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。

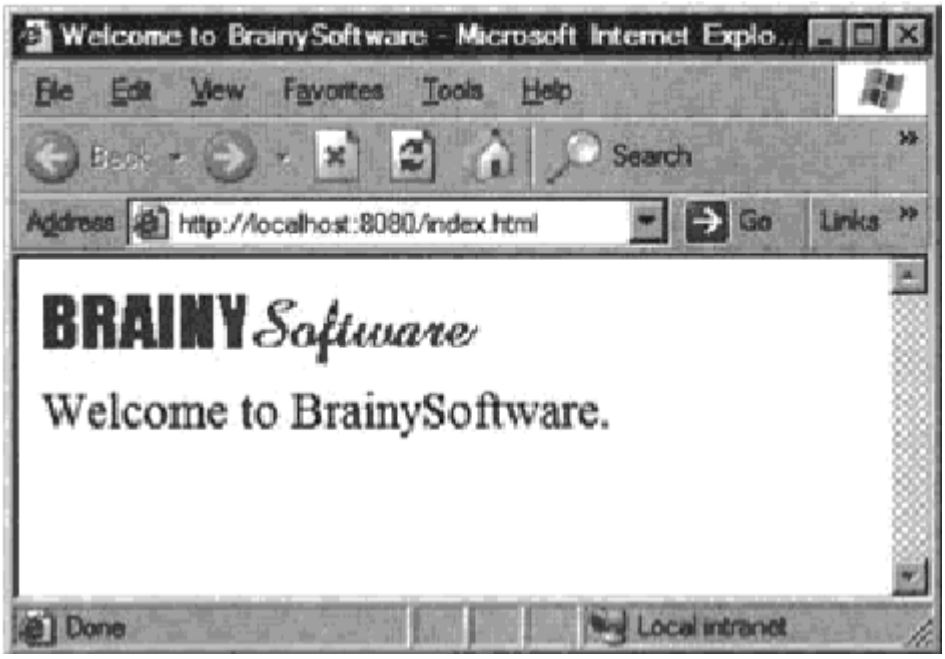


Figure 1.1: web 服务器的输出

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。

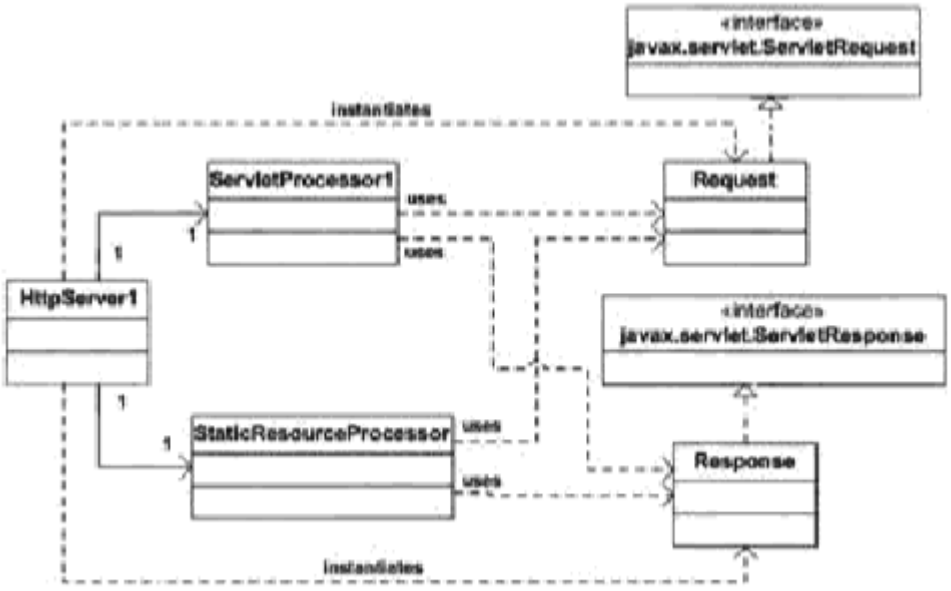


Figure 2.1: 第一个 servlet 容器的 UML 图

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。

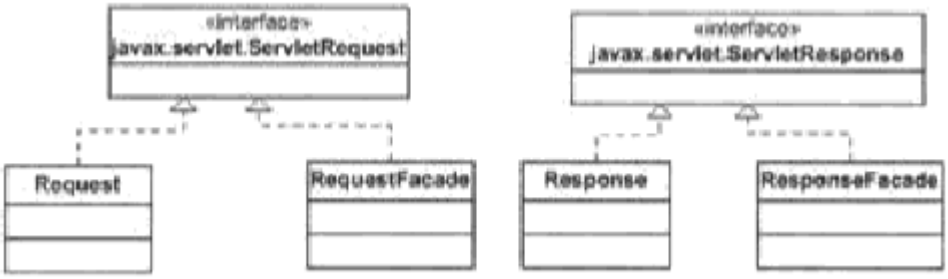
```
try {
    servlet = (Servlet) myClass.newInstance();
    servlet.service((ServletRequest) request,(ServletResponse)
```

