

《JSP网络开发技术与整合应用》 清华大学出版社

网址：

<http://www.china-pub.com/computers/common/info.asp?id=353947>

<http://www.dearbook.com.cn/book/119931>



本书已经不是简单的JSP书籍了，涉及了现在流行的J2EE框架，和各种开源的技术，很有吸引力！

适合阅读本书的读者

- JSP/Java编程基础的初中级读者
- JSP编程爱好者
- Java开发人员
- Java Web开发人员
- J2EE开发人员
- 高校相关专业师生

目 录

第一部分 JSP技术基础

第一章 JSP 技术简介

- 1.1 认识 HTTP 协议
 - 1.1.1 HTTP 请求介绍
 - 1.1.2 HTTP 响应介绍
- 1.2 客户端 Web 程序设计介绍
 - 1.2.1. CSS (Cascading Style Sheets)
 - 1.2.2. JavaScript
 - 1.2.3. VBScript
 - 1.2.4. 动态 HTML
 - 1.2.5. Java Applet (Java 小应用程序)
 - 1.2.6. 浏览器技术的作用与局限性
- 1.3 JSP 与其他动态网页技术
 - 1.3.1 JSP 的开发背景及发展历史
 - 1.3.2 CGI
 - 1.3.3 ASP
 - 1.3.4 PHP
- 1.4 JSP 的运行原理与优点
- 1.5 JSP 的运行环境介绍
 - 1.5.1 Tomcat 介绍
 - 1.5.2 BEAWebLogic 介绍
 - 1.5.3 IBMWebSphere 介绍
- 1.6 运行第一个 JSP 应用程序
 - 1.6.1 安装 JDK
 - 1.6.2 安装 Tomcat
 - 1.6.3 编写并发布运行 JSP 文件
- 1.7 小结

第二章 JSP 语法

- 2.1 JSP 容器与 JSP 页面生命周期
 - 2.1.1 JSP 容器介绍
 - 2.1.2 JSP 页面生命周期
- 2.2 JSP 注释
- 2.3 JSP 声明
- 2.4 JSP 指令
 - 2.4.1 定义 JSP 文件的全局属性——page 指令
 - 2.4.2 包含一个文件——include 指令
- 2.5 JSP 动作
 - 2.5.1 包含一个静态文件或者其他 Web 组件的结果——include 动作
 - 2.5.2 重定向资源——forward 动作
 - 2.5.3 声明使用的 java 插件——plugin 动作
 - 2.5.4 声明使用一个 Java Bean——useBean 动作
 - 2.5.5 设置 Java Bean 的属性值——setProperty 动作
 - 2.5.6 获取 Java Bean 的属性值——getProperty 动作

2.6 Java 小程序段 (Scriptlet)

2.7 JSP 表达式

2.8 小结

第三章 JSP 内置对象

3.1 JSP 内置对象介绍

3.1 处理客户请求信息——request 对象

3.1.1 request 对象介绍

3.1.2 访问请求参数

3.1.3 在请求作用域中管理属性

3.1.4 获取 Cookie

3.1.5 访问请求报头

3.1.6 访问请求行元素

3.1.7 访问安全信息

3.1.8 访问国际化信息

3.2 控制服务器的响应信息——response 对象

3.2.1 response 对象介绍

3.2.2 输出缓冲

3.2.3 设置响应报头

3.2.4 重定向资源

3.3 管理客户的会话——session 对象

3.3.1 URL 重写

3.3.2 安装 cookie

3.3.3 SSL 会话

3.3.4 会话服务

3.4 Web 应用全局对象——application 对象

3.4.1 访问应用程序初始化参数

3.4.2 管理应用程序环境属性

3.4.3 支持资源的提取

3.4.4 RequestDispatcher 方法

3.4.5 其他实用方法

3.5 当前页面的引用——page 变量

3.5.1 契约服务

3.5.2 访问服务器小程序信息

3.6 Servlet 的配置信息——config 对象

3.7 向客户输出数据——out 对象

3.7.1 管理响应缓冲

3.7.2 写入内容

3.8 异常处理——exception 变量

3.6 小结

第四章 Servlet 技术介绍

4.1 Servlet 简介

4.2 快速体验 Servlet——Servlet 的简单例子

4.2.1 编写 Servlet

4.2.2 编译并发布 Servlet

- 4.3 Servlet 的生命周期
 - 4.3.1 加载 Servlet——Servlet 容器负责
 - 4.3.2 初始化 Servlet——init()方法
 - 4.3.3 请求处理——service()方法
 - 4.3.4 终结 Servlet——destroy()方法
 - 4.3.5 卸载 Servlet
- 4.4 HTTP Servlets
 - 4.4.1 处理 HTTP 请求
 - 4.4.2 处理 HTTP 响应
 - 4.4.3 HTTP Servlets 实例
- 4.5 Web 程序的异常处理
 - 4.5.1 Java 的异常处理
 - 4.5.2 Web 程序中的异常处理
- 4.8 小结

第五章 JavaBeans 与 JSP 技术

- 5.1 JavaBeans 技术简介
- 5.2 JavaBeans 规范
 - 5.2.1 JavaBeans 构造方法
 - 5.2.2 JavaBeans 属性
 - 5.2.3 JavaBeans 事件
 - 5.2.4 JavaBeans 的持续性
- 5.3 JavaBeans 与 EJB 的比较
- 5.4 JavaBeans 在 JSP 技术中的应用
 - 5.4.1 快速体验在 JSP 中使用 JavaBeans
 - 5.4.2 尝试使用 JavaBeans
 - 5.4.3 不共享的 JavaBeans——作用域为页面
 - 5.4.4 在请求域内共享 JavaBeans
 - 5.4.5 在会话域内共享 JavaBeans
 - 5.4.6 在应用程序域内共享 JavaBeans
 - 5.4.7 JavaBeans 在不同的范围内共享时的效果
- 5.5 使用 JavaBeans 实现购物车
- 5.6 小结

第六章 JSP 文件操作

- 6.1 快速体验 JSP 文件操作——使用 JSP 读取文本文件的简单例子
 - 6.1.1 编写进行文件操作的 JavaBean 类
 - 6.1.2 编写 JSP 文件
 - 6.1.3 发布运行 Web 应用
- 6.2 Java 文件操作基础介绍
 - 6.2.1 使用文件类——File 类
 - 6.2.2 使用字节流
 - 6.2.3 使用字符流
- 6.3 文件的上传
 - 6.3.1 组件安装
 - 6.3.2 commons-fileupload 文件上传介绍

- 6.3.3 文件上传的例子
- 6.4 JSP 操作 Excel 文件
 - 6.4.1 安装配置 POI
 - 6.4.2 使用 POI 操作 Excel 工作簿介绍
 - 6.4.3 使用 POI 操作 Excel 工作簿的例子
- 6.5 小结

第七章 JSP Web 应用的会话管理

- 7.1 JSP 的会话管理机制简介
 - 7.1.1 Session 简介
 - 7.1.2 一个利用了会话跟踪的例子
- 7.2 会话跟踪实现方法介绍
 - 7.2.1 使用隐藏表单字段
 - 7.2.2 使用 Cookie 跟踪 Session
 - 7.2.3 使用 URL - Rewriting 跟踪 Session
- 7.3 在 Tomcat 配置 Session
 - 7.3.1 为单个 Web 应用配置 Session
 - 7.3.2 为所有的 Web 应用配置 Session
- 7.4 使用 Tomcat 实现会话持久化
- 7.5 小结

第八章 JSP 2.0 技术

- 8.1 JSP 2.0 的表达式语言
- 8.2 JSP 2.0 新加指令介绍
 - 8.2.1 提供参数——<jsp:param>
 - 8.2.2 提示错误信息——<jsp:fallback>
 - 8.2.3 动态产生 XML 元素——<jsp:element>
 - 8.2.4 定义属性——<jsp:attribute>
 - 8.2.5 定义元素体——<jsp:body>
 - 8.2.6 对标签体内容求值——<jsp:doBody>
 - 8.2.7 调用标签片段——<jsp:invoke>
 - 8.2.8 指定文件类型声明——<jsp:output>
 - 8.2.9 定义标签库的标准元素和命名空间——<jsp:root>
 - 8.2.10 封装模板数据——<jsp:text>
- 8.3 JSP 2.0 简单标签扩展
 - 8.3.1 使用 SimpleTag 接口
 - 8.3.2 使用标签文件
- 8.4 小结

第二部分 JSP技术的应用

第九章 自定义 JSP 标签

- 9.1 自定义 JSP 标签实例
- 9.2 自定义 JSP 标签相关 API
 - 9.2.1 BodyTagSupport 类

- 9.2.2 SimpleTagSupport 类
- 9.2.3 标签处理中处理异常
- 9.3 标签描述文件
 - 9.3.1 taglib 标签库元素
 - 9.3.2 tag 标签元素
 - 9.3.3 attribute 元素
 - 9.3.4 在 Web 应用中使用自定义标签
 - 9.3.5 打包自定义标签库
- 9.4 相关问题
- 9.5 小结

第十章 JSP 标准标签库 (JSTL)

- 10.1 JSTL 简介
- 10.2 JSTL 快速入门——一个简单的例子
 - 10.2.1 下载安装 JSTL 的例子程序
 - 10.2.2 一个新的应用
- 10.3 EL 表达式语言
 - 10.3.1 限制了作用域的变量
 - 10.3.2 隐含对象
 - 10.3.3 存取器
 - 10.3.4 运算符
- 10.4 使用 JSTL 的核心标签库(Core tag library)
 - 10.4.1 通用标签
 - 10.4.2 流程控制标签
 - 10.4.3 循环迭代标签
 - 10.4.4 导入文件和 URL 操作标签
- 10.5 常见问题
- 10.6 小结

第十一章 关系数据库与 JDBC 基础

- 11.1 使用 JSP 和 JDBC 开发第一个数据库应用
 - 11.1.1 准备数据库驱动程序和数据库
 - 11.1.2 编写访问数据库的 JSP 文件
 - 11.1.3 发布访问数据库的 Web 应用
- 11.2 关系数据库与 SQL 语言介绍
 - 11.2.1 表操作 (定义、删除和修改)
 - 11.2.2 查询操作
 - 11.2.3 数据更新 (插入、更新和删除)
- 11.3 JDBC 基础知识
 - 11.3.1 JDBC 简介
 - 11.3.2 JDBC 两层模型和三层模型
 - 11.3.3 指定数据库的 URL 连接串
 - 11.3.4 JDBC API 介绍
- 11.4 通过 JDBC 访问数据库
 - 11.4.1 使用 JDBC-ODBC 桥连接数据库
 - 11.4.2 使用本地协议纯 Java 驱动程序连接数据库

- 11.4.3 使用 PreparedStatement 接口发送 SQL 语句——数据录入例子
- 11.4.4 使用 JDBC 的数据库事务操作——银行转帐
- 11.5 JSP 与数据库连接池
- 11.6 JSP 与数据源 (DataSource)
 - 11.6.1 数据源 (DataSource) 简介
 - 11.6.2 配置使用数据源
- 11.7 小结

