How Tomcat Works 中文版

介绍

概要

欢迎阅读《How Tomcat Works》这本书。这本书解剖了 Tomcat4.1.12 和 5.0.18 版本，解释了它的 servlet 容器的内部运行机制，那是一个免费的，开源的，最受欢迎的 servlet 容器，代号为 Catalina。Tomcat 是一个复杂的系统，由许多不同的组件构成。那些想要学习 Tomcat 运行机制的朋友大部分知道从何入手。这本书会提供一个蓝图，然后为每一个组件构造一个简化版本，使得可以更加容易的理解这些组件。在这之后才会对真实的组件进行解释。

你应该从这份简介开始阅读，因为它解释了这本书的结构，同时给你勾画了这个项目构造的简洁轮廓。“准备前提软件”这一节会给你一些指示，例如你需要下载什么样的软件，如何为你的代码创建目录结构等等。

本书为谁而作

这本书是为任何一个使用 Java 技术进行工作的人而准备的。

- 假如你是一个 servlet/jsp 程序员或者一个 Tomcat 用户，而且对一个 servlet 容器是如何工作这个问题你感兴趣的话，这本书就是为你准备的。
- 假如你想加入 Tomcat 的开发团队的话，这本书就是为你准备的，因为你首先需要学习那些已存在的代码是如何工作的。
- 假如你从未涉及 web 开发，但你对一般意义上的软件开发感兴趣的话，你可以在这本书学到一个像 Tomcat 一样的大型项目是如何进行设计和开发的。
- 假如你想配置和自定义 Tomcat，你也应该读读这本书。

为了理解书中的讨论，你需要了解 Java 面向对象编程技术以及 servlet 编程。假如你对这些不熟悉的话，这里有很多书籍可以参考，包括 Budi 的《Java for the Web with Servlets, JSP, and EJB》。为了让这些材料更容易理解，每一章开始都会有便于理解所讨论主题的必要的背景资料介绍。

Servlet 容器是如何工作的

servlet 容器是一个复杂的系统。不过，一个 servlet 容器要为一个 servlet 的请求提供服务，基本上有三件事要做:

- 创建一个 request 对象并填充那些有可能被所引用的 servlet 使用的信息，如参数、头部、cookies、查询字符串、URI 等等。一个 request 对象是 javax.servlet.ServletRequest 或 javax.servlet.http.ServletRequest 接口的一个实例。
- 创建一个 response 对象，所引用的 servlet 使用它来给客户端发送响应。一个 response 对象 javax.servlet.ServletResponse 或 javax.servlet.http.ServletResponse 接口的一个实例。

- 调用 servlet 的 service 方法，并传入 request 和 response 对象。在这里 servlet 会从 request 对象取值，给 response 写值。

当你读这些章节的时候，你将会找到关于 catalina servlet 容器的详细讨论。

Catalina 架构图

Catalina 是一个非常复杂的，并优雅的设计开发出来的软件，同时它也是模块化的。基于“Servlet 容器是如何工作的”这一节中提到的任务，你可以把 Catalina 看成是由两个主要模块所组成的：连接器(connector)和容器(container)。在 Figure I.1 中的架构图，当然是简化了。在稍后的章节里边，你将会一个个的揭开所有更小的组件的神秘面纱。

Figure I.1: Catalina's main modules

现在重新回到 Figure I.1，连接器是用来“连接”容器里边的请求的。它的工作是为接收到每一个 HTTP 请求构造一个 request 和 response 对象。然后它把流程传递给容器。容器从连接器接收到 requset 和 response 对象之后调用 servlet 的 service 方法用于响应。谨记，这个描述仅仅是冰山一角而已。这里容器做了相当多事情。例如，在它调用 servlet 的 service 方法之前，它必须加载这个 servlet，验证用户(假如需要的话)，更新用户会话等等。一个容器为了处理这个进程使用了很多不同的模块，这也并不奇怪。例如，管理模块是用来处理用户会话，而加载器是用来加载 servlet 类等等。

Tomcat 4 和 5

这本书涵盖了 Tomcat4 和 5.这两者有一些不同之处:

- Tomcat 5 支持 Servlet 2.4 和 JSP 2.0 规范，而 Tomcat 4 支持 Servlet 2.3 和 JSP 1.2。
- 比起 Tomcat 4，Tomcat 5 有一些更有效率的默认连接器。
- Tomcat 5 共享一个后台处理线程，而 Tomcat 4 的组件都有属于自己的后台处理线程。因此，就这一点而言，Tomcat 5 消耗较少的资源。
- Tomcat 5 并不需要一个映射组件(mapper component)用于查找子组件，因此简化了代码。

各章概述

这本书共 20 章，其中前面两章作为导言。

第 1 章说明一个 HTTP 服务器是如何工作的，第 2 章突出介绍了一个简单的 servlet 容器。接下来的两章关注连接器，第 5 章到第 20 章涵盖容器里边的每一个组件。以下是各章节的摘要。

注意:对于每个章节，会有一个附带程序，类似于正在被解释的组件。

第 1 章从这本书一开始就介绍了一个简单的 HTTP 服务器。要建立一个可工作的 HTTP 服务器，你需要知道在 java.net 包里边的 2 个类的内部运作：Socket 和 ServerSocket。这里有关于这 2 个类足够的背景资料，使得你能够理解附带程序是如何工作的。

第 2 章说明简单的 servlet 容器是如何工作的。这一章带有 2 个 servlet 容器应用，可以处理静态资源和简单的 servlet 请求。尤其是你将会学到如何创建 request 和 response 对象，然

后把它们传递给被请求的 servlet 的 service 方法。在 servlet 容器里边还有一个 servlet，你可以从一个 web 浏览器中调用它。

第 3 章介绍了一个简化版本的 Tomcat 4 默认连接器。这章里边的程序提供了一个学习工具，用于理解第 4 章里边的讨论的连接器。

第 4 章介绍了 Tomcat 4 的默认连接器。这个连接器已经不推荐使用，推荐使用一个更快的连接器，Coyote。不过，默认的连接器更简单，更易于理解。

第 5 章讨论 container 模块。container 指的是 org.apache.catalina.Container 接口，有 4 种类型的 container:engine, host, context 和 wrapper。这章提供了两个工作于 context 和 wrapper 的程序。

第 6 章解释了 Lifecycle 接口。这个接口定义了一个 Catalina 组件的生命周期，并提供了一个优雅的方式，用来把在该组件发生的事件通知其他组件。另外，Lifecycle 接口提供了一个优雅的机制，用于在 Catalina 通过单一的 start/stop 来启动和停止组件

第 7 章包括日志，该组件是用来记录错误信息和其他信息的。

第 8 章解释了加载器(loader)。加载器是一个重要的 Catalina 模块，负责加载 servlet 和一个 web 应用所需的其他类。这章还展示了如何实现应用的重新加载。

第 9 章讨论了管理器(manager)。这个组件用来管理会话管理中的会话信息。它解释了各式各样类型的管理器，管理器是如何把会话对象持久化的。在章末，你将会学到如何创建一个的应用，该应用使用 StandardManager 实例来运行一个使用会话对象进行储值的 servlet。

第 10 章包括 web 应用程序安全性的限制，用来限制进入某些内容。你将会学习与安全相关的实体，例如
主角(principals)，角色(roles)，登陆配置，认证等等。你也将会写两个程序，它们在 StandardContext 对象中安装一个身份验证阀(authenticator valve)并且使用了基本的认证来对用户进行认证。

第 11 章详细解释了在一个 web 应用中代表一个 servlet 的 org.apache.catalina.core.StandardWrapper 类。特别的是，这章解释了过滤器(filter)和一个 servlet 的 service 方法是怎样给调用的。这章的附带程序使用 StandardWrapper 实例来代表 servlet。

第 12 章包括了在一个 web 应用中代表一个 servlet 的 org.apache.catalina.core.StandardContext 类。特别是这章讨论了一个 StandardContext 对象是如何给配置的，对于每个传入的 HTTP 请求在它里面会发生什么，是怎样支持自动重新加载的，还有就是，在一个在其相关的组件中执行定期任务的线程中，Tomcat 5 是如何共享的。

第 13 章介绍了另外两个容器：host 和 engine。你也同样可以找到这两个容器的标准实现:org.apache.catalina.core.StandardHost 和 org.apache.catalina.core.StandardEngine。

第 14 章提供了服务器和服务组件的部分。服务器为整个 servlet 容器提供了一个优雅的启动和停止机制，而服务为容器和一个或多个连接器提供了一个支架。这章附带的程序说明了如何使用服务器和服务。

第 15 章解释了通过 Digester 来配置 web 应用。Digester 是来源于 Apache 软件基金会的一个令人振奋的开源项目。对那些尚未初步了解的人，这章通过一节略微介绍了 Digester 库以及 XML 文件中如何使用它来把节点转换为 Java 对象。然后解释了用来配置一个 StandardContext 实例的 ContextConfig 对象。

第 16 章解释了 shutdown 钩子，Tomcat 使用它总能获得一个机会用于 clean-up，而无论用户是怎样停止它的(即适当的发送一个 shutdown 命令或者不适当的简单关闭控制台)。

第 17 章讨论了通过批处理文件和 shell 脚本对 Tomcat 进行启动和停止。

第 18 章介绍了部署工具(deployer)，这个组件是负责部署和安装 web 应用的。

第 19 章讨论了一个特殊的接口，ContainerServlet，能够让 servlet 访问 Catalina 的内部对象。特别是，它讨论了 Manager 应用，你可以通过它来部署应用程序。

第 20 章讨论了 JMX 以及 Tomcat 是如何通过为其内部对象创建 MBeans 使得这些对象可管理的。

各章的程序

每一章附带了一个或者多个程序，侧重于 Catalina 的一个特定的组件。通常你可以找到这些简化版本，无论是正在被解释的组件或者解释如何使用 Catalina 组件的代码。各章节的程序的所有的类和接口都放在 ex[章节号].pyrmont 包或者它的子包。例如第 1 章的程序的类就是放在 ex01.pyrmont 包中。

准备的前提软件

这本书附带的程序运行于 J2SE1.4 版本。压缩源文件可以从作者的网站 www.brainysoftware.com 中下载。它包括 Tomcat 4.1.12 和这本书所使用的程序的源代码。假设你已经安装了 J2SE 1.4 并且你的 path 环境变量中已经包括了 JDK 的安装目录，请按照下列步骤：

1. 解压缩 ZIP 文件。所有的解压缩文件将放在一个新的目录 howtomcatworks 中。howtomcatworks 将是你的工作目录。在 howtomcatworks 目录下面将会有数个子目录，包括 lib（包括所有所需的库），src（包括所有的源文件），webroot（包括一个 HTML 文件和三个 servlet 样本），和 webapps（包括示例应用程序）。
2. 改变目录到工作目录下并编译 java 文件。加入你使用的是 Windows，运行 win-compile.bat 文件。假如你的计算机是 Linux 机器，敲入以下内容：（如有必要的话不用忘记使用 chmod 更改文件属性）

   ```
   ./linux-compile.sh
   ```

注意：你可以在 ZIP 文件中的 Readme.txt 文件找到更多信息。

第一章:一个简单的 Web 服务器

本章说明 java web 服务器是如何工作的。Web 服务器也成为超文本传输协议(HTTP)服务器，

因为它使用 HTTP 来跟客户端进行通信的，这通常是个 web 浏览器。一个基于 java 的 web 服务器使用两个重要的类：java.net.Socket 和 java.net.ServerSocket，并通过 HTTP 消息进行通信。因此这章就自然是从 HTTP 和这两个类的讨论开始的。接下去，解释这章附带的一个简单的 web 服务器。

超文本传输协议(HTTP)

HTTP 是一种协议，允许 web 服务器和浏览器通过互联网进行来发送和接受数据。它是一种请求和响应协议。客户端请求一个文件而服务器响应请求。HTTP 使用可靠的 TCP 连接--TCP 默认使用 80 端口。第一个 HTTP 版是 HTTP/0.9，然后被 HTTP/1.0 所替代。正在取代 HTTP/1.0 的是当前版本 HTTP/1.1，它定义于征求意见文档(RFC) 2616，可以从 http://www.w3.org/Protocols/HTTP/1.1/rfc2616.pdf 下载。

注意：本节涵盖的 HTTP 1.1 只是简略的帮助你理解 web 服务器应用发送的消息。假如你对更多详细信息感兴趣，请阅读 RFC 2616。

在 HTTP 中，始终都是客户端通过建立连接和发送一个 HTTP 请求从而开启一个事务。web 服务器不需要联系客户端或者对客户端做一个回调连接。无论是客户端或者服务器都可以提前终止连接。举例来说，当你正在使用一个 web 浏览器的时候，可以通过点击浏览器上的停止按钮来停止一个文件的下载进程，从而有效的关闭与 web 服务器的 HTTP 连接。

HTTP 请求

一个 HTTP 请求包括三个组成部分：

- 方法—统一资源标识符(URI)—协议/版本
- 请求的头部
- 主体内容

下面是一个 HTTP 请求的例子：

```
POST /examples/default.jsp HTTP/1.1
Accept: text/plain; text/html
Accept-Language: en-gb
Connection: Keep-Alive
Host: localhost
User-Agent: Mozilla/4.0 (compatible; MSIE 4.01; Windows 98)
Content-Length: 33
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate

lastName=Franks&firstName=Michael
```

方法—统一资源标识符(URI)—协议/版本出现在请求的第一行。

```
POST /examples/default.jsp HTTP/1.1
```

这里 POST 是请求方法，/examples/default.jsp 是 URI，而 HTTP/1.1 是协议/版本部分。

每个 HTTP 请求可以使用 HTTP 标准里边提到的多种方法之一。HTTP 1.1 支持 7 种类型的请求：GET, POST,
HEAD, OPTIONS, PUT, DELETE 和 TRACE。GET 和 POST 在互联网应用里边最普遍使用的。

URI 完全指明了一个互联网资源。URI 通常是相对服务器的根目录解释的。因此，始终一斜线/开头。统一资源定位器(URL)其实是一种 URI(查看 http://www.ietf.org/rfc/rfc2396.txt)

来的。该协议版本代表了正在使用的 HTTP 协议的版本。

请求的头部包含了关于客户端环境和请求的主体内容的有用信息。例如它可能包括浏览器设置的语言，主体内容的长度等等。每个头部通过一个回车换行符(CRLF)来分隔的。

对于 HTTP 请求格式来说，头部和主体内容之间有一个回车换行符(CRLF)是相当重要的。CRLF 告诉HTTP服务器主体内容是在什么地方开始的。在一些互联网编程书籍中，CRLF 还被认为是HTTP 请求的第四部分。

在前面一个 HTTP 请求中，主体内容只不过是下面一行：

```
lastName=Franks&firstName=Michael
```

实体内容在一个典型的 HTTP 请求中可以很容易的变得更长。

HTTP 响应

类似于 HTTP 请求，一个 HTTP 响应也包括三个组成部分：

- 方法—统一资源标识符(URI)—协议/版本
- 响应的头部
- 主体内容

下面是一个 HTTP 响应的例子：

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/4.0
Date: Mon, 5 Jan 2004 13:13:33 GMT
Content-Type: text/html
Last-Modified: Mon, 5 Jan 2004 13:13:12 GMT
Content-Length: 112

<html>
<head>
<title>HTTP Response Example</title>
</head>
<body>
Welcome to Brainy Software
</body>
</html>
```

响应头部的第一行类似于请求头部的第一行。第一行告诉你该协议使用 HTTP 1.1，请求成功(200=成功)，表示一切都运行良好。

响应头部和请求头部类似，也包括很多有用的信息。响应的主体内容是响应本身的 HTML 内容。头部和主体内容通过 CRLF 分隔开来。

Socket 类

套接字是网络连接的一个端点。套接字使得一个应用可以从网络中读取和写入数据。放在两个不同计算机上的两个应用可以通过连接发送和接受字节流。为了从你的应用发送一条信息到另一个应用，你需要知道另一个应用的 IP 地址和套接字端口。在 Java 里边，套接字指的是

java.net.Socket 类。

要创建一个套接字，你可以使用 Socket 类众多构造方法中的一个。其中一个接收主机名称和端口号：

```
public Socket (java.lang.String host, int port)
```

在这里主机是指远程机器名称或者 IP 地址，端口是指远程应用的端口号。例如，要连接 yahoo.com 的 80 端口，你需要构造以下的 Socket 对象：

```
new Socket ("yahoo.com", 80);
```

一旦你成功创建了一个 Socket 类的实例，你可以使用它来发送和接受字节流。要发送字节流，你首先必须调用 Socket 类的 getOutputStream 方法来获取一个 java.io.OutputStream 对象。要发送文本到一个远程应用，你经常要从返回的 OutputStream 对象中构造一个 java.io.PrintWriter 对象。要从连接的另一端接受字节流，你可以调用 Socket 类的 getInputStream 方法用来返回一个 java.io.InputStream 对象。

以下的代码片段创建了一个套接字，可以和本地 HTTP 服务器(127.0.0.1 是指本地主机)进行通讯，发送一个 HTTP 请求，并从服务器接受响应。它创建了一个 StringBuffer 对象来保存响应并在控制台上打印出来。

```
Socket socket = new Socket("127.0.0.1", "8080");
OutputStream os = socket.getOutputStream();
boolean autoflush = true;
PrintWriter out = new PrintWriter(
socket.getOutputStream(), autoflush);
BufferedReader in = new BufferedReader(
new InputStreamReader( socket.getInputstream() ));
// send an HTTP request to the web server
out.println("GET /index.jsp HTTP/1.1");
out.println("Host: localhost:8080");
out.println("Connection: Close");
out.println();
// read the response
boolean loop = true;
StringBuffer sb = new StringBuffer(8096);
while (loop) {
    if ( in.ready() ) {
        int i=0;
        while (i!=-1) {
            i = in.read();
            sb.append((char) i);
        }
    loop = false;
    }
    Thread.currentThread().sleep(50);
}
```

```
// display the response to the out console
System.out.println(sb.toString());
socket.close();
```

请注意，为了从 web 服务器获取适当的响应，你需要发送一个遵守 HTTP 协议的 HTTP 请求。假如你已经阅读了前面一节超文本传输协议(HTTP)，你应该能够理解上面代码提到的 HTTP 请求。

注意：你可以本书附带的 com.brainysoftware.pyrmont.util.HttpSniffer 类来发送一个 HTTP 请求并显示响应。要使用这个 Java 程序，你必须连接到互联网上。虽然它有可能并不会起作用，假如你有设置防火墙的话。

ServerSocket 类

Socket 类代表一个客户端套接字，即任何时候你想连接到一个远程服务器应用的时候你构造的套接字，现在，假如你想实施一个服务器应用，例如一个 HTTP 服务器或者 FTP 服务器，你需要一种不同的做法。这是因为你的服务器必须随时待命，因为它不知道一个客户端应用什么时候会尝试去连接它。为了让你的应用能随时待命，你需要使用 java.net.ServerSocket 类。这是服务器套接字的实现。

ServerSocket 和 Socket 不同，服务器套接字的角色是等待来自客户端的连接请求。一旦服务器套接字获得一个连接请求，它创建一个 Socket 实例来与客户端进行通信。

要创建一个服务器套接字，你需要使用 ServerSocket 类提供的四个构造方法中的一个。你需要指定 IP 地址和服务器套接字将要进行监听的端口号。通常，IP 地址将会是 127.0.0.1，也就是说，服务器套接字将会监听本地机器。服务器套接字正在监听的 IP 地址被称为是绑定地址。服务器套接字的另一个重要的属性是 backlog，这是服务器套接字开始拒绝传入的请求之前，传入的连接请求的最大队列长度。

其中一个 ServerSocket 类的构造方法如下所示:

```
public ServerSocket(int port, int backLog, InetAddress bindingAddress);
```

对于这个构造方法，绑定地址必须是 java.net.InetAddress 的一个实例。一种构造 InetAddress 对象的简单的方法是调用它的静态方法 getByName，传入一个包含主机名称的字符串，就像下面的代码一样。

```
InetAddress.getByName("127.0.0.1");
```

下面一行代码构造了一个监听的本地机器 8080 端口的 ServerSocket，它的 backlog 为 1。

```
new ServerSocket(8080, 1, InetAddress.getByName("127.0.0.1"));
```

一旦你有一个 ServerSocket 实例，你可以让它在绑定地址和服务器套接字正在监听的端口上等待传入的连接请求。你可以通过调用 ServerSocket 类的 accept 方法做到这点。这个方法只会在有连接请求时才会返回，并且返回值是一个 Socket 类的实例。Socket 对象接下去可以发送字节流并从客户端应用中接受字节流，就像前一节"Socket 类"解释的那样。实际上，这章附带的程序中，accept 方法是唯一用到的方法。

应用程序

我们的 web 服务器应用程序放在 ex01.pyrmont 包里边，由三个类组成：

- HttpServer
- Request
- Response

这个应用程序的入口点(静态 main 方法)可以在 HttpServer 类里边找到。main 方法创建了一个 HttpServer 的实例并调用了它的 await 方法。await 方法，顾名思义就是在一个指定的端口上等待 HTTP 请求,处理它们并发送响应返回客户端。它一直等待直至接收到 shutdown 命令。

应用程序不能做什么,除了发送静态资源,例如放在一个特定目录的 HTML 文件和图像文件。它也在控制台上显示传入的 HTTP 请求的字节流。不过，它不给浏览器发送任何的头部例如日期或者 cookies。

现在我们将在以下各小节中看看这三个类。

HttpServer 类

HttpServer 类代表一个 web 服务器并展示在 Listing 1.1 中。请注意，await 方法放在 Listing 1.2 中，为了节省空间没有重复放在 Listing 1.1 中。

Listing 1.1: HttpServer 类

```
package ex01.pyrmont;
import java.net.Socket;
import java.net.ServerSocket;
import java.net.InetAddress;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.IOException;
import java.io.File;
public class HttpServer {
    /** WEB_ROOT is the directory where our HTML and other files reside.
    * For this package, WEB_ROOT is the "webroot" directory under the
    * working directory.
    * The working directory is the location in the file system
    * from where the java command was invoked.
    */
    public static final String WEB_ROOT =
    System.getProperty("user.dir") + File.separator + "webroot";
    // shutdown command
    private static final String SHUTDOWN_COMMAND = "/SHUTDOWN";
    // the shutdown command received
    private boolean shutdown = false;
    public static void main(String[] args) {
        HttpServer server = new HttpServer();
        server.await();
    }
    public void await() {
```

```
        ...
    }
}
```

Listing 1.2: HttpServer 类的 await 方法

```
public void await() {
    ServerSocket serverSocket = null;
    int port = 8080;
    try {
        serverSocket = new ServerSocket(port, 1,
        InetAddress.getByName("127.0.0.1"));
    }
    catch (IOException e) {
    e.printStackTrace();
    System.exit(1);
    }
    // Loop waiting for a request
    while (!shutdown) {
        Socket socket = null;
        InputStream input = null;
        OutputStream output = null;
        try {
            socket = serverSocket.accept();
            input = socket.getInputStream();
            output = socket.getOutputStream();
            // create Request object and parse
            Request request = new Request(input);
            request.parse();
            // create Response object
            Response response = new Response(output);
            response.setRequest(request);
            response.sendStaticResource();
            // Close the socket
            socket.close();
            //check if the previous URI is a shutdown command
            shutdown = request.getUri().equals(SHUTDOWN_COMMAND);
        }
        catch (Exception e) {
            e.printStackTrace ();
            continue;
        }
    }
}
```

web 服务器能提供公共静态 final 变量 WEB_ROOT 所在的目录和它下面所有的子目录下的静态资源。如下所示，WEB_ROOT 被初始化：

```
public static final String WEB_ROOT =
System.getProperty("user.dir") + File.separator + "webroot";
```

代码列表包括一个叫 webroot 的目录，包含了一些你可以用来测试这个应用程序的静态资源。你同样可以在相同的目录下找到几个 servlet 用于测试下一章的应用程序。为了请求一个静态资源，在你的浏览器的地址栏或者网址框里边敲入以下的 URL：

```
http://machineName:port/staticResource
```

如果你要从一个不同的机器上发送请求到你的应用程序正在运行的机器上，machineName 应该是正在运行应用程序的机器的名称或者 IP 地址。假如你的浏览器在同一台机器上，你可以使用 localhost 作为 machineName。端口是 8080，staticResource 是你需要请求的文件的名称，且必须位于 WEB_ROOT 里边。

举例来说，假如你正在使用同一台计算机上测试应用程序，并且你想要调用 HttpServer 对象去发送一个 index.html 文件，你可以使用一下的 URL：

```
http://localhost:8080/index.html
```

要停止服务器，你可以在 web 浏览器的地址栏或者网址框里边敲入预定义字符串，就在 URL 的 host:port 的后面，发送一个 shutdown 命令。shutdown 命令是在 HttpServer 类的静态 final 变量 SHUTDOWN 里边定义的：

```
private static final String SHUTDOWN_COMMAND = "/SHUTDOWN";
```

因此，要停止服务器，使用下面的 URL：

```
http://localhost:8080/SHUTDOWN
```

现在我们来看看 Listing 1.2 印出来的 await 方法。

使用方法名 await 而不是 wait 是因为 wait 方法是与线程相关的 java.lang.Object 类的一个重要方法。

await 方法首先创建一个 ServerSocket 实例然后进入一个 while 循环。

```
serverSocket = new ServerSocket(port, 1,
    InetAddress.getByName("127.0.0.1"));
...
// Loop waiting for a request
while (!shutdown) {
    ...
}
```

while 循环里边的代码运行到 ServletSocket 的 accept 方法停了下来，只会在 8080 端口接收到一个 HTTP 请求的时候才返回：

```
socket = serverSocket.accept();
```

接收到请求之后，await 方法从 accept 方法返回的 Socket 实例中取得 java.io.InputStream 和 java.io.OutputStream 对象。

```
input = socket.getInputStream();
```

```
output = socket.getOutputStream();
```

await 方法接下去创建一个 ex01.pyrmont.Request 对象并且调用它的 parse 方法去解析 HTTP 请求的原始数据。

```
// create Request object and parse
Request request = new Request(input);
request.parse ();
```

在这之后，await 方法创建一个 Response 对象，把 Request 对象设置给它，并调用它的 sendStaticResource 方法。

```
// create Response object
Response response = new Response(output);
response.setRequest(request);
response.sendStaticResource();
```

最后，await 关闭套接字并调用 Request 的 getUri 来检测 HTTP 请求的 URI 是不是一个 shutdown 命令。假如是的话，shutdown 变量将被设置为 true 且程序会退出 while 循环。

```
// Close the socket
socket.close ();
//check if the previous URI is a shutdown command
shutdown = request.getUri().equals(SHUTDOWN_COMMAND);
```

Request 类

ex01.pyrmont.Request 类代表一个 HTTP 请求。从负责与客户端通信的 Socket 中传递过来 InputStream 对象来构造这个类的一个实例。你调用 InputStream 对象其中一个 read 方法来获取 HTTP 请求的原始数据。

Request 类显示在 Listing 1.3。Request 对象有 parse 和 getUri 两个公共方法，分别在 Listings 1.4 和 1.5 列出来。

Listing 1.3: Request 类

```
package ex01.pyrmont;
import java.io.InputStream;
import java.io.IOException;
public class Request {
    private InputStream input;
    private String uri;
    public Request(InputStream input) {
        this.input = input;
    }
    public void parse() {
        ...
    }
    private String parseUri(String requestString) {
        ...
    }
```

```
    public String getUri() {
        return uri;
    }
}
```

Listing 1.4: Request 类的 parse 方法

```
public void parse() {
    // Read a set of characters from the socket
    StringBuffer request = new StringBuffer(2048);
    int i;
    byte[] buffer = new byte[2048];
    try {
        i = input.read(buffer);
    }
    catch (IOException e) {
        e.printStackTrace();
        i = -1;
    }
    for (int j=0; j<i; j++) {
        request.append((char) buffer[j]);
    }
    System.out.print(request.toString());
    uri = parseUri(request.toString());
}
```

Listing 1.5: Request 类的 parseUri 方法

```
private String parseUri(String requestString) {
    int index1, index2;
    index1 = requestString.indexOf(' ');
    if (index1 != -1) {
        index2 = requestString.indexOf(' ', index1 + 1);
        if (index2 > index1)
        return requestString.substring(index1 + 1, index2);
    }
    return null;
}
```

parse 方法解析 HTTP 请求里边的原始数据。这个方法没有做很多事情。它唯一可用的信息是通过调用 HTTP 请求的私有方法 parseUri 获得的 URI。parseUri 方法在 uri 变量里边存储 URI。公共方法 getUri 被调用并返回 HTTP 请求的 URI。

注意：在第 3 章和下面各章的附带程序里边，HTTP 请求将会对原始数据进行更多的处理。

为了理解 parse 和 parseUri 方法是怎样工作的，你需要知道上一节“超文本传输协议(HTTP)”讨论的 HTTP 请求的结构。在这一章中，我们仅仅关注 HTTP 请求的第一部分，请求行。请求行从一个方法标记开始，接下去是请求的 URI 和协议版本，最后是用回车换行符(CRLF)结束。请求行里边的元素是通过一个空格来分隔的。例如，使用 GET 方法来请求 index.html 文件的请求行如下所示。

GET /index.html HTTP/1.1

parse 方法从传递给 Requst 对象的套接字的 InputStream 中读取整个字节流并在一个缓冲区中存储字节数组。然后它使用缓冲区字节数据的字节来填入一个 StringBuffer 对象，并且把代表 StringBuffer 的字符串传递给 parseUri 方法。

parse 方法列在 Listing 1.4。

然后 parseUri 方法从请求行里边获得 URI。Listing 1.5 给出了 parseUri 方法。parseUri 方法搜索请求里边的第一个和第二个空格并从中获取 URI。

Response 类

ex01.pyrmont.Response 类代表一个 HTTP 响应，在 Listing 1.6 里边给出。

Listing 1.6: Response 类

```
package ex01.pyrmont;
import java.io.OutputStream;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.File;
/*
HTTP Response = Status-Line
*(( general-header | response-header | entity-header ) CRLF)
CRLF
[ message-body ]
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
*/
public class Response {
    private static final int BUFFER_SIZE = 1024;
    Request request;
    OutputStream output;
    public Response(OutputStream output) {
        this.output = output;
    }
    public void setRequest(Request request) {
        this.request = request;
    }
    public void sendStaticResource() throws IOException {
        byte[] bytes = new byte[BUFFER_SIZE];
        FileInputStream fis = null;
        try {
            File file = new File(HttpServer.WEB_ROOT, request.getUri());
            if (file.exists()) {
                fis = new FileInputStream(file);
                int ch = fis.read(bytes, 0, BUFFER_SIZE);
```

```
                while (ch!=-1) {
                    output.write(bytes, 0, ch);
                    ch = fis.read(bytes, 0, BUFFER_SIZE);
                }
            }
            else {
            // file not found
            String errorMessage = "HTTP/1.1 404 File Not Found\r\n" +
                "Content-Type: text/html\r\n" +
                "Content-Length: 23\r\n" +
                "\r\n" +
                "<h1>File Not Found</h1>";
            output.write(errorMessage.getBytes());
            }
        }
        catch (Exception e) {
            // thrown if cannot instantiate a File object
            System.out.println(e.toString() );
        }
        finally {
            if (fis!=null)
            fis.close();
        }
    }
}
```

首先注意到它的构造方法接收一个 java.io.OutputStream 对象，就像如下所示。

```
public Response(OutputStream output) {
    this.output = output;
}
```

响应对象是通过传递由套接字获得的 OutputStream 对象给 HttpServer 类的 await 方法来构造的。Response 类有两个公共方法：setRequest 和 sendStaticResource。setRequest 方法用来传递一个 Request 对象给 Response 对象。

sendStaticResource 方法是用来发送一个静态资源，例如一个 HTML 文件。它首先通过传递上一级目录的路径和子路径给 File 累的构造方法来实例化 java.io.File 类。

```
File file = new File(HttpServer.WEB_ROOT, request.getUri());
```

然后它检查该文件是否存在。假如存在的话，通过传递 File 对象让 sendStaticResource 构造一个 java.io.FileInputStream 对象。然后，它调用 FileInputStream 的 read 方法并把字节数组写入 OutputStream 对象。请注意，这种情况下，静态资源是作为原始数据发送给浏览器的。

```
if (file.exists()) {
    fis = new FileInputstream(file);
```

```
        int ch = fis.read(bytes, 0, BUFFER_SIZE);
        while (ch!=-1) {
            output.write(bytes, 0, ch);
            ch = fis.read(bytes, 0, BUFFER_SIZE);
        }
    }
```

假如文件并不存在，sendStaticResource 方法发送一个错误信息到浏览器。

```
String errorMessage =
    "Content-Type: text/html\r\n" +
    "Content-Length: 23\r\n" +
    "\r\n" +
    "<h1>File Not Found</h1>";
output.write(errorMessage.getBytes());
```

运行应用程序

为了运行应用程序，可以在工作目录下敲入下面的命令：

```
java ex01.pyrmont.HttpServer
```

为了测试应用程序，可以打开你的浏览器并在地址栏或网址框中敲入下面的命令：

```
http://localhost:8080/index.html
```

正如 Figure 1.1 所示，你将会在你的浏览器里边看到 index.html 页面。

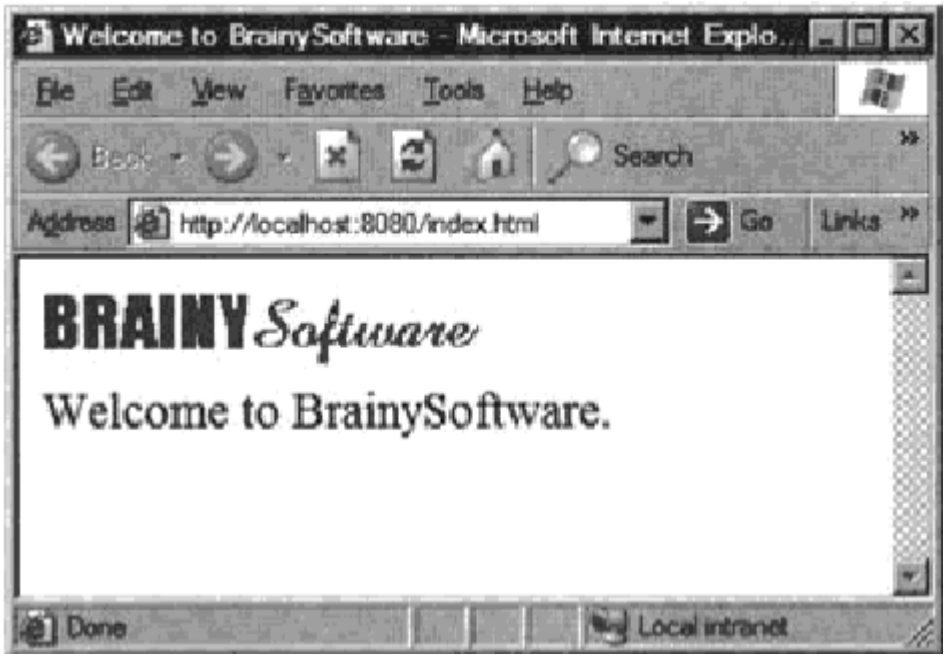


Figure 1.1: web 服务器的输出

在控制台中，你可以看到类似于下面的 HTTP 请求：

```
GET /index.html HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
```

```
application/vnd.ms-excel, application/msword, application/vnd.ms-
powerpoint, application/x-shockwave-flash, application/pdf, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR
1.1.4322)
Host: localhost:8080
Connection: Keep-Alive

GET /images/logo.gif HTTP/1.1
Accept: */*
Referer: http://localhost:8080/index.html
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR
1.1.4322)
Host: localhost:8080
Connection: Keep-Alive
```

总结

在这章中你已经看到一个简单的 web 服务器是如何工作的。这章附带的程序仅仅由三个类组成，并不是全功能的。不过，它提供了一个良好的学习工具。下一章将要讨论动态内容的处理过程。

第 2 章:一个简单的 Servlet 容器

概要

本章通过两个程序来说明你如何开发自己的 servlet 容器。第一个程序被设计得足够简单使得你能理解一个 servlet 容器是如何工作的。然后它演变为第二个稍微复杂的 servlet 容器。

注意：每一个 servlet 容器的应用程序都是从前一章的应用程序逐渐演变过来的，直至一个全功能的 Tomcat servlet 容器在第 17 章被建立起来。

这两个 servlet 容器都可以处理简单的 servlet 和静态资源。你可以使用 PrimitiveServlet 来测试这个容器。PrimitiveServlet 在 Listing 2.1 中列出并且它的类文件可以在 webroot 目录下找到。更复杂的 servlet 就超过这些容器的能力了，但是你将会在以下各章中学到如何建立更复杂的 servlet 容器。

Listing 2.1: PrimitiveServlet.java

```
import javax.servlet.*;
import java.io.IOException;
import java.io.PrintWriter;
public class PrimitiveServlet implements Servlet {
    public void init(ServletConfig config) throws ServletException {
    System.out.println("init");
    }
    public void service(ServletRequest request, ServletResponse response)
        throws ServletException, IOException {
        System.out.println("from service");
        PrintWriter out = response.getWriter();
        out.println("Hello. Roses are red.");
        out.print("Violets are blue.");
    }
    public void destroy() {
        System.out.println("destroy");
    }
    public String getServletInfo() {
        return null;
    }
    public ServletConfig getServletConfig() {
        return null;
    }
}
```

两个应用程序的类都放在 ex02.pyrmont 包里边。为了理解应用程序是如何工作的，你需要熟悉 javax.servlet.Servlet 接口。为了给你复习一下，将会在本章的首节讨论这个接口。在这之后，你将会学习一个 servlet 容器做了什么工作来为一个 servlet 提供 HTTP 请求。

javax.servlet.Servlet 接口

Servlet 编程是通过 javax.servlet 和 javax.servlet.http 这两个包的类和接口来实现的。其中一个至关重要的就是 javax.servlet.Servlet 接口了。所有的 servlet 必须实现实现或者继承实现该接口的类。

Servlet 接口有五个方法，其用法如下。

```
public void init(ServletConfig config) throws ServletException
public void service(ServletRequest request, ServletResponse response)
    throws ServletException, java.io.IOException
public void destroy()
public ServletConfig getServletConfig()
public java.lang.String getServletInfo()
```

在Servlet的五个方法中,init,service和destroy是servlet的生命周期方法。在servlet类已经初始化之后，init方法将会被servlet容器所调用。servlet容器只调用一次，以此表明servlet已经被加载进服务中。init方法必须在servlet可以接受任何请求之前成功运行完毕。一个servlet程序员可以通过覆盖这个方法来写那些仅仅只要运行一次的初始化代码,例如加载数据库驱动，值初始化等等。在其他情况下，这个方法通常是留空的。

servlet容器为servlet请求调用它的service方法。servlet容器传递一个javax.servlet.ServletRequest对象和javax.servlet.ServletResponse对象。ServletRequest对象包括客户端的HTTP请求信息，而ServletResponse对象封装servlet的响应。在servlet的生命周期中，service方法将会给调用多次。

当从服务中移除一个servlet实例的时候，servlet容器调用destroy方法。这通常发生在servlet容器正在被关闭或者servlet容器需要一些空闲内存的时候。仅仅在所有servlet线程的service方法已经退出或者超时淘汰的时候，这个方法才被调用。在servlet容器已经调用完destroy方法之后，在同一个servlet里边将不会再调用service方法。destroy方法提供了一个机会来清理任何已经被占用的资源，例如内存，文件句柄和线程，并确保任何持久化状态和servlet的内存当前状态是同步的。

Listing 2.1介绍了一个名为PrimitiveServlet的servlet的代码，是一个非常简单的的servlet，你可以用来测试本章里边的servlet容器应用程序。PrimitiveServlet类实现了javax.servlet.Servlet(所有的servlet都必须这样做)，并为Servlet的这五个方法都提供了实现。PrimitiveServlet做的事情非常简单。在init，service或者destroy中的任何一个方法每次被调用的时候，servlet把方法名写到标准控制台上面去。另外，service方法从ServletResponse对象获得java.io.PrintWriter实例，并发送字符串到浏览器去。

应用程序1

现在，让我们从一个servlet容器的角度来研究一下servlet编程。总的来说，一个全功能的servlet容器会为servlet的每个HTTP请求做下面一些工作：

- 当第一次调用servlet的时候，加载该servlet类并调用servlet的init方法(仅仅一次)。
- 对每次请求，构造一个javax.servlet.ServletRequest实例和一个javax.servlet.ServletResponse实例。
- 调用servlet的service方法，同时传递ServletRequest和ServletResponse对象。
- 当servlet类被关闭的时候，调用servlet的destroy方法并卸载servlet类。

本章的第一个servlet容器不是全功能的。因此,她不能运行什么除了非常简单的servlet，而且也不调用servlet的init方法和destroy方法。相反它做了下面的事情：

- 等待HTTP请求。

- 构造一个 ServletRequest 对象和一个 ServletResponse 对象。
- 假如该请求需要一个静态资源的话，调用 StaticResourceProcessor 实例的 process 方法，同时传递 ServletRequest 和 ServletResponse 对象。
- 假如该请求需要一个 servlet 的话，加载 servlet 类并调用 servlet 的 service 方法，同时传递 ServletRequest 和 ServletResponse 对象。

注意：在这个 servlet 容器中，每一次 servlet 被请求的时候，servlet 类都会被加载。

第一个应用程序由 6 个类组成：

- HttpServer1
- Request
- Response
- StaticResourceProcessor
- ServletProcessor1
- Constants

Figure 2.1 显示了第一个 servlet 容器的 UML 图。

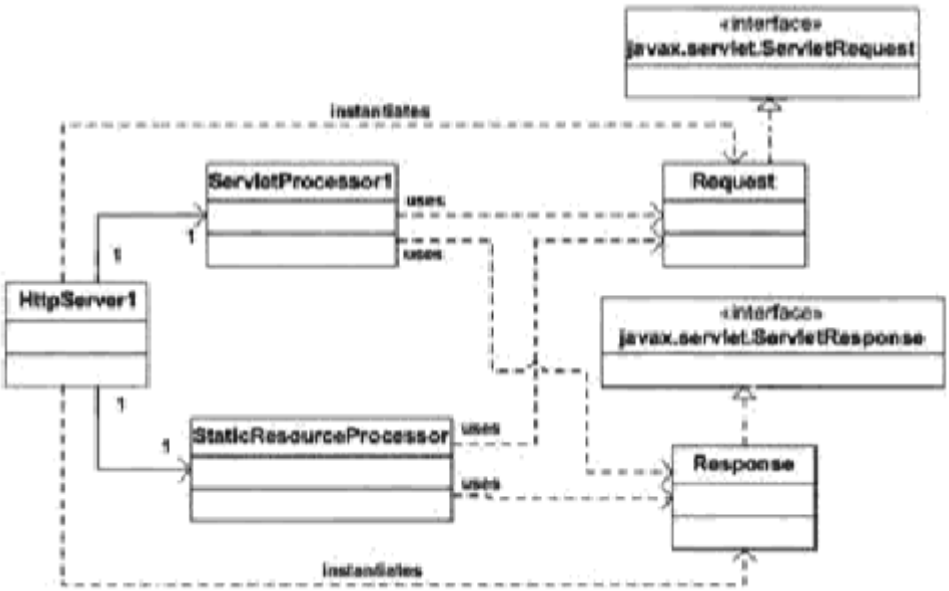


Figure 2.1: 第一个 servlet 容器的 UML 图

这个应用程序的入口点(静态 main 方法)可以在 HttpServer1 类里边找到。main 方法创建了一个 HttpServer1 的实例并调用了它的 await 方法。await 方法等待 HTTP 请求，为每次请求创建一个 Request 对象和一个 Response 对象，并把他们分发到一个 StaticResourceProcessor 实例或者一个 ServletProcessor 实例中去，这取决于请求一个静态资源还是一个 servlet。

Constants 类包括涉及其他类的静态 final 变量 WEB_ROOT。WEB_ROOT 显示了 PrimitiveServlet 和这个容器可以提供的静态资源的位置。

HttpServer1 实例会一直等待 HTTP 请求，直到接收到一个 shutdown 的命令。你科研用第 1 章的做法发送一个 shutdown 命令。

应用程序里边的每个类都会在以下各节中进行讨论。

HttpServer1 类

这个应用程序里边的 HttpServer1 类类似于第 1 章里边的简单服务器应用程序的

HttpServer 类。不过，在这个应用程序里边 HttpServer1 类可以同时提供静态资源和 servlet。要请求一个静态资源，你可以在你的浏览器地址栏或者网址框里边敲入一个 URL：

http://machineName:port/staticResource

就像是在第 1 章提到的，你可以请求一个静态资源。

为了请求一个 servlet，你可以使用下面的 URL：

http://machineName:port/servlet/servletClass

因此，假如你在本地请求一个名为 PrimitiveServlet 的 servlet，你在浏览器的地址栏或者网址框中敲入：

http://localhost:8080/servlet/PrimitiveServlet

servlet 容器可以就提供 PrimitiveServlet 了。不过，假如你调用其他 servlet，如 ModernServlet，servlet 容器将会抛出一个异常。在以下各章中，你将会建立可以处理这两个情况的程序。

HttpServer1 类显示在 Listing 2.2 中。

Listing 2.2: HttpServer1 类的 await 方法

```
package ex02.pyrmont;
import java.net.Socket;
import java.net.ServerSocket;
import java.net.InetAddress;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.IOException;
public class HttpServer1 {
    /** WEB_ROOT is the directory where our HTML and other files reside.
    * For this package, WEB_ROOT is the "webroot" directory under the
    * working directory.
    * The working directory is the location in the file system
    * from where the java command was invoked.
    */
    // shutdown command
    private static final String SHUTDOWN_COMMAND = "/SHUTDOWN";
    // the shutdown command received
    private boolean shutdown = false;
    public static void main(String[] args) {
        HttpServer1 server = new HttpServer1();
        server.await();
    }
    public void await() {
        ServerSocket serverSocket = null;
        int port = 8080;
        try {
            serverSocket = new ServerSocket(port, 1,
```

```
            InetAddress.getByName("127.0.0.1"));
        }
        catch (IOException e) {
            e.printStackTrace();
            System.exit(1);
        }
        // Loop waiting for a request
        while (!shutdown) {
            Socket socket = null;
            InputStream input = null;
            OutputStream output = null;
            try {
                socket = serverSocket.accept();
                input = socket.getInputstream();
                output = socket.getOutputStream();
                // create Request object and parse
                Request request = new Request(input);
                request.parse();
                // create Response object
                Response response = new Response(output);
                response.setRequest(request);
                // check if this is a request for a servlet or
                // a static resource
                // a request for a servlet begins with "/servlet/"
                if (request.getUri().startsWith("/servlet/")) {
                    ServletProcessor1 processor = new ServletProcessor1();
                    processor.process(request, response);
                }
                else {
                    StaticResoureProcessor processor =
                    new StaticResourceProcessor();
                    processor.process(request, response);
                }
                // Close the socket
                socket.close();
                //check if the previous URI is a shutdown command
                shutdown = request.getUri().equals(SHUTDOWN_COMMAND);
            }
            catch (Exception e) {
                e.printStackTrace();
                System.exit(1);
            }
        }
    }
```

}

类的 await 方法等待 HTTP 请求直到一个 shutdown 命令给发出，让你想起第 1 章的 await 方法。Listing 2.2 的 await 方法和第 1 章的区别是，在 Listing 2.2 里边，请求可以分发给一个 StaticResourceProcessor 或者一个 ServletProcessor。假如 URI 包括字符串/servlet/的话，请求将会转发到后面去。

不然的话，请求将会传递给 StaticResourceProcessor 实例 instance. 请注意，这部分在 Listing 2.2 中灰暗显示。

Request 类

servlet 的 service 方法从 servlet 容器中接收一个 javax.servlet.ServletRequest 实例和一个 javax.servlet.ServletResponse 实例。这就是说对于每一个 HTTP 请求，servlet 容器必须构造一个 ServletRequest 对象和一个 ServletResponse 对象并把它们传递给正在服务的 servlet 的 service 方法。

ex02.pyrmont.Request 类代表一个 request 对象并被传递给 servlet 的 service 方法。就本身而言，它必须实现 javax.servlet.ServletRequest 接口。这个类必须提供这个接口所有方法的实现。不过，我们想要让它非常简单并且仅仅提供实现其中一些方法，我们在以下各章中再实现全部的方法。要编译 Request 类，你需要把这些方法的实现留空。假如你看过 Listing 2.3 中的 Request 类，你将会看到那些需要返回一个对象的方法返回了 null

Listing 2.3: Request 类

```
package ex02.pyrmont;
import java.io.InputStream;
import java.io.IOException;
import java.io.BufferedReader;
import java.io.UnsupportedEncodingException;
import java.util.Enumeration;
import java.util.Locale;
import java.util.Map;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletInputStream;
import javax.servlet.ServletRequest;
public class Request implements ServletRequest {
    private InputStream input;
    private String uri;
    public Request(InputStream input){
        this.input = input;
    }
    public String getUri() {
        return uri;
    }
    private String parseUri(String requestString) {
        int index1, index2;
        index1 = requestString.indexOf(' ');
```

```
        if (index1 != -1) {
            index2 = requestString.indexOf(' ', index1 + 1);
            if (index2 > index1)
                return requestString.substring(index1 + 1, index2);
        }
        return null;
    }
    public void parse() {
        // Read a set of characters from the socket
        StringBuffer request = new StringBuffer(2048);
        int i;
        byte[] buffer = new byte[2048];
        try {
            i = input.read(buffer);
        }
        catch (IOException e) {
            e.printStackTrace();
            i = -1;
        }
        for (int j=0; j<i; j++) {
            request.append((char) buffer(j));
        }
        System.out.print(request.toString());
        uri = parseUri(request.toString());
    }
    /* implementation of ServletRequest */
    public Object getAttribute(String attribute) {
        return null;
    }
    public Enumeration getAttributeNames() {
        return null;
    }
    public String getRealPath(String path) {
        return null;
    }
    public RequestDispatcher getRequestDispatcher(String path) {
        return null;
    }
    public boolean isSecure() {
        return false;
    }
    public String getCharacterEncoding() {
        return null;
    }
```

```
public int getContentLength() {
    return 0;
}
public String getContentType() {
    return null;
}
public ServletInputStream getInputStream() throws IOException {
    return null;
}
public Locale getLocale() {
    return null;
}
public Enumeration getLocales() {
    return null;
}
public String getParameter(String name) {
    return null;
}
public Map getParameterMap() {
    return null;
}
public Enumeration getParameterNames() {
    return null;
}
public String[] getParameterValues(String parameter) {
    return null;
}
public String getProtocol() {
    return null;
}
public BufferedReader getReader() throws IOException {
    return null;
}
public String getRemoteAddr() {
    return null;
}
public String getRemoteHost() {
    return null;
}
public String getScheme() {
    return null;
}
public String getServerName() {
    return null;
```

```
    }
    public int getServerPort() {
        return 0;
    }
    public void removeAttribute(String attribute) { }
    public void setAttribute(String key, Object value) { }
    public void setCharacterEncoding(String encoding)
        throws UnsupportedEncodingException { }
}
```

另外，Request 类仍然有在第 1 章中讨论的 parse 和 getUri 方法。

Response 类

在 Listing 2.4 列出的 ex02.pyrmont.Response 类，实现了 javax.servlet.ServletResponse。就本身而言，这个类必须提供接口里边的所有方法的实现。类似于 Request 类，我们把除了 getWriter 之外的所有方法的实现留空。

Listing 2.4: Response 类

```
package ex02.pyrmont;
import java.io.OutputStream;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.File;
import java.io.PrintWriter;
import java.util.Locale;
import javax.servlet.ServletResponse;
import javax.servlet.ServletOutputStream;
public class Response implements ServletResponse {
    private static final int BUFFER_SIZE = 1024;
    Request request;
    OutputStream output;
    PrintWriter writer;
    public Response(OutputStream output) {
        this.output = output;
    }
    public void setRequest(Request request) {
        this.request = request;
    }
    /* This method is used to serve static pages */
    public void sendStaticResource() throws IOException {
        byte[] bytes = new byte[BUFFER_SIZE];
        FileInputstream fis = null;
        try {
            /* request.getUri has been replaced by request.getRequestURI */
```

```
            File file = new File(Constants.WEB_ROOT, request.getUri());
            fis = new FileInputstream(file);
            /*
            HTTP Response = Status-Line
            *(( general-header | response-header | entity-header ) CRLF)
            CRLF
            [ message-body ]
            Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
            */
            int ch = fis.read(bytes, 0, BUFFER_SIZE);
            while (ch!=-1) {
                output.write(bytes, 0, ch);
                ch = fis.read(bytes, 0, BUFFER_SIZE);
            }
        }
        catch (FileNotFoundException e) {
            String errorMessage = "HTTP/1.1 404 File Not Found\r\n" +
                "Content-Type: text/html\r\n" +
                "Content-Length: 23\r\n" +
                "\r\n" +
                "<h1>File Not Found</h1>";
            output.write(errorMessage.getBytes());
        }
        finally {
            if (fis!=null)
            fis.close();
        }
    }
    /** implementation of ServletResponse */
    public void flushBuffer() throws IOException { }
    public int getBufferSize() {
        return 0;
    }
    public String getCharacterEncoding() {
        return null;
    }
    public Locale getLocale() {
        return null;
    }
    public ServletOutputStream getOutputStream() throws IOException {
        return null;
    }
    public PrintWriter getWriter() throws IOException {
        // autoflush is true, println() will flush,
```

```
        // but print() will not.
        writer = new PrintWriter(output, true);
        return writer;
    }
    public boolean isCommitted() {
        return false;
    }
    public void reset() { }
    public void resetBuffer() { }
    public void setBufferSize(int size) { }
    public void setContentLength(int length) { }
    public void setContentType(String type) { }
    public void setLocale(Locale locale) { }
}
```

在 getWriter 方法中，PrintWriter 类的构造方法的第二个参数是一个布尔值表明是否允许自动刷新。传递 true 作为第二个参数将会使任何 println 方法的调用都会刷新输出(output)。不过，print 方法不会刷新输出。

因此，任何 print 方法的调用都会发生在 servlet 的 service 方法的最后一行，输出将不会被发送到浏览器。这个缺点将会在下一个应用程序中修复。

Response 类还拥有在第 1 章中谈到的 sendStaticResource 方法。

StaticResourceProcessor 类

ex02.pyrmont.StaticResourceProcessor 类用来提供静态资源请求。唯一的方法是 process 方法。Listing 2.5 给出了 StaticResourceProcessor 类。

Listing 2.5: StaticResourceProcessor 类

```
package ex02.pyrmont;
import java.io.IOException;
public class StaticResourceProcessor {
    public void process(Request request, Response response) {
    try {
        response.sendStaticResource();
    }
    catch (IOException e) {
        e.printStackTrace();
    }
    }
}
```

process 方法接收两个参数：一个 ex02.pyrmont.Request 实例和一个 ex02.pyrmont.Response 实例。这个方法只是简单的呼叫 Response 对象的 sendStaticResource 方法。

ServletProcessor1 类

Listing 2.6 中的 ex02.pyrmont.ServletProcessor1 类用于处理 servlet 的 HTTP 请求。

Listing 2.6: ServletProcessor1 类

```
package ex02.pyrmont;
import java.net.URL;
import java.net.URLClassLoader;
import java.net.URLStreamHandler;
import java.io.File;
import java.io.IOException;
import javax.servlet.Servlet;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
public class ServletProcessor1 {
    public void process(Request request, Response response) {
        String uri = request.getUri();
        String servletName = uri.substring(uri.lastIndexOf("/") + 1);
        URLClassLoader loader = null;
        try {
            // create a URLClassLoader
            URL[] urls = new URL[1];
            URLStreamHandler streamHandler = null;
            File classPath = new File(Constants.WEB_ROOT);
            // the forming of repository is taken from the
            // createClassLoader method in
            // org.apache.catalina.startup.ClassLoaderFactory
            String repository =(new URL("file", null, classPath.getCanonicalPath() +
                File.separator)).toString() ;
            // the code for forming the URL is taken from
            // the addRepository method in
            // org.apache.catalina.loader.StandardClassLoader.
            urls[0] = new URL(null, repository, streamHandler);
            loader = new URLClassLoader(urls);
        }
        catch (IOException e) {
            System.out.println(e.toString() );
        }
        Class myClass = null;
        try {
            myClass = loader.loadClass(servletName);
        }
        catch (ClassNotFoundException e) {
            System.out.println(e.toString());
```

```
        }
        Servlet servlet = null;
        try {
            servlet = (Servlet) myClass.newInstance();
            servlet.service((ServletRequest) request,
            (ServletResponse) response);
        }
        catch (Exception e) {
            System.out.println(e.toString());
        }
        catch (Throwable e) {
            System.out.println(e.toString());
        }
    }
}
```

ServletProcessor1 类出奇的简单，仅仅由一个方法组成：process。这个方法接受两个参数：一个 javax.servlet.ServletRequest 实例和一个 javax.servlet.ServletResponse 实例。该方法从 ServletRequest 中通过调用 getRequestUri 方法获得 URI：

```
String uri = request.getUri();
```

请记住 URI 是以下形式的：

```
/servlet/servletName
```

在这里 servletName 是 servlet 类的名字。

要加载 servlet 类，我们需要从 URI 中知道 servlet 的名称。我们可以使用 process 方法的下一行来获得 servlet 的名字：

```
String servletName = uri.substring(uri.lastIndexOf("/") + 1);
```

接下去，process 方法加载 servlet。要完成这个，你需要创建一个类加载器并告诉这个类加载器要加载的类的位置。对于这个 servlet 容器，类加载器直接在 Constants 指向的目录里边查找。WEB_ROOT 就是指向工作目录下面的 webroot 目录。

注意： 类加载器将在第 8 章详细讨论。

要加载 servlet，你可以使用 java.net.URLClassLoader 类，它是 java.lang.ClassLoader 类的一个直接子类。一旦你拥有一个 URLClassLoader 实例，你使用它的 loadClass 方法去加载一个 servlet 类。现在举例说明 URLClassLoader 类是 straightforward 直接转发的。这个类有三个构造方法，其中最简单的是：

```
public URLClassLoader(URL[] urls);
```

这里 urls 是一个 java.net.URL 的对象数组，这些对象指向了加载类时候查找的位置。任何以/结尾的 URL 都假设是一个目录。否则，URL 会 Otherwise, the URL 假定是一个将被下载并在需要的时候打开的 JAR 文件。

注意：在一个 servlet 容器里边，一个类加载器可以找到 servlet 的地方被称为资源库(repository)。

在我们的应用程序里边，类加载器必须查找的地方只有一个，如工作目录下面的 webroot 目录。因此，我们首先创建一个单个 URL 组成的数组。URL 类提供了一系列的构造方法，所以有很多中构造一个 URL 对象的方式。对于这个应用程序来说，我们使用 Tomcat 中的另一个类的相同的构造方法。这个构造方法如下所示。

```
public URL(URL context, java.lang.String spec, URLStreamHandler hander)
throws MalformedURLException
```

你可以使用这个构造方法，并为第二个参数传递一个说明，为第一个和第三个参数都传递 null。不过，这里有另外一个接受三个参数的构造方法：

```
public URL(java.lang.String protocol, java.lang.String host,
    java.lang.String file) throws MalformedURLException
```

因此，假如你使用下面的代码时，编译器将不会知道你指的是那个构造方法：

```
new URL(null, aString, null);
```

你可以通过告诉编译器第三个参数的类型来避开这个问题，例如。

```
URLStreamHandler streamHandler = null;
new URL(null, aString, streamHandler);
```

你可以使用下面的代码在组成一个包含资源库(servlet 类可以被找到的地方)的字符串，并作为第二个参数，

```
String repository = (new URL("file", null,
classPath.getCanonicalPath() + File.separator)).toString() ;
```

把所有的片段组合在一起，这就是用来构造适当的 URLClassLoader 实例的 process 方法中的一部分:

```
// create a URLClassLoader
URL[] urls = new URL[1];
URLStreamHandler streamHandler = null;
File classPath = new File(Constants.WEB_ROOT);
String repository = (new URL("file", null,
classPath.getCanonicalPath() + File.separator)).toString() ;
urls[0] = new URL(null, repository, streamHandler);
loader = new URLClassLoader(urls);
```

注意： 用来生成资源库的代码是从 org.apache.catalina.startup.ClassLoaderFactory 的 createClassLoader 方法来的，而生成 URL 的代码是从 org.apache.catalina.loader.StandardClassLoader 的 addRepository 方法来的。不过，在以下各章之前你不需要担心这些类。

当有了一个类加载器，你可以使用 loadClass 方法加载一个 servlet：

```
Class myClass = null;
try {
    myClass = loader.loadClass(servletName);
}
```

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

然后，process 方法创建一个 servlet 类加载器的实例，把它向下转换(downcast)为 javax.servlet.Servlet，并调用 servlet 的 service 方法：

```
Servlet servlet = null;
try {
    servlet = (Servlet) myClass.newInstance();
    servlet.service((ServletRequest) request,(ServletResponse) response);
}
catch (Exception e) {
    System.out.println(e.toString());
}
catch (Throwable e) {
    System.out.println(e.toString());
}
```

运行应用程序

要在 Windows 上运行该应用程序，在工作目录下面敲入以下命令：

```
java -classpath ./lib/servlet.jar;./ ex02.pyrmont.HttpServer1
```

在 Linux 下，你使用一个冒号来分隔两个库：

```
java -classpath ./lib/servlet.jar:./ ex02.pyrmont.HttpServer1
```

要测试该应用程序，在浏览器的地址栏或者网址框中敲入：

```
http://localhost:8080/index.html
```

或者

```
http://localhost:8080/servlet/PrimitiveServlet
```

当调用 PrimitiveServlet 的时候，你将会在你的浏览器看到下面的文本：

```
Hello. Roses are red.
```

请注意,因为只是第一个字符串被刷新到浏览器,所以你不能看到第二个字符串 Violets are blue。我们将在第 3 章修复这个问题。

应用程序 2

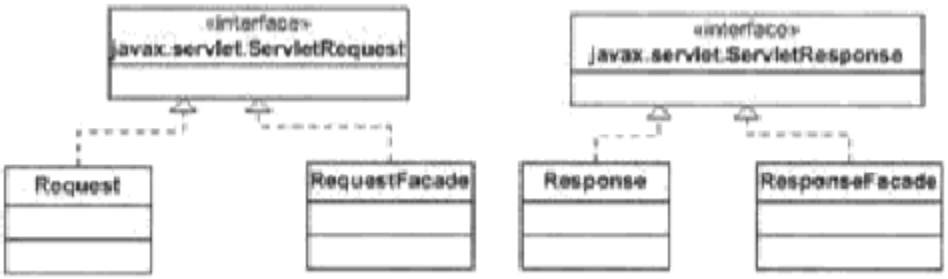
第一个应用程序有一个严重的问题。在 ServletProcessor1 类的 process 方法，你向上转换 ex02.pyrmont.Request 实例为 javax.servlet.ServletRequest，并作为第一个参数传递给 servlet 的 service 方法。你也向下转换 ex02.pyrmont.Response 实例为 javax.servlet.ServletResponse，并作为第二个参数传递给 servlet 的 service 方法。

```
try {
    servlet = (Servlet) myClass.newInstance();
    servlet.service((ServletRequest) request,(ServletResponse) response);
}
```

这会危害安全性。知道这个 servlet 容器的内部运作的 Servlet 程序员可以分别把 ServletRequest 和 ServletResponse 实例向下转换为 ex02.pyrmont.Request 和 ex02.pyrmont.Response，并调用他们的公共方法。拥有一个 Request 实例，它们就可以调用 parse 方法。拥有一个 Response 实例，就可以调用 sendStaticResource 方法。

你不可以把 parse 和 sendStaticResource 方法设置为私有的，因为它们将会被其他的类调用。不过，这两个方法是在个 servlet 内部是不可见的。其中一个解决办法就是让 Request 和 Response 类拥有默认访问修饰，所以它们不能在 ex02.pyrmont 包的外部使用。不过，这里有一个更优雅的解决办法：通过使用 facade 类。请看 Figure 2.2 中的 UML 图。

Figure 2.2: Façade classes



在这第二个应用程序中，我们增加了两个 façade 类：RequestFacade 和 ResponseFacade。RequestFacade 实现了 ServletRequest 接口并通过在构造方法中传递一个引用了 ServletRequest 对象的 Request 实例作为参数来实例化。ServletRequest 接口中每个方法的实现都调用了 Request 对象的相应方法。然而 ServletRequest 对象本身是私有的，并不能在类的外部访问。我们构造了一个 RequestFacade 对象并把它传递给 service 方法，而不是向下转换 Request 对象为 ServletRequest 对象并传递给 service 方法。Servlet 程序员仍然可以向下转换 ServletRequest 实例为 RequestFacade，不过它们只可以访问 ServletRequest 接口里边的公共方法。现在 parseUri 方法就是安全的了。

Listing 2.7 显示了一个不完整的 RequestFacade 类

Listing 2.7: RequestFacade 类

```
package ex02.pyrmont;
public class RequestFacade implements ServletRequest {
    private ServleLRequest request = null;
    public RequestFacade(Request request) {
        this.request = request;
    }
    /* implementation of the ServletRequest*/
    public Object getAttribute(String attribute) {
        return request.getAttribute(attribute);
    }
    public Enumeration getAttributeNames() {
        return request.getAttributeNames();
```

```
    }
    ...
}
```

请注意 RequestFacade 的构造方法。它接受一个 Request 对象并马上赋值给私有的 servletRequest 对象。还请注意，RequestFacade 类的每个方法调用 ServletRequest 对象的相应的方法。

这同样使用于 ResponseFacade 类。

这里是应用程序 2 中使用的类：

- HttpServer2
- Request
- Response
- StaticResourceProcessor
- ServletProcessor2
- Constants

HttpServer2 类类似于 HttpServer1，除了它在 await 方法中使用 ServletProcessor2 而不是 ServletProcessor1：

```
if (request.getUri().startWith("/servlet/")) {
    servletProcessor2 processor = new ServletProcessor2();
    processor.process(request, response);
}
else {
    ...
}
```

ServletProcessor2 类类似于 ServletProcessor1，除了 process 方法中的以下部分：

```
Servlet servlet = null;
RequestFacade requestFacade = new RequestFacade(request);
ResponseFacade responseFacade = new ResponseFacade(response);
try {
    servlet = (Servlet) myClass.newInstance();
    servlet.service((ServletRequest) requestFacade,(ServletResponse)responseFacade);
}
```

运行应用程序

要在 Windows 上运行该应用程序，在工作目录下面敲入以下命令：

```
java -classpath ./lib/servlet.jar;./ ex02.pyrmont.HttpServer2
```

在 Linux 下，你使用一个冒号来分隔两个库：

```
java -classpath ./lib/servlet.jar:./ ex02.pyrmont.HttpServer2
```

你可以使用与应用程序 1 一样的地址，并得到相同的结果。

总结

本章讨论了两个简单的可以用来提供静态资源和处理像 PrimitiveServlet 这么简单的 servlet 的 servlet 容器。同样也提供了关于 javax.servlet.Servlet 接口和相关类型的背景信息。

第 3 章:连接器

概要

在介绍中提到，Catalina 中有两个主要的模块：连接器和容器。本章中你将会写一个可以

创建更好的请求和响应对象的连接器，用来改进第 2 章中的程序。一个符合 Servlet 2.3 和 2.4 规范的连接器必须创建 javax.servlet.http.HttpServletRequest 和 javax.servlet.http.HttpServletResponse，并传递给被调用的 servlet 的 service 方法。在第 2 章中，servlet 容器只可以运行实现了 javax.servlet.Servlet 的 servlet，并传递 javax.servlet.ServletRequest 和 javax.servlet.ServletResponse 实例给 service 方法。因为连接器并不知道 servlet 的类型(例如它是否实现了 javax.servlet.Servlet，继承了 javax.servlet.GenericServlet，或者继承了 javax.servlet.http.HttpServlet)，所以连接器必须始终提供 HttpServletRequest 和 HttpServletResponse 的实例。

在本章的应用程序中，连接器解析 HTTP 请求头部并让 servlet 可以获得头部, cookies, 参数名/值等等。你将会完善第 2 章中 Response 类的 getWriter 方法，让它能够正确运行。由于这些改进，你将会从 PrimitiveServlet 中获取一个完整的响应，并能够运行更加复杂的 ModernServlet。

本章你建立的连接器是将在第 4 章详细讨论的 Tomcat4 的默认连接器的一个简化版本。Tomcat 的默认连接器在 Tomcat4 中是不推荐使用的,但它仍然可以作为一个非常棒的学习工具。在这章的剩余部分，"connector"指的是内置在我们应用程序的模块。

注意：和上一章的应用程序不同的是，本章的应用程序中，连接器和容器是分离的。

本章的应用程序可以在包 ex03.pyrmont 和它的子包中找到。组成连接器的这些类是包 ex03.pyrmont.connector 和 ex03.pyrmont.connector.http 的一部分。在本章的开头，每个附带的程序都有个 bootstrap 类用来启动应用程序。不过，在这个阶段，尚未有一个机制来停止这个应用程序。一旦运行，你必须通过关闭控制台(Windows)或者杀死进程(UNIX/Linux)的方法来鲁 莽的关闭应用程序。

在我们解释该应用程序之前，让我们先来说说包 org.apache.catalina.util 里边的 StringManager 类。这个类用来处理这个程序中不同模块和 Catalina 自身的错误信息的国际化。之后会讨论附带的应用程序。

StringManager 类

一个像 Tomcat 这样的大型应用需要仔细的处理错误信息。在 Tomcat 中，错误信息对于系统管理员和 servlet 程序员都是有用的。例 如，Tomcat 记录错误信息，让系统管理员可以定位发生的任何异常。对 servlet 程序员来说，Tomcat 会在抛出的任何一个 javax.servlet.ServletException 中发送一个错误信息，这样程序员可以知道他/她的 servlet 究竟发送什么错误了。

Tomcat 所采用的方法是在一个属性文件里边存储错误信息，这样，可以容易的修改这些信息。不过，Tomcat 中有数以百计的类。把所有类使用的错误信 息存储到一个大的属性文件里边将会容易产生维护的噩梦。为了避免这一情况，Tomcat 为每个包都分配一个属性文件。例如，在包 org.apache.catalina.connector 里边的属性文件包含了该包所有的类抛出的所有错误信息。每个属性文件都会被一个 org.apache.catalina.util.StringManager 类的实例所处理。当 Tomcat 运行时，将会有许多 StringManager 实例，每个实例会读取包对应的一个属性文件。此外，由于 Tomcat 的受欢迎程度，提供多种语言的错误信息也是有意义的。目前，有三种语言是被支持的。英语的错误信息属性文件名为 LocalStrings.properties。另外两个是西班牙语和日语，分别放在 LocalStrings_es.properties 和 LocalStrings_ja.properties 里边。

当包里边的一个类需要查找放在该包属性文件的一个错误信息时，它首先会获得一个 StringManager 实例。不过，相同包里边的许多类可能也需要 StringManager，为每个对象创建一个 StringManager 实例是一种资源浪费。因此,StringManager 类被设计成一个 StringManager

实例可以被包里边的所有类共享。假如你熟悉设计模式，你将会正确的猜到 StringManager 是一个单例 (singleton)类。仅有的一个构造方法是私有的，所有你不能在类的外部使用 new 关键字来实例化。你通过传递一个包名来调用它的公共静态方法 getManager 来获得一个实例。每个实例存储在一个以包名为键(key)的 Hashtable 中。

```
private static Hashtable managers = new Hashtable();
public synchronized static StringManager
getManager(String packageName) {
    StringManager mgr = (StringManager)managers.get(packageName);
    if (mgr == null) {
        mgr = new StringManager(packageName);
        managers.put(packageName, mgr);
    }
    return mgr;
}
```

注意：一篇关于单例模式的题为"The Singleton Pattern"的文章可以在附带的 ZIP 文件中找到。

例如，要在包 ex03.pyrmont.connector.http 的一个类中使用 StringManager，可以传递包名给 StringManager 类的 getManager 方法：

```
StringManager sm =
    StringManager.getManager("ex03.pyrmont.connector.http");
```

在包 ex03.pyrmont.connector.http 中，你会找到三个属性文件：LocalStrings.properties, LocalStrings_es.properties 和 LocalStrings_ja.properties。StringManager 实例是根据运行程序的服务器的区域设置来决定使用哪个文件的。假如你打开 LocalStrings.properties，非注释的第一行是这样的：

```
httpConnector.alreadyInitialized=HTTP connector has already been initialized
```

要获得一个错误信息，可以使用 StringManager 类的 getString，并传递一个错误代号。这是其中一个重载方法：

```
public String getString(String key)
```

通过传递 httpConnector.alreadyInitialized 作为 getString 的参数，将会返回"HTTP connector has already been initialized"。

应用程序

从本章开始，每章附带的应用程序都会分成模块。这章的应用程序由三个模块组成：connector,
startup 和 core。

startup 模块只有一个类，Bootstrap，用来启动应用的。connector 模块的类可以分为五组：

- 连接器和它的支撑类(HttpConnector 和 HttpProcessor)。
- 指代 HTTP 请求的类(HttpRequest)和它的辅助类。
- 指代 HTTP 响应的类(HttpResponse)和它的辅助类。

- Facade 类(HttpRequestFacade 和 HttpResponseFacade)。
- Constant 类

core 模块由两个类组成：ServletProcessor 和 StaticResourceProcessor。

Figure 3.1 显示了这个应用的类的 UML 图。为了让图更具可读性，HttpRequest 和 HttpResponse 相关的类给省略了。你可以在我们讨论 Request 和 Response 对象的时候分别找到 UML 图。

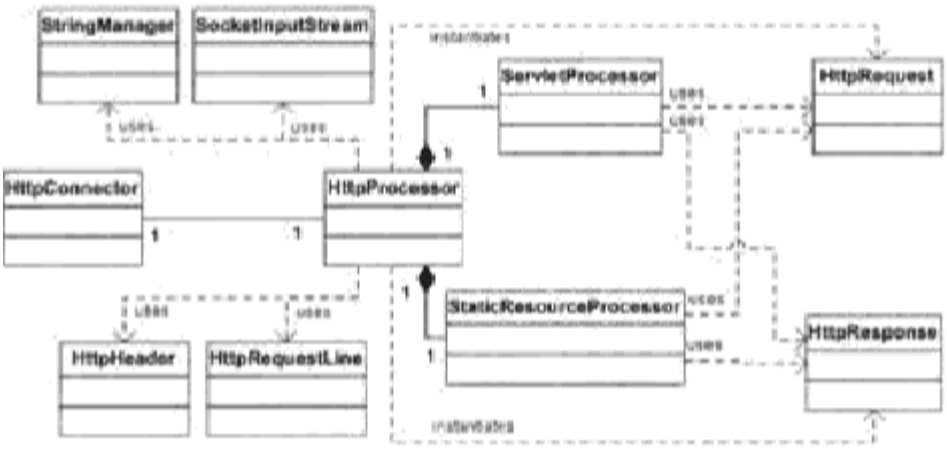


Figure 3.1: 应用程序的 UML 图

和 Figure 2.1 的 UML 图相比，第 2 章中的 HttpServer 类被分离为两个类：HttpConnector 和 HttpProcessor，Request 被 HttpRequest 所取代，而 Response 被 HttpResponse 所取代。同样，本章的应用使用了更多的类。

第 2 章中的 HttpServer 类的职责是等待 HTTP 请求并创建请求和响应对象。在本章的应用中，等待 HTTP 请求的工作交给 HttpConnector 实例，而创建请求和响应对象的工作交给了 HttpProcessor 实例。

本章中,HTTP 请求对象由实现了 javax.servlet.http.HttpServletRequest 的 HttpRequest 类来代表。一个 HttpRequest 对象将会给转换为一个 HttpServletRequest 实例并传递给被调用的 servlet 的 service 方法。因此，每个 HttpRequest 实例必须适当增加字段，以便 servlet 可以使用它们。值需要赋给 HttpRequest 对象，包括 URI，查询字符串，参数，cookies 和其他的头部等等。因为连接器并不知道被调用的 servlet 需要哪个值，所以连接器必须从 HTTP 请求中解析所有可获得的值。不过，解析一个 HTTP 请求牵涉昂贵的字符串和其他操作，假如只是解析 servlet 需要的值的话，连接器就能节省许多 CPU 周期。例如，假如 servlet 不 解析任何一个请求参数(例如不调用 javax.servlet.http.HttpServletRequest 的 getParameter, getParameterMap,getParameterNames 或者 getParameterValues 方法)，连接器就不需要从查询字符串或者 HTTP 请求内容中解析这些参数。Tomcat 的默认连接器(和本章应用程序的连接器)试图不解析参数直到 servlet 真正需要它的时候，通过这样来获得更高效率。

Tomcat 的默认连接器和我们的连接器使用 SocketInputStream 类来从套接字的 InputStream 中读取字节流。一个 SocketInputStream 实例对从套接字的 getInputStream 方法中返回的 java.io.InputStream 实例进行包装。 SocketInputStream 类提供了两个重要的方法：readRequestLine 和 readHeader。readRequestLine 返回一个 HTTP 请求的第一行。例如，这行包括了 URI，方法和 HTTP 版本。因为从套接字的输入流中处理字节流意味着只读取一次，从第一个字节到最后一个字节(并且不回退)，因此 readHeader 被调用之前，readRequestLine 必须只被调用一次。readHeader 每次被调用来获得一个头部的名/值对，并且应该被重复的调用知道所有的头部被读取到。readRequestLine 的返回值是一个 HttpRequestLine 的实例，而 readHeader 的返回值是一个 HttpHeader 对象。我们将在下节中讨论类 HttpRequestLine 和

HttpHeader。

HttpProcessor 对象创建了 HttpRequest 的实例，因此必须在它们当中增加字段。HttpProcessor 类使用它的 parse 方法 来解析一个 HTTP 请求中的请求行和头部。解析出来并把值赋给 HttpProcessor 对象的这些字段。不过，parse 方法并不解析请求内容或者请求 字符串里边的参数。这个任务留给了 HttpRequest 对象它们。只是当 servlet 需要一个参数时，查询字符串或者请求内容才会被解析。

另一个跟上一个应用程序比较的改进是用来启动应用程序的 bootstrap 类 ex03.pyrmont.startup.Bootstrap 的出现。

我们将会在下面的子节里边详细说明该应用程序：

- 启动应用程序
- 连接器
- 创建一个 HttpRequest 对象
- 创建一个 HttpResponse 对象
- 静态资源处理器和 servlet 处理器
- 运行应用程序

启动应用程序

你可以从 ex03.pyrmont.startup.Bootstrap 类来启动应用程序。这个类在 Listing 3.1 中给出。

Listing 3.1: Bootstrap 类

```
package ex03.pyrmont.startup;
import ex03.pyrmont.connector.http.HttpConnector;
public final class Bootstrap {
    public static void main(String[] args) {
        HttpConnector connector = new HttpConnector();
        connector.start();
    }
}
```

Bootstrap 类中的 main 方法实例化 HttpConnector 类并调用它的 start 方法。HttpConnector 类在 Listing 3.2 给出。

Listing 3.2: HttpConnector 类的 start 方法

```
package ex03.pyrmont.connector.http;
import java.io.IOException;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
public class HttpConnector implements Runnable {
    boolean stopped;
    private String scheme = "http";
    public String getScheme() {
        return scheme;
```

```
    }
    public void run() {
        ServerSocket serverSocket = null;
        int port = 8080;
        try {
            serverSocket = new
            ServerSocket(port, 1, InetAddress.getByName("127.0.0.1"));
        }
        catch (IOException e) {
            e.printStackTrace();
            System.exit(1);
        }
        while (!stopped) {
            // Accept the next incoming connection from the server socket
            Socket socket = null;
            try {
                socket = serverSocket.accept();
            }
            catch (Exception e) {
                continue;
            }
            // Hand this socket off to an HttpProcessor
            HttpProcessor processor = new HttpProcessor(this);
            processor.process(socket);
        }
    }
    public void start() {
        Thread thread = new Thread(this);
        thread.start ();
    }
}
```

连接器

ex03.pyrmont.connector.http.HttpConnector 类指代一个连接器，职责是创建一个服务器套接字用来等待前来的 HTTP 请求。这个类在 Listing 3.2 中出现。

HttpConnector 类实现了 java.lang.Runnable，所以它能被它自己的线程专用。当你启动应用程序，一个 HttpConnector 的实例被创建，并且它的 run 方法被执行。

注意： 你可以通过读"Working with Threads"这篇文章来提醒你自己怎样创建 Java 线程。

run 方法包括一个 while 循环，用来做下面的事情：

- 等待 HTTP 请求
- 为每个请求创建个 HttpProcessor 实例
- 调用 HttpProcessor 的 process 方法

注意：run 方法类似于第 2 章中 HttpServer1 类的 await 方法。

马上你就会看到 HttpConnector 类和 ex02.pyrmont.HttpServer1 类非常相像，除了从 java.net.ServerSocket 类的 accept 方法中获得一个套接字之后，一个 HttpProcessor 实例会被创建，并且通过传递该套 接字给它的 process 方法调用。

注意：HttpConnector 类有另一个方法叫 getScheme，用来返回一个 scheme(HTTP)。

HttpProcessor 类的 process 方法接受前来的 HTTP 请求的套接字，会做下面的事情：

1. 创建一个 HttpRequest 对象。
2. 创建一个 HttpResponse 对象。
3. 解析 HTTP 请求的第一行和头部，并放到 HttpRequest 对象。
4. 解析 HttpRequest 和 HttpResponse 对象到一个 ServletProcessor 或者 StaticResourceProcessor。像第 2 章里边说的，ServletProcessor 调用被请求的 servlet 的 service 方 法，而 StaticResourceProcessor 发送一个静态资源的内容。

process 方法在 Listing 3.3 给出.

Listing 3.3: HttpProcessor 类 process 方法

```
public void process(Socket socket) {
    SocketInputStream input = null;
    OutputStream output = null;
    try {
        input = new SocketInputStream(socket.getInputStream(), 2048);
        output = socket.getOutputStream();
        // create HttpRequest object and parse
        request = new HttpRequest(input);
        // create HttpResponse object
        response = new HttpResponse(output);
        response.setRequest(request);
        response.setHeader("Server", "Pyrmont Servlet Container");
        parseRequest(input, output);
        parseHeaders(input);
        //check if this is a request for a servlet or a static resource
        //a request for a servlet begins with "/servlet/"
        if (request.getRequestURI().startsWith("/servlet/")) {
            ServletProcessor processor = new ServletProcessor();
            processor.process(request, response);
        }
        else {
            StaticResourceProcessor processor = new
            StaticResourceProcessor();
            processor.process(request, response);
        }
        // Close the socket
        socket.close();
        // no shutdown for this application
    }
    catch (Exception e) {
```

```
        e.printStackTrace ();
    }
}
```

process 首先获得套接字的输入流和输出流。请注意，在这个方法中，我们适合继承了java.io.InputStream 的 SocketInputStream 类。

```
SocketInputStream input = null;
OutputStream output = null;
try {
    input = new SocketInputStream(socket.getInputStream(), 2048);
    output = socket.getOutputStream();
```

然后，它创建一个 HttpRequest 实例和一个 instance and an HttpResponse instance and assigns
the HttpRequest to the HttpResponse.

