http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508
,子午学术论坛的“图书交流版”，现在已经分享了 4 万多份 IT、技术、英语和经济相关的图书资料。欢迎大家前来参与交流。
推荐的链接中有我的 ID 号，愿意的话，就请您用这个链接来注册，为我增加一个贡献值。

另外，尤其值得关注的是，ziwu 论坛的图书求助区，聚集了众多高手可以免费帮你分享一些优秀的电子资料。

想得到更多好书的话，就请来 http://www.ziwu.org/bbs/?u=39508

多谢您的参与和帮助。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

第 1 章　XML 基础

作为一种通用的数据交换格式，XML 的应用越来越广泛。目前许多系统的配置文件都使用 XML 格式，本书中所涉及的众多配置文件都是 XML 格式的，新的 JSP 规范也推出了采用 XML 语法的 JSP 文档。因此，要深入学习和掌握 Java Web 开发技术，必须有一些基本的 XML 语言知识。由于 XML 的通用性和广泛应用，在读者以后实际项目开发和应用中，很多地方都要用到 XML 方面的知识，所以，对 XML 知识的了解和掌握，已经成为从事电子商务网站和相关系统开发的编程人员的基本要求。

指点迷津

什么是配置文件

配置文件就是记录应用程序的配置信息的文件。例如，某个程序要连接网络上的数据库系统时，需要指定数据库服务器的 ip 地址、服务器的网络监听端口号、数据库的名称、用户名和密码等信息，这些信息可以作为应用程序的配置信息来对待，即不要将这些信息写死在源程序中，而是写在一个文本文件中。应用程序要连接数据库系统时，从该文本文件中读取所需要的信息，如果连接的数据库系统的信息发生了改变，则只需要修改该文本文件，而不用修改源程序，这个文本文件就是应用程序的配置文件。传统的配置文件通常采用如下格式：

```
#Db.Properties
DBServer=192.168.0.100
Port=1443
Database=emp
Username=zxx
Password=123
```

上面这种格式的配置文件只能表述一些简单的信息，而很难表达具有层次结构和数据项目要动态增加的信息。例如，如果要在一个文件中表达出“一个国家中的每个省及省长的名称、每个省下面的每个市及市长的名称、每个市下面的每个镇及镇长名称”，则使用上面的格式就很难表达。采用 XML 格式的配置文件很容易表达出这种层次结构，具体细节请参看本章后面部分的讲解。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

1.1 XML 技术概述

1.1.1 XML 的起源与作用

几个世纪以来，人类在商务活动中已经成功地采用了标准化的纸张文档来描述和交换信息，例如采购订单、发票、货物清单、收据等。这些标准文档能够在商务活动中起作用，是因为它所描述的信息能够被发送者和接收者都很好地理解。传统的商务文档是人脑制作出来，并被人脑识别和解释。随着 Internet 的迅猛发展，越来越多的商务活动都将在 Internet 上开展，在线进行商务活动同样也离不开交换文档的方式，但其所交换的是电子文档。电子文档需要计算机程序来产生，且被计算机程序自动识别并处理。譬如，采购方程序产生的一份采购订单电子文档，能够被供货商程序自动提取出采购信息并通过查询产品信息数据库，又返回给采购方程序一份报价清单电子文档；采购方程序能够从各个供货商返回的报价清单中提取报价信息，挑选报价最低的供货商。所有这一切都需要计算机程序自动处理，信息交换双方所采用的电子文档格式必须是双方都能理解和识别的，让软件具有解读文义的能力，这就需要电子文档遵循某一标准。可见，统一电子文档的标准规范正是电子商务的基础，而电子商务的发展迫切需要统一的电子文档交换标准。

虽然 HTML 也是一种电子文档的标准规范，但它并不是为此项任务而创建的，HTML 并不是来描述数据信息的，它将数据和显示效果混在一起。随着 Web 应用的迅猛发展和电子商务的兴起，必须有一种方法能够把数据和它的显示分离开来，这样就导致了 XML 的诞生。XML 是 eXtensible Markup Language（可扩展标记语言）的缩写，是为了实现计算机之间的文档交换而设计的文档内容编写规范，它的语法与 HTML 非常相似。

XML 和 HTML 一样，都衍生于 SGML（Standard Generalized Markup Language，标准通用化标记语言）。SGML 从 20 世纪 80 年代初开始使用，它可用于创建成千上万的标记，具有极好的扩展性，在分类和索引数据中非常有用，多用于科技文献和政府办公文件中。SGML 有非常强大的适应性，同时也非常复杂，其复杂程度无法适应网络上的日常应用，以致在小型的应用中难以普及。XML 的产生就是为了简化 SGML，以便用于更加通用的目的。W3C 于 1998 年 2 月批准了 XML 的 1.0 版本，XML 既具有 SGML 的强大功能和可扩展性，同时又具有 HTML 的简单性。它是一个非常简单的 SGML 语言，方便普通人的使用。可以认为，XML 是 SGML 的一个子集，而 HTML 则是 SGML 的一个应用。

XML 文档很容易创建，并且结构清晰，非常便于阅读理解。XML 的根本作用就是要统一信息的结构，使文档对人和机器都可阅读，使越来越多的信息不仅让人能够明白，还让计算机也能够明白，并且实现不同系统之间的相互通信。需要额外提醒读者，发送者在发送文档之前是通过哪些环节制作出文档的，而接收者在接收到文档之后在内部又是怎样处理的，这都不是 XML 本身所要关心的问题。要想用程序来处理各个商业公司的电子文档和获取其中的信息，编程人员必须了解每个文档的信息存储格式。XML 作为一种公订的、开放的标准，它为各商业公司的电子文档统一了格式，不受版权、专利、商业秘密或