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。

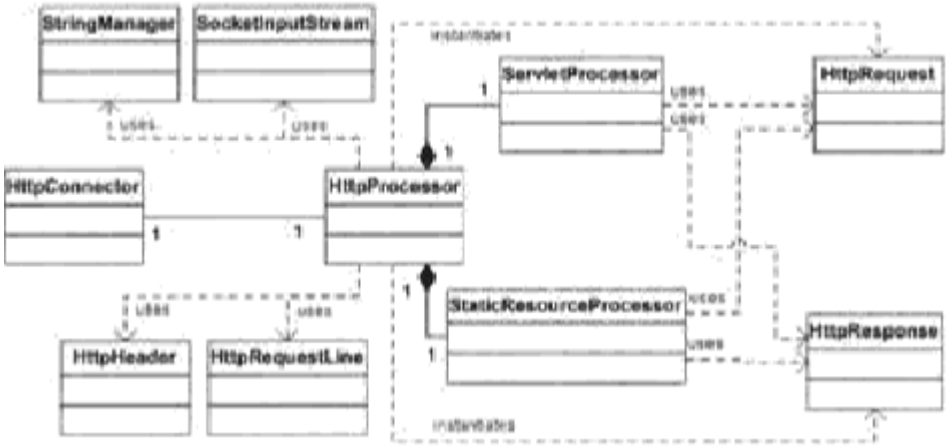


Figure 3.1: 应用程序的 UML 图

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：

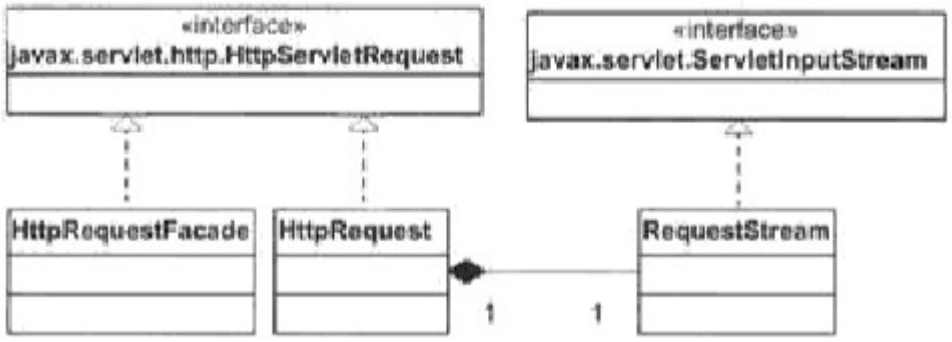
```
protected StringManager sm =
    StringManager.getManager("ex03.pyrmont.connector.http");
```

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.parseHeader
s.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。