第十二章 JSP 与 Java Mail Web 应用

- 12.1 Java Mail 的简单实例
 - 12.1.1 准备邮件服务器
 - 12.1.2 编写程序
 - 12.1.3 运行程序并查看效果
- 12.2 Java Mail API 简介
- 12.3 使用 Java Mail API 发送带附件的邮件应用
 - 12.3.1 编写程序
 - 12.3.2 运行程序并查看结果
- 12.4 创建可发送附件的 Java Mail Web 应用
 - 12.4.1 Java Mail Web 应用的程序分析
 - 12.4.2 邮件帐户管理
 - 12.4.3 包含文件
 - 12.4.4 登录邮件服务器
 - 12.4.5 管理邮件夹中邮件
 - 12.4.6 查看邮件
 - 12.4.7 写新邮件
 - 12.4.8 退出系统
 - 12.4.9 发布 Java Mail Web 应用
- 12.5 小结

第十三章 XML 在 JSP 中的应用

- 13.1 XML 与 JSP
 - 13.1.1 什么是 XML
 - 13.1.2 XML 的特点
 - 13.1.3 XML 与 JSP 的工具介绍
- 13.2 使用 DOM 解析接口操作 XML 文件
 - 13.2.1 DOM API
 - 13.2.2 使用 DOM 读写 XML 文件例子
- 13.3 使用 JDOM 操作 XML 文件
 - 13.3.1 JDOM 的安装与简介
 - 13.3.2 使用 JDOM 读写 XML 文件
- 13.4 使用 SAX 操作 XML 文件
 - 13.4.1 SAX API
 - 13.4.2 使用 SAX 读写 XML 文件
- 13.5 使用 XSLT 给 XML 定制样式
 - 13.5.1 建立 XML 文件
 - 13.5.2 建立 XSLT 文件

- 13.5.3 建立源数据的对象
- 13.5.4 建立结果数据的对象
- 13.5.5 转换数据

13.6 小结

第十四章 使用 Servlet 过滤器和监听器

- 14.1 Servlet 过滤器简介
- 14.3 实现一个 Servlet 过滤器
 - 14.3.1 编写实现类的程序
 - 14.3.2 配置发布 Servlet 过滤器
- 14.4 ServletRequest 和 ServletResponse 的包装类
- 14.5 用 Servlet 过滤器过滤文本信息
 - 14.5.1 输出流管理类
 - 14.5.2 编写响应包装类
 - 14.5.3 编写 Servlet 过滤器
 - 14.5.4 编写 JSP 和 Servlet 文件
 - 14.5.5 发布运行这个 Web 应用
- 14.6 Servlet 监听器简介
 - 14.6.1 监听服务器 ServletContext 对象
 - 14.6.2 监听客户会话
 - 14.6.3 监听客户请求
- 14.7 网站计数器实例——使用 Servlet 监听器
- 14.8 小结

第十五章 JSP Web 应用的安全性

- 15.1 JSP/Servlet 容器认证
 - 15.1.1 使用基本认证 (BASIC)
 - 15.1.2 使用摘要认证 (DIGEST)
 - 15.1.3 使用基于表单的认证 (FORM)
- 15.2 为 Web 应用配置使用 SSL
 - 15.2.1 SSL 简介
 - 15.2.2 在 Tomcat 中为 Web 应用配置使用 SSL
- 15.3 小结

第三部分 整合应用

第十六章 Tomcat 容器的 JSP 特色应用

- 16.1 使用 Tomcat 阀
 - 16.1.1 客户访问日志阀
 - 16.1.2 远程地址过滤器
 - 16.1.3 远程主机过滤器
 - 16.1.4 客户请求记录器
 - 16.1.5 单点登录阀
- 16.2 使用基于 JNDI 的应用程序开发 (介绍 tomcat 的 JNDI 资源)
 - 16.2.1 使用通用 JavaBean 资源

16.2.2 使用 JavaMail Sessions 资源

16.2.3 使用 JDBC Data Sources

16.3 小结

第十七章 在 JSP 使用 Hibernate 实现数据持久化

17.1 快速体验 JSP 结合 Hibernate——JSP 和 Hibernate 结合的简单例子

17.1.1 Hibernate 简介

17.1.2 配置 Hibernate 环境

17.1.3 准备数据库和数据库连接池

17.1.4 编写持久化类

17.1.5 编写 Hibernate 配置文件

17.1.6 编写映射文件

17.1.7 编写 JSP 应用文件

17.1.8 编译并发布 Web 应用

17.2 Hibernate 技术介绍

17.2.1 映射定义

17.2.2 Hibernate 的类型

17.2.3 Hibernate 事务

17.3 Hibernate 配置

17.3.1 可编程的配置方式

17.3.2 XML 配置文件方式

17.4 小结

第十八章 JSP Web 应用的设计概述

18.1 可维护性与可扩展性设计

18.1.1 可维护性

18.1.2 可扩展性

18.2 JSP Web 应用的设计

18.2.1 以页面为中心的设计 (Model 1)

18.2.2 MVC 设计模式 (Model 2)

18.3 Web 应用的架构框架

18.3.1 Struts——最流行的 MVC 框架

18.3.2 Webwork2——基于 Xwork 的 MVC 框架

18.3.3 Spring——以控制倒置原则为基础的 MVC 框架

18.3.4 JavaServer Faces——Sun 力推的 MVC 框架

18.4 Web 应用的测试

18.4.1 JUnit——优秀的单元测试工具

18.4.2 Cactus——基于 JUnit 框架的服务器端测试工具

18.5 日志

18.5.1 Log4j——最流行的日志工具

18.5.2 Jakarta Commons Logging——Jakarta 的优秀日志工具

18.6 小结

第十九章 MVC 模式实现—Struts

19.1 快速体验 Struts——使用 Struts 框架的简单应用实例

19.1.1 建立 Struts 开发环境

- 19.1.2 实例介绍
- 19.1.3 创建 Model 组件
- 19.1.4 创建 View 组件
- 19.1.5 编写配置文件
- 19.1.6 发布运行 Web 应用
- 19.2 Struts 框架的体系结构与运行原理
 - 19.2.1 从 struts 的组件来看 Struts 的工作原理
 - 19.2.2 Struts 配置文件 struts-config.xml
- 19.3 Struts 组件介绍
 - 19.3.1 使用 ActionServlet 控制器类分发请求
 - 19.3.2 使用 Action 组件分离请求和业务
 - 19.3.3 使用 Action Mapping 映射 Action
 - 19.3.4 ActionForm Bean 获取表单数据
- 19.4 使用 Struts 开发用户登录系统——Struts 实例
 - 19.4.1 开发模型组件
 - 19.4.2 开发视图组件
 - 19.4.3 开发辅助类
 - 19.4.4 创建配置文件
 - 19.4.5 发布 Web 应用
 - 19.4.6 Web 应用分析
- 19.5 小结

第二十章 MVC 模式实现—WebWork2

- 20.1 快速体验 WebWork2——使用 WebWork2 框架的简单应用实例
 - 20.1.1 WebWork2 简介
 - 20.1.2 建立 WebWork2 开发环境
 - 20.1.3 实例介绍
 - 20.1.4 开发构成类和 JSP 文件
 - 20.1.5 编译发布 Web 应用
- 20.2 Webwork2 组件介绍
 - 20.2.1 使用 Action 组件响应客户请求
 - 20.2.2 使用 ActionContext 获取用户请求信息
 - 20.2.3 使用 ServletDispatcher 分发客户请求
 - 20.2.4 使用 Interceptor(拦截器)框架
 - 20.2.5 使用验证框架验证用户输入
 - 20.2.6 配置 XWork
- 20.3 使用 Webwork2 开发产品录入系统——Webwork2 实例
 - 20.3.1 创建 Action 组件
 - 20.3.2 创建视图组件
 - 20.3.3 验证框架
 - 20.3.4 编写配置文件
 - 20.3.5 运行基于 WebWork2 的 Web 应用
- 20.4 小结

第二十一章 JavaServer Faces

- 21.1 快速体验 JavaServer Faces——使用 JavaServer Faces 开发的简单例子

- 21.1.1 JavaServer Faces 技术介绍
- 21.1.2 建立 JavaServer Faces 开发环境
- 21.1.3 编写相关类和文件
- 21.1.4 编译发布 Web 应用
- 21.2 JavaServer Faces 的生命周期
- 21.3 JSF 定义的组件
 - 21.3.1 管理 Bean
 - 21.3.2 验证器和验证器
- 21.4 Java Server Faces 导航规则
- 21.5 小结

第二十二章 JSP Web 应用测试

- 22.1 JUnit 简介安装配置
 - 22.1.1 JUnit 简介
 - 22.1.2 安装配置
- 22.2 JUnit 中常用的接口和类
- 22.3 使用 JUnit 测试的例子
- 22.3 使用 Cactus 测试 Web 应用
 - 22.4.1 Cactus 介绍
 - 22.4.2 使用 Cactus 测试 Web 应用
- 22.5 使用 StrutsTestCase 测试 Struts 应用程序
 - 22.5.1 StrutsTestCase 介绍
 - 22.5.2 使用 StrutsTestCase 测试 Struts 应用程序的例子
- 22.6 小结

第二十三章 使用 Log4J 进行 Web 应用的日志管理与程序调试

- 23.1 快速体验 Log4J——使用 Log4J 的简单例子
 - 23.1.1 Log4J 简介
 - 23.1.2 建立 Log4J 环境
 - 23.1.4 使用 Log4J 的 Web 应用
- 23.2 Log4J 关键类和接口介绍
 - 23.2.1 Logger：日志写出器
 - 23.2.2 Level：日志级别
 - 23.2.3 Appender 接口
 - 23.2.4 Layout 类：日志输出格式
 - 23.2.5 Log4j 的 Logger 继承性
- 23.3 配置 Log4J
 - 23.3.1 使用 Java properties 配置
 - 23.3.2 配置 log4j.xml
 - 23.3.3 Log4j 配置实现过程
- 23.4 Web 应用中使用 Log4J 的例子
- 23.5 小结

第二十四章 使用 XDoclet 简化 JSP 的 Web 开发

- 24.1 快速体验 XDoclet——使用 XDoclet 的简单例子
 - 24.1.1 XDoclet 介绍

- 24.1.2 安装配置 XDoclet
- 24.1.3 Java 源程序和添加注释
- 24.1.4 书写 build.xml 文件
- 24.1.5 运行实例
- 24.2 XDoclet 生成配置文件过程介绍
- 24.3 使用 XDoclet 进行 Web 开发
 - 24.3.1 开发 Struts
 - 24.3.2 开发 Servlet 过滤器
 - 24.3.3 开发自定义标签
 - 24.3.4 运行例子
- 24.4 小结

第二十五章 使用 Ant 管理 JSP Web 应用

- 25.1 快速体验 Ant——使用 Ant 的简单例子
 - 25.1.1 Ant 简介
 - 25.1.2 安装配置 Ant
 - 25.1.3 编写应用类文件
 - 25.1.4 编写 build.xml 文件
 - 25.1.5 使用 Ant 运行
- 25.2 编写 build.xml 文件
 - 25.2.1 project 元素介绍
 - 25.2.2 Target 元素介绍
 - 25.2.3 Task 元素介绍
 - 25.2.4 Property 元素介绍
 - 25.2.5 常用 Ant Task 介绍
 - 25.2.6 build.xml 实例分析
- 25.3 用 Ant 发布复杂 Web 应用
 - 25.3.1 build.xml 文件
 - 25.3.2 init 目标
 - 25.3.3 compile 目标
 - 25.3.4 copyjar 目标
 - 25.3.5 deploy 目标
 - 25.3.6 clean 和 help 目标
 - 25.3.7 使用 Ant 发布 Web 应用
- 25.4 小结