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

是其他种类的知识产权的限制，人人都可以拿它来开发软件。目前存在着许多成熟的 XML 开发包和应用工具，人们直接使用这些开发包和应用工具，将极大地简化编程的处理过程，使得 XML 相关的编程和应用非常简单。因为 XML 的众多优点，使得它成为在不同的应用间交换数据的理想格式，XML 文档变得越来越通用，不少公司都开始以 XML 为基础来实施它们的 EDI（Electronic Data Interchange）系统。

1.1.2 XML 文档结构

XML 的语法和 HTML 类似，也是一种标记语言，它通过使用标记来说明文本的意义和所代表的信息。因此，有了 HTML 的基础，学习 XML 将非常简单和容易。XML 文档的后缀名通常为.xml，但这不等于所有采用 XML 语法的文件的扩展名都必须是.xml。下面是一个 XML 文档的内容：

```
<书架>
    <书>
        <书名>Java 就业培训教程</书名>
        <作者>张孝祥</作者>
        <售价>39.00 元</售价>
    </书>
    <书>
        <书名>JavaScript 网页开发</书名>
        <作者>张孝祥</作者>
        <售价>28.00 元</售价>
    </书>
</书架>
```

其中<书架>、<书>、<书名>等都是用户自己创建的标记。标记可帮助说明文本的内涵和所表示的信息，例如，没有使用标记时，人们（特别是计算机软件）难以知道一段文本内容中的“张孝祥”是什么信息，但使用<作者>进行标记后，就可以清楚地知道“张孝祥”表示一个作者的姓名。在 HTML 文档中，只能使用 HTML 规范中定义好的标记，不能使用 HTML 规范里没有的标记，所以，HTML 是一个定型的标记语言，它用固有的标记来描述、显示网页内容。相反，XML 不能描述网页具体的外观，它只是描述数据的结构和内容，它没有固定的标记，允许用户随便发明和创建自己的标记，标记名可以是非英文字符（如上面 XML 文档片断的标记名采用的就是中文字符），这就是 XML 的可扩展性。

XML 文档是树型结构的，在上面的 XML 片段中，最上层的元素是<书架>，在<书架>的下层又定义了一个名为<书>的子元素，在<书>下层又定义了三个子元素：<书名>、<作者>和<售价>。上面的 XML 片段的结构如下：

```
<书架>
    <书>
        <书名> </书名>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        <作者> </作者>
        <售价> </售价>
    </书>
    </书架>
```

上面的 XML 片段的结构可以用图 1.1 来描述，这种“树”状的结构即称为文档结构树。

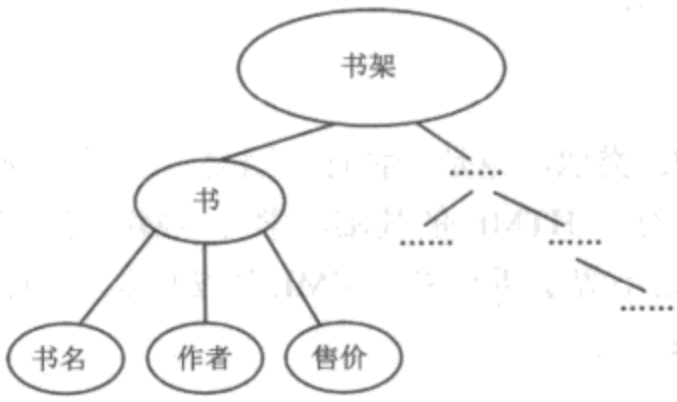


图 1.1

可见，XML 也是结构化的，其结构化主要体现在以下两个方面：

（1）每一部分（每一个元素）都和其他元素有关联，这种关联就形成了树结构。

（2）标记本身的含义与它描述的信息相分离。

由于 XML 文档是树状结构的，所以它可以很好地描述具有树状层次结构的复杂信息。例如，描述一台计算机上有哪几个分区，每个分区的大小及它下面直接包含哪些子目录和文件，每个子目录中又包含哪些子目录和文件，以及每个文件的创建日期、修改日期、大小、访问权限等这样复杂的配置信息。并且由于 XML 是一种通用和标准的文档格式，可以采用现成的 XML API 来读取其中的信息，所以，越来越多的应用程序都开始采用 XML 文档来存放有关的配置信息。本书所讲解的 Tomcat 服务器和在其中部署的 Web 应用程序，都可以进行各种灵活的配置，而这些配置文件采用的都是 XML 格式。另外，JSP 2.0 规范也鼓励采用的 XML 语法来编写 JSP 程序，所以，要学习和应用 Java Web 开发技术，必须对 XML 的基本语法有所了解。

1.1.3 XML 与 HTML 的比较

XML 的语法和 HTML 类似，都以一对相互匹配的起始和结束标记符来标记信息。但是二者功能不同，HTML 的标记用于显示数据，而 XML 的标记则是用于描述数据的性质和结构，两者之间有如下一些区别：

（1）HTML 将数据和其显示效果混在一起，它是一种表现技术，定义了如何显示信息的标签；而 XML 文档只是存储了数据和描述了数据之间的关系，没有规定该如何显示数据，即将数据和其显示进行了分离。

（2）HTML 的格式要求比较松散，HTML 解释器通常采用尽量解释的机制，这样容易造成同一个页面在不同的浏览器中可能显示出大不相同的结果；而 XML 是非常严格的标