```
// create HttpRequest object and parse
request = new HttpRequest(input);
// create HttpResponse object
response = new HttpResponse(output);
response.setRequest(request);
```

本章应用程序的 HttpResponse 类要比第 2 章中的 Response 类复杂得多。举例来说，你可以通过调用他的 setHeader 方法来发送头部到一个客户端。

```
response.setHeader("Server", "Pyrmont Servlet Container");
```

接下去，process 方法调用 HttpProcessor 类中的两个私有方法来解析请求。

```
parseRequest(input, output);
parseHeaders (input);
```

然后，它根据请求 URI 的形式把 HttpRequest 和 HttpResponse 对象传给 ServletProcessor 或者 StaticResourceProcessor 进行处理。

```
if (request.getRequestURI().startsWith("/servlet/")) {
    ServletProcessor processor = new ServletProcessor();
    processor.process(request, response);
}
else {
    StaticResourceProcessor processor =
        new StaticResourceProcessor();
    processor.process(request, response);
}
```

最后，它关闭套接字。

```
socket.close();
```

也要注意的是，HttpProcessor 类使用 org.apache.catalina.util.StringManager 类来发

送错误信息：

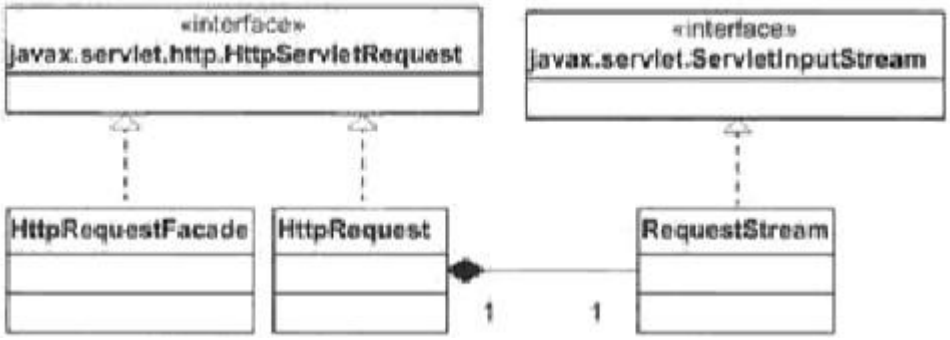
```
protected StringManager sm =
    StringManager.getManager("ex03.pyrmont.connector.http");
```

HttpProcessor 类中的私有方法--parseRequest，parseHeaders 和 normalize，是用来帮助填充 HttpRequest 的。这些方法将会在下节"创建一个 HttpRequest 对象"中进行讨论。

创建一个 HttpRequest 对象

HttpRequest 类实现了 javax.servlet.http.HttpServletRequest。跟随它的是一个叫做 HttpRequestFacade 的 facade 类。Figure 3.2 显示了 HttpRequest 类和它的相关类的 UML 图。

Figure 3.2: HttpRequest 类和它的相关类



HttpRequest 类的很多方法都留空(你需要等到第 4 章才会完全实现)，但是 servlet 程序员已经可以从到来的 HTTP 请求中获得头部，cookies 和参数。这三种类型的值被存储在下面几个引用变量中：

```
protected HashMap headers = new HashMap();
protected ArrayList cookies = new ArrayList();
protected ParameterMap parameters = null;
```

注意：ParameterMap 类将会在“获取参数”这节中解释。

因此，一个 servlet 程序员可以从 javax.servlet.http.HttpServletRequest 中的下列方法中取得正确的返回 值：getCookies,getDateHeader,getHeader, getHeaderNames, getHeaders, getParameter, getPrameterMap,getParameterNames 和 getParameterValues 。就像你在 HttpRequest 类中看到的一样，一旦你取得了头部，cookies 和填充了正确的值的参数，相关的方法的实现是很简单的。

不用说，这里主要的挑战是解析 HTTP 请求和填充 HttpRequest 类。对于头部和 cookies，HttpRequest 类提供了 addHeader 和 addCookie 方法用于 HttpProcessor 的 parseHeaders 方法调用。当需要的时候，会使用 HttpRequest 类的 parseParameters 方法来解析参数。在本节中所有的方法都会被讨论。

因为 HTTP 请求的解析是一项相当复杂的任务，所以本节会分为以下几个小节：

- 读取套接字的输入流
- 解析请求行
- 解析头部
- 解析 cookies

- 获取参数

读取套接字的输入流

在第 1 章和第 2 章中,你在 ex01.pyrmont.HttpRequest 和 ex02.pyrmont.HttpRequest 类中做了一点请求解析。你通过调用 java.io.InputStream类的 read 方法获取了请求行,包括方法,URI 和 HTTP 版本:

```
byte[] buffer = new byte [2048];
try {
    // input is the InputStream from the socket.
    i = input.read(buffer);
}
```

你没有试图为那两个应用程序去进一步解析请求。不过，在本章的应用程序中，你拥有 ex03.pyrmont.connector.http.SocketInputStream 类，这是 org.apache.catalina.connector.http.SocketInputStream 的一个拷贝。这个类提供了方法不仅用来获取请求行，还有请求头部。

你通过传递一个 InputStream 和一个指代实例使用的缓冲区大小的整数，来构建一个 SocketInputStream 实例。在本章中，你在 ex03.pyrmont.connector.http.HttpProcessor 的 process 方法中创建了一个 SocketInputStream 对象，就像下面的代码片段一样：

```
SocketInputStream input = null;
OutputStream output = null;
try {
    input = new SocketInputStream(socket.getInputStream(), 2048);
    ...
```

就像前面提到的一样,拥有一个 SocketInputStream 是为了两个重要方法:readRequestLine 和 readHeader。请继续往下阅读。

解析请求行

HttpProcessor 的 process 方法调用私有方法 parseRequest 用来解析请求行,例如一个 HTTP 请求的第一行。这里是一个请求行的例子：

```
GET /myApp/ModernServlet?userName=tarzan&password=pwd HTTP/1.1
```

请求行的第二部分是 URI 加上一个查询字符串。在上面的例子中，URI 是这样的：

```
/myApp/ModernServlet
```

另外，在问好后面的任何东西都是查询字符串。因此，查询字符串是这样的：

```
userName=tarzan&password=pwd
```

查询字符串可以包括零个或多个参数。在上面的例子中，有两个参数名/值对，userName/tarzan 和 password/pwd。在 servlet/JSP 编程中，参数名 jsessionid 是用来携带一个会话标识符。会话标识符经常被作为 cookie 来嵌入，但是程序员可以选择把它嵌入到查询字

符串去，例如，当浏览器的 cookie 被禁用的时候。

当 parseRequest 方法被 HttpProcessor 类的 process 方法调用的时候，request 变量指向一个 HttpRequest 实例。parseRequest 方法解析请求行用来获得几个值并把这些值赋给 HttpRequest 对象。现在，让我们来关注一下在 Listing 3.4 中的 parseRequest 方法。

Listing 3.4：HttpProcessor 类中的 parseRequest 方法

```
private void parseRequest(SocketInputStream input, OutputStream output)
throws IOException, ServletException {
    // Parse the incoming request line
    input.readRequestLine(requestLine);
    String method =
        new String(requestLine.method, 0, requestLine.methodEnd);
    String uri = null;
    String protocol = new String(requestLine.protocol, 0,
    requestLine.protocolEnd);
    // Validate the incoming request line
    if (method, length () < 1) {
        throw new ServletException("Missing HTTP request method");
    }
    else if (requestLine.uriEnd < 1) {
        throw new ServletException("Missing HTTP request URI");
    }
    // Parse any query parameters out of the request URI
    int question = requestLine.indexOf("?");
    if (question >= 0) {
        request.setQueryString(new String(requestLine.uri, question + 1,
        requestLine.uriEnd - question - 1));
        uri = new String(requestLine.uri, 0, question);
    }
    else {
        request.setQueryString(null);
        uri = new String(requestLine.uri, 0, requestLine.uriEnd);
    }
    // Checking for an absolute URI (with the HTTP protocol)
    if (!uri.startsWith("/")) {
        int pos = uri.indexOf("://");
        // Parsing out protocol and host name
        if (pos != -1) {
            pos = uri.indexOf('/', pos + 3);
            if (pos == -1) {
                uri = "";
            }
            else {
                uri = uri.substring(pos);
            }
```

```
        }
    }
    // Parse any requested session ID out of the request URI
    String match = ";jsessionid=";
    int semicolon = uri.indexOf(match);
    if (semicolon >= 0) {
        String rest = uri.substring(semicolon + match.length());
        int semicolon2 = rest.indexOf(';');
        if (semicolon2 >= 0) {
            request.setRequestedSessionId(rest.substring(0, semicolon2));
            rest = rest.substring(semicolon2);
        }
        else {
            request.setRequestedSessionId(rest);
            rest = "";
        }
        request.setRequestedSessionURL(true);
        uri = uri.substring(0, semicolon) + rest;
    }
    else {
        request.setRequestedSessionId(null);
        request.setRequestedSessionURL(false);
    }
    // Normalize URI (using String operations at the moment)
    String normalizedUri = normalize(uri);
    // Set the corresponding request properties
    ((HttpRequest) request).setMethod(method);
    request.setProtocol(protocol);
    if (normalizedUri != null) {
        ((HttpRequest) request).setRequestURI(normalizedUri);
    }
    else {
        ((HttpRequest) request).setRequestURI(uri);
    }
    if (normalizedUri == null) {
        throw new ServletException("Invalid URI: " + uri + "'");
    }
}
```

parseRequest 方法首先调用 SocketInputStream 类的 readRequestLine 方法：

```
input.readRequestLine(requestLine);
```

在这里 requestLine 是 HttpProcessor 里边的 HttpRequestLine 的一个实例：

```
private HttpRequestLine requestLine = new HttpRequestLine();
```

调用它的 readRequestLine 方法来告诉 SocketInputStream 去填入 HttpRequestLine 实例。

接下去，parseRequest 方法获得请求行的方法，URI 和协议：

```
String method =
    new String(requestLine.method, 0, requestLine.methodEnd);
String uri = null;
String protocol = new String(requestLine.protocol, 0, requestLine.protocolEnd);
```

不过，在 URI 后面可以有查询字符串，假如存在的话，查询字符串会被一个问好分隔开来。因此，parseRequest 方法试图首先获取查询字符串。并调用 setQueryString 方法来填充 HttpRequest 对象：

```
// Parse any query parameters out of the request URI
int question = requestLine.indexOf("?");
if (question >= 0) { // there is a query string.
    request.setQueryString(new String(requestLine.uri, question + 1,
    requestLine.uriEnd - question - 1));
    uri = new String(requestLine.uri, 0, question);
}
else {
    request.setQueryString (null);
    uri = new String(requestLine.uri, 0, requestLine.uriEnd);
}
```

不过，大多数情况下，URI 指向一个相对资源，URI 还可以是一个绝对值，就像下面所示：

```
http://www.brainysoftware.com/index.html?name=Tarzan
```

parseRequest 方法同样也检查这种情况：

```
// Checking for an absolute URI (with the HTTP protocol)
if (!uri.startsWith("/")) {
    // not starting with /, this is an absolute URI
    int pos = uri.indexOf("://");
    // Parsing out protocol and host name
    if (pos != -1) {
        pos = uri.indexOf('/', pos + 3);
        if (pos == -1) {
            uri = "";
        }
        else {
            uri = uri.substring(pos);
        }
    }
}
```

然后，查询字符串也可以包含一个会话标识符，用 jsessionid 参数名来指代。因此，parseRequest 方法也检查一个会话标识符。假如在查询字符串里边找到 jessionid，方法就取得

会话标识符，并通过调用 setRequestedSessionId 方法把值交给 HttpRequest 实例：

```
// Parse any requested session ID out of the request URI
String match = ";jsessionid=";
int semicolon = uri.indexOf(match);
if (semicolon >= 0) {
    String rest = uri.substring(semicolon + match.length());
    int semicolon2 = rest.indexOf(';');
    if (semicolon2 >= 0) {
        request.setRequestedSessionId(rest.substring(0, semicolon2));
        rest = rest.substring(semicolon2);
    }
    else {
        request.setRequestedSessionId(rest);
        rest = "";
    }
    request.setRequestedSessionURL (true);
    uri = uri.substring(0, semicolon) + rest;
}
else {
    request.setRequestedSessionId(null);
    request.setRequestedSessionURL(false);
}
```

当 jsessionid 被找到，也意味着会话标识符是携带在查询字符串里边，而不是在 cookie 里边。因此，传递 true 给 request 的 setRequestSessionURL 方法。否则，传递 false 给 setRequestSessionURL 方法并传递 null 给 setRequestedSessionURL 方法。

到这个时候，uri 的值已经被去掉了 jsessionid。

接下去，parseRequest 方法传递 uri 给 normalize 方法，用于纠正“异常”的 URI。例如，任何\的出现都会给/替代。假如 uri 是正确的格式或者异常可以给纠正的话，normalize 将会返回相同的或者被纠正后的 URI。假如 URI 不能纠正的话，它将会给认为是非法的并且通常会返回 null。在这种情况下(通常返回 null)，parseRequest 将会在方法的最后抛出一个异常。

最后，parseRequest 方法设置了 HttpRequest 的一些属性：

```
((HttpRequest) request).setMethod(method);
request.setProtocol(protocol);
if (normalizedUri != null) {
    ((HttpRequest) request).setRequestURI(normalizedUri);
}
else {
    ((HttpRequest) request).setRequestURI(uri);
}
```

还有，假如 normalize 方法的返回值是 null 的话，方法将会抛出一个异常：

```
if (normalizedUri == null) {
```

```
    throw new ServletException("Invalid URI: " + uri + "'");
}
```

解析头部

一个 HTTP 头部是用类 HttpHeader 来代表的。这个类将会在第 4 章详细解释，而现在知道下面的内容就足够了：

- 你可以通过使用类的无参数构造方法构造一个 HttpHeader 实例。
- 一旦你拥有一个 HttpHeader 实例,你可以把它传递给 SocketInputStream 的 readHeader 方法。假如这里有头部需要读取，readHeader 方法将会相应的填充 HttpHeader 对象。假如再也没有头部需要读取了,HttpHeader 实例的 nameEnd 和 valueEnd 字段将会置零。
- 为了获取头部的名称和值，使用下面的方法：
- String name = new String(header.name, 0, header.nameEnd);
- String value = new String(header.value, 0, header.valueEnd);

parseHeaders 方法包括一个 while 循环用于持续的从 SocketInputStream 中读取头部，直到再也没有头部出现为止。循环从构建一个 HttpHeader 对象开始，并把它传递给类 SocketInputStream 的 readHeader 方法：

```
HttpHeader header = new HttpHeader();
// Read the next header
input.readHeader(header);
```

然后,你可以通过检测 HttpHeader 实例的 nameEnd 和 valueEnd 字段来测试是否可以从输入流中读取下一个头部信息：

```
if (header.nameEnd == 0) {
    if (header.valueEnd == 0) {
        return;
    }
    else {
        throw                                                    new
ServletException              (sm.getString("httpProcessor.parseHeaders.colon"));
    }
}
```

假如存在下一个头部，那么头部的名称和值可以通过下面方法进行检索：

```
String name = new String(header.name, 0, header.nameEnd);
String value = new String(header.value, 0, header.valueEnd);
```

一旦你获取到头部的名称和值,你通过调用 HttpRequest 对象的 addHeader 方法来把它加入 headers 这个 HashMap 中：

```
request.addHeader(name, value);
```

一些头部也需要某些属性的设置。例如，当 servlet 调用 javax.servlet.ServletRequest 的 getContentLength 方法的时候,content-length 头部的值将被返回。而包含 cookies 的 cookie 头部将会给添加到 cookie 集合中。就这样，下面是其中一些过程：

```
if (name.equals("cookie")) {
    ... // process cookies here
}
else if (name.equals("content-length")) {
    int n = -1;
    try {
        n = Integer.parseInt (value);
    }
    catch (Exception e) {
        throw new ServletException(sm.getString(
            "httpProcessor.parseHeaders.contentLength"));
    }
    request.setContentLength(n);
}
else if (name.equals("content-type")) {
    request.setContentType(value);
}
```

Cookie 的解析将会在下一节“解析 Cookies”中讨论。

解析 Cookies

Cookies 是作为一个 Http 请求头部通过浏览器来发送的。这样一个头部名为"cookie"并且它的值是一些 cookie 名/值对。这里是一个包括两个 cookie:username 和 password 的 cookie 头部的例子。

```
Cookie: userName=budi; password=pwd;
```

Cookie 的解析是通过类 org.apache.catalina.util.RequestUtil 的 parseCookieHeader 方法来处理的。这个方法接受 cookie 头部并返回一个 javax.servlet.http.Cookie 数组。数组内的元素数量和头部里边的 cookie 名/值对个数是一样的。parseCookieHeader 方法在 Listing 3.5 中列出。

Listing 3.5: The org.apache.catalina.util.RequestUtil class's parseCookieHeader method

```
public static Cookie[] parseCookieHeader(String header) {
    if ((header == null) || (header.length 0 < 1) )
        return (new Cookie[0]);
    ArrayList cookies = new ArrayList();
    while (header.length() > 0) {
        int semicolon = header.indexOf(';');
        if (semicolon < 0)
            semicolon = header.length();
        if (semicolon == 0)
            break;
        String token = header.substring(0, semicolon);
```

```
        if (semicolon < header.length())
            header = header.substring(semicolon + 1);
        else
            header = "";
        try {
            int equals = token.indexOf('=');
            if (equals > 0) {
                String name = token.substring(0, equals).trim();
                String value = token.substring(equals+1).trim();
                cookies.add(new Cookie(name, value));
            }
        }
        catch (Throwable e) {
            ;
        }
    }
    return ((Cookie[]) cookies.toArray (new Cookie [cookies.size ()]));
}
```

还有，这里是 HttpProcessor 类的 parseHeader 方法中用于处理 cookie 的部分代码:

```
else if (header.equals(DefaultHeaders.COOKIE_NAME)) {
    Cookie cookies[] = RequestUtil.ParseCookieHeader (value);
    for (int i = 0; i < cookies.length; i++) {
        if (cookies[i].getName().equals("jsessionid")) {
            // Override anything requested in the URL
            if (!request.isRequestedSessionIdFromCookie()) {
                // Accept only the first session id cookie
                request.setRequestedSessionId(cookies[i].getValue());
                request.setRequestedSessionCookie(true);
                request.setRequestedSessionURL(false);
            }
        }
        request.addCookie(cookies[i]);
    }
}
```

获取参数

你不需要马上解析查询字符串或者 HTTP 请求内容，直到 servlet 需要通过调用 javax.servlet.http.HttpServletRequest 的 getParameter, getParameterMap, getParameterNames 或者 getParameterValues 方法来读取参数。因此，HttpRequest 的这四个方法开头调用了 parseParameter 方法。

这些参数只需要解析一次就够了，因为假如参数在请求内容里边被找到的话，参数解析将会使得 SocketInputStream 到达字节流的尾部。类 HttpRequest 使用一个布尔变量 parsed 来指示是否已经解析过了。

参数可以在查询字符串或者请求内容里边找到。假如用户使用GET方法来请求servlet的话，所有的参数将在查询字符串里边出现。假如使用POST方法的话，你也可以在请求内容中找到一些。所有的名/值对将会存储在一个HashMap里边。Servlet程序员可以以Map的形式获得参数(通过调用HttpServletRequest的getParameterMap方法)和参数名/值。There is a catch, though. Servlet 程序员不被允许修改参数值。因此，将使用一个特殊的 HashMap：org.apache.catalina.util.ParameterMap。

类ParameterMap继承java.util.HashMap，并使用了一个布尔变量locked。当locked是false的时候，名/值对仅仅可以添加，更新或者移除。否则，异常IllegalStateException会抛出。而随时都可以读取参数值。

类ParameterMap将会在Listing 3.6中列出。它覆盖了方法用于增加，更新和移除值。那些方法仅仅在locked为false的时候可以调用。

Listing 3.6: The org.apache.Catalina.util.ParameterMap class.

```
package org.apache.catalina.util;
import java.util.HashMap;
import java.util.Map;
public final class ParameterMap extends HashMap {
    public ParameterMap() {
        super ();
    }
    public ParameterMap(int initialCapacity) {
        super(initialCapacity);
    }
    public ParameterMap(int initialCapacity, float loadFactor) {
        super(initialCapacity, loadFactor);
    }
    public ParameterMap(Map map) {
        super(map);
    }
    private boolean locked = false;
    public boolean isLocked() {
        return (this.locked);
    }
    public void setLocked(boolean locked) {
        this.locked = locked;
    }
    private static final StringManager sm =
        StringManager.getManager("org.apache.catalina.util");
    public void clear() {
        if (locked)
            throw new IllegalStateException
                (sm.getString("parameterMap.locked"));
        super.clear();
    }
```

```
    public Object put(Object key, Object value) {
        if (locked)
            throw new IllegalStateException
                (sm.getString("parameterMap.locked"));
        return (super.put(key, value));
    }
    public void putAll(Map map) {
        if (locked)
            throw new IllegalStateException
                (sm.getString("parameterMap.locked"));
        super.putAll(map);
    }
    public Object remove(Object key) {
        if (locked)
            throw new IllegalStateException
                (sm.getString("parameterMap.locked"));
        return (super.remove(key));
    }
}
```

现在，让我们来看 parseParameters 方法是怎么工作的。

因为参数可以存在于查询字符串或者 HTTP 请求内容中，所以 parseParameters 方法会检查查询字符串和请求内容。一旦解析过后，参数将会在对象变量 parameters 中找到，所以方法的开头会检查 parsed 布尔变量，假如已经解析过的话，parsed 将会返回 true。

```
if (parsed)
return;
```

然后，parseParameters 方法创建一个名为 results 的 ParameterMap 变量，并指向 parameters。假如
parameters 为 null 的话，它将创建一个新的 ParameterMap。

```
ParameterMap results = parameters;
if (results == null)
    results = new ParameterMap();
```

然后，parseParameters 方法打开 parameterMap 的锁以便写值。

```
results.setLocked(false);
```

下一步，parseParameters 方法检查字符编码，并在字符编码为 null 的时候赋予默认字符编码。

```
String encoding = getCharacterEncoding();
if (encoding == null)
    encoding = "ISO-8859-1";
```

然后，parseParameters 方法尝试解析查询字符串。解析参数是使用 org.apache.Catalina.util.RequestUtil 的 parseParameters 方法来处理的。

```
// Parse any parameters specified in the query string
String queryString = getQueryString();
try {
    RequestUtil.parseParameters(results, queryString, encoding);
}
catch (UnsupportedEncodingException e) {
    ;
}
```

接下来，方法尝试查看 HTTP 请求内容是否包含参数。这种情况发生在当用户使用 POST 方法发送请求的时候，内容长度大于零，并且内容类型是 application/x-www-form-urlencoded 的时候。所以，这里是解析请求内容的代码：

```
// Parse any parameters specified in the input stream
String contentType = getContentType();
if (contentType == null)
    contentType = "";
int semicolon = contentType.indexOf(';');
if (semicolon >= 0) {
    contentType = contentType.substring (0, semicolon).trim();
}
else {
    contentType = contentType.trim();
}
if ("POST".equals(getMethod()) && (getContentLength() > 0)
    && "application/x-www-form-urlencoded".equals(contentType)) {
    try {
        int max = getContentLength();
        int len = 0;
        byte buf[] = new byte[getContentLength()];
        ServletInputStream is = getInputStream();
        while (len < max) {
            int next = is.read(buf, len, max - len);
            if (next < 0 ) {
                break;
            }
            len += next;
        }
        is.close();
        if (len < max) {
            throw new RuntimeException("Content length mismatch");
        }
        RequestUtil.parseParameters(results, buf, encoding);
    }
```

```
    catch (UnsupportedEncodingException ue) {
        ;
    }
    catch (IOException e) {
        throw new RuntimeException("Content read fail");
    }
}
```

最后，parseParameters 方法锁定 ParameterMap，设置 parsed 为 true，并把 results 赋予 parameters。

```
// Store the final results
results.setLocked(true);
parsed = true;
parameters = results;
```

创建一个 HttpResponse 对象

HttpResponse 类实现了 javax.servlet.http.HttpServletResponse。跟随它的是一个叫做 HttpResponseFacade 的 façade 类。Figure 3.3 显示了 HttpResponse 类和它的相关类的 UML 图。

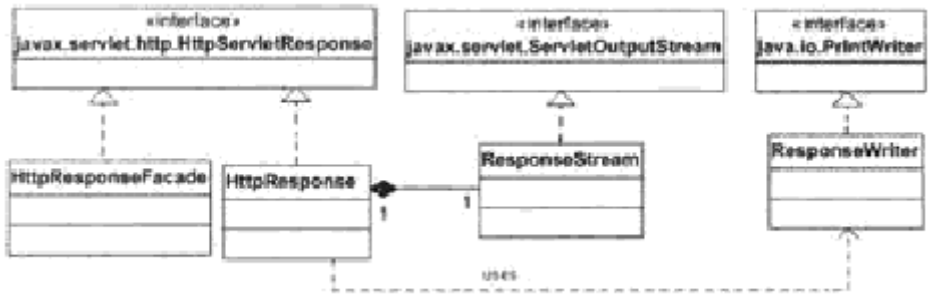


Figure 3.3: The HttpResponse class and related classes

在第 2 章中，你使用的是一个部分实现的 HttpResponse 类。例如，它的 getWriter 方法，在它的其中一个 print 方法被调用的时候，返回一个不会自动清除的 java.io.PrintWriter 对象。在本章中应用程序将会修复这个问题。为了理解它是如何修复的，你需要知道 Writer 是什么东西来的。

在一个 servlet 里边，你使用 PrintWriter 来写字节。你可以使用任何你希望的编码，但是这些字节将会以字节流的形式发送到浏览器去。因此，第 2 章中 ex02.pyrmont.HttpResponse 类的 getWriter 方法就不奇怪了：

```
public PrintWriter getWriter() {
// if autoflush is true, println() will flush,
// but print() will not.
// the output argument is an OutputStream
    writer = new PrintWriter(output, true);
    return writer;
}
```

请看，我们是如何构造一个 PrintWriter 对象的?就是通过传递一个 java.io.OutputStream 实例来实现的。你传递给 PrintWriter 的 print 或 println 方法的任何东西都是通过底下的 OutputStream 进行发送的。

在本章中，你为 PrintWriter 使用 ex03.pyrmont.connector.ResponseStream 类的一个实例来替代
OutputStream。需要注意的是，类 ResponseStream 是间接的从类 java.io.OutputStream 传递过去的。

同样的你使用了继承于 PrintWriter 的类 ex03.pyrmont.connector.ResponseWriter。类 ResponseWriter 覆盖了所有的 print 和 println 方法，并且让这些方法的任何调用把输出自动清除到底下的
OutputStream 去。因此，我们使用一个带底层 ResponseStream 对象的 ResponseWriter 实例。

我们可以通过传递一个 ResponseStream 对象实例来初始化类 ResponseWriter。然而，我们使用一个 java.io.OutputStreamWriter 对象充当 ResponseWriter 对象和 ResponseStream 对象之间的桥梁。

通过 OutputStreamWriter，写进去的字符通过一种特定的字符集被编码成字节。这种字符集可以使用名字来设定，或者明确给出，或者使用平台可接受的默认字符集。write 方法的每次调用都会导致在给定的字符上编码转换器的调用。在写入底层的输出流之前，生成的字节都会累积到一个缓冲区中。缓冲区的大小可以自己设定，但是对大多数场景来说，默认的就足够大了。注意的是，传递给 write 方法的字符是没有被缓冲的。

因此，getWriter 方法如下所示：

```
public PrintWriter getWriter() throws IOException {
    ResponseStream newStream = new ResponseStream(this);
    newStream.setCommit(false);
    OutputStreamWriter osr =
        new OutputStreamWriter(newStream, getCharacterEncoding());
    writer = new ResponseWriter(osr);
    return writer;
}
```

静态资源处理器和 Servlet 处理器

类 ServletProcessor 类似于第 2 章中的类 ex02.pyrmont.ServletProcessor。它们都只有一个方法：process。然而 ex03.pyrmont.connector.ServletProcessor 中的 process 方法接受一个 HttpRequest 和
HttpResponse，代替了 Requese 和 Response 实例。下面是本章中 process 的方法签名：

```
public void process(HttpRequest request, HttpResponse response) {
```

另外，process 方法使用 HttpRequestFacade 和 HttpResponseFacade 作为
request 和 response 的 facade 类。另外，在调用了 servlet 的 service 方法之后，它调用了类 HttpResponse 的
finishResponse 方法。

```
servlet = (Servlet) myClass.newInstance();
HttpRequestFacade requestPacade = new HttpRequestFacade(request);
HttpResponseFacade responseFacade = new HttpResponseFacade(response);
servlet.service(requestFacade, responseFacade);
((HttpResponse) response).finishResponse();
```

类 StaticResourceProcessor 几乎等同于类 ex02.pyrmont.StaticResourceProcessor。

运行应用程序

要在 Windows 上运行该应用程序，在工作目录下面敲入以下命令：

```
java -classpath ./lib/servlet.jar;./ ex03.pyrmont.startup.Bootstrap
```

在 Linux 下，你使用一个冒号来分隔两个库：

```
java -classpath ./lib/servlet.jar:./ ex03.pyrmont.startup.Bootstrap
```

要显示 index.html，使用下面的 URL:

```
http://localhost:8080/index.html
```

要调用 PrimitiveServlet，让浏览器指向下面的 URL：

```
http://localhost:8080/servlet/PrimitiveServlet
```

在你的浏览器中将会看到下面的内容：

```
Hello. Roses are red.
Violets are blue.
```

注意：在第 2 章中运行 PrimitiveServlet 不会看到第二行。

你也可以调用 ModernServet，在第 2 章中它不能运行在 servlet 容器中。下面是相应的 URL：

```
http://localhost:8080/servlet/ModernServlet
```

注意：ModernServlet 的源代码在工作目录的 webroot 文件夹可以找到。

你可以加上一个查询字符串到 URL 中去测试 servlet。加入你使用下面的 URL 来运行 ModernServlet 的话，将显示 Figure 3.4 中的运行结果。

```
http://localhost:8080/servlet/ModernServlet?userName=tarzan&password=pwd
```

Figure 3.4: Running ModernServlet

总结

在本章中，你已经知道了连接器是如何工作的。建立起来的连接器是 Tomcat4 的默认连接器的简化版本。正如你所知道的，因为默认连接器并不高效，所以已经被弃用了。例如，所有的 HTTP 请求头部都被解析了，即使它们没有在 servlet 中使用过。因此，默认连接器很慢，并且已经被 Coyote 所代替了。Coyote 是一个更快的连接器，它的源代码可以在 Apache 软件基金会的网站中下载。不管怎样，默认连接器作为一个优秀的学习工具，将会在第 4 章中详细讨论。

第四章:Tomcat 的默认连接器

概要

第 3 章的连接器运行良好，可以完善以获得更好的性能。但是，它只是作为一个教育工具，设计来介绍 Tomcat4 的默认连接器用的。理解第 3 章中的连接器是理解 Tomcat4 的默认连接器的关键所在。现在，在第 4 章中将通过剖析 Tomcat4 的默认连接器的代码，讨论需要什么来创建一个真实的 Tomcat 连接器。

注意:本章中提及的“默认连接器”是指 Tomcat4 的默认连接器。即使默认的连机器已经被弃用，被更快的，代号为 Coyote 的连接器所代替，它仍然是一个很好的学习工具。

Tomcat 连接器是一个可以插入 servlet 容器的独立模块，已经存在相当多的连接器了，包括 Coyote, mod_jk, mod_jk2 和 mod_webapp。一个 Tomcat 连接器必须符合以下条件：

1. 必须实现接口 org.apache.catalina.Connector。
2. 必须创建请求对象，该请求对象的类必须实现接口 org.apache.catalina.Request。
3. 必须创建响应对象，该响应对象的类必须实现接口 org.apache.catalina.Response。

Tomcat4 的默认连接器类似于第 3 章的简单连接器。它等待前来的 HTTP 请求，创建 request 和 response 对象，然后把 request 和 response 对象传递给容器。连接器是通过调用接口 org.apache.catalina.Container 的 invoke 方法来传递 request 和 response 对象的。invoke 的方法签名如下所示：

```
public void invoke(
    org.apache.catalina.Request request,
    org.apache.catalina.Response response);
```

在 invoke 方法里边，容器加载 servlet，调用它的 service 方法，管理会话，记录出错日志等等。

默认连接器同样使用了一些第 3 章中的连接器未使用的优化。首先就是提供一个各种各样对象的对象池用于避免昂贵对象的创建。接着，在很多地方使用字节数组来代替字符串。

本章中的应用程序是一个和默认连接器管理的简单容器。然而，本章的焦点不是简单容器而是默认连接器。我们将会在第 5 章中讨论容器。不管怎样，为了展示如何使用默认连接器，将会在接近本章末尾的“简单容器的应用程序”一节中讨论简单容器。

另一个需要注意的是默认连接器除了提供 HTTP0.9 和 HTTP1.0 的支持外，还实现了 HTTP1.1 的所有新特性。为了理解 HTTP1.1 中的新特性，你首先需要理解本章首节解释的这些新特性。在这之后，我们将会讨论接口
org.apache.catalina.Connector 和如何创建请求和响应对象。假如你理解第 3 章中连接器如何工作的话，那么在理解默认连接器的时候你应该不会遇到任何问题。

本章首先讨论 HTTP1.1 的三个新特性。理解它们是理解默认连接器内部工作机制的关键所在。然后，介绍所有连接器都会实现的接口 org.apache.catalina.Connector。你会发现第 3 章中遇到的那些类，例如 HttpConnector, HttpProcessor 等等。不过，这个时候，它们比第 3 章那些类似的要高级些。

HTTP 1.1 新特性

本节解释了 HTTP1.1 的三个新特性。理解它们是理解默认连接器如何处理 HTTP 请求的关键。

持久连接

在 HTTP1.1 之前，无论什么时候浏览器连接到一个 web 服务器，当请求的资源被发送之后，连接就被服务器关闭了。然而，一个互联网网页包括其他资源， 例如图片文件，applet 等等。因此，当一个页面被请求的时候,浏览器同样需要下载页面所引用到的资源。加入页面和它所引用到的全部资源使用不同连接来 下载的话，进程将会非常慢。那就是为什么 HTTP1.1 引入持久连接的原因了。使用持久连接的时候，当页面下载的时候，服务器并不直接关闭连接。相反，它等待 web 客户端请求页面所引用的全部资源。这种情况下，页面和所引用的资源使用同一个连接来下载。考虑建立和解除 HTTP 连接的宝贵操作的话，这就为 web 服务器，客户端和网络节省了许多工作和时间。

持久连接是 HTTP1.1 的默认连接方式。同样，为了明确这一点，浏览器可以发送一个值为 keep-alive 的请求头部 connection:

```
connection: keep-alive
```

块编码

建立持续连接的结果就是，使用同一个连接，服务器可以从不同的资源发送字节流，而客户端可以使用发送多个请求。结果就是，发送方必须为每个请求或响应发送 内容长度的头部，以便接收方知道如何解释这些字节。然而，大部分的情况是发送方并不知道将要发送多少个字节。例如，在开头一些字节已经准备好的时 候，servlet 容器就可以开始发送响应了，而不会等到所有都准备好。这意味着，在 content-length 头部不能提前知道的情况下，必须有一种 方式来告诉接收方如何解释字节流。

即使不需要发送多个请求或者响应，服务器或者客户端也不需要知道将会发送多少数据。在 HTTP1.0 中，服务器可以仅仅省略 content-length 头部，并保持写入连接。当写入完成的时候，它将简单的关闭连接。在这种情况下，客户端将会保持读取状态，直到获取到-1，表示已经到达文件的尾部。

HTTP1.1 使用一个特别的头部 transfer-encoding 来表示有多少以块形式的字节流将会被发送。对每块来说，在数据之前，长度(十六进 制)后面接着 CR/LF 将被发送。整个事务通过一个

零长度的块来标识。假设你想用 2 个块发送以下 38 个字节，第一个长度是 29，第二个长度是 9。

I'm as helpless as a kitten up a tree.

你将这样发送：

1D\r\n

I'm as helpless as a kitten u

9\r\n

p a tree.

0\r\n

1D,是 29 的十六进制，指示第一块由 29 个字节组成。0\r\n 标识这个事务的结束。

状态 100(持续状态)的使用

在发送请求内容之前，HTTP 1.1 客户端可以发送 Expect: 100-continue 头部到服务器，并等待服务器的确认。这个一般发生在当客户端需要发送一份长的请求内容而未能确保服务器愿意接受它的时候。如果你 发送一份长的请求内容仅仅发现服务器拒绝了它，那将是一种浪费来的。

当接受到 Expect: 100-continue 头部的时候，假如乐意或者可以处理请求的话，服务器响应 100-continue 头部，后边跟着两对 CRLF 字符。

HTTP/1.1 100 Continue

接着，服务器应该会继续读取输入流。

Connector 接口

Tomcat 连接器必须实现 org.apache.catalina.Connector 接口。在这个接口的众多方法中，最重要的是 getContainer,setContainer, createRequest 和 createResponse。

setContainer 是用来关联连接器和容器用的。getContainer 返回关联的容器。createRequest 为前来的 HTTP 请求构造一个请求对象，而 createResponse 创建一个响应对象。

类 org.apache.catalina.connector.http.HttpConnector 是 Connector 接口的一个实现，将会在下一 节“HttpConnector 类”中讨论。现在，仔细看一下 Figure 4.1 中的默认连接器的 UML 类图。注意的是，为了保持图的简单化，Request 和 Response 接口的实现被省略了。除了 SimpleContainer 类，org.apache.catalina 前缀也同样从类型名中被省略了。

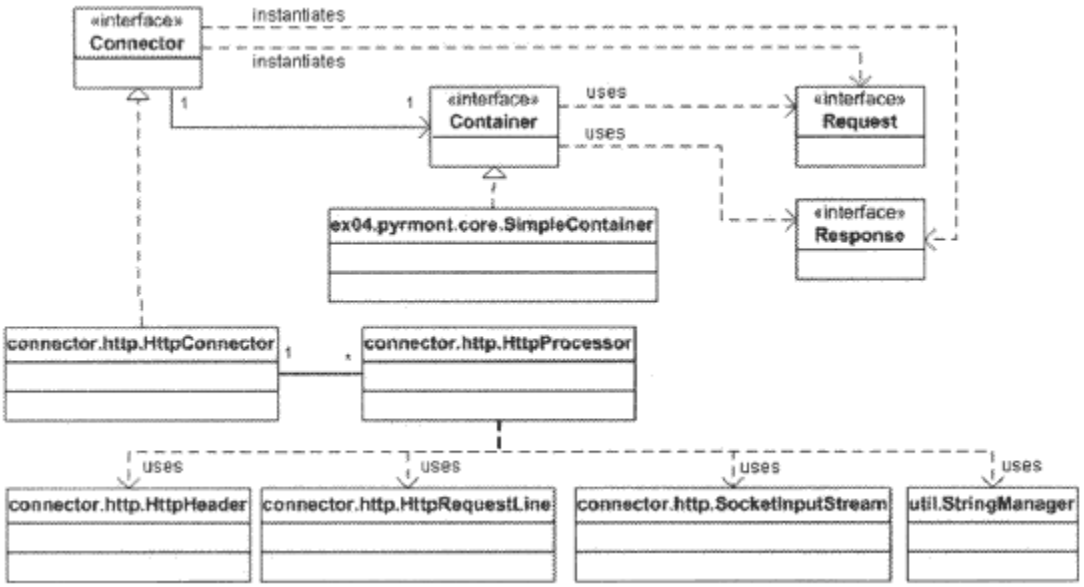


Figure 4.1: The default connector class diagram

因此，Connector 需要被 org.apache.catalina.Connector,util.StringManager org.apache.catalina.util.StringManager 等等访问到。

一个 Connector 和 Container 是一对一的关系。箭头的方向显示出 Connector 知道 Container 但反过来就不成立了。同样需要注意的是，不像第 3 章的是，HttpConnector 和 HttpProcessor 是一对多的关系。

HttpConnector 类

由于在第 3 章中 org.apache.catalina.connector.http.HttpConnector 的简化版本已经被解释过了，所以你已 经知道这个类是怎样的了。它实现了 org.apache.catalina.Connector (为了和 Catalina 协调), java.lang.Runnable （因此它的实例可以运行在自己的线程上）和 org.apache.catalina.Lifecycle。接口 Lifecycle 用来维护每个已经实现它的 Catalina 组件的生命周期。

Lifecycle 将在第 6 章中解释，现在你不需要担心它，只要明白通过实现 Lifecycle,在你创建 HttpConnector 实例之后，你应该 调用它的 initialize 和 start 方法。这两个方法在组件的生命周期里必须只调用一次。我们将看看和第 3 章的 HttpConnector 类的那些 不同方面:HttpConnector 如何创建一个服务器套接字，它如何维护一个 HttpProcessor 对象池，还有它如何处理 HTTP 请求。

创建一个服务器套接字

HttpConnector 的 initialize 方法调用 open 这个私有方法，返回一个 java.net.ServerSocket 实例，并把它赋予 serverSocket。然而，不是调用 java.net.ServerSocket 的构造方法，open 方法是从一个服务端套接字工厂中获得一个 ServerSocket 实例。如果你想知道这工厂的详细信息，可以阅读包 org.apache.catalina.net 里边的接口 ServerSocketFactory 和类 DefaultServerSocketFactory。它们是很容易理解的。

维护 HttpProcessor 实例

在第 3 章中，HttpConnector 实例一次仅仅拥有一个 HttpProcessor 实例，所以每次只能处理一个 HTTP 请求。在默认连接器 中，HttpConnector 拥有一个 HttpProcessor 对象池，每个 HttpProcessor 实例拥有一个独立线程。因 此，HttpConnector 可以同时处理多个 HTTP 请求。

HttpConnector 维护一个 HttpProcessor 的实例池,从而避免每次创建 HttpProcessor 实例。这些 HttpProcessor 实例是存放在一个叫 processors 的 java.io.Stack 中：

```
private Stack processors = new Stack();
```

在 HttpConnector 中,创建的 HttpProcessor 实例数量是有两个变量决定的:minProcessors 和 maxProcessors。默认情况下，minProcessors 为 5 而 maxProcessors 为 20，但是你可以通过 setMinProcessors 和 setMaxProcessors 方法来改变他们的值。

```
protected int minProcessors = 5;
private int maxProcessors = 20;
```

开始的时候，HttpConnector 对象创建 minProcessors 个 HttpProcessor 实例。如果一次有比 HtppProcessor 实例更多的请求需要处理时,HttpConnector 创建更多的 HttpProcessor 实例，直到实例数量达到 maxProcessors 个。在到 达这点之后，仍不够 HttpProcessor 实例的话，请

来的请求将会给忽略掉。如果你想让 HttpConnector 继续创建 HttpProcessor 实例的话，把 maxProcessors 设置为一个负数。还有就是变量 curProcessors 保存了 HttpProcessor 实例的当前数量。

下面是类 HttpConnector 的 start 方法里边关于创建初始数量的 HttpProcessor 实例的代码：

```
while (curProcessors < minProcessors) {
    if ((maxProcessors > 0) && (curProcessors >= maxProcessors))
        break;
    HttpProcessor processor = newProcessor();
    recycle(processor);
}
```

newProcessor 方法构造一个 HttpProcessor 对象并增加 curProcessors。recycle 方法把 HttpProcessor 队会栈。

每个 HttpProcessor 实例负责解析 HTTP 请求行和头部，并填充请求对象。因此，每个实例关联着一个请求对象和响应对象。类 HttpProcessor 的构造方法包括了类 HttpConnector 的 createRequest 和 createResponse 方法的调用。

为 HTTP 请求服务

就像第 3 章一样，HttpConnector 类在它的 run 方法中有其主要的逻辑。run 方法在一个服务端套接字等待 HTTP 请求的地方存在一个 while 循环，一直运行直至 HttpConnector 被关闭了。

```
while (!stopped) {
    Socket socket = null;
    try {
        socket = serverSocket.accept();
        ...
```

对每个前来的 HTTP 请求，会通过调用私有方法 createProcessor 获得一个 HttpProcessor 实例。

```
HttpProcessor processor = createProcessor();
```

然而，大部分时候 createProcessor 方法并不创建一个新的 HttpProcessor 对象。相反，它从池子中获取一个。如果在栈中已经存在一 个 HttpProcessor 实例，createProcessor 将弹出一个。如果栈是空的并且没有超过 HttpProcessor 实例的最大数 量，createProcessor 将会创建一个。然而，如果已经达到最大数量的话，createProcessor 将会返回 null。出现这样的情况的 话，套接字将会简单关闭并且前来的 HTTP 请求不会被处理。

```
if (processor == null) {
    try {
        log(sm.getString("httpConnector.noProcessor"));
        socket.close();
    }
    ...
    continue;
```

如果 createProcessor 不是返回 null，客户端套接字会传递给 HttpProcessor 类的 assign 方法：

```
processor.assign(socket);
```

现在就是 HttpProcessor 实例用于读取套接字的输入流和解析 HTTP 请求的工作了。重要的一点是，assign 方法不会等到 HttpProcessor 完成解析工作，而是必须马上返回，以便下一个

前来的 HTTP 请求可以被处理。每个 HttpProcessor 实例有自己的线程 用于解析，所以这点不是很难做到。你将会在下节“HttpProcessor 类”中看到是怎么做的。

HttpProcessor 类

默认连接器中的 HttpProcessor 类是第 3 章中有着类似名字的类的全功能版本。你已经学习了它是如何工作的,在本章中,我们很有兴趣知道 HttpProcessor 类怎样让 assign 方法异步化，这样 HttpProcessor 实例就可以同时间为很多 HTTP 请求服务了。

注意： HttpProcessor 类的另一个重要方法是私有方法 process，它是用于解析 HTTP 请求和调用容器的 invoke 方法的。我们将会在本章稍后部分的“处理请求”一节中看到它。

在第 3 章中，HttpConnector 在它自身的线程中运行。但是，在处理下一个请求之前，它必须等待当前处理的 HTTP 请求结束。下面是第 3 章中 HttpProcessor 类的 run 方法的部分代码：

```
public void run() {
    ...
    while (!stopped) {
        Socket socket = null;
        try {
            socket = serversocket.accept();
        }
        catch (Exception e) {
            continue;
        }
        // Hand this socket off to an Httpprocessor
        HttpProcessor processor = new Httpprocessor(this);
        processor.process(socket);
    }
}
```

第 3 章中的 HttpProcessor 类的 process 方法是同步的。因此，在接受另一个请求之前，它的 run 方法要等待 process 方法运行结束。

在默认连接器中，然而，HttpProcessor 类实现了 java.lang.Runnable 并且每个 HttpProcessor 实例运行在称作处理器线程(processor thread)的自身线程上。对 HttpConnector 创建的每个 HttpProcessor 实例，它的 start 方法将被调用，有效的启动了 HttpProcessor 实例的处理线程。Listing 4.1 展示了默认处理器中的 HttpProcessor 类的 run 方法：

Listing 4.1: The HttpProcessor class's run method.

```
public void run() {
    // Process requests until we receive a shutdown signal
    while (!stopped) {
        // Wait for the next socket to be assigned
        Socket socket = await();
        if (socket == null)
            continue;
        // Process the request from this socket
        try {
            process(socket);
```

```
        }
        catch (Throwable t) {
            log("process.invoke", t);
        }
        // Finish up this request
        connector.recycle(this);
    }
    // Tell threadStop() we have shut ourselves down successfully
    synchronized (threadSync) {
        threadSync.notifyAll();
    }
}
```

run 方法中的 while 循环按照这样的循序进行：获取一个套接字，处理它，调用连接器的 recycle 方法把当前的 HttpProcessor 实例推回栈。这里是 HttpConenctor 类的 recycle 方法：

```
void recycle(HttpProcessor processor) {
    processors.push(processor);
}
```

需要注意的是，run 中的 while 循环在 await 方法中结束。await 方法持有处理线程的控制流，直到从 HttpConnector 中获取到一个新的套接字。用另外一种说法就是，直到 HttpConnector 调用 HttpProcessor 实例的 assign 方法。但是，await 方法和 assign 方 法运行在不同的线程上。assign 方法从 HttpConnector 的 run 方法中调用。我们就说这个线程是 HttpConnector 实例的 run 方法运行的处理线程。assign 方法是如何通知已经被调用的 await 方法的？就是通过一个布尔变量 available 并且使用 java.lang.Object 的 wait 和 notifyAll 方法。

注意：wait 方法让当前线程等待直到另一个线程为这个对象调用 notify 或者 notifyAll 方法为止。

这里是 HttpProcessor 类的 assign 和 await 方法：

```
synchronized void assign(Socket socket) {
    // Wait for the processor to get the previous socket
    while (available) {
        try {
            wait();
        }
        catch (InterruptedException e) {
        }
    }
    // Store the newly available Socket and notify our thread
    this.socket = socket;
    available = true;
    notifyAll();
    ...
}
private synchronized Socket await() {
    // Wait for the Connector to provide a new Socket
    while (!available) {
```

```
        try {
            wait();
        }
        catch (InterruptedException e) {
        }
    }
    // Notify the Connector that we have received this Socket
    Socket socket = this.socket;
    available = false;
    notifyAll();
    if ((debug >= 1) && (socket != null))
    log(" The incoming request has been awaited");
    return (socket);
}
```

两个方法的程序流向在 Table 4.1 中总结。

Table 4.1: Summary of the await and assign method

The processor thread (the await method)	The connector thread (the assign method)
while (!available) {	while (available) {
wait();	wait();
}	}
Socket socket = this.socket;	this.socket = socket;
available = false;	available = true;
notifyAll();	notifyAll();
return socket; // to the run	...
// method	

刚开始的时候，当处理器线程刚启动的时候，available 为 false，线程在 while 循环里边等待(见 Table 4.1 的第 1 列)。它将等待另一个线程调用 notify 或 notifyAll。这就是说，调用 wait 方法让处理器线程暂停，直到连接器线程调用 HttpProcessor 实例的 notifyAll 方法。

现在，看看第 2 列，当一个新的套接字被分配的时候，连接器线程调用 HttpProcessor 的 assign 方法。available 的值是 false，所以 while 循环给跳过，并且套接字给赋值给 HttpProcessor 实例的 socket 变量：

```
this.socket = socket;
```

连接器线程把 available 设置为 true 并调用 notifyAll。这就唤醒了处理器线程，因为 available 为 true，所以程序控制跳出 while 循环：把实例的 socket 赋值给一个本地变量，并把 available 设置为 false，调用 notifyAll，返回最后需要进行处理的 socket。

为什么 await 需要使用一个本地变量(socket)而不是返回实例的 socket 变量呢？因为这样一来，在当前 socket 被完全处理之前，实例的 socket 变量可以赋给下一个前来的 socket。

为什么 await 方法需要调用 notifyAll 呢？这是为了防止在 available 为 true 的时候另一个 socket 到来。在这种情况下，连接器线程将会在 assign 方法的 while 循环中停止，直到接收到处理器线程的 notifyAll 调用。

请求对象

默认连接器哩变得 HTTP 请求对象指代 org.apache.catalina.Request 接口。这个接口被类 RequestBase 直接实现了，也是 HttpRequest 的父接口。最终的实现是继承于 HttpRequest 的

HttpRequestImpl。像第 3 章一样，有几个 facade 类：RequestFacade 和 HttpRequestFacade。Request 接口和它的实现类的 UML 图在 Figure 4.2 中给出。注意的是，除了属于 javax.servlet 和 javax.servlet.http 包的类，前缀 org.apache.catalina 已经被省略了。

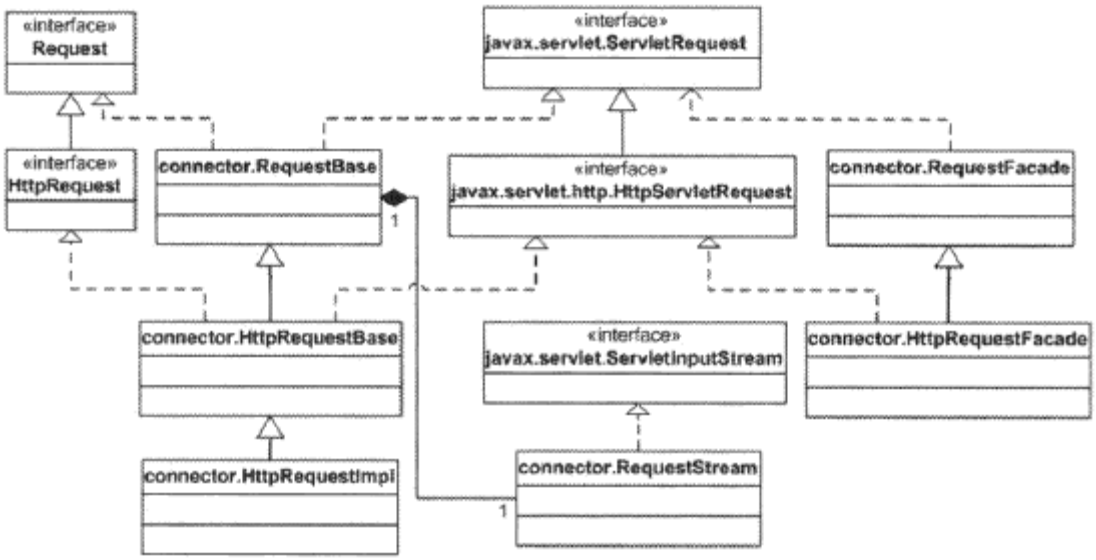


Figure 4.2: The Request interface and related types

如果你理解第 3 章的请求对象，理解这个结构图你应该不会遇到什么困难。

响应对象

Response 接口和它的实现类的 UML 图在 Figure 4.3 中给出。

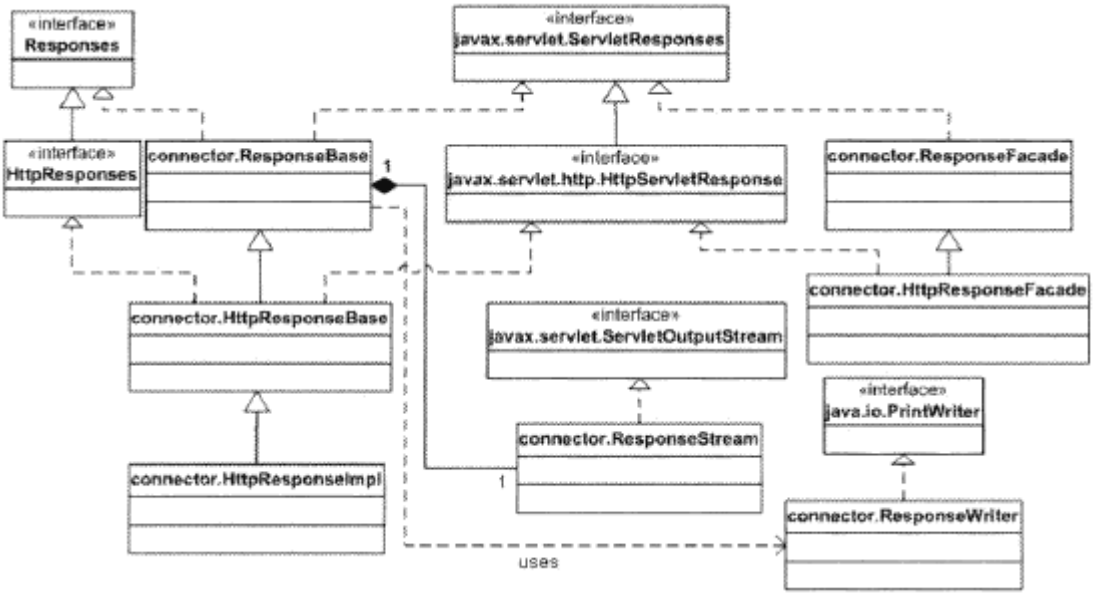


Figure 4.3: The Response interface and its implementation classes

处理请求

到这个时候，你已经理解了请求和响应对象，并且知道 HttpConnector 对象是如何创建它们的。现在是这个过程的最后一点东西了。在这节中我们关注 HttpProcessor 类的 process 方法，它是一个套接字赋给它之后，在 HttpProcessor 类的 run 方法中调用的。process 方法会做下面

这些工作：

- 解析连接
- 解析请求
- 解析头部

在解释完 process 方法之后，在本节的各个小节中将讨论每个操作。

process 方法使用布尔变量 ok 来指代在处理过程中是否发现错误，并使用布尔变量 finishResponse 来指代 Response 接口中的 finishResponse 方法是否应该被调用。

```
boolean ok = true;
boolean finishResponse = true;
```

另外，process 方法也使用了布尔变量 keepAlive,stopped 和 http11。keepAlive 表示连接是否是持久的，stopped 表示 HttpProcessor 实例是否已经被连接器终止来确认 process 是否也应该停止，http11 表示 从 web 客户端过来的 HTTP 请求是否支持 HTTP 1.1。

像第 3 章那样，有一个 SocketInputStream 实例用来包装套接字的输入流。注意的是，SocketInputStream 的构造方法同样传递了从连接器获得的缓冲区大小，而不是从 HttpProcessor 的本地变量获得。这是因为对于默认连接器的用户而言，HttpProcessor 是不可访问的。通过传递 Connector 接口的缓冲区大小，这就使得使用连接器的任何人都可以设置缓冲大小。

```
SocketInputStream input = null;
OutputStream output = null;
// Construct and initialize the objects we will need
try {
    input = new SocketInputStream(socket.getInputstream(),
    connector.getBufferSize());
}
catch (Exception e) {
    ok = false;
}
```

然后，有个 while 循环用来保持从输入流中读取，直到 HttpProcessor 被停止，一个异常被抛出或者连接给关闭为止。

```
keepAlive = true;
while (!stopped && ok && keepAlive) {
    ...
}
```

在 while 循环的内部，process 方法首先把 finishResponse 设置为 true，并获得输出流，并对请求和响应对象做些初始化处理。

```
finishResponse = true;
try {
    request.setStream(input);
    request.setResponse(response);
    output = socket.getOutputStream();
    response.setStream(output);
    response.setRequest(request);
    ((HttpServletResponse) response.getResponse()).setHeader("Server", SERVER_INFO);
}
```

```
catch (Exception e) {
    log("process.create", e); //logging is discussed in Chapter 7
    ok = false;
}
```

接着，process 方法通过调用 parseConnection，parseRequest 和 parseHeaders 方法开始解析前来的 HTTP 请求，这些方法将在这节的小节中讨论。

```
try {
    if (ok) {
        parseConnection(socket);
        parseRequest(input, output);
        if (!request.getRequest().getProtocol().startsWith("HTTP/0"))
            parseHeaders(input);
```

parseConnection 方法获得协议的值，像 HTTP0.9, HTTP1.0 或 HTTP1.1。如果协议是 HTTP1.0，keepAlive 设置为 false，因为 HTTP1.0 不支持持久连接。如果在 HTTP 请求里边找到 Expect: 100-continue 的头部信息，则 parseHeaders 方法将把 sendAck 设置为 true。

如果协议是 HTTP1.1，并且 web 客户端发送头部 Expect: 100-continue 的话，通过调用 ackRequest 方法它将响应这个头部。它将会测试组块是否是允许的。

```
if (http11) {
    // Sending a request acknowledge back to the client if requested.
    ackRequest(output);
    // If the protocol is HTTP/1.1, chunking is allowed.
    if (connector.isChunkingAllowed())
        response.setAllowChunking(true);
}
```

ackRequest 方法测试 sendAck 的值，并在 sendAck 为 true 的时候发送下面的字符串：

```
HTTP/1.1 100 Continue\r\n\r\n
```

在解析 HTTP 请求的过程中，有可能会抛出异常。任何异常将会把 ok 或者 finishResponse 设置为 false。在解析过后，process 方法把请求和响应对象传递给容器的 invoke 方法：

```
try {
                    ((HttpServletResponse)                response).setHeader("Date",
FastHttpDateFormat.getCurrentDate());
    if (ok) {
        connector.getContainer().invoke(request, response);
    }
}
```

接着，如果 finishResponse 仍然是 true，响应对象的 finishResponse 方法和请求对象的 finishRequest 方法将被调用，并且结束输出。

```
if (finishResponse) {
    ...
    response.finishResponse();
    ...
    request.finishRequest();
    ...
    output.flush();
```

while 循环的最后一部分检查响应的 Connection 头部是否已经在 servlet 内部设为 close，或者协议是 HTTP1.0.如果是这种情况的话，keepAlive 设置为 false。同样，请求和响应对象接着会被回收利用。

```
if ( "close".equals(response.getHeader("Connection")) ) {
    keepAlive = false;
}
// End of request processing
status = Constants.PROCESSOR_IDLE;
// Recycling the request and the response objects
request.recycle();
response.recycle();
}
```

在这个场景中，如果哦 keepAlive 是 true 的话，while 循环将会在开头就启动。因为在前面的解析过程中和容器的 invoke 方法中没有出现错误，或者 HttpProcessor 实例没有被停止。否则，shutdownInput 方法将会调用，而套接字将被关闭。

```
try {
    shutdownInput(input);
    socket.close();
}
...
```

shutdownInput 方法检查是否有未读取的字节。如果有的话，跳过那些字节。

解析连接

parseConnection 方法从套接字中获取到网络地址并把它赋予 HttpRequestImpl 对象。它也检查是否使用代理并把套接字赋予请求对象。parseConnection 方法在 Listing4.2 中列出。

Listing 4.2: The parseConnection method

```
private void parseConnection(Socket socket) throws IOException, ServletException {
    if (debug >= 2)
        log(" parseConnection: address=" + socket.getInetAddress() +
            ", port=" + connector.getPort());
    ((HttpRequestImpl) request).setInet(socket.getInetAddress());
    if (proxyPort != 0)
        request.setServerPort(proxyPort);
    else
    request.setServerPort(serverPort);
    request.setSocket(socket);
}
```

解析请求

parseRequest 方法是第 3 章中类似方法的完整版本。如果你很好的理解第 3 章的话，你通过阅读这个方法应该可以理解这个方法是怎么运行的。

解析头部

默认链接器的 parseHeaders 方法使用包 org.apache.catalina.connector.http 里边的 HttpHeader 和 DefaultHeaders 类。类 HttpHeader 指代一个 HTTP 请求头部。类 HttpHeader 不是像第 3 章那样使用字符串，而是使用字符数据用来避免昂贵的字符串操作。类 DefaultHeaders 是一个 final 类，在字符数组中包含了标准的 HTTP 请求头部：

```
standard HTTP request headers in character arrays:
static final char[] AUTHORIZATION_NAME = "authorization".toCharArray();
static final char[] ACCEPT_LANGUAGE_NAME = "accept-language".toCharArray();
static final char[] COOKIE_NAME = "cookie".toCharArray();
...
```

parseHeaders 方法包含一个 while 循环，可以持续读取 HTTP 请求直到再也没有更多的头部可以读取到。while 循环首先调用请求对象的 allocateHeader 方法来获取一个空的 HttpHead 实例。这个实例被传递给 SocketInputStream 的 readHeader 方法。

```
HttpHeader header = request.allocateHeader();
// Read the next header
input.readHeader(header);
```

假如所有的头部都被已经被读取的话，readHeader 方法将不会赋值给 HttpHeader 实例，这个时候 parseHeaders 方法将会返回。

```
if (header.nameEnd == 0) {
    if (header.valueEnd == 0) {
        return;
    }
    else {
                                                                    throw
new         ServletException(sm.getString("httpProcessor.parseHeaders.colon"));
    }
}
```

如果存在一个头部的名称的话，这里必须同样会有一个头部的值：

```
String value = new String(header.value, 0, header.valueEnd);
```

接下去，像第 3 章那样，parseHeaders 方法将会把头部名称和 DefaultHeaders 里边的名称做对比。注意的是，这样的对比是基于两个字符数组之间，而不是两个字符串之间的。

```
if (header.equals(DefaultHeaders.AUTHORIZATION_NAME)) {
    request.setAuthorization(value);
}
else if (header.equals(DefaultHeaders.ACCEPT_LANGUAGE_NAME)) {
    parseAcceptLanguage(value);
}
else if (header.equals(DefaultHeaders.COOKIE_NAME)) {
    // parse cookie
}
else if (header.equals(DefaultHeaders.CONTENT_LENGTH_NAME)) {
    // get content length
}
else if (header.equals(DefaultHeaders.CONTENT_TYPE_NAME)) {
```

```
    request.setContentType(value);
}
else if (header.equals(DefaultHeaders.HOST_NAME)) {
    // get host name
}
else if (header.equals(DefaultHeaders.CONNECTION_NAME)) {
    if (header.valueEquals(DefaultHeaders.CONNECTION_CLOSE_VALUE)) {
        keepAlive = false;
        response.setHeader("Connection", "close");
    }
}
else if (header.equals(DefaultHeaders.EXPECT_NAME)) {
    if (header.valueEquals(DefaultHeaders.EXPECT_100_VALUE))
        sendAck = true;
    else
        throw new ServletException(sm.getstring
            ("httpProcessor.parseHeaders.unknownExpectation"));
}
else if (header.equals(DefaultHeaders.TRANSFER_ENCODING_NAME)) {
    //request.setTransferEncoding(header);
}
request.nextHeader();
```

简单容器的应用程序

本章的应用程序的主要目的是展示默认连接器是怎样工作的。它包括两个类：

ex04.pyrmont.core.SimpleContainer 和 ex04 pyrmont.startup.Bootstrap。类 SimpleContainer 实现了 org.apache.catalina.container 接口，所以它可以和连接器关联。类 Bootstrap 是用来启动应用程序的，我们已经移除了第 3 章带的应用程序中的连接器模块，类 ServletProcessor 和 StaticResourceProcessor，所以你不能请求一个静态页面。

类 SimpleContainer 展示在 Listing 4.3.

Listing 4.3: The SimpleContainer class

```
package ex04.pyrmont.core;
import java.beans.PropertyChangeListener;
import java.net.URL;
import java.net.URLClassLoader;
import java.net.URLStreamHandler;
import java.io.File;
import java.io.IOException;
import javax.naming.directory.DirContext;
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
import org.apache.catalina.Cluster;
import org.apache.catalina.Container;
import org.apache.catalina.ContainerListener;
import org.apache.catalina.Loader;
import org.apache.catalina.Logger;
import org.apache.catalina.Manager;
import org.apache.catalina.Mapper;
import org.apache.catalina.Realm;
import org.apache.catalina.Request;
import org.apache.catalina.Response;
public class SimpleContainer implements Container {
    public static final String WEB_ROOT =
        System.getProperty("user.dir") + File.separator + "webroot";
    public SimpleContainer() { }
    public String getInfo() {
        return null;
    }
    public Loader getLoader() {
        return null;
    }
    public void setLoader(Loader loader) { }
    public Logger getLogger() {
        return null;
    }
    public void setLogger(Logger logger) { }
    public Manager getManager() {
        return null;
    }
    public void setManager(Manager manager) { }
    public Cluster getCluster() {
        return null;
    }
    public void setCluster(Cluster cluster) { }
    public String getName() {
        return null;
    }
    public void setName(String name) { }
    public Container getParent() {
        return null;
    }
    public void setParent(Container container) { }
    public ClassLoader getParentClassLoader() {
        return null;
    }
```

```
    public void setParentClassLoader(ClassLoader parent) { }
    public Realm getRealm() {
        return null;
    }
    public void setRealm(Realm realm) { }
    public DirContext getResources() {
        return null;
    }
    public void setResources(DirContext resources) { }
    public void addChild(Container child) { }
    public void addContainerListener(ContainerListener listener) { }
    public void addMapper(Mapper mapper) { }
    public void addPropertyChangeListener(
PropertyChangeListener listener) { }
public Container findchild(String name) {
return null;
}
public Container[] findChildren() {
return null;
}
public ContainerListener[] findContainerListeners() {
return null;
}
public Mapper findMapper(String protocol) {
return null;
}
public Mapper[] findMappers() {
return null;
}
public void invoke(Request request, Response response)
throws IoException, ServletException {
string servletName = ( (Httpservletrequest)
request).getRequestURI();
servletName = servletName.substring(servletName.lastIndexof("/") +
1);
URLClassLoader loader = null;
try {
URL[] urls = new URL[1];
URLStreamHandler streamHandler = null;
File classpath = new File(WEB_ROOT);
string repository = (new URL("file",null,
classpath.getCanonicalpath() + File.separator)).toString();
urls[0] = new URL(null, repository, streamHandler);
loader = new URLClassLoader(urls);
```

```
}
catch (IOException e) {
System.out.println(e.toString() );
}
Class myClass = null;
try {
myClass = loader.loadclass(servletName);
}
catch (classNotFoundException e) {
System.out.println(e.toString());
}
servlet servlet = null;
try {
servlet = (Servlet) myClass.newInstance();
servlet.service((HttpServletRequest) request,
(HttpServletResponse) response);
}
catch (Exception e) {
System.out.println(e.toString());
}
catch (Throwable e) {
System.out.println(e.toString());
}
}
public Container map(Request request, boolean update) {
return null;
}
public void removeChild(Container child) { }
public void removeContainerListener(ContainerListener listener) { }
public void removeMapper(Mapper mapper) { }
public void removoPropertyChangeListener(
PropertyChangeListener listener) {
}
}
```

我只是提供了 SimpleContainer 类的 invoke 方法的实现，因为默认连接器将会调用这个方法。invoke 方法创建了一个类加载器，加载 servlet 类，并调用它的 service 方法。这个方法和第 3 章的 ServletProcessor 类在哦个的 process 方法非常类似。

Bootstrap 类在 Listing 4.4 在列出.

Listing 4.4: The ex04.pyrmont.startup.Bootstrap class

```
package ex04.pyrmont.startup;
import ex04.pyrmont.core.simplecontainer;
import org.apache.catalina.connector.http.HttpConnector;
public final class Bootstrap {
public static void main(string[] args) {
```

```
HttpConnector connector = new HttpConnector();
SimpleContainer container = new SimpleContainer();
connector.setContainer(container);
try {
connector.initialize();
connector.start();
// make the application wait until we press any key.
System in.read();
}
catch (Exception e) {
e.printStackTrace();
}
}
}
```

Bootstrap 类的 main 方法构造了一个 org.apache.catalina.connector.http.HttpConnector 实例和一个 SimpleContainer 实例。它接下去调用 conncetor 的 setContainer 方法传递 container，让 connector 和 container 关联起来。下一步，它调用 connector 的 initialize 和 start 方法。这将会使得 connector 为处理 8080 端口上的任何请求做好了准备。

你可以通过在控制台中输入一个按键来终止这个应用程序。

运行应用程序

要在 Windows 中运行这个程序的话，在工作目录下输入以下内容：

```
java -classpath ./lib/servlet.jar;./ ex04.pyrmont.startup.Bootstrap
```

在 Linux 的话，你可以使用分号来分隔两个库。

```
java -classpath ./lib/servlet.jar:./ ex04.pyrmont.startup.Bootstrap
```

你可以和第三章那样调用 PrimitiveServlet 和 ModernServlet。

注意的是你不能请求 index.html，因为没有静态资源的处理器。

总结

本章展示了如何构建一个能和 Catalina 工作的 Tomcat 连接器。剖析了 Tomcat4 的默认连接器的代码并用这个连接器构建了一个小应用程序。接下来的章节的所有应用程序都会使用默认连接器。

第五章 容器

容器是一个处理用户 servlet 请求并返回对象给 web 用户的模块。org.apache.catalina.Container 接口定义了容器的形式，有四种容器：Engine（引擎），Host（主机），Context（上下文），和 Wrapper（包装器）。这一章将会介绍 context 和 wrapper，而 Engine 和 Host 会留到第十三章介绍。这一章首先介绍容器接口，然后介绍容器的工作流程。然后介绍的内容是 Wrapper 和 Context 接口。然后用两个例子来总结 wrapper 和 context 容器。

容器接口

一个容器必须实现 org.apache.catalina.Container 接口。就如在第四章中看到的，传递一个 Container 实例给 Connector 对象的 setContainer 方法，然后 Connector 对象就可以使用 container 的 invoke 方法，重新看第四章中 Bootstrap 类的代码如下：