第二十六章 使用 Eclipse 开发 JSP

- 26.1 Eclipse 简介与 Eclipse 安装
 - 26.1.1 Eclipse 简介
 - 26.1.2 安装 Eclipse 和多国语言包
- 26.2 使用 Eclipse 的 Lomboz 插件开发 JSP
 - 26.2.1 Lomboz 插件介绍
 - 26.2.2 安装配置 Lomboz 插件
 - 26.2.3 安装 Tomcat 插件
 - 26.2.4 使用 Lomboz 插件开发 JSP 程序
 - 26.2.5 使用 Lomboz 插件调试 JSP 程序

26.4 小结

第二十七章 使用 JBuilder 开发 JSP

27.1 快速体验 JBuilder

27.1.1 安装 JBuilder

27.1.2 JBuilder 开发环境简介

27.2 使用 JBuilder 开发 JSP (实例)

27.2.1 建立工程及相关准备工作

27.2.2 设置项目相关属性

27.2.3 新建 Web Module

27.2.4 开发 JSP

27.3 使用 JBuilder 开发 Servlet (实例)

27.5 使用 JBuilder 进行程序调试 (实例)

27.6 小结

第二十八章 JSP 结合 EJB 开发 J2EE 应用

28.1 快速体验 EJB——JSP 结合 EJB 开发的简单例子

28.1.1 J2EE 简介 (J2EE 体系结构)

28.1.2 JBoss 入门

28.1.3 一个简单的 J2EE 应用介绍

28.1.4 开发 EJB 组件

28.1.5 在 Web 应用中访问 EJB 组件

28.1.6 在 JBoss 中发布运行 J2EE 应用

28.2 EJB 技术介绍

28.2.1 EJB 组件介绍

28.2.2 实体 EJB

28.2.3 会话 EJB

28.2.4 Web 应用中访问 EJB 组件过程介绍

28.3 网上书店——J2EE 应用实例

28.3.1 创建 EJB 组件

28.3.2 在 Web 应用中访问 EJB 组件

28.3.3 发布 J2EE 应用

28.4 小结

第二十九章 JSP 作为客户访问 CORBA 服务

29.1 快速体验 CORBA——使用 JSP 访问 CORBA 的简单例子

29.1.1 CORBA 简介

29.1.2 实现 CORBA 服务的小例子

29.2 CORBA 技术构成

29.2.1 对象请求代理 (ORB, Object Request Broker)

29.2.2 接口定义语言 (IDL, Interface Definition Language)

29.2.3 接口仓库 (IR, Interface Repository)

29.2.4 对象适配器 (OA, Object Adapter)

29.2.5 动态调用接口和静态调用接口 (DII, Dynamic Invocation Interface)

29.2.6 GIOP 和 IIOP

29.3 股票查询服务——CORBA 服务实例

- 29.3.1 使用 IDL 语言定义 IDL 接口并编译映射到 Java 程序
- 29.3.2 实现 IDL 接口
- 29.3.3 编写服务端实现
- 29.3.4 准备数据库
- 29.3.5 实现 Corba 客户端
- 29.3.6 配置 Corba 服务的 Servlet 客户端
- 29.3.7 测试 Corba 服务与 Web 应用

29.4 小结

第三十章 Velocity 模板语言

30.1 Velocity 入门

- 30.1.1 简介
- 30.1.2 安装 Velocity
- 30.1.3 开始使用 Velocity

30.2 注释

30.3 引用

- 30.3.1 变量引用
- 30.3.2 属性引用
- 30.3.3 方法引用
- 30.3.4 正式引用符 (Formal Reference Notation)
- 30.3.5 安静应用符(Quiet Reference Notation)

30.4 指令

- 30.4.1 #set 指令——变量赋值
- 30.4.2 #if/#elseif/#else 指令——条件语句
- 30.4.3 #foreach 指令——循环语句
- 30.4.4 #include 指令——包含指令
- 30.4.5 #parse 指令——导入 VM 文件
- 30.4.6 #stop 指令——停止模板执行
- 30.4.7 #macro 指令——定义 VTL 模板

30.5 VTL 的其他特征

- 30.5.1 关系运算和逻辑运算
- 30.5.2 转义 VTL 指令
- 30.5.3 数学运算
- 30.5.4 范围操作

30.4 小结

第四部分 JSP应用实例分析

第三十一章 JSP 范例：在线图书订购系统

31.1 BookStore 实例介绍

- 31.1.1 BookStore 的结构描述
- 31.1.2 BookStore 的功能介绍

31.2 开发并发布 JavaBean

- 31.2.1 图书基本信息 Bean - Book

- 31.2.2 购物车 Bean - BookShopCart
 - 31.2.3 购买项目 Bean - ShopBookItem
- 31.3 数据库操作
- 31.4 JSP 页面
 - 31.4.1 图书列表页面
 - 31.4.2 添加图书确认页面
 - 31.4.3 中转页面
 - 31.4.3 查看购物车
- 31.4 发布运行 BookStore 应用
- 31.12 小结

第二十五章 使用Ant管理JSP Web应用

Ant是用于构建、部署Java程序的一个工具,使用Ant可以明显加快Java应用开发的进程,对于Web应用而言也不例外。在本章中简单介绍一下Ant,并主要介绍如何编写Ant的build文件。

25.1 快速体验Ant——使用Ant的简单例子

25.1.1 Ant简介

Ant是一个用于简单或复杂Java工程的自动化构建、部署工具,它对于那些具有分布式开发团队或者通过频繁的构建来进行不间断集成的公司尤其有用。那些建立传统全Java应用程序以及那些使用HTML、JSP和Java servlets创建Web应用程序的公司来说, Ant极具价值。无论Java开发者使用什么操作系统、集成开发环境或者构建环境, Ant都可以将这些工程集合在一起,用于那些重要的构建。Ant也能够自动化并且同步文档部署。

在构建和部署Java应用程序的时候, Ant处理着大量有用的任务。最基本的包括添加和移除目录、使用FTP拷贝和下载文件、创建JAR和ZIP文件以及创建文档。更高级的特性包括用源代码控制系统诸如CVS或者SourceSafe来检查源代码、执行SQL查询或脚本、将XML文件转换为HTML,以及为远程方法调用生成stub(存根)文件。

25.1.2 安装配置Ant

Ant的稳定版可以在Ant官方网站<http://ant.apache.org/>上免费下载得到。从网站<http://cvs.apache.org/builds/ant/nightly/>可以下载最新版本的Ant,不过最新版本可能会有一些Bug,所以不能保证程序运行的正确性。本书的随书光盘中提供Ant1.6.5版本。把文件apache-ant-1.6.5-bin.zip文件解压到本地硬盘,然后按照下面的步骤设置一下就可以了。在操作系统的环境变量中设置:

- 新建名字为ANT_HOME的变量,并将其指示为ant的安装目录
- 确保JAVA_HOME指向JDK的安装目录
- 在PATH变量中新添加%ANT_HOME%/bin,用于从命令行可以直接运行ant程序。

上述设置完成后,在DOS环境下输入ant,如果命令行所指示的目录下有build.xml文件,则会提示ant运行成功或出现错误的信息,如果没有该文件,也会出现一定的关于ant运行失败的提示信息,则表明ant的安装和配置都成功了。如图25.1。

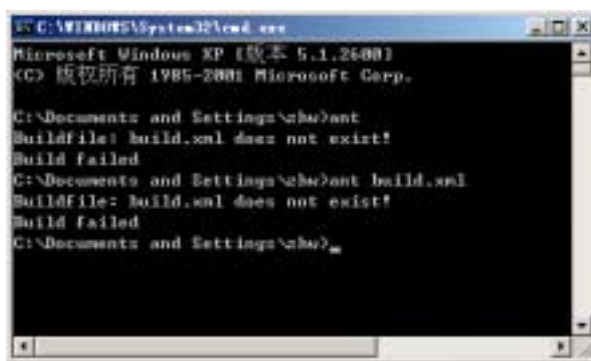


图25.1 Ant安装成功测试

Ant安装完成后,可以看到如下的目录布局。


```

ant
+--- bin    //包括各种二进制启动命令
|
+--- lib    // 包括ant运行需要的包文件
|
+--- docs  //包括这种文档
|
|      +--- ant2    // ant2运行需要的条件的一个简单描述
|      |
|      +--- images  // html文档中的各种图片
|      |
|      +--- manual  // Ant 文档
|
+--- etc // 包括具有各种功能的xml文件

```

25.1.3 编写应用类文件

在这里编写一个JavaBean文件和JSP文件，然后使用Ant编译JavaBean，并把编译后的字节码文件拷贝到某个Web应用的合适目录下，使用的JavaBean文件代码如下：

```

package cn.ac.ict;

import java.io.Serializable;
import java.util.Date ;

public class Product implements Serializable{
//JavaBean的属性
    private String pname;
    private String pcomp;
    private Date pmadeyear;
    private float price;
    private int amount;

    public Product(){

    }
//JavaBean属性的获取和设置方法
    public String getPname(){
        return pname;
    }
    public String getPcomp(){
        return pcomp;
    }

    public Date getPmadeyear(){
        return pmadeyear;
    }
    public float getPrice(){
        return price;
    }
}

```

```

    public int getAmount(){
        return amount;
    }

    public void setPname(String productname){
        pname = productname;
    }
    public void setPcomp(String productcomp){
        pcomp = productcomp;
    }

    public void setPmadeyear(Date madeyear){
        pmadeyear = madeyear;
    }
    public void setPrice(float price){
        this.price = price;
    }

    public void setAmount(int pamount){
        amount = pamount;
    }

}

```

使用的JSP文件的代码如下：

```

<%@ page language="java" pageEncoding="GB2312" %>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
    <head>
        <title>添加商品JSP页面</title>
    </head>
    <body bgcolor="#FFFFFF">
        <jsp:useBean id="product" scope="request" class="cn.ac.ict.Product" />
        <jsp:setProperty name="product" property="*" />

        <form action="addproduct.jsp" method="get">
            <table width="580" border="0" cellspacing="0" cellpadding="0"
align="center">
                <thead>添加新的商品</thead>
                <tr>
                    <td>商品名称</td>
                    <td><input name="pname" type="text"></td>
                </tr>
                <tr>
                    <td>生产商家</td>
                    <td><input name="pcomp" type="text"></td>
                </tr>
                <tr>
                    <td>生产日期</td>
                    <td><input name="pmadeyear" type="text"></td>
                </tr>
            </table>
        </form>
    </body>
</html>

```

```
|  |  |
| --- | --- |
| 价格 |  |
| 数量 |  |
|  | |

```

25.1.4 编写build.xml文件

准备好了需要使用的Java文件和JSP文件后，就可以编写运行Ant需要使用的build.xml文件了，下面是build.xml文件的完整代码：

```

<!-- 定义的Project名称、默认执行的target是dist -->
<project name="SimpleProject" default="dist" basedir=".">
  <description>
    simple example build file
  </description>
  <!-- 定义全局属性 -->
  <property name="src" location="src"/>
  <property name="dist" location="dist"/>

  <!-- 名为init的target，它完成建立临时目录的工作，并把需要使用的源文件拷贝到合适位置-->
  <target name="init">
    <mkdir dir="${dist}"/>
    <mkdir dir="${dist}/WEB-INF"/>
    <mkdir dir="${dist}/WEB-INF/classes"/>
    <copy todir="${dist}">
      <fileset dir="${src}">
        <include name="*.jsp" />
        <exclude name="build.xml" />
      </fileset>
    </copy>
    <copy todir="${dist}/WEB-INF">
      <fileset dir="${src}">
        <include name="web.xml" />
        <exclude name="build.xml" />
      </fileset>
    </copy>
  </target>