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

记语言，例如严格区分大小写等问题。

（3）HTML 的标记集合是固定的，你不能在自己的应用中扩展 HTML 标记；而 XML 没有提供一组事先已经定义好的标签，只是提供一个标准，人们可以按照这个标准来定义自己专用的标记，所以 XML 的标记是可无限扩展的，它可以用于描述各种应用领域的数据。

（4）XML 允许粒度更新，不必在 XML 文档每次有局部改变时都发送整个文档内容，只有改变的元素才必须从服务器发送到客户机；HTML 不支持类似功能。

XML 不是 HTML 的升级，也不是 HTML 的替代产品，虽然两者有些相似，但它们的应用领域和范围完全不同。HTML 规范的最终版本 HTML 4.01，HTML 目前已经不再是 Web 标记的标准，它已经被 XHTML 所取代。XHTML 是 HTML 和 XML 的混合物，它完全采用 XML 的语法和规则来编写 Web 页面。XHTML 有效地结合了 HTML 的简单性和 XML 的可扩展性，它所运用的标记和属性基本上还是 HTML 中所定义的那些标记。XML 可以应用在金融、科研等各个领域，XHTML 只是 XML 在 Web 领域的一种应用。

1.1.4 XML 与数据库的比较

XML 与数据库一样具有存储信息的作用，可以认为一个 XML 文档就像数据库中一个表。事实上，一些没有数据库系统的小型网站就使用 XML 文件来存储其信息，如留言板系统中的留言。XML 是一种文档，它具有三个要素：文档结构、文档数据和文档样式。通过与数据库的对比，可以更好地理解这三个要素。

数据库的表结构是设计者自定义的，譬如一个表有多少个字段，每个字段的名称是什么；XML 文档结构就像数据库中的表结构，XML 文档中的标签也是设计者自定义的，标签就相当于数据库表的字段名，这些标签的关系就形成了 XML 文档的结构。

数据库表用于存储数据，XML 文档也用于承载数据，在一个标签对中嵌套的数据就相当于数据库表中的某一条记录中的某个字段的数据。

数据库表中的数据可以被各种应用程序使用，可以采用表格、表单或其他形式显示，也可以只显示其中的部分数据，还可以被应用程序查询、运算、修改。同数据库表一样，XML 文档本身没有定义的数据显示方式，它只提供数据，这些数据也可以像数据库表中的数据一样被其他应用程序查询、运算、修改，还可以使用其他技术将 XML 文档中的内容以多种方式显示。XML 文档本身没有包含文档样式，但 XML 文档中可以指定用来控制该文档样式的 CSS 或 XSL 文件，1.2.8 节的“动手体验”中就有一个使用 CSS 来显示 XML 文档的例子。

多个 XML 文档的元素之间还可以像数据库的表字段一样定义主外键关系，可以把 XML 文档集看成一个非常小型的数据库。XML 比数据库更适合于人们阅读，它是一种通用存储格式，可以由一个应用程序交换给其他程序去处理，非常适合信息交换，但不适合于大批量数据的存储和处理。在实际应用中也经常涉及数据库表与 XML 文档之间的相互转换问题，并且有很多人正在从事这方面的研究工作。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

1.2 XML 的基本语法

1.2.1 文档声明

在一个完整的 XML 文档中必须包含一个 XML 文档声明，该声明必须位于文档的第一行。这个声明表示该文档是一个 XML 文档，以及遵循的是哪个 XML 版本的规范。最简单的 XML 文档声明语法如下：

```
<?xml version="1.0" ?>
```

在“<”和“?”之间、“?”和“>”之间以及第一个“?”和 xml 之间不能有空格，这就是 XML 语法严格性的一个体现。在第二个“?”之前可以没有、也可以有一个或多个空格。声明语句中的 version 属性表示 XML 的版本，因为解析器对不同的版本的解析肯定会有区别，尽管目前只有 1.0 版本，但在声明中必须指定 version 属性。

由于人们可以采用不同的字符编码集来书写一个字符内容完全相同的 XML 文档，所以，XML 软件工具（包括分析器）就需要知道该 XML 文档所使用的字符编码方式。这可以通过在 XML 文档声明中指定 encoding 属性来说明，例如，使用下面的语句来指明文档中的字符编码方式为 GB2312 编码。

```
<?xml version="1.0" encoding="GB2312" ?>
```

W3C 的 XML1.0 规范里规定，所有的 XML 解析器必须接受 UTF-8 和 UTF-16 编码的 Unicode 字符，所以，符合 XML 规范的软件工具一定都支持这两种 Unicode 编码。如果 XML 声明中没有设置 encoding 属性来明确指定文档所用的字符编码方式，则一律以 Unicode 编码看待。XML 解析器通过寻找 XML 文档开始处的字节顺序标记，能够自动检测出文档中的 Unicode 编码是 UTF8，还是 UTF16。也就是说，encoding 属性默认的设置是 Unicode 编码，如果文档中的字符是以 UTF-8 或者是 UTF-16 作为编码，则可以不设置这个属性。

在 XML 文档声明语句中，还可以指定另外一个可选属性：standalone，例如：

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
```

standalone 属性用于说明文档是否是独立的，如果将其设置为“yes”，则表示该文档没有依赖外面的任何文件而可以独立存在，既不需要 DTD 文件来验证其中的标识是否有效，也不需要 XSL、CSS 文件来控制其显示外观；将 standalone 属性设置为“no”时，则表示该文档依赖于外面的某个文件，例如，依赖于某个 DTD 文件或 XSL、CSS 文件。standalone 属性的默认值为“yes”。

注意　如果同时设置了 encoding 和 standalone 属性，standalone 属性要位于 encoding 属性之后。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

加深对 encoding 属性的理解

对于含有中文字符的 XML，其中的字符可以采用 unicode 来编码或 GB2312（简体中文字符编码）编码来表示，如果文档中的字符使用的是 GB2312 编码，则必须将 encoding 属性设置为 GB2312，下面通过一个实验来说明这个问题。

01 用 Windows 自带的记事本程序创建一个名为 book.xml 的文件，文件内容如例程 1-1 所示。

例程 1-1　book.xml