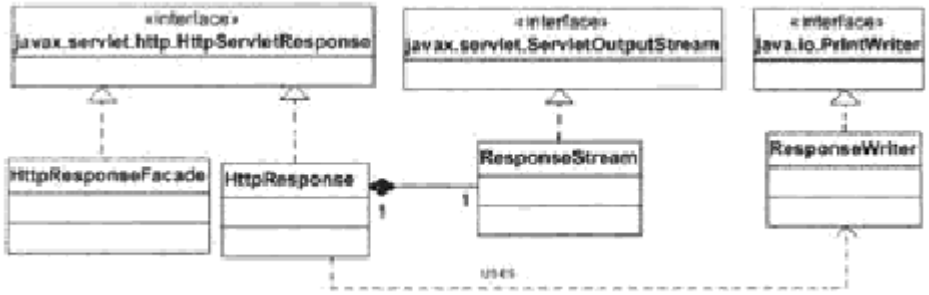


Figure 3.3: The HttpResponse class and related classes

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

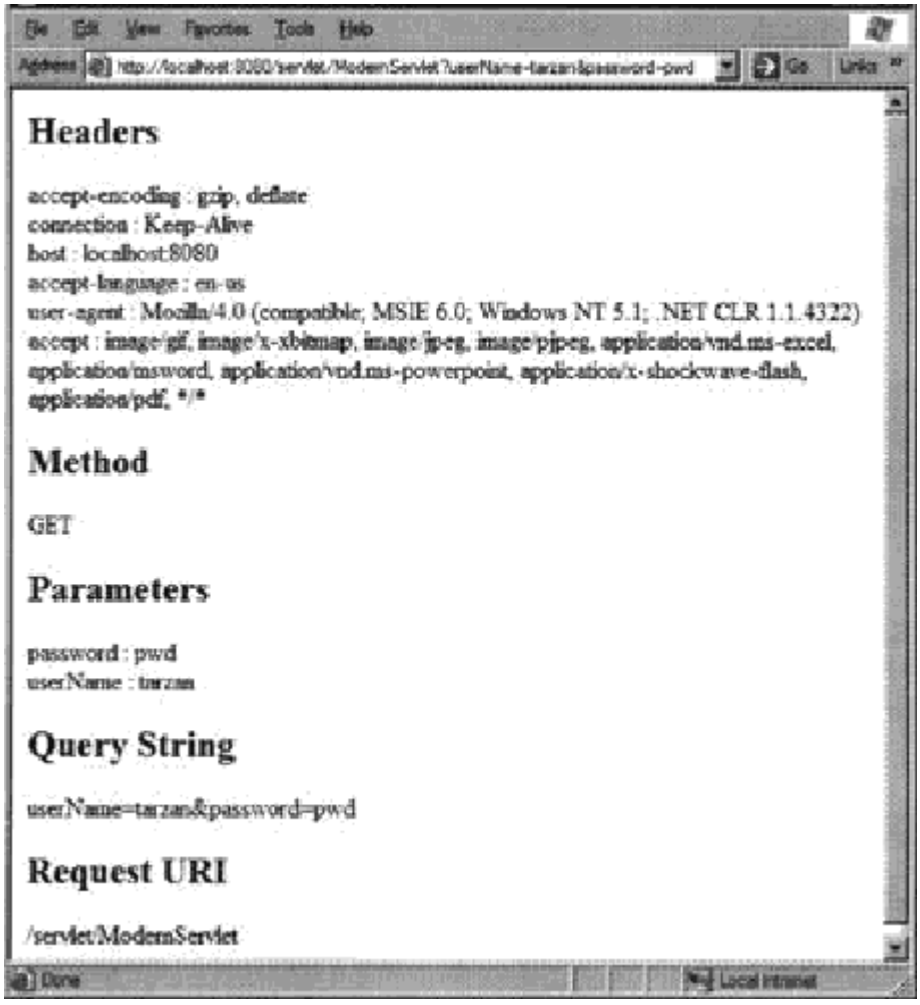


Figure 3.4: Running ModernServlet

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。

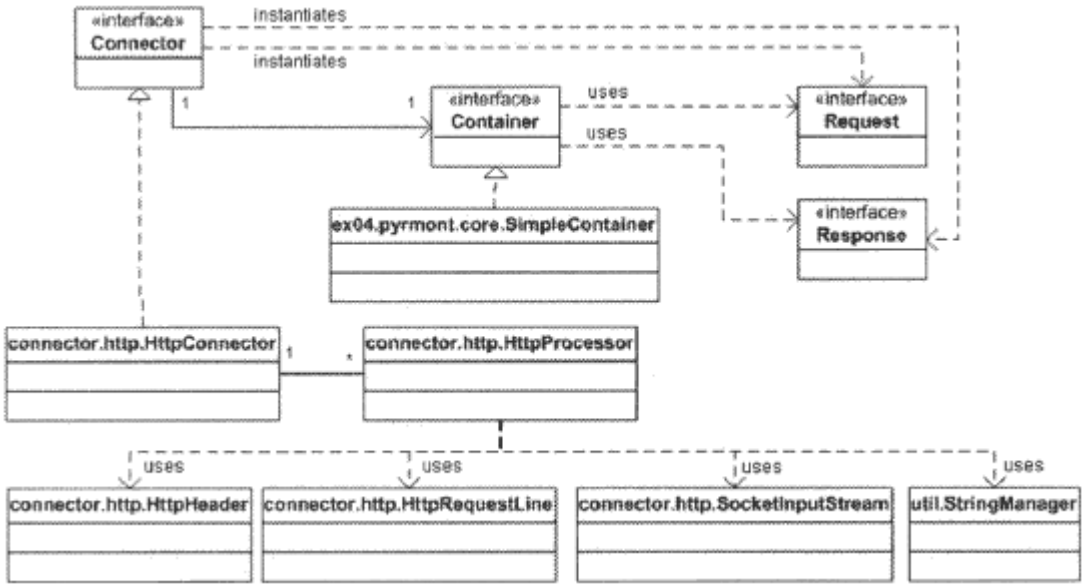


Figure 4.1: The default connector class diagram

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。

