
     * @param b 评论所对应的 Blog
     * @param pageNo 查找指定页的 Blog
     * @return 查找到的评论集合
     */
    public List findByBlogByPage(Blog b, int pageNo)
    {
        int offset = (pageNo - 1) * pageSize;
```

```
return findByPage("from Comment as c where c.blog = ? order by c.id  
desc", b, offset, pageSize);  
}  
}
```

提供这两个 DAO 组件的实现类后，必须将它们配置在 Spring 容器中，因为 HibernateDaoSupport 已经提供了 setSessionFactory 方法，这个方法用于为 DAO 组件依赖注入 SessionFactory 的引用，这两个 DAO 组件一旦获得了 Hibernate 的 SessionFactory 引用，就具有了数据库操作能力。SessionFactory 通过 Spring 的 IoC 容器注入。

### 19.2.4 配置 DAO 组件

提供了两个 DAO 组件的实现类后，将它们配置在 Spring 容器中，让 Spring 容器为其注入 SessionFactory 的引用，并将 DAO 组件注入到业务逻辑组件中。通过这种依赖注入，可以提供应用各组件之间的良好解耦。下面是 DAO 组件的配置文件的代码片段：

```
<!-- 配置 Blog DAO 组件 -->  
  
<bean id="blogDao" class="org.yeekeu.dao.impl.BlogDaoHibernate">  
  
<!-- 依赖注入 SessionFactory 引用 -->  
  
<property name="sessionFactory" ref="sessionFactory"/>  
  
<!-- 设置每页显示的记录数-->  
  
<property name="pageSize" value="3"/>  
  
</bean>  
  
<!-- 配置 Comment DAO 组件 -->  
  
<bean id="commentDao" class="org.yeekeu.dao.impl.CommentDaoHibernate">  
  
<!-- 依赖注入 SessionFactory 引用 -->  
  
<property name="sessionFactory" ref="sessionFactory"/>  
  
<!-- 设置每页显示的记录数-->  
  
<property name="pageSize" value="3"/>  
  
</bean>
```

因为上面的代码引用了一个 SessionFactory，因此必须在 Spring 容器中定义 SessionFactory 实例。SessionFactory 的配置在前面部分已经给出了，此处不再赘叙。

## 19.3 实现业务逻辑组件

DWR 的魅力在很大程度上得益于与 Spring 的整合，通过 DWR，可让客户端的 JavaScript 代码直接调用 Spring 容器中的 Bean 实例。进行 J2EE 应用开发时，DAO 组件、业务逻辑组件都可按传统 J2EE 应用的开发模式进行实现。业务逻辑组件一样需要接口、实现类，并将其配置在 Spring 容器中。

### 19.3.1 业务逻辑组件的接口

业务逻辑组件里提供业务逻辑方法的定义。每个系统有多少个业务逻辑方法，完全根据业务需要决定。对本系统而言，大致包含如下几个业务逻辑方法：

- u 发表 Blog 文章。
- u 获取指定页的所有 Blog 文章。
- u 发表评论。
- u 根据指定 Blog 文章和页码获取评论列表。
- u 根据指定 Blog 文章 ID 获取文章的详细信息。
- u 获取最新的 Blog 文章。

通过 BlogManager 接口定义上面的系列业务逻辑方法，下面是该接口的代码：

```
public interface BlogManager
{
    /**
     * 创建一篇 Blog 文章
     * @param title Blog 文章的标题
     * @param content Blog 文章的内容
     * @return 新创建 Blog 文章的主键，如果创建失败，则返回-1
     */
    int createBlog(String title, String content)
        throws BlogException;
    /**
```

```
* 创建一个评论
* @param user 新创建评论的用户
* @param email 新创建评论的用户的 E-mail
* @param url 新创建评论的 url
* @param content 新创建评论的内容
* param blogId 新创建评论所对应的主题 ID
* @return 新创建用户的主键
*/
```

```
int createComment(String user, String email, String url,
    String content, int blogId) throws BlogException;
```

```
/**
```

```
* 返回指定页的 Blog 文章
* @param pageNo 指定页面
* @return 指定页面的全部 Blog 文章
*/
```

```
List<BlogBean> getAllBlogByPage(int pageNo)
    throws BlogException;
```

```
/**
```

```
* 返回指定 Blog 文章、指定页所对应的评论
* @param blogId 指定 Blog 文章的 ID
* @param pageNo 指定页面
* @return 指定 Blog 文章、指定页码的所有评论
*/
```

```
List<CommentBean> getCommentsByBlogAndPage(int blogId , int pageNo)
    throws BlogException;
```

```
/**
```

```
* 返回指定 Blog 文章
```

```
* @param blogId 指定 Blog 文章的 ID
* @return 指定 Blog 文章
*/
BlogBean getBlog(int blogId) throws BlogException;

/**
* 返回最新 Blog 文章
* @return 最新的 Blog 文章
*/
BlogBean getNewestBlog() throws BlogException;
}
```

接口仅提供了方法定义，定义了业务逻辑组件应该完成哪些事情，而没有定义业务逻辑的具体实现，方法的具体实现是通过实现类提供的。

### 19.3.2 业务逻辑组件的实现类

业务逻辑组件仅提供业务逻辑方法的实现，不再包含数据库访问逻辑，数据库访问逻辑由 DAO 组件提供。业务逻辑组件的实现依赖于 DAO 组件的方法，因此，必须将 DAO 组件注入业务逻辑组件，这种注入由 Spring IoC 容器提供，业务逻辑组件的实现类则必须提供对应的 setter 方法。下面是业务逻辑组件实现类的代码：