```
HttpConnector connector = new HttpConnector();
SimpleContainer container = new SimpleContainer();
connector.setContainer(container);
```

对于 Catalina 的容器首先需要注意的是它一共有四种不同的容器：

- Engine：表示整个 Catalina 的 servlet 引擎
- Host：表示一个拥有数个上下文的虚拟主机
- Context：表示一个 Web 应用，一个 context 包含一个或多个 wrapper
- Wrapper：表示一个独立的 servlet

每一个概念之上是用 org.apache.catalina 包来表示的。Engine、Host、Context 和 Wrapper 接口都实现了 Container 即可。它们的标准实现是 StandardEngine, StandardHost, StandardContext, and StandardWrapper，它们都是 org.apache.catalina.core 包的一部分。

图 5.1 表示了 Container 接口和它的子接口的结构图。注意接口都是 org.apache.catalina 包的，而所有的类都是 org.apache.catalina.core 包的。

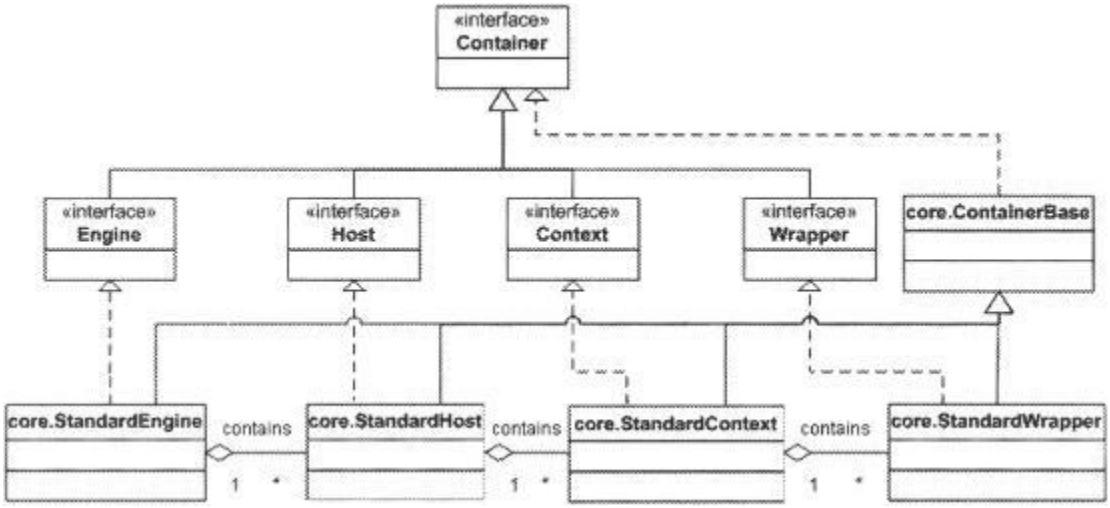


Figure 5.1: The class diagram of Container and its related types

注意 所有的类都扩展自抽象类 ContainerBase.

一个 Catalina 功能部署不一定需要所有的四种类型的容器。例如本章的第一个应用程序仅仅包括一个 wrapper，而第二个应用程序是一个包含 Context 和 wrapper 的容器模块。

一个容器可以有一个或多个低层次上的子容器。例如，一个 Context 有一个或多个 wrapper，而 wrapper 作为容器层次中的最底层，不能包含子容器。讲一个容器添加到另一容器中可以使用在 Container 接口中定义的 addChild()方法，如下定义：

```
public void addChild(Container child);
```

删除一个容器可以使用 Container 接口中定义的 removeChild()方法，删除方法如下表示：

```
public void removeChild(Container child);
```

另外容器接口支持子接口查找和获得所有子接口集合的方法 findChild 和 findChildren 方法。如下表示：

```
public Container findChild(String name);
public Container[] findChildren();
```

一个容器还包含一系列的部分如 Lodder、Loggee、Manager、Realm 和 Resources。

这些组成部分将会在后边章节中进行讨论。需要注意的一点是 Container 接口对于这些组件都定义了 set 和 get 方法包括：getLoader and setLoader, getLogger and setLogger, getManager and setManager, getRealm and setRealm, and getResources and setResources.

更有意思的是 Container 接口被设计成 Tomcat 管理员可以通过 server.xml 文件配置来决定其工作方式的模式。它通过一个 pipeline（流水线）和一系列的阀门来实现，这些内容将会在下一节 Pipelining Task 中讨论。

注意 Tomcat4 中的 Container 接口和 Tomcat5 中的接口有稍许不同。例如第四种有一个 map 方法，但是在 Tomcat5 中，该方法被删除了。

Pipelining Tasks（流水线任务）

这一章介绍了 connector 调用容器的 Invoke 方法后做的工作，然后讨论了 org.apache.catalina 中四个相关的接口：Pipeline, Valve, ValveContext, 和 Contained。

一个 pipeline 包含了改容器要唤醒的所有任务。每一个阀门表示了一个特定的任务。一个容器的流水线有一个基本的阀门，但是你可以添加任意你想要添加的阀门。阀门的数目定义为添加的阀门的个数（不包括基本阀门）。有趣的是，阀门可以痛苦编辑 Tomcat 的配置文件 server.xml 来动态的添加。

Figure 5.2: Pipeline and valves

如果你已经理解了 servlet 过滤器，那么流水线和它的阀门的工作方式不难想象。一个流水线就像一个过滤链，每一个阀门像一个过滤器。跟过滤器一样，一个阀门可以操作传递给它的 request 和 response 方法。让一个阀门完成了处理，则进一步处理流水线中的下一个阀门，基本阀门总是在最后才被调用。

一个容器可以有一个流水线。当容器的 invoke 方法被调用的时候，容器将会处理流水线中的阀门，并一个接一个的处理，直到所有的阀门都被处理完毕。可以想象流水线的 invoke 方法的伪代码如下所示：

```
// invoke each valve added to the pipeline
for (int n=0; n<valves.length; n++) {
  valve[n].invoke( ... );
}
// then, invoke the basic valve
basicValve.invoke( ... );
```

但是，Tomcat 的设计者选择了一种不同的通过 org.apache.catalina.ValveContext 定义的方式来处理，这里介绍它如何工作的：

容器的 invoke 方法在被 connector 调用的时候所作的工作不难进行编码。容器调用的是流水线的 invoke 方法。流水线接口的 invoke 方法前面跟容器接口的 invoke 方法签名相同

```
public void invoke(Request request, Response response)
  throws IOException, ServletException;
```

Here is the implementation of the Container interface's invoke method in the org.apache.catalina.core.ContainerBase class.

这里是 Container 接口中 invoke 方法在
org.apache.catalina.core.ContainerBase 的实现：

```
public void invoke(Request request, Response response)
  throws IOException, ServletException {
  pipeline.invoke(request, response);
}
```

Pipeline 是容器中 Pipeline 接口的一个实例。

现在，流水线必须保证说要添加给它的阀门必须被调用一次，流水线通过创建一个 ValveContext 接口的实例来实现它。ValveContext 是流水线的的内部类，这样 ValveContext 就可以访问流水线中所有的成员。ValveContext 中最重要的方法是 invokeNext 方法：

```
public void invokeNext(Request request, Response response)
  throws IOException, ServletException
```

在创建一个 ValveContext 实例之后，流水线调用 ValveContext 的 invokeNext 方法。ValveContext 会先唤醒流水线的第一个阀门，然后第一个阀门会在完成它的任务之前唤醒下一个阀门。ValveContext 将它自己传递给每一个阀门，那么该阀门就可以调用 ValveContext 的 invokeNext 方法。Valve 接口的 invoke 签名如下：

```
public void invoke(Request request, Response response,
  ValveContext ValveContext) throws IOException, ServletException
```

一个阀门的 invoke 方法可以如下实现：

```
public void invoke(Request request, Response response,
  ValveContext valveContext) throws IOException, ServletException {
  // Pass the request and response on to the next valve in our pipeline
  valveContext.invokeNext(request, response);
  // now perform what this valve is supposed to do
    ...
}
```

org.apache.catalina.core.StandardPipeline 类是容器流水线的实现。在 Tomcat4 中，这个类中有一个内部类 StandardPipelineValveContext 实现了 ValveContext 接口，Listing5.1 展示了 StandardPipelineValveContext 类：

Listing 5.1: The StandardPipelineValveContext class in Tomcat 4

```
protected class StandardPipelineValveContext implements ValveContext {
  protected int stage = 0;
  public String getInfo() {
    return info;
  }
  public void invokeNext(Request request, Response response)
    throws IOException, ServletException {

    int subscript = stage;
```

```
    stage = stage + 1;

    // Invoke the requested Valve for the current request thread
    if (subscript < valves.length) {
      valves[subscript].invoke(request, response, this);
    }
    else if ((subscript == valves.length) && (basic != null)) {
      basic.invoke(request, response, this);
    }
    else {
      throw new ServletException
        (sm.getString("standardPipeline.noValve"));
    }
  }
}
```

InvokeNext 方法使用下标（subscript）和级别（stage）记住哪个阀门被唤醒。当第一次唤醒的时候，下标的值是 0，级的值是 1。以你次，第一个阀门被唤醒，流水线的阀门获得 ValveContext 实例调用它的 invokeNext 方法。这时下标的值是 1 所以下一个阀门被唤醒，然后一步步的进行。

Tomcat5 从 StandardPipeline 中删除了 StandardPipelineValveContext 类，而是使用 rg.apache.catalina.core.StandardValveContext 类来代替，如 Listing 5.2 所示：

Listing 5.2: The StandardValveContext class in Tomcat 5

```
package org.apache.catalina.core;

import java.io.IOException;
import javax.servlet.ServletException;
import org.apache.catalina.Request;
import org.apache.catalina.Response;
import org.apache.catalina.Valve;
import org.apache.catalina.ValveContext;
import org.apache.catalina.util.StringManager;

public final class StandardValveContext implements ValveContext {
  protected static StringManager sm =
    StringManager.getManager(Constants.Package);
  protected String info =
    "org.apache.catalina.core.StandardValveContext/1.0";
  protected int stage = 0;
  protected Valve basic = null;
  protected Valve valves[] = null;
  public String getInfo() {
```

```
    return info;

  }

  public final void invokeNext(Request request, Response response)
    throws IOException, ServletException {
    int subscript = stage;
    stage = stage + 1;
    // Invoke the requested Valve for the current request thread
    if (subscript < valves.length) {
      valves[subscript].invoke(request, response, this);
    }
    else if ((subscript == valves.length) && (basic != null)) {
      basic.invoke(request, response, this);
    }
    else {
      throw new ServletException
       (sm.getString("standardPipeline.noValve"));
    }
  }

  void set(Valve basic, Valve valves[]) {
    stage = 0;
    this.basic = basic;
    this.valves = valves;
  }
}
```

你能看到 Tomcat4 中 StandardPipelineValveContext 和 Tomcat 5 中 StandardValveContext 相似的地方吗？

接下来我们会讨论流水线、阀门和阀门上下文 Pipeline, Valve, and ValveContext 的更多细节。

The Pipeline Interface 流水线接口

我们提到的流水线的第一个方法是它的 Pipeline 接口的 invoke 方法，该方法会开始唤醒流水线的阀门。流水线接口允许你添加一个新的阀门或者删除一个阀门。最后，可以使用 setBasic 方法来分配一个基本阀门给流水线，getBasic 方法会得到基本阀门。最后被唤醒的基本阀门，负责处理 request 和回复 response。Pipeline 接口如 Listing5.3

Listing 5.3: The Pipeline interface

```
package org.apache.catalina;
import java.io.IOException;
```

```
import javax.servlet.ServletException;

public interface Pipeline {
  public Valve getBasic();
  public void setBasic(Valve valve);
  public void addValve(Valve valve);
  public Valve[] getValves();
  public void invoke(Request request, Response response)
    throws IOException, ServletException;
  public void removeValve(Valve valve);
}
```

The Valve Interface 阀门接口

阀门接口表示一个阀门，该组件负责处理请求。该接口有两个方法，invoke 和 getInfo 方法。Invoke 方法如上所述，getInfo 方法返回阀门的信息。阀门接口如 Listing5.4

Listing 5.4: The Valve interface

```
package org.apache.catalina;
import java.io.IOException;
import javax.servlet.ServletException;

public interface Valve {
  public String getInfo();
  public void invoke(Request request, Response response,
    ValveContext context) throws IOException, ServletException;
}
```

The ValveContext Interface 阀门上下文接口

阀门上下文接口有两个方法，invokeNext 方法如上所述，getInfo 方法会返回阀门上下文的信息。ValveContext 接口如下：

Listing 5.5: The ValveContext interface

```
package org.apache.catalina;
import java.io.IOException;
import javax.servlet.ServletException;

public interface ValveContext {
  public String getInfo();
  public void invokeNext(Request request, Response response)
    throws IOException, ServletException;
```

}

The Contained Interface Contained 接口

一个阀门可以选择性的实现 org.apache.catalina.Contained 接口。该接口定义了其实现类跟一个容器相关联。Contained 如 Listing5.6

Listing 5.6: The Contained interface

```
package org.apache.catalina;
public interface Contained {
  public Container getContainer();
  public void setContainer(Container container);
```

the Wrapper Interface Wrapper 接口

org.apache.catalina.Wrapper 接口表示了一个包装器。一个包装器是表示一个独立 servlet 定义的容器。包装器继承了 Container 接口,并且添加了几个方法。包装器的实现类负责管理其下层 servlet 的生命中期，包括 servlet 的 init,service,和 destroy 方法。由于包装器是最底层的容器，所以不可以将子容器添加给它。如果 addChild 方法被调用的时候会产生 IllegalArgumantException 异常。

包装器接口中重要方法有 allocate 和 load 方法。allocate 方法负责定位该包装器表示的 servlet 的实例。Allocate 方法必须考虑一个 servlet 是否实现了 avax.servlet.SingleThreadModel 接口，该部分内容将会在 11 章中进行讨论。Load 方法负责 load 和初始化 servlet 的实例。它们的签名如下：

```
public javax.servlet.Servlet allocate() throws
javax.servlet.ServletException;
public void load() throws javax.servlet.ServletException;
```

其它的方法将会在第 11 章中介绍 org.apache.catalina.core.StandardWrapper 类的时候涉及到。

The Context Interface 上下文（Context）接口

一个 context 在容器中表示一个 web 应用。一个 context 通常含有一个或多个包装器作为其子容器。

重要的方法包括 addWrapper, createWrapper 等方法。该接口将会在第 12 章中详细介绍。

The Wrapper Application（包装器应用程序）

这个应用程序展示了如何写一个简单的容器模型。该应用程序的核心类是 ex05.pyrmont.core.SimpleWrapper，它实现了 Wrapper 接口。SimpleWrapper 类包括一个 Pipeline（由 ex05.pyrmont.core.SimplePipeline 实现）和一个

Loader 类（ex05.pyrmont.core.SimpeLoader）来加载一个 servlet。流水线包括一个基本阀门（ex05.pyrmont.core.SimpleWrapperValve）和两个另外的阀门(ex05.pyrmont.core.ClientIPLoggerValve 和
ex05.pyrmont.core.HeaderLoggerValve).该应用的类结构图如图 5.3 所示：

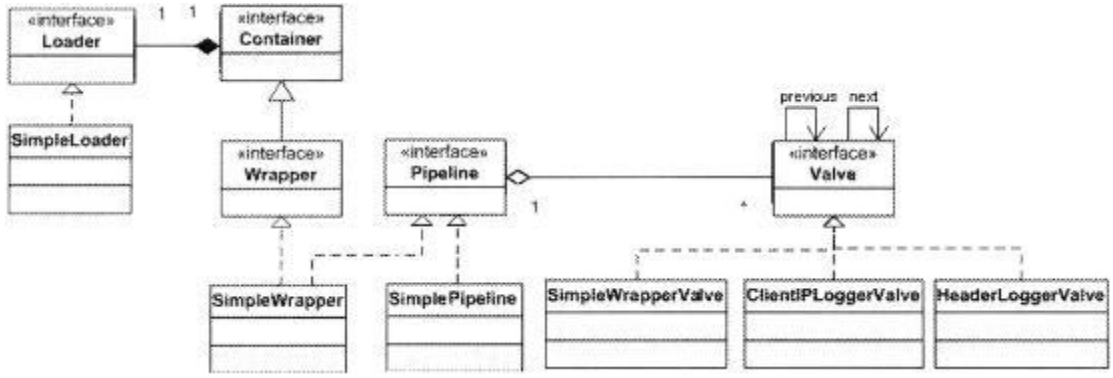


Figure 5.3: The Class Diagram of the Wrapper Application

Note The container uses Tomcat 4's default connector.

包装器包装签名使用的 ModernServlet。这个应用程序表示一个 servlet 容器可以由一个单一的包装器构成。这些类都没有完整的实现，只是实现了必须的方法。接下来看程序的具体实现。

ex05.pyrmont.core.SimpleLoader

容器中加载 servlet 的任务被分配给了 Loader 实现。在该程序中 SimpleLoader 就是一个 Loader 实现。它知道如何定位一个 servlet，并且通过 getClassLoader 获得一个 java.lang.ClassLoader 实例用来查找 servlet 类位置。SimpleLoader 定义了 3 个变量，第一个是 WEB_ROOT 用来指明在哪里查找 servlet 类。

```
public static final String WEB_ROOT =
  System.getProperty("user.dir") + File.separator  + "webroot";
```

The other two variables are object references of type ClassLoader and Container:

另外两个变量时 ClassLoader 和 Container：

```
ClassLoader classLoader = null;
Container container = null;
```

SimpleLoader 类的构造器初始化类加载器，那样准备返回一个 SimpleWrapper 实例。

```
public SimpleLoader() {
    try {
      URL[] urls = new URL[1];
      URLStreamHandler streamHandler = null;
      File classPath = new File(WEB_ROOT);
```

```
      String repository = (new URL("file", null,
        classPath.getCanonicalPath() + File.separator)).toString() ;
      urls[0] = new URL(null, repository, streamHandler);
      classLoader = new URLClassLoader(urls);
    }
    catch (IOException e) {
      System.out.println(e.toString() );
    }
  }
```

该程序的构造器用于初始化一个类加载器如前面章节所用的。容器变量表示容器跟该加载器是相关联的。

Note Loaders will be discussed in detail in Chapter 8.

ex05.pyrmont.core.SimplePipeline

SimplePipeline 实现了 org.apache.catalina.Pipeline 接口。该类中最重要的方法是 invoke 方法，其中包括了一个内部类 SimplePipelineValveContext。SimplePipelineValveContext 实现了 org.apache.catalina.ValveContext 接口如上面章节所介绍的。

ex05.pyrmont.core.SimpleWrapper

该类实现了 org.apache.catalina.Wrapper 接口并且实现了 allocate 方法和 load 方法，该类声明了如下变量：

```
private Loader loader;
protected Container parent = null;
```

loader 变量用于加载一个 servlet 类。Parent 变量表示该包装器的父容器。这意味着，该容器可以是其它容器的子容器，例如 Context。

需要特别注意 getLoader 方法，如 Listing5.7 所示：

Listing 5.7: The SimpleWrapper class's getLoader method

```
public Loader getLoader() {
  if (loader != null)
    return (loader);
  if (parent != null)
    return (parent.getLoader());
  return (null);
}
```

getLoader 方法用于返回一个 Loader 对象用于加载一个 servlet 类。如果一个包装器跟一个加载器相关联，会返回该加载器。否则返回其父容器的加载器，如果没有父容器，则返回 null。

SimpleWrapper 类有一个流水线和该流水线的基本阀门。这些工作在 SimpleWrapper 的构造函数中完成。

Listing 5.8: The SimpleWrapper class's constructor

```
public SimpleWrapper() {

  pipeline.setBasic(new SimpleWrapperValve());
}
```

其中，ipeline 是 SimplePipeline 类的一个实例。

```
private SimplePipeline pipeline = new SimplePipeline(this);
```

ex05.pyrmont.core.SimpleWrapperValve

SimpleWrapperValve 类是一个给 SimpleWrapper 类专门处理请求的基本阀门。它实现了 org.apache.catalina.Valve 接口和 org.apache.catalina.Contained 接口。最重要的方法是 invoke 方法如 Listing5.9 所示

Listing 5.9: The SimpleWrapperValve class's invoke method

```
public void invoke(Request request, Response response,
  ValveContext valveContext)
  throws IOException, ServletException {

  SimpleWrapper wrapper = (SimpleWrapper) getContainer();
  ServletRequest sreq = request.getRequest();
  ServletResponse sres = response.getResponse();
  Servlet servlet = null;
  HttpServletRequest hreq = null;
  if (sreq instanceof HttpServletRequest)
    hreq = (HttpServletRequest) sreq;
  HttpServletResponse hres = null;
  if (sres instanceof HttpServletResponse)
    hres = (HttpServletResponse) sres;
  // Allocate a servlet instance to process this request
  try {
    servlet = wrapper.allocate();
    if (hres!=null && hreq!=null) {
      servlet.service(hreq, hres);
    }
    else {
      servlet.service(sreq, sres);
    }
  }
  catch (ServletException e) {
```

```
  }
}
```

由于 SimpleWrapperValve 被当做一个基本阀门来使用，它的 invoke 方法不需要 invokeNext 方法。Invoke 方法调用 SimpleWrapper 的 allocate 方法获得 servlet 的一个实例。然后调用 servlet 的 service 方法。注意包装器流水线的基本阀门唤醒的是 servlet 的 service 方法，而不是 wrapper 方法自己的。

ex05.pyrmont.valves.ClientIPLoggerValve

ClientIPLoggerValve 是一个阀门，它打印出客户端的 IP 地址到控制台。该类如 Listing5.10

Listing 5.10: The ClientIPLoggerValve class

```
package ex05.pyrmont.valves;

import java.io.IOException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletException;
import org.apache.catalina.Request;
import org.apache.catalina.Response;
import org.apache.catalina.Valve;
import org.apache.catalina.ValveContext;
import org.apache.catalina.Contained;
import org.apache.catalina.Container;

public class ClientIPLoggerValve implements Valve, Contained {
  protected Container container;
  public void invoke(Request request, Response response,
    ValveContext valveContext) throws IOException, ServletException {

    // Pass this request on to the next valve in our pipeline
    valveContext.invokeNext(request, response);
    System.out.println("Client IP Logger Valve");
    ServletRequest sreq = request.getRequest();
    System.out.println(sreq.getRemoteAddr());
    System, out.println("-----------------------------------");
  }

  public String getInfo() {
    return null;
  }
  public Container getContainer() {
    return container;
  }
```

```
  public void setContainer(Container container) {
    this.container = container;
  }
}
```

注意 invoke 方法，它的第一件事情是调用阀门上下文 invokeNext 方法来唤醒下一个阀门，然后它会打印出请求对象的 getRemoteAddr 方法的输出。

ex05.pyrmont.valves.HeaderLoggerValve

该类跟 ClientIPLoggerValve 类非常相似。HeaderLoggerValve 是一个阀门打印请求头部到控制台上。该类如 Listing5.11

Listing 5.11: The HeaderLoggerValve class

```
package ex05.pyrmont.valves;

import java.io.IOException;
import java.util.Enumeration;
import javax.servlet.ServletRequest;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import org.apache.catalina.Request;
import org.apache.catalina.Response;
import org.apache.catalina.Valve;
import org.apache.catalina.ValveContext;
import org.apache.catalina.Contained;
import org.apache.catalina.Container;

public class HeaderLoggerValve implements Valve, Contained {
  protected Container container;

  public void invoke(Request request, Response response,
    ValveContext valveContext) throws IOException, ServletException {

    // Pass this request on to the next valve in our pipeline
    valveContext.invokeNext(request, response);
    System.out.println("Header Logger Valve");
    ServletRequest sreq = request.getRequest();
    if (sreq instanceof HttpServletRequest) {
      HttpServletRequest hreq = (HttpServletRequest) sreq;
      Enumeration headerNames = hreq.getHeaderNames();
      while (headerNames.hasMoreElements()) {
        String headerName = headerNames.nextElement().toString();
        String headerValue = hreq.getHeader(headerName);
```

```
        System.out.println(headerName + ":" + headerValue);
      }

    }
    else
      System.out.println("Not an HTTP Request");

    System.out.println ("-----------------------------------");
  }

  public String getInfo() {
    return null;
  }
  public Container getContainer() {
    return container;
  }
  public void setContainer(Container container) {

    this.container = container;
  }
}
```

注意其 invoke 方法，该方法首先调用阀门的 invokeNext 方法唤醒下一个阀门。然后打印出头部的值。

ex05.pyrmont.startup.Bootstrap1

Bootstrap1 用于启动这个应用程序。

Listing 5.12: The Bootstrap1 class

```
package ex05.pyrmont.startup;
import ex05.pyrmont.core.SimpleLoader;
import ex05.pyrmont.core.SimpleWrapper;
import ex05.pyrmont.valves.ClientIPLoggerValve;
import ex05.pyrmont.valves.HeaderLoggerValve;
import org.apache.catalina.Loader;
import org.apache.catalina.Pipeline;
import org.apache.catalina.Valve;
import org.apache.catalina.Wrapper;
import org.apache.catalina.connector.http.HttpConnector;

public final class Bootstrap1 {
  public static void main(String[] args) {
    HttpConnector connector = new HttpConnector();
```

```
    Wrapper wrapper = new SimpleWrapper();
    wrapper.setServletClass("ModernServlet");
    Loader loader = new SimpleLoader();
    Valve valve1 = new HeaderLoggerValve();
    Valve valve2 = new ClientIPLoggerValve();

    wrapper.setLoader(loader);
    ((Pipeline) wrapper).addValve(valve1);
    ((Pipeline) wrapper).addValve(valve2);

    connector.setContainer(wrapper);

    try {
      connector.initialize();
      connector.start();

      // make the application wait until we press a key.
      System.in.read();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

创建 HttpConnector 和 SimpleWrapper 类的实例以后，主方法里分配 ModernServlet 给 SimpleWrapper 的 setServletClass 方法，告诉包装器要加载的类的名字以便于加载。

```
wrapper.setServletClass("ModernServlet");
```

然后它创建了加载器和两个阀门然后将把加载器给包装器：

```
Loader loader = new SimpleLoader();
Valve valve1 = new HeaderLoggerValve();
Valve valve2 = new ClientIPLoggerValve();
wrapper.setLoader(loader);
```

The two valves are then added to the wrapper's pipeline.

然后把两个阀门添加到包装器流水线中。

```
((Pipeline) wrapper).addValve(valve1);
((Pipeline) wrapper).addValve(valve2);
```

最后，把包装器当做容器添加到连接器中，然后初始化并启动连接器。

```
connector.setContainer(wrapper);
```

```
    try {
      connector.initialize();
      connector.start();
```

下一行允许用户在控制台键入回车键以停止程序。

```
    // make the application wait until we press Enter.
    System.in.read();
```

Running the Application

运行程序

在 windows 下面可以在工作目录下面如下运行该程序：

java -classpath ./lib/servlet.jar;./ ex05.pyrmont.startup.Bootstrap1

在 Linux 下面使用冒号分开两个库

java -classpath ./lib/servlet.jar:./ ex05.pyrmont.startup.Bootstrap1

可以使用下面的 URL 来唤醒 servlet

http://localhost:8080

浏览器将会显示从 ModernServlet 得到的回复。跟下面内容相似的内容会显示在控制台上。

```
ModernServlet -- init
Client IP Logger Valve
127.0.0.1
-----------------------------------
Header Logger Valve
accept:image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
application/vnd.ms-excel, application/msword, application/vnd.ms-
powerpoint, */*
accept-language:en-us
accept-encoding:gzip, deflate
user-agent:Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR
1.1.4322)
host:localhost:8080
connection:Keep-Alive
-----------------------------------
```

The Context Application

在本章的第一个程序中，介绍了如何部署一个仅仅包括一个包装器的简单的 web 应用。该程序仅仅包括一个 servlet，也许会有一些应用仅仅需要一个 servlet 名单是大多数的网络应用需要多个 servlet。在这些应用中，你需要一个跟包装器不同的容器：上下文。

第二个应用程序将会示范如何使用一个包含两个包装器的上下文来包装两个 servlet 类。当有多于一个得包装器的时候，需要一个 map 来处理这些子容器，对于特殊的请求可以使用特殊的子容器来处理。

Note A mapper can only be found in Tomcat 4. Tomcat 5 uses another approach to finding a child container.

注意 使用 map 方法是在 Tomcat4 中，Tomcat 5 使用了另一种机制来查找子容器

在这个程序中，mapper 是 ex05.pyrmont.core.SimpleContextMapper 类的一个实例，它继承了 Tomcat 4 中的 org.apache.catalina.Mapper 接口。一个容器也可以有多个 mapper 来支持多协议。例如容器可以用一个 mapper 来支持 HTTP 协议，而使用另一个 mapper 来支持 HTTPS 协议。Listing5.13 提供了 Tomcat4 中的 Mapper 接口。

Listing 5.13: The Mapper interface

```
package org.apache.catalina;
public interface Mapper {

  public Container getContainer();
  public void setContainer(Container container);
  public String getProtocol();
  public void setProtocol(String protocol);
  public Container map(Request request, boolean update);
}
```

getContainer 返回该容器的 mapper，setContainer 方法用于联系一个容器到 mapper。

getProtocol 返回该 mapper 负责处理的协议，setProtocol 用于分配该容器要处理的协议。map 方法返回处理一个特殊请求的子容器。

Figure 5.4 presents the class diagram of this application.

图 5.4 是该程序的类结构图

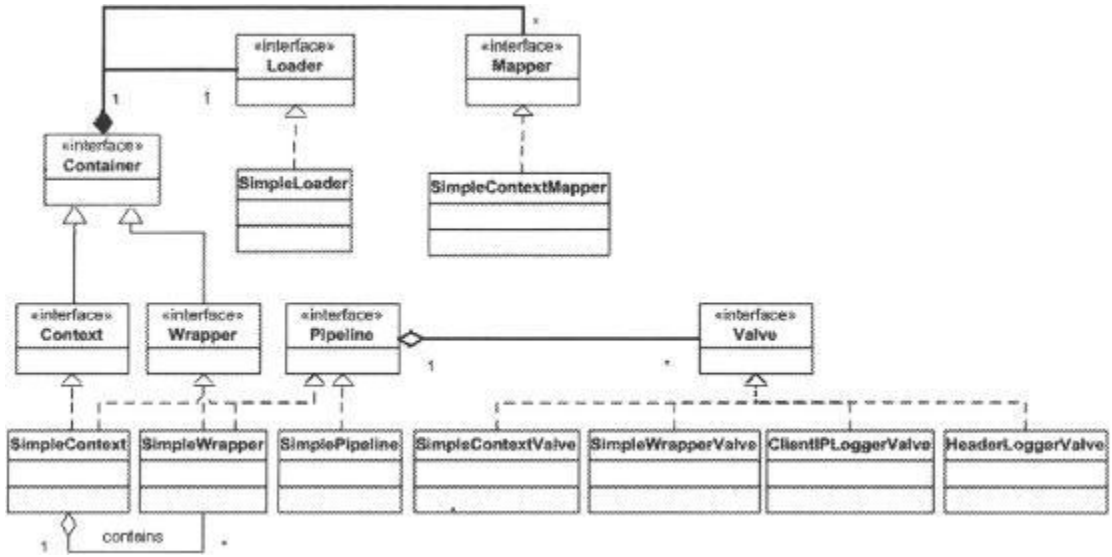


Figure 5.4: The Context application class diagram.

SimpleContext 表示一个上下文，它使用 SimpleContextMapper 作为它的 mapper，SimpleContextValve 作为它的基本阀门。该上下文包括两个阀门 ClientIPLoggerValve 和 HeaderLoggerValve。用 SimpleWrapper 表示的两个包装器作为该上下文的子容器被添加。包装器吧 SimpleWrapperValve 作为它的基本阀门，但是没有其它的阀门了。

该上下文应用程序使用同一个加载器、两个阀门。但是加载器和阀门时跟该上下文关联的，而不是跟包装器关联。这样，两个加载器就可以都使用该加载器。该上下文被当做连接器的容器。因此，连接器每次收到一个 HTTP 请求可以使用上下文的 invoke 方法。根据前面介绍的内容，其余的工作不难理解。

1. 一个容器有一个流水线，容器的 invoke 方法会调用流水线的 invoke 方法。
2. 流水线的 invoke 方法会调用添加到容器中的阀门的 invoke 方法，然后调用基本阀门的 invoke 方法。
3. 在一个包装器中，基本阀门负责加载相关的 servlet 类并对请求作出相应。
4. 在一个有子容器的上下文中，基本法门使用 mapper 来查找负责处理请求的子容器。如果一个子容器被找到，子容器的 invoke 方法会被调用，然后返回步骤 1。

接下来看处理的流程是如何实现的。

SimpleContext 的 invoke 方法调用流水线的 invoke 方法。

```
public void invoke(Request request, Response response)
  throws IOException, ServletException {
  pipeline.invoke(request, response);
}
```

SimplePipeline 类用来表示流水线，它的 invoke 方法如下所示：

```
public void invoke(Request request, Response response)
  throws IOException, ServletException {
  // Invoke the first Valve in this pipeline for this request
  (new SimplePipelineValveContext()).invokeNext(request, response);
}
```

如流水下任务一节中介绍的，该段代码唤醒所有的阀门然后调用基本阀门的

第六章 生命周期

综述

Catalina 由多个组件组成，当 Catalina 启动的时候，这些组件也会启动。当 Catalina 停止的时候，这些组件也必须有机会被清除。例如，当一个容器停止工作的时候，它必须唤醒所有加载的 servlet 的 destroy 方法，而 session 管理器要保存 session 到二级存储器中。保持组件启动和停止一致的的机制通过实现 org.apache.catalina.Lifecycle 接口来实现。

一个实现了 Lifecycle 接口的组件同是会触发一个或多个下列事件：BEFORE_START_EVENT, START_EVENT, AFTER_START_EVENT, BEFORE_STOP_EVENT, STOP_EVENT, and AFTER_STOP_EVENT。当组件被启动的时候前三个事件会被触发，而组件停止的时候会触发后边三个事件。另外，如果一个组件可以触发事件，那么必须存在相应的监听器来对触发的事件作出回应。监听器使用 org.apache.catalina.LifecycleListener 来表示。

本章会对 Lifecycle, LifecycleEvent, and LifecycleListener 进行讨论。另外，还会解释一个公用类 LifecycleSupport，它给组件提供了一个简单方式来触发生命周期事件和处理事件监听器。在本章中，会建立一个有实现了 Lifecycle 接口的类的工程。该程序时基于第五章的应用程序的。

Lifecycle 接口

Catalina 的设计允许一个组件包含其它的组件。例如一个容器可以包含一系列的组件如加载器、管理器等。一个父组件负责启动和停止其子组件。Catalina 的设计成所有的组件被一个父组件来管理（in custody），所以启动 bootstrap

类只需启动一个组件即可。这种单一的启动停止机制通过继承 Lifecycle 来实现。看 Listing6.1 所示的 Lifecycle 接口。

Listing 6.1: The Lifecycle interface

```
package org.apache.catalina;
public interface Lifecycle {
  public static final String START_EVENT = "start";
  public static final String BEFORE_START_EVENT = "before_start";
  public static final String AFTER_START_EVENT = "after_start";
  public static final String STOP_EVENT = "stop";
  public static final String BEFORE_STOP_EVENT = "before_stop";
  public static final String AFTER_STOP_EVENT = "after_stop";

  public void addLifecycleListener(LifecycleListener listener);
  public LifecycleListener[] findLifecycleListeners();
  public void removeLifecycleListener(LifecycleListener listener);
  public void start() throws LifecycleException;
  public void stop() throws LifecycleException;
}
```

Lifecycle 中最重要的方法是 start 和 stop 方法。一个组件提供了这些方法的实现，所以它的父组件可以通过这些方法来启动和停止他们。另外 3 个方法 addLifecycleListener, findLifecycleListeners, 和 removeLifecycleListener 事跟监听器相关的类。组件的监听器对组件可能触发的时间“感兴趣”，当一个事件被触发的时候，相应监听器会被通知。一个 Lifecycle 实例可以触发使用静态最终字符串定义的六个事件。

LifecycleEvent 类

org.apache.catalina.LifecycleEvent 表示一个生命周期事件，如 Listing6.2 所示：

Listing 6.2: The org.apache.catalinaLifecycleEvent interface

```
package org.apache.catalina;
import java.util.EventObject;

public final class LifecycleEvent extends EventObject {
  public LifecycleEvent(Lifecycle lifecycle, String type) {
    this(lifecycle, type, null);
  }
  public LifecycleEvent(Lifecycle lifecycle, String type,
    Object data) {

    super(lifecycle);
    this.lifecycle = lifecycle;
    this.type = type;
```

```
    this.data = data;
  }
  private Object data = null;
  private Lifecycle lifecycle = null;
  private String type = null;

  public Object getData() {
    return (this.data);
  }
  public Lifecycle getLifecycle() {
    return (this.lifecycle);
  }
  public String getType() {
    return (this.type);
  }
}
```

LifecycleListener 接口

org.apache.catalina.LifecycleListener 接口可以表示生命周期监听器，如 Listing6.3 所示：

Listing 6.3: The org.apache.catalina.LifecycleListener interface

```
package org.apache.catalina;
import java.util.EventObject;
public interface LifecycleListener {
  public void lifecycleEvent(LifecycleEvent event);
}
```

在该接口中，只有一个方法 lifecycleEvent，该方法在事件触发的时候唤醒对其“感兴趣”的监听器。

LifecycleSupport 类

org.apache.catalina.util.LifecycleSupport. The LifecycleSupport class is given in Listing 6.4.

一个实现了 Lifecycle 接口的组件并且允许监听器注册其“感兴趣”的事件必须 Lifecycle 接口提供跟事件相关的方法的代码（addLifecycleListener, findLifecycleListeners, 和 removeLifecycleListener）。这样就可以将组件的监听器添加到 ArrayList 或者其他相似的对象中。Catalina 提供了一个公用类 org.apache.catalina.util.LifecycleSupport 来简化组件处理监听器和触发生命周期事件。LifecycleSupport 如 Listing6.4 所示

Listing 6.4: The LifecycleSupport class

```
package org.apache.catalina.util;
import org.apache.catalina.Lifecycle;
```

```
import org.apache.catalina.LifecycleEvent;
import org.apache.catalina.LifecycleListener;

public final class LifecycleSupport {
  public LifecycleSupport(Lifecycle lifecycle) {
    super();
    this.lifecycle = lifecycle;
  }

  private Lifecycle lifecycle = null;
  private LifecycleListener listeners[] = new LifecycleListener[0];
  public void addLifecycleListener(LifecycleListener listener) {
    synchronized (listeners) {
      LifecycleListener results[] =
        new LifecycleListener[listeners.length + 1];
      for (int i = 0; i < listeners.length; i++)
        results[i] = listeners[i];
      results[listeners.length] = listener;
        listeners = results;
    }
  }

  public LifecycleListener[] findLifecycleListeners() {
    return listeners;
  }

  public void fireLifecycleEvent(String type, Object data) {
    LifecycleEvent event = new LifecycleEvent(lifecycle, type, data);
    LifecycleListener interested[] = null;
    synchronized (listeners) {
      interested = (LifecycleListener[]) listeners.clone();
    }
    for (int i = 0; i < interested.length; i++)
      interested[i].lifecycleEvent(event);
  }

  public void removeLifecycleListener(LifecycleListener listener) {
    synchronized (listeners) {
      int n = -1;
      for (int i = 0; i < listeners.length; i++) {
        if (listeners[i] == listener) {
          n = i;
          break;
        }
```

```
      }
      if (n < 0)
        return;
      LifecycleListener results[] =
        new LifecycleListener[listeners.length - 1];

      int j = 0;
      for (int i = 0; i < listeners.length; i++) {
        if (i != n)
        results[j++] = listeners[i];
      }
      listeners = results;
    }
  }
}
```

如 Listing 6.4 所示，LifecycleSupport 类存储所有的生命周期监听器到一个数组中，该数组为 listenens，它初始化的时候没有任何成员。

```
private LifecycleListener listeners[] = new LifecycleListener[0];
```

当一个监听器通过 addLifecycleListener 方法被添加的时候，一个新的数组（长度比旧数组大 1）会被创建。然后就数组中的元素会被拷贝到新数组中并把新事件添加到数组中。当一个事件被删除的时候，一个新的数组（长度为旧数组-1）的数组会被创建并将所有的元素存储到其中。

fireLifecycleEvent 方法触发一个生命周期事件。首先，它会克隆整个监听器数组，然后它调用每个成员的 lifecycleEvent 方法，传递被触发的事件。

一个实现了 Lifecycle 的组件可以使用 LifecycleSupport 类。例如，该程序的 SimpleContext 类声明了如下变量

```
protected LifecycleSupport lifecycle = new LifecycleSupport(this);
```

SimpleContext 调用 LifecycleSupport 的 addLifecycleListener 方法添加一个生命周期事件：

```
public void addLifecycleListener(LifecycleListener listener) {
  lifecycle.addLifecycleListener(listener);
}
```

要删除一个生命周期监听事件，SimpleContext 调用 LifecycleSupport 类的 removeLifecycleListener 方法。

```
public void removeLifecycleListener(LifecycleListener listener) {
  lifecycle.removeLifecycleListener(listener);
}
```

要触发一个事件，SimpleContext 需要调用 LifecycleSupport 的 fireLifecycleEvent 方法：

lifecycle.fireLifecycleEvent(START_EVENT, null);

The Application

本章的应用程序构建在第五章的程序之上，用来说明 Lifecycle 接口以及与生命周期相关的其他类型。它包括一个上下文和两个包装器以及一个加载器和一个映射。该应用程序中的组件实现了 Lifecycle 接口，上下文有一个监听器。为了使程序简便，第五章中的阀门并没有使用。该程序的结构如图 6.1 所示。注意一些接口（Container, Wrapper, Context, Loader, Mapper）一些类（SimpleContextValve, SimpleContextMapper, and SimpleWrapperValve）并没有出现在该结构图中。

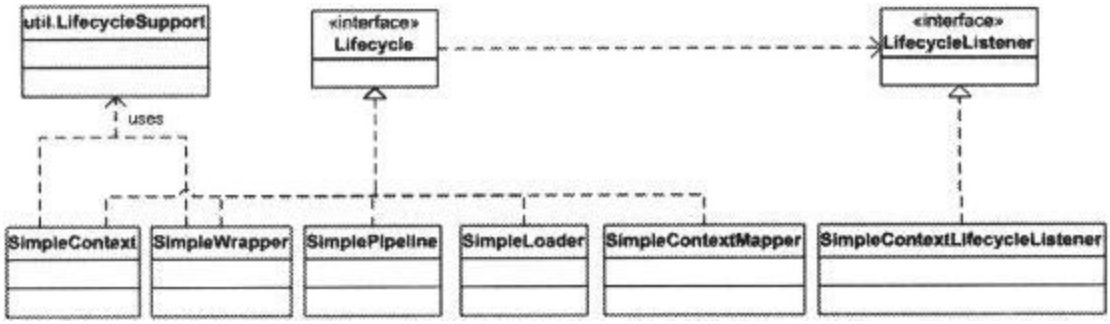


Figure 6.1: The class diagram of the accompanying application

注意 SimpleContextLifecycleListener 类表示了 SimpleContext 的监听器。SimpleContextValve, SimpleContextMapper, 和 SimpleWrapperValve 类跟第五章的程序相同，这里不再涉及。

ex06.pyrmont.core.SimpleContext

该程序中的 SimpleContext 类跟第五章中的相似，不同的是它实现了 Lifecycle 接口。SimpleContext 类使用如下变量来引用一个 LifecycleSupport 实例。

```
protected LifecycleSupport lifecycle = new LifecycleSupport(this);
```

它还使用一个名为 started 的变量来标记该 SimpleContext 实例是否已经启动。SimpleContext 类实现了 Lifecycle 接口中的方法，这些方法如 Listing6.5 所示

Listing 6.5: Methods from the Lifecycle interface.

```
 public void addLifecycleListener(LifecycleListener listener) {
  lifecycle.addLifecycleListener(listener);
}
public LifecycleListener[] findLifecycleListeners() {
  return null;
}
public void removeLifecycleListener(LifecycleListener listener) {
  lifecycle.removeLifecycleListener(listener);
```

```
}
public synchronized void start() throws LifecycleException {
  if (started)
    throw new LifecycleException("SimpleContext has already started");
  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(BEFORE_START_EVENT, null);
  started = true;
  try {
    // Start our subordinate components, if any
    if ((loader != null) && (loader instanceof Lifecycle))
      ((Lifecycle) loader).start();

    // Start our child containers, if any
    Container Children[] = findChildren();
    for (int i = 0; i < children.length; i++) {
      if (children[i] instanceof Lifecycle)
        ((Lifecycle) children[i]).start();
    }

    // Start the Valves in our pipeline (including the basic),
    // if any
    if (pipeline instanceof Lifecycle)
      ((Lifecycle) pipeline).start();
    // Notify our Interested LifecycleListeners
    lifecycle.firelifecycleEvent(START_EVENT, null);
  }
  catch (Exception e) {
    e.printStackTrace();
  }
  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(AFTER_START_EVENT, null);
}

public void stop() throws LifecycleException {
  if (!started)
    throw new LifecycleException("SimpleContext has not been started");
  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(BEFORE_STOP_EVENT, null);
  lifecycle.fireLifecycleEvent(STOP_EVENT, null);
  started = false;
  try {
    // Stop the Valves in our pipeline (including the basic), if any
    if (pipeline instanceof Lifecycle) (
```

```
      ((Lifecycle) pipeline).stop();
    }
    // Stop our child containers, if any
    Container children[] = findChildren();
    for (int i = 0; i < children.length; i++) {
      if (children[i] instanceof Lifecycle)
        ((Lifecycle) children[i]).stop();
    }
    if ((loader != null) && (loader instanceof Lifecycle)) {
      ((Lifecycle) loader).stop();
    }
  }
  catch (Exception e) {
    e.printStackTrace();
  }
  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(AFTER_STOP_EVENT, null);
}
```

注意 start 方法是如何启动所有子容器以及其相关组件（容易加载器、流水线和映射器）的，以及 stop 方法是如何停止这些组件的。使用该机制，可以启动容器模型中所有的组件，你只需要启动最高层的组件即可（在该例子中 SimpleContext 实例）。而停止它们的时候只需简单停止相同的组件即可。

SimpleContext 的 start 方法首先会检查是否已经启动，如果已经启动了，方法会抛出 LifecycleException 异常。

```
if (started)
  throw new LifecycleException(
    "SimpleContext has already started");
```

It then raises the BEFORE_START_EVENT event.

然后它产生了 BEFORE_START_EVENT 事件。

```
// Notify our interested LifecycleListeners
lifecycle.fireLifecycleEvent(BEFORE_START_EVENT, null);
```

其结果就是 SimpleContext 实例中注册的监听器都会被唤醒，一个 SimpleContextLifecycleListener 类型的监听器会注册它“感兴趣”的事件。接下来会看监听器会发生什么事。

接下来，start 方法设置 started 布尔变量为真来标记该组件已经启动。

```
started = true;
```

start 方法接下来会启动所有组件以及其子容器。现在有两个组件实现了 Lifecycle 接口：SimpleLoader 和 SimplePipeline。SimpleContext 有两个包装器作为其子容器。这些包装器也实现了 Lifecycle 接口。

```
try {
```

```
    // Start our subordinate components, if any
    if ((loader != null) && (loader instanceof Lifecycle))
      ((Lifecycle) loader).start();

    // Start our child containers, if any
    Container children[] = findChildren();
    for (int i = 0; i < children.length; i++) {
      if (children[i] instanceof Lifecycle)
        ((Lifecycle) children[i]).start();
    }

    // Start the Valves in our pipeline (including the basic),
    // if any
    if (pipeline instanceof Lifecycle)
      ((Lifecycle) pipeline).start();
```

组件和子容器都被启动之后，start 方法产生两个事件 START_EVENT 和 AFTER_START_EVENT。

```
   // Notify our interested LifecycleListeners
   lifecycle.fireLifecycleEvent(START_EVENT, null);
   .
   .
   .
   // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(AFTER_START_EVENT, null);
```

stop 方法首先检查该实例是否已经启动。如果没有启动会产生一个 LifecycleException 类型的异常。

```
   if (!started)
     throw new LifecycleException(
       "SimpleContext has not been started");
```

接下来会触发 BEFORE_STOP_EVENT 和 STOP_EVENT 事件，并重置 started 变量。

```
   // Notify our interested LifecycleListeners
   lifecycle.fireLifecycleEvent(BEFORE_STOP_EVENT, null);
   lifecycle.fireLifecycleEvent(STOP_EVENT, null);
   started = false;
```

接下来 stop 方法会停止所有相关组件以及 SimpleContext 的子容器。

```
   try {

     // Stop the Valves in our pipeline (including the basic), if any
     if (pipeline instanceof Lifecycle) {
       ((Lifecycle) pipeline).stop();
     }
```

```
    // Stop our child containers, if any
    Container children[] = findChildren();
    for (int i = 0; i < children.length; i++) {
      if (children[i] instanceof Lifecycle)
        ((Lifecycle) children[i]).stop();
    }
    if ((loader != null) && (loader instanceof Lifecycle)) {
      ((Lifecycle) loader).stop();
    }
  }
```

最后，产生 AFTER_STOP_EVENT 事件。

```
  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(AFTER_STOP_EVENT, null);
```

ex06.pyrmont.core.SimpleContextLifecycleListener

SimpleContextLifecycleListener 表示 SimpleContext 实例的监听器。

Listing 6.6: The SimpleContextLifecycleListener class

```
package ex06.pyrmont.core;
import org.apache.catalina.Context;
import org.apache.catalina.Lifecycle;
import org.apache.catalina.LifecycleEvent;
import org.apache.catalina.LifecycleListener;

public class SimpleContextLifecycleListener implements
LifecycleListener {

  public void lifecycleEvent(LifecycleEvent event) {
    Lifecycle lifecycle = event.getLifecycle();
    System.out.println("SimpleContextLifecycleListener's event " +
      event.getType().toString());
    if (Lifecycle.START_EVENT.equals(event.getType())) {
      System.out.println("Starting context.");
    }
    else if (Lifecycle.STOP_EVENT.equals(event.getType())) {
      System.out.println("Stopping context.");
    }
  }
}
```

SimpleContextLifecycleListener 类中 lifecycleEvent 方法的实现很简单。只是打印出触发的时间，如果事件为 START_EVENT 事件，则打印出“Starting context.”，如果事件为 STOP_EVENTT，则打印出“Stopping contex”。

ex06.pyrmont.core.SimpleLoader

SimpleLoader 跟第五章中相似，不同之处在于它实现了 Lifecycle 接口，所实现的方法什么都没做只是打印出一些字符串到控制台上。更重要的是，实现了 Lifecycle 接口，一个 SimpleLoader 实例就可以通过其相关容器启动它。

从 Lifecycle 接口的来得到方法如 Listing6.7 所示。

Listing 6.7: The methods from Lifecycle in the SimpleLoader class

```
public void addLifecycleListener(LifecycleListener listener) { }
public LifecycleListener[] findLifecycleListeners() {
  return null;
}
public void removeLifecycleListener(LifecycleListener listener) { }
public synchronized void start() throws LifecycleException {
  System.out.println("Starting SimpleLoader");
}
public void stop() throws LifecycleException { }
```

ex06.pyrmont.core.SimplePipeline

另外 Pipeline 接口和 SimplePipeline 类也实现了 Lifecycle 接口。从 Lifecycle 接口的来的方法被留空，现在这个类的的实例可以由其相关容器来启动，该类的其他的部分跟第五章中的相似。

ex06.pyrmont.core.SimpleWrapper

该类跟 ex05.pyrmont.core.SimpleWrapper 很相似。在该应用中，它实现了 Lifecycle 接口，所以它可以由其父容器来启动。在该程序中，出 start 和 stop 方法之外的大多数从 Lifecycle 接口获得的方法被留空。Listing6.8 展示了这些方法的实现。

Listing 6.8: The methods from the Lifecycle interface

```
public void addLifecycleListener(LifecycleListener listener) {  }
public LifecycleListener[] findLifecycleListeners() {
  return null;
}
public void removeLifecycleListener(LifecycleListener listener) ( }
public synchronized void start() throws LifecycleException {
```

```
  System.out.println("Starting Wrapper " + name);
  if (started)
    throw new LifecycleException("Wrapper already started");
  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(BEFORE_START_EVENT, null);
  started = true;

  // Start our subordinate components, if any
  if ((loader != null) && (loader instanceof Lifecycle))
    ((Lifecycle) loader).start();
  // Start the Valves in our pipeline (including the basic), if any
  if (pipeline instanceof Lifecycle)
    ((Lifecycle) pipeline).start();
  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(START_EVENT, null);
  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(AFTER_START_EVENT, null);
}

public void stop() throws LifecycleException {
  System.out.println("Stopping wrapper " + name);
  // Shut down our servlet instance (if it has been initialized)
  try {
    instance.destroy();
  }
  catch (Throwable t) {
  }
  instance = null;
  if (!started)
    throw new LifecycleException("Wrapper " + name + " not started");
  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(BEFORE_STOP_EVENT, null);
  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(STOP_EVENT, null);
  started = false;

  // Stop the Valves in our pipeline (including the basic), if any
  if (pipeline instanceof Lifecycle) {
    ((Lifecycle) pipeline).stop();
  }
  // Stop our subordinate components, if any
  if ((loader != null) && (loader instanceof Lifecycle)) {
    ((Lifecycle) loader).stop();
  }
```

```
  // Notify our interested LifecycleListeners

  lifecycle.fireLifecycleEvent(AFTER_STOP_EVENT, null);
}
```

SimpleWrapper 中的 start 方法跟 SimpleContext 类中的 start 方法相似。它启动所有的组件并触发 BEFORE_START_EVENT, START_EVENT,和 AFTER_START_EVENT 事件。

SimpleWrapper 的 stop 方法更有趣,它打印出一个简单的字符串,并唤醒 servlet 实例的 destroy 方法。

```
System.out.println("Stopping wrapper " + name);
// Shut down our servlet instance (if it has been initialized)
try {
  instance.destroy();
}
catch (Throwable t) {
}
instance = null;
```

然后它检查该包装器是否被启动了，如果没有会抛出 LifecycleException。

```
if (!started)
  throw new LifecycleException("Wrapper " + name + " not started");
```

接下来,它触发 BEFORE_STOP_EVENT 和 STOP_EVENT 事件并重置 started 布尔变量。

```
// Notify our interested LifecycleListeners
lifecycle.fireLifecycleEvent(BEFORE_STOP_EVENT, null);
// Notify our interested LifecycleListeners
lifecycle.fireLifecycleEvent(STOP_EVENT, null);
started = false;
```

接下来它停止跟其相关的流水线组件和加载器。在该应用程序中,SimpleWrapper 实例没有加载器。

```
// Stop the Valves in our pipeline (including the basic), if any
if (pipeline instanceof Lifecycle) {
  ((Lifecycle) pipeline).stop();
}
// Stop our subordinate components, if any
if ((loader != null) && (loader instanceof Lifecycle)) {
  ((Lifecycle) loader).stop();
}
```

最后，它触发了 AFTER_STOP_EVENT 事件。

```
// Notify our interested LifecycleListeners
lifecycle.fireLifecycleEvent(AFTER_STOP_EVENT, null);
```

Running the Application

运行程序
在 windows 下面，可以在工作目录下面键入如下命令启动程序：
java -classpath ./lib/servlet.jar;./ ex06.pyrmont.startup.Bootstrap

第七章：日志系统

综述

日志系统是一个记录信息的组件。在 Catalina 中，日志系统是一个相对简单的跟容器相关联的组件。Tomcat 在 org.apache.catalina.logger 包中提供了多个不同的日志系统。本章的应用程序在 ex07.pyrmont 包中。SimpleContext 和 Bootstrap 是从第六章中修改得到的。

本章有三节组成，第一节介绍了 org.apache.catalina.Logger 接口，该接口是所有的日志系统都要实现的。第二节介绍了 Tomcat 中的日志系统，第三节详细讲解了本章的例子，该例子基于 Tomcat 的日志系统。

Logger 接口

一个日志系统必须实现 org.apache.catalina.Logger 接口，该接口如 Listing7.1 所示

Listing 7.1: The Logger interface

```
package org.apache.catalina;
import java.beans.PropertyChangeListener;

public interface Logger {
  public static final int FATAL = Integer.MIN_VALUE;
  public static final int ERROR = 1;
  public static final int WARNING = 2;
  public static final int INFORMATION = 3;
  public static final int DEBUG = 4;

  public Container getContainer();
  public void setContainer(Container container);
  public String getInfo();
  public int getVerbosity();
  public void setVerbosity(int verbosity);
  public void addPropertyChangeListener(PropertyChangeListener
    listener);
  public void log(String message);

  public void log(Exception exception, String msg);
  public void log(String message, Throwable throwable);
  public void log(String message, int verbosity);
  public void log(String message, Throwable throwable, int verbosity);
  public void removePropertyChangeListener(PropertyChangeListener
    listener);
}
```

日志接口提供了日志系统要实现的方法，最简单的方法是接受一个字符串并将其记录，

最后两个方法会接受一个冗余级别（verbosity level），如果传递的数字低于该类的实例设置的冗余级别，就将信息记录下来，否则就忽略信息。使用静态变量定义了五个冗余级别：FATAL, ERROR, WARNING, INFORMATION,和 DEBUG。getVerbosity 和 setVerbosity 分别用来获得和设置冗余级别。

另外，日志接口还有 getContainer 和 setContainer 方法用来将日志系统跟容器关联起来。还有 addPropertyChangeListener 和 removePropertyChangeListener 方法删除和添加 PropertyChangeListener。

看了 Tomcat 中日志系统的实现之后你就会清楚这些方法了。

Tomcat 日志系统

Tomcat 提供了三种日志系统，它们分别是 FileLogger, SystemErrLogger, 和 SystemOutLogger。这些类可以在 org.apache.catalina.logger 包中找到，它们都继承了 org.apache.catalina.logger.LoggerBase 类。在 Tomcat 4 中 LoggerBase 实现了 org.apache.catalina.Logger 接口，在 Tomcat 5 中，它还实现了 Lifecycle 接口（第六章）和 MBeanRegistration 接口（20 章介绍）。

The UML diagram of these classes is shown in Figure 7.1.

它们的 UML 结果图如图 7.1 所示

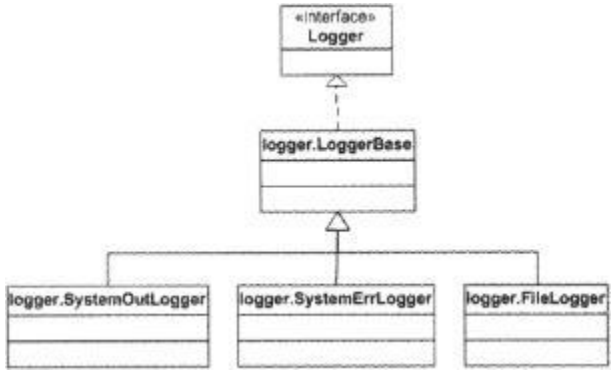


Figure 7.1: Tomcat's Loggers

LoggerBase 类

在 Tomcat5 中，LoggerBase 类由于集成了 MBeans 而比较复杂，所以本章看的是 Tomcat4 中的 LoggerBase 类。等讨论完 20 章后，就可以理解 Tomcat5 中的 LoggerBase 类。

在 Tomcat 4 中，LoggerBase 类是一个抽象类，它实现了 Logger 接口中除 log(String msg)之外的所有方法。

```
public abstract void log(String msg);
```

该方法需要在子类 ongoing 进行覆盖（overload），所有的其他的 log 方法都调用了该方法。因为每一个子类都将信息记录到不同的地方，所以该方法在 LoggerBase 中北留空。

现在来看该类的冗余级别。它被定义为一个 protected 的名为 verbosity 的变量，默认值为 ERROR。

```
protected int verbosity = ERROR;
```

冗余级别可以使用 setVerbosity 方法改变，传递这些字符串给方法即可 FATAL, ERROR, WARNING, INFORMATION, 或 DEBUG。Listing7.2 展示了 LoggerBase 类中 setVobosity 方法的实现。

Listing 7.2: The setVerbosity method

```
public void setVerbosityLevel(String verbosity) {
  if ("FATAL".equalsIgnoreCase(verbosity))
    this.verbosity = FATAL;
  else if ("ERROR".egualsIgnoreCase(verbosity))
    this.verbosity = ERROR;
  else if ("WARNING".equalsIgnoreCase(verbosity))
    this.verbosity = WARNING;
  else if ("INFORMATION".equalsIgnoreCase(verbosity))
    this.verbosity = INFORMATION;
  else if ("DEBUG".equalsIgnoreCase(verbosity))
    this.verbosity = DEBUG;
}
```

有两个 log 方法会接受一个整型参数作为它的冗余级别。log(String message) 的覆盖如 Listing7.3

Listing 7.3: The log method overloads that accept verbosity

```
public void log(String message, int verbosity) {
  if (this.verbosity >= verbosity)
    log(message);
}
public void log(String message, Throwable throwable, int verbosity) {
  if (this.verbosity >= verbosity)
    log(message, throwable);
}
```

在后边介绍的 LoggerBase 的三个子类中，可以看到 log(String message)方法的实现。

SystemOutLogger 类

SystemOutLogger 作为 LoggerBase 的子类提供了 log(String message)方法的实现。每一个收到的信息都被传递给 System.out.println 方法，SystemOutLogger 类如 Listing7.4 所示。

Listing 7.4: The SystemOutLogger Class

```
package org.apache.catalina.logger;
public class SystemOutLogger extends LoggerBase {

  protected static final String info =
    "org.apache.catalina.logger.SystemOutLogger/1.0";

  public void log(String msg) {
    System.out.println(msg);
  }
}
```

SystemErrLogger 类

This class is very similar to the SystemOutLogger class, except that the message argument to the log(String message) method overload calls the System.err.println() method. The SystemErrLogger class is given in Listing 7.5.

SystemErrLogger 类跟 SystemOutLogger 类十分相似，只是它覆盖 log(String message)方法的时候使用的是 System.erro.println()方法。SystemErrLogger 类如 Listing7.5

Listing 7.5: The SystemErrLogger class

```
package org.apache.catalina.logger;
public class SystemErrLogger extends LoggerBase {

  protected static final String info =
    "org.apache.catalina.logger.SystemErrLogger/1.0";
  public void log(String msg) {
    System.err.println(msg);
  }
}
```

The FileLogger Class

FileLogger 类

FileLogger 是 LoggerBase 类中最复杂的。它将从关联容器收到的信息写到文件中，每个信息可以选择性的加上时间戳。在第一次实例化的时候，该类的实例会创建一个文件，该文件的名字带有日期信息。如果日期改变了，它会创建一个新的文件并把信息写在里面。类的实例允许在日志文件的名字上添加前缀和后缀。

在 Tomcat4 中，FileLogger 类实现了 Lifecycle 接口，所以它可以跟其它实现 org.apache.catalina.Lifecycle 接口的组件一样启动和停止。在 Tomcat5 中，它是实现了 Lifecycle 接口的 LoggerBase 类的子类。

Tomcat 4 中 LoggerBase 类的 start 和 stop 方法实现仅仅触发了监听器“感兴趣的”文件日志的开始和停止事件。这两个方法如 Listing7.6 所示，注意 stop 方法调用了该类的关闭日志文件的私有方法（close 方法）。关闭方法会在本节后边的内容中介绍。

Listing 7.6: The start and stop methods

```
public void start() throws LifecycleException {
  // Validate and update our current component state
  if (started)
    throw new LifecycleException
      (sm.getString("fileLogger.alreadyStarted"));
  lifecycle.fireLifecycleEvent(START_EVENT, null);
  started = true;
}

public void stop() throws LifecycleException {
  // Validate and update our current component state
  if (!started)
    throw new LifecycleException
     (sm.getString("fileLogger.notStarted"));
  lifecycle.fireLifecycleEvent(STOP__EVENT, null);
  started = false;
  close ();
}
```

FileLogger 类中最重要的方法是 log 方法，如 Listing7.7 所示。

Listing 7.7: The log method

```
public void log(String msg) {
  // Construct the timestamp we will use, if reguested
  Timestamp ts = new Timestamp(System.currentTimeMillis());
  String tsString = ts.toString().substring(0, 19);
  String tsDate = tsString.substring(0, 10);

  // If the date has changed, switch log files
  if (!date.equals(tsDate)) {
    synchronized (this) {
      if (!date.equals(tsDate)) {
```

```
        close ();
        date = tsDate;
        open ();
      }
    }
  }

  // Log this message, timestamped if necessary
  if (writer != null) {
    if (timestamp) {
      writer.println(tsString + " " + msg);
    }
    else {
      writer.println(msg);
    }
  }
}
```

log 方法接受一个消息并把消息写到日志文件中。在 FileLogger 实例的生命周期中，log 方法可以打开或关闭多个日志文件。如果日期改变了的话，log 方法关闭当前文件并打开一个新文件。接下来看看 open、close 和 log 这些方法是如何工作的。

open 方法

如 Listing7.8 所示，open 方法在指定目录中创建一个新日志文件。

Listing 7.8: The open method

```
private void open() {
  // Create the directory if necessary
  File dir = new File(directory);
  if (!dir.isAbsolute())
    dir = new File(System.getProperty("catalina.base"), directory);
  dir.mkdirs();

  // Open the current log file
  try {
    String pathname = dir.getAbsolutePath() + File.separator +
      prefix + date + suffix;
    writer = new PrintWriter(new FileWriter(pathname, true), true);
  }
  catch (IOException e) {
    writer = null;
  }
```

```
}
```

open 方法首先应该创建日志的目录是否存在，如果目录不存在，则首先创建目录。目录存放在该类的变量中。

```
  File dir = new File(directory);
  if (!dir.isAbsolute())
    dir = new File(System.getProperty("catalina.base"), directory);
  dir.mkdirs();
```

然后组成该文件的路径名，由目录路径、前缀、日期和后缀组成。

```
  try (
    String pathname = dir.getAbsolutePath() + File.separator +
      prefix + date + suffix;
```

接下来构造 java.io.PrintWriter 类的一个实例，然后将该 PringWriter 实例给改了的变量 writer。log 方法使用 writer 来记录信息。

```
   writer = new PrintWriter(new FileWriter(pathname, true), true);
```

The close method

Close 方法清空 PrintWriter 变量 writer，然后关闭 PrintWriter 并将 writer 设置为 null，并将 date 设置为空字符串。该方法如 Listing7.9。

Listing 7.9: The close method

```
private void close() {
  if (writer == null)
    return;
  writer.flush();
  writer.close();
  writer = null;
  date = "";
}
```

log 方法

Log 方法首先创建一个 java.sql.Timestamp 类的实例，该类是 java.util.Date 的瘦包装器 (thin wrapper)。初始化时间戳的目的是更容易的得到当前时间。在该方法中，将当前时间的 long 格式传递给 Timestamp 类并构建 Timestamp 类实例。

```
  Timestamp ts = new Timestamp(System.currentTimeMillis());
```

使用 Timestamp 类的 toString 方法，可以得到当前时间的字符串表示形式，字符串输出的形式如下格式：

yyyy-mm-dd hh:mm: SS.ffffffff

其中 ffffffff 表示从 00:00:00 开始的毫微秒。在方法中使用 subString 方法得到日期和小时。

```
  String tsString = ts.toString().substring(0, 19);
```

接下来使用如下语句得到日期：

```
  String tsDate = tsString.substring(0, 10);
```

接下来 log 方法比较 tsData 和 String 变量 date 的值，如果 tsDate 和 date 的值不同,它关闭当前日志文件,将 tsDate 的值赋给 date 并打开一个新日志文件。

```
// If the date has changed, switch log files
  if (!date.equals(tsDate)) {
    synchronized (this) {
      if (!date.equals(tsDate)) {
        close();
        date = tsDate;
        open();
      }
    }
  }
```

最后，日志方法将 PrintWriter 实例的输出流写入到日志文件中。如果布尔变量 timestamp 的值为真,将 timestamp(tsString)的值作为前缀,否则不使用前缀。

```
  // Log this message, timestamped if necessary
  if (writer != null) {
    if (timestamp) {
      writer.println(tsString + " " + msg);
    }
    else {
      writer.println(msg);
    }
  }
```

The Application

该章的应用程序跟第六章的程序很相似,只是多了个跟 SimpleContext 对象相关的 FileLogger。程序的改变可以在 ex07.pyrmont.startup.Bootstrap 类的主方法里找到，如 Listing7.10 所示。注意要仔细看高亮的代码。

Listing 7.10: The Bootstrap class

```
package ex07.pyrmont.startup;

import ex07.pyrmont.core.SimpleContext;
import ex07.pyrmont.core.SimpleContextLifecycleListener;
import ex07.pyrmont.core.SimpleContextMapper;
import ex07.pyrmont.core.SimpleLoader;
```

```
import ex07.pyrmont.core.SimpleWrapper;
import org.apache.catalina.Connector;
import org.apache.catalina.Context;

import org.apache.catalina.Lifecycle;
import org.apache.catalina.LifecycleListener;
import org.apache.catalina.Loader;
import org.apache.calalina.loggor.FileLogger;
import org.apache.catalina.Mapper;
import org.apache.catalina.Wrapper;
import org.apache.catalina.connector.http.HttpConnector;

public final class Bootstrap {
  public static void main(String[] args) {
    Connector connector = new HttpConnector();
    Wrapper Wrapper1 = new SimpleWrapper();
    Wrapper1.setName("Primitive");
    Wrapper1.setServletClass("PrimitiveServlet");
    Wrapper wrapper2 = new SimpleWrapper();
    wrapper2.setName("Modern");
    Wrapper2.setServletClass("ModernServlet");
    Loader loader = new SimpleLoader();

    Context context = new SimpleContext();
    context.addChild(wrapper1);
    context.addChild(wrapper2);

    Mapper mapper = new SimpleContextMapper();
    mapper.setProtocol("http");
    LifecycleListener listener = new SimpleContextLifecycleListener();
    ((Lifecycle) context).addLifecycleListener(listener);
    context.addMapper(mapper);
    context.setLoader(loader);
    // context.addServletMapping(pattern, name);
    context.addServletMapping("/Primitive", "Primitive");
    context.addServletMapping("/Modern", "Modern");

    // ------ add logger --------
    System.setProperty("catalina.base",
System.getProperty("user.dir"));
    FileLogger logger = new FileLogger();
    logger.setPrefix("FileLog_");
    logger.setSuffix(".txt");
    logger.setTimestamp(true);
```

```
    logger.setDirectory("webroot");
    context.setLogger(logger);

    //-------------------------

    connector.setContainer(context);
    try {
      connector.initialize();
      ((Lifecycle) connector).start();
      ((Lifecycle) context).start();

      // make the application wait until we press a key.
      System.in.read();
      ((Lifecycle) context).stop();
    }
    catch (Exception e) {

      e.printStackTrace();
    }
  }
}
```

总结

在本章中，学习了日志组件，并介绍了 org.apache.catalina.Logger 接口，并仔细的分析了 Tomcat 中对于 Logger 接口中的三个实现类。另外，使用应用程序来说明了 FileLogger 类的使用。

第八章：加载器

综述

在前面的章节中已经介绍了一个简单的加载器，用它来加载 servlet 类。这一章会介绍标准网络应用加载器(standard web application loader)，简单的说就是加载器。一个 servlet 容器需要一个定制的容器，而不是简单的使用系统的加载器。如果像前面章节中那样使用系统的加载器来加载 servlet 和其他需要的类，这样 servlet 就可以进入 Java 虚拟机 CLASSPATH 环境下面的任何类和类库，这会带来安全隐患。Servlet 只允许访问 WEB-INF/目录及其子目录下面的类以及部署在 WEB-INF/lib 目录下的类库。所以一个 servlet 容器需要一个自己的加载器，该加载器遵守一些特定的规则来加载类。在 Catalina 中，加载器使用 org.apache.catalina.Loader 接口表示。

Tomcat 需要一个自己的加载器的另一个原因是它需要支持在 WEB-INF/classes 或者是 WEB-INF/lib 目录被改变的时候会重新加载。Tomcat 的加载器实现中使用一个单独的线程来检查 servlet 和支持类文件的时间戳。要支持类的自动加载功能，一个加载器类必须实现 org.apache.catalina.loader.Reloader 接口。

本章的第一节先简要的回顾下 Java 的类加载机制。接下来介绍了加载器必须实现的 Loader 接口，然后是 Reloader 接口。接下来看到的是加载器的实现，最后本章使用一个程序说明了怎么使用 Tomcat 类的加载器。

本章广泛使用的是两个词：库（repository）和源（resources）。库表示加载器查找的地方,源表示加载器中的 DirContext 对象,它的文档基(document base)指向了上下文的文档基。

Java 类加载器

在每次创建一个 Java 类的实例时候，必须先将该类加载到内存中。Java 虚拟机（JVM）使用类加载器来加载类。Java 加载器在 Java 核心类库和 CLASSPATH 环境下面的所有类中查找类。如果需要的类找不到，会抛出 java.lang.ClassNotFoundException 异常。

从 J2SE1.2 开始，JVM 使用了三种类加载器：bootstrap 类加载器、extension 类加载器和 systen 类加载器。这三个加载器是父子关系，其中 bootstrap 类加载器在顶端，而 system 加载器在结构的最底层。

其中 bootstrap 类加载器用于引导 JVM，一旦调用 java.exe 程序，bootstrap 类加载器就开始工作。因此，它必须使用本地代码实现，然后加载 JVM 需要的类到函数中。另外，它还负责加载所有的 Java 核心类，例如 java.lang 和 java.io 包。另外 bootstrap 类加载器还会查找核心类库如 rt.jar、i18n.jar 等，这些类库根据 JVM 和操作系统来查找。

extension 类加载器负责加载标准扩展目录下面的类。这样就可以使得编写程序变得简单，只需把 JAR 文件拷贝到扩展目录下面即可，类加载器会自动的在下面查找。不同的供应商提供的扩展类库是不同的，Sun 公司的 JVM 的标准扩展目录是/jdk/jre/lib/ext。

system 加载器是默认的加载器，它在环境变量 CLASSPATH 目录下面查找相应的类。

这样，JVM 使用哪个类加载器？答案在于委派模型(delegation model)，这是出于安全原因。每次一类需要加载，system 类加载器首先调用。但是，它不会马上加载类。相反，它委派该任务给它的父类-extension 类加载器。extension 类加载器也把任务委派给它的父类 bootstrap 类加载器。因此，bootstrap 类加载器总是首先加载类。如果 bootstrap 类加载器不能找到所需要的类的 extension 类加载器会尝试加载类。如果扩展类加载器也失败，system 类加载器将执行任务。如果系统类加载器找不到类，一个 java.lang.ClassNotFoundException 异常。为什么需要这样的往返模式？
委派模型对于安全性是非常重要的。如你所知，可以使用安全管理器来限制访问某个目录。现在，恶意的意图有人能写出一类叫做 java.lang.Object，可用于访问任何在硬盘上的目录。因为 JVM 的信任 java.lang.Object 类，它不会关注这方面的活动。因此，如果自定义 java.lang.Object 被允许加载的安全管理器将很容易瘫痪。幸运的是，这将不会发生，因为委派模型会阻止这种情况的发生。下面是它的工作原理。

当自定义 java.lang.Object 类在程序中被调用的时候，system 类加载器将该请求委派给 extension 类加载器，然后委派给 bootstrap 类加载器。这样 bootstrap 类加载器先搜索的核心库，找到标准 java.lang.Object 并实例化它。这样，自定义 java.lang.Object 类永远不会被加载

关于在 Java 类加载机制的优势在于可以通过扩展

java.lang.ClassLoader 抽象类来扩展自己的类加载器。Tomcat 的需求

自定义自己的类加载器原因包括以下内容

- 要制定类加载器的某些特定规则
- 缓存以前加载的类
- 事先加载类以预备使用

Loader 接口

在 Web 应用程序中加载 servlet 和其他类需要遵循一些规则。例如，在一个应用程序中 Servlet 可以使用部署到 WEB-INF/classes 目录和任何子目录下面的类。然而，没有 servlet 的不能访问其他类，即使这些类是在运行 Tomcat 所在的 JVM 的 CLASSPATH 中。此外，一个 servlet 只能访问 WEB-INF/lib 目录下的类库，而不能访问其他目录下面的。

一个 Tomcat 类加载器表示一个 Web 应用程序加载器，而不是一个类加载器。一个加载器必须实现 org.apache.catalina.Loader 接口。加载器的实现使用定制的类加载器 org.apache.catalina.loader.WebappClassLoader。可以使用 Loader 接口的 getClassLoader 方法获取一个网络加载器 ClassLoader。

值得一提的是 Loader 接口定义了一系列方法跟库协作。Web 应用程序的 WEB-INF/classes 和 WEB-INF/lib 目录作为库添加上。Loader 接口的 addReposity 方法用于添加一个库，findRepositories 方法用于返回一个所有库的队列。