```

```

<!-- 名为compile的target，它编译JavaBean文件 -->
<target name="compile" depends="init"
        description="compile the source " >
    <javac srcdir="${src}" destdir="${dist}/WEB-INF/classes"/>
</target>

<!-- 名为dist的target，它将Web应用打包 -->
<target name="dist" depends="compile"
        description="generate the distribution" >
    <jar destfile="${basedir}/FirstAnt.war" basedir="${dist}"/>
</target>

<!-- 删除编译和发布时使用的临时目录 -->
<target name="clean"
        description="clean up" >
    <delete dir="${dist}"/>
</target>
</project>

```

25.1.5 使用Ant运行

所有的文件都准备好后，这个Web应用的目录结构如图25.2。在该目录下运行ant命令，可以看到命令行提示效果如图25.3。



图25.2 Web应用的目录结构

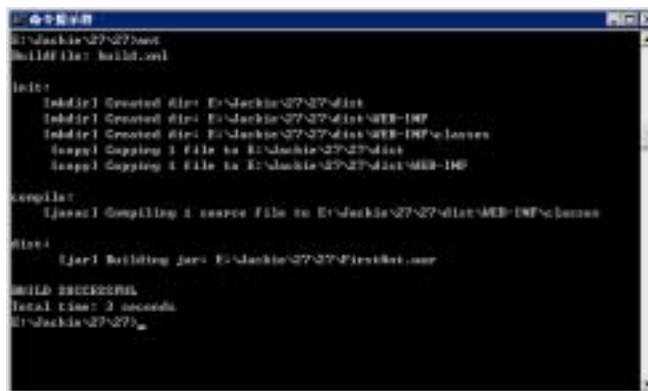


图25.3 命令行提示效果

这时在该目录下会生成一个FirstAnt.war文件，把这个文件放到<TOMCAT_HOME>/webapps目录下可以发布这个Web应用，启动Tomcat，在浏览器地址栏中输入如下地址：<http://192.168.1.26:8080/FirstAnt/addproduct.jsp>，可以看到页面效果如图25.4。



图25.4 应用发布成功效果

25.2 编写build.xml文件

用Ant编译或运行比较大的工程是非常方便的，每个工程都有一个包含与这个工程以及需要Ant执行的任务信息的文件，这个文件在ant中默认是build.xml，也可以在ant执行时指定使用的构建文件（本书默认使用build.xml）。ant会根据build.xml提供的信息对工程进行操作，ant从以前简单的编译工程已经发展到具有数据库等很多其他功能的强大工具。下面简单介绍build.xml的构成元素和编写方法。

25.2.1 project元素介绍

每个build.xml文件都包含一个project元素和至少一个（默认的）target元素。一个project元素包含3个属性，name、default和basedir，分别介绍如下。

表25.1 project元素的3个属性及描述

属性	描述	是否必须
name	Project的名字	否
default	不指定target时默认执行的target	否
basedir	所有路径计算的基目录	否

注意：basedir 属性可能被 basedir 属性覆盖，这时在 project 元素中它必须被忽略，如果两个都没有指定，则使用 build.xml 的父目录作为所有路径计算的基目录。

下面是使用project元素时的基本形式。

```
<project name="typical" default="all" basedir=".">
.....
</project>
```

在这个例子中定义了project的名字是typical，默认的Target是all，基目录就是当前目录。

25.2.2 Target元素介绍

一个项目可以定义一个或多个target，一个target是一系列想要执行的任务的集合。执行Ant时，可以选择执行那个target。当没有给定target时，使用project的default属性所确定的target。

一个target可以依赖于其他的target。例如，可能会有一个target用于编译程序，一个target用于生成可执行文件。在生成可执行文件之前必须先编译通过，所以生成可执行文件的target依赖于编译target。Ant会自动处理这种依赖关系。

然而，Ant的depends属性只指定了target应该被执行的顺序，如果被依赖的target无法运行，这种depends对于指定了依赖关系的target就没有影响。

Ant会依照depends属性中target出现的顺序（从左到右）依次执行每个target。然而，要记住的是只要某个target依赖于一个target，后者就会被先执行。

```
<target name="A" />
<target name="B" depends="A" />
<target name="C" depends="B" />
<target name="D" depends="C,B,A" />
```

假定要执行target D。从它的依赖属性来看，可能认为先执行C，然后B，最后A被执行。不过，由于C依赖于B，B依赖于A，所以先执行A，然后B，然后C，最后D被执行。

Targets元素支持的属性包括name、depends、if、unless和description，如表25.2。

表25.2 Target元素的属性描述

属性名	描述	是否必须
name	target的名字	是
depends	依赖表，用逗号分隔的target的名字列表，	否
if	执行target所需要设定的属性名	否
unless	执行target需要清除设定的属性名	否
description	关于target功能的简短描述	否

如果（或如果不）某些属性被设定，才执行某个target。这样，允许根据系统的状态（java version、OS、命令行属性定义等等）来更好地控制build的过程。要想让一个target这样做，应该在target元素中，加入if（或unless）属性，带上target因该有所判断的属性。例如：

```
<target name="build-module-A" if="module-A-present" />
<target name="build-own-fake-module-A" unless="module-A-present" />
```

如果没有if或unless属性，target总会被执行。

25.2.3 Task元素介绍

一个task是一段可执行的代码。一个task可以有多个属性。属性只可能包含对property的引用。这些引用会在task执行前被解析。

下面是Task的一般构造形式：

```
<name attribute1="value1" attribute2="value2" ... />
```

其中name是task的名字，attributeN是属性名，valueN是属性值。

Ant有一套内置的task，以及一些可选task，但开发者也可以编写自己的task。所有的task都有一个task名字属性（Ant用属性值来产生日志信息）。

一个Task除了有一个名字外，还可以给task赋一个id属性：

```
<taskname id="taskID" ... />
```

这里taskname是task的名字，而taskID是这个task的唯一标识符。通过这个标识符，可以在脚本中引用相应的task。

25.2.4 Property元素介绍

一个project可以有很多的properties。可以在buildfile中用property task来设定，或在Ant之外设定。一个property有一个名字和一个值。property可用于task的属性值。这是通过将属性名放在"\${"和"}"之间并放在属性值的位置来实现的。例如如果有一个property builddir的值是"build"，这个property就可用于属性值：\${build}/classes。这个值就可被解析为build/classes。

在Ant中可以访问所有的系统属性（不需要重新定义），例如\${os.name}被解析成操作

系统的名字。其他的系统属性可以参考Java文档中的System.getProperties部分，除了这些系统属性外，Ant本身还提供了一些内置属性：

- basedir：project基目录的绝对路径(与<project>的basedir属性一样)。
- ant.file：buildfile的绝对路径。
- ant.version：Ant的版本。
- ant.project：name 当前执行的project的名字；由<project>的name属性设定。
- ant.java.version：Ant检测到的JVM的版本；目前的值有"1.1","1.2","1.3","1.4"和"1.5"。

25.2.5 常用Ant Task介绍

上面介绍了project元素、Target元素、Task元素和Property元素，它们在编写build.xml文件过程中基本都会用到，但是仅仅知道这些还是不够的，下面介绍几个常用的Task的使用方法，这样就可以自己动手编写build.xml文件了。

```
<javac srcdir="${src}:${src2}"
      destdir="${build}"
      includes="mypackage/p1/**,mypackage/p2/**"
      excludes="mypackage/p1/testpackage/**"
      classpath="xyz.jar"
      debug="on"
/>
```

这个例子是用于编译Java源程序的，其中srcdir指定源文件所在的目录；destdir指定编译后的class文件存放的目录；includes表示只有mypackage/p1和mypackage/p2目录下的文件必须被包含；而excludes表示mypackage/p1/testpackage/目录下的文件被排除在外，classpath指定了编译这写文件使用的类路径；debug表示是否有调试信息。

```
<jar destfile="${dist}/lib/app.jar"
     basedir="${build}/classes"
     excludes="**/Test.class"
/>
```

这个例子是用于把文件打包的，其中destfile指定了打包后包的文件名；basedir指定被包含文件的基路径；而excludes则表示任何目录下的Test.class文件都不被打包。

```
<java classname="test.Main">
  <arg value="-h"/>
  <classpath>
    <pathelement location="dist/test.jar"/>
    <pathelement path="${java.class.path}"/>
  </classpath>
</java>
```

这个例子用于执行Java程序，classname指定了被执行的类的完整类名；<arg>定义了传给这个类的参数；<classpath>元素定义了运行需要类路径信息。

25.2.6 build.xml实例分析

在25.1节中介绍了一个使用Ant的实际例子，现在对其使用的build.xml文件进行分析，以便更好的理解如何编写build.xml文件。

```
<!-- 定义的Project名称、默认执行的target是dist -->
<project name="SimpleProject" default="dist" basedir=".">
  <description>
    simple example build file
```

```

        </description>
<!-- 定义全局属性 -->
<property name="src" location="src"/>
<property name="dist" location="dist"/>

<!--名为init的target ,它完成建立临时目录的工作 ,并把需要使用的源文件拷贝到合适位置-->
<target name="init">
    <mkdir dir="${dist}"/>
    <mkdir dir="${dist}/WEB-INF"/>
    <mkdir dir="${dist}/WEB-INF/classes"/>
    <copy todir="${dist}" >
        <fileset dir="${src}">
            <include name="*.jsp" />
            <exclude name="build.xml" />
        </fileset>
    </copy>
    <copy todir="${dist}/WEB-INF" >
        <fileset dir="${src}">
            <include name="web.xml" />
            <exclude name="build.xml" />
        </fileset>
    </copy>
</target>

<!-- 名为compile的target ,它编译JavaBean文件 -->
<target name="compile" depends="init"
    description="compile the source " >
    <javac srcdir="${src}" destdir="${dist}/WEB-INF/classes"/>
</target>

<!-- 名为dist的target ,它将Web应用打包 -->
<target name="dist" depends="compile"
    description="generate the distribution" >
    <jar destfile="${basedir}/FirstAnt.war" basedir="${dist}"/>
</target>

<!-- 删除编译和发布时使用的临时目录 -->
<target name="clean"
    description="clean up" >
    <delete dir="${dist}"/>
</target>
</project>

```

在这个文件中首先定义了project的名称是SimpleProject及其属性（默认执行的target是dist、基路径是当前路径），之后定义了2个全局的属性，src表示源程序目录，另外以个为发布需要的临时目录。

之后定义了4个target，分别完成初始化、编译、发布和清除的工作，由于默认的target是dist，所以，如果在运行时不指定target就会运行dist，dist依赖于compile，而compile又依赖于init，所以，先执行init，然后执行compile，最后才是执行dist。