```
public class BlogManagerImpl implements BlogManager
{
    //所依赖的 BlogDao 组件
    private BlogDao blogDao;
    //所依赖的 CommentDao 组件
    private CommentDao commentDao;
    //依赖注入 BlogDao 组件所需的 setter 方法
    public void setBlogDao(BlogDao blogDao)
    {
        this.blogDao = blogDao;
    }
    //依赖注入 CommentDao 组件所需的 setter 方法
    public void setCommentDao(CommentDao commentDao)
    {
        this.commentDao = commentDao;
    }
}
```

```
/**
 * 创建一篇 Blog 文章
 * @param title Blog 文章的标题
 * @param content Blog 文章的内容
 * @return 新创建 Blog 文章的主键，如果创建失败，则返回-1
 */
public int createBlog(String title, String content)
    throws BlogException
{
    try
    {
        Blog b = new Blog();
        b.setTitle(title);
        b.setContent(content);
        b.setAddTime(new Date());
        blogDao.save(b);
        return b.getId();
    }
    catch (Exception e)
    {
        e.printStackTrace();
        throw new BlogException("保存 Blog 文章出错");
    }
}

/**
 * 创建一个评论
 * @param user 新创建评论的用户
 * @param email 新创建评论的用户的 E-mail
 * @param url 新创建评论的 url
 * @param content 新创建评论的内容
 * @param addTime 新创建评论的添加时间
 * @param blogId 新创建评论所对应的主题 ID
 * @return 新创建评论的主键
 */
public int createComment(String user, String email, String url,
    String content, int blogId) throws BlogException
{
    Blog b = blogDao.get(blogId);

    if (b == null)
    {
```

```
        return -1;
    }
    try
    {
        Comment c = new Comment();
        c.setUser(user);
        c.setEmail(email);
        c.setUrl(url);
        c.setContent(content);
        c.setAddTime(new Date());
        c.setBlog(b);
        commentDao.save(c);
        return c.getId();
    }
    catch (Exception e)
    {
        e.printStackTrace();
        throw new BlogException("保存文章评论出错");
    }
}

/**
 * 返回指定页的 Blog 文章标题
 * @param pageNo 指定页面
 * @return 指定页面的全部 Blog 标题
 */
public List<BlogBean> getAllBlogByPage(int pageNo)
    throws BlogException
```

```
{  
    List<BlogBean> result = new ArrayList<BlogBean>();  
    try  
    {  
        List bl = blogDao.findAllByPage(pageNo);  
        for (Object o : bl)  
        {  
            Blog b = (Blog)o;  
            result.add(new BlogBean(b.getId(), b.getTitle(), null, null));  
        }  
        return result;  
    }  
    catch (Exception e)  
    {  
        e.printStackTrace();  
        throw new BlogException("获取文章标题列表出错");  
    }  
}  
  
/**  
 * 返回指定 Blog 文章、指定页的评论  
 * @param blogId 指定 Blog 文章的 ID  
 * @param pageNo 指定页面  
 * @return 指定 Blog 文章、指定页码的所有评论  
 */  
public List<CommentBean> getCommentsByBlogAndPage(int blogId, int pageNo)  
    throws BlogException  
{
```



```
Blog b = blogDao.get(blogId);

if (b == null)

{

    return null;

}

List<CommentBean> result = new ArrayList<CommentBean>();

try

{

    List cl = commentDao.findByBlogByPage(b, pageNo);

    for (Object o : cl)

    {

        Comment c = (Comment)o;

        result.add(new CommentBean(c.getUser(), c.getEmail(),

            c.getUrl(), c.getContent(), c.getAddTime()));

    }

    return result;

}

catch (Exception e)

{

    e.printStackTrace();

    throw new BlogException("获取文章评论列表出错");

}

}

/**

 * 返回指定 Blog 文章

 * @param blogId 指定 Blog 文章的 ID

 * @return 指定 Blog 文章
```

```
*/  
  
public BlogBean getBlog(int blogId)  
    throws BlogException  
{  
    try  
    {  
        Blog b = blogDao.get(blogId);  
        return new BlogBean(b.getId(), b.getTitle(), b.getContent(),  
            b.getAddTime());  
    }  
    catch (Exception e)  
    {  
        e.printStackTrace();  
        throw new BlogException("获取指定 Blog 文章出错");  
    }  
}  
  
/**  
 * 返回最大的 Blog 文章 ID  
 * @return 最新的 Blog 文章  
 */  
  
public BlogBean getNewestBlog() throws BlogException  
{  
    try  
    {  
        Blog b = blogDao.findLastest();  
        if (b != null)  
        {
```