一个 Tomcat 的加载器通常跟一个上下文相关联，Loader 接口的和 getContainer 及 setContainer 方法是建立此关联。一个加载器还可以支持重新加载，如果在上下文中的一个或多个类已被修改。这样，一个 servlet 程序员可以重新编译 servlet 或辅助类，新类将被重新加载而不需要不重新启动 Tomcat 加载。为了达到重新加载的目的，Loader 接口有修改方法。在加载器的实现中，如果在其库中一个或多个类别已被修改，modeify 方法必须返回 true，因此需要重新加载。一个加载器自己进行重新加载，而是调用上下文接口的重载方法。另外两种方法，setReloadable 和 getReloadable，用于确定加载器中是否可以使用重加载。默认情况下，在标准的上下文实现中（org.apache.catalina.core.StandardContext 类将在第 12 章讨论）重载机制并未启用。因此，要使得上下文启动重载机制，需要在 server.xml 文件添加一些元素如下：

```
<Context path="/myApp" docBase="myApp" debug="0" reloadable="true"/>
```

另外，一个加载器的实现可以确定是否委派给父加载器类。为了实现这一点，Loader 接口提供了 getDelegate 和 setDelegate 方法。

Loader 接口如 Listing8.1 所示

Listing 8.1: The Loader interface

```
package org.apache.catalina;
import java.beans.PropertyChangeListener;

public interface Loader {
  public ClassLoader getClassLoader();
  public Container getContainer();
  public void setContainer(Container container);
  public DefaultContext getDefaultContext();
  public void setDefaultContext(DefaultContext defaultContext);
  public boolean getDelegate();
  public void setDelegate(boolean delegate);
  public String getInfo();
  public boolean getReloadable();
  public void setReloadable(boolean reloadable);
  public void addPropertyChangeListener(PropertyChangeListener

    listener);
  public void addRepository(String repository);
  public String[] findRepositories();
  public boolean modified();
  public void removePropertyChangeListener(PropertyChangeListener
```

```
    listener);
}
```

Catalina 提供了 org.apache.catalina.loader.WebappLoader 作为 Load 接口的实现。WebappLoader 对象包含一个 org.apache.catalina.loader.WebappClassLoader 类的实例，该类扩展了 Java.netURLClassLoader 类。

注意 无论一个跟容器相关的加载器何时需要一个 servlet 类，当它的 invoke 方法被调用的时候，容器首先调用加载器的 getClassLoader 方法获得一个加载器。然后容器调用 loadClass 方法来加载 servlet 类，更多的细节会在第 11 章中介绍。

Loader 接口和它的实现类的结构图如图 8.1

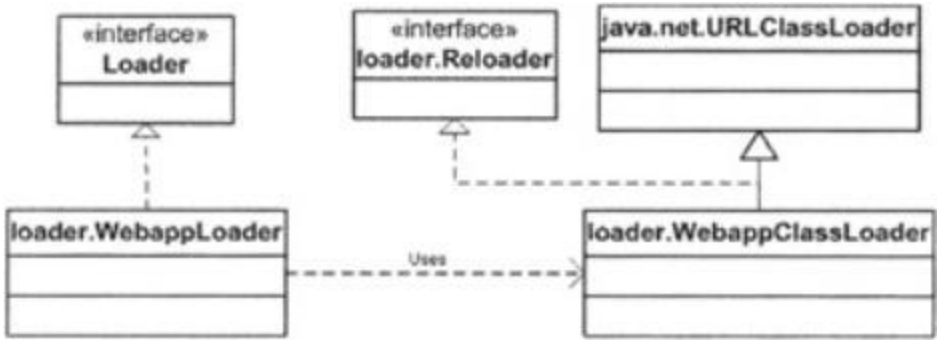


Figure 8.1: The Loader interface and its implementation

Reloader 接口

要支持自动重新加载，一个加载器的实现必须实现 org.apache.catalina.loader.Reloader 接口，该接口如 Listing8.2

Listing 8.2: The Reloader interface

```
package org.apache.catalina.loader;
public interface Reloader {
  public void addRepository(String repository);
  public String[] findRepositories ();

  public boolean modified();
}
```

Reloader 接口里最重要的方法是 modified 方法，如果在 web 应用程序中的 servlet 任何支持类被修改的时候该方法返回 true。addRepository 方法用于添加一个库而 findRepositories 方法用于返回实现了 Reloader 接口的加载器的所有的库。

WebappLoader 类

org.apache.catalina.loader.WebappLoader 类是 Loader 接口的实现，它表示一个 web 应用程序的加载器，负责给 web 应用程序加载类。WebappLoader 创建一个 org.apache.catalina.loader.WebappClassLoader 类的实例作为它的类加载器。像其他的 Catalina 组件一样，WebappLoader 实现了 org.apache.catalina.Lifecycle 接口，可有由关联容器启动和停止。WebappLoader 类还实现了 java.lang.Runnable 接口，所以可以通过一个线程来重复的调用 modified 方法，如果 modified 方法返回 true，WebappLoader 实例同志它的关联容器。类通过上下文重新加载自己，而不是 WebappLoader。上下文的怎么实现该功能会在第 12 章，标准上下文中介绍。

WebappLoader 类的 start 方法被调用的时候，将会完成下面几项重要任务：

- Creating a class loader
- Setting repositories
- Setting the class path
- Setting permissions
- Starting a new thread for auto-reload.
- 创建一个类加载器
- 设置库
- 设置类路径
- 设置访问权限
- 开启一个新线程用来进行自动重载

在接下来的内容中会讨论这些任务。

创建类加载器

WebappLoader 使用一个内部类加载器来加载类。可以回头看 Loader 接口，该接口提供了 getClassLoader 方法但是并没有 setClassLoader 方法。因此，不能通过传递一个 WebappLoader 来初始化它。这样没有默认类加载器是否意味着 WebappLoader 不够灵活的？

答案当然是否定的，WebappLoader 类提供了 getLoaderClass 和 setLoaderClass 方法来获得或者改变它的私有变量 loaderClass 的值。该变量是一个的表示加载器类名 String 类型表示形式。默认的 loaderClass 值是 org.apahce.catalina.loader.WebappClassLoader，如果你愿意，可以创建继承 WebappClassLoader 类的自己的加载器，然后使用 setLoaderClass 方法来强制 WebappLoader 使用你创建的加载器。否则，当它 WebappLoader 启动的时候，它会使用它的私有方法 createClassLoader 创建 WebappClassLoader 的实例，该方法如 Listing8.3 所示：

Listing 8.3: The createClassLoader method

```
private WebappClassLoader createClassLoader() throws Exception {
  Class clazz = Class.forName(loaderClass);
  WebappClassLoader classLoader = null;
  if (parentClassLoader == null) {
    // Will cause a ClassCast if the class does not extend
    // WebappClassLoader, but this is on purpose (the exception will be
    // caught and rethrown)
    classLoader = (WebappClassLoader) clazz.newInstance();
    // in Tomcat 5, this if block is replaced by the following:
    // if (parentClassLoader == null) {
    //   parentClassLoader =
    //     Thread.currentThread().getContextClassLoader();
    // }

  }
  else {
    Class[] argTypes = { ClassLoader.class };
    Object[] args = { parentClassLoader };
    Constructor constr = clazz.getConstructor(argTypes);
    classLoader = (WebappClassLoader) constr.newInstance(args);
  }
  return classLoader;
}
```

也可以不使用 WebappClassLoader 而用其他的类，但是注意 createClassLoader 的返回值类型是 WebappClassLoader，所以你的类必须继承 WebappClassLoader 类，否则该方法会抛出异常。

设置库

WebappLoader 的 start 方法会调用 setRepositories 方法来给类加载器添加一个库。WEB-INF/classes 目录传递给加载器 addRepository 方法，而 WEB-INF/lib 传递给加载器的 setJarPath 方法。这样，类加载器能能从 WEB-INF/classes 目录下面和 WEB-INF/lib 目录下面部署的类库里加载类。

设置类路径

该任务由 start 方法调用 setClassPath 方法完成，setClassPath 方法会给 servlet 上下文分配一个 String 类型属性保存 Jasper JSP 编译的类路径，该内容先不予讨论。

设置访问权限

如果 Tomcat 使用了安全管理器，setPermissions 给类加载器给必要的目录添加访问权限，例如 WEB-INF/classes 和 WEB-INF/lib。如果不使用管理器，该方法马上返回。

开启自动重载线程

WebappLoader 支持自动重载，如果 WEB-INF/classes 或者 WEB-INF/lib 目录被重新编译过，在不重启 Tomcat 的情况下必须自动重新载入这些类。为了实现这个目的，WebappLoader 有一个单独的线程每个 x 秒会检查源的时间戳。x 的值由 checkInterval 变量定义，它的默认值是 15，也就是每隔 15 秒会进行一次检查是否需要自动重载。该类还提供了两个方法 getCheckInterval 和 setCheckInterval 方法来访问或者设置 checkInterval 的值。

在 Tomcat4 中,WebappLoader 实现了 java.lang.Runnable 接口来支持自动重载。WebappLoader 对 run 方法的实现如 Listing8.3 所示：

Listing 8.3: The run method

```
public void run() {
  if (debug >= 1)
    log("BACKGROUND THREAD Starting");

  // Loop until the termination semaphore is set
  while (!threadDone) {
    // Wait for our check interval
    threadSleep();
    if (!started)
      break;
    try {
      // Perform our modification check
      if (!classLoader.modified())
        continue;
    }
    catch (Exception e) {
      log(sm.getString("webappLoader.failModifiedCheck"), e);
      continue;
    }
    // Handle a need for reloading
    notifyContext();
    break;
  }

  if (debug >= 1)
    log("BACKGROUND THREAD Stopping");
}
```

注 在 Tomcat5 中检查类是否被修改的任务由

意 org.apache.catalina.core.StandardContext 类的 backgroundProcess 方法完成。该方法会被 org.apache.catalina.core.ContainerBase 类中一个专门的线程周期性的调用，ContainerBase 是 StandardContext 类的父类。注意 ContainerBase 类的 ContainerBackgroundProcessor 内部类实现的 Runnable 接口。

Listing8.3 所示的 run 方法作为该线程的核心部分，包括一个 while 循环知道 started 变量为 false。该 while 循环完成了下面的工作：

- 休眠由 checkInterval 变量定义的一段时间
- 检查是否有类被改变，如果有责调用 WebappLoader 实例的 modified 方法，否则继续循环。
- 如果一个类被修改了，调用私有方法 notifyContext 来让跟 WebappLoader 实例相关联的上下文重新载入。

方法 notifyContext 如 Listing8.4 所示：

Listing 8.4: The notifyContext method

```
private void notifyContext() {
  WebappContextNotifier notifier = new WebappContextNotifier();
  (new Thread(notifier)).start();
}
```

方法 notifyContext 并不是直接调用 Context 接口中的 reload 方法，它首先初始化一个内部类 WebappContextNotifier 的实例，并把它传递给一个线程对象，调用它的 start 方法。这样重载的提交就由另一个线程完成，WebappContextNotifier 类如 Listing8.5 所示：

Listing 8.5: The WebappContextNotifier inner class

```
protected class WebappContextNotifier implements Runnable {
  public void run() {
    ((Context) container).reload();
  }
}
```

当 WebappContextNotifier 类的一个实例被传递给一个线程的时候，该线程的 start 方法被唤醒，WebappContextNotifier 实例的 run 方法会被执行。然后，run 方法调用 Context 接口的 reload 方法。可以在第 12 章看到 org.apache.catalina.core.StandardContext 类是如何实现 reload 方法的。

WebappClassLoader 类

类 org.apache.catalina.loader.WebappClassLoader 表示在一个 web 应用程序中使用的加载器。WebappClassLoader 类继承了 java.net.URLClassLoader 类，该类在前面章节中用于加载 Java 类。

WebappClassLoader 被可以的进行了优化和安全方面的考虑。例如它缓存了以前加载的类以改进性能，下一次收到第一次没有找到的类的请求的时候，可以直接抛出 ClassNotFound 异常。WebappClassLoader 在源列表以及特定的 JAR 文件中查找类。

处于安全性的考虑，WebappClassLoader 类不允许一些特定的类被加载。这些类被存储在一个 String 类型的数组中，现在仅仅有一个成员。

```
private static final String[] triggers = {
  "javax.servlet.Servlet"                       // Servlet API
};
```

另外在委派给系统加载器的时候，你也不允许加载属于该包的其它类或者它的子包：

```
private static final String[] packageTriggers = {
  "javax",                                      // Java extensions
  "org.xml.sax",                                // SAX 1 & 2
  "org.w3c.dom",                                // DOM 1 & 2
  "org.apache.xerces",                          // Xerces 1 & 2
  "org.apache.xalan"                            // Xalan
};
```

接下来让我们看看该类是如何实现缓存和加载的。

缓存

为了提高性能，当一个类被加载的时候会被放到缓存中，这样下次需要加载该类的时候直接从缓存中调用即可。缓存由 WebappClassLoader 类实例自己管理。另外，java.lang.ClassLoader 维护了一个 Vector，可以避免前面加载过的类被当做垃圾回收掉。在这里，缓存被该超类管理。

每一个可以被加载的类(放在 WEB-INF/classes 目录下的类文件或者 JAR 文件)都被当做一个源。一个源被 org.apache.catalina.loader.ResourceEntry 类表示。一个 ResourceEntry 实例保存一个 byte 类型的数组表示该类、最后修改的数据或者副本等等。

ResourceEntry 类如 Listing8.6 所示：

Listing 8.6: The ResourceEntry class.

```
 package org.apache.catalina.loader;
import java.net.URL;
import java.security.cert.Certificate;
import java.util.jar.Manifest;

public class ResourceEntry {

  public long lastModifled = -1;
  // Binary content of the resource.
```

```
  public byte[] binaryContent = null;
  public Class loadedClass = null;
  // URL source from where the object was loaded.
  public URL source = null;
  // URL of the codebase from where the object was loaded.
  public URL CodeBase = null;
  public Manifest manifest = null;
  public Certificate[] certificates = null;
}
```

所有缓存的源被存放在一个叫做 resourceEntries 的 HashMap 中，键值为源名，所有找不到的源都被放在一个名为 notFoundResources 的 HashMap 中。

加载类

当加载一个类的时候，WebappClassLoader 类遵循以下规则：

- 所有加载过的类都要进行缓存，所以首先需要检查本地缓存。
- 如果无法再本地缓存找到类，使用 java.langClassLoader 类的 findLoaderClass 方法在缓存查找类、
- 如果在两个缓存中都无法找到该类，使用系统的类加载器避免从 J2EE 类中覆盖来的 web 应用程序。
- 如果使用了安全管理器，检查该类是否允许加载，如果该类不允许加载，则抛出 ClassNotFoundException 异常。
- 如果要加载的类使用了委派标志或者该类属于 trigger 包中，使用父加载器来加载类，如果父加载器为 null，使用系统加载器加载。
- 从当前的源中加载类
- 如果在当前的源中找不到该类并且没有使用委派标志，使用父类加载器。如果父类加载器为 null，使用系统加载器
- 如果该类仍然找不到，抛出 ClassNotFoundException 异常

应用程序

本章的应用程序演示类如何使用一个跟上下文相关联的 WebappLoader 加载器。一个上下问的标准实现是 org.apache.catalina.core.StandardContext，所以该应用程序使用了 StandardContext 类。但是关于 StarndardContext 类在 12 章中才会详细讨论。这里不需要了解该类的细节，只需要知道它有些监听器和可以触发的事件即可，例如 START_EVENT 和 STOP_EVENT，监听器必须实现 org.apache.catalina.lifecycle.LifecycleListener 接口并且调用 StandardContext 类的 setConfigured 方法。在该应用程序中，使用 ex08.pyrmont.core.SimpleContextConfig 类表示监听器，如 Listing8.6 所示。

Listing 8.6: The SimpleContextConfig class

```
package ex08.pyrmont.core;
import org.apache.catalina.Context;
import org.apache.catalina.Lifecycle;
import org.apache.catalina.LifecycleEvent;
import org.apache.catalina.LifecycleListener;

public class SimpleContextConfig implements LifecycleListener {
  public void lifecycleEvent(LifecycleEvent event) {
    if (Lifecycle.START_EVENT.equals(event.getType())) {
      Context context = (Context) event.getLifecycle();
      context.setConfigured(true);
    }
  }
}
```

你需要做的仅仅是初始化一个 StandardContext 和 SimpleContextConfig 对象，然后调用 org.apache.catalina.Lifecycle 接口的 addLifecycleListener 方法来注册它。

另外，该应用程序使用了前面章节的类：SimplePipeline, SimpleWrapper, 和 SimpleWrapperValve

该应用程序可以使用 PrimitiveServlet 和 ModernServlet 测试，但是使用 StandardContext 将 WEB-INF/classes 下面的 servlets 存储起来。该应用程序的目录名为 myApp，如果你下载了 ZIP 文件需要该文件夹应该已经创建好了。设置系统属性 catalina.base 的值为 user.dir，这样可以告诉 StandardContext 实例查找应用程序目录。

```
System.setProperty("catalina.base", System.getProperty("user.dir"));
```

实际上，它是 Bootstrap 类的第一行，然后主方法初始化连接器：

```
Connector connector = new HttpConnector();
```

然后初始化两个 servlet 的两个包装器，如下：

```
Wrapper wrapper1 = new SimpleWrapper();
wrapper1.setName("Primitive");
wrapper1.setServletClass("PrimitiveServlet");
Wrapper wrapper2 = new SimpleWrapper();
wrapper2.setName("Modern");
wrapper2.setServletClass("ModernServlet");
```

然后创建了一个 StandardContext 得实例并设置该上下文的文档基（document base）.

```
Context context = new StandardContext();
// StandardContext's start method adds a default mapper
context.setPath("/myApp");
```

```
context.setDocBase("myApp");
```

这样做跟设置如下设置 server.xml 效果相同：

```
<Context path="/myApp" docBase="myApp"/>
```

然后，两个包装器被添加到上下文中，并且添加映射关系，这样上下文就可以找到包装器。

```
context.addChild(wrapper1);
context.addChild(wrapper2);
context.addServletMapping("/Primitive", "Primitive");
context.addServletMapping("/Modern", "Modern");
```

接下来的工作是初始化一个监听器并且在上下文中注册

```
LifecycleListener listener = new SimpleContextConfig();
((Lifecycle) context).addLifecycleListener(listener);
```

接下来初始化该上下文相关联的 WebappLoader:

```
Loader loader = new WebappLoader();
context.setLoader(loader);
```

R 然后建立连接器跟上下文的关联，然后调用初始化和 start 方法，接下来是上下文的 start 方法，这样 servlet 容器就可以工作了。

```
connector.setContainer(context);
try {
  connector.initialize();
  ((Lifecycle) connector).start();

  ((Lifecycle) context).start();
```

接下来的几行是打印出源的 docBase 已经改加载器的所有源：

```
  // now we want to know some details about WebappLoader
  WebappClassLoader classLoader = (WebappClassLoader)
    loader.getClassLoader();
  System.out.println("Resources' docBase: " +
   ((ProxyDirContext)classLoader.getResources ()).getDocBase());
  String[] repositories = classLoader.findRepositories();
  for (int i=0; i<repositorles.length; i++) {
    System.out.println("  repository: " + repositories[i]);
  }
```

当运行应用程序的时候会显示下面几行：

```
Resources' docBase: C:\HowTomcatWorks\myApp
  repository: /WEB-INF/classes/
```

The value of docBase may be different on your machine, depending where you install the application.

docBase 的值可能会有不同，这取决于该应用程序被部署在哪里。

最后，该应用程序等待用户输入终止命令来停止应用程序。

```
// make the application wait until we press a key.
System.in.read();
((Lifecycle) context).stop();
```

Running the Application

在 Windows 下面可以在工作目录下面使用如下命令启动该程序：
java -classpath ./lib/servlet.jar;./lib/commons-collections.jar;./ex08.pyrmont.startup.Bootstrap

在 Linux 下面使用冒号分开两个库：

java -classpath ./lib/servlet.jar:./lib/commons-collections.jar:./ex08.pyrmont.startup.Bootstrap

可以在浏览器中输入如下 URL 调用 PrinitiveServlet servlet。

http://localhost:8080/Primitive

要调用 ModernServlet，可以使用如下 URL：

http://localhost:8080/Modern

总结

一个应用程序加载器，简单的说就是加载器是 Catalina 中最重要的组件之一。它使用一个内部的类加载器来完成加载类的工作。Tomcat 使用该内部类加载器加载应用类，它属于一个应用上下文并且遵循一系列规则。另外，该加载器还支持缓存以及检测类修改情况的功能。

第九章：session 管理

综述

Catalina 通过一个叫管理器的组件来完成 session 管理工作，该组件由 org.apache.catalina.Manager interface 接口表示。一个管理器通常跟一个上下文容器相关联，它负责创建、更行以及销毁 session 对象并能给任何请求组件返回一个合法的 session。

一个 servlet 可以使用 getSession 方法获得一个 session 对象，该方法在 javax.servlet.http.HttpServletRequest 定义。它在默认连接器里由 org.apache.catalina.connector.HttpRequestBase 类实现。这里是 HttpRequestBase 类的一些相关方法。

```
public HttpSession getSession() {
  return (getSession(true));
}
public HttpSession getSession(boolean create) {
  ...
  return doGetSession(create);
}
private HttpSession doGetSession(boolean create) {
  // There cannot be a session if no context has been assigned yet
  if (context == null)
    return (null);
  // Return the current session if it exists and is valid
  if ((session != null) && !session.isValid())
    session = null;
  if (session != null)
    return (session.getSession());

  // Return the requested session if it exists and is valid
  Manager manager = null;
  if (context != null)
    manager = context.getManager();
  if (manager == null)
    return (null);      // Sessions are not supported
  if (requestedSessionId != null) {
    try {
      session = manager.findSession(requestedSessionId);
    }

    catch (IOException e) {
      session = null;
    }
```

```
    if ((session != null) && !session.isValid())
      session = null;
    if (session != null) {
      return (session.getSession());
    }
  }

  // Create a new session if requested and the response is not
  // committed
  if (!create)
    return (null);
  ...
  session = manager.createSession();
  if (session != null)
    return (session.getSession());
  else
    return (null);
}
```

默认情况下管理器将 session 对象存储在内存中,但是 Tomcat 也允许将 session 对象存储在文件或者数据库中（通过 JDBC）。Catalina 在 org.apache.catalina.session 包中提供了 session 对象和 session 管理的相关类型。

本章使用了三节解释了 session 管理：Session、Managers 和 Stores。最后一节介绍了一个使用上下文容器以及相关联的管理器的应用程序。

Sessions

在 servlet 编程中，一个 session 对象使用 javax.servlet.http.HttpSession 接口表示。该接口的标准实现是 StandardSession 类，该类在 org.apache.catalina.session 包中。但是出于安全的原因，管理器并不会将一个 StandardSession 实例传递给 servlet。而是使用 org.apache.catalina.session 包中的外观类 StandardSession 类。在内部，一个管理器使用了另一个外观：org.apache.catalina.Session 接口。Session 相关类型的 UML 结构图如图 9。1，注意，出于简便的考虑，并没有将 Session, StandardSession, 和 StandardSessionFacade 前面的类添加在里面。

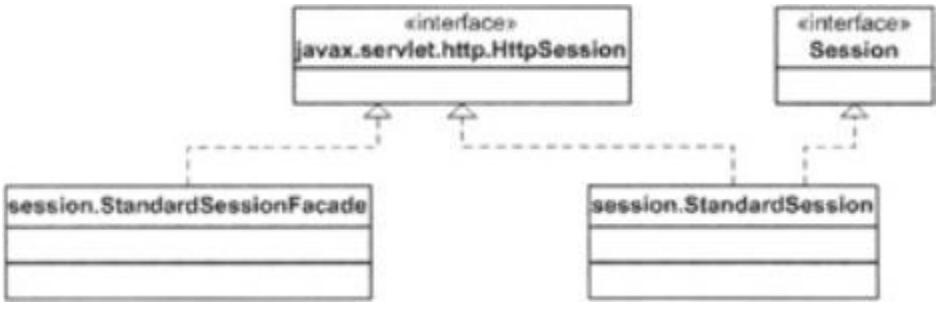


Figure 9.1: Session-related types

Session 接口

Session 接口扮演了一个 Catalina 内部外观的角色。它的标准实现 StandardSession 还是先了 javax.servlet.http.HttpSession 接口，Session 接口如Listing9.1所示：

Listing 9.1: The Session interface

```
package org.apache.catalina;
import java.io.IOException;
import java.security.Principal;
import java.util.Iterator;
import javax.servlet.ServletException;
import javax.servlet.http.HttpSession;

public interface Session {
  public static final String SESSION_CREATED_EVENT = "createSession";
  public static final String SESSION_DESTROYED_EVENT =
    "destroySession";
  public String getAuthType();
  public void setAuthType(String authType);
  public long getCreationTime();
  public void setCreationTime(long time);
  public String getId();
  public void setId(String id);
  public String getInfo();
  public long getLastAccessedTime();
  public Manager getManager();
  public void setManager(Manager manager);
  public int getMaxInactiveInterval();
  public void setMaxInactiveInterval(int interval);
  public void setNew(boolean isNew);
  public Principal getPrincipal();
  public void setPrincipal(Principal principal);
  public HttpSession getSession();
  public void setValid(boolean isValid);
  public boolean isValid();

  public void access();
  public void addSessionListener(SessionListener listener);
  public void expire();
  public Object getNote(String name);
  public Iterator getNoteNames();
  public void recycle();
  public void removeNote(String name);
```

```
  public void removeSessionListener(SessionListener listener);
  public void setNote(String name, Object value);
}
```

由于一个 Session 对象常常被一个管理器持有，所以接口提供了 setManager 和 getManager 方法来关联一个 Session 对象和一个管理器。另外，一个 Session 实例在跟管理器相关联的容器有一个唯一的 ID。对于该 ID 有 setId 和 getId 方法相关。getLastAccessedTime 方法由管理器来调用，以确定一个 Session 对象是否合法。管理器调用 setValid 方法来重置一个 session 的合法性。每次一个 Session 被进入的时候，都会调用 access 方法更新它的最后访问时间。最后，管理器可以调用 expire 方法来终止一个 expire 方法，使用 getSession 可以获得一个包装在该外观内的 HttpSession 对象。

StandardSession 类

StandardSession 类是 Session 接口的标准是实现。另外，实现了 javax.servlet.http.HttpSession 和 org.apache.catalina.Session 之外，它还实现了 java.lang.Serializable 接口来使得 Session 对象可序列化。

该类的构造器获得一个管理器实例来强制使得每个 Session 对象都有一个管理器。

```
public StandardSession(Manager manager);
```

接下来是几个重要的变量在存放 Session 状态。注意 transient 使得该关键字不可序列化。

```
// session attributes
private HashMap attributes = new HashMap();
// the authentication type used to authenticate our cached Principal,
if any
private transient String authType = null;
private long creationTime = 0L;
private transient boolean expiring = false;
private transient StandardSessionFacade facade = null;
private String id = null;
private long lastAccessedTime = creationTime;
// The session event listeners for this Session.

private transient ArrayList listeners = new ArrayList();
private Manager manager = null;
private int maxInactiveInterval = -1;
// Flag indicating whether this session is new or not.
private boolean isNew = false;
private boolean isValid = false;
private long thisAccessedTime = creationTime;
```

注意 在 Tomcat5 中上述变量都是 protected，在 Tomcat 4 中是 private 的。每个变量都有一个 set/get 方法。

方法 getSession 首先会将该实例创建一个 StandardSessionFacade 对象

```
public HttpSession getSession() {
  if (facade == null)
    facade = new StandardSessionFacade(this);
  return (facade);
}
```

一个 Session 对象如果在由 maxInactiveInterval 变量的时间内没有被进入则被终结。使用 Session 接口中定义的 expire 方法可以终结一个 Session 对象。该方法在 Tomcat4 中 StandardSession 中的实现如 Listing9.2

Listing 9.2: The expire method

```
public void expire(boolean notify) {
  // Mark this session as "being expired" if needed
  if (expiring)
    return;
  expiring = true;
  setValid(false);

  // Remove this session from our manager's active sessions
  if (manager != null)
    manager.remove(this);
  // Unbind any objects associated with this session
  String keys [] = keys();
  for (int i = 0; i < keys.length; i++)
    removeAttribute(keys[i], notify);
  // Notify interested session event listeners
  if (notify) {
    fireSessionEvent(Session.SESSION_DESTROYED_EVENT, null);
  }

  // Notify interested application event listeners
  // FIXME - Assumes we call listeners in reverse order
  Context context = (Context) manager.getContainer();
  Object listeners[] = context.getApplicationListeners();
  if (notify && (listeners != null)) {
    HttpSessionEvent event = new HttpSessionEvent(getSession());

    for (int i = 0; i < listeners.length; i++) {
      int j = (listeners.length - 1) - i;
      if (!(listeners[j] instanceof HttpSessionListener))
```

```
        continue;
      HttpSessionListener listener =
       (HttpSessionListener) listeners[j];
      try {
        fireContainerEvent(context, "beforeSessionDestroyed",
          listener);
        listener.sessionDestroyed(event);
        fireContainerEvent(context, "afterSessionDestroyed",
listener);
      }
      catch (Throwable t) {
        try {
          fireContainerEvent(context, "afterSessionDestroyed",
            listener);
        }
        catch (Exception e) {
          ;
        }
        // FIXME - should we do anything besides log these?
        log(sm.getString("standardSession.sessionEvent"), t);
      }
    }
  }

  // We have completed expire of this session
  expiring = false;
  if ((manager != null) && (manager instanceof ManagerBase)) {
    recycle();
  }
}
```

在 Listing9.2 的过程中包括设置内部变量、删除管理器的 Session 对象、并触发一个事件。

StandardSessionFacade 类

要将一个 Session 对象传递给一个 servlet，Catalina 会初始化一个 StandardSession 类填充它并把它传递给 servlet。但是它传递的是 StandardSession 对象，该类是先了 javax.servlet.http.HttpSession 接口中的方法。这样，servlet 就不能将 HttpSession 向下转化为 StandardSessionFacade 类来访问它的共有方法。

管理器

管理器用来管理 Session 对象。例如它创建 Session 对象并销毁它们。管理器由 org.apache.catalina.Manager 接口表示。在 Catalina 中，org.apache.catalina.session 包中类 ManagerBase 类提供了常用函数的基本实现。ManagerBase 类有两个直接子类：StandardManager 和 PersistentManagerBase 类。

在运行的时候，StandardManager 将 session 对象存放在内存中。但是，当停止的时候，它将 Session 对象存放到文件中。当它再次启动的时候，重新载入 Session 对象。

PersistentManagerBase 类作为一个管理器组件将 Session 对象存放到二级存储器中。它有两个直接子类：PersistentManager 和 DistributedManager 类(DistributedManager)类只在 Tomcat4 中有。管理器接口和它的实现类的 UML 结构图如图 9.2 所示

第十章：安全

综述

有些 web 应用程序的内容是有限制的，只允许有权限的用户在提供正确的用户名和密码的情况下才允许访问。Servlet 通过配置部署文件 web.xml 来对安全性提供技术支持。本章的主要内容是容器对于安全性限制的支持。

一个 servlet 通过一个叫 authenticator 的阀门（valve）来支持安全性限制。当容器启动的时候，authenticator 被添加到容器的流水线上。如果你忘了流水线是如何工作的，需要重新复习下第六章的内容。

authenticator 阀门会在包装器阀门之前被调用。authenticator 用于对用户进行验证，如果用户熟人了正确的用户名和密码，authenticator 阀门调用下一个用于处理请求 servlet 的阀门。如果验证失败，authenticator 不唤醒下一个阀门直接返回。由于验证失败，用户并不能看到请求的 servlet。

在用户验证的时候 authenticator 阀门调用的是上下文域（realm）内的 authenticate 方法，将用户名和密码传递给它。该容器域可以访问合法的用户名密码。

本章首先介绍跟安全性相关的类（realms、principal、roles），然后通过一个应用程序演示了如何在你的 servlets 上使用安全管理。

注意 这里假设你已经熟悉 servlet 编程安全性的相关概念，包括：主要 principals, 角色 roles,域 realms, 登陆配置 login configuration 等。如果对这些概念还不清楚可以阅读《*Java for the Web with Senlets, JSP, and EJB*》或者其它相关书籍

（域）Realm

域是用于进行用户验证的一个组件，它可以告诉你一个用户名密码对是否是合法的。一个域跟一个上下文容器相联系，一个容器可以只有一个域。可以使用容器的 setRealm 方法来建立它们之间的联系。

一个域是如何验证一个用户的合法性的？一个域拥有所有的合法用户的密码或者是可以访问它们。至于它们存放在哪里则取决于域的实现。在 Tomcat 的默认实现里，合法用户被存储在 tomcat-users.xml 文件里。但是可以使用域的其它实现来访问其它的源，如关系数据库。

在 Catalina 中，一个域用接口 org.apache.catalina.Realm 表示。该接口最重要的方法是四个 authenticate 方法：

```
public Principal authenticate(String username, String credentials);
public Principal authenticate(String username, byte[] credentials);
public Principal authenticate(String username, String digest,
  String nonce, String nc, String cnonce, String qop, String realm,
  String md5a2);
public Principal authenticate(X509Certificate certs[]);
```

第一个方法是最常用的方法，Realm 接口还有一个 getRole 方法，签名如下：

```
public boolean hasRole(Principal principal, String role);
```

另外，域还有 getContainer 和 setContainer 方法用于建立域与容器的联系。

一个域的基本上实现是抽象类 org.apache.catalina.realm.RealmBase。org.apache.catalina.realm 包中海提供了其它一些类继承了 RealmBase 如：JDBCRealm, JNDIRealm, MemoryRealm,和 UserDatabaseRealm。默认情况下使用的域是 MemoryRealm。

注意在 Catalina 中，验证器阀门使用相关域的 authenticate 来验证一个用户

GenericPrincipal

一个 principal 使用 java.security.Principal 接口来表示，Tomcat 中该接口的实现为 org.apache.catalina.realm.GenericPrincipal 接口。一个 GenericPrincipal 必须跟一个域相关联，这个是通过构造函数实现的：

```
  public GenericPrincipal(Realm realm, String name, String password) {
    this(realm, name, password, null);
  }
  public GenericPrincipal(Realm realm, String name, String password,
    List roles) {
    super();
    this.realm = realm;
   this.name = name;
    this.password = password;
    if (roles != null) {
      this.roles = new String[roles.size()];
      this.roles = (String[]) roles.toArray(this.roles);
      if (this.roles.length > 0)
        Arrays.sort(this.roles);
    }
  }
```

GenericPrincipal 必须拥有一个用户名和一个密码，此外还可选择性的传递一列角色。可以使用 hasRole 方法来检查一个 principal 是否有一个特定的角色，传递的参数为角色的字符串表示形式。这里是 Tomcat4 中的 hasRole 方法：

```
  public boolean hasRole(String role) {
    if (role == null)
      return (false);
    return (Arrays.binarySearch(roles, role) >= 0);
  }
```

Tomcat5 支持 servlet2.4 所以必须支持用*来匹配任何角色。

```
public boolean hasRole(String role) {
  if ("*".equals(role)) // Special 2.4 role meaning everyone
```

```
    return true;
  if (role == null)
    return (false);
  return (Arrays.binarySearch(roles, role) >= 0);
}
```

LoginConfig 类

一个 login configuration 包括一个域名，用 org.apache.catalina.deploy.LoginConfig 类表示。LoginConfig 的实例封装了域名和验证要用的方法。可以使用 LoginConfig 实例的 getRealmName 方法来获得域名，可以使用 getAuthName 方法来验证用户。一个验证（authentication）的名字必须是下面的之一：BASIC, DIGEST, FORM, o 或者 CLIENT-CERT。如果用到的是基于表单（form）的验证，该 LoginConfig 对象还包括登录或者错误页面像对应的 URL。

Tomcat 一个部署启动的时候，先读取 web.xml。如果 web.xml 包括一个 login-confgi 元素，Tomcat 创建一 LoginConfig 对象并相应的设置它的属性。验证阀门调用 LoginConfig 的 getRealmName 方法并将域名发送给浏览器显示登录表单。如果 getRealmName 名字返回值为 null，则发送给浏览器服务器的名字和端口名。图 10.1 是 IE6 的验证会话。

Figure 10.1: The basic authentication dialog

Authenticator 类

org.apache.catalina.Authenticator 接口用来表示一个验证器。该方接口并没有方法，只是一个组件的标志器，这样就能使用 instanceof 来检查一个组件是否为验证器。

Catalina 提供了 Authenticator 接口的基本实现：
org.apache.catalina.authenticator.AuthenticatorBase 类。除了实现 Authenticator 接口外，AuthenticatorBase 还继承了
org.apache.catalina.valves.ValveBase 类。这就是说 AuthenticatorBase 也是一个阀门。可以在 org.apache.catalina.authenticator 包中找到该接口的几个类：BasicAuthenticator 用于基本验证，FormAuthenticator 用于基于表单的验证，DigestAuthentication 用于摘要（digest）验证，SSLAuthenticator 用于 SSL 验证。NonLoginAuthenticator 用于 Tomcat 没有指定验证元素的时候。NonLoginAuthenticator 类表示只是检查安全限制的验证器，但是不进行用户验证。

org.apache.catalina.authenticator 包中类的 UML 结构图如图 10.2 所示：

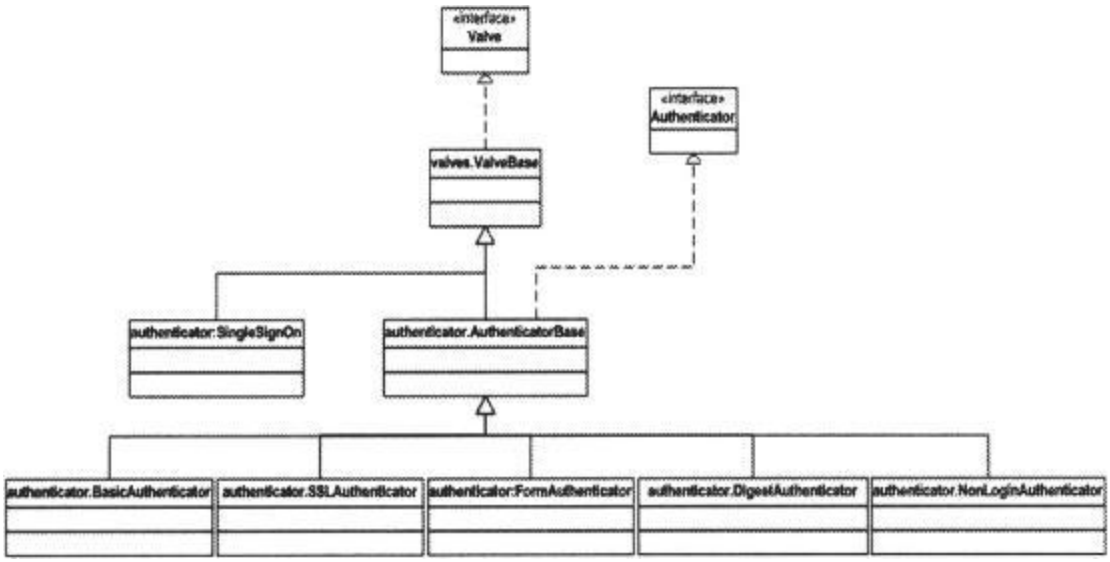


Figure 10.2: Authenticator-related classes

一个验证器的主要工作是验证用户。因此，AuthenticatorBase 类的 invoke 方法调用了抽象方法 authenticate，该方法的具体实现由子类完成。在 BasicAuthenticator 中，它 authenticate 使用基本验证器来验证用户。

安装 Authenticator 阀门

在部署文件中，只能出现一个 login-config 元素，login-config 元素包括了 auth-method 元素用于定义验证方法。这也就是说一个上下文容器只能有一个 LoginConfig 对象来使用一个 authentication 的实现类。

AuthenticatorBase 的子类在上下文中被用作验证阀门，这依赖于部署文件中 auth-method 元素的值。表 10.1 为 auth-method 元素的值，可以用于确定验证器。

Table 10.1: The authenticator implementation class

Value of the auth-method element	Authenticator class
BASIC	BasicAuthenticator
FORM	FormAuthenticator
DIGEST	DigestAuthenticator
CLIENT-CERT	SSLAuthenticator

如果没有使用 auth-method 值，则认为 LoginConfig 对象的 auth-method 属性值为

第十一章：StandardWrapper

在第五章中已经说过，一共有四种容器：engine（引擎）,host（主机）,context（上下文）和 wrapper（包装器）。在前面的章节里也介绍了如何建立自己的 context 和 wrapper。一个上下文一般包括一个或者多个包装器，每一个包装器表示一个 servlet。本章将会看到 Catalina 中 Wrapper 接口的标准实现。首先介绍了一个 HTTP 请求会唤醒的一系列方法，接下来介绍了 javax.servlet.SingleThreadModel 接口。最后介绍了 StandardWrapper 和 StandardWrapperValve 类。本章的应用程序说明了如何用 StandardWrapper 实例来表示 servlet。

方法调用序列 Sequence of Methods Invocation

对于每一个连接，连接器都会调用关联容器的 invoke 方法。接下来容器调用它的所有子容器的 invoke 方法。例如，如果一个连接器跟一个 StadardContext 实例相关联，那么连接器会调用 StandardContext 实例的 invoke 方法，该方法会调用所有它的子容器的 invoke 方法。图 11.1 说明了一个连接器收到一个 HTTP 请求的时候会做的一系列事情。

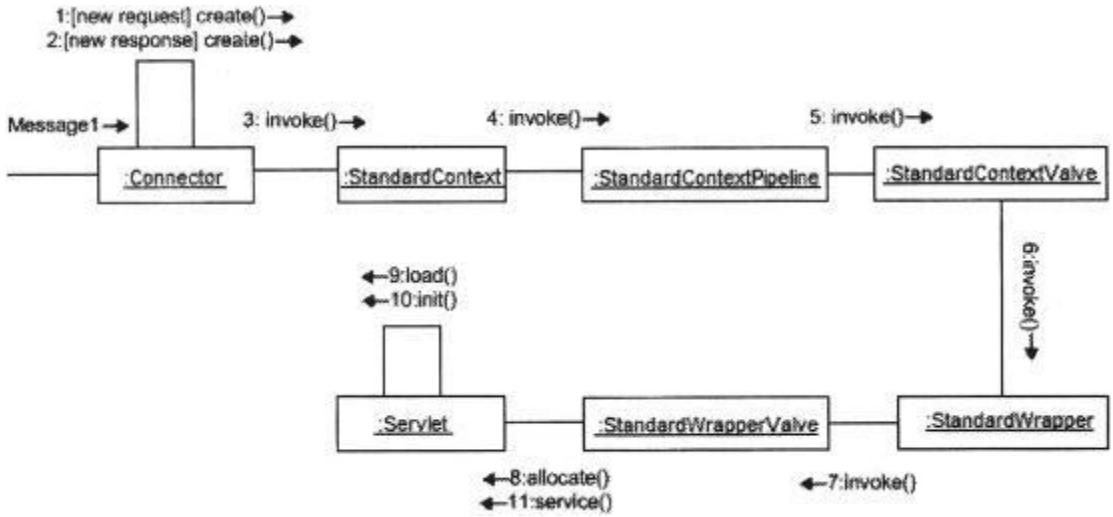


Figure 11.1: The collaboration diagram of methods invocation

- The StandardContextValve's invoke method obtains the appropriate wrapper to serve the request and calls the wrapper's invoke method.
- 连接器创建请求和响应对象
- 连接器调用 StandardContext 的 invoke 方法
- StandardContext 的 invoke 方法必须调用该上下文容器的流水线的 invoke 方法，所以 StandardContext 的流水线会调用 StandardContextValve 的 invoke 方法

- StandardContextValve 的 invoke 方法得到合适的包装器来对请求进行服务并调用包装器的 invoke 方法
- StandardWrapper 是包装器的标准实现，StandardWrapper 对象的 invoke 方法调用流水线的 invoke 方法。
- StandardWrapper 流水线的基本阀门时 StandardWrapperValve。因此 StandardWrapperValve 的 invoke 方法会被调用。StandardWrapperValve 的 invoke 方法会调用包装器的 allocate 方法获得一个 servlet 的实例。
- 当一个 servlet 需要被加载的时候，方法 allocate 调用方法 load 来加载一个 servlet
- 方法 load 会调用 servlet 的 init 方法

注意 StandardContext 类的构造函数设置 StandardContextValve 实例为它的基本阀门

```
publicStandardContext() {

super();
pipeline.setBasic(new StandardContextValve());
namingResources.setContainer(this);
}
```

注意 StandardWrapper 类的构造函数将 StandardWrapperValve 作为它的基本阀门

```
public StandardWrapper() {
super();
pipeline.setBasic(new StandardWrapperValve());
}
```

本章关注的是一个 servlet 被调用的时候发生的细节。因此我们需要自习看 StandardWrapper 和 StandarWrapperValve 类。在学习它们之前，我们需要首先关注下 javax.servlet.SingleThreadModel。理解该接口对于理解一个包装器是如何工作的是非常重要的。

SingleThreadModel

一个 servlet 可以实现 javax.servlet.SingleThreadModel 接口，实现此接口的一个 servlet 通俗称为 SingleThreadModel（STM）的程序组件。根据 Servlet 规范，实现此接口的目的是保证 servlet 一次只能有一个请求。Servlet 2.4 规范的第 SRV.14.2.24 节（Servlet 2.3 的有 SingleThreadModel 接口上的类似说明）

- 如果一个 Servlet 实现此接口，将保证不会有两个线程同是使用 servlet 的 service 方法。 servlet 容器可以保证同步进入一个 servlet 的一个实例，或维持的

Servlet 实例池和处理每个新请求。该接口并不能避免同步而产生的问题，如访问静态类变量或该 servlet 以外的类或变量。

很多程序员并没有仔细阅读它，只是认为实现了 SIngleThreadModel 就能保证它们的 servlet 是线程安全的。单显然并非如此，重新阅读上面的引文。

一个 servlet 实现了 SIngelThreadModel 之后确实能保证它的 service 方法不会被两个线程同时使用。为了提高 servlet 容器的性能，可以创建 STM servlet 的多个实例。

False Sense of Multi-Thread Safety	该 SingleThreadModel 接口在 Servlet 2.4 中已经废弃，因为它使 Servlet 程序员产生虚假的安全感，认为它是线程安全的。然而，无论 Servlet 2.3 和 Servlet 2.4 的容器仍然必须支持此接口

注意 可以在下面网址看到 SingleThreadModel 的相关讨论 http://w4.metronet.com/~wjm/tomcat/ToFeb11/msg02655.html.

StandardWrapper

一个 StandardWrapper 对象的主要职责是：加载它表示的 servlet 并分配它的一个实例。该 StandardWrapper 不会调用 servlet 的 service 方法。这个任务留给 StandardWrapperValve 对象，在 StandardWrapper 实例的基本阀门管道。StandardWrapperValve 对象通过调用 StandardWrapper 的 allocate 方法获得 Servlet 实例。在获得 Servlet 实例之后的 StandardWrapperValve 调用 servlet 的 service 方法

在 servlet 第一次被请求的时候，StandardWrapper 加载 servlet 类。它是动态的加载 servlet，所以需要知道 servlet 类的完全限定名称。通过 StandardWrapper 类的 setServletClass 方法将 servlet 的类名传递给 StandardWrapper。另外，使用 setName 方法也可以传递 servlet 名。

考虑到 StandardWrapper 负责在 StandardWrapperValve 请求的时候分配一个 servlet 实例，它必须考虑一个 servlet 是否实现了 SingleThreadModel 接口。

如果一个 servlet 没有实现 SingleThreadModel 接口，StandardWrapper 加载该 servlet 一次，对于以后的请求返回相同的实例即可。StandardWrapper 假设 servlet 的 service 方法是现场安全的，所以并没有创建 servlet 的多个实例。如果需要的话，由程序员自己解决资源同步问题。

对于一个 STM servlet，情况就有所不同了。StandardWrapper 必须保证不能同时有两个线程提交 STM servlet 的 service 方法。如果 StandardWrapper 维持一个 STM servlet 的实例，下面是它如何调用 servlet 的 service 方法：

```
Servlet instance = <get an instance of the servlet>;
if ((servlet implementing SingleThreadModel>)  {
  synchronized (instance) {
    instance.service(request, response);
```

```
    }
  }
  else {
      instance.service(request, response);
  }
```

但是，为了性能起见，StandardWrapper 维护了一个 STM servlet 实例池。

一个包装器还负责准备一个 javax.servlet.ServletConfig 的实例，这可以在 servlet 内部完成，接下来两小节讨论如何分配和加载 servlet。

Allocating the Servlet

在本节开始的时候介绍到StandardWrapperValve 的 invoke 方法调用了包装器的 allocate 方法来获得一个请求 servlet 的实例。因此 StandardWrapper 类必须实现该接口。该方法的签名如下：

```
public javax.servlet.Servlet allocate() throws ServletException;
```

注意 allocate 方法返回的是请求 servlet 的一个实例。

由于要支持 STM servlet，这使得该方法更复杂了一点。实际上，该方法有两部分组成，一部分负责非 STM servlet 的工作，另一部分负责 STM servlet。第一部分的结构如下：

```
if (!singleThreadModel) {
  // returns a non-STM servlet instance
}
```

布尔变量 singleThreadModel 负责标志一个 servlet 是否是 STM servlet。它的初始值是 false，loadServlet 方法会检测加载的 servlet 是否是 STM 的，如果是则将它的值该为 true。loadServlet 方法在下面会介绍到。

该方法的第二部分处理 singleThreadModel 为 true 的情况，第二部分的框架如下：

```
synchronized (instancepool) {
  // returns an instance of the servlet from the pool
}
```

现在来看一下第一部分和第二部分。

对于非 STM servlet，StandardWrapper 定义一个 java.servlet.Servlet 类型的实例

```
private Servlet instance = null;
```

方法 allocate 检查该实例是否为 null，如果是调用 loadServlet 方法来加载 servlet。然后增加 contAllocated 整型并返回该实例。

```
  if (!singleThreadModel) {
    // Load and initialize our instance if necessary
```

```
    if (instance == null) {
      synchronized (this) {
        if (instance == null) {
          try {
            instance = loadServlet();
          }
          catch (ServletException e) {
            throw e;
          }
          catch (Throwable e) {
            throw new ServletException
              (sm.getString("standardWrapper.allocate"), e);
          }
        }
      }
      if (!singleThreadModel) {
        if (debug >= 2)
          log("  Returninq non-STM instance");
        countAllocated++;
        return (instance);
      }
    }
```

如果 StandardWrapper 表示的是一个 STM servlet，方法 allocate 尝试返回池中的一个实例，变量 intancePool 是一个 java.util.Stack 类型的 STM servlet 实例池。

```
private Stack instancePool = null;
```

该变量在 loadServlet 方法内初始化，该部分在接下来的小节进行讨论。

方法 allocate 负责分配 STMservlet 实例，前提是实例的数目不超过最大数目，该数目由 maxInstances 整型定义，默认值是 20.

```
private int maxInstances = 20;
```

StandardWrapper 提供了 nInstances 整型变量来定义当前 STM 实例的个数。

```
private int nInstances = 0;
```

这里是 allocate 方法的第二部分

```
    synchronized (instancePool) {
      while (countAllocated >= nInstances) {
        // Allocate a new instance if possible, or else wait
        if (nInstances < maxInstances) {
          try {
            instancePool.push(loadServlet());
            nInstances++;
```

```
        }
        catch (ServletException e) {
          throw e;
        }
        catch (Throwable e) {
          throw new ServletException
           (sm.getString("StandardWrapper.allocate"), e);
        }
      }
      else {
        try {
          instancePool.wait();
        }
        catch (InterruptedException e) {
          ;
        }
      }
    }
    if (debug >= 2)
      log("  Returning allocated STM instance");
    countAllocated++;
    return (Servlet) instancePool.pop();

  }
```

上面的代码使用一个 while 循环等待直到 nInstances 的数目少于或等于 countAllocated（应该是多余或等于把！！）。在循环里，allocate 方法检查 nInstance 的值，如果低于 maxInstances 的值，调用 loadServlet 方法并将该实例添加到池中，增加 nInstances 的值。如果 nInstances 的值等于或大于 maxInstances 的值，它调用实例池堆栈的 wait 方法，知道一个实例被返回。

Loading the Servlet

StandardWrapper 实现了 Wrapper 接口的 load 方法,load 方法调用 loadServlet 方法来加载一个 servlet 类，并调用该 servlet 的 init 方法，传递一个 javax.servlet.ServletConfig 实例。这里是 loadServlet 是如何工作的。

方法 loadServlet 首先检查 StandardWrapper 是否表示一个 STM servlet。如果不是并且该实例不是 null（即以前已经加载过），直接返回该实例：

```
  // Nothing to do if we already have an instance or an instance pool
  if (!singleThreadModel && (instance != null))
    return instance;
```

如果该实例是 null 或者是一个 STM servlet，继续该方法的其它部分：.

首先获得 System.out 和 System.err 输出，接下来以就可以使用 javax.servlet.ServletContext 的 log 方法来记录信息

```
PrintStream out = System.out;
SystemLogHandler.startCapture();
```

然后，定义了一个 javax.servlet.Servlet 类型的变量，它表示 loadServlet 方法加载 servlet 后返回的实例。

```
Servlet servlet = null;
```

方法 loadServlet 负责加载 servlet 类,类名应该被分配给 servletClass 变量，该方法将该值分配给一个 String 类型变量 actualClass。

```
String actualClass = servletclass;
```

但是，由于 Catalina 也是一个 JSP 容器，在请求的是 JSP 页面的时候，loadServlet 必须也能工作，如果是 JSP 页面，则得到相应的 Servlet 类。

```
if ((actualClass == null) && (jspFile != null)) {
  Wrapper jspWrapper = (Wrapper)
  ((Context) getParent()).findChild(Constants.JSP_SERVLET_NAME);
  if (jspWrapper != null)
    actualClass = jspWrapper.getServletClass();
}
```

如果 JSP 页面的 Servlet 名字找不到，就是用 servletclass 变量的值。但是，如果该变量的值没有使用 StandardWrapper 类中的 setServletClass 方法设置，会产生异常，剩余部分不会被执行。

```
// Complain if no servlet class has been specified
if (actualClass == null) {
  unavailable(null);
  throw new ServletException
    (sm.getString("StandardWrapper.notClass", getName()));
}
```

现在，servlet 的名字已经获得了，接下来是 loadServlet 方法获得加载器。如果找不到加载器，则产生异常并停止执行。

```
// Acquire an instance of the class loader to be used
Loader loader = getLoader();
if (loader == null) {
  unavailable(null);
  throw new ServletException
  (sm.getString("StandardWrapper.missingLoader", getName()));
}
```

如果找到加载器，loadServlet 方法调用它的 getClassLoader 方法获得一个 ClassLoader。

```
ClassLoader classLoader = loader.getClassLoader();
```

Catalina 提供了特殊 Servlet，从属于 org.apache.catalina 包。这些 Servlet 可以进入 Servlet 容器的内部。如果该 Servlet 是一个特殊 Servlet，isContainerProvidedServlet 方法返回 true 值。classLoader 会获得另一个 ClassLoader 的实例，这样就可以访问 Catalina 的内部了。

```
// Special case class loader for a container provided servlet
if (isContainerProvidedServlet(actualClass)) {
  ClassLoader = this.getClass().getClassLoader();
  log(sm.getString
    ("standardWrapper.containerServlet", getName()));
}
```

有了类加载器和腰加载的 Servlet 名字，就可以使用 loadServlet 方法来加载类了。

```
// Load the specified servlet class from the appropriate class
// loader
Class classClass = null;
try {
  if (ClassLoader != null) {
    System.out.println("Using classLoader.loadClass");
    classClass = classLoader.loadClass(actualClass);
  }
  else {
    System.out.println("Using forName");
    classClass = Class.forName(actualClass);
  }
}
catch (ClassNotFoundException e) {
  unavailable(null);
  throw new ServletException
    (sm.getstring("standardWrapper.missingClass", actualClass), e);
}
if (classClass == null) {
  unavailable(null);
  throw new ServletException
    (sm.getString("standardWrapper.missingClass", actualClass));
}
```

Then, it can instantiate the servlet.

```
// Instantiate and initialize an instance of the servlet class
// itself
try {
  servlet = (Servlet) classClass.newInstance();
}
catch (ClassCastException e) {
  unavailable(null);
```

```
    // Restore the context ClassLoader
    throw new ServletException
      (sm.getString("standardWrapper.notServlet", actualClass), e);
  }
  catch (Throwable e) {
    unavailable(null);
    // Restore the context ClassLoader
   throw new ServletException
      (sm.getstring("standardWrapper.instantiate", actualClass), e);

  }
```

但是，loadServlet 方法在初始化 Servlet 之前，它使用 isServletAllowed 方法来检查该 Servlet 是否可以访问。

```
  // Check if loading the servlet in this web application should be
  // allowed
  if (!isServletAllowed(servlet)) {
    throw new SecurityException
      (sm.getString("standardWrapper.privilegedServlet",
        actualClass));
  }
```

如果通过了安全性检查,接下来检查该 Servlet 是否是一个 ContainerServlet。ContainerServlet 是实现了 org.apache.catalina.ContainerServlet 接口的 Servlet，它可以访问 Catalina 的内部函数。如果该 Servlet 是 ContainerServlet，loadServlet 方法调用 ContainerServlet 的 setWrapper 方法，传递该 StandardWrapper 实例。

```
  // Special handling for ContainerServlet instances
  if ((servlet instanceof ContainerServlet) &&
    isContainerProvidedServlet(actualClass)) {
    ((ContainerServlet) servlet).setWrapper(this);
  }
```

接下来 loadServlet 方法触发 BEFORE_INIT_EVENT 事件，并调用发送者的 init 方法。

```
  try {
    instanceSupport.fireInstanceEvent(
      InstanceEvent.BEFORE_INIT_EVENT, servlet);
    servlet.init(facade);
```

注意该方法传递一个 façade 变量,改变量是一个 javax.servlet.ServletConfig 对象。怎样创建一个 ServletConfig 对象会在本章的 Creating ServletConfig 小节看到。

如果 loadOnStartup 变量的值大于零并且 Servlet 是一个 JSP 页面，调用该 Servlet 的 service 方法。

```
// Invoke jspInit on JSP pages
if ((loadOnStartup > 0) && (jspFile != null)) {
  // Invoking jspInit
  HttpRequestBase req = new HttpRequestBase();
  HttpResponseBase res = new HttpResponseBase();
  req.setServletPath(jspFile};
  req.setQueryString("jsp_precompile=true");
  servlet.service(req, res);
}
```

接下来，loadServlet 方法触发 AFTER_INIT_EVENT 事件

```
instanceSupport.firelnstanceEvent (InstanceEvent.AFTER_INIT_EVENT,
  servlet);
```

如果该 StandardWrapper 对象表示的是一个 STM Servlet，将该实例添加到实例池中，因此，如果实例池如果为 null，首先需要创建它。

```
  // Register our newly initialized instance
  singleThreadModel = servlet instanceof SingleThreadModel;
  if (singleThreadModel) {
    if (instancePool == null)
      instancePool = new Stack();
  }
  fireContainerEvent("load", this);
}
```

在 finally 块中，loadservlet 方法会停止捕获 System.out 和 System.err，并将加载过程中信息使用该容器的 log 方法记录到日志系统中。

```
finally {
  String log = SystemLogHandler.stopCapture();
  if (log != null && log.length() > 0) {
    if (getServletContext() != null) {
      getServletContext().log(log);
    }
    else {
      out.println(log);
    }
  }
}
```

最后，loadServlet 方法返回 Servlet 实例。

```
return servlet;
```

ServletConfig 对象

StandardWrapper 的 loadServlet 方法在加载了 loaded 方法之后调用的发送者的 init 方法 。init 方法传递一个 javax.servlet.ServletConfig 实例，你可能想知道一个 StandardWrapper 对象如何获得 ServletConfig 对象。

只要看 StandardWrapper 类即可，该类实现 javax.servlet.ServletConfig 接口和 Wrapper 接口。

ServletConfig 接口有以下四个方法 getServletContext, getServletName, getInitParameter, 和 getInitParameterNames。接下来看 StandardWrapper 对这四个类的实现。

注意 StandardWrapper 并不将自己传递给 Servlet 的 init 方法，它将自己包装到一个 StandardWrapperFacade 实例中来因此它的 public 方法。可以在 StandardWrapperFacade 一节看到这些内容。

getServletContext

该方法的签名如下：

```
public ServletContext getServletContext()
```

一个 StandardWrapper 实例必须是一个 StandardContext 容器的子容器。也就是说，StandardWrapper 的父容器时 StandardContext。可以使用 StandardContext 对象的 getServletContext 来获得 ServletContext 对象。这里是 StandardWrapper 中方法 getServletContext 的实现

```
public ServletContext getServletContext() {
  if (parent == null)
    return (null);
  else if (!(parent instanceof Context))
    return (null);
  else
    return (((Context) parent).getServletContext());
}
```

注意 现在你知道不能单独部署一个包装器来表示一个 Servlet，包装器必须从属于一个上下文容器，这样才能使用 ServletConfig 对象使用 getServletContext 方法获得一个 ServletContext 实例。

getServletName

该方法返回 Servlet 的名字，该方法签名如下：

```
public java.lang.String getServletName()
```

这里是 getServletName 方法在 StandardWrapper 类中实现：

```
public String getServletName() {
  return (getName());
}
```

它简单的调用 StandardWrapper 的父类 ContainerBase 类的 getName 方法，该方法如下在 ContainerBase 中如下实现：

```
public String getName() {
  return (name);
}
```

可以使用 setName 方法来设置 name 的值。回忆是如何调用 StandardWrapper 实例的 setName 方法来传递 Servlet 的 name 的。

getInitParameter

该方法返回指定参数的值，该方法的签名如下：

```
public java.lang.String getInitParameter(java.lang.String name)
```

在 StandardWrapper 中，初始化参数被存放在一个名为 parameters 的 HashMap 中

```
private HashMap parameters = new HashMap();
```

可以调用 StandardWrapper 类的 addInitParameter 方法来填充 parameters。传递参数的名字和值。

```
public void addInitParameter(String name, String value) {
  synchronized (parameters) {
    parameters.put(name, value);
  }
  fireContainerEvent("addInitParameter", name);
}
```

下面是 StandardWrapper 对 getInitParameter 的实现：

```
public String getInitParameter(String name) {
  return (findInitParameter(name));
}
```

方法 findInitParameter 的参数为参数名，并调用 parameters HashMap 的 get 方法。下面是 findInitParameter 的实现：

```
public String findInitParameter(String name) {

  synchronized (parameters) {
    return ((String) parameters.get(name));
  }
}
```

getInitParameterNames

该方法返回所有初始化参数名字的枚举（Enumeration），它的签名如下：

```
public java.util.Enumeration getInitParameterNames()
```

下面是 StandardWrapper 类中 getInitParameterNames 的实现：

```
public Enumeration getInitParameterNames() {
  synchronized (parameters) {
    return (new Enumerator(parameters.keyset()));
  }
}
```

Enumerator 实现了 java.util.Enumeration 接口，是 org.apache.catalina.util 包的一部分。

Parent and Children

一个包装器表示一个独立 Servlet 的容器。这样，包装器就不能再有子容器，因此不可以调用它的 addChild 方法，如果调用了会得到一个 java.langIllegalStateException。这里是 StandardWrapper 对 addChild 方法的实现：

```
public void addChild(Container child) {
  throw new IllegalStateException
    (sm.getString("StandardWrapper.notChild"));
}
```

一个包装器的父容器只能是一个上下文容器。如果传递的参数不是一个上下文容器，它的 setParent 方法会抛出 java.lang.IllegalArgumentException。

```
public void setParent(Container container) {
  if ((container != null) && !(container instanceof Context))
    throw new IllegalArgumentException
      (sm.getString("standardWrapper.notContext"));
  super.setParent(container);
}
```

StandardWrapperFacade

StandardWrapper 调用它价值的 Servlet 的 init 方法。该方法需要一个 javax.servlet.ServletConfig 的参数，而 StandardWrapper 类自己就实现了 ServletConfig 接口。所以，理论上 StandardWrapper 可以将它自己作为参数传递给 init 方法。但是 StandardWrapper 需要对 Servlet 隐藏他的大多数 public 方法。为了实现这一点，StandardWraper 将它自己包装的一个 StandardWrapperFacade 实例中。图 11.2 表示了 StandardWrapper 和

StandardWrapperFacade 的实现，它们都实现了 java.servlet.ServletConfig 接口。

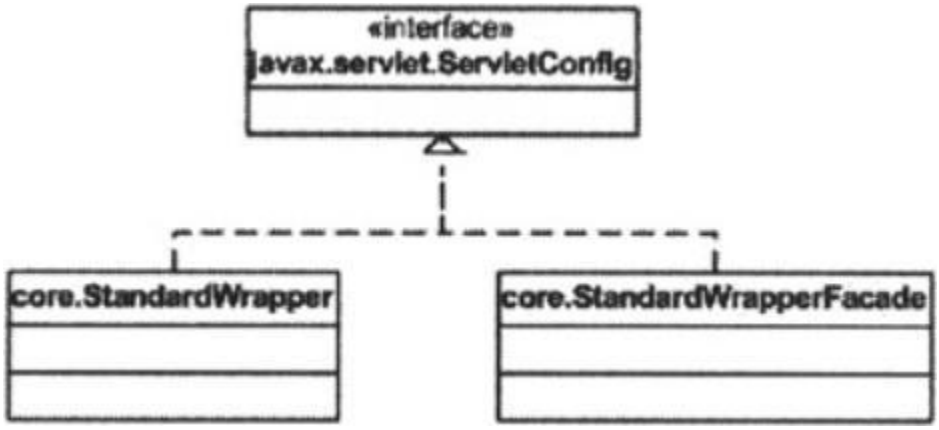


Figure 11.2: The relationship between StandardWrapper and StandardWrapperFacade

下面几行是 StandardWrapperFacade 的构造函数，它获得一个 StandardWrapper 类型的参数。

```
private StandardWrapperFacade facade = new StandardWrapperFacade(this);
```

StandardWrapperFacade 类有一个 ServletConfig 类型的类级变量 config

```
private ServletConfig config = null;
```

当一个 StandardWrapperFacade 对象创建的时候，构造函数将该 StandardWrapper 负值给 config 变量

```
public StandardWrapperFacade(StandardWrapper config) {
  super();
  this.config = (ServletConfig) config;
}
```

因此，当 StandardWrapper 对象调用 Servlet 实例的 init 方法的时候，它传递的是一个 StandardWrapperFacade 对象。在 Servlet 内部调用 ServletConfig 的 getServletName, getInitParameter, 和 getInitParameterNames 方法只需要调用它们在 StandardWrapper 的实现就行。

```
public String getServletName() {
  return config.getServletName();
}
public String getInitParameter(String name) {
  return config.getInitParameter(name);
}
public Enumeration getInitParameterNames() {
  return config.getInitParameterNames();
}
```

调用 getServletContext 方法稍微复杂点：

```
public ServletContext getServletContext() {
```

```
  ServletContext theContext = config.getServletContext();
  if ((theContext != null) && (theContext instanceof
ApplicationContext))
    theContext = ((ApplicationContext) theContext).getFacade();
  return (theContext);
}
```

Gauge 方法调用 StandardWrapper 类的 getServletContext 方法，但是它返回一个 ServletContext 的外观对象，而不是 ServletContext 对象自己。

StandardWrapperValve

StandardWrapperValve 是 StandardWrapper 实例上的基本阀门，该阀门做两件事情：

- 提交 Servlet 的所有相关过滤器
- 调用发送者的 service 方法

要实现这些内容，下面是 StandardWrapperValve 在他的 invoke 方法要实现的：

- 调用 StandardWrapper 的 allocate 的方法来获得一个 servlet 实例
- 调用它的 private createFilterChain 方法获得过滤链
- 调用过滤器链的 doFilter 方法。包括调用 servlet 的 service 方法
- 释放过滤器链
- 调用包装器的 deallocate 方法
- 如果 Servlet 无法使用了，调用包装器的 unload 方法

接下来是 invoke 方法：

```
  // Allocate a servlet instance to process this request
  try {
    if (!unavailable) {
      servlet = wrapper.allocate();
    }
  }
  ...
  // Acknowlege the request
  try {
    response.sendAcknowledgement();
  }
  ...
  // Create the filter chain for this request
  ApplicationFilterChain filterChain = createFilterChain(request,
    servlet);
  // Call the filter chain for this request
```

```
// This also calls the servlet's servicet() method
try {
  String jspFile = wrapper.getJspFile();
  if (jspFile != null)
    sreq.setAttribute(Globals.JSP_FILE_ATTR, jspFile);
  else
    sreq.removeAttribute(Globals.JSP_FILE_ATTR);
  if ((servlet != null) && (filterChain != null)) {
    filterChain.doFilter(sreq, sres);
  }
  sreq.removeAttribute(Globals.JSP_FILE_ATTR);
}
...
// Release the filter chain (if any) for this request
try {
  if (filterChain != null)
    filterChain.release();
}
...
// Deallocate the allocated servlet instance
try {
  if (servlet != null) {
    wrapper.deallocate(servlet);
  }
}
...
// If this servlet has been marked permanently unavailable,
// unload it and release this instance
try {
  if ((servlet != null) && (wrapper.getAvailable() ==
    Long.MAX_VALUE)) {
    wrapper.unload();
  }

}
...
```

最重要的方法是 createFilterChain 方法并调用过滤器链的 doFilter 方法。方法 createFilterChain 创建了一个 ApplicationFilterChain 实例，并将所有的过滤器添加到上面。ApplicationFilterChain 类将在下面的小节中介绍。要完全的理解这个类，还需要理解 FilterDef 和 ApplicationFilterConfig 类。这些内容将在下面介绍

FilterDef

org.apache.catalina.deploy.FilterDef 表示一个过滤器定义，就像是在部署文件中定义一个过滤器元素那样。Listing11.1 展示了该类：

Listing 11.1: The FilterDef class

```
package org.apache.catalina.deploy;

import java.util.HashMap;
import java.util.Map;

public final class FilterDef {
  /**
   * The description of this filter.
   */
  private String description = null;
  public String getDescription() {
    return (this.description);
  }
  public void setDescription(String description) {
    this.description = description;
  }

  /**
   * The display name of this filter.
   */
  private String displayName = null;
  public String getDisplayName() {
    return (this.displayName);
  }
  public void setDisplayName(String displayName) {
    this.displayName = displayName;
  }

  /**

   * The fully qualified name of the Java class that implements this
   * filter.
   */
  private String filterClass = null;
  public String getFilterClass() {
    return (this.filterClass);
  }
  public void setFilterclass(String filterClass) {
    this.filterClass = filterClass;
  }
```

```
/**
 * The name of this filter, which must be unique among the filters
 * defined for a particular web application.
 */
private String filterName = null;
public String getFilterName() {
  return (this.filterName);
}
public void setFilterName(String filterName) {
  this.filterName = filterName;
}

/**
 * The large icon associated with this filter.
 */
private String largeIcon = null;
public String getLargeIcon() {
  return (this.largeIcon);
}
public void setLargeIcon(String largeIcon) {
  this.largeIcon = largeIcon;
}

/**
 * The set of initialization parameters for this filter, keyed by
 * parameter name.
 */
private Map parameters = new HashMap();
public Map getParameterMap() {
  return (this.parameters);
}

/**
 * The small icon associated with this filter.
 */
private String smallIcon = null;
public String getSmallIcon() {
  return (this.smallIcon);
}
public void setSmallIcon(String smallIcon) {
  this.smallIcon = smallIcon;
}

public void addInitParameter(String name, String value) {
```

```
    parameters.put(name, value);

  }
  /**
   * Render a String representation of this object.
   */
  public String toString() {
    StringBuffer sb = new StringBuffer("FilterDef[");
    sb.append("filterName=");
    sb.append(this.filterName);
    sb.append(", filterClass=");
    sb.append(this.filterClass);
    sb.append("]");
    return (sb.toString());
  }
}
```

FilterDef 类中的每一个属性都代表一个可以在过滤器中出现的子元素。该类包括一个 Map 类型的变量表示一个包含所有初始参数的 Map。方法 addInitParameer 添加一个 name/value 对到该 Map。

ApplicationFilterConfig

org.apache.catalina.core.ApplicationFilterConfig 实现了 javax.servlet.FilterConfig 接口。ApplicationFilterConfig 负责管理 web 应用程序启动的时候创建的过滤器实例。

传递一个 org.apache.catalina.Context 对象和 ApplicationFilterConfig 对象给 ApplicationFilterConfig 的构造来创建一个 ApplicationFilterConfig 实例：

```
public ApplicationFilterConfig(Context context, FilterDef filterDef)
  throws ClassCastException, ClassNotFoundException,
    IllegalAccessException, InstantiationException, ServletException
```

Context 对象表示一个一个 web 应用而 FilterDef 表示一个过滤器定义。ApplicationFilterConfig 的 getFilter 方法可以返回一个 javax.servlet.Filter 方法，该方法加载过滤器类并初始化它。

```
Filter getFilter() throws ClassCastException, ClassNotFoundException,
  IllegalAccessException, InstantiationException, ServletException {

  // Return the existing filter instance, if any
  if (this.filter != null)
    return (this.filter);

  // Identify the class loader we will be using
```

```
  String filterClass = filterDef.getFilterClass();
  ClassLoader classLoader = null;
  if (filterClass.startsWith("org.apache.catalina."))
    classLoader = this.getClass().getClassLoader();
  else
    classLoader = context.getLoader().getClassLoader();

  ClassLoader oldCtxClassLoader =
    Thread.currentthread().getContextClassLoader();

  // Instantiate a new instance of this filter and return it
  Class clazz = classLoader.loadClass(filterClass);
  this.filter = (Filter) clazz.newInstance();
  filter.init(this);
  return (this.filter);
}
```

ApplicationFilterChain

org.apache.catalina.core.ApplicationFilterChain 类是实现了 javax.servlet.FilterChain 接口。StandardWrapperValve 类中的 invoke 方法创建一个该类的实例并且调用它的 doFilter 方法。ApplicationFilterChain 类的 doFilter 的调用该链中第一个过滤器的 doFilter 方法。Filter 接口中 doFilter 方法的签名如下：

```
public void doFilter(ServletRaquest request, ServletResponse response,
  FilterChain chain) throws java.io.IOException, ServletException
```

ApplicationFilterChain 的 doFilter 方法，并将它自己作为第三个参数传递给它。

在他的 doFilter 方法中，一个过滤器可以调用另一个过滤器链的 doFilter 来唤醒另一个过来出去。这里是一个过滤器的 doFilter 实现的例子

```
public void doFilter(ServletRequest request, ServletResponse response,
  FilterChain chain) throws IOException, ServletException {
  // do something here
  ...
  chain.doFilter(request, response);
}
```

如你看到的，在 doFilter 方法最好一行，它调用过滤链的 doFilter 方法。如果该过滤器是过滤链的最后一个过滤器，它叫调用请求的 Servlet 的 service 方法。如果过滤器没有调用 chain.doFilter， 下一个过滤器就不会被调用。

The Application

本应用程序有两个类组成 x11.pyrmont.core.SimpleContextConfig 和 ex11.pyrmont.startup.Bootstrap。SimpleContextConfig 类是前一章程序里的副本，而 Bootstrap 类如 Listing11.2 所示：