```
<?xml version="1.0" ?>
<书架>
    <书>
        <书名>Java 就业培训教程</书名>
        <作者>张孝祥</作者>
        <售价>39.00 元</售价>
    </书>
    <书>
        <书名>JavaScript 网页开发</书名>
        <作者>张孝祥</作者>
        <售价>28.00 元</售价>
    </书>
</书架>
```

02 用 IE 5.0 以上的浏览器打开 book.xml 文件，其结果如图 1.2 所示。

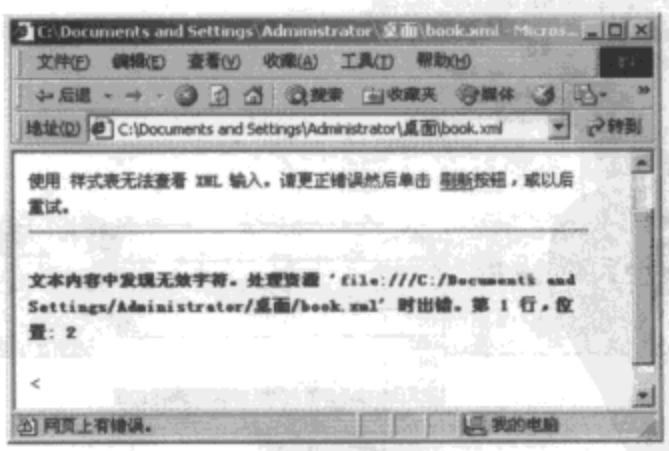


图 1.2

错误提示的信息是“文本内容中发现无效字符”，这就是因为在 book.xml 的 XML 文档声明语句中没有明确指定文档中的字符编码方式，浏览器就会用默认的 Unicode 编码来解析该文档，而该文档中的字符实际上使用的是 GB2312 编码，而非 Unicode 编码。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

03 单击浏览器的“查看”→“源文件”菜单，将打开的 book.xml 文件内容中的第一行修改成如下形式：

```
<?xml version="1.0" encoding="GB2312" ?>
```

保存修改后，刷新显示 book.xml 文件的浏览器窗口，看到的结果如图 1.3 所示。单击某个标签前面的减号（–），嵌套在该标签中的所有内容将被折叠起来，标签前面的减号（–）也将变成加号（+）。单击某个标签前面的加号（+），嵌套在该标签中的所有内容将被展开，标签前面的加号（+）也将变成减号（–）。

图 1.3

04 在上面打开 book.xml 文件的记事本程序中，单击“文件”→“另存为”菜单，在打开的“另存为”对话框中，选择“保存类型”为“所有文件”，编码为“UTF-8”，如图 1.4 所示。

图 1.4

以 UTF-8 编码保存 book.xml 文件后，尽管在记事本程序窗口中显示的效果没有任何变化，但是 book.xml 文件内部存储的数据已经改变，这通过比较 book.xml 保存前后的文件大小就可以看出来。刷新显示 book.xml 文件的浏览器窗口，看到的结果如图 1.5 所示。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

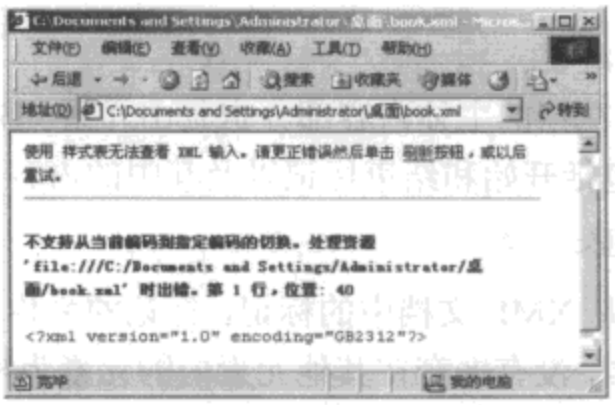


图 1.5

因为现在的 book.xml 文件的字符编码已经转换为 UTF-8，而文档的起始声明中指定 encoding 属性为 GB2312，所以，浏览器在解析 book.xml 文件时会发生错误。

05 将 book.xml 的文档起始声明的 encoding 属性修改为 UTF-8，保存后刷新显示 book.xml 文件的浏览器窗口，就又可以看到类似图 1.3 所示的正常显示效果了。

06 删除 book.xml 文档的声明语句中的 encoding 属性部分，这时候在记事本程序中看到的内容和步骤 **01** 中看到的内容并无什么区别，但是它们内部的数据是不同的。此时的 book.xml 文件中存储的是文字的 UTF-8 编码，而步骤 **01** 中创建的 book.xml 文件中存储的是文字的 GB2312 编码。由于浏览器能够自动检测文档中的 UTF8 编码，即使没有指定字符集编码，用浏览器打开这时候的 book.xml 文件，可以看到与步骤 **05** 完全相同的正常显示效果。可见，当 XML 文件使用非 Unicode 编码时，必须在 XML 文档声明中指定其字符集编码，而采用 Unicode 编码时则可以在 XML 文档声明中省略字符集编码。因不能保证各种 XML 处理器都能处理非 Unicode 编码的 XML 文件，只有实在不能使用 UTF-8 或者 UTF-16 编码文本文件时，才应考虑使用本地字符集，这只能作为最后的一种不得已的手段。

07 将声明语句<?xml version="1.0"?>内的“<”和“?”之间增加一个空格，即将“<?”修改成“< ?”；或者在“?”和“>”之间增加一个空格，即将“?>”修改成“? >”；或者在第一个“?”和 xml 增加一个空格，即将“?xml”修改成“? xml”；或者将元素“书架”的起始标签修改成<书 架>。对于上面的任意一种修改情况，浏览器打开 book.xml 文件时，都会显示类似图 1.6 所示的错误提示信息。

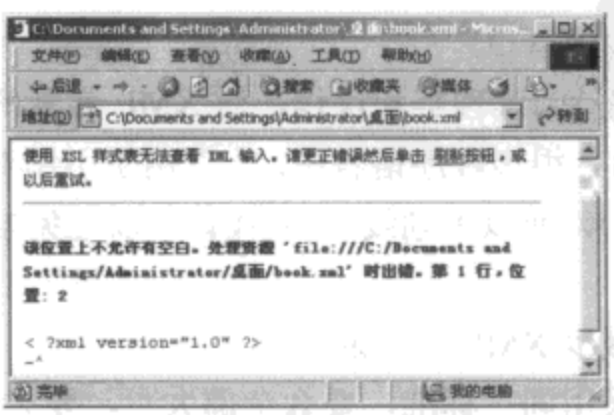


图 1.6

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

1.2.2 元素定义

与 HTML 一样，XML 文档的主体内容部分也是由元素（Element）组成的。一个 XML 元素由一个标记来定义，包括开始和结束标记以及其中的内容，例如：