```
        return new BlogBean(b.getId(), b.getTitle(), b.getContent(),  
            b.getAddTime());  
    }  
    return null;  
}  
catch (Exception e)  
{  
    e.printStackTrace();  
    throw new BlogException("获取最新的 Blog 文章出错");  
}  
}  
}
```

提供了业务逻辑组件的实现类后，该业务逻辑组件的实例化过程由 Spring 容器完成，Spring 容器负责创建业务逻辑组件的实例，并管理其依赖关系。为此，必须将该业务逻辑组件部署在 Spring 容器中。

### 19.3.3 配置业务逻辑组件

配置业务逻辑组件应为其注入 DAO 组件，下面是配置业务逻辑组件的代码片段：

```
<!-- 配置业务逻辑组件 -->  
  
<bean id="blogManager" class="org.yeekeu.service.impl.BlogManagerImpl">  
  
<!-- 依赖注入业务逻辑组件所必需的 DAO 组件 -->  
  
<property name="blogDao" ref="blogDao"/>  
  
<property name="commentDao" ref="commentDao"/>  
  
</bean>
```

除此之外，还应为业务逻辑组件配置事务，本应用的事务采用 BeanNameAutoProxyCreator 配置策略，下面是业务逻辑组件的事务配置片段：

```
<!-- 配置事务管理器 -->  
  
<bean id="transactionManager" class="org.springframework.  
orm.hibernate3.HibernateTransactionManager">  
  
<!-- 为事务管理器依赖注入 SessionFactory 实例 -->
```

```
<property name="sessionFactory" ref="sessionFactory"/>
</bean>
<!-- 配置事务拦截器 -->
<bean id="transactionInterceptor" class="org.springframework.
transaction.interceptor.TransactionInterceptor">
<!-- 事务拦截器 bean 需要依赖注入一个事务管理器 -->
<property name="transactionManager" ref="transactionManager"/>
<property name="transactionAttributes">
  <!-- 下面定义事务传播属性-->
  <props>
    <prop key="get*">PROPAGATION_REQUIRED,readonly</prop>
    <prop key="*">PROPAGATION_REQUIRED</prop>
  </props>
</property>
</bean>
<!-- 定义 BeanNameAutoProxyCreator-->
<bean class="org.springframework.aop.framework.
autoproxy.BeanNameAutoProxyCreator">
<!-- 指定对满足哪些 bean name 的 bean 自动生成业务代理 -->
<property name="beanNames">
  <!-- 下面是所有需要自动创建事务代理的 bean-->
  <list>
    <value>blogManager</value>
  </list>
  <!-- 此处可增加其他需要自动创建事务代理的 bean-->
</property>
<!-- 下面定义 BeanNameAutoProxyCreator 所需的事务拦截器-->
<property name="interceptorNames">
  <list>
    <!-- 此处可增加其他新的 Interceptor -->
    <value>transactionInterceptor</value>
  </list>
</property>
</bean>
```

至此，本系统的后台部分基本已经完成，可以通过 DWR 将这些业务逻辑方法暴露给客户端 JavaScript。

## 19.4 在客户端暴露业务逻辑组件

为了在客户端暴露业务逻辑组件，必须先让 Spring 容器随 Web 应用的启动而初始化，还必须在 web.xml 文件中配置 DWR 的核心 Servlet，还必须定义 dwr.xml 文件，将 Spring 容器中的 Bean 暴露成 JavaScript 对象。

### 19.4.1 初始化 Spring 容器

为了让 Spring 容器随 Web 应用的启动而初始化，可在 web.xml 文件中使用 listener。listener 配置的 Listener 比 load-on-startup Servlet 更早初始化。下面是配置 Spring 容器随 Web 应用启动而初始化的代码片段：

```
<!-- 指定 Spring 配置文件的位置 -->
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/classes/applicationContext.xml</param-value>
</context-param>
<!-- 通过 Listener 配置 Spring 容器随 Web 应用的启动而初始化 -->
<listener>      <listener-class>org.springframework.web.context.ContextLoaderListener
</listener-class>
</listener>
```

在 web.xml 文件中增加上面的配置片段，可以让 Spring 容器随 Web 应用的启动而初始化，从而允许 DWR 将 Spring 容器中的 Bean 暴露成 JavaScript 对象。

### 19.4.2 定义 DWR 的核心 Servlet

DWR 的核心 Servlet 是 DWRServlet，该 Servlet 负责拦截来自 JavaScript 的请求，将该 Servlet 配置到某个 URL 下，DWR 暴露的 JavaScript 对象都将暴露在该 URL 下。下面是定义 DWR 的核心 Servlet 的配置片段：

```
<!-- 配置 DWR 的核心 Servlet -->
<servlet>
  <!-- 指定 DWR 核心 Servlet 的名字 -->
  <servlet-name>dwr-invoker</servlet-name>
  <!-- 指定 DWR 核心 Servlet 的实现类 -->
  <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>
  <!-- 指定 DWR 核心 Servlet 处于调试状态 -->
  <init-param>
    <param-name>debug</param-name>
    <param-value>true</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<!-- 指定核心 Servlet 的 URL 映射 -->
<servlet-mapping>
  <servlet-name>dwr-invoker</servlet-name>
  <!-- 指定核心 Servlet 映射的 URL -->
  <url-pattern>/leedwr/*</url-pattern>
</servlet-mapping>
```

在 web.xml 文件中增加上面的配置片段，表示 DWR 所暴露的 JavaScript 对象都位于 leedwr URL 下。在客户端 JavaScript 代码中必须导入位于该 URL 的 JavaScript 代码库。

### 19.4.3 将 Spring 容器中的 Bean 转化成 JavaScript 对象

为了将 Spring 容器中的 Bean 转化成 JavaScript 对象，需要使用 spring 创建器，这个创建器负责将 Spring 容器中的 Bean 创建成一个 JavaScript 对象。

除此之外，配置文件中还定义了需要暴露的业务逻辑方法。这一系列的定义都是通过 dwr.xml 文件完成的。下面是 dwr.xml 文件的代码：