25.3 用Ant发布复杂Web应用

本节继续介绍一个实际使用Ant的例子，在这里例子中，需要使用特定的包文件才能编译一些文件，相对复杂一点，本例是使用的20章中使用的Struts用户登录的例子。

使用Ant之前，先来看一下Struts用户登录应用的程序结构如图25.5所示。

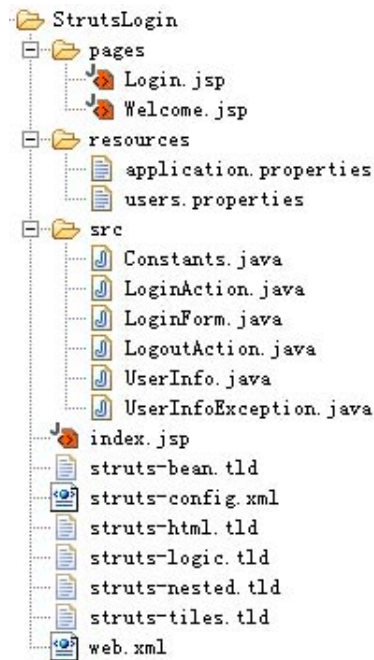


图25.5 Struts用户登录应用的程序结构

25.3.1 build.xml文件

build.xml的作用是创建一个临时目录，然后在这个临时目录下创建Web应用，把Web应用打包后发布到Web服务器。下面是build.xml文件的源代码：

```
<?xml version="1.0" encoding="GBK"?>
<project name="filtering" default="help" basedir=". ">
  <description>Struts登录Web实例</description>

  <!-- 载入属性文件 -->
  <property file="build.properties"/>

  <!-- 定义类路径 -->
  <path id="web.classpath">
    <pathelement location="${tomcat.home}/common/lib/servlet-api.jar"/>
    <pathelement location="${tomcat.home}/common/lib/jsp-api.jar"/>
  </path>
  <path id="struts.classpath">
    <fileset dir="${struts.home}">
      <include name="antlr.jar"/>
      <include name="commons-beanutils.jar"/>
      <include name="commons-digester.jar"/>
    </fileset>
  </path>
</project>
```

```

<include name="commons-fileupload.jar"/>
<include name="commons-logging.jar"/>
<include name="commons-validator.jar"/>
<include name="jakarta-oro.jar"/>
<include name="struts.jar"/>
</fileset>
<path refid="web.classpath"/>
</path>

<!-- 初始化,建立目录 -->
<target name="init">
  <mkdir dir="${dist.dir}"/>
  <mkdir dir="${dist.dir}/WEB-INF"/>
  <mkdir dir="${dist.dir}/WEB-INF/classes"/>
  <mkdir dir="${dist.dir}/WEB-INF/lib"/>
  <copy todir="${dist.dir}" >
    <fileset dir="${basedir}" >
      <include name="pages/*.jsp" />
      <include name="*.jsp" />
      <include name="images/**" />
    </fileset>
  </copy>
  <copy todir="${dist.dir}/WEB-INF/classes" >
    <fileset dir="${basedir}">
      <include name="resources/*.properties" />
    </fileset>
  </copy>
  <copy todir="${dist.dir}/WEB-INF/" >
    <fileset dir="${basedir}" >
      <include name="*.tld" />
      <include name="*.xml" />
      <exclude name="build.xml" />
    </fileset>
  </copy>
</target>

<!-- 编译文件 ---->
<target name="compile" depends="init">
  <javac srcdir="${src.dir}" destdir="${dist.dir}/WEB-INF/classes">
    <classpath refid="web.classpath"/>
  </javac>
</target>

<!-- 拷贝类库文件 ---->
<target name="copyjar" depends="compile">
  <copy todir="${dist.dir}/WEB-INF/lib">
    <fileset dir="${struts.home}">
      <include name="antlr.jar"/>
      <include name="commons-beanutils.jar"/>
      <include name="commons-digester.jar"/>
    </fileset>
  </copy>
</target>

```

```

        <include name="commons-fileupload.jar"/>
        <include name="commons-logging.jar"/>
        <include name="commons-validator.jar"/>
        <include name="jakarta-oro.jar"/>
        <include name="struts.jar"/>
    </fileset>
</copy>

</target>
<!--部署 -->
<target name="deploy" depends="copyjar">
    <jar destfile="${tomcat.home}/webapps/${app.name}.war"
basedir="${dist.dir}"/>
</target>

<target name="clean" depends="init">
<!--删除临时文件 -->
    <delete dir="${dist.dir}"/>
</target>

<target name="help">
<!-- 帮助信息 -->
    <echo>target list </echo>
    <echo>init:make dir and copy file </echo>
    <echo>compile: compile java file </echo>
    <echo>copyjar:prepar struts enviroment </echo>
    <echo>deploy:deploy struts web </echo>
    <echo>clean:delete temp dir </echo>
</target>
</project>

```

可以看到build.xml文件在开始加载了一个属性配置文件build.properties，其中配置了一些在build.xml文件中需要用到的属性，下面是配置文件的代码：

```

# 环境设置 #
# Tomcat安装主目录
tomcat.home=C:/Tomcat 5.0
struts.home=C:/Tomcat 5.0/common
# web的临时目录
dist.dir=./dist
# 源文件目录
src.dir=./src
# 发布的程序名
app.name=StrutsLogin

```

各个属性的作用可以通过注释了解。这样把可变的信息提取出来，使得Ant的使用灵活性更大了。

build.xml文件定义了init、compile、copyjar、deploy、clean、help等6个target。下面分别做简单的介绍。

25.3.2 init目标

init target完成初始化的工作，新建了几个目录，并把不需要编译的JSP文件和图片等拷

到合适的位置。其中使用的临时目录dist.dir就是在属性配置文件中定义的。

注意：dist.dir 是属性的名称，在应用这个属性的时候要这样引用\${dist.dir}。

25.3.3 compile目标

compile target完成编译Java源程序的任务，在编译这些Java文件时引用了前面定义的路径struts.classpath：

```
<path id="struts.classpath">
  <fileset dir="${struts.home}/lib">
    <include name="antlr.jar"/>
    <include name="commons-beanutils.jar"/>
    <include name="commons-digester.jar"/>
    <include name="commons-fileupload.jar"/>
    <include name="commons-logging.jar"/>
    <include name="commons-validator.jar"/>
    <include name="jakarta-oro.jar"/>
    <include name="struts.jar"/>
  </fileset>
  <path refid="web.classpath"/>
</path>
```

它使用了fileset元素和path元素把struts和一些辅助使用的包包含到一起。

25.3.4 copyjar目标

由于Struts应用的运行离不开Struts相关包的支持，所以需要把这些包文件拷贝到WEB-INF/lib目录下，copyjar target就是用来完成这个任务的。

25.3.5 deploy目标

deploy target完成把需要的文件打包并把打包后的Web应用放到<TOMCAT_HOME>\webapps目录下，从而发布这个应用。

25.3.6 clean和help目标

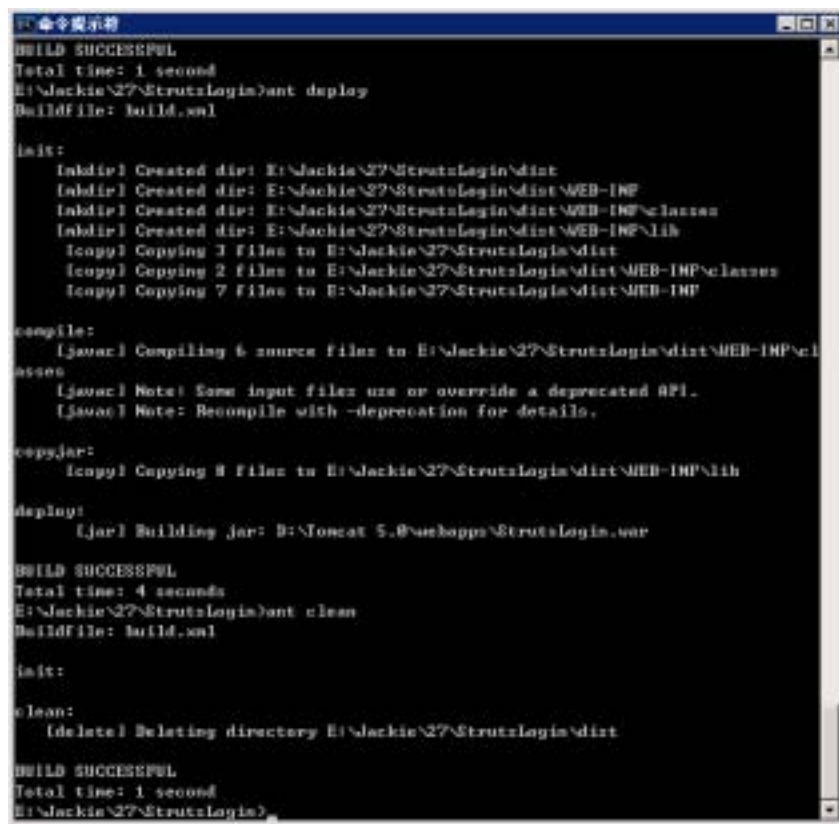
clean和help target的作用就像它们的名字一样明显，clean target删除了发布过程中使用的临时目录，help target是一个帮助的目标，列出了所有可使用的target，并是这个project的默认target。

25.3.7 使用Ant发布Web应用

在Struts应用目录下先后执行如下命令：

```
ant deploy
ant clean
```

命令行提示如图25.6。



```
命令提示符
BUILD SUCCESSFUL
Total time: 1 second
E:\Jackie\27\StrutsLogin>ant deploy
Buildfile: build.xml

build:
[mkdir] Created dir: E:\Jackie\27\StrutsLogin\dist
[mkdir] Created dir: E:\Jackie\27\StrutsLogin\dist\WEB-INF
[mkdir] Created dir: E:\Jackie\27\StrutsLogin\dist\WEB-INF\classes
[mkdir] Created dir: E:\Jackie\27\StrutsLogin\dist\WEB-INF\lib
[copy] Copying 3 files to E:\Jackie\27\StrutsLogin\dist
[copy] Copying 2 files to E:\Jackie\27\StrutsLogin\dist\WEB-INF\classes
[copy] Copying 7 files to E:\Jackie\27\StrutsLogin\dist\WEB-INF

compile:
[javac] Compiling 6 source files to E:\Jackie\27\StrutsLogin\dist\WEB-INF\classes
[javac] Note: Some input files use or override a deprecated API.
[javac] Note: Recompile with -deprecation for details.

copyjar:
[copy] Copying 8 files to E:\Jackie\27\StrutsLogin\dist\WEB-INF\lib

deploy:
[jar] Building jar: D:\Tomcat 5.0\apache\StrutsLogin.war

BUILD SUCCESSFUL
Total time: 4 seconds
E:\Jackie\27\StrutsLogin>ant clean
Buildfile: build.xml

clean:
[delete] Deleting directory E:\Jackie\27\StrutsLogin\dist

BUILD SUCCESSFUL
Total time: 1 second
E:\Jackie\27\StrutsLogin>
```

图25.6 发布Web应用

重新启动Tomcat后，这个StrutsLogin这个Web应用就被成功发布了，可以输入如下地址访问：<http://localhost:8080/StrutsLogin/login.jsp>。

注意：读者在练习发布这个应用时要注意修改 build.properties 文件中 Tomcat 的主目录和 struts 的包文件所在的主目录。

25.4 小结

Ant工具是Apache的一个开放源代码的项目，它是一个非常优秀的软件构建工具。用Ant编译或运行比较大的工程是非常方便的，每个工程都有一个包含与这个工程以及需要Ant执行的任务信息的文件，这个文件在ant中默认是build.xml，也可以在ant执行时指定使用的构建文件。在本章中介绍了如何编写Ant的构建文件，这个文件的编写是使用Ant的重点和难点希望读者多尝试，体会其用法。在第4节中介绍了使用Ant发布JMail Web应用的例子，并解释了每个target完成的工作。