```
<书名>Java 就业培训教程</书名>
```

其中，<书名>和</书名>就是 XML 文档中的标记，标记的名称也就是元素的名称。在一个元素中可以嵌套若干子元素，没有嵌套在其他元素内的元素为根元素，根元素通常是 XML 文档中定义的第一个元素。格式良好的 XML 文档必须有且仅有一个根元素，其他元素都是这个根元素的子孙元素。

如果在一个元素中没有嵌套子元素和数据，也就是标记对之间没有内容，这样的元素称为空元素。空元素可以不使用结束标记，但必须在起始标记的结束定界符（>）前面增加一个正斜杠（/）字符来说明该标记是一个空元素，例如，<img></img>可以改写成<img />。空元素标记一般都与属性配合使用，例如，<img src="log.gif" />。

在 HTML 中，有的标记没有或者可以不用结束标记或结束说明，如<img src="log.gif">。在 XML 中，所有的标记都必须有结束标记或者使用空元素的结束说明。另外，许多 HTML 浏览器很宽容，对于下面的语句：

```
<b>welcome to <i>www.it315.org</b></i>
```

它们很可能用粗体显示“welcome to”部分，用粗体和斜体显示“www.it315.org”部分。但是在 XML 中，所有的元素都必须合理嵌套，绝对不允许使用类似上述的标签交叉来破坏文档的结构化层次关系。

XML 元素的标记名称中可以包含字母、数字以及其他一些可见字符，但必须遵守下面的一些规范：

- 区分大小写，例如，<P>和<p>是两个不同的标记。HTML 不区分大小写，而 XML 对大小写要求非常严格，如果开始的标识是大写，那么结束的标识也一定是大写。
- 不能以数字或“_”（下划线）开头。
- 不能以字符组合 xml（或 XML、或 Xml 等）开头。
- 不能包含空格。
- 名称中间不能包含冒号（:）。

为了使 XML 元素更容易阅读理解和操作，对 XML 元素的标记名称还有一些建议：

- 不要使用“.”，因为在很多程序语言中，“.”用于引用对象的属性。
- 最好不要用减号（-），而以下划线（_）代替，以避免与表达式中的减号（-）运算符发生冲突。
- 名称尽量简短，以减少 XML 文档的大小。
- 名称的大小写尽量采用同一标准，要么全部大写，要么全部小写。
- 名称可以使用非英文字符，例如中文，但是有些软件可能不支持非英文字符以外的字符，在使用时应考虑这种情况。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

1.2.3 属性定义

与 HTML 一样，也可以为 XML 标签设置属性（Attribute）。XML 中的属性也是自己随便定义的，用于对标签进一步描述和说明。一个标签可以有多个属性，每个属性都有名称和取值，下面是一条使用属性的例子语句：

```
<售价 单位="元">58</售价>
```

在上面的语句中，标签<售价>定义了一个名为“单位”属性。在 HTML 中，属性值可以用双引号（"）、单引号（'）引起来，也可以不使用任何引号。在 XML 中，属性值一定要用双引号（"）或单引号（'）引起来，否则将被视为错误。属性遵循的命名规范与标签相同，比如区分大小写。

属性可以被改成用子元素来描述同样的信息，例如，下面的片断可以与上面的语句表达同样的信息：

```
<售价>
    <单位>元</单位>
    <价格>58</价格>
</售价>
```

属性不易被扩充和被程序操作，但程序处理的速度比使用子元素快。

1.2.4 注释

在 XML 文档中可以加入注释来为 XML 文档添加附加信息，注释的内容会被程序忽略而不被解释和处理。XML 文档与 HTML 文档的注释方式完全一样，语法如下：