```
<?xml version="1.0" encoding="GBK"?>
<!-- DWR 配置文件的 DTD 信息 -->
<!DOCTYPE dwr PUBLIC "-//GetAhead Limited//DTD Direct Web Remoting 2.0//EN"
"http://getahead.ltd.uk/dwr/dwr20.dtd">
<!-- DWR 配置文件的根元素 -->
<dwr>
<allow>
  <!-- create 元素定义创建一个 JavaScript 对象，该对象的名字为 bm -->
  <create creator="spring" javascript="bm">
    <!-- 定义将 Spring 容器中的 blogManager Bean 创建成 JavaScript 对象 -->
    <param name="beanName" value="blogManager"/>
    <!-- 下面每个 include 元素定义了一个可被暴露的方法 -->
    <include method="createComment"/>
    <include method="createBlog"/>
    <include method="getAllBlogByPage"/>
    <include method="getCommentsByBlogAndPage"/>
    <include method="getBlog"/>
    <include method="getNewestBlog"/>
  </create>
  <!-- 下面定义了两个 JavaBean 可被 DWR 通过 bean 转换器转换成 JavaScript 对象 -->
  <convert converter="bean" match="org.yeeku.vo.CommentBean"/>
  <convert converter="bean" match="org.yeeku.vo.BlogBean"/>
</allow>
</dwr>
```

上面的配置文件配置了 JavaScript 对象 bm，该对象由 DWR 通过 Spring 容器中的 blogManager 创建。除此之外，还定义了 org.yeeku.vo.CommentBean 和 org.yeeku.vo.BlogBean 两个 JavaBean 对象可被转换成 JavaScript 对象。

经过上面的步骤后，可以在 JavaScript 代码中直接调用 bm 对象的方法，也就是调用 blogManager 的方法。因为在配置 DWRServlet 时指定了 DWR 处于调试状态，因此可以看到 DWR 的调试控制台。在浏览器中输入调试控制台的地址，将看到如图 19.3 所示的界面。

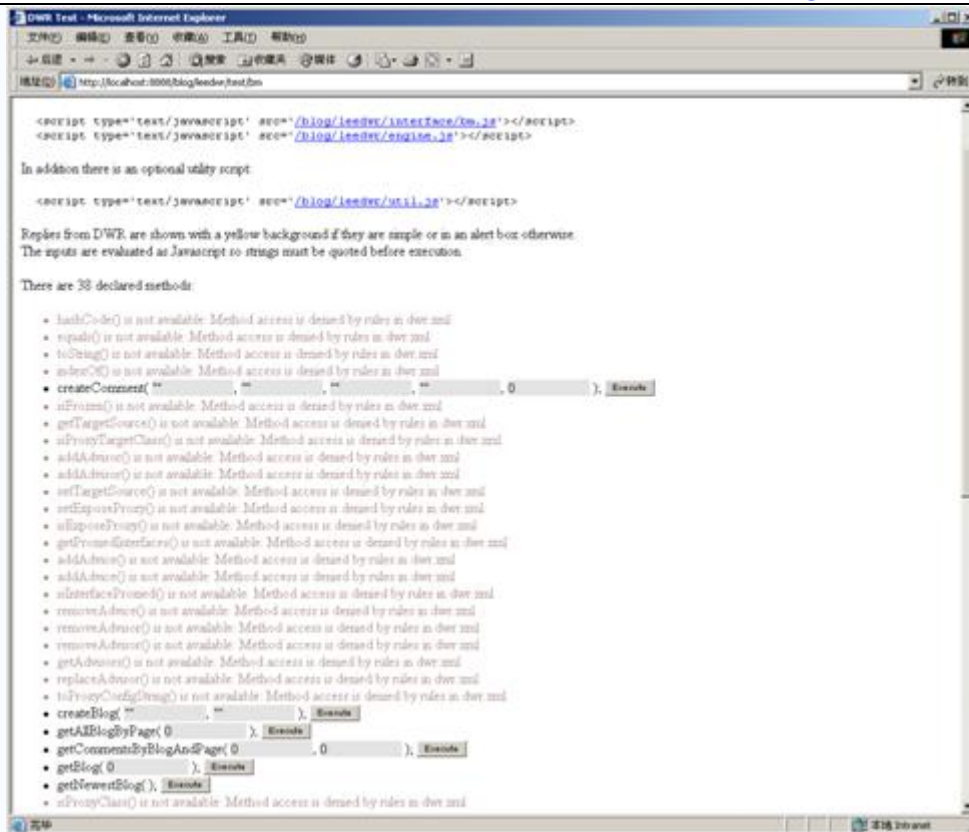


图 19.3 通过 DWR 控制台查看 bm 对象的方法

## 19.5 在客户端调用 JavaScript 对象

正如在图 19.3 中所见到的，bm 对象中包含了 5 个方法。该控制台还提示了如果需要使用 DWR 的 JavaScript 对象，应该在客户端导入哪些 JavaScript 代码库。在图 19.3 所示页面的上部有 3 个超链接，代码如下：