Listing 11.2: The Bootstrap class

```
package ex11.pyrmont.startup;
//use StandardWrapper
import ex11.pyrmont.core.SimpleContextConfig;
import org.apache.catalina.Connector;
import org.apache.catalina.Context;
import org.apache.catalina.Lifecycle;
import org.apache.catalina.LifecycleListener;
import org.apache.catalina.Loader;
import org.apache.catalina.Wrapper;
import org.apache.catalina.connector.http.HttpConnector;
import org.apache.catalina.core.StandardContext;
import org.apache.catalina.core.StandardWrapper;
import org.apache.catalina.loader.WebappLoader;

public final class Bootstrap {
  public static void main(String[] args) {
    System.setProperty("catalina.base",
      System.getProperty("user.dir"));
    Connector connector = new HttpConnector();
    Wrapper wrapper1 = new StandardWrapper();
    wrapper1.setName("Primitive");
    wrapper1.setServletClass("PrimitiveServlet");
    Wrapper wrapper2 = new StandardWrapper();
    wrapper2.setName("Modern");
    wrapper2.setServletClass("ModernServlet");

    Context context = new StandardContext();
    // StandardContext's start method adds a default mapper
    context.setPath("/myApp");
    context.setDocBase("myApp");
    LifecycleListener listener = new SimpleContextConfig();
    ((Lifecycle) context).addLifecycleListener(listener);
    context.addChild(wrapper1);
    context.addChild(wrapper2);
    // for simplicity, we don't add a valve, but you can add
    // valves to context or wrapper just as you did in Chapter 6
    Loader loader = new WebappLoader();
    context.setLoader(loader);
    // context.addServletMapping(pattern, name);
    context.addServletMapping("/Primitive", "Primitive");
```

```
    context.addServletMapping("/Modern", "Modern");
    // add ContextConfig. This listener is important because it
    // configures StandardContext (sets configured to true), otherwise
    // StandardContext won't start

    connector.setContainer(context);
    try {
      connector.initialize();
      ((Lifecycle) connector).start();
      ((Lifecycle) context).start();
      // make the application wait until we press a key.
      System.in.read();
      ((Lifecycle) context).stop();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Bootstrap 创建一个 StandardContext 实例并调用它的 myApp。Bootstrap 类在该 StandardContext 容器中，添加了两个 StandardWrapper 实例：Primitive 和 Modern

Running the Applications

要在 Windows 中运行该应用程序，可以在工作目录下面使用如下命令：

java -classpath ./lib/servlet.jar;./lib/commons-collections.jar;./ex11.pyrmont.startup.Bootstrap

在 Linux 中，需要使用分号来分开两个库

java -classpath ./lib/servlet.jar:./lib/commons-collections.jar:./ex11.pyrmont.startup.Bootstrap

可以在浏览器中输入如下 URL 来调用 PrimitiveServlet

http://localhost:8080/Primitive

使用如下 URL 来调用 ModernServlet

http://localhost:8080/Modern

总结

在本章中,你学习到了 Wrapper 接口在 Catalina 中的标准实现 StandardWrapper 类。并且接受了过滤器以及跟过滤器相关的类。在本章的最后接受了一个使用 StandardWrapper 类的应用程序。

第十二章：StandardContext

综述

在前面的章节中已经看到，一个上下文容器（Context）代表一个 web 应用，每一个上下文包括多个包装器（Wrapper），每个包装器代表一个 Servlet。但是，上下文还需要其它的一些组件如加载器和管理器。本章要介绍 Catalina 中 Context 接口的标准实现，org.apache.catalina.core.StandardContext 类。

我们首先介绍 StandardContext 对象的初始化和配置，然后讨论跟其相关的类 StandardContextMapper（Tomcat 4）和 ContextConfig 类。接下来看，当容器接受到 HTTP 请求的时候方法调用顺序。然后，在讨论该类几点重要特性，最后一节讨论 Tomcat5 中的 backgroundProcess 方法。

注意 本章没有相关配套的应用程序，StandardContext 类已经在 11 章中用过了

StandardContext 配置

创建一个 StandardContext 实例之后，必须调用它的 start 方法，这样它就能为受到的 HTTP 请求服务了。一个 StandardContext 对象可能启动失败，这时候属性 available 被设置为 false，属性 available 表示了 StandardContext 对象的可用性。

如果 start 方法启动成功，StandardContext 对象需要配置它的属性。在一个 Tomcat 部署中，StandardContext 的配置过程做了以下事情：准备读取和解析%CATALINA_HOME%/conf 目录下面的 web.xml，部署所有应用程序，确保 StandardContext 实例可以处理应用级别的 web.xml。另外，配置需要添加一个验证器阀门和证书阀门（authenticator valve and a certificate valve）

注意更多 StandardContext 配置的细节将在 15 章中讨论。

StandardContext 的属性之一是它属性 configured,用来表示该 StandardContext 是否已经配置了。StandardContext 使用一个事件监听器来作为它的配置器。当 StandardContext 实例的 start 方法被调用的时候，首先触发一个生命周期事件。该事件唤醒一个监听器来配置该 StandardContext 实例。配置成功后，该监听器将 configured 属性设置为 true。否则，StandardContext 对象拒绝启动，这样就不能对 HTTP 请求进行服务了。

在第 11 章中已经看到一个生命周期监听器被添加到 StandardContext 实例上，它的类型是 ch11.pyrmont.core.SimpleContextConfig，它仅仅将 StandardContext 实例的 configured 属性设置为 true,这样就认为配置完成了。在一个 Tomcat 部署中，配置 StandardContext 的生命周期监听器类型为 org.apache.catalina.startup.ContextConfig,具体内容将在第 15 章中进行介绍。

现在你应该已经明白了配置过程对于 StandardContext 的重要性,接下来看更多 StandardContext 类的细节，首先是它的构造函数。

StandardContext 构造函数

下面是 StandardContext 类的构造函数：

```
public StandardContext() {
  super();
  pipeline.setBasic(new StandardContextValve());
  namingResources.setContainer(this);
}
```

在构造函数中，最重要的事情是在 StandardContext 的流水线上添加了一个类型为 org.apache.catalina.core.StandardContextValve 的基本阀门，该阀门用于裁判美国连接器获得 HTTP 请求。

启动 StandardContext

Start 方法初始化 StandardContext 对象并让生命周期监听器配置该 StandardContext 实例。如果配置成功，生命周期监听器会将 configured 属性设置为 true。最后 start 方法，将 available 属性设置为 true 或者 false。如果是 true 的话表示该 StandardContext 属性配置完毕并且所有相关子容器和组件已经成功启动，这样就能对 HTTP 请求进行服务了，如果是 false 则表示出现了错误。

StandardContext 类将 configured 的值初始化为 false，如果生命周期监听器的配置过程成功，则将该值设置为 true。在 start 方法的最后，它检查 StandardContext 对象的 configured 属性，如果该值为 true，则启动该 StandardContext 成，否则调用 stop 方法停止所有已经启动的组件。

Tomcat 4 中 StandardContext 类中的 start 方法如 Listing12.1:

Listing 12.1: The start method of the StandardContext class in Tomcat 4

```
public synchronized void start() throws LifecycleException {
  if (started)
    throw new LifecycleException
      (sm.getString("containerBase.alreadyStarted", logName()));
  if (debug >= 1)
    log("Starting");

  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(BEFORE_START_EVENT, null);

  if (debug >= 1)
    log("Processing start(), current available=" + getAvailable());
  setAvailable(false);
  setConfigured(false);
  boolean ok = true;
```

```
// Add missing components as necessary
if (getResources() == null) {   // (1) Required by Loader
  if (debug >= 1)
    log("Configuring default Resources");
  try {
    if ((docBase != null) && (docBase.endsWith(".war")))
      setResources(new WARDirContext());
    else
      setResources(new FileDirContext());
  }
  catch (IllegalArgumentException e) {
    log("Error initializing resources: " + e.getMessage());
    ok = false;
  }
}

if (ok && (resources instanceof ProxyDirContext)) {
  DirContext dirContext =
    ((ProxyDirContext) resources).getDirContext();
  if ((dirContext != null)
    && (dirContext instanceof BaseDirContext)) {
    ((BaseDirContext) dirContext).setDocBase(getBasePath());
    ((BaseDirContext) dirContext).allocate();
  }
}
if (getLoader() == null) {      // (2) Required by Manager
  if (getPrivileged()) {
    if (debug >= 1)
      log("Configuring privileged default Loader");
    setLoader(new WebappLoader(this.getClass().getClassLoader()));
  }
  else {
    if (debug >= 1)
      log("Configuring non-privileged default Loader");
    setLoader(new WebappLoader(getParentClassLoader()));
  }
}
if (getManager() == null) {     // (3) After prerequisites
  if (debug >= 1)
    log("Configuring default Manager");
  setManager(new StandardManager());
}
```

```
// Initialize character set mapper
getCharsetMapper();

// Post work directory
postWorkDirectory();

// Reading the "catalina.useNaming" environment variable
String useNamingProperty = System.getProperty("catalina.useNaming");
if ((useNamingProperty != null)
  && (useNamingProperty.equals("false"))) {
  useNaming = false;
}

if (ok && isUseNaming()) {
  if (namingContextListener == null) {
    namingContextListener = new NamingContextListener();
    namingContextListener.setDebug(getDebug());
    namingContextListener.setName(getNamingContextName());
              addLifecycleListener(namingContextListener);
  }
}

// Binding thread
ClassLoader oldCCL = bindThread();

// Standard container startup
if (debug >= 1)
  log("Processing standard container startup");

if (ok) {
  try {
    addDefaultMapper(this.mapperClass);
    started = true;

    // Start our subordinate components, if any
    if ((loader != null) && (loader instanceof Lifecycle))
      ((Lifecycle) loader).start();
    if ((logger != null) && (logger instanceof Lifecycle))
      ((Lifecycle) logger).start();

    // Unbinding thread
    unbindThread(oldCCL);

    // Binding thread
```

```
    oldCCL = bindThread();

    if ((cluster != null) && (cluster instanceof Lifecycle))
      ((Lifecycle) cluster).start();
    if ((realm != null) && (realm instanceof Lifecycle))
      ((Lifecycle) realm).start();
    if ((resources != null) && (resources instanceof Lifecycle))
      ((Lifecycle) resources).start();

    // Start our Mappers, if any
    Mapper mappers[] = findMappers();
    for (int i = 0; i < mappers.length; i++) {
      if (mappers[i] instanceof Lifecycle)
      ((Lifecycle) mappers[i]).start();
    }

    // Start our child containers, if any
    Container children[] = findChildren();
    for (int i = 0; i < children.length; i++) {
      if (children[i] instanceof Lifecycle)
      ((Lifecycle) children[i]).start();
    }

    // Start the Valves in our pipeline (including the basic),
    // if any
    if (pipeline instanceof Lifecycle)
      ((Lifecycle) pipeline).start();

    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(START_EVENT, null);

    if ((manager != null) && (manager instanceof Lifecycle))
      ((Lifecycle) manager).start();
  }
  finally {
    // Unbinding thread
    unbindThread(oldCCL);
  }
}

if (!getConfigured())
  ok = false;

// We put the resources into the servlet context
```

```
if (ok)
  getServletContext().setAttribute
    (Globals.RESOURCES_ATTR, getResources());

// Binding thread
      oldCCL = bindThread();

// Create context attributes that will be required
if (ok) {
  if (debug >= 1)
    log("Posting standard context attributes");
  postWelcomeFiles();
}

// Configure and call application event listeners and filters
if (ok) {
  if (!listenerStart())
    ok = false;
}
if (ok) {
  if (!filterStart())
    ok = false;
}

// Load and initialize all "load on startup" servlets
if (ok)
  loadOnStartup(findChildren());

// Unbinding thread
unbindThread(oldCCL);

// Set available status depending upon startup success
if (ok) {
  if (debug >= 1)
    log("Starting completed");
  setAvailable(true);
}
else {
  log(sm.getString("standardContext.startFailed"));
  try {
    stop();
  }
  catch (Throwable t) {
    log(sm.getString("standardContext.startCleanup"), t);
```

```
    }
    setAvailable(false);
  }

  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(AFTER_START_EVENT, null);
}
```

注意 在 Tomcat5 中，start 方法跟 Tomcat4 的相似，只是包括了 JMX 相关的代码，JMX 的内容将在第 20 章讨论。读完该章后就可以查看 Tomcat5 中的 start 方法了。

如在 Listing12.1 中看到的，下面是该方法做的事情：

- 触发 BEFORE_START 事件
- 设置 availability 属性为 false
- 设置 configured 属性为 false
- 设置源（resources）
- 设置加载器
- 设置管理器
- 初始化属性 map
- 启动跟该上下文相关的组件
- 启动子容器（包装器）
- 启动流水线
- 启动管理器
- 触发 START 事件。监听器（ContextConfig）会进行一系列配置操作，配置成功后，将 StandardContext 实例的 configured 属性设置为 true。
- 检查 configured 属性的值，如果为 true：调用 postWelcomPages 方法，加载子包装器，并将 available 属性设置为 true。如果 configured 属性为 false 调用 stop 方法
- 触发 AFTER_START 事件

Invoke 方法

在 Tomcat4 中，StandardContext's 方法由相关联的连接器调用，如果该上下文是一个主机（host）的子容器，有该主机的 invoke 方法调用。StandardContext 的 invoke 方法首先检查是否正在重加载该应用程序，是的话，等待知道加载完毕。然后调用它的父类 ContainerBase 的 invoke 方法。Listing12.2 展示了 StandardContext 的 invoke 方法。

Listing 12.2: The invoke method of the StandardContext Class

```
public void invoke(Request request, Response response)
  throws IOException, ServletException {

  // Wait if we are reloading
  while (getPaused()) {
    try {
      Thread.sleep(1000);
    }
    catch (InterruptedException e) {
      ;
    }
  }
  // Normal request processing
  if (swallowOutput) {
    try {
      SystemLogHandler.startCapture();
      super.invoke(request, response);
    }
    finally {
      String log = SystemLogHandler.stopCapture();
      if (log != null && log.length() > 0) {
        log(log);
      }
    }
  }
  else {
    super.invoke(request, response);
  }
}
```

方法 getPaused 获得属性 paused 的值，当应用程序正在加载的时候该属性为 ture。应用程序加载将在下一节讨论。

在 Tomcat5 中，StandardContext 并没有提供 invoke 方法的实现，因此它会执行 ContainerBase 的 invoke 方法。检查应用程序加载的任务在 StandardContextValve 类的 invoke 方法中完成。

StandardContextMapper

对于每一个请求，invoke 方法都会调用 StandarContext 流水线基本阀门的 invoke 方法。StandarContext 的基本阀门用 org.apache.catalina.core.StandardContextValve 类表示。StandardContextValve 的 invoke 方法需要得到一个处理请求的包装器。

在 Tomcat4 中，StandardContextValve 实例查找包含它的 StandardContext。StandardContextValve 使用上下文容器的 map 来查找合适的包装器。一旦它得

到一个合适的包装器，调用该包装器的 invoke 方法。在介绍 StandardContextValve 的做法的时候，本节先介绍该 map 组件。

StandardContext 的父类 ContainerBase 定义了 addDefaultMapper 方法来添加一个默认的映射（map）如下，

```
protected void addDefaultMapper(String mapperClass) {
  // Do we need a default Mapper?
  if (mapperClass == null)
    return;
  if (mappers.size() >= 1)
    return;

  // Instantiate and add a default Mapper
  try {
    Class clazz = Class.forName(mapperClass);
    Mapper mapper = (Mapper) clazz.newInstance();
    mapper.setProtocol("http");
    addMapper(mapper);
  }
  catch (Exception e) {
    log(sm.getString("containerBase.addDefaultMapper", mapperClass),
      e);
  }
}
```

StandardContext 在 start 方法中调用它的 addDefaultMapper 方法，传递一个 mapperClass 变量。

```
public synchronized void start() throws LifecycleException {
  ...
  if (ok) {
    try {
      addDefaultMapper(this.mapperClass);
  ...
}
```

StandardContext 的 mapperClass 变量定义如下：

```
private String mapperClass =
"org.apache.catalina.core.StandardContextMapper";
```

你必须使用映射器的 setContainer 方法来讲映射跟容器相关联。org.apache.catalina.Mapper 接口的标准实现是 org.apache.catalina.core.StandardContextMapper。StandardContextMapper 只能跟上下文相关联，使用的方法是 setCOntainer

```
public void setContainer(Container container) {
  if (!(container instanceof StandardContext))
```

```
    throw new IllegalArgumentException
      (sm.getstring("httpContextMapper.container"));
  context = (StandardContext) container;
}
```

在一个映射器（mapper）中最重要的方法是 map 方法，它获得一个 HTTP 请求返回一个子容器，该方法的签名如下：

```
public Container map(Request request, boolean update)
```

在 StandardContextMapper 中 map 方法返回一个包装器来处理请求。如果找不到合适的包装器，方法返回 null。

回到本节开始讨论的内容，StandardContextValve 调用上下文容器的 map 方法来处理 HTTP 请求，传递的参数是 org.apache.catalina.Request 对象。map 方法（ContainerBase 类中）返回一个对应相应协议的映射器，然后调用该映射器的 map 方法。

```
// Select the Mapper we will use
Mapper mapper = findMapper(request.getRequest().getProtocol());
if (mapper == null)
  return (null);
// Use this Mapper to perform this mapping
return (mapper.map(request, update));
```

The map method in StandardContextMapper first identifies the context-relative URI to be mapped:

StandardContextMapper 的 map 方法首先识别上下文的相关 URL 映射

```
// Identify the context-relative URI to be mapped
String contextPath =
  ((HttpServletRequest) request.getRequest()).getContextPath();
String requestURI = ((HttpRequest) request).getDecodedRequestURI();

String relativeURI = requestURI.substring(contextPath.length());
```

然后通过匹配方法尝试获得一个包装器

```
// Apply the standard request URI mapping rules from the specification
Wrapper wrapper = null;
String servletPath = relativeURI;
String pathInfo = null;
String name = null;

// Rule 1 -- Exact Match
if (wrapper == null) {
  if (debug >= 2)
    context.log("  Trying exact match");
  if (!(relativeURI.equals("/")))
    name = context.findServletMapping(relativeURI);
```

```
  if (name != null)
    wrapper = (Wrapper) context.findChild(name);
  if (wrapper != null) {
    servletPath = relativeURI;
    pathInfo = null;
  }
}

// Rule 2 -- Prefix Match
if (wrapper == null) {
  if (debug >= 2)
    context.log("  Trying prefix match");
  servletPath = relativeURI;
  while (true) {
    name = context.findServletMapping(servletPath + "/*");
    if (name != null)
      wrapper = (Wrapper) context.findChild(name);
    if (wrapper != null) {
      pathInfo = relativeURI.substring(servletPath.length());
      if (pathInfo.length() == 0)
        pathInfo = null;
      break;
    }
    int slash = servletPath.lastIndexOf('/');
    if (slash < 0)
      break;
    servletPath = servletPath.substring(0, slash);
  }
}

// Rule 3 -- Extension Match
if (wrapper == null) {
  if (debug >= 2)
    context.log("  Trying extension match");
  int slash = relativeURI.lastIndexOf('/');
  if (slash >= 0) {
    String last = relativeURI.substring(slash);
    int period = last.lastIndexOf('.');
    if (period >= 0) {
      String pattern = "*" + last.substring(period);

      name = context.findServletMapping(pattern);
      if (name != null)
        wrapper = (Wrapper) context.findChild(name);
```

```
      if (wrapper != null) {
        servletPath = relativeURI;
        pathInfo = null;
      }
    }
  }
}

// Rule 4 -- Default Match
if (wrapper == null) {
  if (debug >= 2)
    context.log("  Trying default match");
  name = context.findServletMapping("/");
  if (name != null)
    wrapper = (Wrapper) context.findChild(name);
  if (wrapper != null) {
    servletPath = relativeURI;
    pathInfo = null;
  }
}
```

你可能会问，上下文容器时如何获得 Servlet 映射的，回头看第十一章中的 Bootstrap 类，它添加了两个 Servlet 映射到 StandarContext。

```
context.addServletMapping("/Primitive", "Primitive");
context.addServletMapping("/Modern", "Modern");
```

它还将包装器作为子容器添加到上下文容器中。

```
context.addChild(wrapper1);
context.addChild(wrapper2);
```

Tomcat 5 删除了 Mapper 接口及其相关类。实际上 StandardContextValve 的 invoke 方法就能得到请求的合适包装器。

```
Wrapper wrapper = request.getWrapper();
```

这表示映射信息已经包装在请求对象中了。

重加载支持

StandardContext 定义了 reloadable 属性来标识是否支持应用程序的重加载。当允许重加载的时候，当 web.xml 或者 WEB-INF/classes 目录下的文件被改变的时候会重加载。

StandardContext 用它的加载器来加载应用程序。在 Tomcat4 中，StandardContext 中 Loader 接口的标准实现 WebappLoader 类，有一个单独线程来检查 WEB-INF 目录下面所有类和 JAR 文件的时间戳。你需要做的是启动该线程，将 WebappLoader 关联到 StandardContext，使用 setContainer 方法即可。下面是 Tomcat4 中 WebappLoader 的实现：

```
public void setContainer(Container container) {
  // Deregister from the old Container (if any)
  if ((this.container != null) && (this.container instanceof Context))
    ((Context) this.container).removePropertyChangeListener(this);
  // Process this property change
  Container oldContainer = this.container;
  this.container = container;
  support.firePropertyChange("container", oldContainer,
    this.container);
  // Register with the new Container (if any)
  if ((this.container!=null) && (this.container instanceof Context)) {
    setReloadable( ((Context) this.container).getReloadable() );
    ((Context) this.container).addPropertyChangeListener(this);
  }
}
```

注意最后一个 if 语句块中，如果容器是一个上下文容器，调用 setReloadable 方法，也就是说 WebappLoader 的 reloadable 属性跟 StandardContext 的 reloadable 属性相同。

下面是 WebappLoader 对 setReload 方法的实现：

```
public void setReloadable(boolean reloadable) {
  // Process this property change
  boolean oldReloadable = this.reloadable;
  this.reloadable = reloadable;
  support.firePropertyChange("reloadable",
    new Boolean(oldReloadable), new Boolean(this.reloadable));
  // Start or stop our background thread if required
  if (!started)
    return;
  if (!oldReloadable && this.reloadable)
    threadStart();
  else if (oldReloadable && !this.reloadable)
    threadStop();
}
```

如果将 reloadable 属性设置为 true，调用 threadStart 方法。如果从 true 到 false，则调用 threadStop 方法。threadStart 方法启动一个线程持续的检查 WEB-INF 目录下面的类文件和 JAR 文件的时间戳。threadStop 方法用于停止该线程。

在 Tomcat5 中，类的时间戳是由 backgroundProcess 方法调用的，下一节将会结束该方法。

backgroundProcess 方法

一个上下文容器需要其它组件如加载器和管理器的支持。这些组件通常需要一个单独的线程来处理后台过程（background processing）。例如，加载器通过一个线程检查类文件和 JAR 文件的时间戳来支持自动重载。管理器需要一个线程来检查它管理的Session对象过期时间。在Tomcat4中,这些组件都有自己的线程。

为了节省资源，Tomcat 使用了一种不同的方式来处理。所有的后台过程都分享同一个线程。如果一个组件或者是容器需要定期的来执行操作，它需要做的是将这些代码写入到 backgroundProcess 方法即可。

共享线程有 ContainerBase 对象创建，ContainerBase 在他的 start 方法中调用 threadStard 方法。

```
protected void threadStart() {
  if (thread != null)
    return;
  if (backgroundProcessorDelay <= 0)
    return;
  threadDone = false;
  String threadName = "ContainerBackgroundProcessor[" + toString() +
    "]";
  thread = new Thread(new ContainerBackgroundProcessor(), threadName);
  thread.setDaemon(true);
  thread.start();
}
```

方法 threadStart 传递一个 ContainerBackgroundProcessor 对象创建一个新线程。ContainerBackgroundProcessor 实现了 java.lang.Runnable 接口，该类如 Listing12.3 所示

Listing 12.3: The ContainerBackgroundProcessor class

```
protected class ContainerBackgroundProcessor implements Runnable {
  public void run() {
    while (!threadDone) {
      try {
        Thread.sleep(backgroundProcessorDelay * 1000L);
      }
      catch (InterruptedException e) {
        ;
      }
      if (!threadDone) {
        Container parent = (Container) getMappingObject();
        ClassLoader cl =
          Thread.currentThread().getContextClassLoader();
        if (parent.getLoader() != null) {
          cl = parent.getLoader().getClassLoader();
        }
        processChildren(parent, cl);
      }
```

```
    }
  }

  protected void processChildren(Container container, ClassLoader cl) {
    try {
      if (container.getLoader() != null) {
        Thread.currentThread().setContextClassLoader
          (container.getLoader().getClassLoader());
      }
      container.backgroundProcess();
    }
    catch (Throwable t) {
      log.error("Exception invoking periodic operation: ", t);
    }
    finally {
      Thread.currentThread().setContextClassLoader(cl);
    }
    Container[] children = container.findChildren();
    for (int i = 0; i < children.length; i++) {
      if (children[i].getBackgroundProcessorDelay() <= 0) {
        processChildren(children[i], cl);
      }
    }
  }
}
```

ContainerBackgroundProcessor 是 ContainerBase 的内部类，在他的 run 方法里,有一个 while 循环定期的调用它的 processChildren 方法。processChildren 调用 backgroundProcess 来处理它的每个孩子的 processChildren 方法。要实现 backgroundProcess 方法，以 ContainerBase 的子类可以有一个线程来周期性的执行任务，例如检查时间戳或者 Session 对象的终结时间。Listing12.4 展示了 Tomcat5 中 StandardContext 的 backgroundProcess 方法的实现。

Listing 12.4: The backgroudProccss method of the StandardContext class

```
public void backgroundProcess() {
  if (!started)
    return;
  count = (count + 1) % managerChecksFrequency;
  if ((getManager() != null) && (count == 0)) {
    try {
      getManager().backgroundProcess();
    }
    catch ( Exception x ) {
      log.warn("Unable to perform background process on manager", x);
    }
```

```
  }
  if (getloader() != null) {
    if (reloadable && (getLoader().modified())) {
      try {
        Thread.currentThread().setContextClassloader
          (StandardContext.class.getClassLoader());
        reload();
      }
      finally {
        if (getLoader() != null) {
          Thread.currentThread().setContextClassLoader
            (getLoader().getClassLoader());
        }
      }
    }
    if (getLoader() instanceof WebappLoader) {
      ((WebappLoader) getLoader()).closeJARs(false);
    }
  }
}
```

需要明白 StandardContext 怎样让它的相关管理器和加载器来执行任务。

总结

在本章中,介绍了 StandardContext 极其相关类。另外还看到了 StandardContext 对象是如何配置的以及如何处理 HTTP 请求，最后一张讨论了 Tomcat5 中 backgroundProcess 方法的实现。

第 13 章：主机(host)和引擎(engine)

综述

本章要讨论的两个主题是主机（host）和引擎（Engine）。如果需要在一个 Tomcat 部署中部署多个上下文，需要使用一个主机。理论上，当只有一个上下文容器的时候不需要主机，正如 org.apache.catalina.Context 接口中描述的那样。

- 上下文容器的父容器是主机，但是可能有一些其它实现，没有必要的时候也可以忽略。

但是实践中，一个 Tomcat 部署往往需要一个主机。至于为什么，你会在本章后面的 Why You Cannot Live without a Host 一节中看到。

引擎表示整个 Catalina 的 Servlet 引擎。如果使用的话，它位于容器等级的最高层。可以添加到引擎上的容器包括 org.apache.catalina.Host 或者 org.apache.catalina.Context。在一个 Tomcat 部署中，默认的容器是引擎。在该部署中，引擎只有一个主机，默认主机。

本章讨论了跟 Host 和 Engine 接口接口相关的类。首先介绍了 Host 相关的 StandardHost、StandardHostMapper（Tomcat4）以及 StandardHostValve 类。接下来是，用一个示例来说明了主机作为顶层容器的情况。引擎是本章讨论的第二个主题。接下来是本章的第二个应用程序，说明了如何将引擎作为顶层容器使用。

Host 接口

主机是用 org.apache.catalina.Host 接口表示的。本接口继承了 Container 接口，如 Listing13.1 所示：

Listing 13.1: The Host interface

```
package org.apache.catalina;

public interface Host extends Container {
  public static final String ADD_ALIAS_EVENT = "addAlias";
  public static final String REMOVE_ALIAS_EVENT = "removeAlias";
  /**
   * Return the application root for this Host.
   * This can be an absolute
   * pathname, a relative pathname, or a URL.
   */
  public String getAppBase();

  /**
   * Set the application root for this Host.  This can be an absolute
   * pathname, a relative pathname, or a URL.
   *
```

```
     * @param appBase The new application root
     */
    public void setAppBase(String appBase);

    /**
     * Return the value of the auto deploy flag.
     * If true, it indicates that
     * this host's child webapps should be discovred and automatically
     * deployed.
     */
    public boolean getAutoDeploy();

    /**
     * Set the auto deploy flag value for this host.
     *
     * @param autoDeploy The new auto deploy flag
     */
    public void setAutoDeploy(boolean autoDeploy);

    /**
     * Set the DefaultContext
     * for new web applications.
     *
     * @param defaultContext The new DefaultContext
     */
    public void addDefaultContext(DefaultContext defaultContext);

    /**
     * Retrieve the DefaultContext for new web applications.
     */

    public DefaultContext getDefaultContext();

    /**
     * Return the canonical, fully qualified, name of the virtual host
     * this Container represents.
     */
    public String getName();

    /**
     * Set the canonical, fully qualified, name of the virtual host
     * this Container represents.
     *
     * @param name Virtual host name
```

```
     *
     * @exception IllegalArgumentException if name is null
     */
    public void setName(String name);

    /**
     * Import the DefaultContext config into a web application context.
     *
     * @param context web application context to import default context
     */
    public void importDefaultContext(Context context);

    /**
     * Add an alias name that should be mapped to this same Host.
     *
     * @param alias The alias to be added
     */
    public void addAlias(String alias);

    /**
     * Return the set of alias names for this Host.  If none are defined,
     * a zero length array is returned.
     */
    public String[] findAliases();

    /**
     * Return the Context that would be used to process the specified
     * host-relative request URI, if any; otherwise return
     * <code>null</code>.
     *
     * @param uri Request URI to be mapped
     */
    public Context map(String uri);

    /**
     * Remove the specified alias name from the aliases for this Host.
     * @param alias Alias name to be removed
     */
    public void removeAlias(String alias);
}
```

最终要的方法是用合适的上下文来处理请求的 map 方法，该方法的实现可以在 StandardHost 类中找到。

StandardHost 类

org.apache.catalina.core.StandardHost 类是对 Host 接口的标准实现。该继承了 org.apache.catalina.core.ContainerBase 类并实现了 Host 接口和 Deployer 接口。Deployer 接口将在第 17 章讨论。

跟 StandardContext 和 StandardWrapper 类相似，StandardHost 类的构造函数在它的流水线中添加一个基本阀门。

```
public StandardHost() {
  super();
  pipeline.setBasic(new StandardHostValve());
}
```

该阀门的类型为 org.apache.catalina.core.StandardHostValve。

当它的 start 方法被调用的时候，StandardHost 上面添加两个阀门：ErrorReportValve 和 ErrorDispatcherValve。它们都在 org.apache.catalina.valves 包中。Tomcat4 中 StandardHost 的 start 方法如 Listing13.2 所示

Listing 13.2: The start method of StandardHost

```
public synchronized void start() throws LifecycleException {
  // Set error report valve
  if ((errorReportValveClass != null)
    && (!errorReportValveClass.equals(""))) {
    try {
      Valve valve =
        (Valve) Class.forName(errorReportValveClass).newInstance();
      addValve(valve);
    }
    catch (Throwable t) {
      log(sm.getString
        ("StandardHost.invalidErrorReportValveClass",
          errorReportValveClass));
    }
  }
  // Set dispatcher valve
  addValve(new ErrorDispatcherValve());
  super.start();
}
```

注意 在 Tomcat5 中，start 方法相似，不同点在于包括了构建 JXM 对象的代码，JMX 将在第 20 章讨论

errorReportValveClass 的值由 StandardHost 类如下决定：

```
private String errorReportValveClass =
```

```
  "org.apache.catalina.valves.ErrorReportValve";
```

对于每一个请求，都会调用主机的 invoke 方法。由于 StanardHost 类并没有实现 invoke 方法,所以会调用它的父类 ContainerBase 类的 invoke 方法。该 invoke 方法会转而调用 StandardHost 的基本阀门 StandardHostValve 的 invoke 方法。StandardHostValve 阀门的 invoke 方法将在 StandardHostValve 小节讨论。StandardHostValve 的 invoke 方法调用 StandardHost 类的 map 方法获得一个合适的上下文容器来处理请求。StandardHost 的 map 方法如 Listing13.3 所示

Listing 13.3: The map method in the StandardHost class

```
public Context map(String uri) {
  if (debug > 0)
    log("Mapping request URI '" + uri + "'");
  if (uri == null)
    return (null);

  // Match on the longest possible context path prefix
  if (debug > 1)
    log("  Trying the longest context path prefix");
  Context context = null;
  String mapuri = uri;
  while (true) {
    context = (Context) findChild(mapuri);
    if (context != null)
      break;
    int slash = mapuri.lastIndexOf('/');
    if (slash < 0)
      break;
    mapuri = mapuri.substring(0, slash);
  }

  // If no Context matches, select the default Context
  if (context == null) {
    if (debug > 1)
      log("  Trying the default context");
    context = (Context) findChild("");
  }

  // Complain if no Context has been selected

  if (context == null) {
    log(sm.getString("standardHost.mappingError", uri));
    return (null);
  }

  // Return the mapped Context (if any)
```

```
  if (debug > 0)
    log(" Mapped to context '" + context.getPath() + "'");
  return (context);
}
```

注意在 Tomcat4 中，ContainerBase 类也声明了一个 map 方法如下签名：

```
public Container map(Request request, boolean update);
```

在 Tomcat4 中，StandardHostVavle 的 invoke 方法调用 ContainerBase 的 map 方法，它由转而调用 StandardHost 的 map 方法。在 Tomcat5 中，没有映射器组件，适当的上下文由请求对象获得。

StandardHostMapper 类

在 Tomcat4 中，StandardHost 的父类 ContainerBase 使用 addDefaultMapper 方法创建一个默认映射器。默认映射器的类型由 mapperClass 属性指定。这里是 ContainerBase 的 addDefaulstMapper 方法：

```
protected void addDefaultMapper(String mapperClass) {
  // Do we need a default Mapper?
  if (mapperClass == null)
    return;
  if (mappers.size() >= 1)
    return;

  // Instantiate and add a default Mapper
  try {
    Class clazz = Class.forName(mapperClass);
    Mapper mapper = (Mapper) clazz.newInstance();
    mapper.setProtocol("http");
    addMapper(mapper);
  }
  catch (Exception e) {
    log(sm.getString("containerBase.addDefaultMapper", mapperClass),
      e);
  }
}
```

StandardHost 定义 mapperClass 变量如下：

```
private String mapperClass =
  "org.apache.catalina.core.StandardHostMapper";
```

另外，StandardHost 类的 start 方法在它的最后调用 super.start()，这样保证了创建一个默认的映射器。

注意 Tomcat4 中的 standardContext 使用了不同的方法来创建一个默认映射器。它的 start 方法中并没有调用 super.start()。相反 Standardcontext 的

start 方法调用 addDefaultMapper 来传递 mapperClass 变量。

The most important method in StandardHostMapper is, of course, map. Here it is.

StandardHostMapper 中最重要的方法是 map 方法，下面是它的实现：

```
public Container map(Request request, boolean update) {
  // Has this request already been mapped?
  if (update && (request.getContext() != null))
    return (request.getContext());

  // Perform mapping on our request URI
  String uri = ((HttpRequest) request).getDecodedRequestURI();
  Context context = host.map(uri);

  // Update the request (if requested) and return the selected Context
  if (update) {
    request.setContext(context);
    if (context != null)
      ((HttpRequest) request).setContextPath(context.getPath());
    else
      ((HttpRequest) request).setContextPath(null);
  }
  return (context);
}
```

注意，map 方法仅仅是简单的调用了 Host 的 map 方法。

StandardHostValve 类

org.apache.catalina.core.StandardHostValve 类是 StandardHost 的基本阀门类型。当有 HTTP 请求的时候会调用它的 invoke 方法：

Listing 13.4: The invoke method of StandardHostValve

```
public void invoke(Request request, Response response,
  ValveContext valveContext)
  throws IOException, ServletException {

  // Validate the request and response object types

  if (!(request.getRequest() instanceof HttpServletRequest) ||
    !(response.getResponse() instanceof HttpServletResponse)) {
    return;     // NOTE - Not much else we can do generically
  }

  // Select the Context to be used for this Request
  StandardHost host = (StandardHost) getContainer();
```

```
  Context context = (Context) host.map(request, true);
  if (context == null) {
    ((HttpServletResponse) response.getResponse()).sendError
    (HttpServletResponse.SC_INTERNAL_SERVER_ERROR,
    sm.getstring("StandardHost.noContext"));
    return;
  }

  // Bind the context CL to the current thread
  Thread.currentThread().setContextClassLoader
    (context.getLoader().getClassLoader());

  // Update the session last access time for our session (if any)
  HttpServletRequest hreq = (HttpServletRequest) request.getRequest();
  String sessionId = hreq.getRequestedSessionId();
  if (sessionId != null) {
    Manager manager = context.getManager();
    if (manager != null) {
      Session session = manager.findSession(sessionId);
      if ((session != null) && session.isValid())
        session.access();
      }
    }

  // Ask this Context to process this request
  context.invoke(request, response);
}
```

在 Tomcat4 中的 invoke 方法中调用 StandardHost 的 map 方法来获得一个合适的上下文。

```
  // Select the Context to be used for this Request
  StandardHost host = (StandardHost) getContainer();
  Context context = (Context) host.map(request, true);
```

注意 在得到上下对象的时候需要一个往返过程，map 方法介绍两个参数，该方法是在 ContainerBase 中的。然后 ContainerBase 类又在它的子对象中查找合适的映射器并调用它的 map 方法。

Invoke 方法解析来得到一个 Session 对象并调用它的 access 方法，access 方法更新它的最后进入时间，这里是
org.apache.catalina.session.StandardSession 类中的 access 方法。

```
public void access() {
  this.isNew = false;
```

```
  this.lastAccessedTime = this.thisAccessedTime;
  this.thisAccessedTime = System.currentTimeMillis();
}
```

最后，invoke 方法调用上下文容器的 invoke 方法，让上下文来处理请求。

为什么 Host 是必须的

一个 Tomcat 部署必须有一个主机如果该上下文使用 ContextConfig 来配置。原因如下：

ContextConfig 需要应用文件 web.xml 的位置，它在它的 applicationConfig 方法中尝试打开该文件，下面是该方法的片段：

```
synchronized (webDigester) {
  try {
    URL url =
      servletContext.getResource(Constants.ApplicationWebXml);
    InputSource is = new InputSource(url.toExternalForm());
    is.setByteStream(stream);
    ...
    webDigester.parse(is);
    ...
```

Constants.ApplicationWebXml 定义的是 /WEB-INF/web.xml，servletContext 是一个 org.apache.catalina.core.ApplicationContext 类型的对象。

下面是 ApplicationContext 类的 getResource 方法

```
public URL getResource(String path)
  throws MalformedURLException {

  DirContext resources = context.getResources();
  if (resources != null) {
    String fullPath = context.getName() + path;
    // this is the problem. Host must not be null
    String hostName = context.getParent().getName();
```

最后一行清楚的现实上下文的父容器(主机)是必须的，如果使用 ContextConfig 来配置的话。在第 15 章中将会介绍如何解析 web.xml 文件。简单的说，除非你自己写 ContextConfig 类，否则你必须有一个主机。

Application 1 程序 1

本章第一个应用程序说明了如何将一个主机作为顶层容器使用。该程序有两个类组成：ex13.pyrmont.core.SimpleContextConfig 和 ex13.pyrmont.startup.Bootstrap1 类。SimpleContextConfig 类跟第 11 章中的相同，而 Boostrap2 类如 Listing13.5 所示：

Listing 13.5: The Bootstrap1 Class

```
package ex13.pyrmont.startup;

import ex13.pyrmont.core.SimpleContextConfig;
import org.apache.catalina.Connector;
import org.apache.catalina.Context;
import org.apache.catalina.Host;
import org.apache.catalina.Lifecycle;
import org.apache.catalina.LifecycleListener;
import org.apache.catalina.Loader;
import org.apache.catalina.Wrapper;
import org.apache.catalina.connector.http.HttpConnector;
import org.apache.catalina.core.StandardContext;
import org.apache.catalina.core.StandardHost;
import org.apache.catalina.core.StandardWrapper;
import org.apache.catalina.loader.WebappLoader;

public final class Bootstrap1 {
  public static void main(String[] args) {
    System.setProperty("catalina.base",
      System.getProperty("user.dir"));
    Connector connector = new HttpConnector();
    Wrapper wrapper1 = new StandardWrapper();
    wrapper1.setName("Primitive");
    wrapper1.setServletClass("PrimitiveServlet");
    Wrapper wrapper2 = new StandardWrapper();
    wrapper2.setName("Modern");
    wrapper2.setServletClass("ModernServlet");
    Context context = new StandardContext();
    // StandardContext's start method adds a default mapper
    context.setPath("/app1");
    context.setDocBase("app1");
    context.addChild(wrapper1);
    context.addChild(wrapper2);
    LifecycleListener listener = new SimpleContextConfig();
    ((Lifecycle) context).addLifecycleListener(listener);
    Host host = new StandardHost();
    host.addChild(context);
    host.setName("localhost");
    host.setAppBase("webapps");
    Loader loader = new WebappLoader();
    context.setLoader(loader);
    // context.addServletMapping(pattern, name);

    context.addServletMapping("/Primitive", "Primitive");
```

```
    context.addServletMapping("/Modern", "Modern");
    connector.setContainer(host);
    try {
      connector.initialize();
      ((Lifecycle) connector).start();
      ((Lifecycle) host).start();
      // make the application wait until we press a key.
      System.in.read();
      ((Lifecycle) host).stop();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Running the Applications

在 Windows 下面可以在工作目录下输入如下命令运行该程序

```
java -classpath ./lib/servlet.jar;./lib/commons-
collections.jar;./lib/commons—digester.jar;./
ex13.pyrmont.startup.Bootstrap1
```

在 Linux 下面需要使用分号来分隔开两个库

```
java -classpath ./lib/servlet.jar:./lib/commons-
collections.jar:./lib/commons-digester.jar:./
ex13.pyrmont.startup.Bootstrap1
```

使用如下 URL 可以调用 PrimitiveServlet

```
http://localhost:8080/app1/Primitive
```

如下 URL 来调用 ModernServlet。

```
http://localhost:8080/app1/Modern
```

Engine 接口

org.apache.catalina.Engine 接口用来表示一个引擎。引擎表示整个 Catalina 的 Servlet 引擎。当你想要支持多个虚拟主机的时候，需要一个引擎，实际上，Tomcat 部署正是使用了引擎。

Engine 接口如 Listing 13.6 所示：

Listing 13.6: The Engine Interface

```
package org.apache.catalina;
```

```
public interface Engine extends Container {
  /**
   * Return the default hostname for this Engine.
   */
  public String getDefaultHost();
  /**
   * Set the default hostname for this Engine.
   *
   * @param defaultHost The new default host
   */
  public void setDefaultHost(String defaultHost);
  /**
   * Retrieve the JvmRouteId for this engine.
   */
  public String getJvmRoute();
  /**
   * Set the JvmRouteId for this engine.
   *
   * @param jvmRouteId the (new) JVM Route ID. Each Engine within a
   * cluster must have a unique JVM Route ID.
   */
  public void setJvmRoute(String jvmRouteId);
  /**
   * Return the <code>Service</code> with which we are associated (if
   * any).
   */
  public Service getService();
  /**
   * Set the <code>Service</code> with which we are associated (if
   * any).
   *
   * @param service The service that owns this Engine
   */
  public void setService(Service service);
  /**
   * Set the DefaultContext
   * for new web applications.
   *
   * @param defaultContext The new DefaultContext
   */
  public void addDefaultContext(DefaultContext defaultContext);
  /**
   * Retrieve the DefaultContext for new web applications.
   */
```

```
  public DefaultContext getDefaultContext();
  /**
   * Import the DefaultContext config into a web application context.
   *
   * @param context web application context to import default context
   */
  public void importDefaultContext(Context context);
}
```

可以给引擎设置默认主机或者默认上下文。注意引擎也可以跟服务相关联（service），Service 的内容在第 14 章介绍。

StandardEngine 类

类 org.apache.catalina.core.StandardEngine 是 Engine 接口的标准实现，跟 StandardContext 和 StandardHost 相比，StandardEngine 类相对较小。初始化的时候，StandardEngine 类需要添加一个基本阀门，下面是该类构造函数：

```
public StandardEngine() {
  super();
  pipeline.setBasic(new StandardEngineValve());
}
```

在 Container 容器的顶层，StandardEngine 可以有子容器，它的子容器必须是主机（host）。如果你尝试给它添加一个非主机容器，会产生异常。这里是 StandardEngine 类的 addChile 方法。

```
public void addChild(Container child) {
  if (!(child instanceof Host))
    throw new IllegalArgumentException
      (sm.getString("StandardEngine.notHost"));
  super.addChild(child);
}
```

由于位于容器的顶层，所以引擎不能有父容器，当你尝试给引擎设置父容器的时候会产生异常，下面是 StandardEngine 类的 setParent 方法

```
public void setParent(Container container) {
  throw new IllegalArgumentException
   (sm.getString("standardEngine.notParent"));
}
```

StandardEngineValve 类

org.apache.catalina.core.StandardEngineValve 是 StandardEngine 的基本阀门，它的 invoke 方法如 Listing13.7

Listing 13.7: The invoke method of StandardEngineValve

```
public void invoke(Request request, Response response,
```

```
  ValveContext valveContext)
  throws IOException, ServletException {
  // Validate the request and response object types
  if (!(request.getRequest() instanceof HttpServletRequest) ||
    !(response.getResponse() instanceof HttpServletResponse)) {
    return;     // NOTE - Not much else we can do generically
  }
  // Validate that any HTTP/1.1 request included a host header
  HttpServletRequest hrequest = (HttpServletRequest) request;
  if ("HTTP/1.1".equals(hrequest.getProtocol()) &&
    (hrequest.getServerName() == null)) {
    ((HttpServletResponse) response.getResponse()).sendError
    (HttpServletResponse.SC_BAD_REQUEST,
      sm.getString("standardEngine.noHostHeader",
    request.getRequest().getServerName()));
    return;
  }
  // Select the Host to be used for this Request
  StandardEngine engine = (StandardEngine) getContainer();
  Host host = (Host) engine.map(request, true);
  if (host == null) {
    ((HttpServletResponse) response.getResponse()).sendError
      (HttpServletResponse.SC_BAD_REQUEST,
      sm.getString("standardEngine.noHost",
      request.getRequest().getServerName()));
    return;
  }
  // Ask this Host to process this request
  host.invoke(request, response);
}
```

在验证了请求对象和响应对象之后,invoke 方法获得一个 Host 实例来处理请求。它得到主机的方法是调用引擎的 map 方法。一旦获得了一个主机，它的 invoke 方法将会被调用。

Application 2 程序 2

本章的第二个应用程序用于说明容器最顶层的引擎。该应用程序使用了两个类，ex13.pyrmont.core.SimpleContextConfig 和
ex13.pyrmont.startup.Bootstrap2 类。Bootstrap2 类如 Listing13.8 所示

Listing 13.8: The Bootstrap2 class

```
package ex13.pyrmont.startup;

//Use engine
```

```
import ex13.pyrmont.core.SimpleContextConfig;
import org.apache.catalina.Connector;
import org.apache.catalina.Context;
import org.apache.catalina.Engine;
import org.apache.catalina.Host;
import org.apache.catalina.Lifecycle;
import org.apache.catalina.LifecycleListener;
import org.apache.catalina.Loader;
import org.apache.catalina.Wrapper;
import org.apache.catalina.connector.http.HttpConnector;
import org.apache.catalina.core.StandardContext;
import org.apache.catalina.core.StandardEngine;
import org.apache.catalina.core.StandardHost;
import org.apache.catalina.core.StandardWrapper;
import org.apache.catalina.loader.WebappLoader;

public final class Bootstrap2 {
  public static void main(String[] args) {
    System.setProperty("catalina.base",
      System.getProperty("user.dir"));
    Connector connector = new HttpConnector();
    Wrapper wrapper1 = new StandardWrapper();
    wrapper1.setName("Primitive");
    wrapper1.setServletClass("PrimitiveServlet");
    Wrapper wrapper2 = new StandardWrapper();
    wrapper2.setName("Modern");
    wrapper2.setServletClass("ModernServlet");
    Context context = new StandardContext();
    // StandardContext's start method adds a default mapper
    context.setPath("/app1");
    context.setDocBase("app1");
    context.addChild(wrapper1);
    context.addChild(wrapper2);
    LifecycleListener listener = new SimpleContextConfig();
    ((Lifecycle) context).addLifecycleListener(listener);
    Host host = new StandardHost();
    host.addChild(context);
    host.setName("localhost");
    host.setAppBase("webapps");
    Loader loader = new WebappLoader();
    context.setLoader(loader);
    // context.addServletMapping(pattern, name);
    context.addServletMapping("/Primitive", "Primitive");
    context.addServletMapping("/Modern", "Modern");
```

```
    Engine engine = new StandardEngine();
    engine.addChild(host);
    engine.setDefaultHost("localhost");
    connector.setContainer(engine);
    try {
      connector.initialize();
      ((Lifecycle) connector).start();
      ((Lifecycle) engine).start();
      // make the application wait until we press a key.
      System.in.read();
      ((Lifecycle) engine).stop();

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

Running the Applications

要在 Windows 下面运行该程序，可以在工作目录下面输入如下命令

```
java -classpath ./lib/servlet.jar;./lib/commons-
collections.jar;./lib/commons-digester.jar;./
ex13.pyrmont.startup.Bootstrap2
```

在 Linux 里面，需要使用分号分隔开两个库

```
java -classpath ./lib/servlet.jar:./lib/commons-
collections.jar:./lib/commons-digester.jar:./
ex13.pyrmont.startup.Bootstrap2
```

可以在浏览器中输入如下 URL 来调用 PrimitiveServlet

```
http://localhost:8080/app1/Primitive
```

要调用 ModernServlet，可以使用如下 URL

```
http://localhost:8080/app1/Modern
```

总结

在本章中，我们讨论了两种类型的容器：主机和引擎。本章还介绍了这两种容器的相关类。并且用两个程序说明了如何让这两种容器作为顶层容器来工作。

第 14 章：服务器和服务

综述

在前面的章节中已经看到了如何初始化连接器和容器并将它们关联起来作为 Servlet 容器。前面只有一个连接器，通过端口 8080 对 HTTP 请求进行服务。你不能添加另一个连接器来处理例如 HTTPS 的请求。

另外，所有的配套程序都确实一件东西，启动和停止 Servlet 容器的机制。在本章中，我们将看提供了该机制以及其它一些属性的组件：服务器（server）和服务（service）。

Server 服务器

org.apache.catalina.Server 接口表示整个 Catalina Servlet 容器以及其它组件。一个服务器相当有用，因为它提供了一种优雅的机制来启动和停止整个系统。不必再单独的启动连接器和容器了。

下面是启动和停止机制是如何工作的。当服务器启动的时候，它启动它内部的所有组件。然后无限期的等待关闭命令，如果你想要关闭系统，发送一个关闭命令道指定端口即可。当服务器收到正确的关闭指令后，它停止所有组件的服务。

服务器还使用了另外一个组件，服务，它用来持有组件，例如容器或者一个多个的连接器。服务将在本章的 service 小节中介绍。

Server 接口如 Listing14.1 所示

Listing 14.1: The Server interface

```
package org.apache.catalina;

import org.apache.catalina.deploy.NamingResources;

public interface Server {
  /**
   * Return descriptive information about this Server implementation
   * and the corresponding version number, in the format
   * <code>&lt;description&gt;/&lt;version&gt;</code>.
   */
  public String getInfo();
  /**
   * Return the global naming resources.
   */
  public NamingResources getGlobalNamingResources();
  /**
   * Set the global naming resources.
   *
   * @param namingResources The new global naming resources
```

```
 */
public void setGlobalNamingResources
  (NamingResources globalNamingResources);
/**
 * Return the port number we listen to for shutdown commands.
 */
public int getPort();
/**
 * Set the port number we listen to for shutdown commands.
 *
 * @param port The new port number
 */
public void setPort(int port);
/**
 * Return the shutdown command string we are waiting for.
 */
public String getShutdown();
/**
 * Set the shutdown command we are waiting for.
 *
 * @param shutdown The new shutdown command
 */
public void setShutdown(String shutdown);
/**
 * Add a new Service to the set of defined Services.
 *
 * @param service The Service to be added
 */
public void addService(Service service);
/**
 * Wait until a proper shutdown command is received, then return.
 */
public void await();
/**
 * Return the specified Service (if it exists); otherwise return
 * <code>null</code>.
 *
 * @param name Name of the Service to be returned

 */
public Service findService(String name);
/**
 * Return the set of Services defined within this Server.
 */
```

```
  public Service[] findServices();
  /**
   * Remove the specified Service from the set associated from this
   * Server.
   *
   * @param service The Service to be removed
   */
  public void removeService(Service service);
  /**
   * Invoke a pre-startup initialization. This is used to allow
   * onnectors to bind to restricted ports under Unix operating
   * environments.
   *
   * @exception LifecycleException If this server was already
   * initialized.
   */
  public void initialize() throws LifecycleException;
}
```

属性 shutdown 用来持有一个停止服务的指令。属性 port 则是服务器等待关闭命令的端口。可以调用服务器的 addService 方法将服务添加到服务器。使用 removeService 方法将服务删除。findServices 返回所有服务器中所有的服务。Initialize 方法包括在启动之前需要执行的代码。

StandardServer 类

org.apache.catalina.core.StandardServer 类是服务器的标准实现。我们会特别注意这个类最重要的特性，它的关闭机制。而关于服务器通过 server.xml 配置的相关方法，这里不予讨论。感兴趣的话可以自己阅读，并不难理解。

一个服务器可以有零个或多个服务，StandardServer 类提供了 addService、removeService、findServices 方法的实现。另外还有四个跟生命周期相关的方法：initialize 、tart 、stop 以及 await。跟其他组件相似，initialize 初始化和 start 启动一个组件。然后可以使用 await 方法和 stop 方法。Await 方法在收到端口 8085（或其他端口）关闭指令之前会一直等待。当 await 方法返回的时候，调用 stop 方法停止所有字组件。在本章配套的应用程序中，你可以看到如何实现关闭机制。

The initialize, start, stop, and await methods are discussed in the following sub-sections.

关于 initialize, start, stop, 和 await 方法的内容将在下面的子节讨论。

initialize 方法

Initialize 方法用于初始化要添加到服务器实例上的服务，StandardServer 的 initialize 方法如 Listing14.2

Listing 14.2: The initialize method

```
public void initialize() throws LifecycleException {
  if (initialized)
    throw new LifecycleException (
      sm.getString("StandardServer.initialize.initialized"));
  initialized = true;

  // Initialize our defined Services
  for (int i = 0; i < services.length; i++) {
    services[i].initialize();
  }
}
```

注意该方法使用了一个名为 initialized 的变量来避免多次启动该服务器。在 Tomcat5 中，initialize 方法相似，但是包括了 JMX 的相关代码。Stop 方法不会改变 initialized 的值，所以当服务器被停止后再次启动的时候，不会再次调用 initialized 方法。

start 方法

可以使用 start 方法来启动一个服务器，StandardServer 的 start 方法的实现将会启动所有服务及其相关组件，例如连接器和容器。Listing14.3 展示了 start 方法

Listing 14.3: The start method

```
public void start() throws LifecycleException {
  // Validate and update our current component state
  if (started)
    throw new LifecycleException
      (sm.getString("standardServer.start.started"));
  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(BEFORE_START_EVENT, null);
  lifecycle.fireLifecycleEvent(START_EVENT, null);
  started = true;
  // Start our defined Services
  synchronized (services) {
    for (int i = 0; i < services.length; i++) {
      if (services[i] instanceof Lifecycle)
        ((Lifecycle) services[i]).start();
    }
  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(AFTER_START_EVENT, null);
```

```
}
```

该方法使用了一个 started 布尔变量来避免一个服务器被启动两次。Stop 方法会重置该变量的值。

stop 方法

Stop 方法用于停止一个服务器，该方法如 Listing14.4 所示

Listing 14.4: The stop method

```
public void stop() throws LifecycleException {
  // Validate and update our current component state
  if (!started)
    throw new LifecycleException
      (sm.getString("standardServer.stop.notStarted"));
  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(BEFORE_STOP_EVENT, null);
  lifecycle.fireLifecycleEvent(STOP_EVENT, null);
  started = false;

  // Stop our defined Services
  for (int i = 0; i < services.length; i++) {
    if (services[i] instanceof Lifecycle)
      ((Lifecycle) services[i]).stop();
  }
  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(AFTER_STOP_EVENT, null);
}
```

调用 stop 方法可以停止所有的服务并重置 started 变量的值，这样就可以再次启动服务器了。

await 方法

Await 负责整个 Tomcat 部署的停止机制，它的代码如 Listing14.5 所示：

Listing 14.5: The await method

```
/**
* Wait until a proper shutdown command is received, then return.
*/
public void await() {
  // Set up a server socket to wait on
  ServerSocket serverSocket = null;
  try {
```

```
  serverSocket = new ServerSocket(port, 1,
    InetAddress.getByName("127.0.0.1"));
}
catch (IOException e) {
  System.err.println("StandardServer.await: create[" +
    port + "]: " + e);
  e.printStackTrace();
  System.exit(1);
}
// Loop waiting for a connection and a valid command
while (true) {
  // Wait for the next connection
  Socket socket = null;
  InputStream stream = null;
  try {
    socket = serverSocket.accept();
    socket.setSoTimeout(10 * 1000); // Ten seconds
    stream = socket.getInputStream();
  }
  catch (AccessControlException ace) {
    System.err.println("StandardServer.accept security exception: "
      + ace.getMessage());
    continue;
  }
  catch (IOException e) {
    System.err.println("StandardServer.await: accept: " + e);
    e.printStackTrace();
    System.exit(1);
  }

  // Read a set of characters from the socket
  StringBuffer command = new StringBuffer();
  int expected = 1024; // Cut off to avoid DoS attack
  while (expected < shutdown.length()) {
    if (random == null)
      random = new Random(System.currentTimeMillis());
    expected += (random.nextInt() % 1024);
  }
  while (expected > 0) {
    int ch = -1;
    try {

      ch = stream, read();
    }
```

```
      catch (IOException e) {
        System.err.println("StandardServer.await: read: " + e);
        e.printStackTrace();
        ch = -1;
      }
      if (ch < 32)  // Control character or EOF terminates loop
        break;
      command.append((char) ch);
      expected--;
    }

    // Close the socket now that we are done with it
    try {
      socket.close();
    }
    catch (IOException e) {
      ;
    }

    // Match against our command string
    boolean match = command.toString().equals(shutdown);
    if (match) {
      break;
    }
    else
      System.err.println("StandardServer.await: Invalid command '" +
        command.toString() + "' received");
  }

  // Close the server socket and return
  try {
    serverSocket.close();
  }
  catch (IOException e) {
    ;
  }
}
```

方法 await 在 8085 端口创建一个 ServerSocket 对象，在 while 循环调用它的 accept 方法。Accept 方法仅仅接受 8085 端口的信息。它将接受到的信息跟 shutdown 命令进行匹配，如果匹配的话跳出循环关闭 SocketServer，如果不匹配继续 while 循环等待另一个命令。

服务

org.apache.catalina.Service 接口用于表示服务。一个服务可以可以有一个容器和多个连接器。你可以添加多个连接器 ，并将它们跟容器相关联。Service 接口如 Listing14.6 所示

Listing 14.6: The Service interface

```
package org.apache.catalina;

public interface Service {

  /**
   * Return the <code>Container</code> that handles requests for all
   * <code>Connectors</code> associated with this Service.
   */
  public Container getContainer();

  /**
   * Set the <code>Container</code> that handles requests for all
   * <code>Connectors</code> associated with this Service.
   *
   * @param container The new Container
   */
  public void setContainer(Container container);

  /**
   * Return descriptive information about this Service implementation
   * and the corresponding version number, in the format
   * <code>&lt;description&gt;/&lt;version&gt;</code>.
   */
  public String getInfo();

  /**
   * Return the name of this Service.
   */
  public String getName();

  /**
   * Set the name of this Service.
   *
   * @param name The new service name
   */
  public void setName(String name);

  /**
   * Return the <code>Server</code> with which we are associated
   * (if any).
```

```
     */
    public Server getServer();

    /**
     * Set the <code>Server</code> with which we are associated (if any).
     *
     * @param server The server that owns this Service
     */
    public void setServer(Server server);

    /**

     * Add a new Connector to the set of defined Connectors,
     * and associate it with this Service's Container.
     *
     * @param connector The Connector to be added
     */
    public void addConnector(Connector connector);

    /**
     * Find and return the set of Connectors associated with
     * this Service.
     */
    public Connector[] findConnectors();

    /**
     * Remove the specified Connector from the set associated from this
     * Service.  The removed Connector will also be disassociated
     * from our Container.
     *
     * @param connector The Connector to be removed
     */
    public void removeConnector(Connector connector);

    /**
     * Invoke a pre-startup initialization. This is used to
     * allow connectors to bind to restricted ports under
     * Unix operating environments.
     *
     * @exception LifecycleException If this server was
     * already initialized.
     */
    public void initialize() throws LifecycleException;
}
```

StandardService 类

org.apache.catalina.core.StandardService 类是 Service 接口的标准实现。StandardService 类的 initialize 方法初始化所有的添加到该服务的连接器。该类还实现了 org.apache.catalina.Lifecycle 接口，所以调用它的 start 方法能启动所有的连接器和容器。

容器和连接器

一个 StandardService 实例包括两种组件：一个容器和多个连接器。多个连接器可以使得 Tomcat 能服务于多个协议。一个协议用处处理 HTTP 请求，另一个用于处理 HTTPS 请求。

StandardService 类用 container 变量来持有容器实例，用 connectors 数组来持有所有的连接器

```
private Container container = null;
private Connector connectors[] = new Connector[0];
```

要将一个容器跟一个服务相关联，可以使用它的 setContainer 方法，如 Listing14.7 所示

Listing 14.7: The setContainer method

```
public void setContainer(Container container) {
  Container oldContainer = this.container;
  if ((oldContainer != null) && (oldContainer instanceof Engine))
    ((Engine) oldContainer).setService(null);
  this.container = container;
  if ((this.container != null) && (this.container instanceof Engine))
   ((Engine) this.container).setService(this);
  if (started && (this.container != null) &&
    (this.container instanceof Lifecycle)) {
    try {
      ((Lifecycle) this.container).start();
    }
    catch (LifecycleException e) {
      ;
    }
  }
  synchronized (connectors) {
    for (int i = 0; i < connectors.length; i++)
      connectors[i].setContainer(this.container);
  }
  if (started && (oldContainer != null) &&
    (oldContainer instanceof Lifecycle)) {
    try {
```

```
      ((Lifecycle) oldContainer).stop();
    }
    catch (LifecycleException e) {
      ;
    }
  }

  // Report this property change to interested listeners
  support.firePropertyChange("container", oldContainer,
    this.container);
}
```

要与服务相关联的容器传递给该每个连接器，这样来建立容器和每个连接器的关系。

要给一个服务添加连接器，可以使用 addConnector 方法。要删除一个连接器，可以使用 removeConnector 方法。它们分别如 Listing14.8 和 Listing14.8 所示

Listing 14.8: The addConnector method

```
public void addConnector(Connector connector) {
  synchronized (connectors) {
    connector.setContainer(this.container);
    connector.setService(this);
    Connector results[] = new Connector[connectors.length + 1];
    System.arraycopy(connectors, 0, results, 0, connectors.length);
    results[connectors.length] = connector;
    connectors = results;

    if (initialized) {
      try {
        connector.initialize();
      }
    catch (LifecycleException e) {
      e.printStackTrace(System.err);
    }
  }

  if (started && (connector instanceof Lifecycle)) {
    try {
      ((Lifecycle) connector).start();
    }
    catch (LifecycleException e) {
      ;
    }
  }
```

```
  // Report this property change to interested listeners
  support.firePropertyChange("connector", null, connector);
  }
}
```

Listing 14.9: The removeConnector method

```
public void removeConnector(Connector connector) {
  synchronized (connectors) {
  int j = -1;
  for (int i = 0; i < connectors.length; i++) {
    if (connector == connectors [i]) {
      j = i;
      break;
    }
  }
  if (j < 0)
    return;
  if (started && (connectors[j] instanceof Lifecycle)) {
    try {

      ((Lifecycle) connectors[j]).stop();
    }
    catch (LifecycleException e) {
      ;
    }
  }
  connectors[j].setContainer(null);
  connector.setService(null);
  int k = 0;
  Connector results[] = new Connector[connectors.length - 1];
  for (int i = 0; i < connectors.length; i++) {
    if (i != j)
      results[k++] = connectors[i];
  }
  connectors = results;

  // Report this property change to interested listeners
  support.firePropertyChange("connector", connector, null);
  }
}
```

The addConnector method initializes and starts the added connector.

Lifecycle 方法

Lifecycle 方法是从 Lifecycle 接口继承而来。Initialize 方法调用每个连接器的 initialize 方法，该方法如 Listing14.10 所示

Listing 14.10: The initialize method of StandardService

```
public void initialize() throws LifecycleException {
  if (initialized)
    throw new LifecycleException (
      sm.getString("StandardService.initialize.initialized"));
  initialized = true;

  // Initialize our defined Connectors
  synchronized (connectors) {
    for (int i = 0; i < connectors.length; i++) {
      connectors[i].initialize();
    }
  }
}
```

Start 方法用于启动相关联的连接器和容器，如 Listing14.11 所示

Listing 14.11: The start method

```
public void start() throws LifecycleException {
  // Validate and update our current component state
  if (started) {

    throw new LifecycleException
      (sm.getString("standardService.start.started"));
  }

  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(BEFORE_START_EVENT, null);

  System.out.println
    (sm.getString("standardService.start.name", this.name));
  lifecycle.fireLifecycleEvent(START_EVENT, null);
  started = true;

  // Start our defined Container first
  if (container != null) {
    synchronized (container) {
      if (container instanceof Lifecycle) {
        ((Lifecycle) container).start();
```

```
      }
    }
  }

  // Start our defined Connectors second
  synchronized (connectors) {
    for (int i = 0; i < connectors.length; i++) {
      if (connectors[i] instanceof Lifecycle)
        ((Lifecycle) connectors[i]).start();
    }
  }

  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(AFTER_START_EVENT, null);
}
```

Stop 方法关闭所有关联连接器和容器。Stop 方法如 Listing14.12 所示

Listing 14.12: The stop method

```
public void stop() throws LifecycleException {
  // Validate and update our current component state
  if (!started) {
    throw new LifecycleException
      (sm.getString("standardService.stop.notStarted"));
  }

  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(BEFORE_STOP_EVENT, null);

  lifecycle.fireLifecycleEvent(STOP_EVENT, null);

  System.out.println
    (sm.getString("standardService.stop.name", this.name));
  started = false;

  // Stop our defined Connectors first
  synchronized (connectors) {

    for (int i = 0; i < connectors.length; i++) {
      if (connectors[i] instanceof Lifecycle)
        ((Lifecycle) connectors[i]).stop();
    }
  }

  // Stop our defined Container second
```

```
  if (container != null) {
    synchronized (container) {
      if (container instanceof Lifecycle) {
        ((Lifecycle) container).stop();
      }
    }
  }

  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(AFTER_STOP_EVENT, null);
}
```

The Application

这个应用程序展示了如何使用服务器和服务，特别是它说明了如何使用 StandardServer 类的启动和停止机制。本程序由三个类组成，SimpleContextConfig 跟第 13 章中的相同，另外两个类一个是 Bootstrap 启动程序，另一个是 Stopper 类用来停止它。

Bootstrap 类

Bootstrap 类如 Listing14.13 所示

Listing 14.13: The Bootstrap Class

```
package ex14.pyrmont.startup;

import ex14.pyrmont.core.SimpleContextConfig;
import org.apache.catalina.Connector;
import org.apache.catalina.Context;
import org.apache.catalina.Engine;
import org.apache.catalina.Host;
import org.apache.catalina.Lifecycle;
import org.apache.catalina.LifecycleException;
import org.apache.catalina.LifecycleListener;
import org.apache.catalina.Loader;
import org.apache.catalina.Server;
import org.apache.catalina.Service;
import org.apache.catalina.Wrapper;
import org.apache.catalina.connector.http.HttpConnector;

import org.apache.catalina.core.StandardContext;
import org.apache.catalina.core.StandardEngine;
import org.apache.catalina.core.StandardHost;
import org.apache.catalina.core.StandardServer;
import org.apache.catalina.core.StandardService;
```

```
import org.apache.catalina.core.StandardWrapper;
import org.apache.catalina.loader.WebappLoader;

public final class Bootstrap {
  public static void main(String[] args) {
    System.setProperty("catalina.base",
      System.getProperty("user.dir"));
    Connector connector = new HttpConnector();
    Wrapper wrapper1 = new StandardWrapper();
    wrapper1.setName("Primitive");
    wrapper1.setServletClass("PrimitiveServlet");
    Wrapper wrapper2 = new StandardWrapper();
    wrapper2.setName("Modern");
    wrapper2.setServletClass("ModernServlet");
    Context context = new StandardContext();
    // StandardContext's start method adds a default mapper
    context.setPath("/app1");
    context.setDocBase("app1");
    context.addChild(wrapper1);
    context.addChild(wrapper2);
    LifecycleListener listener = new SimpleContextConfig();
    ((Lifecycle) context).addLifecycleListener(listener);
    Host host = new StandardHost();
    host.addChild(context);
    host.setName("localhost");
    host.setAppBase("webapps");
    Loader loader = new WebappLoader();
    context.setLoader(loader);
    // context.addServletMapping(pattern, name);
    context.addServletMapping("/Primitive", "Primitive");
    context.addServletMapping("/Modern", "Modern");
    Engine engine = new StandardEngine();
    engine.addChild(host);
    engine.setDefaultHost("localhost");
    Service service = new StandardService();
    service.setName("Stand-alone Service");
    Server server = new StandardServer();
    server.addService(service);
    service.addConnector(connector);
    // StandardService class's setContainer method calls
    // its connectors' setContainer method
    service.setContainer(engine);
    // Start the new server
    if (server instanceof Lifecycle) {
```

```
      try {
        server.initialize();
        ((Lifecycle) server).start();
        server.await();
        // the program waits until the await method returns,
        // i.e. until a shutdown command is received.
      }

      catch (LifecycleException e) {
        e.printStackTrace(System.out);
      }
    }

    // Shut down the server
    if (server instanceof Lifecycle) {
      try {
        ((Lifecycle) server).stop();
      }
      catch (LifecycleException e) {
        e.printStackTrace(System.out);
      }
    }
  }
}
```

Bootstrap 主方法的开始部分跟第 13 章相似。它创建了一个连接器、两个包装器、一个上下文、一个主机以及一个引擎。然后将包装器添加到上下文，将上下文添加到主机，主机添加到引擎。它并没有将连接器和引擎相关联，而是创建了一个一个服务对象，设置它的名字，创建一个服务器对象，并给该服务器添加服务。

```
Service service = new StandardService();
service.setName("Stand-alone Service");
Server server = new StandardServer();
server.addService(service);
```

然后，主方法将连接器和引擎添加到服务上。

```
service.addConnector(connector);
service.setContainer(engine);
```

这样讲连接器添加到服务上，连接器在服务上跟容器相关联。

然后主方法调用服务器的 initialize 和 start 方法。初始化连接器并启动它以及容器。

```
if (server instanceof Lifecycle) {
  try {
```

```
        server.initialize();
        ((Lifecycle) server).start();
```

接下来，调用服务器的 await 方法让服务器等待一个关闭命令。注意现在连接器已经启动了，在 8080 端口为 HTTP 服务进行服务。

```
    server.await() ;
```

Await 方法在接受到关闭指令以前会一直等待，接受到关闭指令后，调用服务器的 stop 方法，关闭其它所有组件。

接下来看用于停止服务器的 Stopper 类。

Stopper 类

在前面的应用程序中，是通过按键来关闭容器。在本章中，Stopper 类提供了一种优雅的方式来关闭 Catalina 服务器。它还保证了所有生命周期组件的 stop 方法会被调用。Stopper 类如 Listing14.14 所示

Listing 14.14: The Stopper class

```
package ex14.pyrmont.startup;

import java.io.OutputStream;
import java.io.IOException;
import java.net.Socket;

public class Stopper {
  public static void main(String[] args) {
    // the following code is taken from the Stop method of
    // the org.apache.catalina.startup.Catalina class
    int port = 8005;
    try {
      Socket socket = new Socket("127.0.0.1", port);
      OutputStream stream = socket.getOutputStream();
      String shutdown = "SHUTDOWN";
      for (int i = 0; i < shutdown.length(); i++)
        stream.write(shutdown.charAt(i));
      stream.flush();
      stream.close();
      socket.close();
      System.out.println("The server was successfully shut down.");
    }
    catch (IOException e) {
      System.out.println("Error. The server has not been started.");
    }
  }
}
```

主方法创建一个 Socket 对象然后将关闭命令发送到 8085 端口。如果 Catalina 服务器正在运行，它将会被关闭。

运行应用程序

在 Windows 里面，可以在工作目录输入如下命令来运行该应用程序：

```
java -classpath./lib/servlet.jar;./lib/commons-collections.jar;./lib/commons-digester.jar;./lib/naming-factory.jar;./lib/naming-common.jar;./ ex14.pyrmont.startup.Bootstrap
```

在 Linux 里，需要使用冒号来分隔开两个库

```
java -classpath ./lib/servlet.jar:./lib/commons-collections.jar:./lib/commons-digester.jar:./lib/naming-factory.jar:./lib/naming-common.jar:./ ex14.pyrmont.startup.Bootstrap
```

可以在浏览器输入如下 URL 来调用 PrimitiveServlet

```
http://localhost:8080/app1/Primitive
```

要调用 invoke 方法，可以使用如下 URL

```
http://localhost:8080/app1/Modern
```

可以在工作目录下面运行 Stopper 来停止该应用程序：

```
java ex14.pyrmont.startup.Stopper
```

注意在一个真正的 Catalina 部署中，提供停止服务的 Stopper 类的功能会被包装在 Bootstrap 类中

总结

本章解释了两个 Catalina 的重要组件：服务器和服务。服务器是非常有用，它提供了一种优雅的机制来启动和停止一个 Catalina 部署。而一个服务器

第 15 章：Digester

综述

在前面章节中已经看到，使用 Bootstrap 类来初始化连接器、上下文、包装器以及其它组件。一旦你获得了它们的对象就可以使用 set 方法来关联它们。例如可以如下初始化连接器和上下文

```
Connector connector = new HttpConnector();
Context context = new StandardContext();
```

将连接器和上下文关联起来可以如下实现：

```
connector.setContainer(context);
```

可以使用相应的 set 方法来配置这些对象的属性。例如可以使用 setPath 和 setDocBase 方法来设置 path 和 docBase 属性。

```
context.setPath("/myApp") ;
context.setDocBase("myApp");
```

另外，可以初始化各种组件，然后使用相应的 add 方法将其添加到上下文容器中。例如，下面是如何在上下文对象中添加生命周期监听器和加载器：

```
LifecycleListener listener = new SimpleContextConfig();
((Lifecycle) context).addLifecycleListener(listener);
Loader loader = new WebappLoader();
context.setLoader(loader);
```

一旦必要的关联和添加设置完毕，就可以调用连接器的 initialize 和 start 方法和上下文的 start 方法了。

```
connector.initialize();
((Lifecycle) connector).start ();
((Lifecycle) context).start();
```

这种方式来配置应用程序有一个很明显的缺点，所有的东西都是硬编码的。要更改一个组件或者一个属性的值都需要重新编译整个 Bootstrap 类。幸运的是，Tomcat 的设计者选择了一种更优雅的方式来进行配置，使用名为 server.xml 的 XML 文档。Server.xml 中的每一个元素都被转换为一个 Java 对象，元素的属性用来设置属性。这样，就可以通过编辑 server.xml 来改变 Tomcat 的配置。例如，上下文容器元素就可以这样在 server.xml 中表示

```
<context/>
```

To set the path and docBase properties you use attributes in the XML element:

```
<context docBase="myApp" path="/myApp"/>
```

Tomcat 使用开源工具 Digester 来讲 XML 元素转换为 Java 对象。Digester 将会在本章第一节介绍。

接下来的一节介绍了如何配置一个 web 应用程序，一个上下文被用来表示一个 web 应用程序，一次配置初始化该上下文实例即可达到配置该 web 应用的目的。配置 web 应用所使用的 web 应用所使用的 XML 文件时 web.xml，该文件必须存放在该应用程序的 WEB-INF 目录下面。

Digester

Digester 是 Apache Jakarta 项目下面的开源项目。可以在 http://jakarta.apache.org/commons/digester/下载到 Digester。Digester 由三个包组成，被包装到 commons-digeser.jar 文件中：

- org.apache.commons.digester.提供了基于规则（rules-based）的任意 XML 文档处理。
- org.apache.commons.digester.rss.演示了如何使用 Digester 解析 XML 文档，跟很多地方使用的 rich site summary 格式比较。
- org.apache.commons.digester.xmlrules.该包提供了给 Digester 提供基于 XML 文档的规则。

我们不会介绍着三个包的所有成员，而是重点介绍在 Tomcat 中重要的几个类型。本节首先介绍 Digester 库里最重要的 Digester 类：

Digester 类

org.apache.commons.digester.Digester 类是 Digester 库里的主类。使用它来解析 XML 文档。对于该文档中的每一个元素，Digester 都检查它是否需要做点事情，程序员只需决定 Digester 实例在调用 parser 方法之前需要做什么即可。

你怎样告诉一个 Digester 对象遇到一个 XML 元素的时候怎么做？很简单，你定义模式并且将模式跟一条或多条规则相关联即可。XML 的根元素有一个跟他的元素名相同的模式，例如：Listing15.1 所示的 XML 文档。

Listing 15.1: The example.xml file

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<employee firstName="Brian" lastName="May">
  <office>
    <address streeName="Wellington Street" streetNumber="110"/>
  </office>
</employee>
```

该文档的根元素师 employee，该元素的模式 emploee。它的 office 元素是<employee>的子元素。一个子元素的模式是它的名字加上它属于的元素作为前缀。所以 office 元素的模式是 employee/office，而 address 元素的模式是：

the parent element's pattern + "/" + the name of the element

Address 元素的父元素师<office>，<office>元素的模式 employee/office。因此，<address>的模式 employee/office/address。

现在你已经明白了如何从 XML 文档中获得模式，接下来讨论下规则（rules）。

一个规则定义了 Digester 遇到特别的模式的时候必须做的动作。一个规则用 org.apache.commons.digester.Rule 类。Digester 类包括零个或多个 Rule 对象。在一个 Digester 实例中，规则及其模式被存储在接口 org.apache.commons.digester.Rules 定义的类型中。每一次给 Digester 实例添加规则，都将 Rule 对象添加到 Rules 对象。