第二十九章 JSP作为客户访问CORBA服务

CORBA是用于分布式计算的成熟技术，在实际应用中得到了广泛的关注。它与Web应用的结合也是很重要的一个方面。在本章中介绍一些CORBA的基础知识，并讲述如何实现

CORBA服务，以及如何在Tomcat中配置，使得CORBA技术和Web开发结合起来应用。

29.1 快速体验CORBA——使用JSP访问CORBA的简单例子

29.1.1 CORBA简介

随着互联网技术的日益成熟，学校、商业企业和很多宽带用户正享受着高速、低价网络信息传输所带来的高品质数字生活。但是，随着网络规模的不断扩大以及计算机软硬件技术水平的大幅提高，传统的应用软件系统的实现方式面临了巨大挑战。

首先，在企业级应用中，硬件系统集成商基于性能、价格、服务等方面的考虑，通常在同一系统中集成来自不同厂商的硬件设备、操作系统、数据库平台和网络协议，由此带来的异构性给应用软件的互操作性、兼容性以及平滑升级能力带来了严重问题。

另外，随着基于网络的业务不断增多，传统的客户/服务器（C/S）模式的分布式应用方式越来越显示出在运行效率、系统网络安全性和系统升级能力等方面的局限性。

为了解决分布式计算环境（DCE，Distributed Computing Environment）中不同硬件设备和软件系统的互联，增强网络间软件的互操作性，解决传统分布式计算模式中的不足等问题，对象管理组织（OMG）提出了公共对象请求代理体系结构（CORBA），以增强软件系统间的互操作能力，使构造灵活的分布式应用系统成为可能。

正是基于面向对象技术的发展和成熟、客户/服务器软件系统模式的普遍应用以及集成已有系统等方面的需求，推动了CORBA技术的成熟与发展。作为面向对象系统的对象通信的核心，CORBA为当今网络计算环境带来了真正意义上的互联。

CORBA（Common Object Request Broker Architecture）是由OMG提出的应用软件体系结构和对象技术规范，其核心是一套标准的语言、接口和协议，以支持异构分布应用程序间的互操作及平台无关的编程语言的对象重用。

OMG于1990年初步提出了CORBA思想，到现在的最新版本是3.0版本，

29.1.2 实现CORBA服务的小例子

在本节中介绍一个实现CORBA服务的小例子，读者可以按照下面介绍的步骤来进行。

1. 使用IDL语言定义IDL接口并编译映射到Java程序

本例使用的IDL文件内容如下：

```
module ArithApp {  
    interface calculate {  
        void add(in long a,in long b, out long c);  
        void muti(in long a,in long b, out long c);  
        void div(in long a,in long b, out long c);  
        void sub(in long a,in long b, out long c);  
    };  
};
```

在该文件所在目录下运行如下命令：

```
idlj -fall calculator.idl
```

可以看到在目录下生成了ArithApp文件夹，下面有六个文件，分别如下。

- _calculateStub.java
- calculate.java
- calculateHelper.java
- calculateHolder.java

- calculateOperations.java
- calculatePOA.java

2. 实现IDL接口

这一步实现在IDL文件中定义的接口，其中类CalImpl是calculatePOA的子类（calculatePOA是由idlj生成的类），下面是CalImpl.java的内容：

```
import org.omg.CORBA.IntHolder;
import org.omg.CORBA.ORB;

import ArithApp.calculatePOA;

public class CalImpl extends calculatePOA {
    private ORB orb;
    public CalImpl(ORB orb){
        this.ORB =orb;
    }
    //以下4个方法分别是四则运算的具体实现
    public void add(int a, int b, IntHolder c) {
        c.value=a+b;
    }

    public void muti(int a, int b, IntHolder c) {
        c.value=a*b;
    }

    public void div(int a, int b, IntHolder c) {
        c.value=a/b;
    }

    public void sub(int a, int b, IntHolder c) {
        c.value=a-b;
    }
}
```

其中，各个方法涉及到的c变量是a和b的运算结果，都使用了IntHolder类型，用于返回结果。

3. 编写服务端实现

下面是服务器CalServer.java的内容：

```
import org.omg.CORBA.ORB;
import org.omg.CosNaming.NameComponent;
import org.omg.CosNaming.NamingContextExt;
import org.omg.CosNaming.NamingContextExtHelper;
import org.omg.PortableServer.POA;
import org.omg.PortableServer.POAHelper;

import ArithApp.calculate;
import ArithApp.calculateHelper;

public class CalServer {
```

```

public CalServer() {
    super();
}

public static void main(String args[]) {
    try{
        //创建并初始化ORB
        ORB orb = ORB.init(args, null);

        //创建一个接口实现的例子，并把它向ORB注册
        CalImpl impl = new CalImpl(orb);

        //获得RootPOA 的引用并激活POAManager
        POA rootpoa = POAHelper.narrow(
            orb.resolve_initial_references("RootPOA"));
        rootpoa.the_POAManager().activate();

        //从服务中得到对象的引用
        org.omg.CORBA.Object ref =
            rootpoa.servant_to_reference(impl);
        calculate href = calculateHelper.narrow(ref);

        //从命名服务中获取根命名上下文并把这个新对象注册为“Cal”
        org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
        NamingContextExt ncRef =
            NamingContextExtHelper.narrow(objRef);

        // 在命名上下文中绑定这个对象
        String name = "Cal";
        NameComponent path[] = ncRef.to_name( name );
        ncRef.rebind(path, href);

        System.out.println("CalServer ready to calculate....");

        //等待客户端的调用
        orb.run();
    } catch (Exception e) {
        System.err.println("ERROR: " + e);
        e.printStackTrace(System.out);
    }
    System.out.println("CalServer Exiting ....");
}
}

```

4. 实现Corba客户端

下面是客户端CalClient.java的内容：

```

import ArithApp.*;
import org.omg.CORBA.*;

```



```

import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;

public class CalClient {

    public CalClient() {
        super();
    }

    public static void main(String args[]) {
        try {
            //创建并初始化ORB
            ORB orb = ORB.init(args, null);

            // 获得根命名上下文
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");

            NamingContextExt ncRef =
                NamingContextExtHelper.narrow(objRef);

            //从命名上下文中获取接口实现对象
            String name = "Cal";
            calculate impl = calculateHelper.narrow(ncRef.resolve_str(name));

            System.out.println("Handle obtained on server object: " + impl);

            int a = 90;
            int b = 7;
            IntHolder c = new IntHolder();
            // 调用乘法
            impl.muti(a,b,c);
            // 输出结果
            System.out.println("The result of a×b is: "+c.value);
            impl.div(a,b,c);
            //输出结果
            System.out.println("The result of a÷b is: "+c.value);
            impl.add(a,b,c);
            //输出结果
            System.out.println("The result of a+b is: "+c.value);
            impl.sub(a,b,c);
            //输出结果
            System.out.println("The result of a-b is: "+c.value);

        } catch (Exception e) {
            System.out.println("ERROR : " + e) ;
            e.printStackTrace(System.out);
        }
    }
}

```

5. 运行程序

把所有的文件都编写并编译完成后，各个Java源文件的存放位置如图29.1，生成的字节码文件放在bin目录下，与Java源文件的字节码文件在bin中的相对位置与在src中相同。按照下面的步骤执行该应用程序：

(1) 启动orbd：在应用程序根目录CorbaApp下运行：

```
orbd -ORBInitialPort 3588
```

(2) 启动服务器端：在应用程序根目录CorbaApp下运行：

```
java CalServer -ORBInitialPort 3588
```

(3) 这时可以看到命令行提示服务器运行成功并等待客户端调用

(4) 启动客户端：在应用程序根目录CorbaApp下运行：

```
java CalClient -ORBInitialPort 3588
```

客户端成功运行就可以看到图29.2的效果：

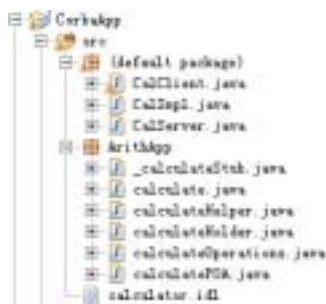


图29.1 Corba应用目录结构图

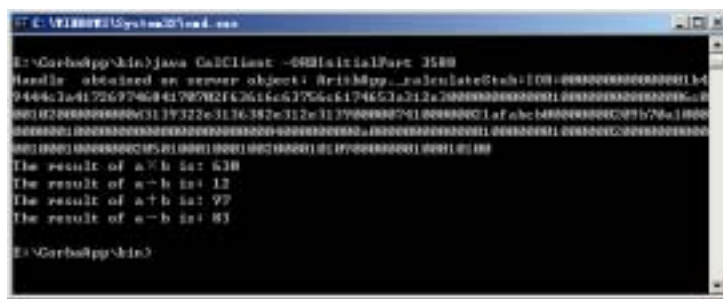


图29.2 Corba应用效果

29.2 CORBA技术构成

CORBA体系结构中设计的几个核心概念有：对象请求代理、接口定义语言、对象适配器等，在本节中结合CORBA的体系结构介绍CORBA架构中的一些核心组件和概念。

29.2.1 对象请求代理 (ORB, Object Request Broker)

CORBA体系结构的核心是ORB，简单而言，ORB就是使客户应用程序能调用远端对象方法的一种机制。

当客户端的应用程序需要调用远程对象的某个方法时，首先需要得到这个远程对象的引用，这样才可以象调用本地对象的方法一样调用远程对象的方法。当客户端发出一个调用请求时，ORB会截获这个请求（通过客户Stub完成），因为客户和服务端有可能在不同的网络、不同的操作系统上，甚至用不同的语言实现，ORB还要负责将调用的名字、参数等编码成标准的方式（称Marshaling），然后通过网络传输到服务器方（即使在同一台机器上也要这样进行），并通过将参数Unmarshaling的过程，传到正确的对象上（这个过程叫重定向，Redirecting），服务器对象完成方法调用后，ORB通过同样的Marshaling/Unmarshaling方式将结果返回给客户。

下面简单介绍一下ORB的结构，图29.3说明的是客户端发送一个请求到对象的实现的过程。



图29.3 客户端发送一个请求到对象的实现

客户端是希望对某对象执行操作的实体。对象的实现是一片代码和数据来实际实现对象。ORB 负责实现下面的必要功能：对象定位（对该请求找到对象的实现）、确信服务器端能接受请求、把客户请求重新定位到服务器端的对象实现上并让对象的实现准备好接受请求、交换数据。客户端的接口完全独立于对象的位置，其实现的语言等因素也不影响对象接口的实现。图29.4展示的是一个独立的ORB的结构。

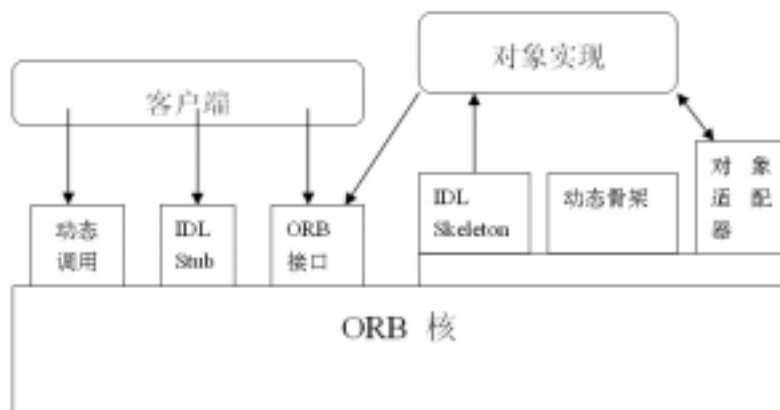


图29.4 一个独立的ORB的结构

客户端可以使用动态调用接口(DII ,Dynamic Invocation Interface)或者一个OMG的IDL 占位程序来提出一个请求。其中DII和IDL 占位程序的区别在于DII是个目标对象的接口独立的，而IDL 占位程序则依赖于目标对象的接口。客户端也可以直接和ORB在某些地方交互。