```
u /blog/leedwr/interface/bm.js
u /blog/leedwr/engine.js
u /blog/leedwr/util.js
```

这表明应该在客户端 JavaScript 代码中导入上面的 3 个 JavaScript 函数库。

### 19.5.1 获取 Blog 文章列表

为了在 HTML 页面左边显示 Blog 文章列表，必须调用业务逻辑组件的 getAllBlogByPage 方法，该方法根据当前页码获取 Blog 文章列表。

为了控制 Blog 文章列表的翻页，系统将 Blog 文章列表的当前页设置为一个全局变量。获取当前页的全部 Blog 文章的函数如下：

```
//获取所有的文章

function getBlogList()

{

//对应为调用业务逻辑组件的 getAllBlogByPage 方法

bm.getAllBlogByPage(curBlogPage, bICb);

}
```

上面函数的 `getAllBlogByPage` 函数中有一个 `curBlogPage` 参数，该参数是一个全局变量，用于控制 Blog 文章列表的分页。`bICb` 是获取 Blog 文章列表的回调函数，该回调函数用于将 Blog 文章加载到页面中显示。下面是回调函数的代码：

```
//获取所有的文章列表的回调函数

function bICb(data)

{

//因为该函数还作为翻页函数的回调函数，所以需要判断是否已经是最后一页

//如果当前页码不是第一页，而且当前页没有记录

if ((curBlogPage > 1 ) && (data == null || data == "undefined" || data.length < 1))

{

    alert("已经是最后一页了，无法向后翻页");

    //自动回到上一页

    curBlogPage--;

    return false;

}

//当前页没有记录

else if (data == null || data == "undefined" || data.length < 1)

{

    alert("暂时还没有任何 Blog 文章");

    return false;

}
```



```
else
{
    //用于获取显示 Blog 文章列表的 HTML 元素
    var listElement = $("blogList");
    //清空原有的 Blog 文章列表
    listElement.innerHTML="";
    //遍历 getAllBlogByPage 函数返回的每条记录
    for (var i = 0; i < data.length; i++ )
    {
        //对于每条记录都需创建一个 DIV 元素
        var titleDiv = document.createElement("div");
        //设置显示每条记录的 DIV 元素的 HTML 部分
        titleDiv.innerHTML = '<a href="#" onClick="getBlog(' +
            data[i]['id'] + ');">' + data[i]['title'] + '</a>';
        listElement.appendChild(titleDiv);
    }
}
}
```

上面的回调函数也是翻页 Blog 文章列表的回调函数。

### 19.5.2 控制 Blog 文章列表的翻页

控制翻页比较简单，通过修改页面的全局变量 curBlogPage 即可。curBlogPage 变量减 1 就是向前翻一页，curBlogPage 变量加 1 就是向后翻一页。下面是控制 Blog 文章列表向前翻页的代码：

```
//查看 Blog 文章列表的上一页函数
function blogPre()
{
    //如果当前页已经是第一页，无法翻页
```

```
if (curBlogPage == 1)
{
    alert("已经是第一页，无法向前翻页");
}

//执行翻页，curBlogPage 减 1 后翻页

else
{
    bm.getAllBlogByPage(--curBlogPage, bICb);
}
}
```

向前翻页时先判断当前页是否为 1，如果当前页为 1，则表明无法翻页，否则将 curBlogPage 变量减 1，再次执行请求完成翻页。下面是控制向后翻页的代码：

```
//查看 Blog 文章的下一页函数

function blogNext()
{
    bm.getAllBlogByPage(++curBlogPage, bICb);
}
```

因为本系统并未保存 Blog 文章列表的总页数，因此在翻页时没有判断当前页是否为最后一页，而是直接将当前页加 1 后发送请求。

判断当前页是否为最后一页在回调函数中完成，如果在回调函数中获取的 Blog 文章列表为空且不是第一页，则表明当前页已经是最后一页。如果在最后一页进行翻页，将看到如图 19.4 所示的警告框。



图 19.4 在文章列表最后一页翻页的效果

### 19.5.3 页面加载时的动作

本系统控制当页面加载时，在页面左边加载当前所有 Blog 文章的标题，而右边显示当前最新的 Blog 文章。

这个动作通过页面加载时的初始化函数控制。初始化函数需要完成两个动作：

- u 获取当前最新的 Blog 文章。
- u 获取当前页面的 Blog 文章列表。

其中，第一个动作通过调用业务逻辑组件的 `getNewestBlog` 方法完成，而第二个动作通过上面定义的 `getBlogList` 函数完成。除此之外，初始化函数还设计了 DWR 的全局错误处理函数。

初始化函数的代码如下：

```
function init()
{
    DWREngine.setErrorHandler(errHandler);
    getBlogList();
    bm.getNewestBlog(blogCb);
}
```

上面的代码中定义了 `blogCb` 回调函数，该函数用于根据返回的 Blog 文章实例在页面中显示 Blog 文章的内容。下面是 `blogCb` 回调函数的代码：

