

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.ServletException或javax.servlet.http.HttpServletRequest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.HttpServletResponse或javax.servlet.http.HttpServletResponse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request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。

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。你也将要写两个程序，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

计算机是Linux机器，敲入以下内容：(如有必要的话不要忘记使用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，它定义于征求意见稿(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.getBy_name("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Request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类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，但是你将会在以后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 {
            StaticResourceProcessor 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 UnsupportedOperationException { }
}

```

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.ServletException;
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 handler)
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.HttpServletRequest，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.HttpServletResponse，并作为第二个参数传递给servlet的service方法。

```
try {  
    servlet = (Servlet) myClass.newInstance();  
    servlet.service((HttpServletRequest) request, (HttpServletResponse) 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HttpServletRequest和HttpServletResponse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 ServletRequest 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().startsWith("/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 HttpServletResponse instance and assigns the HttpRequest to the HttpServletResponse.

```

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

```

本章应用程序的HttpServletResponse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：

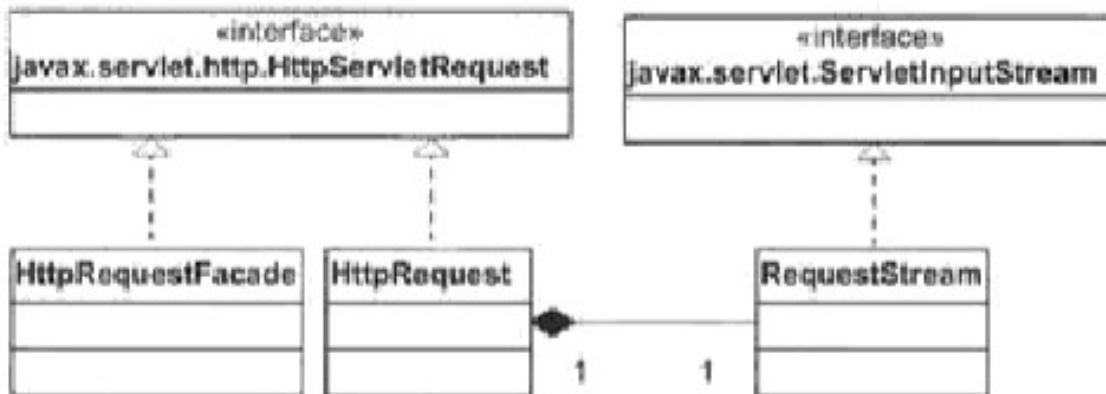
```
protected StringManager sm =
    StringManager.getManager("ex03.pyrmont.connector.http");
```

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类和相关类的UML图。

Figure 3.2: HttpRequest类和相关类



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, getParameterMap, 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`jsessionid`，方法就取得会话标识符，并通过调用`setRequestedSessionId`方法把值交给`HttpRequest`实例：

```

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

```

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

到这个时候，`uri`的值已经被去掉了`jsessionid`。

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

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

```

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

```

```
((HttpRequest) request).setRequestURI(uri);
}
```

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

```
if (normalizedUri == null) {
    throw new ServletException("Invalid URI: " + uri + "");
}
```

解析头部

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

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

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

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

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

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

假如存在下一个头部，那么头部的名称和值可以通过下面方法进行检索：

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

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

中：

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

一些头部也需要某些属性的设置。例如，当servlet调用javax.servlet.ServletRequest的getLength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() < 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

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，假如已经解析过他的话，`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HttpServletResponse对象

HttpServletResponse类实现了javax.servlet.http.HttpServletResponse。跟随它的是一个叫做HttpServletResponseFacade的facade类。Figure 3.3显示了HttpServletResponse类和相关类的UML图。



在第2章中，你使用的是一个部分实现的HttpServletResponse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.HttpServletResponse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ModernServlet，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`HttpProcessor`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HttpConnector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)
<pre>while (!available) { wait(); } Socket socket = this.socket; available = false; notifyAll(); return socket; // to the run // method</pre>	<pre>while (available) { wait(); } this.socket = socket; available = true; notifyAll(); ...</pre>

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Listing 4.2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`HttpHeader`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 removePropertyChangeListener(
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connceotor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。

[Google Docs -- Web word processing, presentations and spreadsheets.](#)

[Edit this page \(if you have permission\)](#) |