```
<!-- 注释信息 -->
```

注释可以用来解释文档中的某些部分的作用和功能，便于阅读和理解，或提示应注意的问题。也可以使用注释的形式在文档中插入作者的姓名、地址和电话号码等个人信息。此外，还可以使用注释来暂时屏蔽某些 XML 语句，让解释程序暂时不要理会这些语句而不用直接删除它们，等到需要时只需简单地取消注释标记即可使其再次发挥作用了。

注意，XML 声明只能作为 XML 文件中的第一行，所以不要把注释放在 XML 声明之前。注释不能嵌套使用，在一对注释符号之中不能包含另一对注释符号，例如下面的注释就是非法的：

```
<!--大段注释
......
    <!--局部注释-->
......
-->
```

因为第一个“<!--”会以在它后面第一次出现的“-->”作为与它配对的结束注释符。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

1.2.5 空格和换行的处理

在 HTML 文本中连续出现的多个空格和回车换行，总是显示为一个空格。在 XML 文本中出现的所有空格和换行，XML 解析程序都要一个个如实地交给下游程序去处理。例如下面两段内容的意义是不一样的。

第一段：

```
<网址>www.it315.org</网址>
```

第二段：

```
<网址>
    www.it315.org
</网址>
```

第一段中的元素内容仅有“www.it315.org”这一个文本串，而第二段中的元素内容为“回车换行”符、“TAB”制表符、“www.it315.org”文本串、又一个“回车换行”符等的总和。在书写 HTML 和其他一些程序代码时，可以使用换行和缩进等方式来让原文件中的内容清晰可读。但是，由于在 XML 中，空格和换行都作为原始内容被处理，所以，在编写 XML 文件时，编写者可能要改变原来的一些“良好”书写习惯。

1.2.6 CDATA 区

CDATA 是 character data 的简写，即字符数据。CDATA 区指的是不想被解析程序解析的一片原始数据区，它以“<![CDATA[”开始，以“]]>”结束。例如，如果直接在 XML 文档中包含类似“Elements look like <this>”的文本，其中的“<this>”将被解析程序解释成一个元素的标签，而人们实际想要的是“<this>”所表示的原义文本，则只要将这段文本放在 CDATA 区中就可以了。这样解析器就不会对 CDATA 区中的内容进行解析，而是将这些数据原封不动地交给下游程序去处理。当要 XML 文档通常将 Javascript 之类的程序代码嵌入 CDATA 区中，如下所示：

```
<script type="text/javascript">
<![CDATA[
    for(var x = 0; x<3; x++)
    {
        alert(x);
    }
]]>
</script>
```

只有这样写 XML 文档，用浏览器打开才能显示出正常的效果。如果去掉上面的 CDATA 区的开始标记符（<![CDATA[）和结束标记符（]]>>），浏览器将报告错误，因为在这种情况下，浏览器将把 Javascript 代码中的小于号（<）当做一个元素标记的开始标记符。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

在某些情况下，还要注意 CDATA 区中的起始和结束处的空格、换行等空白字符的影响，例如：

```
<![CDATA[www.it315.org]]>
```

和

```
<![CDATA[
www.it315.org
]]>
```

这两段文本的意义是不一样的，后者相对前者而言，在字符串“www.it315.org”的前后分别包含了一个换行字符。

与 XML 区分大小写的特性相一致，使用 CDATA 区还应注意其中的“<![CDATA[”不能写成“<![cdata[”或“<![Cdata[”等形式。另外，CDATA 区内部不能出现字符串“]]>”，因为它是 CDATA 区的结束标志。

1.2.7 特殊字符

在 XML 文档中，有些特殊字符需要用转义字符序列表示其原始字面意义。这类字符在 XML 中称为预定义实体。当解析器碰到这样的字符时，不是把它当作一般的字符按其原始意义来处理。例如，字符“<”只能作为元素标记定界符，或在注释、处理指令和 CDATA 区中直接使用，如果要在 XML 中表示字符“<”的原始字面意义，需要使用字符序列“>”来表示。在 XML 文档中一共有 5 个预定义实体，它们与其转义字符序列的对照关系如表 1.1 所示。

表 1.1

特殊字符	替代符号
&	&
<	<
>	>
"	"
'	'

加深对特殊字符的了解

01 修改并验证 1.2.1 节的实验中所创建的 book.xml，保证其没有语法格式问题后，在其中的“Java 就业培训教程”的“就业”和“培训”之间增加一个“&”字符，即修改成“Java 就业&培训教程”。

02 再用 IE5.0 以上浏览器打开 book.xml 文件，将会看到如图 1.7 所示的结果。

错误提示的信息是“应有分号。”这是因为浏览器在 XML 文档中发现“&”字符时，

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

认为这个“&”字符将和后面的若干字符共同组成一个转义字符序列，并要有一个分号（;）来作为当前转义序列的结束。如果找不到这个分号（;），浏览器就会报告如图 1.7 所示的错误信息。

03 将刚才在 book.xml 文件中增加的“&”字符修改成“&”字符序列，保存后刷新显示 book.xml 文件的浏览器窗口，浏览器中显示出了正常的“&”字符，如图 1.8 所示。

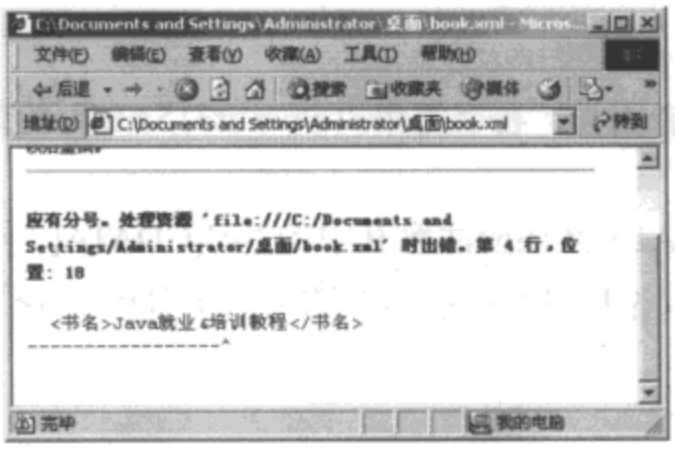


图 1.7

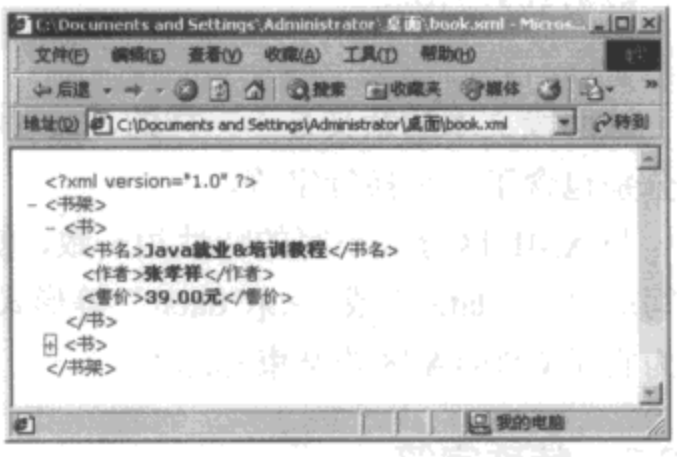


图 1.8

通过这个实验和前面 1.2.1 节中的实验，可以看到 5.0 以上版本的 IE 是一个很好的 XML 文档的格式校验器。

1.2.8 处理指令

处理指令是 Processing Instruction 的中文翻译，通常简称 PI，用来为处理 XML 文档的应用程序提供指示信息。处理指令以“<?”作为开头，以“?>”作为结尾，XML 声明语句就是最常见的一种处理指令。XML 分析器把处理指令原封不动地传给应用程序，由应用程序来解释这个指令，按照它所提供的信息进行处理。例如，在某个 XML 文档中可以包含下面的语句来告诉浏览器使用 book.css 控制其显示效果：