```
//获取 Blog 文章的回调函数
function blogCb(data)
{
    if (data != null && data != "undefined")
    {
        //curBlog 是一个全局变量
        curBlog = data['id'];

        //将 Blog 文章中的内容在页面中显示出来
        DWRUtil.setValue('addTime', data['addTime'].toLocaleString());
        DWRUtil.setValue('title', data['title']);
        DWRUtil.setValue('content', data['content']);
    }
}
```

```
}
```

完成上面的定义后，在浏览器中浏览系统的开发页面，将看到如图 19.5 所示的效果。



图 19.5 页面的初始化状态

正如在图 19.5 中所见到的，在页面左边看到的是当前页的 Blog 文章列表，右边看到的是最新 Blog 文章的内容。

#### 19.5.4 查看评论

在图 19.5 中并未看到“新年来了”这篇 Blog 文章的评论，本系统默认关闭文章的评论，如果需要查看评论，则单击“查看”超链接。单击该超链接将触发如下函数：

```
//查看评论的函数

function viewComment()
{
    bm.getCommentsByBlogAndPage(curBlog, curCommentPage, commentCb );
}
```

在上面的函数中调用业务逻辑组件的 `getCommentsByBlogAndPage` 方法，该方法根据当前页码、Blog 文章 ID 获取该文章的所有评论。其中，`commentCb` 是回调函数，该回调函数负责获取所有评论，并在页面中显示评论。与前面类似的是，该回调函数也是控制评论翻页的回调函数，因此有一些控制页码的代码。下面是该回调函数的代码：

```
//查看评论的回调函数

function commentCb(data)
{
    //如果当前的记录为空，并且不是第一页
```

```
if ((curCommentPage > 1 ) && (data == null || data == "undefined" || data.length < 1))
{
    alert("已经是最后一页了，无法向后翻页");
    //系统自动前翻一页
    curCommentPage--;
    return false;
}
//如果没有评论
else if (data == null || data == "undefined" || data.length < 1)
{
    alert("暂时没有评论");
    return false;
}
//如果有评论
else
{
    //获取用于显示评论列表的 HTML 元素
    var comElement = $("comList");
    comElement.innerHTML="";
    //遍历当前页的所有评论
    for (var i = 0 ; i < data.length ; i++ )
    {
        //将评论中的内容在 HTML 页面中显示出来
        var commentDiv = document.createElement("div");
        commentDiv.className="comment";
        commentDiv.innerHTML += data[i]['content'];
        commentDiv.innerHTML += "<br><br>发布者: <strong>
```

```

<a href='mailto:" + data[i]['email'] + "'>" + data[i]['user']
+ "</a> (" + data[i]['url'] + ") </strong>—" + data[i]
['addTime'].toLocaleString());
comElement.appendChild(commentDiv);
$("#viewComment").style.display = "";
}
}
}

```

单击“查看”超链接后，将看到如图 19.6 所示的界面。



图 19.6 查看 Blog 文章评论的界面

### 19.5.5 控制评论的翻页

系统提供了一个全局变量 `curCommentPage`，该变量用于保存当前的评论页。翻页就是通过修改该全局变量的值控制的，将该变量值加 1 就是向后翻页，减 1 就是向前翻页。如果当前页是第一页，则取消翻页动作。翻页的回调函数也是上面定义的 `commentCb` 函数。控制向前翻页的代码如下：

```

//查看评论的上一页函数
function commentPre()
{
//如果当前页是第一页
if (curCommentPage == 1)
{

```

```

        alert("已经是第一页，无法向前翻页");
    }
    else
    {
        //将 curCommentPage 变量减 1，然后请求获取所有评论
        bm.getCommentsByBlogAndPage(curBlog, --curCommentPage, commentCb );
    }
}

```

控制向后翻页的代码如下：

```

//查看评论的下一页函数
function commentNext()
{
    bm.getCommentsByBlogAndPage(curBlog, ++curCommentPage, commentCb );
}

```

与前面类似的是，系统并未保存 Blog 评论的总页数，所以无法知道当前页是否是最后一页，系统只是简单地将 curCommentPage 变量加 1，在回调函数中判断是否已经到达最后一页。如果在评论的最后一页单击“下一页”超链接，将出现如图 19.7 的界面。



图 19.7 在评论的最后一页翻页的效果

### 19.5.6 添加评论

添加评论的输入框默认被隐藏起来了，可以通过单击“发表”超链接来打开添加评论的输入框。单击“发表”超链接触发如下函数：

```

//显示添加评论的表格
function showAdd()
{

```

```
//获取添加评论的 HTML 表格

var addComment = $("addComment");

//如果添加评论的表格被隐藏

if (addComment.style.display == "none")

{

    //显示添加评论的表格

    addComment.style.display = "";

}

else

{

    //隐藏添加评论的表格

    addComment.style.display = "none";

}

}
```

单击“发表”超链接后，将看到如图 19.8 所示的界面。

正如在上面代码中看到的，单击“发表”超链接可让发表评论的输入框在显示和隐藏之间切换。如果在输入框中输入相应内容，然后单击“评论”按钮，那么将触发如下函数：

```
//添加评论

function addComment()

{

    //必须指定对哪篇文章进行评论

    if (curBlog < 0)