在 Rule 中有两个方法：begin 方法和 end 方法。当解析一个 XML 文档的时候，遇到开始元素，Digester 实例调用 Rule 对象的 begin 方法，而遇到结束元素的时候调用 stop 方法。

当解析如 Listing15.1 所示的 example.xml 所示的文档的时候，下面是 Digester 对象所作的。

- 第一次遇到 employee 开始元素，检查是否已经有存在的规则模式 employee。如果有，Digester 调用 Rule 对象的 begin 方法，从 begin 方法开始添加第一个模式。
- 然后检查 office 元素，所以 Digester 对象检查是否存在规则模式为 employee/office。如果有，，调用 Rule 对象的 begin 方法、
- 接下来检查模式 **employee/office/address**，如果找到了规则，则调用 begin 方法
- 接下来 Digester 遇到了 address 结束符，调用对应规则的 end 方法。
- 解析来遇到了 office 结束符，调用相应规则的 end 方法。
- 最后遇到了 employee 结束符，调用相应规则的 end 方法。

可以使用什么规则？Digester 预先定义了一些规则，甚至在不了解 Rule 类的时候都可以使用这些规则。但是，如果这些规则不足够，你需要建立自己的规则。实现定义的规则包括创建对象，设置属性值等。

创建对象

如果想让 Digester 给特定的规则创建对象，可以调用它的 addObjectCreate 方法。该方法有四个实现，其中两个最常用方法的签名如下：

```
public void addObjectCreate(java.lang.String pattern,
  java.lang.Class clazz)

public void addObjectCreate(java.lang.String pattern,
  java.lang.String className)
```

传递一个模式和类对象(或者是类名)给它们。例如,你想 Digester 根据 employee 模式，创建一个 Employee 对象（类为 ex15.pyrmont.digestertest.Employee）可以使用如下代码：

```
digester.addObjectCreate("employee",
  ex15.pyrmont.digestertest.Employee.class);
```

或者

```
digester.addObjectCreate("employee",
  "ex15.pyrmont.digestertest.Employee");
```

方法 addObjectCreate 的另外两个实现允许在 XML 文档中定义类名,而不是通过作为方法参数。这个特性带来了强大的功能，它使得类名可以在运行时决定，下面是这两个方法的签名：

```
public void addObjectCreate(java.lang.String pattern,
  java.lang.String className, java.lang.String attributeName)

public void addObjectCreate(java.lang.String pattern,
  java.lang.String attributeName, java.lang.Class clazz)
```

在这两个实现中，attributeName 参数定义了 XML 文档中的属性，名字有 className 指定。例如，如果使用下面的代码来定义创建对象规则：

```
digester.addObjectCreate("employee", null, "className");
```

其中属性名为 className

然后在 XML 中传递类名：

```
<employee firstName="Brian" lastName="May"
  className="ex15.pyrmont.digestertest.Employee">
```

或者可以如下在 addObjectCreate 方法中定义默认的类名：

```
digester.addObjectCreate("employee",
  "ex15.pyrmont.digestertest.Employee", "className");
```

如果 employee 元素包括一个 className 属性，就用该属性的值来进行类的初始化，如果没有改属性，使用默认值来进行初始化。

使用 addObejectCreate 方法创建的对象被压到一个内部堆栈中,并定义来 peek、push 以及 pop 方法来操作创建的对象。

设置属性

另一个重要的方法是 addSetProperties，Digester 对象可以通过它设置对象属性。该方法的一个实现的签名如下：

```
public void addSetProperties(java.lang.String pattern)
```

传递一个模式给该方法，例如下面的代码：

```
digester.addObjectCreate("employee",
```

```
  "ex15.pyrmont.digestertest.Employee");
digester.addSetProperties("employee");
```

上面的 Digester 实例有两个规则：创建对象，设置属性。都是关于 employee 模式的。根据添加的顺序来执行这些规则。XML 文档中如下的 employee 元素

```
<employee firstName="Brian" lastName="May">
```

Digester 实例首先创建一个 ex15.pyrmont.digestertest.Employee 类的实例，由第一个规则得到。然后 ex15.pyrmont.digestertest.Employee 使用第二条规则来根据 XML 文档调用 setFirstName 和 setLasetName 属性来设置 Employee 对象的值。属性的值跟 XML 文档中相应元素的属性值是一致的，如果 Employee 类没有定义该属，会产生错误。

方法调用

Digester 允许通过添加规则，见到相应的模式的时候就尅调用栈最高层中对象的方法。该方法名为 addCallMethod，它的一个实现的签名如下：

```
public void addCallMethod (java.lang.String pattern,
  java.lang.String methodName)
```

建立对象间的联系

Digester 实例有一个栈用来临时存储对象。当调用 addObjectCreate 创建对象后，将对象压入堆栈中。可以把堆栈想象成一口井，可以将对象放入到井中，而 pop 方法相当于取出井中最上边的元素。

当通过 addObjectCreate 方法创建两个对象的时候，第一个对象被放入井中，然后是第二个。addSetNext 用于建立第一个对象和第二个对象之间的关系，它把第二个对象作为参数传递给第一个对象。下面是 addSetNext 方法的签名：

```
public void addSetNext(java.lang.String pattern,
  java.lang.String methodName)
```

参数 argument 定义了触发该规则的模式，methodName 参数是是第一个对象要被调用的方法名。该模式的形式如 **firstObject/secondObject。**

例如，一个 employee 可以有一个 office，要创建一个 employee 和他的 office 之间的关系，首先需要使用两个 addObjectCreate 方法。

```
digester.addObjectCreate("employee",
  "ex15.pyrmont.digestertest.Employee");
digester.addObjectCreate("employee/office",
  "ex15.pyrmont.digestertest.Office");
```

第一个 addObjectCreate 方法根据 employee 元素创建一个 Employee 类实例。第二个 addObjectCreate 方法根据<employee>下面的<office>创建一个 Office 实例。

这两个 addObjectCreate 方法将两个对象压入到堆栈中，现在对象在栈底部，Office 对象在栈顶部。要建立它们之间的关系，可以使用 addSetNext 方法方法：

digester.addSetNext("employee/office", "addOffice");

其中 addOffice 是 Employee 类的方法，该方法必须接受一个 Office 对象作为参数，第二个 Digester 实例将会说明仔细介绍 setSetNext 方法。

验证 XML 文档

可以使用 Digester 来对 XML 文档的结构进行验证，一个 XML 文档是否合法取决于由 Digester 定义的 validating 属性，该属性的默认值为 false。

方法 setValidating 用来用来设置是否要验证 XML 文档，该方法的签名如下：

public void setValidating(boolean validating)

如果想要验证 XML 文档，可以传递一个 true 值该该方法：

Digester Example 1

第一个例子说明了如何使用 Digester 动态的创建对象并设置它的属性。考虑 Listing15.2 所示的类 Employee 用 Digester 来初始化：

Listing 15.2: The Employee Class

```
package ex15.pyrmont.digestertest;
import java.util.ArrayList;
public class Employee {
  private String firstName;
  private String lastName;
  private ArrayList offices = new ArrayList();

  public Employee () {
    System.out.println ("Creating Employee");
  }
  public String getFirstName() {
    return firstName;
  }
  public void setFirstName(String firstName) {
    System.out.println("Setting firstName : " + firstName);
    this.firstName = firstName;
  }
  public String getLastName() {
    return lastName;
```

```
  }
  public void setLastName(String lastName) {
    System.out.println("Setting lastName : " + lastName);
    this.lastName = lastName;
  }
  public void addOffice(Office office) {

    System.out.println("Adding Office to this employee");
    offices.add(office);
  }
  public ArrayList getOffices() {
    return offices;
  }
  public void printName() {
    System.out.println("My name is " + firstName + " " + lastName);
  }
}
```

Employee 类有三个属性:firstName、lastName 和 office。firstName 和 lastName 类型为 String 类型，office 类型为 ex15.pyrmont.digester.Office 类型。Office 属性用作 Digester 的第二个例子。

Employee 还有一个方法：printName 方法将 first name 和 last name 打印到 console 上面。

接下来写一个测试类用 Digester 来创建 Employee 对象并设置它的属性，如 Listing15.3 所示的 Test01 所示的类即可。

Listing 15.3: The Test01 Class

```
package ex15.pyrmont.digestertest;

import java.io.File;
import org.apache.commons.digester.Digester;

public class Test01 {

  public static void main(String[] args) {
    String path = System.getProperty("user.dir") + File.separator +
      "etc";
    File file = new File(path, "employee1.xml");
    Digester digester = new Digester();
    // add rules
    digester.addObjectCreate("employee",
      "ex15.pyrmont.digestertest.Employee");
    digester.addSetProperties("employee");
    digester.addCallMethod("employee", "printName");
```

```
    try {
      Employee employee = (Employee) digester.parse(file);
      System.out.println("First name : " + employee.getFirstName());
      System.out.println("Last name : " + employee.getLastName());
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

首先定义定义 XML 所在的路径并将其传递给 File 类的构造函数，然后创建一个 Digester 对象并添加 employee 的如下规则：

```
digester.addObjectCreate("employee",
  "ex15.pyrmont.digestertest.Employee");
digester.addSetProperties("employee");
digester.addCallMethod("employee", "printName");
```

接下来调用调用 Digester 类的 parse 方法，将该 XML 文档作为参数。该方法的返回值是 Digester 类的内部栈的第一个对象。

```
Employee employee = (Employee) digester.parse(file);
```

这样通过 Digester 获得了一个 Employee 对象的对象，接下来看该对象的属性，调用 getFirstName 方法和 getLastName 方法即可：

```
System.out.println("First name : " + employee.getFirstName());
System.out.println("Last name : " + employee.getLastName());
```

现在看 Listing15.4 所示的 employee1.xml 文档，根为 employee，该元素有两个属性，firstName 和 lastName。

Listing 15.4: The employee1.xml file

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<employee firstName="Brian" lastName="May">
</employee>
```

运行 Test01 类得到如下运行结果：

```
Creating Employee
Setting firstName : Brian
Setting lastName : May
My name is Brian May
First name : Brian
Last name : May
```

下面看问什么会这样

当调用 Digester 对象的 parse 方法的时候，它打开 XML 文档开始解析它。首先，Digester 看到了 employee 的开始元素。这样触发了关于 employee 模式的三个规则，第一个是创建一个对象，所以 Digester 初始化一个 Employee 类的对象，这样需要调用 Employee 类的构造函数，该构造函数打印出字符串“Creating Employee”。

第二个规则设置 Employee 对象的属性，该元素有两个属性：firstName 和 lastName。该规则调用方法这两个属性的 set 方法。这两个 set 方法打印出如下字符串：

```
Setting firstName : Brian
Setting lastName : May
```

第三个规则调用 printName 方法，打印出如下内容：

```
My name is Brian May
```

接下来，最后两行诗调用 getFirstName 和 getLastName 方法的结果。

```
First name : Brian
Last name : May
```

Digester Example 2

Digester 类的第二个例子说明了如何创建两个对象，并建立它们之间的关系。关系的定义需要事先定义好。例如，一个 employee 在一个或多个 office 里工作。一个 office 用 Office 类表示。可以创建一个 Employee 类和 Office 类的对象并建立它们之间的关系。Office 类 Listing15.5 所示：

Listing 15.5: The Office Class

```
package ex15.pyrmont.digestertest;

public class Office {
  private Address address;
  private String description;
  public Office() {
    System.out.println("..Creating Office");
  }
  public String getDescription() {
    return description;
  }
  public void setDescription(String description) {
    System.out.println("..Setting office description : " +
description);
    this.description = description;
  }
  public Address getAddress() {
    return address;
```

```
  }
  public void setAddress(Address address) {
    System.out.println("..Setting office address : " + address);

    this.address = address;
  }
}
```

调用父对象的方法来建立它们之间的关系，注意该例子使用的是 Listing15.2 所示的 Employee 类，它由 addOffice 方法来建立它们之间的关系。

如果不适用 Digester 的话，可以使用如下 Java 代码来实现

```
Employee employee = new Employee();
Office office = new Office();
employee.addOffice(office);
```

每一个 office 都有一个 address，而 address 如 Listing15.6 所示：

Listing 15.6: The Address Class

```
package ex15.pyrmont.digestertest;

public class Address {
  private String streetName;
  private String streetNumber;
  public Adress () {
    System.out.println("....Creating Address");
  }
  public String getStreetName() {
    return streetName;
  }
  public void setStreetName(String streetName) {
    System.out.println("....Setting streetName : " + streetName);
    this.streetName = streetName;
  }
  public String getStreetNumber() {
    return streetNumber;
  }
  public void setStreetNumber(String streetNumber) {
    System.out.println("....Setting streetNumber : " + streetNumber);
    this.streetNumber = streetNumber;
  }
  public String toString() {
    return "...." + streetNumber + " " + streetName;
  }
}
```

要将一个 address 赋值给 office，可以使用 Office 类的 setAddress 方法。如果不适用 Digester，可以使用如下代码实现：

```
Office office = new Office();
Address address = new Address();
office.setAddress (address);
```

第二个 Digester 例子说明如何创建多个对象并建立它们之间的关系。例子中会使用到 Employee、Office 和 Address 类，Test02 使用 Digester 并向上添加规则：

Listing 15.7: The Test02 Class

```
package ex15.pyrmont.digestertest;

import java.io.File;
import java.util.*;
import org.apache.commons.digester.Digester;

public class Test02 {

  public static void main(String[] args) {
    String path = System.getProperty("user.dir") + File.separator +
      "etc";
    File file = new File(path, "employee2.xml");
    Digester digester = new Digester();
    // add rules
    digester.addObjectCreate("employee",
      "ex15.pyrmont.digestertest.Employee");
    digester.addSetProperties("employee");
    digester.addObjectCreate("employee/office",
      "ex15.pyrmont.digestertest.Office");
    digester.addSetProperties("employee/office");
    digester.addSetNext("employee/office", "addOffice");
    digester.addObjectCreate("employee/office/address",
      "ex15.pyrmont.digestertest.Address");
    digester.addSetProperties("employee/office/address");
    digester.addSetNext("employee/office/address", "setAddress");
    try {
      Employee employee = (Employee) digester.parse(file);
      ArrayList offices = employee.getOffices();
      Iterator iterator = offices.iterator();
      System.out.println(
        "-----------------------------------------------");
      while (iterator.hasNext()) {
        Office office = (Office) iterator.next();
        Address address = office.getAddress();
```

```
        System.out.println(office.getDescription());
        System.out.println("Address : " +
          address.getStreetNumber() + " " + address.getStreetName());
        System.out.println(" ------------------------------");
      }
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

要看到Digester如何工作的，可以使用Listing15.8所示的XML文档：

Listing 15.8: The employee2.xml file

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<employee firstName="Freddie" lastName="Mercury">
  <office description="Headquarters">
    <address streetName="Wellington Avenue" streetNumber="223"/>
  </office>
  <office description="Client site">
    <address streetName="Downing Street" streetNumber="10"/>
  </office>
</employee>
```

Test02类的运行结果如下：

```
Creating Employee
Setting firstName : Freddie
Setting lastName : Mercury
..Creating Office
..Setting office description : Headquarters
....Creating Address
....Setting streetName : Wellington Avenue
....Setting streetNumber : 223
..Setting office address : ....223 Wellington Avenue
Adding Office to this employee
..Creating Office
..Setting office description : Client site
....Creating Address
....Setting streetName : Downing Street
....Setting streetNumber : 10
..Setting office address : ....10 Downing Street
Adding Office to this employee
------------------------------------------------
```

```
Headquarters
Address : 223 Wellington Avenue
-------------------------------
Client site
Address : 10 Downing Street
-------------------------------
```

Rule类

Rule 类有多个方法，其中两个最重要的是 begin 和 end 方法。当 Digester 实例遇到一个 XML 元素的开始标志的使用，调用所有匹配规则的 begin 方法。Rule 类的 begin 方法签名如下：

```
public void begin(org.xml.sax.Attributes attributes)
  throws java.lang.Exception
```

当 Digester 实例遇到 XML 元素的 end 时候，调用所有匹配规则的 end 方法，Rule 类的 end 方法签名如下：

```
public void end() throws java.lang.Exception
```

Digester 对象在前面的例子是如何做到这些的？每次调用 addObjectCreate、addCallMethod、addSetNext 方法的或者该类其它的方法的时候，都会间接的调用 Digester 类的 addRule 方法，该方法会将一个 Rule 对象以及它的匹配模式添加到 Didgester 内部的规则集合中。

addRule 方法的签名如下

```
public void addRule(java.lang.String pattern, Rule rule)
```

该方法在 Digester 类中的实现如下：

```
public void addRule(String pattern, Rule rule) {
  rule.setDigester(this);
  getRules().add(pattern, rule);
}
```

看一下 Digester 中源代码对 addObjectCreate 方法的实现：

```
public void addObjectCreate(String pattern, String className) {
  addRule(pattern, new ObjectCreateRule(className));
}
public void addObjectCreate(String pattern, Class clazz) {
  addRule(pattern, new ObjectCreateRule(clazz));
}
public void addObjectCreate(String pattern, String className,
  String attributeName) {
  addRule(pattern, new ObjectCreateRule(className, attributeName));
}
public void addObjectCreate(String pattern,
  String attributeName, Class clazz) {
```

```
  addRule(pattern, new ObjectCreateRule(attributeName, clazz));
}
```

这四个实现都调用了 addRule 方法，ObjectCreateRule 类是 Rule 类的子类。你可能会对 begin 和 end 方法在 ObjectCreateRule 类中的实现感兴趣：

```
public void begin(Attributes attributes) throws Exception {
  // Identify the name of the class to instantiate
  String realClassName = className;
  if (attributeName != null) {
    String value = attributes.getValue(attributeName);
    if (value != null) {
      realClassName = value;
    }
  }
  if (digester.log.isDebugEnabled()) {
    digester.log.debug("[ObjectCreateRule]{" + digester.match +
    "}New " + realClassName);
  }

  // Instantiate the new object and push it on the context stack
  Class clazz = digester.getClassLoader().loadclass(realClassName);
  Object instance = clazz.newInstance();
  digester.push(instance);
}

public void end() throws Exception {
  Object top = digester.pop();
  if (digester.log.isDebugEnabled()) {
    digester.log.debug("[ObjectCreateRule]{" + digester.match +
      "} Pop " + top.getdass().getName());
   }
}
```

在 begin 方法中的最后三行创建了对象并将其压到 Digester 类的内部堆栈中，end 方法使用 pop 方法从堆栈中获得对象。

Rule 类的其它子类工作方式是类似的，如果想知道它们是如何工作的可以查看它们的源代码

Digester Example 3: RuleSet 的使用

另一种往 Digester 实例添加规则方法是调用 addRuleSet 方法，该方法的签名如下：

```
public void addRuleSet(RuleSet ruleSet)
```

org.apache.commons.digester.RuleSet 表示了 Rule 对象，该接口定义了两个方法：addRuleInstance 和 getNamespaceURI。addRuleInstance 的签名如下：

public void addRuleInstance(Digester digester)

方法 addRuleInstance 将在当前 RuleSet 中定义的 Rule 对象添加到 Digester 实例中，参数就是该对象。

方法 getNamespaceURI 返回用于请求所有规则对象的名字空间 URI，它的签名如下：

public java.lang.String getNamespaceURI()

因此，在创建完一个 Digester 对象后，可以创建 RuleSet 对象并传递一个 RuleSet 对象给 addRuleSet 方法。

有一个基本类 RuleSetBase 实现了 RuleSet 接口，RuleSetBase 是一个抽象类，它提供了 getNamespaceURI 的实现，你需要做的只是提供 addRuleInstances 的实现即可。

这里修改前面例子中的 Test02 类来介绍 EmployeeRuleSet 类

Listing 15.9: The EmployeeRuleSet Class

```
package ex15.pyrmont.digestertest;

import org.apache.commons.digester.Digester;
import org.apache.commons.digester.RuleSetBase;

public class EmployeeRuleSet extends RuleSetBase  {
  public void addRuleInstances(Digester digester) {
    // add rules
    digester.addObjectCreate("employee",
      "ex15.pyrmont.digestertest.Employee");
    digester.addSetProperties("employee");
    digester.addObjectCreate("employee/office",
      "ex15.pyrmont.digestertest.Office");
    digester.addSetProperties("employee/office");
    digester.addSetNext("employee/office", "addOffice");
    digester.addObjectCreate("employee/office/address",
      "ex15.pyrmont.digestertest.Address");
    digester.addSetProperties("employee/office/address");
    digester.addSetNext("employee/office/address", "setAddress");
  }
}
```

注意 addRuleInstances 方法在 EmployeeRuleSet 中的实现跟 Test02 添加了相同的规则，如 Listing15.10 所示的 Test03 创建了一个 EmployeeRuleSet 兵将其添加到 Digester 对象上。

Listing 15.10: The Test03 Class

```
package ex15.pyrmont.digestertest;

import java.io.File;
import java.util.ArrayList;
import java.util.Iterator;
import org.apache.commons.digester.Digester;

public class Test03 {

  public static void main(String[] args) {
    String path = System.getProperty("user.dir") +
      File.separator  + "etc";
    File file = new File(path, "employee2.xml");
    Digester digester = new Digester();
    digester.addRuleSet(new EmployeeRuleSet());
    try {
      Employee employee = (Employee) digester.parse(file);
      ArrayList offices = employee.getOffices();
      Iterator iterator = offices.iterator();
      System.out.println(
        "-------------------------------------------------");
      while (iterator.hasNext()) {
        Office office = (Office) iterator.next();
        Address address = office.getAddress();
        System.out.println(office.getDescription());
        System.out.println("Address : " +
          address.getStreetNumber() + " " + address.getStreetName());
        System.out.println ("-------------------------------");
      }
    }
    catch(Exception e) {
      e.printStackTrace ();
    }
  }
}
```

运行的时候，Test03 跟 Test02 产生了相同的输出，注意 Test03 比较短，因为它将添加规则对象的操作隐藏到了 EmployeeRuleSet 类中。

接下来你会看到，Catalina 使用 RuleSetBase 的子类来初始化服务器和其他组件，在下一节里，你会看到 Digester 在 Catalina 中扮演的重要角色。

ContextConfig 类

跟其它类型的容器不同，StandardContext 必须有一个监听器，该监听器用于配置 StandardContext 对象并将 StandardContext 的 configured 变量设置为 true。在前面的章节中，使用 SimpleContextConfig 类来作为 StandardContext 的监听器。该类是一个非常简单的类，它的目的是设置 configured 变量那样 StandardContext 的 start 方法可以继续。

在一个实际的 Tomcat 部署中，StandardContext 的标准监听器是 org.apache.catalina.startup.ContextConfig 类的实例。不像简单的 SimpleContextConfig 类，ContextConfig 做了很多有用的工作。例如，ContextConfig 实例给 StandardContext 的流水线安装一个验证阀门。它还给流水线添加一个证书阀门。

更重要的是，ContextConfig 实例还读并解析默认的 web.xml 文件，并将其中的 XML 元素转换为 Java 对象。默认的 web.xml 文档在 CATALINE_HOME 下面的 conf 目录下面。它定义并映射了默认的 Servlet，并映射 MIME 类型的文件扩展，定义默认 Session 失效时间，欢迎文件列表。你可以打开该文件看看里面的内容。

应用程序的 web.xml 是文件应用配置文件，处于应用程序目录下的 WEB-INF 目录下面。这两个文件都不是必须的，ContextConfig 在找不到它们的情况下仍然可以工作。

ContextConfig 为每一个 Servlet 元素创建了一个 StandardWrapper 实例。因此，在本章的应用程序中可以看到配置是很简单的。不需要自己在完成包装器初始化任务。

因此，在你的启动类中，必须初始化 ContextConfig 类并且将其添加到 StandardContext，使用的方法是 org.apache.catalina.Lifecycle 接口的 addLifecycleListener 方法。

```
LifecycleListener listener = new ContextConfig();
((Lifecycle) context).addLifecycleListener(listener);
```

StandardContext 在它启动的时候触发以下事件：

- BEFORE_START_EVENT
- START_EVENT
- AFTER_START_EVENT

停止的时候 StandardContext 触发以下事件：

- BEFORE_STOP_EVENT
- STOP_EVENT
- AFTER_STOP_EVENT

ContextConfig 回应两个事件 START_EVENT 和 STOP_EVENT。lifecycleEvent 方法在 StandardContext 每次触发事件的时候都会被调用。该方法如 Listing15.11 所示。我们在 Listing15.11 中添加了注释以助于对 stop 方法的理解。

Listing 15.11: The lifecycleEvent method of ContextConfig

```
public void lifecycleEvent(LifecycleEvent event) {
  // Identify the context we are associated with
```

```
  try {
    context = (Context) event.getLifecycle();
    if (context instanceof StandardContext) {
      int contextDebug = ((StandardContext) context).getDebug();
      if (contextDebug > this.debug)
        this.debug = contextDebug;
    }
  }
  catch (ClassCastException e) {
    log(sm.getString("contextConfig.cce", event.getLifecycle()), e);
    return;
  }
  // Process the event that has occurred
  if (event.getType().equals(Lifecycle.START_EVENT))
    start();
  else if (event.getType().equals(Lifecycle.STOP_EVENT))
    stop();
}
```

如你看到的在 lifecycleEvent 的最后,它调用自己的 start 方法或者 stop 方法。Start 方法如 Listing15.12 所示。注意在它内部的 start 方法调用了 defaultConfig 和 applicationConfig 方法。它们会在下一节中介绍。

Listing 15.12: The start method of ContextConfig

```
private synchronized void start() {
  if (debug > 0)
    log(sm.getString("ContextConfig.start"));
  // reset the configured boolean
  context.setConfigured(false);
  // a flag that indicates whether the process is still
  // going smoothly
  ok = true;
  // Set properties based on DefaultContext
  Container container = context.getParent();
  if( !context.getOverride() ) {
    if( container instanceof Host ) {
      ((Host)container).importDefaultContext(context);
      container = container.getParent();
    }
    if( container instanceof Engine ) {
      ((Engine)container).importDefaultContext(context);
    }
  }

  // Process the default and application web.xml files
```

```
  defaultConfig();
  applicationConfig();
  if (ok) {
    validateSecurityRoles();
  }

  // Scan tag library descriptor files for additional listener classes
  if (ok) {
    try {
      tldScan();
    }
    catch (Exception e) {
      log(e.getMessage(), e);
      ok = false;
    }
  }
  // Configure a certificates exposer valve, if required
  if (ok)
    certificatesConfig();

  // Configure an authenticator if we need one
  if (ok)
    authenticatorConfig();
  // Dump the contents of this pipeline if requested
  if ((debug >= 1) && (context instanceof ContainerBase)) {
    log("Pipline Configuration:");
    Pipeline pipeline = ((ContainerBase) context).getPipeline();
    Valve valves[] = null;
    if (pipeline != null)
      valves = pipeline.getValves();
    if (valves != null) {
      for (int i = 0; i < valves.length; i++) {
        log("  " + valves[i].getInfo());
      }
    }
    log("======================");
  }
  // Make our application available if no problems were encountered
  if (ok)
    context.setConfigured(true);
  else {
    log(sm.getString("contextConfig.unavailable"));
    context.setConfigured(false);
  }
```

}

defaultConfig 方法

方法 defaultConfig 读取并解析默认的%CATALINA_HOME%/conf 目录下面的 web.xml。defaultConfig 方法如 Listing15.13 所示

Listing 15.13: The defaultConfig method

```
private void defaultConfig() {
  // Open the default web.xml file, if it exists
  File file = new File(Constants.DefaultWebXml);
  if (!file.isAbsolute())
    file = new File(System.getProperty("catalina.base"),
      Constants.DefaultWebXml);
  FileInputStream stream = null;
  try {
    stream = new FileInputStream(file.getCanonicalPath());
    stream.close();
    stream = null;
  }
  catch (FileNotFoundException e) {
    log(sm.getString("contextConfig.defaultMissing"));
    return;
  }
  catch (IOException e) {
    log(sm.getString("contextConfig.defaultMissing"), e);
    return;
  }
  // Process the default web.xml file
  synchronized (webDigester) {
    try {
      InputSource is =
        new InputSource("file://" + file.getAbsolutePath());
      stream = new FileInputStream(file);
      is.setByteStream(stream);
      webDigester.setDebug(getDebug());
      if (context instanceof StandardContext)
        ((StandardContext) context).setReplaceWelcomeFiles(true);
      webDigester.clear();
      webDigester.push(context);
      webDigester.parse(is);
   }
   catch (SAXParseException e) {
```

```
      log(sm.getString("contextConfig.defaultParse"), e);
      log(sm.getString("contextConfig.defaultPosition",
        "" + e.getLineNumber(), "" + e.getColumnNumber()));
      ok = false;
    }
    catch (Exception e) {
      log(sm.getString("contextConfig.defaultParse"), e);
      ok = false;
    }
    finally {
      try {
        if (stream != null) {
          stream.close();
        }
      }
      catch (IOException e) {
        log(sm.getString("contextConfig.defaultClose"), e);
      }
    }

  }
}
```

defaultConfig 方法首先传教一个 File 对象指向默认的 web.xml。

File file = new File(Constants.DefaultWebXml);

DefaultWebXML 的值可以在 org.apache.catalina.startup.Constants 中找到：

public static final String DefaultWebXml = "conf/web.xml";

然后方法 defaultConfig 处理 web.xml 文件。它对 webDigester 对象枷锁，然后解析该文件。

```
  synchronized (webDigester) {
    try {
      InputSource is =
        new InputSource("file://" + file.getAbsolutePath());
      stream = new FileInputStream(file);
      is.setByteStream(stream);
      webDigester.setDebug(getDebug());
      if (context instanceof StandardContext)
        ((StandardContext) context).setReplaceWelcomeFiles(true);
      webDigester.clear();
      webDigester.push(context);
      webDigester.parse(is);
```

webDigester 变量指向一个 Digester 对象的实例，该实例用于处理 web.xml 并添加规则。这些将在下面”creating Web Digester”小节中介绍到。

applicationConfig 方法

applicationConfig 方法 defaultConfig 方法相似，除了处理应用程序部署文件的地方。一个应用的部署文件在该应用目录下的 WEB-INF 目录下面

applicationConfig 方法如 Listing15.14 所示：

Listing 15.14: The applicationConfig method of ContextConfig

```
private void applicationConfig() {
  // Open the application web.xml file, if it exists
  InputStream stream = null;
  ServletContext servletContext = context.getServletContext();
  if (servletContext != null)
    stream = servletContext.getResourceAsStream
  (Constants.ApplicationWebXml);
  if (stream == null) {

    log(sm.getString("contextConfig.applicationMissing"));
    return;
   }

   // Process the application web.xml file
   synchronized (webDigester) {
     try {
       URL url =
         servletContext.getResource(Constants.ApplicationWebXml);

       InputSource is = new InputSource(url.toExternalForm());
       is.setByteStream(stream);
       webDigester.setDebug(getDebug());
       if (context instanceof StandardContext) {
         ((StandardContext) context).setReplaceWelcomeFiles(true);
       }
       webDigester.clear();
       webDigester.push(context);
       webDigester.parse(is);
     }
     catch (SAXParseException e) {
       log(sm.getString("contextConfig.applicationParse"), e);
       log(sm.getString("contextConfig.applicationPosition",
         "" + e.getLineNumber(),
         "" + e.getColumnNumber()));
```

```
      ok = false;
    }
    catch (Exception e) {
      log(sm.getString("contextConfig.applicationParse"), e);
      ok = false;
    }
    finally {
      try {
        if (stream != null) {
          stream.close();
        }
      }
      catch (IOException e) {
        log(sm.getString("contextConfig.applicationClose"),e);
      }
    }
  }
}
```

创建 web Digester

在 ContextConfig 类中存在一个名为 webDigester 的 Digester 对象：

```
private static Digester webDigester = createWebDigester();
```

该 Digester 用于解析默认 web.xml 以及应用程序 web.xml。处理 web.xml 的规则在调用 createWebDigester 方法的时候会被添加。createWebDigester 方法如 Listing15.15 所示：

Listing 15.15: The createWebDigester method

```
private static Digester createWebDigester() {
  URL url = null;
  Digester webDigester = new Digester();
  webDigester.setValidating(true);
  url =  ContextConfig.class.getResource(
    Constants.WebDtdResourcePath_22);
  webDigester.register(Constants.WebDtdPublicId_22,
    url.toString());
  url = ContextConfig.class.getResource(
    Constants.WebDtdResourcePath_23);
  webDigester.register(Constants.WebDtdPublicId_23,
  url.toString());
  webDigester.addRuleSet(new WebRuleSet());
  return (webDigester);
}
```

注意 createWebDigester 方法在 webDigester 中调用 addRuleSet 的时候传递一个 org.apache.catalina.startu. WebRuleSet p 实例. WebRuleSet 是 org.apache.commons.digester.RuleSetBase 类的一个子类。如果你熟悉 Servlet 应用程序部署文件的语法并且读过本章前面的 Digester 部分，可以很容易的理解他是如何工作的。

Listing15.16 所示的 WebRuleSet，注意里面删除了 addRuleInstance 方法的一些内容以节省空间。

Listing 15.16: The WebRuleSet class

```
package org.apache.catalina.startup;

import java.lang.reflect.Method;
import org.apache.catalina.Context;
import org.apache.catalina.Wrapper;
import org.apache.catalina.deploy.SecurityConstraint;
import org.apache.commons.digester.Digester;
import org.apache.commons.digester.Rule;
import org.apache.commons.digester.RuleSetBase;
import org.xml.sax.Attributes;

/**
* <p><strong>RuleSet</strong> for processing the contents of a web
application
* deployment descriptor (<code>/WEB-INF/web.xml</code>) resource.</p>
*
* @author Craig R. McClanahan
* @version $Revision: 1.1 $ $Date: 2001/10/17 00:44:02 $
*/

public class WebRuleSet extends RuleSetBase {
  // ----------------------------------- Instance Variables
  /**
   * The matching pattern prefix to use for recognizing our elements.
   */
  protected String prefix = null;

  // ----------------------------------------- Constructor
  /**
   * Construct an instance of this <code>RuleSet</code> with
   * the default matching pattern prefix.
   */
  public WebRuleSet () {
    this("");
  }
```

```
/**
 * Construct an instance of this <code>RuleSet</code> with
 * the specified matching pattern prefix.
 *
 * @param prefix Prefix for matching pattern rules (including the
 *  trailing slash character)
 */
public WebRuleSet(String prefix) {
  super();
  this.namespaceURI = null;
  this.prefix = prefix;
}

// ----------------------------------------- Public Methods
/**
 * <p>Add the set of Rule instances defined in this RuleSet to the
 * specified <code>Digester</code> instance, associating them with
 * our namespace URI (if any). This method should only be called
 * by a Digester instance.</p>
 *
 * @param digester Digester instance to which the new Rule instances
 * should be added.
 */
public void addRuleInstances(Digester digester) {
  digester.addRule(prefix + "web-app",
    new SetPublicIdRule(digester, "setPublicId"));
  digester.addCallMethod(prefix + "web-app/context-param",
    "addParameter", 2);
  digester.addCallParam(prefix +
    "web-app/context-param/param-name", 0);
  digester.addCallParam(prefix +
    "web-app/context-param/param-value", 1);
  digester.addCallMethod(prefix + "web-app/display-name",
    "setDisplayName", 0);
  digester.addRule(prefix + "web-app/distributable",
    new SetDistributableRule(digester));
  ...
  digester.addObjectCreate(prefix + "web-app/filter",
    "org.apache.catalina.deploy.FilterDef");
  digester.addSetNext(prefix + "web-app/filter", "addFilterDef",

    "org.apache.catalina.deploy.FilterDef");
  digester.addCallMethod(prefix + "web-app/filter/description",
    "setDescription", 0);
```

```
digester.addCallMethod(prefix + "web-app/filter/display-name",
  "setDisplayName", 0);
digester.addCallMethod(prefix + "web-app/filter/filter-class",
  "setFilterClass", 0);
digester.addCallMethod(prefix + "web-app/filter/filter-name",
  "setFilterName", 0);
digester.addCallMethod(prefix + "web-app/filter/large-icon",
  "setLargeIcon", 0);
digester.addCallMethod(prefix + "web-app/filter/small-icon",
  "setSmallIcon", 0);
digester.addCallMethod(prefix + "web-app/filter/init-param",
  "addInitParameter", 2);
digester.addCallParam(prefix +
  "web-app/filter/init-param/param-name", 0);
digester.addCallParam(prefix +
  "web-app/filter/init-param/param-value", 1);
digester.addObjectCreate(prefix + "web-app/filter-mapping",
  "org.apache.catalina.deploy.FilterMap");
digester.addSetNext(prefix + "web-app/filter-mapping",
  "addFilterMap", "org.apache.catalina.deploy.FilterMap");
digester.addCallMethod(prefix +
  "web-app/filter-mapping/filter-name", "setFilterName", 0);
digester.addCallMethod(prefix +
  "web-app/filter-mapping/servlet-name", "setServletName", 0);
digester.addCallMethod(prefix +
  "web-app/filter-mapping/url-pattern", "setURLPattern", 0);
digester.addCallMethod (prefix +
  "web-app/listener/listener-class", "addApplicationListener", 0);
...
digester.addRule(prefix + "web-app/servlet",
  new WrapperCreateRule(digester));
digester.addSetNext(prefix + "web-app/servlet",
  "addChild", "org.apache.catalina.Container");
digester.addCallMethod(prefix + "web-app/servlet/init-param",
  "addInitParameter", 2);
digester.addCallParam(prefix +
  "web-app/servlet/init-param/param-name", 0);
digester.addCallParam(prefix +
  "web-app/servlet/init-param/param-value", 1);
digester.addCallMethod(prefix + "web-app/servlet/jsp-file",
  "setJspFile", 0);
digester.addCallMethod(prefix +
  "web-app/servlet/load-on-startup", "setLoadOnStartupString", 0);
digester.addCallMethod(prefix +
```

```
      "web-app/servlet/run-as/role-name", "setRunAs", 0);
    digester.addCallMethod(prefix +
      "web-app/servlet/security-role-ref", "addSecurityReference", 2);
    digester.addCallParam(prefix +
      "web-app/servlet/security-role-ref/role-link", 1);
    digester.addCallParam(prefix +
      "web-app/servlet/security-role-ref/role-name", 0);
    digester.addCallMethod(prefix + "web-app/servlet/servlet-class",
      "setServletdass", 0);

    digester.addCallMethod(prefix + "web-app/servlet/servlet-name",
      "setName", 0);
    digester.addCallMethod(prefix + "web-app/servlet-mapping",
      "addServletMapping", 2);
    digester.addCallParam(prefix +
      "web-app/servlet-mapping/servlet-name"/ 1);
    digester.addCallParam(prefix +
      "web-app/servlet-mapping/url-pattern", 0);
    digester.addCallMethod (prefix +
      "web-app/session-config/session-timeout", "setSessionTimeout",
1,
      new Class[] { Integer.TYPE });
    digester.addCallParam(prefix +
      "web-app/session-config/session-timeout", 0);
    digester.addCallMethod(prefix + "web-app/taglib",
      "addTaglib", 2);
    digester.addCallParam(prefix + "web-app/taglib/taglib-location",
1);
    digester.addCallParam(prefix + "web-app/taglib/taglib-uri", 0);
    digester.addCallMethod(prefix +
      "web-app/welcome-file-list/welcome-file", "addWelcomeFile", 0);
  }
}
// ------------------------------------------ Private Classes

/**
* A Rule that calls the <code>setAuthConstraint(true)</code> method of
* the top item on the stack, which must be of type
* <code>org.apache.catalina.deploy.SecurityConstraint</code>.
*/
final class SetAuthConstraintRule extends Rule {
  public SetAuthConstraintRule(Digester digester) {
    super(digester);
  }
```

```
  public void begin(Attributes attributes) throws Exception {
    SecurityConstraint securityConstraint =
     (SecurityConstraint) digester.peek();
    securityConstraint.setAuthConstraint(true);
    if (digester.getDebug() > 0)
      digester.log("Calling
SecurityConstraint.setAuthConstraint(true)");
  }
}

...
final class WrapperCreateRule extends Rule {
  public WrapperCreateRule(Digester digester) {
    super(digester);
  }
  public void begin(Attributes attributes) throws Exception {
    Context context =
      (Context) digester.peek(digester.getCount() - 1);
    Wrapper wrapper = context.createWrapper();
      digester.push(wrapper);
    if (digester.getDebug() > 0)
      digester.log("new " + wrapper.getClass().getName());
  }

  public void end() throws Exception {
    Wrapper wrapper = (Wrapper) digester.pop();
    if (digester.getDebug() > 0)
      digester.log("pop " + wrapper.getclass().getName());
  }
}
```

The Application

本章的应用程序说明了如何使用 ContextConfig 实例作为监听器来配置 StandardContext 对象。它只有一个类组成，如 Listing15.17 所示的 BootStrap 类：

Listing 15.17: The Bootstrap class

```
package ex15.pyrmont.startup;

import org.apache.catalina.Connector;
import org.apache.catalina.Container;
import org.apache.catalina.Context;
import org.apache.catalina.Host;
import org.apache.catalina.Lifecycle;
```

```
import org.apache.catalina.LifecycleListener;
import org.apache.catalina.Loader;
import org.apache.catalina.connector.http.HttpConnector;
import org.apache.catalina.core.StandardContext;
import org.apache.catalina.core.StandardHost;
import org.apache.catalina.loader.WebappLoader;
import org.apache.catalina.startup.ContextConfig;

public final class Bootstrap {

  // invoke: http://localhost:8080/app1/Modern or
  // http://localhost:8080/app2/Primitive
  // note that we don't instantiate a Wrapper here,
  // ContextConfig reads the WEB-INF/classes dir and loads all
  // servlets.
  public static void main(String[] args) {
    System.setProperty("catalina.base",
      System.getProperty("user.dir"));
    Connector connector = new HttpConnector();
    Context context = new StandardContext();
    // StandardContext's start method adds a default mapper
    context.setPath("/app1");
    context.setDocBase("app1");
    LifecycleListener listener = new ContextConfig();
    ((Lifecycle) context).addLifecycleListener(listener);
    Host host = new StandardHost();
    host.addChild(context);
    host.setName("localhost");
    host.setAppBase("webapps");

    Loader loader = new WebappLoader();
    context.setLoader(loader);
    connector.setContainer(host);
    try {
      connector.initialize();
      ((Lifecycle) connector).start();
      ((Lifecycle) host).start();
      Container[] c = context.findChildren();
      int length = c.length;
      for (int i=0; i<length; i++) {
        Container child = c[i];
        System.out.println(child.getName());
      }
      // make the application wait until we press a key.
```

```
      System.in.read();
      ((Lifecycle) host) .stop();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Running the Applications

To run the application in Windows, from the working directory, type the following:

```
java -classpath ./lib/servlet.jar;./lib/commons-
collections.jar;./lib/commons-digester.jar;./lib/commons-
logging.jar;./lib/commons-beanutils.jar;./
ex15.pyrmont.startup.Bootstrap
```

In Linux, you use a colon to separate two libraries.

```
java -classpath ./lib/servlet.jar:./lib/commons-
collections.jar:./lib/commons-digester.jar:./lib/commons-
logging.jar:./lib/commons-beanutils.jar:./
ex15.pyrmont.startup.Bootstrap
```

To invoke PrimitiveServlet, use the following URL in your browser.

```
http://localhost:8080/app1/Primitive
```

To invoke ModernServlet, use the following URL.

```
http://localhost:8080/app1/Modern
```

总结

Tomcat 使用的是不同的配置，简单的配置配置使用 server.xml 文件通过 Digester 对象将 XML 元素转换为 Java 对象。另外，一个 web.xml 文档被用于配置 servlet/JSP 应用。Tomcat 必须能够解析 web.xml 文档并基于 XML 文档配置上下问对象，Digester 优雅的解决了这个问题。

第 16 章： 关闭钩子

综述

在很多环境下，在关闭应用程序的时候需要做一些清理工作。问题在于，用户并不是经常的按照要求的流程来退出。例如，在 Tomcat 部署通过初始化一个服务器并调用它的 start 方法来启动一个 servlet 容器，该方法又调用其他组件的 start 方法。正常的情况下，可以通过一个关闭命令来让服务器关闭所有组件（如 14 章中介绍）。如果突然的关闭程序，如关闭运行程序的控制台可能会发生意想不到的事情。

幸运的是，Java 提供了一种优雅的方式供程序员来使用，这样可以保证清理代码的执行。本章将会说明如何使用一个关闭钩子（shutdown hool）来保证清理代码一定会被执行。

在 Java 中，虚拟机遇到两种事件的时候会关闭虚拟机：

- 应用程序正常退出如 System.exit 方法被调用或者最后一个非守护退出。
- 用户突然强制终止虚拟机，例如键入 CTRL+C 或者在关闭 Java 程序之前从系统注销。

幸运的是，当关闭的时候，虚拟机会有以下两个步骤：

1. 虚拟机启动所有注册的关闭钩子。关闭钩子是实现在 Runtime 上面注册的线程。所有的关闭钩子会被同时执行直到完成。
2. 虚拟机调用所有的未被调用的 finalizers

在本章中，我们对第一个步骤感兴趣，它允许虚拟机提交清理代码。一个关闭钩子是 java.lang.Thread 类的子类，可以如下创建一个关闭钩子：

- 写一个类继承 Thread 类
- 提供你的实现类中的 run 方法。该方法是应用程序被关闭的时候要提交的代码，无论是正常退出还是非正常退出。
- 在你的应用程序中，初始化一个关闭钩子
- 在当前的 Runtime 上使用 addShutdownHook 方法来注册该关闭钩子。

你可能已经注意到，你并没有启动该线程。虚拟机在它的关闭步骤中会启动该线程。

Listing16.1 提供了一个简单的类名为 ShutdownHookDemo 以及一个 Thread 类的子类名为 ShutdownHook 类。注意其 run 方法仅仅会打印出一些语句在控制台上，但是，你可以在其中插入任何你想要执行的语句。

Listing 16.1: Using Shutdown Hook

```
package ex16.pyrmont.shutdownhook;
public class ShutdownHookDemo {
```

```
  public void start() {
    System.out.println("Demo");
    ShutdownHook ShutdownHook = new ShutdownHook();
    Runtime.getRuntime().addShutdownHook(ShutdownHook);
  }

  public static void main(String[] args) {
    ShutdownHookDemo demo = new ShutdownHookDemo();
    demo.start();
    try {
      System.in.read();
    }
    catch(Exception e) {
    }
  }
}

class ShutdownHook extends Thread {
  public void run() {
    System.out.println("Shutting down");
  }
}
```

在初始化一个 ShutdownHookDemo 对象之后，主方法调用 start 方法。Start 方法创建一个关闭钩子并在当前 Runtime 中注册。

```
ShutdownHook shutdownHook = new ShutdownHook();
Runtime.getRuntime().addShutdownHook(shutdownHook);
```

然后该程序等待用户键入回车键。

```
System.in.read();
```

用户键入 Enter 键后，应用程序退出。但是虚拟机会运行关闭钩子，结果就是打印出语句“Shutting down”

一个关闭钩子的例子

另一个例子，考虑一个简单的 Swing 应用程序，名为 MySwingApp。该应用程序启动的时候创建一个临时的文件，它关闭的时候，该临时文件必须被删除。

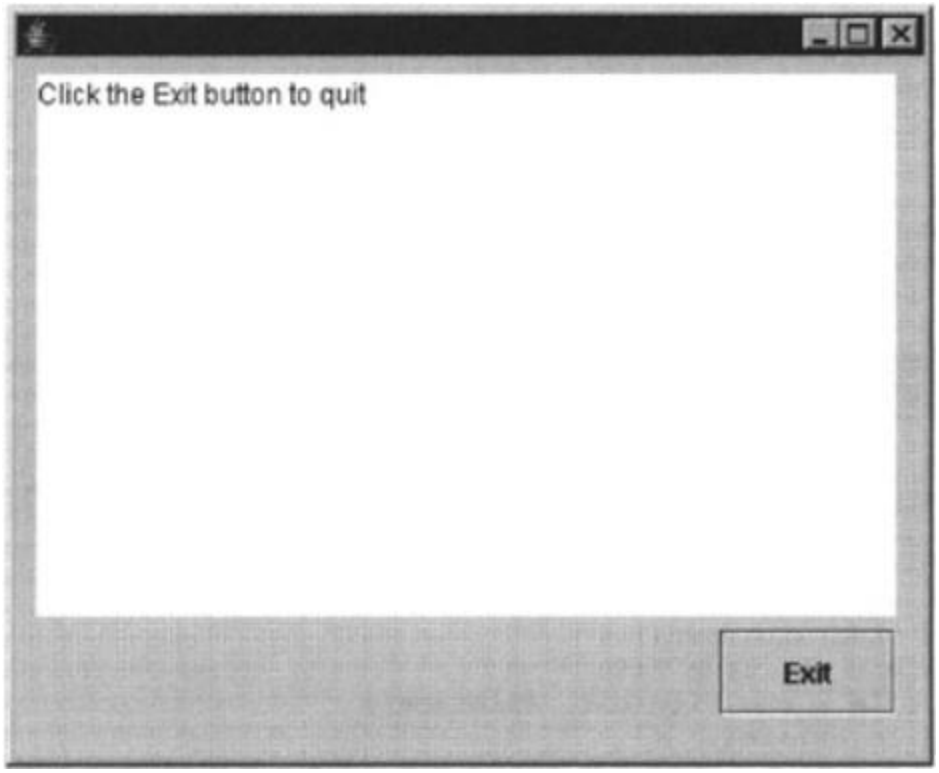


Figure 16.1: A Swing application

该类的代码如 Listing16.2 所示

Listing 16.2: A simple Swing application

```
package ex16.pyrmont.shutdownhook;
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.io.File;
import java.io.IOException;

public class MySwingApp extends JFrame {
  JButton exitButton = new JButton();
  JTextArea jTextArea1 = new JTextArea();
  String dir = System.getProperty("user.dir");
  String filename = "temp.txt";

  public MySwingApp() {
    exitButton.setText("Exit");
    exitButton.setBounds(new Rectangle(304, 248, 76, 37));
    exitButton.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        exitButton_actionPerformed(e);
      }
    });
    this.getContentPane().setLayout(null);
```

```
    jTextArea1.setText("Click the Exit button to quit");
    jTextArea1.setBounds(new Rectangle(9, 7, 371, 235));
    this.getContentPane().add(exitButton, null);
    this.getContentPane().add(jTextArea1, null);
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    this.setBounds(0,0, 400, 330);
    this.setvisible(true);
    initialize();
  }

  private void initialize() {
    // create a temp file
    File file = new File(dir, filename);
    try {
      System.out.println("Creating temporary file");
      file.createNewFile();
    }
    catch (IOException e) {
      System.out.println("Failed creating temporary file.");
    }
  }

  private void shutdown() {
    // delete the temp file
    File file = new File(dir, filename);
    if (file.exists()) {
      System.out.println("Deleting temporary file.");
      file.delete();
    }
  }

  void exitButton_actionPerformed(ActionEvent e) {
    shutdown();
    System.exit(0);
  }

  public static void main(String[] args) {
    MySwingApp mySwingApp = new MySwingApp();
  }
}
```

运行的时候，该应用程序调用它的 initialize 方法。该方法在用户工作目录下创建一个临时文件名为 temp.txt

```
  private void initialize() {
```

```
    // create a temp file
    File file = new File(dir, filename);
    try {
      System.out.println("Creating temporary file");
      file.createNewFile();
    }
    catch (IOException e) {
      System.out.println("Failed creating temporary file.");
    }
  }
```

当用户关闭该应用程序的时候，程序必须删除该临时文件。我们希望用户总是点击 Exit 按钮，这样在 shutdown 方法中就可以总是删除临时文件。但是，临时文件在用户非正常退出的时候也必须被删除。

Listing16.3 所示的类提供了一种解决方案，它提供了一个关闭钩子。该关闭钩子作为一个内部类来声明，所以它可以访问所有主类的方法。在 Listing16.3 中，关闭钩子的 run 方法调用了 shutdown 方法，保证了改方法一定会被执行。

Listing 16.3: Using a shutdown hook in the Swing application

```
package ex16.pyrmont.shutdownhook;

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.io.File;
import java.io.IOException;

public class MySwingAppWithShutdownHook extends JFrame {
  JButton exitButton = new JButton();
  JTextArea jTextArea1 = new JTextArea();

  String dir = System.getProperty("user.dir");
  String filename = "temp.txt";

  public MySwingAppWithShutdownHook() {
    exitButton.setText("Exit");
    exitButton.setBounds(new Rectangle(304, 248, 76, 37));
    exitButton.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        exitButton_actionPerformed(e);
      }
    });
    this.getContentPane().setLayout(null);
    jTextArea1.setText("Click the Exit button to quit");
    jTextArea1.setBounds(new Rectangle(9, 7, 371, 235));
```

```
    this.getContentPane().add(exitButton, null);
    this.getContentPane().add(jTextArea1, null);
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    this.setBounds(0,0, 400, 330);
    this.setVisible(true);
    initialize();
  }

  private void initialize() {
    // add shutdown hook
    MyShutdownHook shutdownHook = new MyShutdownHook();
    Runtime.getRuntime().addShutdownHook(shutdownHook);

    // create a temp file
    File file = new File(dir, filename);
    try {
      System.out.println("Creating temporary file");
      file.createNewFile();
    }
    catch (IOException e) {
      System.out.println("Failed creating temporary file.");
    }
  }

  private void shutdown() {
    // delete the temp file
    File file = new File(dir, filename);
    if (file.exists()) {
      System.out.println("Deleting temporary file.");
      file.delete();
    }
  }

  void exitButton_actionPerformed(ActionEvent e) {
    shutdown();
    System.exit(0);
  }

  public static void main(String[] args) {
    MySwingAppWithShutdownHook mySwingApp = new
MySwingAppWithShutdownHook();
  }

  private class MyShutdownHook extends Thread {
```

```
    public void run() {
      shutdown();
    }
  }
}
```

注意该类的 initialize 方法，它做的第一件事情就是创建一个 MyshutdownHook 内部类对象。

```
// add shutdown hook
MyShutdownHook shutdownHook = new MyShutdownHook();
```

一旦你获得一个该类的实例，就可以将其传递给 Runtime 的 addShutDownHook 方法，如下：

```
Runtime.getRuntime().addShutdownHook(shutdownHook);
```

Initialize 方法的其它部分就跟 Listing16.2 所示的相同了。它创建一个临时文件并打印出“Creating temporary file”

```
    // create a temp file
    File file = new File(dir, filename);
    try {
      System.out.println("Creating temporary file");
      file.createNewFile();
    }
    catch (IOException e) {
      System.out.println("Failed creating temporary file.");
    }
  }
```

接下来启动该应用程序，检查临时是否总是被删除。

Tomcat 中的关闭钩子

如你所料，Tomcat 也有自己的关闭钩子。你可以在 org.apache.catalina.startup.Catalina 类中找到它，该类负责启动服务器对象以管理其它组件。在该类中有一个内部类 CatalinaShutdownHook 继承了 java.lang.Thread 类，在该类的 run 方法中调用了服务器的 stop 方法。

Listing 16.4: Catalina shutdown hook

```
protected class CatalinaShutdownHook extends Thread {
  public void run() {
    if (server != null) {
      try {
        ((Lifecycle) server).stop();
      }
      catch (LifecycleException e) {
        System.out.println("Catalina.stop: " + e);
```

```
            e.printStackTrace(System.out);
            if (e.getThrowable() != null) {
              System.out.println("----- Root Cause -----");
              e.getThrowable().printStackTrace(System.out);
            }
          }
        }
      }
    }
```

该关闭钩子在 Catalina 实例启动的时候被初始化并添加到 Runtime 中。你可以在第 17 章学习到更多的细节

总结

有时候我们需要应用程序在关闭前进行一些清理工作。但是我们并不能保证用户总是正确的推出，本章描述的关闭钩子的方法提供了一种方法来保证清理工作的执行，无论用户是怎么停止应用程序的。

第 17 章： Tomcat 启动

综述

本章的关注重点是 Tomcat 如何使用 org.apache.catalina.startup 下面的 Catalina 类和 Bootstrap 类来启动的。Catalina 类用来启动和停止一个服务器对象并且解析 Tomcat 配置文件,即 server.xml。Bootstrap 类创建一个 Catalina 的实例并调用它的 process 方法。理论上，这两个类可以合成一个类。但是，为了支持 Tomcat 的多模式启动，提供了多个引导类。例如前述的 Bootstrap 类是将 Tomcat 作为一个独立的程序运行，而 org.apache.catalina.startup.BootstrapService 则是将 Tomcat 作为一个 Windows NT 系统的服务来运行。

为了使用方便，Tomcat 允许使用批处理文件以及 Shell 脚本来启动和停止 servlet 容器。有了这些批处理文件和 Shell 脚本的帮助，用户不需要记住 java.exe 的选项来运行 Bootstrap 类，简单的运行批处理文件或者 Shell 脚本即可。

本章的第一节讨论 Catalina 类,第二节讨论 Bootstrap 类。要理解本章的内容，首先要确保您已经读过 14、15、16 章的内容。本章还讨论了如何在 Windows 以及 Unix/Linux 下面运行 Tomcat。一节介绍 Windows 环境下的批处理文件，一节介绍 Unix/Linux 下面的 Shell 脚本。

Catalina 类

org.apache.catalina.startup.Catalina 是 Tomcat 的启动类。它包含一个用于解析%CATALINE_HOME%/conf 目录下面 server.xml 文件的 Digester。理解了如何往该 Digester 添加规则，你可以根据你的想法来配置该 Tomcat。

Catalina 类还封装了一个 Server 对象用来提供服务,如弟 15 章所介绍的那样。一个服务对象（Service）包括一个容器以及一个或多个连接器。可以使用 Catalina 来启动或停止 Server 对象。

可以首先初始化一个 Catalina 对象并调用它的 process 方法来启动 Tomcat，在调用该方法的时候必须传递合适的参数。第一个参数用来确定你是否需要用关闭命令来关闭 Tomcat。还有一些其它的参数，如-help, -config, -debug, -nonaming。

注意 Nonaming 参数出现的时候，说明不支持 JNDI 命名空间。更多的 Tomcat 对于 JNDI 命名空间的支持请看 org.apache.naming 包。

正常情况下，需要 Bootstrap 类来初始化 Catalina 对象并调用它的 process 方法，即使 Catalina 有自己的 main 方法。在下一节里将会介绍一个 Bootstrap 类，本节也会对它是如何工作的介绍。

Tomcat4 中的 process 方法如 Listing17.1 所示

Listing 17.1: The process method of the Catalina class

```
public void process(String args[]) {
```

```
    setCatalinaHome();
    setCatalinaBase();
    try {
      if (arguments(args))
        execute();
    }
    catch (Exception e) {
      e.printStackTrace(System.out);
    }
}
```

该方法设置两个系统属性 catalina.home 和 catalina.base.catalina.home，默认值是 user.dir 属性的值。Catalina.base 被设置为 catalina.home 的值。因此这两个属性的值都跟 user.dir 属性值相同。

注意 User.dir 系统属性是指当前用户工作空间，即运行 Java 命令的目录。可以在 J2SE 的 API 查看 java.lang.System 定义的系统属性列表。

然后 process 方法调用了 arguments 方法（如 Listing17.2），将参数列表传递给该方法。该方法处理参数列表，如果 Catalina 对象能继续工作返回 true。

Listing 17.2: The arguments method

```
protected boolean arguments(String args[]) {
  boolean isConfig = false;
  if (args.length < 1) {
    usage();
    return (false);
  }
  for (int i = 0; i < args.length; i++) {
    if (isConfig) {
      configFile = args[i];
      isConfig = false;
    }
    else if (args[i].equals("-config")) {
      isConfig = true;
    }
    else if (args[i].equals("-debug")) {
      debug = true;
    }
    else if (args[i].equals("-nonaming")) {
      useNaming = false;
    }
    else if (args[i].equals("-help")) {
      usage();
      return (false);
```

```
    }
    else if (args[i].equals("start")) {
      starting = true;
    }
    else if (args[i].equals("stop")) {
      stopping = true;
    }
    else {
      usage();
      return (false);
    }
  }
  return (true);
}
```

Process 方法检查 arguments 方法的返回值,如果为 true 就调用 execute 方法。该方法如 Listing17.3 所示：

Listing 17.3: The execute method

```
protected void execute() throws Exception {
  if (starting)
    start();
  else if (stopping)
    stop();
}
```

方法 execute 用于调用 start 方法或者 stop 方法来启动或者停止 Tomcat。这两个方法将在下面的子节中介绍。

注意 在 Tomcat5 中，没有 execute 方法，process 方法直接调用 start 方法和 stop 方法

start 方法

起始方法创建一个 Digester 实例来处理 server.xml 文件(Tomcat 的配置文件)。在解析 XML 文件之前，start 方法调用的 Digester 的 push 方法，传递当前的 Catalina 对象。这将导致 Catalina 对象称为在 Digester 的内部堆栈第一个对象。解析文件可以设置服务器对象的变量，服务器默认情况下是类型 org.apache.catalina.core.StandardServer。然后 start 方法调用 initialize 方法,并调用服务器对象的 start 方法。然后 Catalina 的 start 方法再调用 await 方法，服务器分配一个线程等待关机命令。该方法不返回，直到收到关机命令。当 await 方法返回,在 Catalina 中的 start 方法调用的服务器对象的 stop 方法用于停止服务器以及所有组件。Start 方法也采用了关闭钩子的方法确保在服务器对象的 stop 方法总是执行，即使用户突然退出该应用程序。

Start 方法如 Listing17.4 所示

Listing 17.4: The start method

```
protected void start() {

  // Create and execute our Digester
  Digester digester = createStartDigester();
  File file = configFile();
  try {
    InputSource is =
      new InputSource("file://" + file.getAbsolutePath());
    FileInputStream fis = new FileInputStream(file);
    is.setByteStream(fis);
    digester.push(this);
    digester.parse(is);
    fis.close();
  }
  catch (Exception e) {
    System.out.println("Catalina.start: " + e);
    e.printstackTrace(System.out);
    System.exit(1);
  }

  // Setting additional variables
  if (!useNaming) {
    System.setProperty("catalina.useNaming", "false");
  }
  else {
    System.setProperty("catalina.useNaming", "true");
    String value = "org.apache.naming";
    String oldValue =
      System.getProperty(javax.naming.Context.URL_PKG_PREFIXES);
    if (oldValue != null) {
      value = value + ":" + oldValue;
    }
    System.setProperty(javax.naming.Context.URL_PKG_PREFIXES, value);
    value = System.getProperty
      (javax.naming.Context.INITIAL_CONTEXT_FACTORY);
    if (value == null) {
      System.setProperty
        (javax.naming.Context.INITIAL_CONTEXT_FACTORY,
        "org.apache.naming.java.javaURLContextFactory");
    }
  }
```

```
// If a SecurityManager is being used, set properties for
// checkPackageAccess() and checkPackageDefinition
if( System.getSecurityManager() != null ) {
  String access = Security.getProperty("package.access");
  if( access != null && access.length() > 0 )
    access += ",";
  else
    access = "sun.,";
  Security.setProperty("package.access",
  access + "org.apache.catalina.,org.apache.jasper.");
  String definition = Security.getProperty("package.definition");
  if( definition != null && definition.length() > 0 )
    definition += ",";
  else
    definition = "sun.,";
  Security.setProperty("package.definition",

  // FIX ME package "javax." was removed to prevent HotSpot
  // fatal internal errors
  definition + "java.,org.apache.catalina.,org.apache.jasper.");
}

// Replace System.out and System.err with a custom PrintStream
SystemLogHandler log = new SystemLogHandler(System.out);
System.setOut(log);
System.setErr(log);

Thread shutdownHook = new CatalinaShutdownHook();

// Start the new server
if (server instanceof Lifecycle) {
  try {
    server.initialize();
    ((Lifecycle) server).start();
    try {
      // Register shutdown hook
      Runtime.getRuntime().addShutdownHook(shutdownHook);
    }
    catch (Throwable t) {
      // This will fail on JDK 1.2. Ignoring, as Tomcat can run
      // fine without the shutdown hook.
    }
    // Wait for the server to be told to shut down
    server.await();
```

```
    }
    catch (LifecycleException e) {
      System.out.println("Catalina.start: " + e);
      e.printStackTrace(System.out);
      if (e.getThrowable() != null) {
        System.out.println("----- Root Cause -----");
        e.getThrowable().printStackTrace(System.out);
      }
    }
  }

  // Shut down the server
  if (server instanceof Lifecycle) {
    try {
      try {
        // Remove the ShutdownHook first so that server.stop()
        // doesn't get invoked twice
        Runtime.getRuntime().removeShutdownHook(shutdownHook);
      }
      catch (Throwable t) {
        // This will fail on JDK 1.2. Ignoring, as Tomcat can run
        // fine without the shutdown hook.
      }
      ((Lifecycle) server).stop();
    }
    catch (LifecycleException e) {
      System.out.println("Catalina.stop: " + e);
      e.printStackTrace(System.out);
      if (e.getThrowable() != null) {

        System.out.println("----- Root Cause -----");
        e.getThrowable().printStackTrace(System.out);
      }
    }
  }
}
```

stop 方法

Stop 方法用于关闭 Catalina 并且关闭 Server 对象。Stop 方法如 Listing17.5 所示

Listing 17.5: The stop Method

```
protected void stop() {
  // Create and execute our Digester
  Digester digester = createStopDigester();
  File file = configFile();
  try {
    InputSource is =
      new InputSource("file://" + file.getAbsolutePath());
    FileInputStream fis = new FileInputStream(file);
    is.setByteStream(fis);
    digester.push(this);
    digester.parse(is);
    fis.close();
  }
  catch (Exception e) {
    System.out.println("Catalina.stop: " + e);
    e.printStackTrace(System.out);
    System.exit(1);
  }

  // Stop the existing server
  try {
    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();
  }
  catch (IOException e) {
    System.out.println("Catalina.stop: " + e);
    e.printStackTrace(System.out);
    System.exit(1);
  }
}
```

注意 stop 方法通过 createStopDigester 方法创建了一个 Digester 实例。把当前 Catalina 对象压入到 Digester 内部栈中，并解析其配置文件，Digester 的规则添加将在下一小节中介绍。

在收到关闭命令后，stop 方法停止服务器对象。

开始 Digester

Catalina 的 createStartDigester 用于创建 Digester 实例然后向其添加规则来解析 server.xml。该文件用于 Tomcat 配置，位于%CATALINE_HOME%/conf 目录下面。向 Digester 添加的规则是理解 Tomcat 配置的关键。

createStartDigester 方法如 Listing17.6 所示

Listing 17.6: The createStartDigester method

```
protected Digester createStartDigester() {

  // Initialize the digester
  Digester digester = new Digester();
  if (debug)
    digester.setDebug(999);
  digester.setValidating(false);

  // Configure the actions we will be using
  digester.addObjectCreate("Server",
    "org.apache.catalina.core.StandardServer", "className");
  digester.addSetProperties("Server");
  digester.addSetNext("Server", "setServer",
    "org.apache.catalina.Server");

  digester.addObjectCreate("Server/GlobalNamingResources",
    "org.apache.catalina.deploy.NamingResources");
  digester.addSetProperties("Server/GlobalNamingResources");
  digester.addSetNext("Server/GlobalNamingResources",
    "setGlobalNamingResources",
    "org.apache.catalina.deploy.NamingResources");

  digester.addObjectCreate("Server/Listener", null, "className");
  digester.addSetProperties("Server/Listener");
  digester.addSetNext("Server/Listener",
    "addLifecycleListener",
    "org.apache.catalina.LifecycleListener");

  digester.addObjectCreate("Server/Service",

    "org.apache.catalina.core.StandardService", "className");
  digester.addSetProperties("Server/Service");
  digester.addSetNext("Server/Service", "addService",
    "org.apache.catalina.Service");

  digester.addObjectCreate("Server/Service/Listener",
    null, "className");
  digester.addSetProperties("Server/Service/Listener");
  digester.addSetNext("Server/Service/Listener",
```

```
      "addLifecycleListener", "org.apache.catalina.LifecycleListener");

    digester.addObjectCreate("Server/Service/Connector",
      "org.apache.catalina.connector.http.HttpConnector",
      "className");
    digester.addSetProperties("Server/Service/Connector");
    digester.addSetNext("Server/Service/Connector",
      "addConnector", "org.apache.catalina.Connector");

    digester.addObjectCreate("Server/Service/Connector/Factory",
      "org.apache.catalina.net.DefaultServerSocketFactory",
      "className");
    digester.addSetProperties("Server/Service/Connector/Factory");
    digester.addSetNext("Server/Service/Connector/Factory",
      "setFactory", "org.apache.catalina.net.ServerSocketFactory");

    digester.addObjectCreate("Server/Service/Connector/Listener",
      null, "className");
    digester.addSetProperties("Server/Service/Connector/Listener");
    digester.addSetNext("Server/Service/Connector/Listener",
      "addLifecycleListener", "org.apache.catalina.LifecycleListener");

    // Add RuleSets for nested elements
    digester.addRuleSet(
      new NamingRuleSet("Server/GlobalNamingResources/"));
    digester.addRuleSet(new EngineRuleSet("Server/Service/"));
    digester.addRuleSet(new HostRuleSet("Server/Service/Engine/"));
    digester.addRuleSet(new
      ContextRuleSet("Server/Service/Engine/Default"));
    digester.addRuleSet(
      new NamingRuleSet("Server/Service/Engine/DefaultContext/"));
    digester.addRuleSet(
      new ContextRuleSet("Server/Service/Engine/Host/Default"));
    digester.addRuleSet(
      new NamingRuleSet("Server/Service/Engine/Host/DefaultContext/"));
    digester.addRuleSet(
      new ContextRuleSet("Server/Service/Engine/Host/"));
    digester.addRuleSet(
      new NamingRuleSet("Server/Service/Engine/Host/Context/"));
    digester.addRule("Server/Service/Engine",
      new SetParentClassLoaderRule(digester, parentClassLoader));

    return (digester);
  }
```

createStartDigester 方法创建一个 org.apache.commons.digester.Digester 类的实例，然后添加规则。

在 server.xml 中的前三个规则是针对 server 元素的，server 元素是根元素，这里是 server 模式的规则。

```
  digester.addObjectCreate("Server",
    "org.apache.catalina.core.StandardServer", "className");
  digester.addSetProperties("Server");
  digester.addSetNext("Server", "setServer",
    "org.apache.catalina.Server");
```

在遇到 server 元素的时候，Digester 要创建 org.apache.catalina.core.StandardServer 的实例。一种例外是 server 元素有一个 className 属性，它表示要初始化的类。

第二条规则适用属性的值填充 server 对象对应属性的值。

第三条规则将 Server 对象压入堆栈并将其与下一个对象（Catalina 对象）相关联，使用的方法是 setServer 方法。是怎样将一个 Catalina 的对象放入 Digester 的栈中的？在 start 方法中调用 Digester 的 push 方法来解析 server.xml 文档：

```
digester.push (this);
```

上面的代码将 Catalina 对象压入 Digester 的内部栈中

其余的规则可以根据方法中的代码得出，如果有困难，请重读 15 章。

停止 Digester

createStopDigester 方法返回一个 Digester 对象用于优雅的停止服务器对象，该方法如 Listing17.7

Listing 17.7: The stop method

```
protected Digester createStopDigester() {
  // Initialize the digester
  Digester digester = new Digester();
  if (debug)
    digester.setDebug(999);

  // Configure the rules we need for shutting down

  digester.addObjectCreate("Server",
    "org.apache.catalina.core.StandardServer", "className");
  digester.addSetProperties("Server");
  digester.addSetNext("Server", "setServer",
    "org.apache.catalina.Server");
  return (digester);
}
```

跟开始 Digester 不同，停止 Digester 仅仅对根元素感兴趣。

Bootstrap 类

org.apache.catalina.startup.Bootstrap 类提供了 Tomcat 的启动入口。当你运行 startup.bat 或者是 startup.sh 的时候，实际上运行的就是该类中的主方法。主方法创建三个类加载器并初始化 Catalina 类，然后调用 Catalina 的 process 方法。

Bootstrap 类如 Listing17.8 所示：

Listing 17.8: The Bootstrap class

```
package org.apache.catalina.startup;

import java.io.File;
import java.lang.reflect.Method;

/**
* Boostrap loader for Catalina.  This application constructs a
* class loader for use in loading the Catalina internal classes
* (by accumulating all of the JAR files found in the "server"
* directory under "catalina.home"), and starts the regular execution
* of the container.  The purpose of this roundabout approach is to
* keep the Catalina internal classes (and any other classes they
* depend on, such as an XML parser) out of the system
* class path and therefore not visible to application level classes.
*
* @author Craig R. McClanahan
* @version $Revision: 1.36 $ $Date: 2002/04/01 19:51:31 $
*/

public final class Bootstrap {
  /**
   * Debugging detail level for processing the startup.
   */
  private static int debug = 0;

  /**
   * The main program for the bootstrap.
   *
   * @param args Command line arguments to be processed
   */

  public static void main(String args[]) {

    // Set the debug flag appropriately
```

```
for (int i = 0; i < args.length; i++)  {
  if ("-debug".equals(args[i]))
    debug = 1;
}

// Configure catalina.base from catalina.home if not yet set
if (System.getProperty("catalina.base") == null)
  System.setProperty("catalina.base", getCatalinaHome());

// Construct the class loaders we will need
ClassLoader commonLoader = null;
ClassLoader catalinaLoader = null;
ClassLoader sharedLoader = null;
try {
  File unpacked[] = new File[1];
  File packed[] = new File[1];
  File packed2[] = new File[2];
  ClassLoaderFactory.setDebug(debug);

  unpacked[0] = new File(getCatalinaHome(),
    "common" + File.separator + "classes");
  packed2[0] = new File(getCatalinaHome(),
    "common" + File.separator + "endorsed");
  packed2[1] = new File(getCatalinaHome(),
    "common" + File.separator + "lib");
  commonLoader =
    ClassLoaderFactory.createClassLoader(unpacked, packed2, null);

  unpacked[0] = new File(getCatalinaHome(),
    "server" + File.separator + "classes");
  packed[0] = new File(getCatalinaHome(),
    "server" + File.separator + "lib");
  catalinaLoader =
    ClassLoaderFactory.createClassLoader(unpacked, packed,
    commonLoader);

  unpacked[0] = new File(getCatalinaBase(),
    "shared" + File.separator + "classes");
  packed[0] = new File(getCatalinaBase(),
    "shared" + File.separator + "lib");
  sharedLoader =
    ClassLoaderFactory.createClassLoader(unpacked, packed,
    commonLoader);
}
```

```
catch (Throwable t) {
  log('Class loader creation threw exception", t);
  System.exit(1);
}

Thread.currentThread().setContextClassLoader(catalinaLoader);

// Load our startup class and call its process() method
try {

  SecurityClassLoad.securityClassLoad(catalinaLoader);
  // Instantiate a startup class instance
  if (debug >= 1)
    log("Loading startup class");
  Class startupClass =
    catalinaLoader.loadClass
      ("org.apache.catalina.startup.Catalina");
  Object startupInstance = startupClass.newInstance();

  // Set the shared extensions class loader
  if (debug >= 1)
    log("Setting startup class properties");
  String methodName = "setParentClassLoader";
  Class paramTypes[] = new Class[1];
  paramTypes[0] = Class.forName("java.lang.ClassLoader");
  Object paramValues[] = new Object[1];
  paramValues[0] = sharedLoader;
  Method method =
    startupInstance.getClass().getMethod(methodName, paramTypes);
  method.invoke(startupInstance, paramValues);

  // Call the process() method
  if (debug >= 1)
    log("Calling startup class process() method");
  methodName = "process";
  paramTypes = new Class[1];
  paramTypes[0] = args.getClass();
  paramValues = new Object[1];
  paramValues[0] = args;
  method =
    startupInstance.getClass().getMethod(methodName, paramTypes);
  method.invoke(startupInstance, paramValues);
}
catch (Exception e) {
```

```
            System.out.println("Exception during startup processing");
            e.printStackTrace(System.out);
            System.exit(2);
        }
    }

    /**
     * Get the value of the catalina.home environment variable.
     */
    private static String getCatalinaHome() {
        return System.getProperty("catalina.home",
            System.getProperty("user.dir"));
    }

    /**
     * Get the value of the catalina.base environment variable.
     */
    private static String getCatalinaBase() {
        return System.getProperty("catalina.base", getCatalinaHome());
    }

    /**

     * Log a debugging detail message.
     *
     * @param message The message to be logged
     */
    private static void log(String message) {
        System.out.print("Bootstrap: ");
        System.out.println(message);
    }

    /**
     * Log a debugging detail message with an exception.
     *
     * @param message The message to be logged
     * @param exception The exception to be logged
     */
    private static void log(String message, Throwable exception) {
        log(message);
        exception.printStackTrace(System.out);
    }
}
```

Bootstrap 类有四个静态方法：两个 log 方法、getCatalinaHome 以及 getCatalinaBase 方法。getCatalinaHome 方法的实现如下：

```
  return System.getProperty("catalina.home",
    System.getProperty("user.dir"));
```

它意味着如果在前面没有提供 catalina.home 的值，它会使用 user.dir 的值。

getCatalinaBase 方法的实现如下：

```
return System.getProperty("catalina.base", getCatalinaHome());
```

它返回 catalina.base 的值，如果该值不存在返回 catalina.home 的值。

getCatalinaHome 和 getCatalinaBase 都会被 Bootstrap 类的主方法调用。

Bootstrap 类的主方法还构造了三个加载器用于不同的目的。使用不同加载器的主要原因是防止 WEB-INF/classes 以及 WEB-INF/lib 下面的类。%CATALINE_HOME%/common/lib 目录下的 jar 包也可以访问。

这三个类加载器如下定义：