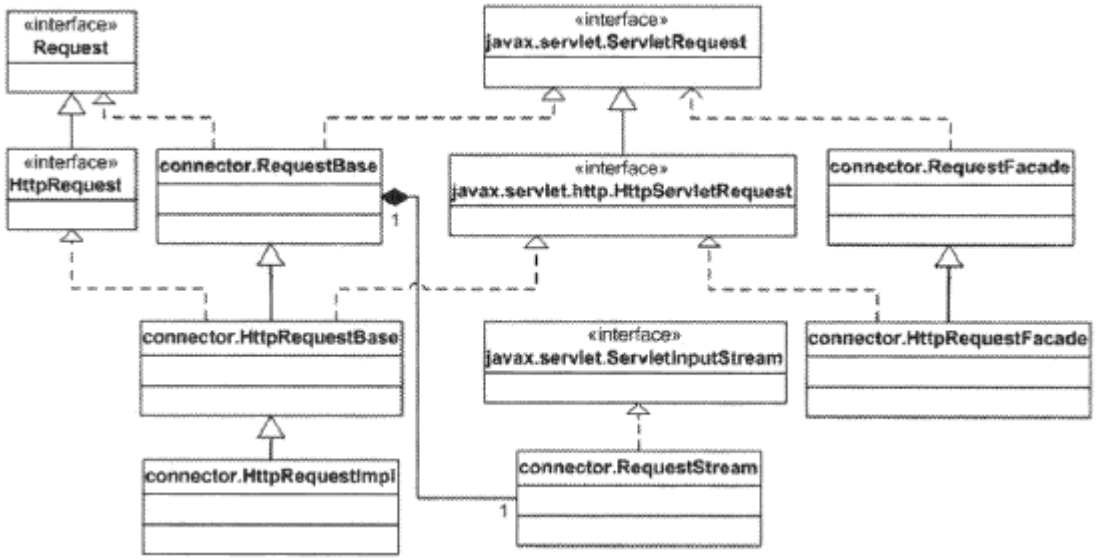


Figure 4.2: The Request interface and related types

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。

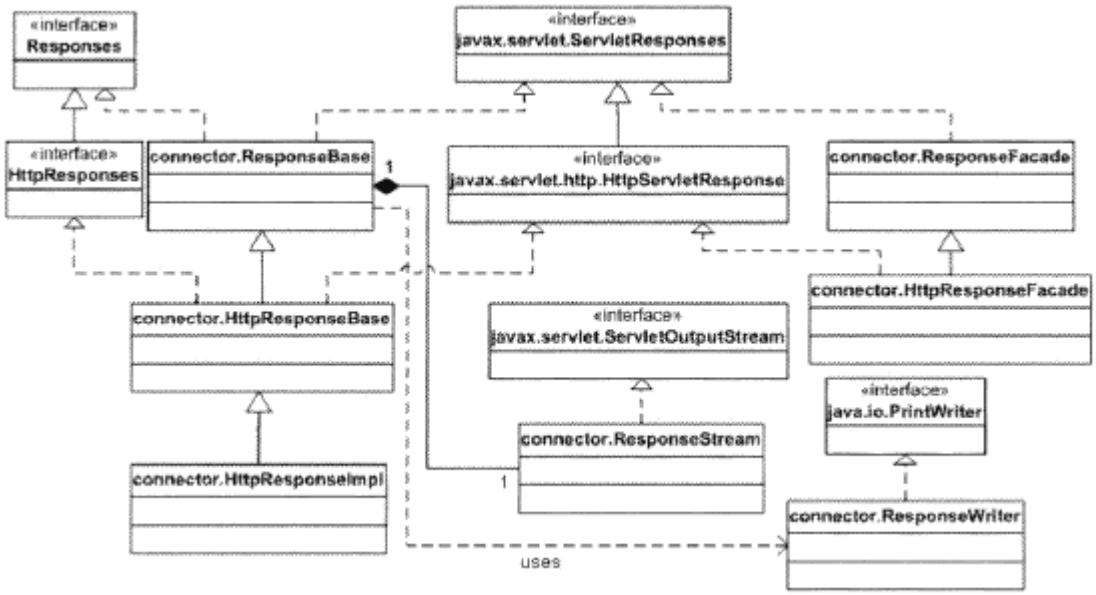


Figure 4.3: The Response interface and its implementation classes

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 removovPropertyChangeListener(
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。