    {

        alert("请先选择需要评论的 Blog 文章");

        return false;

    }

    //获取用户名输入框的值
```



```
var user = $("addUser").value;

//获取电子邮件输入框的值

var email = $("addEmail").value;

//获取添加者 Blog 空间的地址

var url = $("addUrl").value;

var content = $("addContent").value;

//要求必须输入用户名和 Blog 评论的内容

if (user == null || user == "" || content == "" || content == null)

{

    alert("请先输入用户名和您要评论的内容");

    return false;

}

//调用业务逻辑组件的方法添加评论

bm.createComment( user, email, url, content, curBlog, addCommentCb);

}
```

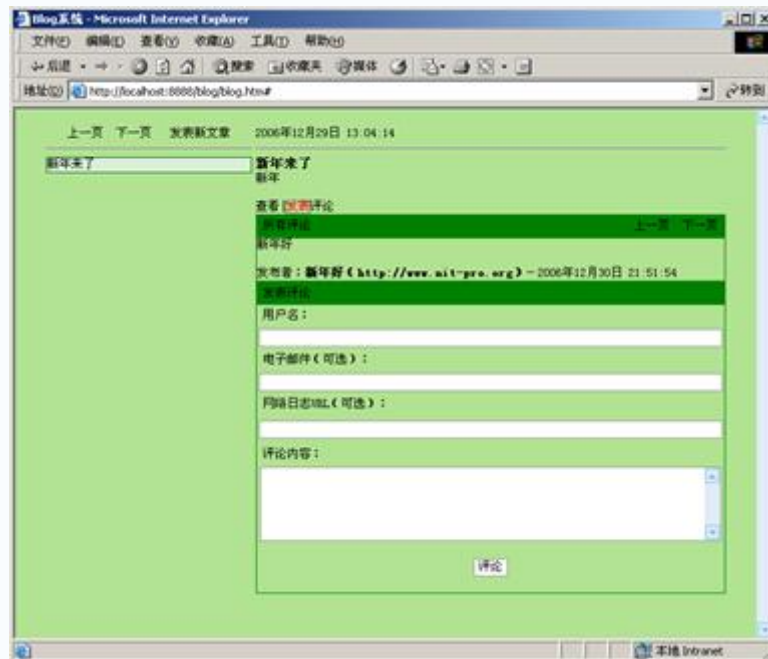


图 19.8 发表评论的输入框

其中，addCommentCb 是添加评论的回调函数，这个回调函数根据返回值判断添加评论是否成功。回调函数的代码如下：

```
//添加评论的回调函数

function addCommentCb(data)

{

//返回值是添加评论的 ID，如果 ID 大于 0，则表明添加成功

if (data > 0)

{

//将各输入框清空

$("#addUser").value = "";

$("#addEmail").value = "";

$("#addUrl").value = "";

$("#addContent").value = "";

//弹出提示框

alert("添加评论成功");

}

else

{

//添加失败

alert("添加评论失败");

}

}
```

如果添加评论成功，将看到如图 19.9 所示的提示框。



图 19.9 添加评论成功

### 19.5.7 查看 Blog 文章内容

在 19.5.1 节的代码中可以看到，在页面左边加载 Blog 文章列表时，每篇 Blog 文章标题都有一个超链接，当单击该文章标题时将触发一个 `getBlog(id)` 函数，该函数用于根据文章 ID 获取文章详细内容。`getBlog(id)` 函数的代码如下：

```
//获取指定 ID 的 Blog 文章

function getBlog(id)
{
    //调用业务逻辑组件的 getBlog 方法，获取 Blog 文章的详细内容

    bm.getBlog(id, blogCb);

    return false;
}
```

上面代码中的 `blogCb` 是回调函数，该回调函数用于在页面中显示系统返回的 Blog 文章内容。该函数的代码在 19.5.3 节中已经给出了，此处不再赘叙。

### 19.5.8 添加新的 Blog 文章

添加新的 Blog 文章通过打开一个新的窗口完成。单击页面中的“发表新文章”链接，将触发如下函数：

```
//打开添加 Blog 的窗口

function showAddBlog()
{
    //打开一个新的 JavaScript 窗口，在该窗口中加载 addBlog.html 页面

    win = window.open("addBlog.html","console","width=400,height=250");

    win.focus();
}
```

单击“发表新文章”链接后，将看到如图 19.10 所示的界面。



图 19.10 发表新的 Blog 文章

图 19.10 所示的界面中包含了一个“发表”按钮，单击该按钮将触发发表新文章的函数，该函数的代码如下：

```
//创建新 Blog 文章

function addBlog()
{
    //获取文章标题
    var blogTitle = $("blogTitle").value;

    //获取文章内容
    var blogContent = $("blogContent").value;

    //文章的标题和内容都不可为空

    if (blogTitle == null || blogTitle == "" || blogContent == null ||
        blogContent == "")
    {
        alert("新增 Blog 文章时，必须指定文章标题，文章内容");
    }
    else
    {
        //调用业务逻辑组件的方法添加新的 Blog 文章

        bm.createBlog(blogTitle, blogContent, addBlogCb);
    }
}
```

在上面的代码中包含了一个 `addBlogCb` 方法，该方法是一个回调函数，用于根据方法返回值判断添加 Blog 文章是否成功。下面是该函数的代码：