对象的实现通过OMG的IDL产生的骨架或者是一个动态骨架的调用来接受请求。对象的实现可能在处理请求或其他的时候调用ORB。

29.2.2 接口定义语言 (IDL, Interface Defination Language)

从本质上讲，IDL在CORBA体系结构中不作为程序设计语言，它用于描述产生对象调用请求的客户对象和服务对象之间的接口的语言。IDL文件描述数据类型和方法框架，而服务器上的对象实现则为一个指定的对象实现提供上述数据和方法。

IDL描述了服务器提供的服务功能，客户机可以根据该接口文件描述的方法向服务器提出业务请求。在大多数CORBA产品中都提供IDL到相关编程语言的编译器（例如J2SDK自带Java的idlj）。程序设计人员只需将定义的接口文件提供给编译器，指定编译选项后，就可以得到与程序设计语言相关的接口框架文件和辅助文件。

在语法规则方面，IDL类似于C++或Java中关于接口或对象的定义。OMG IDL只是一种

说明性的语言，支持C++语法中的常量、类型和方法的声明。采用IDL这样的说明性语言，其目的在于克服特定编程语言在软件系统集成及互操作方面的限制。OMG IDL已经为C、C++、Java等主要高级程序设计语言指定了IDL到高级编程语言的映射标准。项目开发人员可以根据需要选择自己最熟悉的编程语言来独立开发基于CORBA的应用，而对软件系统的互操作能力不产生影响。下面简单介绍一下IDL的语法规则。

1. 词法规则

IDL采用ASCII字符集构成接口定义的所有标识符。标识符由字母、数字和下划线的任意组合构成，但第一个字符必须是ASCII字母，不区分大小写，例如astudent和AStudent是相同的。

与C++和Java类似，采用以“/*”开始，以“*/”结束来注释一段代码，以“//”开始注释从“//”开始直至行尾的所有内容。

另外，IDL保留了47个关键字，程序设计人员不能将关键字用作变量或方法名，关键字区分大小写，例如：

```
typedef double context;  
//错误：定义的变量context是关键字  
typedef double CONTEXT;  
//错误：CONTEXT与关键字context冲突
```

2. IDL数据类型

整数类型：IDL整数类型包括short、long和相应的无符号（unsigned）类型，表示的字长分别为16、32位。

浮点数类型：IDL浮点数类型包括float、double和long double类型。其中float表示单精度浮点数，double表示双精度浮点数，long double表示扩展的双精度浮点数。

字符和超大字符类型：IDL定义字符类型char为面向字节的码集中编码的单字节字符；定义类型wchar为从任意字符集中编码的超大字符。

逻辑类型：用boolean关键字定义的一个变量，取值只有true和false。

八进制类型：用octet关键字定义，在网络传输过程中不进行高低位转换的位元序列。

any数据类型：引入该类型用于表示OMG IDL中任意数据类型。

3. 常量

IDL可以用const关键字声明一个常量，用于模块（module）或接口（interface）中定义保持不变的量，如：

```
const double PI = 3.1415926;
```

在IDL中，可以定义long、unsigned long、unsigned short、char、boolean、float、double、string类型的常量。

4. 构造数据类型

构造数据类型类似于C和C++的语法规则，IDL中的构造数据类型包括结构、联合、枚举等形式。下面一一举例介绍：

（1）结构类型：

```
typedef long StudentInfo;  
struct  
{  
    string name;  
    String sno;
```

```

}
(2) 联合类型：
union stockIn switch( short )
{
case 1: stocker : long;
case 2: goodsName1 : string;
case 3: goodsName2 : string;
}
(3) 枚举类型：
enum GoodsStatus {
    GOODS_SALE,
    GOODS_INSTOCK
};

```

5. 数组类型

IDL的数组类型提供了多维定长、统一数据格式的数据存储方式——数组。每一维的长度必须在定义时给定，所有数据单元必须存储相同类型的元素（与C语言类似）。如下例定义一个长度为20×100的long整数数组：

```
typedef long aDimension[20][100];
```

6. 模板（template）类型

IDL提供两种类型的模板：

（1）序列（sequence）类型

用这种类型定义长度可变的任意数值类型的存储序列，在定义时可以指定长度，也可以不指定，如：

```

typedef sequence <long, 80> aSequence;
//长度定义为80
typedef sequence <long> anotherSequence;
//长度不定

```

（2）字符串（string）序列

同样对于字符串序列类型，也有两种定义方式：

```

typedef string <80> aName; //长度定义为80
typedef string anotherName; //长度不定

```

7. 接口（interface）

用于定义CORBA接口，接口是对服务对象功能的详细描述，包含了服务对象提供服务方法的全部信息，客户对象利用该接口获取服务对象的属性、访问服务对象中的方法。

接口用关键字interface声明，其中包含的属性和方法对所有提出服务请求的客户对象是公开的，如下例：

```

interface JobManager
{
    readonly attribute string FirstName;
    attribute string status;
    string QueryJobStatus( in long Number, out string property);
}

```

29.2.3 接口仓库 (IR, Interface Repository)

接口仓库，顾名思义可以理解为CORBA分布式对象接口定义的集中存储区，可以用来辅助用户使用、发布、管理这个存储区中记录的对象接口。CORBA是个分布计算的模型，所以这里所说的存储区也不一定是某个文件或某个数据库，它也是分布的，这里的集中存储是逻辑上的存储，具体的存放位置可能在不同的机器上。

接口仓库要实现的主要功能有：

- 为不同ORB之间的互操作提供实现。
- 供ORB用来检查请求签名。
- 供ORB检查接口之间大继承关系。

接口仓库提供的功能及工具主要依赖于CORBA提供商，根据不同的工具接口仓库还可以实现更多的功能。

29.2.4 对象适配器 (OA, Object Adapter)

对象适配器是ORB的一部分，它帮助ORB把请求传给对象并激活该对象，一个对象适配器只能与一个对象实现联系，它可以被定义来支持特定的对象实现类型（如OODB 对象适配器用于持续对象，而library 对象适配器用于非远程对象）。对象适配器的作用主要有：

- 产生和解释对象引用。
- 方法调用。
- 相互作用的安全性。
- 对象的激活实现及撤销实现。
- 把对象引用映射到相应的对象实现。
- 注册对象实现。

29.2.5 动态调用接口和静态调用接口 (DII, Dynamic Invocation Interface)

动态调用接口和静态调用接口都位于客户端，它们负责发送客户端的调用请求，

在客户端，ORB使用动态调用接口 (Dynamic Invocation Interface, DII) 来发送操作调用；在服务器端，OA通过动态框架接口 (Dynamic Skeleton Interface, DSI) 来传输一个操作调用，它是服务器端对应DII的行为。动态调用接口(DII)可以和动态骨架接口 (DSI) 结合，使得客户可以在不知道服务器对象的接口的情况下调用服务器对象。通常被用于桥。

在客户端，客户与ORB之间的静态接口被称为静态调用接口 (Static Invocation Interface, SII)，在服务器端，与这个接口对应的被称为静态框架接口 (Static Skeleton Interface, SSI)。

29.2.6 GIOP和IIOP

GIOP是一种通信协议，它是客户方的ORB和服务器的ORB通信的协议，它规定了客户和服务器的ORBs间的通信机制。

GIOP是CORBA方法调用的核心部分。GIOP不基于任何特别的网络协议，如IPX或TCP/IP。为了确保互操作性，OMG必须将GIOP定义在所有供应商都支持的特定传输之上。因此，OMG在最广泛使用的通信传输平台 TCP/IP上标准化GIOP。GIOP加TCP/IP就是IIOP。

29.3 股票查询服务——CORBA服务实例

29.3.1 使用IDL语言定义IDL接口并编译映射到Java程序

这是开发CORBA应用程序的第一步，接口中定义了客户端应用程序可以访问的方法，但是没有具体的实现。文件的具体内容如下（文件名为Stock.idl）：

```
module StockApplication {
    interface StockOperation {
        void getStockName(in string StockID,out string StockName);
        void getStockCurrentPrice(in string StockID,out float
currentprice);
        void getStockDiff(in string StockID,out float diff);
    };
};
```

这个例子是要实现股票的查询功能，在上面的接口中定义了3个方法，分别用于根据股票代码获取股票的名称、当前的价格和浮动状况。保存文件到src文件夹后，在src目录下执行：

```
idlj -fall calculator.idl
```

可以看到在目录下生成了ArithApp文件夹，下面有六个文件，分别如下：

- StockOperationPOA.java
- StockOperationOperations.java
- StockOperationHolder.java
- StockOperationHelper.java
- StockOperation.java
- _StockOperationStub.java

至于各个文件的作用这里就不深入介绍了，读者可以在深入学习CORBA的基础上了解它们的用途。

29.3.2 实现IDL接口

在上面一个小节中，在一个IDL文件中定义了几个方法，虽然也生成了对应的Java文件，但是并没有实际执行的代码，下面是实现这个接口的代码：

```
import org.omg.CORBA.IntHolder;
import org.omg.CORBA.ORB;
import org.omg.CORBA.*;
import java.sql.*;

import StockApplication.StockOperationPOA;

public class StockQueryImpl extends StockOperationPOA {
    private ORB orb;
    private boolean connected = false;
    private String defaultSQLdriver = "com.mysql.jdbc.Driver";
    private String defaultmySQLURL =
"jdbc:mysql://localhost/StockBase?user=root&password=ict";
    private Connection conn;
    public StockQueryImpl(ORB orb){
        this.ORB =orb;
```

```

    }

    private void connectDB(String mySqlDriver,String SQLurl){
        try{
            //加载数据库驱动程序
            Class.forName(mySqlDriver);
        }catch(Exception e){
            System.out.println("Driver Error");
        }

        try{
            //与数据库建立连接,得到Connection的对象
            conn=DriverManager.getConnection(SQLurl);
            connected = true;
        }catch(Exception e){
            System.out.println("无法连接");
        }
    }

    public void getStockName(String StockID, StringHolder StockName){
        if(!connected){
            connectDB(defaultSQLdriver,defaultmySQLURL);
        }
        try{
            //得到Statement的对象
            Statement stmt=conn.createStatement();
            ResultSet rs=null;
            //发送SQL语句,并得到执行结果
            rs=stmt.executeQuery("select * from StockList where
StockID='"+StockID+"'");
            rs.next();
            StockName.value=rs.getString("StockName");
        }catch(Exception e){
            StockName.value = "Error";
            System.out.println("getStockName Error!");
        }
    }

    public void getStockCurrentPrice(String StockID, FloatHolder
currentprice){
        if(!connected){
            connectDB(defaultSQLdriver,defaultmySQLURL);
        }
        try{
            //得到Statement的对象
            Statement stmt=conn.createStatement();
            ResultSet rs=null;
            //发送SQL语句,并得到执行结果
            rs=stmt.executeQuery("select * from StockList where
StockID='"+StockID+"'");