```
    // Construct the class loaders we will need

    ClassLoader commonLoader = null;
    ClassLoader catalinaLoader = null;
    ClassLoader sharedLoader = null;
```

每一个类加载器都给定一个可以访问的路径。commonLoader 可以访问如下目录的类：%CATALINA_HOME%/common/classes，%CATALINA_HOME%/common/endorsed 和 %CATALINA_HOME%/common/lib。

```
   try {
     File unpacked[] = new File[1];
     File packed[] = new File[1];
     File packed2[] = new File[2];
     ClassLoaderFactory.setDebug(debug);

     unpacked[0] = new File(getCatalinaHome(),
       "common" + File.separator + "classes");
     packed2[0] = new File(getCatalinaHome(),
       "common" + File.separator + "endorsed");
     packed2[1] = new File(getCatalinaHome(),
       "common" + File.separator + "lib");
     commonLoader =
       ClassLoaderFactory.createClassLoader(unpacked, packed2, null);
```

catalinaLoader 负责加载 Catalina 容器要求的类，它可以加载%CATALINA_HOME%/server/classes 和%CATALINA_HOME%/server/lib 目录下面的类。

```
     unpacked[0] = new File(getCatalinaHome(),
       "server" + File.separator + "classes");
```

```
    packed[0] = new File(getCatalinaHome(),
      "server" + File.separator + "lib");
    catalinaLoader =
      ClassLoaderFactory.createClassLoader(unpacked, packed,
      commonLoader);
```

sharedLoader 可以访问%CATALINA_HOME%/shared/classes 和%CATALJNA_HOME%/shared/lib 目录下的类以及 commondLoader 类可以访问的类。sharedLoader 是该 Tomcat 容器相关联的所有 web 应用的类加载器的父类加载器。

```
    unpacked[0] = new File(getCatalinaBase(),
      "shared" + File.separator + "classes");
    packed[0] = new File(getCatalinaBase(),
      "shared" + File.separator + "lib");
    sharedLoader =

      ClassLoaderFactory.createClassLoader(unpacked, packed,
      commonLoader);
  }
  catch (Throwable t) {
    log('Class loader creation threw exception", t);
    System.exit(1);
  }
```

注意一点是 sharedLoader 加载器不能加载 Catalina 的内部类加载器以及环境变量下面的 CLASSPATH 下面的类，可以在第八章看到更多的加载器工作原理。

创建完三个类加载器后，主方法加载了 Catalina 类然后创建它的实例并将其赋值给 startupInstance 变量。

```
    Class startupClass =
      catalinaLoader.loadClass
        ("org.apache.catalina.startup.Catalina");
    Object startupInstance = startupClass.newInstance();
```

然后调用 setParentClassLoader 方法，将 sharedLoader 作为参数：

```
    // Set the shared extensions class loader
    if (debug >= 1)
      log("Setting startup class properties");
    String methodName = "setParentClassLoader";
    Class paramTypes[] = new Class[1];
    paramTypes[0] = Class.forName("java.lang.ClassLoader");
    Object paramValues[] = new Object[1];
    paramValues[0] = sharedLoader;
    Method method =
      startupInstance.getClass().getMethod(methodName, paramTypes);
    method.invoke(startupInstance, paramValues);
```

最后，主方法调用 Catalina 对象的 process 方法：

```
// Call the process() method
if (debug >= 1)
  log("Calling startup class process() method");
methodName = "process";
paramTypes = new Class[1];
paramTypes[0] = args.getClass();
paramValues = new Object[1];
paramValues[0] = args;
method =
  startupInstance.getClass().getMethod(methodName, paramTypes);
method.invoke(startupInstance, paramValues);
```

Windows 环境下运行 Tomcat

如在前面的小节中介绍的，可以使用 Bootstrap 类将 Tomcat 作为一个独立程序运行。在 Windows 环境下面，可以使用 starup.bat 批处理启动 Tomcat 以及 shutdown.bat 批处理文件停止 Tomcat。这两个批处理文件都能在%CATALINA_HOME%/bin 目录下找到。本节主要讨论批处理文件，对于不熟悉 DOS 命令的可以使用批处理，首先是一个小的子节：Introduction to Writing Batch Files

批处理文件简介

本节主要介绍批处理文件，这样就能理解用于启动和停止 Tomcat 的批处理文件。特殊的，它解释了如下命令：rem, if, echo, goto, label 等。它并没有全面的覆盖该问题，如果需要更多的内容可以查阅其它资源。

首先一个批处理文件必须是扩展名为.bat。可以在双击启动它也可以在命令行中调用他。一旦被调用，会从头至尾一行行的执行。在 Tomcat 的批处理文件中用到的元素会在下面介绍到。

注意 注意 DOS 命令和环境变量不区分大小写

rem

命令 rem 用作注释，rem 开头的行会被忽略不做处理。

pause

命令 pause 命令停止批处理文件处理并要求用户按键，用户按键后会继续执行处理过程。

echo

该命令将它后面的文本显示到 DOS 控制台上面。例如，下面的语句打印 Hello World 到控制台上并暂停。需要 pause 命令的原因是这样才能看到控制台上显示的信息。

```
echo Hello World
pause
```

要打印出一个环境变量的值，需要使用%将该变量括起来。例如，下面的命令打印出 myVar 的值

```
echo %myVar%.
```

要打印出操作系统的名字，可以使用如下命令：

```
cho %OS%
```

echo off

echo off 防止批处理文件中的命令被显示，只显示执行结果。但是 echo off 命令仍然会显示，要禁止 echo off 命令可以使用@echo off

@echo off

@echo off is similar to echo off, but it also suppresses the echo off command itself.

@echo off 跟 echo off 相似，但是它也禁止 echo off 命令本身

set

该命令用于设置用户定义的或者环境变量。设置的环境变量的值临时存放在内存中，在处理完成后被抛弃。

例如，下面的命令创建一个名为 THE_KING 的环境变量，值为 Elvis 并将其显示在控制台上。

```
set THE_KING=Elvis
echo %THE_KING%
pause
```

注意 要引用变量的值，使用%符号将变量名括起来。例如，echo %the_king%表示显示 THE_KING 的值

label

使用冒号来表示一个标签，你可以将标签传递给 goto 命令，让处理过程跳到该标签处。下面是一个名为 end 的标签

```
:end
```

接下来看一个 goto 命令

goto

命令 goto 强制批处理文件跳到标签定义的行，看下面的例子

```
echo Start
goto end
echo I can guarantee this line will not be executed
:end
echo End
pause
```

在第一行打印出 Start 之后，批处理文件执行 goto 命令，这样控制器跳到 end 标签。第三行被跳过。

if

if 用于测试，它有如下三种使用方式

1. To test the value of a variable.
2. To test the existence of a file
3. To test the error value.
4. 测试一个变量的值
5. 测试文件的存在性
6. 测试错误值

要测试一个变量的值，使用如下的格式

```
if variable==value nextCommand
```

例如,如下的语句测试 myVar 的值是不是 3。如果是打印出 correct 到控制台上。

```
set myVar=3
if %myVar%==3 echo Correct
```

运行上面的命令会测试 myVar 的值并打印出 Correct。

要测试一个文件是否存在，可以使用如下格式：

```
if exist c:\temp\myFile.txt goto start
```

如果在 c:\temp 目录中存在 myFile.txt 文件，控制器会跳到 start 标签。

也可以使用 not 关键字来对一个表达式取反。

not

关键字 not 用于对一个表达式取反，例如，如下命令在 myVar 的值不等于 3 的时候打印出 Correct。

```
set myVar=3
if not %myVar%==3 echo Correct
pause
```

如下命令在 c:\temp 目录中不存在 myFile.txt 文件的时候跳到 end 标签。

```
if not exist c:\temp\myFile.txt goto end
```

exist

关键字 exist 跟 if 语句连接用于测试一个文件是否存在，残酷 if 语句的例子。

Accepting Parameters 接受参数

可以给批处理文件传递参数,可以使用%1 引用第一个参数,%2 引用第二个参数。依次类推。

例如，如下命令打印第一个参数到控制台上：

```
echo %1
```

如果批处理文件的名字为 test.bat，可以使用 test Hello 命令来调用它，这样就会在控制台上显示 Hello。

下面的批处理文件检查第一个参数，如果是 start，就打印出 Starting application。如果是 stop 就打印出 Stopping application。其它情况打印出 Invalid parameter。

```
echo off
if %1==start goto start

if %1==stop goto stop
goto invalid

:start
echo Starting application
goto end

:stop
```

```
echo Stopping application
goto end

:invalid
echo Invalid parameter

:end
```

可以“%1”于空字符串比较来检查批处理文件是否有第一个参数,如果没有参数打印出No parameter。

```
if "%1"=="" echo No parameter
```

The above is the same as

上面的语句跟下面的相同

```
if ""%1""==""" echo No parameter
```

shift

命令 shift 向后移动参数，意味这%2 指向%1，%3 指向%2，依次类推。例如下面的批处理文件使用了 shift 命令。

```
echo off
shift
echo %1
echo %2
```

如果运行该命令的时候传递 3 个参数 a、b、c，会获得如下输出：

```
b
c
```

第一个参数 可以使用%0 引用，最后一个参数丢失

call

命令 call 用于调用另一个命令

setLocal

可以使用 setLocal 命令来指明对于环境变量的改变全市本地的。环境变量的值会在执行完文件或碰到 endLocal 命令后恢复。

start

要打开一个新的窗口，可以使用 start 命令，可以传递个参数作为窗口的标题：

start "Title"

另外，可以传递给该窗口要执行的命令，格式如下：

start "Title" commandName

The catalina.bat Batch File

catalina.bat 文件可以用于启动和停止 Tomcat，而 startup.bat 和 shutdown.bat 更简单的启动和停止 Tomcat。这两个批处理文件都是传递合适的参数给 catalina.bat 文件实现的。

必须在%CATALINA_HOME%下面的 bin 目录下面使用如下命令来调用 catalina.bat。

catalina *command*

或者在%CATALINA_HOME%目录下使用如下命令

bin\catalina *command*

这两种情况下，可以传递的值如下：

- debug. Start Catalina in a debugger
- debug -security. Debug Catalina with a security manager
- embedded. Start Catalina in embedded mode
- jpda start. Start Catalina under JPDA debugger
- run. Start Catalina in the current window
- run -security. Start Catalina in the current window with a security manager
- start. Start Catalina in a separate window
- start -security. Start Catalina in a separate window with security manager
- stop. Stop Catalina

例如，要在一个独立的窗口运行 Catalina，可以使用如下命令：

catalina start

Catalina.bat 如 Listing17.9 所示

Listing 17.9: The catalina.bat File

```
@echo off
if "%OS%" == "Windows_NT" setlocal
rem
---------------------------------------------------------------------
--------
rem Start/Stop Script for the CATALINA Server
rem
```

```
rem Environment Variable Prequisites
rem
rem   CATALINA_HOME   May point at your Catalina "build" directory.
rem
rem   CATALINA_BASE   (Optional) Base directory for resolving dynamic
portions
rem                   of a Catalina installation.  If not present,
resolves to
rem                   the same directory that CATALINA_HOME points to.
rem
rem   CATALINA_OPTS   (Optional) Java runtime options used when the
"start",
rem                   "stop", or "run" command is executed.
rem
rem   CATALINA_TMPDIR (Optional) Directory path location of temporary
directory
rem                   the JVM should use (java.io.tmpdir).  Defaults to
rem                   %CATALINA_BASE%\temp.
rem
rem   JAVA_HOME       Must point at your Java Development Kit
installation.
rem
rem   JAVA_OPTS       (Optional) Java runtime options used when the
"start",
rem                   "stop", or "run" command is executed.
rem
rem   JSSE_HOME       (Optional) May point at your Java Secure Sockets
Extension
rem                   (JSSE) installation, whose JAR files will be
added to the
rem                   system class path used to start Tomcat.
rem
rem   JPDA_TRANSPORT  (Optional) JPDA transport used when the "jpda
start"
rem                   command is executed. The default is "dt_shmem".
rem

rem   JPDA_ADDRESS    (Optional) Java runtime options used when the
"jpda start"
rem                   command is executed. The default is "jdbconn".
rem
rem $Id: catalina.bat,v 1.3 2002/08/04 18:19:43 patrickl Exp $
rem
---------------------------------------------------------------------
```

```
--------

rem Guess CATALINA_HOME if not defined
if not "%CATALINA_HOME%" == "" goto gotHome
set CATALINA_HOME=.
if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome
set CATALINA_HOME=..
:gotHome
if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome
echo The CATALINA_HOME environment variable is not defined correctly
echo This environment variable is needed to run this program
goto end
:okHome

rem Get standard environment variables
if exist "%CATALINA_HOME%\bin\setenv.bat" call
"%CATALINA_HOME%\bin\setenv.bat"

rem Get standard Java environment variables
if exist "%CATALINA_HOME%\bin\setclasspath.bat" goto okSetclasspath
echo Cannot find %CATALINA_HOME%\bin\setclasspath.bat
echo This file is needed to run this program
goto end
:okSetclasspath
set BASEDIR=%CATALINA_HOME%
call "%CATALINA_HOME%\bin\setclasspath.bat"

rem Add on extra jar files to CLASSPATH
if "%JSSE_HOME%" == "" goto noJsse
set
CLASSPATH=%CLASSPATH%;%JSSE_HOME%\lib\jcert.jar;%JSSE_HOME%\lib\jnet.
ja
r;%JSSE_HOME%\lib\jsse.jar
:noJsse
set CLASSPATH=%CLASSPATH%;%CATALINA_HOME%\bin\bootstrap.jar

if not "%CATALINA_BASE%" == "" goto gotBase
set CATALINA_BASE=%CATALINA_HOME%
:gotBase

if not "%CATALINA_TMPDIR%" == "" goto gotTmpdir
set CATALINA_TMPDIR=%CATALINA_BASE%\temp
:gotTmpdir
```

```
rem ----- Execute The Requested Command ---------------------------------
--------

echo Using CATALINA_BASE:   %CATALINA_BASE%
echo Using CATALINA_HOME:   %CATALINA_HOME%
echo Using CATALINA_TMPDIR: %CATALINA_TMPDIR%
echo Using JAVA_HOME:       %JAVA_HOME%

set _EXECJAVA=%_RUNJAVA%
set MAINCLASS=org.apache.catalina.startup.Bootstrap
set ACTION=start
set SECURITY_POLICY_FILE=
set DEBOG_OPTS=
set JPDA=

if not ""%1"" == ""jpda"" goto noJpda
set JPDA=jpda
if not "%JPDA_TRANSPORT%" == "" goto gotJpdaTransport
set JPDA_TRANSPORT=dt_shmem
:gotJpdaTransport
if not "%JPDA_ADDRESS%" == "" goto gotJpdaAddress
set JPDA_ADDRESS=jdbconn
:gotJpdaAddress
shift
:noJpda

if ""%1"" == ""debug"" goto doDebug
if ""%1"" == ""embedded"" goto doEmbedded
if ""%1"" == ""run"" goto doRun
if ""%1"" == ""start"" goto doStart
if ""%1"" == ""stop"" goto doStop

echo Usage:  catalina ( commands ... )
echo commands:
echo   debug             Start Catalina in a debugger
echo   debug -security   Debug Catalina with a security manager
echo   embedded          Start Catalina in embedded mode
echo   jpda start        Start Catalina under JPDA debugger
echo   run               Start Catalina in the current window
echo   run -security     Start in the current window with security
manager
echo   start             Start Catalina in a separate window
echo   start -security   Start in a separate window with security
manager
```

```
echo   stop              Stop Catalina
goto end

:doDebug
shift
set _EXECJAVA=%_RUNJDB%
set DEBUG_OPTS=-sourcepath "%CATALINA_HOME%\..\..\jakarta-tomcat-
4.0\catalina\src\share"
if not ""%1"" == ""-security"" goto execCmd
shift
echo Using Security Manager
set SECURITY_POLICY_FILE=%CATALINA_BASE%\conf\catalina.policy
goto execCmd

:doEmbedded
shift
set MAINCLASS=org.apache.catalina.startup.Embedded
goto execCmd

:doRun
shift
if not ""%1"" == ""-security"" goto execCmd
shift
echo Using Security Manager
set SECURITY_POLICY_FILE=%CATALINA_BASE%\conf\catalina.policy
goto execCmd

:doStart
shift
if not "%OS%" == "Windows_NT" goto noTitle
set _EXECJAVA=start "Tomcat" %_RUNJAVA%
goto gotTitle
:noTitle
set _EXECJAVA=start %_RUNJAVA%
:gotTitle
if not ""%1"" == ""-security"" goto execCmd
shift
echo Using Security Manager
set SECURITY_POLICY_FILE=%CATALINA_BASE%\conf\catalina.policy
goto execCmd

:doStop
shift
set ACTION=stop
```

```
goto execCmd

:execCmd
rem Get remaining unshifted command line arguments and save them in the
set CMD_LINE_ARGS=
:setArgs
if ""%1""=="""" goto doneSetArgs
set CMD_LINE_ARGS=%CMD_LINE_ARGS% %1
shift
goto setArgs
:doneSetArgs

rem Execute Java with the applicable properties
if not "%JPDA%" == "" goto dojpda
if not "%SECURITY_POLICY_FILE%" == "" goto doSecurity
%_EXECJAVA% %JAVA_OPTS% %CATALINA_OPTS% %DEBUG_OPTS% -
Djava.endorsed.dirs="%JAVA_ENDORSED_DIRS%" -classpath "%CLASSPATH%" -
Dcatalina.base="%CATALINA_BASE%" -Dcatalina.home="%CATALINA_HOME%" -
Djava.io.tmpdir="%CATALINA_TMPDIR%" %MAINCLASS% %CMD_LINE_ARGS%
%ACTION%
goto end
:doSecurity
%_EXECJAVA% %JAVA_OPTS% %CATALINA_OPTS% %DEBUG_OPTS% -
Djava.endorsed.dirs="%JAVA_ENDORSED_DIRS%" -classpath "%CLASSPATH%" -
Djava.security.manager
-Djava.security.policy=="%SECURITY_POLICY_FILE%"
-Dcatalina.base="%CATALINA_BASE%" -Dcatalina.home="%CATALINA_HOME%" -
Djava.io.tmpdir="%CATALINA_TMPDIR%" %MAINCLASS% %CMD_LINE_ARGS%
%ACTION%
goto end
:doJpda
if not "%SECURITY_POLICY_FILE%" == "" goto doSecurityJpda

%_EXECJAVA% %JAVA_OPTS% %CATALINA_OPTS% -Xdebug -
Xrunjdwp:transport=%JPDA_TRANSPORT%,address=%JPDA_ADDRESS%,server=y,s
us
pend=n %DEBUG_OPTS% -Djava.endorsed.dirs="%JAVA_ENDORSED_DIRS%" -
classpath "%CLASSPATH%" -Dcatalina.base="%CATALINA_BASE%" -
Dcatalina.home="%CATALINA_HOME%" -Djava.io.tmpdir="%CATALINA_TMPDIR%"
%MAINCLASS% %CMD_LINE_ARGS% %ACTION%
goto end
:doSecurityJpda
%_EXECJAVA% %JAVA_OPTS% %CATALINA_OPTS% -
```

```
Xrunjdwp:transport=%JPDA_TRANSPORT%,address=%JPDA_ADDRESS%,server=y,s
us
pend=n %DEBUG_OPTS% -Djava.endorsed.dirs="%JAVA_ENDORSED_DIRS%" -
classpath "%CLASSPATH%" -Djava.security.manager -
Djava.security.policy=="%SECURITY_POLICY_FILE%" -
Dcatalina.base="%CATALINA_BASE%" -Dcatalina.home="%CATALINA_HOME%" -
Djava.io.tmpdir="%CATALINA_TMPDIR%" %MAINCLASS% %CMD_LINE_ARGS%
%ACTION%
goto end
:end
```

catalina.bat 首先使用@echo off 来避免打印出命令。然后检查 OS 的环境变量的值是否是 Windows_NT。如果是调用 setLocal 来讲环境变量的改变都设置为本地的。

```
if "%OS%" == "Windows_NT" setlocal
```

然后设置 CATALINA_HOME 的值，默认情况下该变量是不存在的。

如果该变量 CATALINA_HOME 变量不存在，批处理文件认为其为批处理文件所在的目录。首先它任务 catalina.bat 文件是运行在安装目录下面下面，即 catalina.bat 所在的 bin 目录。

```
if not "%CATALINA_HOME%" == "" goto gotHome
set CATALINA_HOME=.
if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome
```

如果 catalina.bat 在其 bin 子目录下面没有找到，就无法调用 catalina.bat 文件。然后批处理文件再猜一次，它查找 CATALINA_HOME 目录下面的 bin 子目录下面是否存在该文件。

```
set CATALINA_HOME=..
:gotHome
if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome
```

如果猜测正确，它掉到 okHome。否则，打印出错误消息，告诉用户 CATALINA_HOME 设置错误并跳到 end，end 标签在该批处理文件末端。

```
echo The CATALINA_HOME environment variable is not defined correctly
echo This environment variable is needed to run this program
goto end
```

如果 CATALINA_HOME 定义正确，批处理文件调用 stenv.bat 来设置要求的环境变量，如果 setenv.bat 文件存在。如果不存在，并不会触发错误信息。

```
:okHome
rem Get standard environment variables
if exist "%CATALINA_HOME%\bin\setenv.bat" call
"%CATALINA_HOME%\bin\setenv.bat"
```

接下来，它检查 setclasspath.bat 文件是否存在，如果找不到该文件，它显示错误信息并跳到该批处理文件的 end 来推出批处理文件。

```
if exist "%CATALINA_HOME%\bin\setclasspath.bat" goto okSetclasspath
echo Cannot find %CATALINA_HOME%\bin\setclasspath.bat
echo This file is needed to run this program
goto end
```

如果找到 setclaspath.bat 文件，它设置 BASEDIR 变量的值为 CATALINA_HOME 然后调用调用 setclasspath.bat 文件来设置类路径。

```
:okSetclasspath
set BASEDIR=%CATALINA_HOME%
call "%CATALINA_HOME%\bin\setclasspath.bat"
```

setclasspath.bat 文件检查环境变量 JAVA_HOME 是否定义的正确然后使用 catalina.bat 的其它部分设置接下来的变量：

```
set JAVA_ENDORSED_DIRS=%BASEDIR%\common\endorsed
set CLASSPATH=%JAVA_HOME%\lib\tools.jar
set _RUNJAVA="%JAVA_HOME%\bin\java"
set _RUNJAVAW="%JAVA_HOME%\bin\javaw"
set _RUNJDB="%JAVA_HOME%\bin\jdb"
set _RUNJAVAC="%JAVA_HOME%\bin\javac"
```

Catalina.bat 文件检查是否安装了 Java Secure Socket Extension 以及 JSSE_HOME 的设置是否正确。如果找到了 JSSE_HOME 变量，将其添加到 CLASSPATH 变量。

```
if "%JSSE_HOME%" == "" goto noJsse
set
CLASSPATH=%CLASSPATH%;%JSSE_HOME%\lib\jcert.jar;%JSSE_HOME%\lib\jnet.
ja
r;%JSSE_HOME%\lib\jsse.jar
```

如果找不到 JSSE_HOME 变量，批处理文件继续下一行的执行，它将 bin 目录下面的 bootstrap.jar 添加到 CLASSPATH 变量中

```
:noJsse
set CLASSPATH=%CLASSPATH%;%CATALINA_HOME%\bin\bootstrap.jar
```

接下来 catalina.bat 文件检查 CATALINA_BASE。如果 CATALINA_BASE 找不到，创建它并把 CATALINA_HOME 的值赋给它。

```
if not "%CATALINA_BASE%" == "" goto gotBase
set CATALINA_BASE=%CATALINA_HOME%
:gotBase
```

接下来，它检查 CATALINA_TMPDIR 的，它表示 CATALINA_BASE 下面的 temporary 目录。

```
if not "%CATALINA_TMPDIR%" == "" goto gotTmpdir
set CATALINA_TMPDIR=%CATALINA_BASE%\temp
```

```
:gotTmpdir
```

接下来，它显示几个变量的值：

```
echo Using CATALINA_BASE:   %CATALINA_BASE%
echo Using CATALINA_HOME:   %CATALINA_HOME%
echo Using CATALINA_TMPDIR: %CATALINA_TMPDIR%
echo Using JAVA_HOME:       %JAVA_HOME%
```

然后它设置_EXECJAVA 的值为_RUNJAVA 变量的值。_RUNCJAVA 的值为%JAVA_HOME%\bin\java。换句话说，它指向 JAVA_HOME 目录下面 bin 子目录中的 java.exe。

```
set _EXECJAVA=%_RUNJAVA%
```

然后设置接下来的变量

```
set MAINCLASS=org.apache.catalina.startup.Bootstrap
set ACTION=start
set SECURITY_POLICY_FILE=
set DEBUG_OPTS=
set JPDA=
```

然后 catalina.bat 文件检查传递给它的第一个参数是否是 jpda，如果是将 JPDA 变量的值设置为 jpda。然后检查 JPDA_TRANSPORT 和 JPDA_ADDRESS 变量，并移动参数。

```
if not ""%1"" == ""jpda"" goto noJpda
set JPDA=jpda
if not "%JPDA_TRANSPORT%" == "" goto gotJpdaTransport
set JPDA_TRANSPORT=dt_shmem

:gotJpdaTransport
if not "%JPDA_ADDRESS%" == "" goto gotJpdaAddress
set JPDA_ADDRESS=jdbconn
:gotJpdaAddress
shift
```

在大多数情况下并不会使用 JPDA，因此第一个参数必须下面介个之一 debug, embedded, run, start, 或 stop

```
:noJpda
if ""%1"" == ""debug"" goto doDebug
if ""%1"" == ""embedded"" goto doEmbedded
if ""%1"" == ""run"" goto doRun
if ""%1"" == ""start"" goto doStart
if ""%1"" == ""stop"" goto doStop
```

如果第一个参数不正确或者没有参数，批处理文件显示使用说明

```
echo Usage:  catalina ( commands ... )
echo commands:
```

```
echo   debug             Start Catalina in a debugger
echo   debug -security   Debug Catalina with a security manager
echo   embedded          Start Catalina in embedded mode
echo   jpda start        Start Catalina under JPDA debugger
echo   run               Start Catalina in the current window
echo   run -security     Start in the current window with security
manager
echo   start             Start Catalina in a separate window
echo   start -security   Start in a separate window with security
manager
echo   stop              Stop Catalina
goto end
```

如果第一个参数是 start，它转到 doStart 标签，如果是 stop，将控制权转到 doStop 标签。

在 doStart 标签之后，catalina.bat 文件调用 shift 命令检查下一个参数，如果有必须为-security。否则它被忽略。下一个参数是-security，再次调用 shift 命令然后将 SECURITY_POLICY_FILE 变量值设置为%CATALINA_BASE%\conf\catalina.policy。

```
:doStart
shift
if not "%OS%" == "Windows_NT" goto noTitle
set _EXECJAVA=start "Tomcat" %_RUNJAVA%
goto gotTitle
:noTitle
set _EXECJAVA=start %_RUNJAVA%
:gotTitle
if not ""%1"" == ""-security"" goto execCmd
shift

echo Using Security Manager
set SECURITY_POLICY_FILE=%CATALINA_BASE%\conf\catalina.policy
```

在这一阶段，_EXECJAVA 的值是下面值之一：

```
start "Tomcat" "%JAVA_HOME%\bin\java"
start "%JAVA_HOME%\bin\java"
```

然后跳到 execCmd 标签中：

```
goto execCmd
```

在 execCmd 标签下面的命令获得未移动命令行参数并将其存储在 CMD_LINE_ARGS 并跳到 doneSetArgs。

```
:execCmd
set CMD_LINE_ARGS=
:setArgs
```

```
if ""%1""=="""" goto doneSetArgs
set CMD_LINE_ARGS=%CMD_LINE_ARGS% %1
shift
goto setArgs
```

下面是 doneSetArgs 标签下的命令。

```
:doneSetArgs
rem Execute Java with the applicable properties
if not "%JPDA%" == "" goto doJpda
if not "%SECURITY_POLICY_FILE%" == "" goto doSecurity
%_EXECJAVA% %JAVA_OPTS% %CATALINA_OPTS% %DEBUG_OPTS% -
Djava.endorsed.dirs="%JAVA_ENDORSED_DIRS%" -classpath "%CLASSPATH%" -
Dcatalina.base="%CATALINA_BASE%" -Dcatalina.home="%CATALINA_HOME%" -
Djava.io.tmpdir="%CATALINA_TMPDIR%" %MAINCLASS% %CMD_LINE_ARGS%
%ACTION%
```

例如在我的电脑上，可以使用如下命令来调用 catalina start：

```
start "Tomcat" "C:\j2sdk1.4.2_02\bin\java" -
Djava.endorsed.dirs="..\common\endorsed" -classpath
"C:\j2sdk1.4.2_02\lib\tools.jar;..\bin\bootstrap.jar" -
Dcatalina.base=".." -Dcatalina,home=".." -Djava.io.tmpdir="..\temp"
org.apache.catalina.startup.Bootstrap start
```

你应该明白使用不同的参数调用 catalina.bat 文件的时候命令是什么样子的。

在 Windows 下启动 Tomcat

如 Listing17.10 所示的 startup.bat 文件提供了简单方法来调用 catalina.bat 文件，它传递参数 start 来调用 catalina.bat。

Listing 17.10: The startup.bat file

```
@echo off
if "%OS%" == "Windows_NT" setlocal
rem ---------------------------------------------------------------
rem Start script for the CATALINA Server
rem
rem $Id: startup.bat,v 1.4 2002/08/04 18:19:43 patrickl Exp $
rem ---------------------------------------------------------------

rem Guess CATALINA_HOME if not defined
if not "%CATALINA_HOME%" == "" goto gotHome
set CATALINA_HOME=.
if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome
set CATALINA_HOME=..
:gotHome
if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome
```

```
echo The CATALINA_HOME environment variable is not defined correctly
echo This environment variable is needed to run this program
goto end
:okHome

set EXECUTABLE=%CATALINA_HOME%\bin\catalina.bat

rem Check that target executable exists
if exist "%EXECUTABLE%" goto okExec
echo Cannot find %EXECUTABLE%
echo This file is needed to run this program
goto end
:okExec

rem Get remaining unshifted command line arguments and save them in the
set CMD_LINE_ARGS=
:setArgs
if ""%1""=="""" goto doneSetArgs
set CMD_LINE_ARGS=%CMD_LINE_ARGS% %1
shift
goto setArgs
:doneSetArgs

call "%EXECUTABLE%" start %CMD_LINE_ARGS%

:end
Stripping all rem and echo commands, you get the following:
if "%OS%" == "Windows_NT" setlocal
if not "%CATALINA_HOME%" == "" goto gotHome
set CATALINA_HOME=.
if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome
set CATALINA_HOME=..
:gotHome

if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome
goto end
:okHome
set EXECUTABLE=%CATALINA_HOME%\bin\catalina.bat
if exist "%EXECUTABLE%" goto okExec
goto end
:okExec

rem Get remaining unshifted command line arguments and save them in the
set CMD_LINE_ARGS=
```

```
:setArgs
if ""%1""=="""" goto doneSetArgs
set CMD_LINE_ARGS=%CMD_LINE_ARGS% %1
shift
goto setArgs
:doneSetArgs

call "%EXECUTABLE%" start %CMD_LINE_ARGS%

:end
```

Windows 下面停止 Tomcat

Shutdown.bat 文件提供了一种简单方式来运行 catalina.bat，传递一个参数 stop 给它，该文件如 Listing17.11 所示

Listing 17.11: The shutdown.bat file

```
@echo off
if "%OS%" == "Windows_NT" setlocal
rem
---------------------------------------------------------------------
--------
rem Stop script for the CATALINA Server
rem
rem $Id: shutdown.bat,v 1.3 2002/08/04 18:19:43 patrickl Exp $
rem
---------------------------------------------------------------------
--------

rem Guess CATALINA_HOME if not defined
if not "%CATALINA_HOME%" == "" goto gotHome
set CATALINA_HOME=.
if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome
set CATALINA_HOME=..
:gotHome
if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome
echo The CATALINA_HOME environment variable is not defined correctly
echo This environment variable is needed to run this program
goto end
:okHome

set EXECUTABLE=%CATALINA_HOME%\bin\catalina.bat
```

```
rem Check that target executable exists
if exist "%EXECUTABLE%" goto okExec

echo Cannot find %EXECUTABLE%
echo This file is needed to run this program
goto end
:okExec

rem Get remaining unshifted command line arguments and save them in the
set CMD_LINE_ARGS=
:setArgs
if ""%1""=="""" goto doneSetArgs
set CMD_LINE_ARGS=%CMD_LINE_ARGS% %1
shift
goto setArgs
:doneSetArgs

call "%EXECUTABLE%" stop %CMD_LINE_ARGS%
:end
```

在 Unix/Linux 下面运行 Tomcat

Tomcat 自带了 Shell 脚本用来在 Unix 或者 Linux 下面启动或者停止自己。这些脚本文件扩展名为.sh，并且位于%CATALINA_HOME%目录下的子目录 bin 下面。本节将会介绍 catalina.sh, startup.sh, shutdown.sh, 和 setclasspath.sh 这四个脚本文件。

本节首先介绍 Shell 脚本，对于 Shell 脚本不熟悉的读者可以仔细阅读一下。然后涉及的内容是 catalina.sh、startip.sh 以及 shutdown.sh。setclasspath.sh 会在 catalina.sh 中使用到因此它在介绍 catalina.sh 小节中会对其进行简单的介绍。

本节对于 Shell 脚本的介绍仅限于让你能够读懂 Tomcat 的脚本，尤其是 catalina.sh、startup.sh、shutdown.sh 以及 setclasspath.sh。并不会完全的介绍 Shell 脚本，需要的话读者可以查阅其它资料。

物理角度来看，Shell 脚本是一个文本文件。可以使用 vi 或者其它文本编辑器编辑它。要确定该文件的许可模式，这样该文件可以被执行，使用的语法如下：

```
$ chmod +x scriptName
$ chmod 755 scriptName
```

这样将会给文件所有者设置读写执行权限，组以及其它的用户有该文件的执行权。

接下来可以使用如下命令来执行脚本。

```
bash scriptName
sh scriptName
./scriptName
```

接下来是一些 Shell 脚本中常用的命令，足够你理解 Tomcat 的 Shell 脚本。

comment

使用#表示后面的文本被忽略，#出现在一行的最前面的话那么这行都是注释。

```
# This is a comment
```

它也可以出现在语句的中间，这样#右边的字符都是注释

```
echo Hello # print Hello
```

clear

使用 clear 命令来清除屏幕，例如下面的语句先清理屏幕，然后打印出一个字符串

```
clear
echo Shell scripts are useful
```

exit

使用 exit 命令可以退出 Shell 脚本。退出的情况有以下几种状态，0 表示成功退出，非 0 值表示非正常退出。因此当你遇到一个问题退出，可以使用如下命令。

```
exit 1
```

echo

使用 echo 命令可以在屏幕上打印出一个字符串，例如下面的命令在控制台上打印出 Hello World。

```
echo Hello World
```

Calling A Function

可以使用句号（.）来调用一个函数或者调用其他 Shell 脚本。例如，下面命令调用同一个目录下面的 test.sh。

```
. ./test.sh
```

System and User Defined Variables

变量名必须是以数字字母或下划线开。使用等号来给变量设置值，例如，如下命令设置变量 myVar 值为 Tooksie。

```
myVar=Tootsie
```

注意在等号前后不能存在空白符，另外需要注意变量名是大小写敏感的。

可以设置变量的值为空字符串或者直接将右边留空可以将一个变量设置为 NULL。

```
myVar=
myVar=""
```

要访问一个变量的值，可以使用变量名前面加$来访问变量。例如，可以打印出变量 myVar 的值。

```
echo $myVar
```

Unix/Linux 系统提供了一些系统变量。例如 HOME 表示当前用户的 home 目录。PWD 表示用户当前目录，PATH 表示查找调用命令的路径，等等。

警告在明白更改系统变量值会带来什么后果之前应该不要改变其值。

expr

使用 expr 表示一个表达式，一个表达式必须用引号将其括起来。下面是用 Shell 脚本来表示加法运算。

```
sum=`expr 100 + 200`
echo $sum
```

它创建一个名为 sum 的变量并将其赋值为 300。运行该段脚本可以在控制台上打印出 300.

下面是另一个例子：

```
echo `expr 200 + 300`
```

它在屏幕上打印出如下内容：

```
500
```

特殊的`uname`表达式表示操作系统的名字。例如，如果你使用的是 Linux，下面的命令将会打印出 Linux 在控制台上。

```
echo `uname`
```

特殊的`dirname filePath`返回文件的目录，例如`dirname /home/user1/test.sh` 返回 /home/user1

Accessing Parameters

跟给函数传递参数一样，一个可以传递参数给 Shell 脚本。可以使用$1 来访问第一个参数$2 表示第二个参数，依次类推。$#命令获得参数个数，$@命令获得所有参数。

shift

Shift 命令将参数后移一位，$1 获得$2 的值，$2 获得$1 的值。

if ... then ... [else ...] fi

If 语句块用于测试一个条件并执行适当的命令，它的语法如下：

```
if condition then
  list of commands
[else
  list of commands
]
fi
```

注意可以使用 elif 代替 else if

例如下面的例子在获得一个 start 参数的时候打印出 Starting the application，收到 stop 的时候打印出 Stopping the application。

```
if [ "$1" = "start" ]; then
  echo Starting the application
fi
if [ "$1" = "stop" ]; then
  echo Stopping the application
fi
```

注意在条件中，在[后边必须有一个空格，而]之前必须有一个空格

$1 用双引号括起来的时候，如果没有参数传递给它不会差生异常，

$0 表示用于执行脚本的命令。例如，如果使用如下命令执行 test.sh。

./test.sh

$0 will then contain ./test.sh.

$0 就表示./test.sh

下面的表示可选条件

- *-f file*, true is *file* exists
- *-r file*, true if you have read access to *file*
- *-z string*, true if *string* is empty.
- *-n string*, true if *string* is not empty
- *string1* = *string2*, true if *string1* equals *string2*.
- *string1* != *string2*, true if *string1* is not equal to *string2*.

for Loop

For 循环的语法如下

```
for { var } in {list}
do
  list of commands
done
```

例如：

```
for i in 1 2 3
do
  echo iteration $i
done
```

打印出：

```
iteration 1
iteration 2
iteration 3
```

while Loop

While 循环的语法如下

```
while [ condition ]
do
  list of commands
done
```

例如

```
n=1
while [ $n -lt 3 ];
do
  echo iteration $n
  n=$((n + 1))
done
```

输出为

```
iteration 1
iteration 2
```

[$n -lt 3]中的-lt 表示 less than。所以它表示 n 的值小于 3.

case

Case 运行你写一个选择性执行的程序，语法如下：

```
case $variable-name in
pattern1)
  list of commands
  ;;
pattern2)
  list of commands
  ;;

*)
  list of commands
  ;;
esac
```

;;用于结束执行的命令，*)表示没有其它模式匹配的时候执行。

例如下面的脚本检查操作系统的名字。如果你使用的不是 cygwin, OS400 或 Linux，将打印出 Operating system not recogized。

```
case "`uname`" in
  CYGWIN*) echo cygwin;;
  OS400*) echo OS400;;
  Linux*) echo Linux;;
  *) echo Operating system not recognized
esac
```

Output Redirection

使用>将输出定位到文件中，例如。可以使用如下命令

```
echo Hello > myFile.txt
```

该文件创建一个名为 myFile.txt 的文件并将 Hello 写入到其中。屏幕上不会有显示。

注意 1>&2 将 stdout 上的错误信息显示到 stderr 上而 2>&1 将 stderr 的输出显示到 stdout 上。

Conditional Executions

可以写命令或条件的形式来决定执行哪个命令。这时候使用&&和||

```
command1 && command2
```

如果 command1 返回一个 0 退出状态就执行 command2。Command1 也可以使用一个条件来代替。如果条件为真，command2 将会执行，否则不执行 command2。

```
command1 || command2
```

如果 command1 的退出类型非 0，则执行 command2

command1 && command2 || command3

如果 command1 返回 0 退出状态，执行 command2，否则执行 command3

The catalina.sh File

Catalina.sh 用于在 Unix/Linux 下面启动或停止 Tomcat，启动的时候将 start 传递给 catalina.sh，关闭的时候传递 stop 参数给它。下面是可用参数：

- debug. Start Catalina in a debugger (not available on OS400)
- debug -security. Debug Catalina with a security manager (not available on OS400)
- embedded. Start Catalina in embedded mode
- jpda start. Start Catalina under JPDA debugger
- run. Start Catalina in the current window
- run -security. Start in the current window with security manager
- start. Start Catalina in a separate window
- start -security. Start in a separate window with security manager
- stop. Stop Catalina

Catalina.sh 文件如 Listing17.12 所示，基于前面已将的内容，你应该可以明白它的内容。

Listing 17.12: The catalina.sh file

```
#!/bin/sh
#
-----------------------------------------------------------------------
--------
# Start/Stop Script for the CATALINA Server
#
# Environment Variable Prequisites
#
#   CATALINA_HOME   May point at your Catalina "build" directory.
#
#   CATALINA_BASE   (Optional) Base directory for resolving dynamic
portions
#                   of a Catalina installation.  If not present,
resolves to
#                   the same directory that CATALINA_HOME points to.
```

```
#
#   CATALINA_OPTS   (Optional) Java runtime options used when the
"start",
#                   "stop", or "run" command is executed.
#
#   CATALINA_TMPDIR (Optional) Directory path location of temporary
directory
#                   the JVM should use (java.io.tmpdir).  Defaults to
#                   $CATALINA_BASE/temp.
#
#   JAVA_HOME       Must point at your Java Development Kit
installation.
#

#   JAVA_OPTS       (Optional) Java runtime options used when the
"start",
#                   "stop", or "run" command is executed.
#
#   JPDA_TRANSPORT  (Optional) JPDA transport used when the "jpda
start"
#                   command is executed. The default is "dt_socket".
#
#   JPDA_ADDRESS    (Optional) Java runtime options used when the "jpda
start"
#                   command is executed. The default is 8000.
#
#   JSSE_HOME       (Optional) May point at your Java Secure Sockets
Extension
#                   (JSSE) installation, whose JAR files will be added
to the
#                   system class path used to start Tomcat.
#
#   CATALINA_PID    (Optional) Path of the file which should contains
the pid
#                   of catalina startup Java process, when start (fork)
is used
#
# $Id: catalina.sh,v 1.8 2003/09/02 12:23:13 remm Exp $
#
-----------------------------------------------------------------------
--------

# OS specific support.  $var _must_ be set to either true or false.
cygwin=false
```

```
os400=false
case "`uname`" in
CYGWIN*) cygwin=true;;
OS400*) os400=true;;
esac

# resolve links - $0 may be a softlink
PRG="$0"

while [ -h "$PRG" ]; do
  ls=`ls -ld "$PRG"`
  link=`expr "$ls" : '.*-> \(.*\)$'`
  if expr "$link" : '.*/.*' > /dev/null; then
    PRG="$link"
  else
    PRG=`dirname "$PRG"`/"$link"
  fi
done

# Get standard environment variables
PRGDIR=`dirname "$PRG"`
CATALINA_HOME=`cd "$PRGDIR/.." ; pwd`
if [ -r "$CATALINA_HOME"/bin/setenv.sh ]; then
  . "$CATALINA_HOME"/bin/setenv.sh
fi

# For Cygwin, ensure paths are in UNIX format before anything is
touched
if $cygwin; then
  [ -n "$JAVA_HOME" ] && JAVA_HOME=`cygpath --unix "$JAVA_HOME"`
  [ -n "$CATALINA_HOME" ] && CATALINA_HOME=`cygpath --unix
"$CATALINA_HOME"`
  [ -n "$CATALINA_BASE" ] && CATALINA_BASE=`cygpath --unix
"$CATALINA_BASE"`
  [ -n "$CLASSPATH" ] && CLASSPATH=`cygpath --path --unix "$CLASSPATH"`
  [ -n "$JSSE_HOME" ] && JSSE_HOME=`cygpath --path --unix "$JSSE_HOME"`
fi

# For OS400
if $os400; then
  # Set job priority to standard for interactive (interactive - 6) by
using
  # the interactive priority - 6, the helper threads that respond to
requests
```

```
  # will be running at the same priority as interactive jobs.
  COMMAND='chgjob job('$JOBNAME') runpty(6)'
  system $COMMAND

  # Enable multi threading
  export QIBM_MULTI_THREADED=Y
fi

# Get standard Java environment variables
if [ -r "$CATALINA_HOME"/bin/setclasspath.sh ]; then
  BASEDIR="$CATALINA_HOME"
  . "$CATALINA_HOME"/bin/setclasspath.sh
else
  echo "Cannot find $CATALINA_HOME/bin/setclasspath.sh"
  echo "This file is needed to run this program"
  exit 1
fi

# Add on extra jar files to CLASSPATH
if [ -n "$JSSE_HOME" ]; then

CLASSPATH="$CLASSPATH":"$JSSE_HOME"/lib/jcert.jar:"$JSSE_HOME"/lib/jn
et
.jar:"$JSSE_HOME"/lib/jsse.jar
fi
CLASSPATH="$CLASSPATH":"$CATALINA_HOME"/bin/bootstrap.jar

if [ -z "$CATALINA_BASE" ] ; then
  CATALINA_BASE="$CATALINA_HOME"
fi

if [ -z "$CATALINA_TMPDIR" ] ; then
  # Define the java.io.tmpdir to use for Catalina
  CATALINA_TMPDIR="$CATALINA_BASE"/temp
fi

# For Cygwin, switch paths to Windows format before running java
if $cygwin; then
  JAVA_HOME=`cygpath --path --windows "$JAVA_HOME"`

  CATALINA_HOME=`cygpath --path --windows "$CATALINA_HOME"`
  CATALINA_BASE=`cygpath --path --windows "$CATALINA_BASE"`
  CATALINA_TMPDIR=`cygpath --path --windows "$CATALINA_TMPDIR"`
  CLASSPATH=`cygpath --path --windows "$CLASSPATH"`
```

```
  JSSE_HOME=`cygpath --path --windows "$JSSE_HOME"`
fi

# ----- Execute The Requested Command -----------------------------------
--------

echo "Using CATALINA_BASE:   $CATALINA_BASE"
echo "Using CATALINA_HOME:   $CATALINA_HOME"
echo "Using CATALINA_TMPDIR: $CATALINA_TMPDIR"
echo "Using JAVA_HOME:       $JAVA_HOME"

if [ "$1" = "jpda" ] ; then
  if [ -z "$JPDA_TRANSPORT" ]; then
    JPDA_TRANSPORT="dt_socket"
  fi
  if [ -z "$JPDA_ADDRESS" ]; then
    JPDA_ADDRESS="8000"
  fi
  if [ -z "$JPDA_OPTS" ]; then
    JPDA_OPTS="-Xdebug -
Xrunjdwp:transport=$JPDA_TRANSPORT,address=$JPDA_ADDRESS,server=y,sus
pe
nd=n"
  fi
  CATALINA_OPTS="$CATALINA_OPTS $JPDA_OPTS"
  shift
fi

if [ "$1" = "debug" ] ; then

  if $os400; then
    echo "Debug command not available on OS400"
    exit 1
  else
    shift
    if [ "$1" = "-security" ] ; then
      echo "Using Security Manager"
      shift
      exec "$_RUNJDB" $JAVA_OPTS $CATALINA_OPTS \
        -Djava.endorsed.dirs="$JAVA_ENDORSED_DIRS" -classpath
"$CLASSPATH" \
        -sourcepath "$CATALINA_HOME"/../../jakarta-tomcat-
4.0/catalina/src/share \
        -Djava.security.manager \
```

```
        -Djava.security.policy=="$CATALINA_BASE"/conf/catalina.policy
\
        -Dcatalina.base="$CATALINA_BASE" \
        -Dcatalina.home="$CATALINA_HOME" \
        -Djava.io.tmpdir="$CATALINA_TMPDIR" \
        org.apache.catalina.startup.Bootstrap "$@" start
    else
      exec "$_RUNJDB" $JAVA_OPTS $CATALINA_OPTS \
        -Djava.endorsed.dirs="$JAVA_ENDORSED_DIRS" -classpath
"$CLASSPATH" \

        -sourcepath "$CATALINA_HOME"/../../jakarta-tomcat-
4.0/catalina/src/share \
        -Dcatalina.base="$CATALINA_BASE" \
        -Dcatalina.home="$CATALINA_HOME" \
        -Djava.io.tmpdir="$CATALINA_TMPDIR" \
        org.apache.catalina.startup.Bootstrap "$@" start
    fi
  fi

elif [ "$1" = "embedded" ] ; then

  shift
  echo "Embedded Classpath: $CLASSPATH"
  exec "$_RUNJAVA" $JAVA_OPTS $CATALINA_OPTS \
    -Djava.endorsed.dirs="$JAVA_ENDORSED_DIRS" -classpath "$CLASSPATH"
\
    -Dcatalina.base="$CATALINA_BASE" \
    -Dcatalina.home="$CATALINA_HOME" \
    -Djava.io.tmpdir="$CATALINA_TMPDIR" \
    org.apache.catalina.startup.Embedded "$@"

elif [ "$1" = "run" ]; then

  shift
  if [ "$1" = "-security" ] ; then
    echo "Using Security Manager"
    shift
    exec "$_RUNJAVA" $JAVA_OPTS $CATALINA_OPTS \
      -Djava.endorsed.dirs="$JAVA_ENDORSED_DIRS" -classpath
"$CLASSPATH" \
      -Djava.security.manager \
      -Djava.security.policy=="$CATALINA_BASE"/conf/catalina.policy \
      -Dcatalina.base="$CATALINA_BASE" \
```

```
      -Dcatalina.home="$CATALINA_HOME" \
      -Djava.io.tmpdir="$CATALINA_TMPDIR" \
      org.apache.catalina.startup.Bootstrap "$@" start
  else
    exec "$_RUNJAVA" $JAVA_OPTS $CATALINA_OPTS \
      -Djava.endorsed.dirs="$JAVA_ENDORSED_DIRS" -classpath
"$CLASSPATH" \
      -Dcatalina.base="$CATALINA_BASE" \
      -Dcatalina.home="$CATALINA_HOME" \
      -Djava.io.tmpdir="$CATALINA_TMPDIR" \
      org.apache.catalina.startup.Bootstrap "$@" start
  fi

elif [ "$1" = "start" ] ; then

  shift
  touch "$CATALINA_BASE"/logs/catalina.out
  if [ "$1" = "-security" ] ; then
    echo "Using Security Manager"
    shift
    "$_RUNJAVA" $JAVA_OPTS $CATALINA_OPTS \
      -Djava.endorsed.dirs="$JAVA_ENDORSED_DIRS" -classpath
"$CLASSPATH" \

      -Djava.security.manager \
      -Djava.security.policy=="$CATALINA_BASE"/conf/catalina.policy \
      -Dcatalina.base="$CATALINA_BASE" \
      -Dcatalina.home="$CATALINA_HOME" \
      -Djava.io.tmpdir="$CATALINA_TMPDIR" \
      org.apache.catalina.startup.Bootstrap "$@" start \
      >> "$CATALINA_BASE"/logs/catalina.out 2>&1 &

      if [ ! -z "$CATALINA_PID" ]; then
        echo $! > $CATALINA_PID
      fi
  else
    "$_RUNJAVA" $JAVA_OPTS $CATALINA_OPTS \
      -Djava.endorsed.dirs="$JAVA_ENDORSED_DIRS" -classpath
"$CLASSPATH" \
      -Dcatalina.base="$CATALINA_BASE" \
      -Dcatalina.home="$CATALINA_HOME" \
      -Djava.io.tmpdir="$CATALINA_TMPDIR" \
      org.apache.catalina.startup.Bootstrap "$@" start \
      >> "$CATALINA_BASE"/logs/catalina.out 2>&1 &
```

```
      if [ ! -z "$CATALINA_PID" ]; then
        echo $! > $CATALINA_PID
      fi
  fi

elif [ "$1" = "stop" ] ; then

  shift
  "$_RUNJAVA" $JAVA_OPTS $CATALINA_OPTS \
    -Djava.endorsed.dirs="$JAVA_ENDORSED_DIRS" -clasapath "$CLASSPATH"
\
    -Dcatalina.base="$CATALINA_BASE" \
    -Dcatalina.home="$CATALINA_HOME" \
    -Djava.io.tmpdir="$CATALINA_TMPDIR" \
    org.apache.catalina.startup.Bootstrap "$@" stop

  if [ "$1" = "-force" ] ; then
    shift
    if [ ! -z "$CATALINA_PID" ]; then
       echo "Killing: `cat $CATALINA_PID`"
       kill -9 `cat $CATALINA_PID`
    fi
  fi

else

  echo "Usage: catalina.sh ( commands ... )"
  echo "commands:"
  if $os400; then
    echo "  debug             Start Catalina in a debugger (not
available on OS400)"
    echo "  debug -security   Debug Catalina with a security manager
(not available on OS400)"
  else
    echo "  debug             Start Catalina in a debugger"

    echo "  debug -security   Debug Catalina with a security manager"
  fi
  echo "  embedded          Start Catalina in embedded mode"
  echo "  jpda start        Start Catalina under JPDA debugger"
  echo "  run               Start Catalina in the current window"
  echo "  run -security     Start in the current window with security
manager"
```

```
  echo "  start             Start Catalina in a separate window"
  echo "  start -security   Start in a separate window with security
manager"
  echo "  stop              Stop Catalina"
  exit 1
fi
```

Starting Tomcat on Linux/Unix

简单起见，可以使用 startup.sh 来启动 Tomcat，startup.sh 设置了正确的环境变量调用 catalina.sh。startup.sh 如 Listing17.13 所示

Listing 17.13: The startup.sh file

```
#!/bin/sh
#
-----------------------------------------------------------------------
-
--------
# Start Script for the CATALINA Server
#
# $Id: startup.sh,v 1.3 2002/08/04 18:19:43 patrickl Exp $
#
-----------------------------------------------------------------------
--------

# resolve links - $0 may be a softlink
PRG="$0"

while [ -h "$PRG" ] ; do
  ls=`ls -ld "$PRG"`
  link=`expr "$ls" : '.*-> \(.*\)$'`
  if expr "$link" : '.*/.*' > /dev/null; then
    PRG="$link"
  else
    PRG=`dirname "$PRG"`/"$link"
  fi
done

PRGDIR=`dirname "$PRG"`
EXECUTABLE=catalina.sh

# Check that target executable exists
if [ ! -x "$PRGDIR"/"$EXECUTABLE" ]; then
```

```
  echo "Cannot find $PRGDIR/$EXECUTABLE"
  echo "This file is needed to run this program"
  exit 1
fi

exec "$PRGDIR"/"$EXECUTABLE" start "$@"
```

Stopping Tomcat on Linux/Unix

可以使用 shutdown.sh 脚本来简单的关闭 Tomcat，该脚本传递 stop 作为参数给 catalina.sh。

Listing 17.14 presents shutdown.sh.

Listing 17.14: The shutdown.sh File

```
#!/bin/sh
#
-----------------------------------------------------------------------------
# Stop script for the CATALINA Server
#
# $Id: shutdown.sh,v 1.3 2002/08/04 18:19:43 patrickl Exp $
#
-----------------------------------------------------------------------------
# resolve links - $0 may be a softlink
PRG="$0"

while [ -h "$PRG" ] ; do
  ls=`ls -ld "$PRG"`
  link=`expr "$ls" : '.*-> \(.*\)$'`
  if expr "$link" : '.*/.*' > /dev/null; then
    PRG="$link"
  else
    PRG=`dirname "$PRG"`/"$link"
  fi
done
PRGDIR=`dirname "$PRG"`
EXECUTABLE=catalina.sh

# Check that target executable exists
if [ ! -x "$PRGDIR"/"$EXECUTABLE" ]; then
  echo "Cannot find $PRGDIR/$EXECUTABLE"
  echo "This file is needed to run this program"
  exit 1
fi
```

```
exec "$PRGDIR"/"$EXECUTABLE" stop "S@"
```

总结

本章介绍了用于启动应用程序的两个类，Catalina 和 Bootstrap，它们都是 org.apache.catalina.startup 包下面的成员。另外还学习了批处理文件和 Shell 脚本来启动和停止 Tomcat。

第 18 章：部署器

综述

要使得一个 web 应用可以访问，一个上下文必须先部署在主机上。在 Tomcat 中，一个上下文可以以 WAR 文件的形式部署，也可以直接将整个应用程序部署在 Tomcat 安装目录的 wabapp 目录下面。对你部署的每个应用，都可以有一个配置脚本用来配置该上下文，配置脚本以 XML 文档的形式存在。

注意 在 Tomcat4 和 5 中有两个已经部署好的应用：manager 和 admin。它们的类文件都在%CATALINA_HOME%/server/webapps 目录下面。这两个都有配置脚本，分布式 manager.xml 和 admin.xml。在 Tomcat4 中，配置脚本在%CATALINA_HOME%/webapps 下面，而 Tomcat5 中，它们在相应的应用目录下面，即%CATALINA_HOME%/server/webapps/admin 以及%CATALINA_HOME%/server/webapps/manager

本章介绍的内容是使用部署器来部署一个 web 应用，部署器用 org.apache.catalina.Deployer 接口表示。部署器跟主机相关联，用于安装子容器。往主机安装上下文意味着创建 StandardContext 类的实例并将其添加到主机（host）。当主机启动的时候，子上下文也启动（父容器的 start 方法总是启动子容器的 start 方法，包装器除外）。但是使用部署器可以独立开始和停止单独的上下文。

部署一个 web 上下文

在第 15 章中，使用如下代码初始化一个 StandardHost 并将上下文对象作为子容器添加到上面。

```
Context context = new StandardContext();
context.setPath("/app1");
context.setDocBase("app1");
LifecycleListener listener = new ContextConfig();
((Lifecycle) context).addLifecycleListener(listener);

Host host = new StandardHost();
host.addChild(context);
```

这是我们部署应用程序的方法，但是 Tomcat 中并没有这些代码。那么，在一个实际部署中，上下文是如何添加到主机的呢？答案 StandardHost 实例中 org.apache.catalina.startup.HostConfig 类型的生命周期监听器。

StandardHost 实例的 start 方法启动的时候，它触发 START 事件。HostConfig 的响应是它会调用它自己的 start 方法，它会部署和安装所有的特定目录下面的 web 应用程序。下面是具体的细节。

回忆第 15 章中的内容，它解释了如何使用 Digester 来解析一个 XML 文件。但是那章并没有讨论 Digester 对象的所有规则。它掠过的一个主题就是部署器，这也正是本章要介绍的内容。

org.apache.catalina.startup.Catalina 是一个启动类，它使用一个 Digester 对象将 server.xml 文档中的 XML 元素转换为 Java 对象。Catalina 类定义了 createStartDigester 方法用于往 Digester 对象添加规则。下面是该方法中的一行

```
digester.addRuleSet(new HostRuleSet("Server/Service/Engine/"));
```

org.apache.catalina.startup.HostRuleSet 类继承了
org.apache.commons.digester.RuleSetBase 类。作为一个 RuleSetBase 类的子类,HostRuleSet 类提供了 addRuleInstances 方法的实现,该方法用于为 RuleSet 定义规则。下面是 HostRuleSet 类的 addRuleInstances 方法的一个片段

```
public void addRuleInstances(Digester digester) {
  digester.addObjectCreate(prefix + "Host",

    "org.apache.catalina.core.StandardHost", "className");
  digester.addSetProperties(prefix + "Host");
  digester.addRule(prefix + "Host",
    new CopyParentClassLoaderRule(digester));
  digester.addRule(prefix + "Host",
    new LifecycleListenerRule (digester,
    "org.apache.catalina.startup.HostConfig", "hostConfigClass"));
```

这段代码的意思是，当 Server/Service/Engine/Host 的模式的时候创建一个 org.apache.catalina.startup.HostConfig 类的对象，并将其作为一个生命周期监听器添加到主机上。换句话说，HostConfig 处理 StandardHost 的 start 方法和 stop 方法触发的事件。

HostConfig 类的 lifecycleEvent 方法如 Listing18.1 所示。该方法用于处理事件，因为 HostConfig 是 StandardHost 实例的监听器，每次调用 StandardHost 启动或停止的时候，都会触发 lifecycleEvent 方法。

Listing 18.1: The lifecycleEvent method of the HostConfig class.

```
 public void lifecycleEvent(LifecycleEvent event) {
  // Identify the host we are associated with
  try {
    host = (Host) event.getLifecycle();
    if (host instanceof StandardHost) {
      int hostDebug = ((StandardHost) host).getDebug();
      if (hostDebug > this.debug) {
        this.debug = hostDebug;
      }
      setDeployXML(((StandardHost) host).isDeployXML());
      setLiveDeploy(((StandardHost) host).getLiveDeploy());
      setUnpackWARs(((StandardHost) host).isUnpackWARs());
```

```
    }
  }
  catch (ClassCastException e) {
    log(sm.getString("hostConfig.cce", event.getLifecycle()), e);
    return;
  }

  // Process the event that has occurred
  if (event.getType().equals(Lifecycle.START_EVENT))
    start ();
  else if (event.getType().equals(Lifecycle.STOP_EVENT))
    stop();
}
```

如果主机是 org.apache.catalina.core.StandardHost 的一个实例，将会调用 setDeployXML，setLiveDeploy，setUnpackWARs 方法。

```
    setDeployXML(((StandardHost) host).isDeployXML());
    setLiveDeploy(((StandardHost) host).getLiveDeploy());

    setUnpackWARs(((StandardHost) host).isUnpackWARs());
```

StandardHost 的 isDeployXML 方法标志该主机是否部署一个上下文部署文件。deployXML 属性的默认值是 true。liveDeploy 属性的值标志是否需要周期性检查新部署，unpackWARs 属性定义了是否需要解压来部署 WAR 文件。

根据接收到的 START 事件，HostConfig 对象的 lifecycleEvent 方法调用 start 方法来部署应用程序，该方法如 Listing18.2

Listing 18.2: The start method of the HostConfig class

```
protected void start() {
  if (debug >= 1)
    log(sm.getString("hostConfig.start"));
  if (host.getAutoDeploy()) {
    deployApps();
  }
  if (isLiveDeploy ()) {
    threadStart();
  }
}
```

如果 autoDeploy 属性为真，start 方法调用 deployApps 方法。另外如果 liveDeploy 为真他还调用 threadStart 方法启动一个新线程。Live deploy 将在“live deploy”一节介绍。

deployApps 方法获得主机的 appBase 属性，appBase 默认的有一个值为 webapps。部署过程任务所有的位于%CATALINE_HOME%/webapps 目录下面的子目录为一个应用程序。另外在该目录下面的 WAR 文件和描述文件会被部署。

deployApps 方法如 Listing18.3

Listing 18.3: The deployApps method

```
protected void deployApps() {
  if (!(host instanceof Deployer))
    return;
  if (debug >= 1)
    log(sm.getString("hostConfig.deploying"));
  File appBase = appBase();
  if (!appBase.exists() || !appBase.isDirectory())

    return;
  String files[] = appBase.list();
  deployDescriptors(appBase, files);
  deployWARs(appBase, files);
  deployDirectories(appBase, files);
}
```

deployApps 方法调用了其它三个方法 deployDescriptors, deployWARs, 和 deployDirectories。所有的方法中，deployApps 传递 appBase 文件以及 webapps 目录下的文件数组。一个上下文通过它的路径来鉴别，所有的上下文都有唯一的路径。部署的上下文被添加到 HostConfig 对象的 deployed ArraList 上面。因此在部署一个上下文的时候， deployDescriptors, deployWARs, 和 deployDirectories 要确定在 deployed ArrayList 上面不包含该路径。

接下来看着三个部署方法，看完下面的三个小节你应该能回答下面的问题：这三个方法的调用顺序重要没（答案当然是 yes）

Deploying a Descriptor

可以通过编写一个 XML 文件来描述一个上下文对象。例如，在 Tomcat4 和 5 中的 admin 何 manager 应用有 Listing18.4 和 Listing18.5 部署文件

Listing 18.4: The descriptor for the admin application (admin.xml)

```
 <Context path="/admin" docBase="../server/webapps/admin"
  debug="0" privileged="true">
  <!-- Uncomment this Valve to limit access to the Admin app to
   localhost for obvious security reasons. Allow may be a comma-
   separated list of hosts (or even regular expressions).
  <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>
```

Listing 18.5: The descriptor for the manager application (manager.xml)

```
 <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>
```

注意这两个描述文件都有 Context 元素和 docBase 属性指向%CATALINA_HOME%/server/webapps/admin 和%CATALINA_HOME%/server/webapps/manager，这说明 admin 和 manager 应用没有部署在普通地点。

HostConfig 类使用如 Listing18.6 所示的 deployDecriptors 方法来部署所有的%CATALINA_HOME%/webapps 下面（Tomcat4）或者%CATALINA_HOME%/server/webapps/ （Tomcat5）下面的 XML 文件。

Listing 18.6: The deployDescriptors method in HostConfig

```
protected void deployDescriptors(File appBase, String[] files) {
  if (!deployXML)
    return;

  for (int i = 0; i < files.length; i++) {
    if (files[i].equalsIgnoreCase("META-INF"))
      continue;
    if (files[i].equalsIgnoreCase("WEB-INF"))
      continue;
    if (deployed.contains(files[i]))
      continue;
    File dir = new File(appBase, files[i]);
    if (files[i].toLowerCase().endsWith(".xml")) {
      deployed.add(files[i]);

      // Calculate the context path and make sure it is unique
      String file = files[i].substring(0, files[i].length() - 4);
      String contextPath = "/" + file;
      if (file.equals("ROOT")) {
        contextPath = "";
      }
      if (host.findChild(contextPath) != null) {
```

```
        continue;
      }

      // Assume this is a configuration descriptor and deploy it
      log(sm.getString("hostConfig.deployDescriptor", files[i]));
      try {
        URL config =
          new URL("file", null, dir.getCanonicalPath());
          ((Deployer) host).install(config, null);
      }
      catch (Throwable t) {
        log(sm.getString("hostConfig.deployDescriptor.error",
          files[i]), t);
      }
    }
  }
}
```

Deploying a WAR File

可以部署 war 文件形式的 web 应用。HostConfig 使用如 Listing18.7 所示的 deployWARs 方法来部署%CATALINA_HOME%/webapps 目录下的 WAR 文件。

Listing 18.7: The deployWARs method in HostConfig

```
protected void deployWARs(File appBase, String[] files) {
  for (int i = 0; i < files.length; i++) {
    if (files[i].equalsIgnoreCase("META-INF"))
      continue;
    if (files[i].equalsIgnoreCase("WEB-INF"))
      continue;
    if (deployed.contains(files [i]))
      continue;
    File dir = new File(appBase, files [i]);

    if (files[i].toLowerCase().endsWith(".war")) {
      deployed.add(files [i]);
      // Calculate the context path and make sure it is unique
      String contextPath = "/" + files[i];
      int period = contextPath.lastIndexOf(".");
      if (period >= 0)
      contextPath = contextPath.substring(0, period);
      if (contextPath.equals("/ROOT"))
        contextPath = "";
      if (host.findChild(contextPath) != null)
        continue;
```

```
      if (isUnpackWARs()) {
        // Expand and deploy this application as a directory
        log(sm.getString("hostConfig.expand", files[i]));
        try {
          URL url = new URL("jar:file:" +
            dir.getCanonicalPath() + "!/");
          String path = expand(url);
          url = new URL("file:" + path);
          ((Deployer) host).install(contextPath, url);
        }
        catch (Throwable t) {
          log(sm.getString("hostConfig.expand.error", files[i]), t);
        }
      }
      else {
        // Deploy the application in this WAR file
        log(sm.getString("hostConfig.deployJar", files[i]));
        try {
          URL url = new URL("file", null,
             dir.getCanonicalPath());
          url = new URL("jar:" + url.toString() + "!/");
          ((Deployer) host).install(contextPath, url);
        }
        catch (Throwable t) {
          log(sm.getString("hostConfig.deployJar.error",

          files[i]), t);
        }
      }
    }
  }
}
```

Deploying a Directory

另外也可以将整个目录拷贝到%CATALINA_HOME%/webapps 目录下来部署一个应用。HostConfig 使用如 Listing18.8 所示的 deployDirectories 来部署目录：

Listing 18.8: The deployDirectories method in HostConfig

```
protected void deployDirectories (File appBase, string[] files){
  for (int i = 0; i < files.length; i++) {
    if (files[i].equalsIgnoreCase("META-INF"))
      continue;
    if (files[i].equalsIgnoreCase("WEB-INF"))
      continue;
    if (deployed.contains(files[i]))
```

```
      continue;
    File dir = new File(appBase, files[i]);
    if (dir.isDirectory()) {
      deployed.add(files[i]);

      // Make sure there is an application configuration directory
      // This is needed if the Context appBase is the same as the
      // web server document root to make sure only web applications
      // are deployed and not directories for web space.
      File webInf = new File(dir, "/WEB-INF");
      if (!webInf.exists() || !webInf.isDirectory() ||
        !webInf.canRead())
        continue;

      // Calculate the context path and make sure it is unique
      String contextPath = "/" + files[i];
      if (files[i].equals("ROOT"))
        contextPath = "";
      if (host.findChild(contextPath) != null)
        continue;

      // Deploy the application in this directory
      log(sm.getString("hostConfig.deployDir", files[i]));
      try {
        URL url = new URL("file", null, dir.getCanonicalPath());
        ((Deployer) host).install(contextPath, url);
      }
      catch (Throwable t) {
        log(sm.getString("hostConfig.deployDir.error", files[i]), t);
      }
    }
  }
}
```

Live Deploy

如前面提到的StandardHost实例使用HostConfig对象作为一个生命周期监听器。当 StandardHost 对象开始的时候，它的 start 方法触发一个 START 事件。作为该事件的响应，HostConfig 中的 lifecycleEvent 方法作为它的事件处理器，调用它的 start 方法。在 Tomcat4 中，start 方法的最后一行如果 liveDeploy 属性为真的话（默认为真）调用 threadStart 方法。

```
  if (isLiveDeploy()) {
```

```
    threadStart();
  }
```

threadStart 分配一个新线程并调用它的 run 方法，run 方法周期性的检查在 web.xml 文件中的已存在部署是否有改变。该方法如 Listing18.9 所示：

Listing 18.9: The run method in HostConfig in Tomcat 4

```
/**
* The background thread that checks for web application autoDeploy
* and changes to the web.xml config.
*/
public void run() {
  if (debug >= 1)
    log("BACKGROUND THREAD Starting");
  // Loop until the termination semaphore is set
  while (!threadDone) {
    // Wait for our check interval
    threadSleep();
    // Deploy apps if the Host allows auto deploying
    deployApps();
    // Check for web.xml modification
    checkWebXmlLastModified();
  }
  if (debug >= 1)
    log("BACKGROUND THREAD Stopping");
}
```

threadSleep 方法让线程休眠 checkInterval 属性定义的时间，它的默认值是 15，这意味着检查没 15 秒进行一次。

在 Tomcat5 中，HostConfig 没有独立的线程而是使用 backgroundProcess 方法来周期性的进行检查事件。

```
public void backgroundProcess() {
  lifecycle.fireLifecycleEvent("check", null);
}
```

注意 backgroundProcess 会被周期性的调用，工作由一个特殊的线程来处理容器中的后台处理。

在收到一个”check”事件后，生命周期对象 HostConfig 对象调用它的 check 方法进行检查工作：

```
public void lifecycleEvent(LifecycleEvent event) {
  if (event.getType().equals("check"))
    check();
  ...
```

Tomcat5 中的 HostConfig 的 check 方法如 Listing18.10

Listing 18.10: The check method in HostConfig in Tomcat 5

```
protected void check() {
  if (host.getAutoDeploy()) {
    // Deploy apps if the Host allows auto deploying
    deployApps();
    // Check for web.xml modification
    checkContextLastModified();
  }
}
```

在 check 方法中调用了 deployApps 方法，deployApps 方法在 Tomcat4 和 5 中都是部署一个 web 应用程序，如 Listing18.3 所示。如前面所讨论的该方法调用 eployDescriptors, deployWARs, 和 deployDirectories。

Tomcat5 中的 check 方法调用了 checkCOntextLastModified 方法，迭代所有的部署上下文并检查 web.xml 以及每个上下文 WEB-INF 目录下面内容的时间戳。如果检查到改变，就重启该上下文。另外，checkContextLastModified 方法还检查部署的 WAR 文件的时间戳，如果有改变就进行改变。

在 Tomcat4 中，后台线程的 run 方法调用 checkWebXmlLastModified 跟 Tomcat5 中的 checkContextLastModified 方法完成相同任务。

Deployer 接口

一个部署器由 org.apache.catalina.Deployer 接口表示。StandardHost 类实现了 Deployer 接口，以你次，一个 StandardHost 实例除了是一个容器外也是一个部署器。Deployer 接口如 Listing18.11 所示

Listing 18.11: The Deployer interface

```
package org.apache.catalina;

import java.io.IOException;
import java.net.URL;

/**
* A <b>Deployer</b> is a specialized Container into which web
* applications can be deployed and undeployed. Such a Container
* will create and install child Context instances for each deployed
* application. The unique key for each web application will be the
* context path to which it is attached.
*
* @author Craig R. McClanahan
* @version $Revision: 1.6 $ $Date: 2002/04/09 23:48:21 $
*/
```

```
public interface Deployer  {
  /**
   * The ContainerEvent event type sent when a new application is
   * being installed by <code>install()</code>, before it has been
   * started.
   */
  public static final String PRE_INSTALL_EVENT = "pre-install";

  /**
   * The ContainerEvent event type sent when a new application is
   * installed by <code>install()</code>, after it has been started.
   */
  public static final String INSTALL_EVENT = "install";

  /**
   * The ContainerEvent event type sent when an existing application is
   * removed by <code>remove()</code>.
   */
  public static final String REMOVE_EVENT = "remove";

  /**

   * Return the name of the Container with which this Deployer is
   * associated.
   */
  public String getName();
  /**
   * Install a new web application, whose web application archive is at
   * the specified URL, into this container with the specified context.
    * path. A context path of "" (the empty string) should be used for
   * the root application for this container.  Otherwise, the context
   * path must start with a slash.
   * <p>
   * If this application is successfully installed, a ContainerEvent of
   * type <code>INSTALL_EVENT</code> will be sent to all registered
   * listeners,
   * with the newly created <code>Context</code> as an argument.
   *
   * @param contextPath The context path to which this application
   *  should be installed (must be unique)
   * @param war A URL of type "jar:" that points to a WAR file, or type
   *  "file:" that points to an unpacked directory structure containing
   *  the web application to be installed
   *
```

```
     * @exception IllegalArgumentException if the specified context path
     *  is malformed (it must be "" or start with a slash)
     * @exception IllegalStateException if the specified context path
     *  is already attached to an existing web application
     * @exception IOException if an input/output error was encountered
     *  during installation
     */
    public void install(String contextPath, URL war) throws IOException;

    /**
     * <p>Install a new web application, whose context configuration file
     * (consisting of a <code>&lt;Context&gt;</code> element) and web
     * application archive are at the specified URLs.</p>
     *
     * <p>If this application is successfully installed, a ContainerEvent
     * of type <code>INSTALL_EVENT</code> will be sent to all registered
     * listeners, with the newly created <code>Context</code> as an
     * argument.
     * </p>
     *
     * @param config A URL that points to the context configuration file
     *  to be used for configuring the new Context
     * @param war A URL of type "jar:" that points to a WAR file, or type
     *  "file:" that points to an unpacked directory structure containing
     *  the web application to be installed
     *
     * @exception IllegalArgumentException if one of the specified URLs
     *  is null
     * @exception IllegalStateException if the context path specified in
     *  the context configuration file is already attached to an existing
     *  web application
     * @exception IOException if an input/output error was encountered
     *  during installation
     */

    public void install(URL config, URL war) throws IOException;

    /**
     * Return the Context for the deployed application that is associated
     * with the specified context path (if any); otherwise return
     * <code>null</code>.
     *
     * @param contextPath The context path of the requested web
     * application
```

```
 */
public Context findDeployedApp(String contextPath);

/**
 * Return the context paths of all deployed web applications in this
 * Container.  If there are no deployed applications, a zero-length
 * array is returned.
 */
public String[] findDeployedApps();

/**
 * Remove an existing web application, attached to the specified
 * context path. If this application is successfully removed, a
 * ContainerEvent of type <code>REMOVE_EVENT</code> will be sent to
 * all registered listeners, with the removed <code>Context</code> as
 * an argument.
 *
 * @param contextPath The context path of the application to be
 * removed
 *
 * @exception IllegalArgumentException if the specified context path
 *  is malformed (it must be "" or start with a slash)
 * @exception IllegalArgumentException if the specified context path
 *  does not identify a currently installed web application
 * @exception IOException if an input/output error occurs during
 *  removal
 */
public void remove(String contextPath) throws IOException;

/**
 * Start an existing web application, attached to the specified
 * context path. Only starts a web application if it is not running.
 *
 * @param contextPath The context path of the application to be
 *  started
 * @exception IllegalArgumentException if the specified context path
 *  is malformed (it must be "" or start with a slash)
 * @exception IllegalArgumentException if the specified context path
 *  does not identify a currently installed web application
 * @exception IOException if an input/output error occurs during
 *  startup
 */
public void start(String contextPath) throws IOException;
```

```
  /**
   * Stop an existing web application, attached to the specified
   * context path.  Only stops a web application if it is running.