```
<?xml-stylesheet type="text/css" href="book.css"?>
```

使用 CSS 控制 XML 文档的显示

XML 是一种标记可以自定义的语言，它将内容与显示分离，XML 文档本身不知道如何来显示其中的数据，必须有辅助文件来帮助实现显示。例如，为 HTML 设计的 CSS（Cascading Style Sheets，层叠样式表）就可以对 XML 文档的显示外观进行简单控制，但需要通过类似上面的处理指令来关联 CSS 文件与 XML 文档。下面是一个使用 CSS 来控制 XML 文档内容的显示效果的实验。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

01 编写一个名为 book.xml 的文件（如例程 1-2 所示），由于 CSS 不支持中文，所以 book.xml 中的标签名需使用英文单词。

例程 1-2 book.xml

```
<?xml version="1.0" encoding="GB2312"?>
<?xml-stylesheet type="text/css" href="book.css"?>
<bookstore>
    <book>
        <bookname>Java 就业培训教程</bookname>
        <author>张孝祥</author>
        <price>39 元</price>
    </book>
</bookstore>
```

book.xml 文件中的语句<?xml-stylesheet type="text/css" href="book.css"?>规定 XML 文档内容的表现形式由 book.css 文件控制，即 book.css 定义了 XML 文档的视图（显示格式）。

02 在 book.xml 文件所在目录中，编写一个名为 book.css 的文件。

例程 1-3 book.css

```
<?xml version="1.0" encoding="GB2312"?>
bookname {display:block; color:red}
author {display:block; font-style:italic}
price {display:block; color:olive}
```

03 用 IE 浏览器打开 book.xml 文件，显示效果如图 1.9 所示。

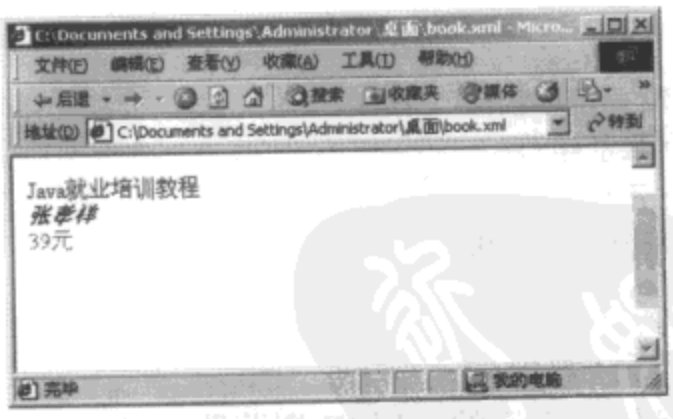


图 1.9

根据图 1.9 中的显示结果可以看到，XML 的内容和表现形式是完全分离的，文档只关心内容，文档内容的外观表现形式则由视图来控制。当视图发生改变时，XML 文档本身不受影响。可以为同一份 XML 文档安排不同的视图，这样，不用修改 XML 文档就可以产生多种显示效果。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

1.3 XML 的约束模式

1.3.1 格式良好的（Well-formed）XML

计算机程序在处理 XML 文档之前，首先必须能够解析出 XML 文档内容中的各个元素的相关信息。解析是处理 XML 文档的第一关，解析出来的元素信息再交给下游程序进行下一步的处理。其实，人们的大脑每天都在做大量的解析工作，如阅读文章或听别人演讲，需要进行从断字、断句一直到语法、语意的分析，大脑这个解析器的复杂程度和精确程度是目前任何电脑都望尘莫及的。只有正确解析别人传达的信息以后，大脑才能做进一步的处理。用于解析 XML 文档的解析器程序对应的英文术语叫 Parser。Parser 可以独立使用，也可以成为编辑软件或浏览器的一部分。

面对一个两岁小孩随手在纸上乱画的几个符号，人们一般是无法解析其所表达的含义的，就更别谈如何对此做出下一步的反应了。人脑尚且如此，要让计算机程序解析出 XML 文档内容所表述的信息，XML 文档更应该严格遵循一定的语法。遵守 XML 基本语法规则和规范的 XML 文档就可以称之为“Well-formed XML”，中文意思就是“格式良好的 XML”。一个 XML 文档首先必须是格式良好的。

1.3.2 XML 的约束模式

一篇文章的语法是非常正确的，但如果有反动言论或逻辑矛盾，这就超出了某种限制，则不允许发表。同样的道理，在有些情况下，XML 文档仅是格式良好的还不够，其中的内容还必须满足某些条件的约束限制。例如，有一个 XML 文档中的内容如下：