```
//创建新 Blog 文章的回调函数

function addBlogCb(data)

{

//返回值是添加 Blog 文章的 ID，如果 ID 不是数字，或者返回值不大于 0

if (typeof data != "number" || data < 1)

{

    alert("添加文章失败");

}

//添加成功

else

{

    //将输入框清空

    $("blogTitle").value = "";

    $("blogContent").value = ""

    alert("添加文章成功");

    window.close();

}

}
```

## 19.6 系统视图

本系统使用两个 HTML 页面作为视图，一个是该系统的主操作页面，另一个是添加 Blog 文章的 HTML 页面。第一个主操作页面的代码如下：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<HTML>

<HEAD>
```

2007-7-27

```
</div>

<br>

<a href="#" onClick="viewComment();">查看</a>|<a href="#" onClick=
    "showAdd();">发表</a>评论

<table id="viewComment" style="display:none;" class="blog" width=
    "100%" border="0" cellpadding="0" cellspacing="0">

    <tr class="tr_title">

        <td align="left">&nbsp;所有评论</td><td align="right"><a href="#"
            onClick="commentPre();">上一页</a>&nbsp;&nbsp;&nbsp;<a href="#"
            onClick="commentNext();">下一页</a>&nbsp; </td>

    </tr>

    <tr>

        <td colspan="2">

            <div id="comList">

                <div class="comment">xxx

            </div>

            <div class="comment">yyy

            </div>

        </td>

    </tr>

</table>

<table id="addComment" style="display:none" class="blog" width="100%"
    border="0" cellpadding="0" cellspacing="0">

    <tr class="tr_title">

        <td>&nbsp;发表评论</td>

    </tr>
```

```
<tr>

  <td height="22">&nbsp;  用户名: </td>

</tr>

<tr>

  <td height="22"><div align="center">

    <input id="addUser" name="addUser" type="text" size="75">

  </div></td>

</tr>

<tr>

  <td height="22">&nbsp;  电子邮件（可选）: </td>

</tr>

<tr>

  <td height="22"><div align="center">

    <input id="addEmail" name="addEmail" type="text" size="75">

  </div></td>

</tr>

<tr>

  <td height="22">&nbsp;  网络日志 URL（可选）: </td>

</tr>

<tr>

  <td height="28"><div align="center">

    <input id="addUrl" name="addUrl" type="text" size="75">

  </div></td>

</tr>

<tr>

  <td height="22">&nbsp;  评论内容: </td>

</tr>
```



```
<tr>

    <td height="40"><div align="center">

        <textarea id="addContent" name="addContent" cols="73" rows="5">

        </textarea>

        </div></td>

</tr>

<tr>

    <td height="50"><div align="center">

        <input type="button" onClick="addComment();" value="评论">

        </div></td>

</tr>

</table>

</td>

</tr>

</table>

<p>&nbsp;</p>

</BODY>

</HTML>
```

另一个页面用于添加新的 Blog 文章，该页面内提供了两个输入框：一个用于输入 Blog 文章标题，另一个用于输入 Blog 文章内容。该页面的代码如下：

```
<%@ page contentType="text/html; charset=gb2312"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"

"http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=gb2312">

<title>发表新文章</title>
```

```
<!-- 导入所需要的 JavaScript 代码库 -->

<link href="css.css" rel="stylesheet" type="text/css">

<script type='text/javascript' src='./leedwr/engine.js'> </script>

<script type='text/javascript' src='./leedwr/interface/bm.js'></script>

<script type='text/javascript' src='./leedwr/util.js'></script>

<script type='text/javascript' src='bm.js'></script>

</head>

<body bgColor=#b2e392>

<table width="400" height="195" border="0" >

<caption>
发表新文章
</caption>

<tr>

<form>

    <td height="25">文章标题: </td>

    <td><input name="blogTitle" type="text" id="blogTitle" size="48"></td>

</tr>

<tr>

    <td height="25" valign="top">文章内容: </td>

    <td><textarea name="blogContent" cols="46" rows="10" id="blogContent">

        </textarea></td>

</tr>

<tr>

    <td height="35" colspan="2" align="center"><input type="button"

        onClick="addBlog();" value="发表">&nbsp;&nbsp;&nbsp;<input type="reset"

        value="重设"></td>

</tr>
```

```
</form>

</table>

</body>

</html>
```

## 19.7 小结

本章通过一个 Blog 系统详细介绍了 Ajax 技术与实际 J2EE 应用的融合。本章使用 DWR 作为 Ajax 引擎，让 JavaScript 直接访问 Spring 容器中的 Bean。从某种程度上讲，DWR 取代了传统的 MVC 框架，通过 DWR 框架，可以让 JavaScript 调用 Spring 容器中 Bean 的方法。

本章既介绍了传统 J2EE 应用的详细设计过程，也介绍了 DWR 的详细应用，还介绍了如何在 JavaScript 中调用 Spring 容器中 Bean 的方法，并通过 DOM 操作直接动态更新 HTML 页面。

下一章将介绍一个更加复杂的 Ajax + J2EE 应用，它是对一个传统 J2EE 应用的改进，增加了 Ajax 引擎，通过 Spring 的 AOP 提供了完善的权限控制。