```



```

        rs.next();
        currentprice.value=rs.getFloat("currentPrice");
    }catch(Exception e){
        currentprice.value = 0.0F;
        System.out.println("getStockCurrentPrice Error!");
    }
}

public void getStockDiff(String StockID, FloatHolder diff){
    if(!connected){
        connectDB(defaultSQLdriver,defaultmySQLURL);
    }
    try{
        //得到Statement的对象
        Statement stmt=conn.createStatement();
        ResultSet rs=null;
        //发送SQL语句，并得到执行结果
        rs=stmt.executeQuery("select * from StockList where
StockID='"+StockID+"'");
        rs.next();
        diff.value=rs.getFloat("different");
    }catch(Exception e){
        diff.value = 0.0F;
        System.out.println("getStockDiff Error!");
    }
}
}
}

```

注意：StockOperationPOA 是由 idlj 生成的类

在这个文件中有4个方法，其中一个connectDB用于连接数据库，在其他3个业务方法中都要先判断是否具有数据库连接，如果没有就要调用connectDB方法连接数据库：

```

    if(!connected){
        connectDB(defaultSQLdriver,defaultmySQLURL);
    }

```

注意：由于这个应用中需要连接 MySQL 数据库，需要把数据库驱动程序放到类路径中。

29.3.3 编写服务端实现

服务器端的主要功能包括：

- 创建并初始化ORB。
- 创建一个接口实现的例子，并把它向ORB注册。
- 获得RootPOA的引用并激活POAManager。
- 从服务中得到对象的引用。
- 从命名服务中获取根命名上下文并把这个新对象注册为“Cal”。
- 等待客户端的调用。

下面是类StockQueryServer的代码：

```

import org.omg.CORBA.ORB;
import org.omg.CosNaming.NameComponent;

```

```

import org.omg.CosNaming.NamingContextExt;
import org.omg.CosNaming.NamingContextExtHelper;
import org.omg.PortableServer.POA;
import org.omg.PortableServer.POAHelper;

import StockApplication.StockOperationPOA;
import StockApplication.StockOperation;
import StockApplication.StockOperationHelper;

public class StockQueryServer {

    public StockQueryServer() {
        super();
    }

    public static void main(String args[]) {
        try{
            //创建并初始化ORB
            ORB orb = ORB.init(args, null);
            //创建一个接口实现的例子，并把它向ORB注册
            StockQueryImpl impl = new StockQueryImpl(orb);
            //获得RootPOA 的引用并激活POAManager
            POA rootpoa = POAHelper.narrow(
                orb.resolve_initial_references("RootPOA"));
            rootpoa.the_POAManager().activate();
            //从服务中得到对象的引用
            org.omg.CORBA.Object ref =
                rootpoa.servant_to_reference(impl);
            StockOperation href = StockOperationHelper.narrow(ref);
            //从命名服务中获取根命名上下文
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContextExt ncRef =
                NamingContextExtHelper.narrow(objRef);

            //把这个新对象注册为“Stock”并在命名上下文中绑定这个对象
            String name = "Stock";
            NameComponent path[] = ncRef.to_name( name );
            ncRef.rebind(path, href);
            System.out.println("StockQuery Server ready to Work....");
            //等待客户端的调用
            orb.run();
        } catch (Exception e) {
            System.err.println("ERROR: " + e);
            e.printStackTrace(System.out);
        }
        System.out.println("StockQuery Server Exiting ....");
    }
}

```

29.3.4 准备数据库

在本实例中使用MySQL数据库系统，需要一个数据名为StockBase，并有一个名为StockList的数据表，使用的SQL语句如下：

```
CREATE TABLE `stocklist` (  
  `StockID` char(50) NOT NULL default '',  
  `StockName` char(50) default NULL,  
  `currentPrice` float default NULL,  
  `topPrice` float default NULL,  
  `downPrice` float default NULL,  
  `different` float default NULL,  
  `description` char(100) default NULL,  
  PRIMARY KEY (`StockID`)  
);
```

还需要插入一些测试的数据，代码如下：

```
INSERT INTO `stocklist` VALUES ('600191','华资实业',5.61,7.65,4.98,10,''),  
( '600192','长江实业',98.63,85.96,120.3,12.9,'豆腐干大');
```

29.3.5 实现Corba客户端

Corba客户端是要调用在IDL接口中定义的接口的程序，它实现调用的过程是这样的：

- 创建并初始化ORB。
- 获得根命名上下文的引用。
- 寻找名为“Stock”的对象，并获得其引用。
- 调用接口实现中实现的方法。

在本实例中，使用一个Servlet首先从服务器端获取一个远程对象，并把这个对象作为Application范围的属性，在其他页面需要使用时只需要从JSP的application隐含对象中获取这个属性就可以了，下面是获取远程对象的Servlet：

```
package cn.ac.ict;  
  
import org.omg.CORBA.*;  
import org.omg.CosNaming.*;  
import StockApplication.StockOperation;  
import StockApplication.StockOperationHelper;  
  
import javax.servlet.http.*;  
  
public class StockQueryClientServlet extends HttpServlet {  
  
  /**  
   * 构造方法  
   */  
  public StockQueryClientServlet() {  
    super();  
  }  
  
  public void init() {  
    try {  
      String args[] = new String[2];
```

```

args[0] = "-ORBInitialPort";
args[1] = "3588";

// 创建并初始化 ORB
ORB orb = ORB.init(args, null);
// 获得根命名上下文
org.omg.CORBA.Object objRef =
    orb.resolve_initial_references("NameService");
// Use NamingContextExt instead of NamingContext. This is
// part of the Interoperable Naming Service.
NamingContextExt ncRef =
    NamingContextExtHelper.narrow(objRef);
// 在命名服务中处理对象引用
String name = "Stock";
StockOperation impl =
    StockOperationHelper.narrow(ncRef.resolve_str(name));
this.getServletContext().setAttribute("StockQuery", impl);

    } catch (Exception e) {
        System.out.println("ERROR : " + e);
        e.printStackTrace(System.out);
    }
}
}

```

下面介绍一个使用Servlet访问的例子，当然读者完全可以使用JSP来访问。

```

package cn.ac.ict;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.omg.CORBA.FloatHolder;
import org.omg.CORBA.StringHolder;
import StockApplication.StockOperation;

public class stockQuery extends HttpServlet {

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

    protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {
        PrintWriter out;
        String title="股票查询";
    }
}

```

```

// 设置响应内容的类型.
response.setContentType("text/html;charset=GB2312");
out = response.getWriter();
out.write("<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">\r\n");
out.write("<html>\r\n");
out.write("<head>\r\n");
out.write("<meta http-equiv=\"Content-Type\" content=\"text/html;
charset=gb2312\">\r\n");
out.println("<title>" + title + "</title>");
out.write("</head>\r\n");
out.write("\r\n");
out.write("<body>\r\n");
out.write("<form action=\"stockQuery.stockquery\"
method=\"post\">\r\n");
out.write("<label>输入股票代码:</label><input name=\"stockID\"
type=\"text\"><input name=\"\" type=\"submit\">\r\n");
out.write("</form>\r\n");
//得到用户的请求信息
String SID =request.getParameter("stockID");
if((SID!=null)&&(!SID.equals(""))){
//从ServletContext的对象中获取一个远程对象
StockOperation impl =
(StockOperation)this.getServletContext().getAttribute("StockQuery");
StringHolder sname = new StringHolder();
FloatHolder currentprice = new FloatHolder();
FloatHolder diff = new FloatHolder();
//调用查询数据的方法
impl.getStockName(SID,sname);
impl.getStockCurrentPrice(SID,currentprice);
impl.getStockDiff(SID,diff);

out.write("\r\n");
out.write("<table width=\"580\" border=\"0\"
cellspacing=\"0\" cellpadding=\"0\" align=\"center\">\r\n");
out.write("<thead align=\"center\">查询结果</thead>\r\n");
out.write("  <tr>\r\n");
out.write("    <td>股票代码</td>\r\n");
out.write("    <td>股票名称</td>\r\n");
out.write("    <td>当前价格</td>\r\n");
out.write("    <td>浮动</td>\r\n");
out.write("  </tr>\r\n");
out.write("  <tr>\r\n");
out.write("    <td>");
out.print( SID );
out.write("</td>\r\n");
out.write("    <td>");
out.print( sname.value );
out.write("</td>\r\n");
out.write("    <td>");

```

```

        out.print( currentprice.value );
        out.write("</td>\r\n");
        out.write("    <td>");
        out.print( diff.value );
        out.write("</td>\r\n");
        out.write("  </tr>\r\n");
        out.write("</table>\r\n");
    }

    out.write("\r\n");
    out.write("</body>\r\n");
    out.write("</html>\r\n");
}
}

```

由于使用了Servlet获取远程对象和调用业务方法，就需要在web.xml文件中配置这些Servlet，配置的代码在下一节介绍。

29.3.6 配置Corba服务的Servlet客户端

配置Corba服务的Servlet客户端与配置普通的Servlet没有什么区别，在发布描述文件中添加关于Servlet的信息就可以了，例如，在本例中在Web应用的web.xml文件中添加如下代码：

```

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

<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">

  <display-name>Stock Query Application with CORBA</display-name>

  <servlet>
    <servlet-name>Stock Query Init</servlet-name>
    <servlet-class>cn.ac.ict.StockQueryClientServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet>
    <servlet-name>Stock Query</servlet-name>
    <servlet-class>cn.ac.ict.stockQuery</servlet-class>
    <load-on-startup>2</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Stock Query</servlet-name>
    <url-pattern>*.stockquery</url-pattern>
  </servlet-mapping>
</web-app>

```

注意：由于用于获取远程对象的 Servlet 不需要让客户直接访问，所以没有设置相应的<servlet-mapping>元素，并且使用<load-on-startup>1</load-on-startup>使得这个 Servlet 在 Web 容器启动时就被初始化，保证了客户访问时总有远程对象可使用。

29.3.7 测试Corba服务与Web应用

把所有的文件都编写并编译完成后，各个Java源文件的存放位置如图29.5，生成的字节码文件放在bin目录下，与Java源文件的字节码文件在bin中的相对位置与在src中相同。

按照下面的步骤执行该应用程序：

- (1) 启动orbd：在应用程序根目录下运行：
orbd -ORBInitialPort 3588
- (2) 启动服务器端：在应用程序根目录下运行：
java StockQueryServer -ORBInitialPort 3588
- (3) 这时可以看到命令行提示服务器运行成功并等待客户端调用
- (4) 启动Tomcat服务器，在浏览器地址栏中输入如下地址：
http://localhost:8080/32/stockQuery.stockquery
可以看到页面显示如图29.6。



图29.5 Web应用目录结构



图29.6 输入页面

如果查询了数据库中不存在的股票，就会显示出错误的股票名称，一个简单的可能页面如图29.7。而输入的股票代号是存在的时就会显示出正确的信息，页面效果如图29.8。



图29.7 查询结果出现错误



图29.8 查询结果正确显示

29.4 小结

本章简单介绍了在分布式计算领域得到广泛应用的CORBA技术，CORBA是支持异构分布应用程序间的互操作及平台无关的，使用它可以开发性能良好的分布式的、平台无关的应用程序。

本章先是通过一个四则运算的例子演示了CORBA技术与Java技术的结合，然后解释了CORBA的一些组件和重要概念，但可以说这只是会常常遇到的一部分，读者要掌握CORBA还需要参考更多的资料，在最后一节演示了一个实际使用CORBA的股票查询例子，读者应该对照例子更好的理解如何使用CORBA。