   *
   * @param contextPath The context path of the application to be
   *  stopped
   * @exception IllegalArgumentException if the specified context path
   *  is malformed (it must be "" or start with a slash)
   * @exception IllegalArgumentException if the specified context path
   *  does not identify a currently installed web application
   * @exception IOException if an input/output error occurs while
   *  stopping the web application
   */
  public void stop(String contextPath) throws IOException;
}
```

StandardHost 使用一个 org.apache.catalina.core.StandardHostDeployer 类型的帮助类来部署和安装 web 应用程序。下面你可以看到 StandardHost 如何使用 StandardDeployer 实例来部署和安装 web 应用程序。

```
/**
* The <code>Deployer</code> to whom we delegate application
* deployment requests.
*/
private Deployer deployer = new StandardHostDeployer(this);

public void install(String contextPath, URL war) throws IOException {
  deployer.install(contextPath, war);
}
public synchronized void install(URL config, URL war) throws
IOException {
  deployer.install(config, war);
}
public Context findDeployedApp(String contextPath) {
  return (deployer.findDeployedApp(contextPath));
}
public String[] findDeployedApps() {
  return (deployer.findDeployedApps());
}
public void remove(String contextPath) throws IOException {
  deployer.remove(contextPath);
}
public void start(String contextPath) throws IOException {
  deployer.start(contextPath);
```

```
}
public void stop(String contextPath) throws IOException {
  deployer.stop(contextPath);
}
```

StandardHostDeployer 将会在下一节讨论：

StandardHostDeployer 类

org.apache.catalina.core.StandardHostDeployer 类是 StandardHost 的一个帮助类用来部署和安装 web 应用程序。StandardHostDeployer 被设计成由 StandardHost 使用，它的构造函数接受一个 StandardHost 实例。

```
public StandardHostDeployer(StandardHost host) {
  super();
  this.host = host;
}
```

该类的方法将在下面的小节中介绍

Installing a Descriptor

StandardHostDeployer 类有两个 install 方法。第一个也是本小节介绍的用于安装描述符（descriptor）。第二个在下一小节介绍用于安装 WAR 文件和目录。

用于安装描述符的 install 方法如 Listing18.12 所示。StandardHost 实例在 HostConfig 通过 deployDecriptors 来调用 install 方法的时候调用。

Listing 18.12: The install method for installing descriptors

```
public synchronized void install(URL config, URL war)
  throws IOException {
  // Validate the format and state of our arguments
  if (config == null)
    throw new IllegalArgumentException
     (sm.getString("StandardHost.configRequired"));
  if (!host.isDeployXML())
    throw new IllegalArgumentException
      (sm.getString("StandardHost.configNotAllowed"));
  // Calculate the document base for the new web application (if
  // needed)
  String docBase = null; // Optional override for value in config file
  if (war != null) {
    String url = war.toString();
    host.log(sm.getString("StandardHost.installingWAR", url));
    // Calculate the WAR file absolute pathname
    if (url.startsWith("jar:")) {
```

```
      url = url.substring(4, url.length() - 2);
    }
    if (url.startsWith("file://"))
      docBase = url.substring(7);
    else if (url.startsWith("file:"))

      docBase = url.substring(5);
    else
      throw new IllegalArgumentException
        (sm.getString("standardHost.warURL", url));
  }

  // Install the new web application
  this.context = null;
  this.overrideDocBase = docBase;
  InputStream stream = null;
  try {
    stream = config.openStream();
    Digester digester = createDigester();
    digester.setDebug(host.getDebug());
    digester.clear();
    digester.push(this);
    digester.parse(stream);
    stream.close();
    stream = null;
  }
  catch (Exception e) {
    host.log
      (sm.getString("standardHost.installError", docBase), e);
        throw new IOException(e.toString());
  }
  finally {
    if (stream != null) {
      try {
        stream.close();
      }
      catch (Throwable t) {
        ;
        }
    }
  }
}
```

Installing a WAR file and a Directory

第二个 install 方法介绍一个上下文路径的字符串表示形式或者一个 URL 来表示 WAR 文件。该 install 方法如 Listing18.13 所示

Listing 18.13: The install method for installing a WAR file or a directory

```
public synchronized void install(String contextPath, URL war)
  throws IOException {
  // Validate the format and state of our arguments
  if (contextPath == null)
    throw new IllegalArgumentException
      (sm.getString("standardHost.pathRequired"));
  if (!contextPath.equals("") && !contextPath.startsWith("/"))
    throw new IllegalArgumentException

      (sm.getString("standardHost.pathFormat", contextPath));
  if (findDeployedApp(contextPath) != null)
    throw new IllegalStateException
      (sm.getString("standardHost.pathUsed", contextPath));
  if (war == null)
    throw new IllegalArgumentException
      (sm.getString("standardHost.warRequired"));

  // Calculate the document base for the new web application
  host.log(sm.getString("standardHost.installing",
  contextPath, war.toString()));
  String url = war.toString();
  String docBase = null;
  if (url.startsWith("jar:")) {
    url = url.substring(4, url.length() - 2);
  }
  if (url.startsWith("file://"))
    docBase = url.substring(7);
  else if (url.startsWith("file:"))
    docBase = url.substring(5);
  else
    throw new IllegalArgumentException
      (sm.getString("standardHost.warURL", url));

  // Install the new web application
  try {
    Class clazz = Class.forName(host.getContextClass());
    Context context = (Context) clazz.newInstance();
    context.setPath(contextPath);
```

```
    context.setDocBase(docBase);
    if (context instanceof Lifecycle) {
      clazz = Class.forName(host.getConfigClass());
      LifecycleListener listener =
        (LifecycleListener) clazz.newInstance();
      ((Lifecycle) context).addLifecycleListener(listener);
    }
    host.fireContainerEvent(PRE_INSTALL_EVENT, context);
    host.addChild(context);
    host.fireContainerEvent(INSTALL_EVENT, context);
  }
  catch (Exception e) {
    host.log(sm.getString("standardHost.installError", contextPath),
      e);
    throw new IOException(e.toString());
  }
}
```

注意一旦一个上下文被安装，它将会被添加到 StandardHost。

Starting A Context

StandardHostDeployer 的 start 方法用于启动一个上下文。它如 Listing18.14 所示

Listing 18.14: The start method of the StandardHostDeployer class

```
public void start(String contextPath) throws IOException {
  // Validate the format and state of our arguments
  if (contextPath == null)
    throw new IllegalArgumentException
      (sm.getString("standardHost.pathRequired"));
  if (!contextPath.equals("") && !contextPath.startsWith("/"))
    throw new IllegalArgumentException
      (sm.getString("standardHost.pathFormat", contextPath));
  Context context = findDeployedApp(contextPath);
  if (context == null)
     throw new IllegalArgumentException
      (sm.getstring("standardHost.pathMissing", contextPath));
  host.log("standardHost.start " + contextPath);
  try {
    ((Lifecycle) context).start();
  }
  catch (LifecycleException e) {
```

```
    host.log("standardHost.start " + contextPath + ": ", e);
      throw new IllegalStateException
    ("standardHost.start " + contextPath + ": " + e);
  }
}
```

Stopping A Context

要停止一个上下文，可以使用 StandardHostDeployer 的 stop 方法如 Listing18.15

Listing 18.15: The stop method in the StandardHostDeployer class

```
public void stop(String contextPath) throws IOException {
  // Validate the format and state of our arguments
  if (contextPath == null)
    throw new IllegalArgumentException
      (sm.getstring("standardHost.pathRequired"));
  if (!contextPath.equals("") && !contextPath.startsWith("/"))
    throw new IllegalArgumentException
      (sm.getstring("standardHost.pathFormat", contextPath));
  Context context = findDeployedApp(contextPath);
  if (context == null)
    throw new IllegalArgumentException
      (sm.getstring("standardHost.pathMissing", contextPath));
  host.log("standardHost.stop " + contextPath);
  try {

    ((Lifecycle) context).stop();
  }
  catch (LifecycleException e) {
    host.log("standardHost.stop " + contextPath + ": ", e);
    throw new IllegalStateException
      ("standardHost.stop " + contextPath + ": " + e);
  }
}
```

总结

部署器用于部署和安装 web 应用，由 org.apache.catalina.Deployer 表示。StandardHost 类实现了 Deployer，这样它就是一个可以部署 web 应用的特殊容器。StandardHost 使用一个帮助类来完成 web 应用的部署和安装，该帮助类为 org.apache.catalina.core.StandardHostDeployer。StandardHostDeployer 类提供了部署和安装应用，以及启动和停止上下文容器的代码。

第 19 章：管理 Servlet

综述

Tomcat4 和 5 有一个 Manager 应用程序用于管理部署的应用程序。跟其它应用程序不同，Manager 并不是在%CATALINA_HOME%/webapps 目录下面而是在%CATALINA_HOME%/server/webapps 下。Manager 有一个描述符 manager.xml 在%CATALINA_HOME$/webapps（Tomcat4）或者%CATALINA_HOME%/server/webapps（Tomcat5），当 Tomcat 启动的时候就安装 Manager。

注意 上下文描述符在第 18 章中讨论了

本章主要用于描述 Manager 应用，首先概括的解释了 Manager 是如何工作的，然后解释了 ContainerServlet 接口。

Manager 应用使用

Manager 应用可以在%CATALINA_HOME%/server/webapps/manager 目录下找到。该应用中的主 servlet 是 ManagerServlet。在 Tomcat4 中，该类属于 org.apache.catalina.servlets 包。在 Tomcat5 中，该类是 org.apache.catalina.manager 包的一部分，以 JAR 包的形式部署在 WEB-INF/lib 目录下：

注意 由于 Tomcat4 中的 Manager 应用程序比 Tomcat5 中的简单，所以它更容易学习，本章主要讨论它。在读完了本章后，你也可以理解 Tomcat5 中的 Manager 是如何工作的。

Here are the servlet elements in the deployment descriptor in Tomcat 4.

这里是 Tomcat4 中的部署描述符的 servlet 元素

```
<servlet>
  <servlet-name>Manager</servlet-name>
  <servlet-class>
    org.apache.catalina.servlets.ManagerServlet
  </servlet-class>
  <init-param>
    <param-name>debug</param-name>
    <param-value>2</param-value>
  </init-param>
</servlet>
<servlet>
  <servlet-name>HTMLManager</servlet-name>
  <servlet-class>
    org.apache,catalina.servlets.HTMLManagerServlet
  </servlet-class>
  <init-param>
```

```
      <param-name>debug</param-name>
      <param-value>2</param-value>
    </init-param>
  </servlet>
```

第一个 servlet 是 org.apache.catalina.servlets.ManagerServlet，第二个是 org.apache.catalina.servlets.HTMLManagerServlet。本章主要介绍 ManagerServerlet。

本应用程序的描述符 manager.xml，说明了本应用程序的上下文路径为/manager

```
<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>
```

第一个 servlet 映射元素说明如何调用 ManagerServlet

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

换句话说，如下形式的 URL 模式将会调用 MangerServlet:

http://localhost:8080/manager/

但是，注意在部署描述符中还有安全限制元素

```
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Entire Application</web-resource-name>
      <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
     <!-- NOTE:  This role is not present in the default users file -—>
       <role-name>manager</role-name>
    </auth-constraint>
  </security-constraint>
```

它的意思是说，整个应用程序只能被 manager 角色的用户使用。auth-login 元素规定用户需要提供正确的用户名密码来通过 BASIC 验证。

```
  <login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>Tomcat Manager Application</realm-name>
  </login-config>
```

在 Tomcat 中，用户角色在%CATALINA_HOME%/conf 目录下的 tomcat-users.xml 文件中有列表。因此,要进入 Manager 应用程序,必须给用户添加 manager 角色。

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

```
<tomcat-users>
  <role rolename="manager"/>
  <user username="tomcat" password="tomcat" roles="manager "/>
</tomcat-users>
```

通过该 tomcat-users.xml，可以使用用户名 tomcat 密码 tomcat 的用户访问 Manager 应用程序。

下面的是 ManagerServlet 的可用函数：

- list
- start
- stop
- reload
- remove
- resources
- roles
- sessions
- undeploy

查看 servlet 的 doGet 方法看下如何调用一个 function。

ContainerServlet 接口

一个实现了 org.apache.catalina.ContainerServlet 接口的 servlet 可以访问 StandardWrapper 对象。可以访问包装器，它也能访问表示该 web 应用的上下文对象，以及该上下文隶属的部署器（StandardHost 实例）以及其它对象。

ContainerServlet 接口如 Listing19.1 所示：

Listing 19.1: The ContainerServlet Interface

```
package org.apache.catalina;

public interface ContainerServlet {
  public Wrapper getWrapper();
  public void setWrapper(Wrapper wrapper);
}
```

ManagerServlet 初始化

通常情况下，一个 servlet 用一个 org.apache.catalina.core.StandardWrapper 实例表示。在第一次调用该 servlet 的时候，StandardedWrapper 对象的 loadServlet 方法会被调用，它又调用 servlet 的 init 方法。接下来你会看到 ManagerServlet 的 loadServlet 方法是如何工作的：

```
...
// Special handling for ContainerServlet instances
```

```
if ((servlet instanceof ContainerServlet) &&
  isContainerProvidedServlet(actualClass)) {
  ((ContainerServlet) servlet).setWrapper(this);
}

// Call the initialization method of this servlet
try {
  instanceSupport.fireInstanceEvent(InstanceEvent.BEFORE_INIT_EVENT,
    servlet);
  servlet.init(facade);
...
```

servlet 表示要加载的 servlet（在这里是 ManagerServlet）

在 if 语句块中，如果 servlet 是 org.apache.catalina.ContainerServlet 的实例并且 isContainerProvidedServlet 方法返回 true。就调用 ContainerServlet 接口的 setWrapper 接口。

ManagerServlet 类实现了 ContainerServlet，因此 servlet 是 ContainerSerlvet 的一个实例。在 StanadardWrapper 中的 isContainerProvidedServlet 方法如 Listing19.2 所示

Listing 19.2: The isContainerProvidedServlet method in the StandardWrapper class

```
private boolean isContainerProvidedServlet(String classname) {
  if (classname.startsWith("org.apache.catalina.")) {
    return (true);
  }
  try {
    Class clazz =
      this.getClass().getClassLoader().loadClass(classname);
    return (ContainerServlet.class.isAssignableFrom(clazz));
  }
  catch (Throwable t) {
    return (false);
  }
}
```

传递给 isContainerProvidedServlet 方法的参数 classname 是 ManagerSerlvet 的完全限定名（fully-qualified），即 org.apache.catalina.servlets.ManagerServlet。因此，isContainerProvidedServlet 方法返回 true。

如果该 servlet 类是 ContainerServlet 的子类，该方法也返回 true，如果 classname 是接口继承了 ContainerServlet 或者是实现了 ContainerServlet 的类名。

注 如果表示当前对象的类或接口跟 clazz 表示的类或接口或相同，或是超类、

意 超接口的时候 java.lang.Class 类的 isAssignableFrom(Class clazz)返回 true。

因此，表示 ManagerServlet 实例的 StandardWrapper 在它的 loadServlet 方法中会调用 ManagerServlet 的 setWrapper 方法。这里是 ManagerServlet 类对 setWrapper 方法的实现：

```
public void setWrapper(Wrapper wrapper) {
  this.wrapper = wrapper;
  if (wrapper == null) {
    context = null;
    deployer = null;
  }
  else {
    context = (Context) wrapper.getParent();
    deployer = (Deployer) context.getParent();
  }
}
```

如果参数 wrapper 不是 null，将会执行 else 语句块。将表示 Manager 应用的上下文赋值给 context 变量并将该上下文部署在 StandardHost 实例上。Deployer 非常重要，它会在 ManagerServlet 的好几个方法中使用到。

在 StandardWrapper 的 loadServlet 调用了 StandardWrapper 的 setWrapper 方法之后。loadServlet 方法调用 MangerServlet 的 init 方法。

列出 Web 应用

可以使用如下 URL 来查看部署的所有应用程序：

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

下面是一则输出的例子

```
OK - Listed applications for virtual host localhost
/admin:stopped:0:../server/webapps/admin
/app1:running:0:C:\123data\JavaProjects\Pyrmont\webapps\app1
/manager:running:0:../server/webapps/manager
```

上面的 URL 将会调用 ManagerServlet 的 list 方法，如 Listing19.3 所示

Listing 19.3: The list method of ManagerServlet

```
protected void list(PrintWriter writer) {
  if (debug >= 1)
    log("list: Listing contexts for virtual host '" +
      deployer.getName() + "'");
  writer.println(sm.getString("managerServlet.listed",
    deployer.getName()));
  String contextPaths[] = deployer.findDeployedApps();
```

```
  for (int i = 0; i < contextPaths.length; i++) {
    Context context = deployer.findDeployedApp(contextPaths[i]);
    String displayPath = contextPaths[i];
    if( displayPath.equals("") )
      displayPath = "/";
    if (context != null ) {
      if (context.getAvailable()) {
        writer.println(sm.getString("managerServlet.listitem",
          displayPath, "running", ""
            + context.getManager().findSessions().length,
            context.getDocBase()));
      }
      else {
        writer.println(sm.getString("managerServlet.listitem",
          displayPath, "stopped", "0", context.getDocBase()));
      }
    }
  }
}
```

方法 list 调用部署器的 findDeployedApps 获得所有部署在 Catalina 中的上下文的路径。然后变量路径数组获得每个独立的上下文并检查该上下文是否可用。对于每个可用的上下文，list 方法打印出上下文路径，running 字符串，用户的 Session 个数以及文档基（document base）。对于不可用的上下文，list 方法打印出上下文路径，stopperd 字符串，0 以及文档基。

启动一个 web 应用

可以使用如下 URL 来启动一个 web 应用：

http://localhost:8080/manager/start?path=/contextPath

contextPath 是要启动的应用的上下文路径。例如，要启动 admin 应用，可以使用如下路径：

http://localhost:8080/manager/start?path=/admin

如果该应用程序已经启动，你会收到错误提示信息：

根据该 URL，ManagerServlet 调用 start 方法，如 Listing19.4 所示

Listing 19.4: The start method of the ManagerServlet class

```
protected void start(PrintWriter writer, String path) {
  if (debug >= 1)
    log("start: Starting web application at '" + path + "'");

  if ((path == null) || (!path.startsWith("/") && path.equals(""))) {
    writer.println(sm.getString("managerServlet.invalidPath", path));
```

```
    return;
  }
  String displayPath = path;
  if( path.equals("/") )
    path = "";

  try {
    Context context = deployer.findDeployedApp(path);
    if (context == null) {
      writer.println(sm.getString("managerServlet.noContext",
        displayPath));
      return;
    }
    deployer.start(path);
    if (context.getAvailable())
      writer.println
        (sm.getString("managerServlet.started", displayPath));
    else
      writer.println
        (sm.getString("managerServlet.startFailed", displayPath));
  }
  catch (Throwable t) {
    getServletContext().log
    (sm.getString("managerServlet.startFailed", displayPath), t);
    writer.println
      (sm.getString("managerServlet.startFailed", displayPath));
    writer.println(sm.getString("managerServlet.exception",
      t.toString()));
  }
}
```

在一些检查之后，start 方法调用部署器的 findDeployedApp 方法，该方法通过 path 路径获得上下文，如果上下文不为空，start 方法调用部署器的 start 方法启动应用程序。

停止 web 应用

可以使用如下命令来停止一个应用程序：

http://localhost:8080/manager/stop?path=/contextPath

contextPath 是你想要停止的 web 应用的路径，如果该应用并没有正在运行，会返回错误信息：

ManagerSerlvet 收到请求的时候，它调用 stop 方法，如 Listing19.5 所示：

Listing 19.5: The stop method of the ManagerServlet class

```
protected void stop(PrintWriter writer, String path) {
  if (debug >= 1)
    log("stop: Stopping web application at '" + path + "'");
  if ((path == null) || (!path.startsWith("/") && path.equals(""))) {
    writer.println(sm.getString("managerServlet.invalidPath", path));
    return;
  }
  String displayPath = path;
  if( path.equals("/") )
    path = "";

  try {
    Context context = deployer.findDeployedApp(path);
    if (context == null) {
      writer.println(sm.getString("managerServlet.noContext",
        displayPath));
      return;
    }
    // It isn't possible for the manager to stop itself
    if (context.getPath().equals(this.context.getPath())) {
      writer.println(sm.getString("managerServlet.noSelf"));
      return;
    }
    deployer.stop(path);
    writer.println(sm.getString("managerServlet.stopped",
      displayPath));
  }
  catch (Throwable t) {
    log("ManagerServlet.stop[" + displayPath + "]", t);
    writer.println(sm.getString("managerServlet.exception",
      t.toString()));
  }
}
```

通过上面的代码可以明白 stop 方法是如何工作的。ManagerServlet 类的其它方法可以在源码中查看。

总结

本章介绍了如何使用一个特殊接口 ContainerServelet 来创建一个可以访问 Catalina 内部类的 Servlet。Manager 应用可以用于管理部署应用，并说明了如何从包装器对象获得其它对象。另外可以设计一个更复杂的 Servlet 来管理 Tomcat。

第 20 章：JMX-Based Management

第 19 章讨论了 Manger 应用程序，演示了如何使用实现了 ContainerServlet 的 ManagerServlet 类来访问 Catalina 的内部对象。本章演示用另一种更成熟的方法来管理 Tomcat，该方法使用 Java 管理扩展（Java Management extentsions，JMX）。对于不熟悉 JMX 的读者，本章开头先进行了简单的介绍。另外本章解释了常用建模库（Commons Modeler library），它可以很简单的写管理 beans（Managed Beans），这些对象用于管理其它的对象。另外还提供了例子来进行 JMX 在 Tomcat 中使用的说明。

JMX 简介

目前为止，ContainerServlet 接口已经足够管理 Catalina 内部类的访问，为什么还要关注 JMX？答案是 JMX 提供了更大的灵活性。很多基于服务的应用程序，如 Tomcat，JBoss，JONAS，Geronimo 以及其它的应用，都使用 JMX 来管理他们的

JMX 目前的版本是 1.21，定义了一个公开的管理 Java 对象的标准。例如，Tomcat4 和 5 使用 JMX 的管理程序来使得各种对象（如服务器、主机、上下文、阀门等等）可用。Tomcat 的开发者编写了 Admin 应用程序用于管理。

一个可以使用 JMX 管理器来管理的 Java 对象称为 JMX 管理资源（JMX manageable resource）。事实上，一个 JMX 管理资源也可以是一个应用程序、一个实现或者一个服务、设备、用户等等。JMX 管理资源用 Java 写或者提供一个 Java 包装。

要想让一个 Java 对象称为 JMX 管理资源，必须创建另一个名为 Managed Bean 或者 MBean 的对象。org.apache.catalina.mbeans 包包括一些 MBeans。ConnectorMBean, StandardEngineMBean, StandardHostMBean, StandardContextMBean 是 Managed Bean 的例子。从他们的名字你可以猜到 ConnectMBean 用于管理连接器，StandardContextMBean 用于管理 org.apache.catalina.core.StandardContext 对象等等。当然，你也可以编写 MBean 管理多个 Java 对象。

MBean 将 Java 对象的属性和方法暴露给管理应用程序（management application）。管理应用程序本身不能直接访问 Java 对象。因此可以选择任意的属性和方法让管理应用程序访问。

一旦你有一个 MBean 类，你需要初始化它的一个对象并将其注册到一个 MBean 服务器的对象（MBean server）。MBean 服务器是应用程序中所有的 MBean 的中心登记处（central registry）。管理应用程序通过 MBean 服务器访问 MBeans。将 JMX 和 Servlet 应用程序相比较，管理应用程序相当于一个 web 浏览器。MBean 服务器相当于一个 Servlet 容器，它为客户端提供管理资源的访问。而 MBeans 相当于 Servlet 或者 JSP 页面。就像是 web 浏览器从来不直接接触 Servlet/JSP 页面，而是通过容器访问。管理应用程序也不会直接访问 MBeans，而是通过 MBean 服务器来进行。

一共有四种 MBean：标准 standard，动态 dynamic，打开 open，和模型 model。其中，标准 MBean 是里面最容易编写的，但是他的灵活性也最小。另外三种更灵

活，我们将会特别关注模型MBeans，因为Catalina就是使用了这种类型的MBean。在后边有对模型 MBeans 的讨论，我们会省略动态和打开 JMX 因为它们与本章没有关系。感兴趣的用户可以阅读 JMX1.21 规范问答了解更多的细节。

从结构上来看，JMX 规范分为 3 层，设备层（instrumentation level），代理层（agent level），和分布服务层（distributed services level）。MBean 服务器处于代理层，而 MBeans 处于设备层。分布服务层将会在 JMX 规范的未来版本涉及到。

设备层定义了编写 JMX 管理资源的标准，也就是怎么写 MBeans。代理层提供了如何创建一个代理。一个代理封装了一个 MBean 服务器以及用于处理 MBeans 的服务。代理和 MBeans 处于同一个 Java 虚拟机中，由于 JMX 规范带有一些参考实现，你不需要自己编写 MBean 服务器。参考实现提供了一种创建默认 MBean 服务器的方式

Note Download the specification and reference implementation from http://java.sun.com/products/JavaManagement/download.html. MX4J, an open source version of JMX whose library is included in the software accompanying this book, is available from http://mx4j.sourceforge.net

Warning The zip file that accompanies this book contains the mx4j.jar file that packages the version 2.0 beta 1 of MX4J, replacing the mx4j-jmx.jar file included in Tomcat 4.1.12. This was done in order for you to use the more recent version of JMX (version 1.2.1).

JMX API

参考实现组成了 Java 标准库中的 javax.management 包以及其它 JMX 编程相关的特殊领域的包。本节会讨论 API 中的一些重要类型。

MBeanServer

javax.management.MBeanServer 是一个表示 MBean 服务器的接口。要创建一个 MBean 服务器，可以使用 javax.management.MBeanServerFactory 中的方法即可，例如 createMBean 方法。

要在 MBean 服务器注册 MBean，调用 MBean 服务器对象的 registerMBean 方法即可。下面是 registerMBean 方法的签名：

```
public ObjectInstance registerMBean(java.lang.Object object,
  ObjectName name) throws InstanceAlreadyExistsException,
  MBeanRegistrationException, NotCompliantMBeanException
```

registerMBean 方法需要传递一个 MBean 实例以及实例的对象名(ObjectName)。ObjectName 类似于 HashMap 中的键值，它必须是唯一的。registerMBean 返回一个 ObjectInstance 对象。 javax.management.ObjectInstance 封装了 MBean 的对象名以及它的类名。

要检索 MBean 对象或对象集是否匹配一个模式，MBeanServer 接口提供了两个方法 queryNames 以及 queryMBeans。queryNames 方法返回一个包括所有对象名匹配模式的 MBean 的 ava.util.Set 集合。下面是 queryName 方法的签名：

public java.util.Set queryNames(ObjectName *name*, QueryExp *query*)

参数 query 指定筛选条件

如果 name 的值为 null 或者没有指定域和键的值，就返回所有注册 MBean 的 ObjectName。如果 query 是 null，没有过滤器使用。

queryMBeans 方法跟 queryName 相似，但是他返回包含所有选择的 ObjectInstance 对象的 java.util.Set 集合。queryMBeans 方法的签名如下：

public java.util.Set queryMBeans(ObjectName name, QueryExp query)

一旦你获得了需要的 MBean，就可以操作暴露的属性和方法。

可以使用 MBeanServer 接口中的 invoke 方法来调用注册 MBean 中的任何方法。MBeanServer 接口的 getAttribute 和 setAttribute 方法用于获得和设置注册 MBean 的属性。

ObjectName

MBean 服务器用于注册 MBean。对于在 MBean 服务器上的 MBean，它们都有唯一的对象名，就像是 HashMap 中的对象都有唯一的键值一样。

对象名由 javax.management.ObjectName 类表示。一个对象名由两部分组成：域（domain）和一个键值集合。域是一个字符串，可以是空字符处。在一个对象名种，域之后的是冒号以及一个或多个键值对。键是一个非空字符串，不能包含下面的符号 equal sign, comma, colon, asterisk, 和 question mark。相同的键在一个对象名种只发生一次。

键和值之间用等号隔开，两个键值对之间用逗号隔开。例如，下面是一个合法的对象名：

myDomain:type=Car,color=blue

ObjectName 也可以表示在 MBean 服务器上查找 MBean 的属性模式。一个 ObjectName 是一个它的域和键值对表示的模式。一个 ObjectName 模式可以零个或多个键。

标准 MBeans

标准 MBean 是最简单的 MBean。要是用标准 MBean 管理一个 Java 对象，需要以下工作：

- 创建一个接口，名为你的类名加上后缀 MBean。例如，如果要管理的 Java 类是 Car，接口名酒味 CarMBean。
- 修改 Java 类，让它实现你创建的接口。
- 创建一个代理，该代理必须包括一个 MBean 服务器。
- 为你的 MBean 创建一个 ObjectName。
- 初始化 MBean 服务器。
- 想 MBean 服务器注册 MBean。

标准 MBean 更易于变现，但是使用它们的话需要修改你自己的类，这在某些情况下可以，但是有时候是不行的。其它类型的 MBean 允许在不修改类的情况下管理对象。

考虑下面的类如何变成 JMX 管理的，作为一个标准 MBean 的例子。

```
package ex20.pyrmont.standardmbeantest;

public class Car {
  private String color = "red";

  public String getColor() {
    return color;
  }
  public void setColor(String color) {

    this.color = color;
  }
  public void drive() {
    System.out.println("Baby you can drive my car.");
  }
}
```

修改的第一步是要实现 CarMBean 接口，新的 car 类如 Listing20.1 所示：

Listing 20.1: The modified Car class

```
package ex20.pyrmont.standardmbeantest;

public class Car implements CarMBean {
  private String color = "red";

  public String getColor() {
    return color;
  }
  public void setColor(String color) {
    this.color = color;
  }
  public void drive() {
```

```
    System.out.println("Baby you can drive my car.");
  }
}
```

接下来创建 CarMBean 接口如 Listing20.2

Listing 20.2: The CarMBean interface

```
package ex20.pyrmont.standardmbeantest;

public interface CarMBean {
  public String getColor();
  public void setColor(String color);
  public void drive();
}
```

基本上，在接口中需要声明所有你需要暴露的方法。在该例子中，CarMBean 接口列出了 Car 类中的所有接口。如果不想让管理程序使用 driver 方法，从 CarMBean 接口中移除 driver 方法即可。

最后,Listing20.3 的 StandardAgent 展示了如何创建一个标准 MBean 来管理 Car 类的对象。

Listing 20.3: The StandardAgent class

```
package ex20.pyrmont.standardmbeantest;

import javax.management.Attribute;

import javax.management.ObjectName;
import javax.management.MBeanServer;
import javax.management.MBeanServerFactory;

public class StandardAgent {
  private MBeanServer mBeanServer = null;
  public StandardAgent() {
    mBeanServer = MBeanServerFactory.createMBeanServer();
  }
  public MBeanServer getMBeanServer() {
    return mBeanServer;
  }
  public ObjectName createObjectName(String name) {
    ObjectName objectName = null;
    try {
      objectName = new ObjectName(name);
    }
    catch (Exception e) {
```

```
    }
    return objectName;
  }
  private void createStandardBean(ObjectName objectName,
    String managedResourceClassName) {
    try {
      mBeanServer.createMBean(managedResourceClassName, objectName);
    }
    catch(Exception e) {
    }
  }

  public static void main(String[] args) {
    StandardAgent agent = new StandardAgent();
    MBeanServer mBeanServer = agent.getMBeanServer();
    String domain = mBeanServer.getDefaultDomain();
    String managedResourceClassName =
      "ex20.pyrmont.standardmbeantest.Car";
    ObjectName objectName = agent.createObjectName(domain + ":type=" +
      managedResourceClassName);
    agent.createStandardBean(objectName, managedResourceClassName);

    // manage MBean
    try {
      Attribute colorAttribute = new Attribute("Color","blue");
      mBeanServer.setAttribute(objectName, colorAttribute);
      System.out.println(mBeanServer.getAttribute(objectName,
        "Color"));
      mBeanServer.invoke(objectName,"drive",null,null);
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

StandardAgent 类是一个代理，它创建一个 MBean 服务器，用它注册一个 CarMBean。首先要做的是要初始化 mBeanServer，在 StandardAgent 类的构造函数中，使用了 MBeanServerFactory 类的 createMBeanServer 方法来构造一个 MBean 服务器。

```
  public StandardAgent() {
    mBeanServer = MBeanServerFactory.createMBeanServer();
  }
```

createMBeanServer 方法返回一个实现了 JMX 的 MBeanServer 对象。高级的 JMX 程序员可能想自己实现 MBeanServer，但是本书并不关注该问题。

StandardAgent 的 createObjectName 根据传递给它的 String 参数返回一个 ObjectName 对象。StandardAgent 的 createObjectName 方法调用了 MBeanServer 的 createMBean 方法。createMBean 接受要管理的资源的类名以及唯一的用于区别它们的 ObjectName 对象。createMBean 还将创建的 MBean 对象注册到 MBeanServer 上。因为标准 MBean 遵循传统的命名。不需要想 createMBean 方法提供 MBean 类型名。如果管理的资源类名为 Car，它的 MBean 就是 CarMBean。

StandardAgent 的 main 方法首先创建了 StandardAgent 实力，然后调用 getMBeanServer 方法获得一个 MBeanServer 实例。

```
StandardAgent agent = new StandardAgent();
MBeanServer mBeanServer = agent.getMBeanServer();
```

然后为 CarMBean 创建了一个 ObjectName。MBeanServer 的默认域用于 ObjectName 的域。一个名为 type 的键 key 被添加到域上，type 的值是管理资源的完全限定名。

```
String domain = mBeanServer.getDefaultDomain();
String managedResourceClassName =
  "ex20.pyrmont.standardmbeantest.Car";
ObjectName objectName = agent.createObjectName(domain + ":type=" +
  managedResourceClassName);
```

然后 main 方法调用 createStandardBean 方法，传递一个 objectName 以及管理资源的类名。

```
agent.createStandardBean(objectName, managedResourceClassName);
```

接下来主方法通过 CarMBean 实例管理 Car 对象。它创建了一个名为 colorAttribute 的 Attribute 对象表示 Color 属性并将其值设置为 blue。然后调用 setAttribute 方法，参数为 objectName 和 colorAttribute。然后使用 MBeanServer 上的 invoke 方法来调用 driver 方法。

```
// manage MBean
try {
  Attribute colorAttribute = new Attribute("Color","blue");
  mBeanServer.setAttribute(objectName, colorAttribute);
  System.out.println(mBeanServer.getAttribute(objectName,
    "Color"));
  mBeanServer.invoke(objectName,"drive",null,null);
}
```

如果运行 StandardAgent 类，会得到以下输出。

```
blue
Baby you can drive my car.
```

你可能会疑问为什么需要 JMX 来管理 Java 对象。在 StandardAgent 类里可以直接访问 Car 对象。的确是这样，但是这里的关键是你可以选择要暴露哪些属性和

方法让管理程序来使用。另外可以在本章的“The Application”一节中看到MBeanServer 扮演了管理对象和管理程序之间的中间层的角色。

模型 MBeans

模型 MBeans 提供了更多的灵活性。它编写起来更难，但是你不需要在修改自己的 Java 类。在不适合修改类的时候使用模型 MBean 是一种更好的选择。

在使用标准 MBean 的时候需要要管理的资源必须实现一个自己编写的接口。跟标准 MBean 不同，使用模型 MBean 的时候，不需要编写任何接口。而是使用 javax.management.modelmbean.ModelMBean 接口来表示模型 MBean。你只需要实现这个接口即可，而 javax.management.modelmbean.RequiredModelMBean 类提供了改接口的默认实现。你可以初始化 RequiredModelMBean 类或者他的子类。

注意 ModelMBean 接口的其它实现也可以，例如在 Commons Modeler 库中，有对该接口的实现，而不是继承 RequiredModelMBean

要编写模型 MBean 的最大挑战是告诉你的模型 MBean 对象哪些属性和方法要暴露给代理。可以通过创建 javax.management.modelmbean.ModelMBeanInfo 对象实现这一点。ModelMBeanInfo 对象用于描述暴露给代理的构造器、属性、操作以及事件监听器。创建一个 ModelMBeanInfo 对象是一个负值的工作，但是一旦你做好了一个，你只需要将其跟 ModelMBean 对象关联即可。

使用 RequiredModelMBean 作为你的 ModelMBean 的实现，有两种方法可以将你的 ModelMBean 关联到 ModelMBeanInfo：

1. 传递一个 ModelMBeanInfo 对象给 RequiredModelMBean 的构造函数
2. 传递一个 ModelMBeanInfo 对象给 RequiredModelMBean 对象的 setModelMBeanInfo 方法。

在创建一个 ModelMBean 之后，必须使用 ModelMBean 接口的 setManagedResource 方法将其关联到管理资源。该方法签名如下：

```
public void setManagedResource(java.lang.Object managedResource,
  java.lang.String managedResourceType) throws MBeanException,
  RuntimeOperationsException, InstanceNotFoundException,
  InvalidTargetObjectTypeException
```

managedResourceType 参数的值可以是如下之一，ObjectReference, Handle, IOR, EJBHandle, or RMIReference。目前只支持 ObjectReference。

然后当然需要创建一个 ObjectName 对象并经模型 MBean 注册到 MBean 服务器。

本节提供了一个例子来说明模型 MBean 的使用，使用了跟上一个例子相同的 Car 对象、在看该实例之前先看用于描述管理资源的属性和操作的 ModelMBeanInfo 接口。

MBeanInfo 和 ModelMBeanInfo

javax.management.mbean.ModelMBeanInfo 接口描述了要暴露给 MOdelMBean 的构造函数、属性、操作和监听器。构造函数使用 javax.management.modelmbean.ModelMBeanConstructorInfo 表示。属性使用 javax.management.modelmbean.ModelMBeanAttributeInfo 表示。操作用 javax.management.modelmbean.ModelMBeanOperationInfo 表示。监听器用 javax.management.modelmbean.ModelMBeanNotificationInfo 表示。在本章中，我们只关注操作和属性。

JMX 提供了 ModelMBeanInfo 的默认实现：javax.management.modelmbean.ModelMBeanInfoSupport 类。下面是 ModelMBeanInfoSupport 类的构造函数的签名，它们在例子中会使用到

```
public ModelMBeanInfoSupport(java.lang.String className,
  java.lang.String description, ModelMBeanAttributeInfo[] attributes,
  ModelMBeanConstructorInfo[] constructors,
  ModelMBeanOperationInfo[] operations,
  ModelMBeanNotificationInfo[] notifications)
```

可以使用 ModelMBeanAttributeInfo 的构造函数构建一个 ModelMBeanAttributeInfo 对象：

```
public ModelMBeanAttributeInfo(java.lang.String name,
  java.lang.String type, java.lang.String description,
  boolean isReadable, boolean isWritable,
  boolean isIs, Descriptor descriptor)
  throws RuntimeOperationsException
```

下面是它的参数列表

- name. The name of the attribute 属性名
- type. The type or class name of the attribute 属性的类型或者类名
- description. The description of the attribute.属性的描述
- isReadable. true if the attribute has a getter method, false otherwise.如果该属性有相应的 get 方法为真，否则为 false
- isWritable. true if the attribute has a setter method, false otherwise.如果该属性有响应的 set 方法为真，否则为 false。
- isIs. true if the attribute has an is getter, false otherwise.如果该属性有 is getter 为真，否则为 false。
- descriptor. An instance of Descriptor containing the appropriate metadata for this instance of the Attribute. If it is null then a default descriptor will be created. Descriptor 的实例，包括该 Attribute 的元数据。如果它是 null，就创建一个默认的 descriptor。

可以使用如下 ModelMBeanOperationInfo 构造函数来创建一个该类的对象

```
public ModelMBeanOperationInfo(java.lang.String name,
  java.lang.String description, MBeanParameterInfo[] signature,
  java.lang.String type, int impact, Descriptor)
  throws RuntimeOperationsException
```

下面是参数列表

- name. The name of the method.方法名
- description. The description of the operation.该操作的描述
- signature, an array of MBeanParameterInfo objects describing the parameters of the method. MBeanParameterInfo 对象数组描述该方法的参数。
- type. The type of the method's return value.返回值类型
- impact. The impact of the method. The value is one of the following: INFO, ACTION, ACTION_INFO, UNKNOWN.该方法的作用：如下值之一 INFO, ACTION, ACTION_INFO, UNKNOWN.
- descriptor. An instance of Descriptor containing the appropriate metadata, for this instance of the MBeanOperationInfo. 包含该 MBeanOperationInfo 对象的元数据的 Deccriptor。

ModelMBean 例子

这个例子说明了如何使用模型 MBean 来管理一个 Car 对象，如 Listing20.4 所示

Listing 20.4: The Car class

```
package ex20.pyrmont.modelmbeantest1;

public class Car {
  private String color = "red";
  public String getColor() {
    return color;
  }
  public void setColor(String color) {
    this.color = color;
  }
  public void drive() {
    System.out.println("Baby you can drive my car.");
  }

}
```

对于模型 MBean，不需要想标准 MBean 那样编写接口。你只需简单的初始化 RequiredMBean 类。Listing20.5 提供了 ModelAgent 类，它创建了 MBean 并管理 Car 对象

Listing 20.5: The ModelAgent class

```
package ex20.pyrmont.modelmbeantest1;

import javax.management.Attribute;
import javax.management.Descriptor;
import javax.management.MalformedObjectNameException;
import javax.management.MBeanOperationInfo;
import javax.management.MBeanParameterInfo;
import javax.management.MBeanServer;
import javax.management.MBeanServerFactory;
import javax.management.ObjectName;
import javax.management.modelmbean.DescriptorSupport;
import javax.management.modelmbean.ModelMBean;
import javax.management.modelmbean.ModelMBeanAttributeInfo;
import javax.management.modelmbean.ModelMBeanInfo;
import javax.management.modelmbean.ModelMBeanInfoSupport;
import javax.management.modelmbean.ModelMBeanOperationInfo;
import javax.management.modelmbean.RequiredModelMBean;

public class ModelAgent {

  private String MANAGED_CLASS_NAME =
    "ex20.pyrmont.modelmbeantest1.Car";
  private MBeanServer mBeanServer = null;

  public ModelAgent() {
    mBeanServer = MBeanServerFactory.createMBeanServer();
  }

  public MBeanServer getMBeanServer() {
    return mBeanServer;
  }

  private ObjectName createObjectName(String name) {
    ObjectName objectName = null;
    try {
      objectName = new ObjectName(name);
    }
    catch (MalformedObjectNameException e) {
      e.printStackTrace();
    }
```

```
  return objectName;
}

private ModelMBean createMBean(ObjectName objectName,
  String mbeanName) {
  ModelMBeanInfo mBeanInfo = createModelMBeanInfo(objectName,

    mbeanName);
  RequiredModelMBean modelMBean = null;
  try {
    modelMBean = new RequiredModelMBean(mBeanInfo);
  }
  catch (Exception e) {
    e.printStackTrace();
  }
  return modelMBean;
}

private ModelMBeanInfo createModelMBeanInfo(ObjectName
  inMbeanObjectName, String inMbeanName) {
  ModelMBeanInfo mBeanInfo = null;
  ModelMBeanAttributeInfo[] attributes = new
    ModelMBeanAttributeInfo[1];
  ModelMBeanOperationInfo[] operations = new
    ModelMBeanOperationInfo[3];
  try {
    attributes[0] = new ModelMBeanAttributeInfo("Color",
      "java.lang.String",
      "the color.", true, true, false, null);
    operations[0] = new ModelMBeanOperationInfo("drive",
      "the drive method",
      null, "void", MBeanOperationInfo.ACTION, null);
    operations[1] = new ModelMBeanOperationInfo("getColor",
      "get color attribute",
      null, "java.lang.String", MBeanOperationInfo.ACTION, null);

    Descriptor setColorDesc = new DescriptorSupport(new String[] {
      "name=setColor", "descriptorType=operation",
      "class=" + MANAGED_CLASS_NAME, "role=operation"});
    MBeanParameterInfo[] setColorParams = new MBeanParameterInfo[] {
      (new MBeanParameterInfo("new color", "java.lang.String",
      "new Color value") )} ;
    operations[2] = new ModelMBeanOperationInfo("setColor",
      "set Color attribute", setColorParams, "void",
```

```
        MBeanOperationInfo.ACTION, setColorDesc);

      mBeanInfo  = new ModelMBeanInfoSupport(MANAGED_CLASS_NAME,
        null, attributes, null, operations, null);
    }
    catch (Exception e) {
      e.printStackTrace();
    }
    return mBeanInfo;
  }

  public static void main(String[] args) {
    ModelAgent agent = new ModelAgent();
    MBeanServer mBeanServer = agent.getMBeanServer();
    Car car = new Car();
    String domain = mBeanServer.getDefaultDomain();
    ObjectName objectName = agent.createObjectName(domain +
      ":type=MyCar");
    String mBeanName = "myMBean";

    ModelMBean modelMBean = agent.createMBean(objectName, mBeanName);
    try {
      modelMBean.setManagedResource(car, "ObjectReference");
      mBeanServer.registerMBean(modelMBean, objectName);
    }
    catch (Exception e) {
    }

    // manage the bean
    try {
      Attribute attribute = new Attribute("Color", "green");
      mBeanServer.setAttribute(objectName, attribute);

      String color = (String) mBeanServer.getAttribute(objectName,
        "Color");
      System.out.println("Color:" + color);

      attribute = new Attribute("Color", "blue");
      mBeanServer.setAttribute(objectName, attribute);
      color = (String) mBeanServer.getAttribute(objectName, "Color");
      System.out.println("Color:" + color);
      mBeanServer.invoke(objectName, "drive", null, null);
    }
    catch (Exception e) {
```

```
        e.printStackTrace();
      }
    }
}
```

如你看到的，编写 MBean 需要做大量的工作，尤其是声明暴露的属性和操作的时候。接下来一节将会看到使用常用模型库（Commons Modeler library）更快的编写模型 MBean。

常用模型

常用模型库是 Apache 软件基金会 Jakarta 项目下的子项目。它提供了更简便的方法来编写模型 MBean。它最大的帮助在于使用它你不需要创建自己的 ModelMBeanInfo 对象。

回想前面的创建 RequiredModelMBean 例子的方法，需要创建 ModelMBeanInfo 对象以传递个 RequiredModelMBean 的构造函数。

```
ModelMBeanInfo mBeanInfo = createModelMBeanInfo(objectName,
  mbeanName);
  RequiredModelMBean modelMBean = null;
  try {
    modelMBean = new RequiredModelMBean(mBeanInfo);

  }
  ...
```

ModelMBeanInfo 对象用于描述暴露的属性和操作，而编写 createModelMBeanInfo 是一项负责的工作，必须列出所有的属性和操作然后将其传递给 createModelMBeanInfo。

使用常用 模型，不需要再使用 createModelMBeanInfo 对象。对模型 MBean 的描述被封住到 org.apache.catalina.modeler.ManagedBean 对象中。不需要在编写代码来暴露属性和操作。只需要编写一个 XMl 文档列出你想创建的 MBean、对于每个 MBean，需要写出 MBean 类和管理资源的完全限定名，以及暴露的方法和属性。然后使用 org.apache.commons.modeler.Registry 对象读取 XML 文档，就可以创建一个包括所有 XML 文档中描述的所有 ManagedBean 实例的 MBeanServer 实例。

然后可以调用 ManagedBean 实例的 createMBean 方法来创建一个模型 MBean。其它的工作就是平常需要做的了。需要创建 ObjectName 实例并将 MBean 注册到 MBean 服务器。接下来看看描述文件是如何工作的，然后讨论最重要的类：Modeler, Registry, ManagedBean, 以及 BaseModelMBean。

注意 Tomcat4 使用旧版本的模型，有些方法现在已不赞成继续使用。我们会讨论 Tomcat4 的版本这样你可以理解 org.apache.catalina.mbeans 中的 MBeans。

MBean Descriptor

一个 MBean 描述符是用于描述 MBean 服务器管理的模型 MBean 的 XML 文档。一个描述符的开头内容如下：

```
<?xml version="1.0"?>
<!DOCTYPE mbeans-descriptors PUBLIC
"-//Apache Software Foundation//DTD Model MBeans Configuration File"
"http://jakarta.apache.org/commons/dtds/mbeans-descriptors.dtd">
```

它的根元素是 mbeans-descriptors

```
<mbeans-descriptors>
...
</mbeans-descriptors>
```

在 mbeans-descriptors 标签内的元素师 mbean 元素,每一个表示一个模型 MBean。Mbean 元素包括表示属性、操作、构造器、监听器的元素。接下的子节中会讨论 Tomcat 的 MBean 描述符中使用到的三种元素：

mbean

Mbean 元素描述模型 Mbean，它包括构造相应的 ModelMBeanInfo 对象的信息，mbean 元素如下定义

```
<!ELEMENT mbean (descriptor?, attribute*, constructor*, notification*,
operation*)>
```

Mbean 可以有选择性的 descriptor 元素，0 或多个 attribute 元素，0 或多个 constructor 元素，0 或多个 notification 元素，0 或多个 opertion 元素。

Mbean 元素可以有如下属性：

- classname. Fully qualified Java class name of the ModelMBean implementation. If this attribute is not present, the org.apache.commons.modeler.BaseModelMBean will be used. ModelMBean 实现的完全限定名。如果该属性没有，会使用 org.apache.commons.modeler.BaseModelMBean。
- description. A description of this model MBean.对该模型 MBean 的描述
- domain. The MBean server's domain in which the ModelMBean created by this managed bean should be registered, when creating its ObjectName.该 managed bean 创建的模型 MBean 要注册的服务器的域。
- group. Optional name of a "grouping classification" that can be used to select groups of similar MBean implementation classes. 选择性的“grouping classification”可用于选择相似的 MBean 的组。

- name. A name that uniquely identifies this model MBean. Normally, the base class name of the corresponding server component is used.name 用于唯一确认 MBean 的 ID,一般使用服务器组件的基类名。
- type. Fully qualified Java class name of the managed resource implementation class.管理资源类的完全限定名。

attribute

使用 attribute 元素描述 MBean 的 JavaBean 属性。Attribute 元素可以有选择性的 decriptor 元素和如下属性：

- description. A description of this attribute.该属性的描述
- displayName. The display name of this attribute.属性的显示名称。
- getMethod. The getter method of the property represented by the attribute element.该属性的 get 方法。
- is. A boolean value indicating whether or not this attribute is a boolean with an is getter method. By default, the value of the is attribute is false.一个 Boolean 值，表示该属性是一个有 getter 方法的 boolean 类型。默认的该值为 false。
- name. The name of this JavaBeans property.该 JavaBean 的属性名
- readable. A boolean value indicating whether or not this attribute is readable by management applications. By default, the value of readable is true.boolean 值，用于表示管理程序是否对该属性可读。默认的该值为 true。
- setMethod. The setter method of the property represented by this attribute element.表示该 attribute 元素的 setter 方法。
- type. The fully qualified Java class name of this attribute.该属性的完全限定 Java 类名。
- writeable. A boolean value indicating whether or not this attribute can be written by management applications. By default, this is set to true.boolean 值，用于表示该管理程序是否对该属性进行写操作，默认为 true。

operation

Operation 元素描述暴露给管理程序的 public 方法，它可以有 0 或多个参数子元素，有如下属性：

- description. The description of this operation.操作的描述
- impact. This attribute indicates the impact of this method. The possible values are ACTION (write like), ACTION-INFO (write+read like), INFO (read like), or UNKNOWN.该方法的作用，可选值为 ACTION (write like), ACTION-INFO (write+read like), INFO (read like), or UNKNOWN。
- name. The name of this public method.该 public 方法的名字。
- returnType. The fully qualified Java class name of the return type of this method.返回值类型。

parameter

Parameter 元素用于描述传递给构造函数或操作的参数，它可以有如下属性：

- description. The description of this parameter.对该参数的描述
- name. The name of this parameter.参数名
- type. The fully qualified Java class name of this parameter.参数类型

mbean 元素例子

Catalina 中有很多模型 MBean，在 org.apache.catalina.mbeans 包中的 mbean-descriptors.xml 中声明。Listing20.6 提供了 Tomcat4 中对 StandardServer 的 Mbean 的声明。

Listing 20.6: The declaration of the StandardServer MBean

```
<mbean name="StandardServer"
  className="org.apache.catalina.mbeans.StandardServerMBean"
  description="Standard Server Component"
  domain="Catalina"
  group="Server"
  type="org.apache.catalina.core.StandardServer">

  <attribute name="debug"
    description="The debugging detail level for this component"
    type="int"/>
  <attribute name="managedResource"
    description="The managed resource this MBean is associated with"
    type="java.lang.Object"/>
  <attribute name="port"
```

```
    description="TCP port for shutdown messages"
    type="int"/>
  <attribute name="shutdown"
    description="Shutdown password"
    type="java.lang.String"/>
  <operation name="store"
    description="Save current state to server.xml file"
    impact="ACTION"
    returnType="void">
  </operation>
</mbean>
```

Listing20.6 中的 mbean 元素声明了一个模型 MBean 用于标识 StandardServer。该 MBean 用 org.apache.catalina.mbeans.StandardServerMBean 表示，管理一个 org.apache.catalina.core.StandardServer 类型的对象。域是 Catalina 而组是 Server。

该模型 MBean 暴露了四个属性：debug，managedResource,port 以及 shutdown，这四个属性都是 attribute 元素位于 mbean 元素里面。MBean 还暴露了 store 方法，用 operation 元素表示。

编写自己的模型 MBean 类

当使用常用模型的时候，在 mbean 元素的 className 属性中定义模型 MBean 的类型。默认的常用模型使用 org.apache.commons.modeler.BaseModelMBean 类，但是有些情况你可能想要继承 BaseModelMBean:

1. 想要覆盖管理资源的属性和方法
2. 想要给管理资源添加为定义的属性和方法

Catalina 在 org.apache.catalina.mbeans 包中提供了 BaseModelMBean 类的一些子类，这里对它们进行简单的介绍：

Registry

该 API 的中心在 org.apache.commons.modeler.Registry 类，这里是可以使用该类做的事情：

- Obtain an instance of javax.management.MBeanServer (so you don't need to call the createMBeanServer method of javax.management.MBeanServerFactory).
- Read an mbean descriptor file using the loadRegistry method.

- Create a ManagedBean object that you can use to construct a model MBean.
- 获得一个 javax.management.MBeanServer 实例（不需要调用 javax.management.MBeanServerFactor 的 createMBeanServer 方法）
- 使用 loadRegistry 方法读取描述文件
- 创建一个可用于创建模型 MBean 的 ManagedBean 对象

ManagedBean

ManagedBean 替代 javax.management.MBeanInfo 来表示一个模型 MBean。

BaseModelMBean

org.apache.commons.modeler.BaseModelMBean 类实现了 javax.management.modelmbean.ModelMBean 接口。使用这个类，不需要使用 javax.management.modelmbean.RequiredModelMBean 类。

该类还有一点很有用是该类用一个 resources 属性来表示管理资源。

protected java.lang.Object resource;

使用 Modeler API

你想要管理的 Car 类如 Listing20.7 所示：

Listing 20.7: The Car class

```
package ex20.pyrmont.modelmbeantest2;

public class Car {
  public Car() {
    System.out.println("Car constructor");
  }
  private String color = "red";

  public String getColor() {
    return color;
  }
  public void setColor(String color) {
    this.color = color;
  }

  public void drive() {
    System.out.println("Baby you can drive my car.");
```

```
  }
}
```

使用常用模型，不需要再在属性和操作上面进行编码，而是简单的在 XML 文档中累出来即可。在这个例子中，用于描述它的文档 car-mbean-descriptor.xml 如 Listing20.8 所示

Listing 20.8: The car-mbean-descriptor.xml file

```
<?xml version="1.0"?>
<!DOCTYPE mbeans-descriptors PUBLIC
"-//Apache Software Foundation//DTD Model MBeans Configuration File"
"http://jakarta.apache.org/commons/dtds/mbeans-descriptors.dtd">

<mbeans-descriptors>
  <mbean name="myMBean"
    className="javax.management.modelmbean.RequiredModelMBean"
    description="The ModelMBean that manages our Car object"
    type="ex20.pyrmont.modelmbeantest.Car">

    <attribute name="Color"
      description="The car color"

      type="java.lang.String"/>
    <operation name="drive"
      description="drive method"
      impact="ACTION"
      returnType="void">
      <parameter name="driver" description="the driver parameter"
        type="java.lang.String"/>
    </operation>
  </mbean>
</mbeans-descriptors>
```

Now, you need the agent class (ModelAgent.java) in Listing 20.9.

接下来需要如 Listing20.9 所示的代理类：

Listing 20.9: The ModelAgent Class

```
package ex20.pyrmont.modelmbeantest2;

import java.io.InputStream;
import java.net.URL;
import javax.management.Attribute;
import javax.management.MalformedObjectNameException;
import javax.management.MBeanServer;
import javax.management.ObjectName;
```

```
import javax.management.modelmbean.ModelMBean;

import org.apache.commons.modeler.ManagedBean;
import org.apache.commons.modeler.Registry;

public class ModelAgent {
  private Registry registry;
  private MBeanServer mBeanServer;

  public ModelAgent() {
    registry = createRegistry();
    try {
      mBeanServer = Registry.getServer();
    }
    catch (Throwable t) {
      t.printStackTrace(System.out);
      System.exit(1);
    }
  }

  public MBeanServer getMBeanServer() {
    return mBeanServer;
  }

  public Registry createRegistry() {
    Registry registry = null;
    try {
      URL url = ModelAgent.class.getResource
        ("/ex20/pyrmont/modelmbeantest2/car-mbean-descriptor.xml");
      InputStream stream = url.openStream();
      Registry.loadRegistry(stream);
      stream.close();
      registry = Registry.getRegistry();

  }
  catch (Throwable t) {
    System.out.println(t.toString());
  }
  return (registry);
}

public ModelMBean createModelMBean(String mBeanName)
  throws Exception {
  ManagedBean managed = registry.findManagedBean(mBeanName);
```

```
    if (managed == null) {
      System.out.println("ManagedBean null");
      return null;
    }
    ModelMBean mbean = managed.createMBean();
    ObjectName objectName = createObjectName();
    return mbean;
  }

  private ObjectName createObjectName() {
    ObjectName objectName = null;
    String domain = mBeanServer.getDefaultDomain();
    try {
      objectName = new ObjectName(domain + ":type=MyCar");
    }
    catch (MalformedObjectNameException e) {
      e.printStackTrace();
    }
    return objectName;
  }

  public static void main(String[] args) {
    ModelAgent agent = new ModelAgent();
    MBeanServer mBeanServer = agent.getMBeanServer();
    Car car = new Car();
    System.out.println("Creating ObjectName");
    ObjectName objectName = agent.createObjectName();
    try {
      ModelMBean modelMBean = agent.createModelMBean("myMBean");
      modelMBean.setManagedResource(car, "ObjectReference");
      mBeanServer.registerMBean(modelMBean, objectName);
    }
    catch (Exception e) {
      System.out.println(e.toString());
    }
    // manage the bean
    try {
      Attribute attribute = new Attribute("Color", "green");
      mBeanServer.setAttribute(objectName, attribute);
      String color = (String) mBeanServer.getAttribute(objectName,
        "Color");
      System.out.println("Color:" + color);

      attribute = new Attribute("Color", "blue");
```

```
    mBeanServer.setAttribute(objectName, attribute);
    color = (String) mBeanServer.getAttribute(objectName, "Color");

      System.out.println("Color:" + color);
      mBeanServer.invoke(objectName, "drive", null, null);
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

可以看到使用常用模型编写的代理类简短。

Catalina's MBeans

如在本章开头提到的，Catalina 在 org.apache.catalina.mbeans 包中提供了一些 MBean 类。这些 MBean 类直接或间接的继承 org.apache.commons.modeler.BaseModelMBean 类。本节主要讨论 Tomcat4 中 3 个最重要的 MBean 类:ClassNameMBean, StandardServerMBean,和 MBeanFactory。如果你明白了这三个类，其它的 MBean 类就不难理解。另外本节还将介绍 org.apache.catalina.mbeans 包中的 MBeanUtil 类。

ClassNameMBean 类

org.apache.catalina.mbeans.ClassNameMBean 继承了 org.apache.commons.modeler.BaseModelMBean。它提供了代表管理资源的类名的只写属性 className。该类如 Listing20.10 所示：

Listing 20.10: The ClassNameMBean class

```
package org.apache.catalina.mbeans;

import javax.management.MBeanException;
import javax.management.RuntimeOperationsException;
import org.apache.commons.modeler.BaseModelMBean;

public class ClassNameMBean extends BaseModelMBean {
  public ClassNameMBean()
  throws MBeanException, RuntimeOperationsException {
    super();
  }
  public String getClassName() {
    return (this.resource.getClass().getName());
  }
```

```
}
```

ClassNameMBean 是 BaseModelMBean 类的一个例子，它提供了一个在管理资源中不可用的属性。mbeans-descriptors.xml 文件中的多个 mbean 元素使用这个类作为模型 MBean 的类型。

StandardServerMBean

StandardServerMBean 继承了 org.apache.commons.modeler.BaseModelMBean 类，用于管理 org.apache.catalina.core.StandardServer。如 Listing20.11 的 StandardServerMBean 类是一个模型 MBean 的例子，它覆盖了管理资源的一个方法（store 方法）。当管理程序调用 store 方法的时候，会提交 StandardServerMBean 类中而不是它管理的 StandardServer 中的 store 方法。

Listing 20.11: The StandardServerMBean class

```
package org.apache.catalina.mbeans;

import javax.management.InstanceNotFoundException;
import javax.management.MBeanException;
import javax.management.MBeanServer;
import javax.management.RuntimeOperationsException;
import org.apache.catalina.Server;
import org.apache.catalina.ServerFactory;
import org.apache.catalina.core.StandardServer;
import org.apache.commons.modeler.BaseModelMBean;

public class StandardServerMBean extends BaseModelMBean {
  private static MBeanServer mserver = MBeanUtils.createServer();
  public StandardServerMBean()
    throws MBeanException, RuntimeOperationsException {
    super();
  }

  public synchronized void stored throws InstanceNotFoundException,
    MBeanException, RuntimeOperationsException {

    Server server = ServerFactory.getServer();
    if (server instanceof StandardServer) {
      try {
        ((StandardServer) server).store();
      }
      catch (Exception e) {
        throw new MBeanException(e, "Error updating conf/server.xml");
      }
```

```
    }
  }
}
```

StandardServerMBean 是 BaseModelMBean 的子类，它覆盖管理资源中的方法。

MBeanFactory

MBeanFactory 表示用于创建模型 MBean 的工厂对象，它管理 Catalina 中各种资源。MBeanFactory 类还提供了用于删除这些 MBeans。

As an example, take a look at the createStandardContext method in Listing 20.12.

Listing 20.12: The createStandardContext method

```
public String createStandardContext(String parent,
  String path, String docBase) throws Exception {
  // Create a new StandardContext instance
  StandardContext context = new StandardContext();
  path = getPathStr(path);
  context.setPath(path);
  context.setDocBase(docBase);
  ContextConfig contextConfig = new ContextConfig();
  context.addLifecycleListener(contextConfig);

  // Add the new instance to its parent component
  ObjectName pname = new ObjectName(parent);
  Server server = ServerFactory.getServer();
  Service service =
    server.findService(pname.getKeyProperty("service"));
  Engine engine = (Engine) service.getContainer();
  Host host = (Host) engine.findChild(pname.getKeyProperty("host"));

  // Add context to the host
  host.addChild(context);

  // Return the corresponding MBean name
  ManagedBean managed = registry.findManagedBean("StandardContext");

  ObjectName oname =
    MBeanUtils.createObjectName(managed.getDomain(), context);
  return (oname.toString());
}
```

MBeanUtil

org.apache.catalina.mbeans.MBeanUtil 类是一个工具类，它提供了静态方法来创建 Mbean 以管理 Catalina 对象、删除 MBean 的静态方法以及创建对象名的静态方法。例如，如 Listing20.13 所示的 createMBean 方法创建一个 org.apache.catalina.Server 对象的模型 MBean。

Listing 20.13: The createMBean method that creates a model MBean that manages a Server object.

```
 public static ModelMBean createMBean(Server server) throws Exception {
  String mname = createManagedName(server);
  ManagedBean managed = registry.findManagedBean(mname);
  if (managed == null) {
    Exception e = new Exception(
      "ManagedBean is not found with "+mname);
    throw new MBeanException(e);
  }
  String domain = managed.getDomain();
  if (domain == null)
    domain = mserver.getDefaultDomain();
  ModelMBean mbean = managed.createMBean(server);
  ObjectName oname = createObjectName(domain, server);
  mserver.registerMBean(mbean, oname);
  return (mbean);
}
```

Catalina 创建 MBean

现在熟悉了 Catalina 中的模型 MBean，接下来看一下是怎么样创建这些 MBean 来管理应用程序的。

Tomcat 的配置文件 server.xml 在 Server 元素中定义了 Listener 元素：

```
<Server port="8005" shutdown="SHUTDOWN" debug="0">
  <Listener
    className="org.apache.catalina.mbeans.ServerLifecycleListener"
    debug="0"/>
...
```

它将给 org.apache.catalina.core.StandardServer 添加一个 org.apache.catalina.mbeans.ServerLifecycleListener 类型的监听器，当 StandardServer 实例启动的时候，会触发一个 START_EVENT 事件，如 StandardServer 类中定义的那样：

```
public void start() throws LifecycleException {
  ...
```

```
  lifecycle.fireLifecycleEvent(START_EVENT, null);
  ...
}
```

StandardServer 对象停止的时候，会触发 STOP_EVENT 事件，如 stop 方法中定义的那样：

```
public void stop() throws LifecycleException {
  ...
  lifecycle.fireLifecycleEvent(STOP_EVENT, null);
  ...
}
```

这些事件会导致 ServerLifecycleListener 中的 lifecycleEvent 方法被提交。Listing20.14 展示了 lifecycleEvent 方法

Listing 20.14: The lifecycleEvent method of the ServerLifecycleListener class

```
public void lifecycleEvent(LifecycleEvent event) {
  Lifecycle lifecycle = event.getLifecycle();
  if (Lifecycle.START_EVENT.equals(event.getType())) {
    if (lifecycle instanceof Server) {
      // Loading additional MBean descriptors
      loadMBeanDescriptors();
      createMBeans();
    }
  }
  else if (Lifecycle.STOP_EVENT.equals(event.getType())) {
    if (lifecycle instanceof Server) {
      destroyMBeans();
    }
  }
  else if (Context.RELOAD_EVENT.equals(event.getType())) {
    if (lifecycle instanceof StandardContext) {
      StandardContext context = (StandardContext)lifecycle;
      if (context.getPrivileged()) {
        context.getServletContext().setAttribute
          (Globals.MBEAN_REGISTRY_ATTR,
        MBeanUtils.createRegistry());
        context.getServletContext().setAttribute
          (Globals.MBEAN_SERVER_ATTR,
        MBeanUtils.createServer());
      }
    }
  }
}
```

createMBeans 是创建 Catalina 中所有 MBeans 方法。该方法首先创建一个

MBeanFactory：

Listing 20.15: The createMBeans method in ServerLifecycleListener

```
protected void createMBeans() {
  try {
    MBeanFactory factory = new MBeanFactory();
    createMBeans(factory);
    createMBeans(serverFactory.getserver());
  }
  catch (MBeanException t) {
    Exception e = t.getTargetException();
    if (e == null)
      e = t;
    log("createMBeans: MBeanException", e);
  }
  catch (Throwable t) {
    log("createMBeans: Throwable", t);
  }
}
```

第一个 createMBeans 方法使用 MBeanUtil 类给 MBeanFactory 创建一个 ObjectName 并将其向 MBean 服务器注册。

第二个 createMBeans 方法有一个 org.apache.catalina.Server 对象并为其创建模型 MBean。阅读如 Listing20.16 所示的 createMBeans 方法的代码很有趣

Listing 20.16: The createMBeans method that creates an MBean for a Server object

```
protected void createMBeans(Server server) throws Exception {
  // Create the MBean for the Server itself
  if (debug >= 2)
    log("Creating MBean for Server " + server);
  MBeanUtils.createMBean(server);
  if (server instanceof StandardServer) {
    ((StandardServer) server).addPropertyChangeListener(this);
  }

  // Create the MBeans for the global NamingResources (if any)
  NamingResources resources = server.getGlobalNamingResources();
  if (resources != null) {
    createMBeans(resources);
  }

  // Create the MBeans for each child Service
  Service services[] = server.findServices();
  for (int i = 0; i < services.length; i++) {
```

```
    // FIXME - Warp object hierarchy not currently supported
    if (services[i].getContainer().getClass().getName().equals
      ("org.apache.catalina.connector.warp.WarpEngine")) {
      if (debug >= 1) {

        log("Skipping MBean for Service " + services[i]);
      }
      continue;
    }
    createMBeans(services[i]);
  }
}
```

注意如 Listing20.16 所示的 createMBeans 方法使用 for 循环来迭代 StandardServer 实例中的所有 Service 对象。

```
createMBeans(services[i]);
```

该方法为服务（Service）创建一个 MBean 实例并调用 createMBeans 方法来为该服务所有的连接器和引擎创建 MBean 对象。用于创建 Service 的 MBean 的 createMBeans 方法如 Listing20.17 所示：

Listing 20.17: The createMBeans method that creates a Service MBean

```
protected void createMBeans(Service service) throws Exception {
  // Create the MBean for the Service itself
  if (debug >= 2)
    log("Creating MBean for Service " + service);
  MBeanUtils.createMBean(service);
  if (service instanceof StandardService) {
   ((StandardService) service).addPropertyChangeListener(this);
  }

  // Create the MBeans for the corresponding Connectors
  Connector connectors[] = service.findConnectors();
  for (int j = 0; j < connectors.length; j++) {
    createMBeans(connectors[j]);
  }

  // Create the MBean for the associated Engine and friends
  Engine engine = (Engine) service.getContainer();
  if (engine != null) {
    createMBeans(engine);
  }
}
```

createMBeans (engine)调用为主机创建 MBeans 的 createMBeans 方法。

```
protected void createMBeans(Engine engine) throws Exception {
  // Create the MBean for the Engine itself
  if (debug >= 2) {
    log("Creating MBean for Engine " + engine);
  }
  MBeanUtils.createMBean(engine);
  ...

  Container hosts[] = engine.findChildren();
  for (int j = 0; j < hosts.length; j++) {
    createMBeans((Host) hosts[j]);
  }
  ...
}
```

The createMBeans (host) method in turns creates a ContextMBean, like the following:

createMBeans 方法又创建 ContextMBean，如下：

```
protected void createMBeans(Host host) throws Exception {
  ...
  MBeanUtils.createMBean(host);
  ...
  Container contexts[] = host.findChildren();
  for (int k = 0; k < contexts.length; k++) {
    createMBeans((Context) contexts[k]);
  }
  ...
}
```

The createMBeans (context) method is as follows:

createMBeans(context)如下所示：

```
protected void createMBeans(Context context) throws Exception {
  ...
  MBeanUtils.createMBean(context);
  ...
  context.addContainerListener(this);
  if (context instanceof StandardContext) {
   ((StandardContext) context).addPropertyChangeListener(this);
   ((StandardContext) context).addLifecycleListener(this);
  }

  // If the context is privileged, give a reference to it
  // in a servlet context attribute
  if (context.getPrivileged()) {
```

```
    context.getServletContext().setAttribute
      (Globals.MBEAN_REGISTRY_ATTR, MBeanUtils.createRegistry());
    context.getServletContext().setAttribute
      (Globals.MBEAN_SERVER_ATTR, MBeanUtils.createServer());
  }
  ...
}
```

如果上下文的privileged属性为真,会给该web应用程序创建和存储两个属性。属性的键为Globals.MBEAN_REGISTRY_ATTR和Globals.MBEAN_SERVER_ATTR。下面是org.apache.catalina.Globals类中的代码片段:

```
/**
* The servlet context attribute under which the managed bean Registry
* will be stored for privileged contexts (if enabled).
*/

public static final String MBEAN_REGISTRY_ATTR =
  "org.apache.catalina.Registry";

/**
* The servlet context attribute under which the MBeanServer will be
* stored for privileged contexts (if enabled).
*/
public static final String MBEAN_SERVER_ATTR =
  "org.apache.catalina.MBeanServer";
```

MBeanUtils.createRegistry返回一个Registry实例。MBeanUtils.createServer方法返回一个javax.management.MBeanServer实例,所有的Catalina的MBean都在上面注册。

另一句话说,可以获得privileged属性为真的web应验的Reigstry 和MBeanServer的实例。下面一节将会讨论如何使用JMX管理应用程序来管理Tomcat。

应用程序

这里的应用程序时一个用于管理Tomcat的应用程序。它很简单但是足够你认识怎么使用MBeans暴露的Catalina。可以使用它它列出Catalina中所有的ObjectName实例,列出当前运行的所有上下文以及删除它们。

首先,需要为该应用程序创建一个描述文件如Listing20.18所示。必须将该文件放在%CATALINA_HOME%/webapps目录下。

Listing 20.18: The myadmin.xml file

```
<Context path="/myadmin" docBase="../server/webapps/myadmin" debug="8"
privileged="true" reloadable="true">
</Context>
```

你需要关系的一件事是确保 Context 元素的 privileged 属性为真。docBase 属性被设置成该应用程序的地址。

该应用程序由一个 Servlet，如 Listing20.19 所示

Listing 20.19: The MyAdminServlet class

```
package myadmin;

import java.io.IOException;

import java.io.PrintWriter;
import java.net.URLEncoder;
import java.util.Iterator;
import java.util.Set;
import javax.management.MBeanServer;
import javax.management.ObjectName;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.commons.modeler.Registry;

public class MyAdminServlet extends HttpServlet {
  private Registry registry;
  private MBeanServer mBeanServer;

  public void init() throws ServletException {
    registry = (Registry)
      getServletContext().getAttribute("org.apache.catalina.Registry"
);
    if(registry == null) {
      System.out.println("Registry not available");
      return;
    }
    mBeanServer = (MBeanServer) getServletContext().getAttribute(
      "org.apache.catalina.MBeanServer");
    if (mBeanServer==null) {
      System.out.println("MBeanServer not available");
      return;
    }
  }

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

```
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    if (registry==null || mBeanServer==null) {
      out.println("Registry or MBeanServer not found");
      return;
    }

    out.println("<html><head></head><body>");
    String action = request.getParameter("action");
    if ("listAllManagedBeans".equals(action)) {
      listAllManagedBeans(out);
    }
    else if ("listAllContexts".equals(action)) {
      listAllContexts(out);
    }
    else if ("removeContext".equals(action)) {
      String contextObjectName =
        request.getParameter("contextObjectName");
      removeContext(contextObjectName, out);
    }
    else {

      out.println("Invalid command");
    }
    out.println("</body></html>");
  }

  private void listAllManagedBeans(PrintWriter out) {
    String[] managedBeanNames = registry.findManagedBeans();
    for (int i=0; i<managedBeanNames.length; i++) {
      out.print(managedBeanNames[i] + "<br/>");
    }
  }

  private void listAllContexts(PrintWriter out) {
    try {
      ObjectName objName = new ObjectName("Catalina:type=Context,*");
      Set set = mBeanServer.queryNames(objName, null);
      Iterator it = set.iterator();
      while (it.hasNext()) {
        ObjectName obj = (ObjectName) it.next();
        out.print(obj +
          " <a href=?action=removeContext&contextObjectName=" +
          URLEncoder.encode (obj.toString(), "UTF-8") +
```

```
            ">remove</a><br/>");
        }
      }
      catch (Exception e) {
        out.print(e.toString());
      }
    }

    private void removeContext(String contextObjectName,
      PrintWriter out) {
      try {
        ObjectName mBeanFactoryObjectName = new
          ObjectName("Catalina:type=MBeanFactory");
        if (mBeanFactoryObjectName!=null) {
          String operation = "removeContext";
          String[] params = new String[1];
          params[0] = contextObjectName;
          String signature[] = { "java.lang.String" };
          try {
            mBeanServer.invoke(mBeanFactoryObjectName, operation,
              params, signature);
            out.println("context removed");
          }
          catch (Exception e) {
            out.print(e.toString());
          }
        }

      }
      catch (Exception e) {
      }
    }
}
```

最后，需要如Listing20.20所示的应用部署文件。

Listing 20.20: The web.xml file

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

<!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>
  <servlet>
```

```
    <servlet-name>myAdmin</servlet-name>
    <servlet-class>myadmin.MyAdminServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>myAdmin</servlet-name>
    <url-pattern>/myAdmin</url-pattern>
  </servlet-mapping>
</web-app>
```

要列出所有的 ObjectName 实例，可以使用如下 URL

http://localhost:8080/myadmin/myAdmin?action=listAllMBeans

可以看到一系列的 MBean 对象，下面是前六个：

MemoryUserDatabase
DigestAuthenticator
BasicAuthenticator
UserDatabaseRealm
SystemErrLogger
Group

可以使用如下 URL 列出所有上下文容器：

http://localhost:8080/myadmin/myAdmin?action=listAllContexts

可以看到所有的运行中的应用，可以点击 remove 超链接删除它们

总结

在本章中学习了如何使用 JMX 来管理 Tomcat。本章介绍了两种类型的 MBeans 并展示了一个简单使用 MBeans 来管理 Catalina 的应用程序。