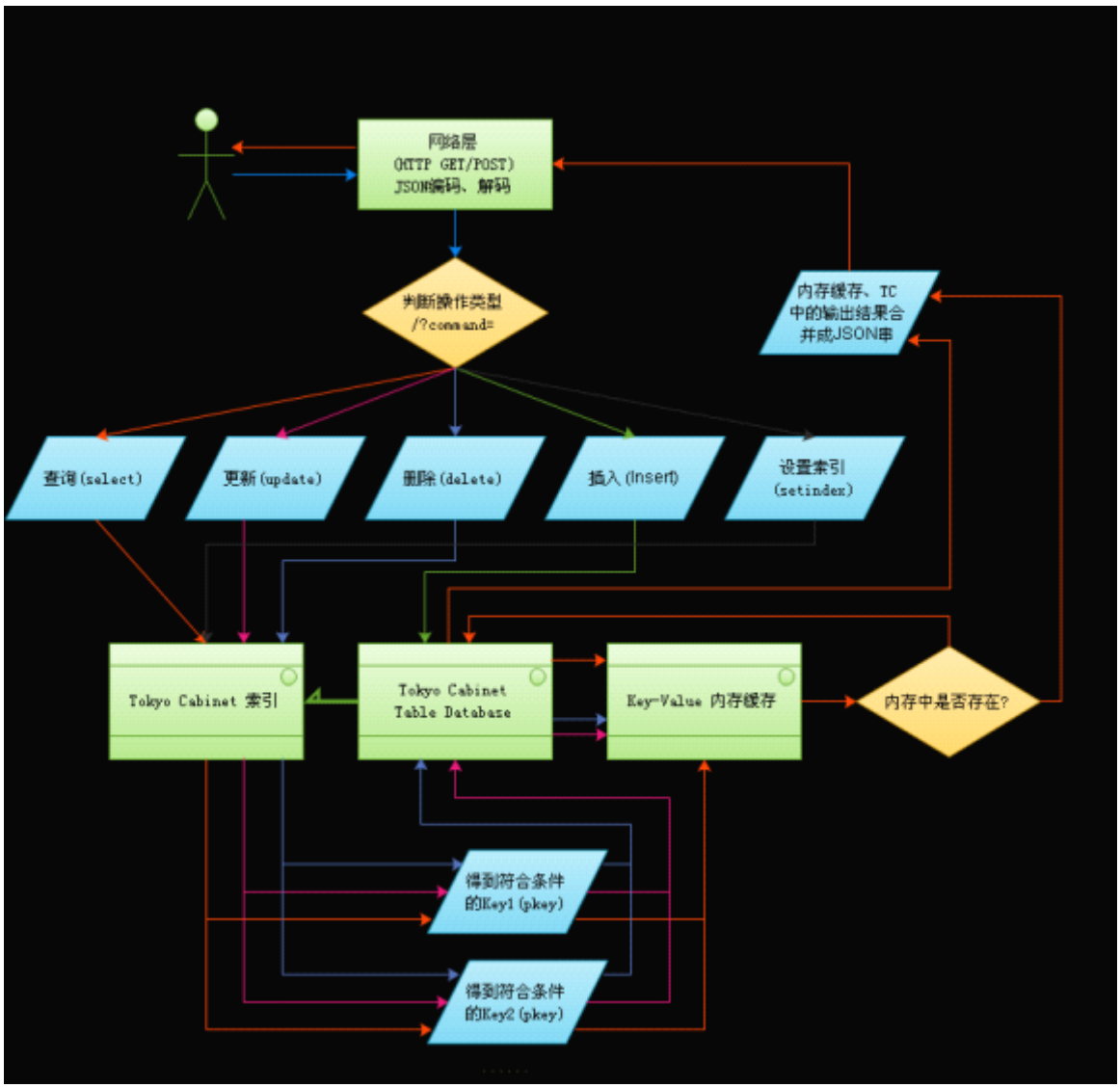
```
java -classpath ./lib/servlet.jar:./ ex04.pyrmont.startup.Bootstrap
```

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。


网络层
(HTTP GET/POST)
JSON编码、解码
判断操作类型
/?command=
内存缓存、TC
中的输出结果合
并成JSON串
查询(select)
更新(update)
删除(delete)
插入(Insert)
设置索引
(setindex)
Tokyo Cabinet 索引
Tokyo Cabinet
Table Database
Key-Value 内存缓存
内存中是否存在?
得到符合条件
的Key1(pkey)
得到符合条件
的Key2(pkey)