```
<?xml version="1.0" encoding="gb2312"?>
<书架>
    <书>
        <书名>Java 就业培训教程</书名>
        <作者 姓名="张孝祥"/>
        <售价 单位="元">38</售价>
        <售价 单位="元">28</售价>
    </书>
</书架>
```

尽管这个 XML 文档是格式良好的，用 IE 浏览器打开它也不会出现任何问题，但是，同一本书出现两种售价，而且从标记名上又无法区分哪个是原价，哪个是优惠价，这在逻辑上是不允许的。因此，在有些情况下，还需要定义一套规则来对 XML 文档中的内容做出限制，这套定义的规则就是 XML 文档的约束模式。

约束模式定义了 XML 文档中允许出现的元素名（也就是标记名）、元素中的属性、内容类型，以及元素之间的嵌套关系和出现顺序。可见，XML 约束模式不仅定义了 XML 文档的

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

词汇表，它还定义了一个 XML 文档必须遵循什么样的结构。如果把一个 XML 文件看做数据库中的一个表，那么 XML 约束模式就相当于数据库表结构的定义，在创建一个表时可以为其中的字段定义约束规则，在 XML 约束模式中则可以为 XML 文档标签定义约束规则。

如果没有为一个 XML 文档指定约束模式，那么该文档中可以包含任何类型的标记；如果为一个 XML 文档指定了约束模式，那么它必须满足约束模式所规定的结构、数据类型和数据关联等内容。

1.3.3 XML 的约束模式语言

XML 约束模式的内容也需要遵循一定的语法规则，这些语法规则就形成了 XML 约束模式语言。计算机行业的各种语言就是指各种语法规则，XML 约束模式语言就是用于定义 XML 约束模式的语法规则。XML 约束模式语言是用来创造 XML 标记语言的语言，这种用于定义另一种语言的语言被称为元语言。约束模式通常都是在一个单独的文件中进行定义，这个文件被称为模式文档。模式文档采用某种约束模式语言编写，用于描述 XML 文档的结构，与模式文档的概念相对应，依照模式文档编写的 XML 文档被称为实例文档，它包含真正的 XML 数据。

由于可以采用多种不同的语法来对同一事物概念进行描述，所以，先后出现的 XML 约束模式语言有 XML DTD、XDR、SOX、XML Schema 等，其中应用最广泛和具有代表意义的是 XML DTD 和 XML Schema。XML DTD（Document Type Definition，文档类型定义）是早期出现的一种 XML 约束模式语言，也是目前使用最广泛的一种 XML 约束模式，它用于定义 XML 文档中允许出现的元素名、元素数据类型和元素所能包含的内容类型、元素中的属性，以及元素之间的关系和出现顺序。XML Schema 是对 XML 文档内容进行约束的另外一种模式，其出现目的是为了克服 DTD 的局限性，为 XML 文档提供丰富的语法结构。XML Schema 已经成为 W3C 的正式推荐标准，由于它比 DTD 具有更多功能和灵活性，而且采用的全是 XML 语法，XML Schema 已逐步呈现出替代 XML DTD 的趋势。

尽管强有力的 XML Schema 已作为 XML 约束模式语言的一种标准和主流，但是，XML DTD 作为一种最简单的 XML 约束模式语言，而且它在 XML schema 出现之前已被广泛使用，很多人现在还在继续使用 XML DTD 为自己的 XML 文档定义约束规则，所以，XML DTD 还将在一段时间内发挥它的作用，读者仍然有必要对 XML DTD 有所了解。

1.3.4 有效的（Valid）XML

一个遵守 XML 的基本语法规则、且符合为它指定的某个 XML 约束模式的 XML 文档就可以称之为“Valid XML”文档，中文意思就是“有效的 XML”文档。一个 XML 文档并不一定需要是有效的，但必须是格式良好的。将一个 XML 文档和它所引用的 XML 约束模式进行比较分析，看其中的内容是否符合 XML 约束模式的过程叫校验（validation），校验过程也是通过解析器程序软件（Parser）来处理的。根据能否对 XML 文档进行约束模式校验，可以将 XML 解析器分为两类：非校验类解析器和校验类解析器。非校验类解析

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

器只检测 XML 文档是否是格式良好的，即是否遵守 XML 的语法规则；校验类解析器不但检测 XML 文档是否是格式良好的，还验证 XML 文档是否符合相应的约束模式。IE 浏览器属于非校验类解析器，它不验证 XML 文档是否符合某个约束模式，它只检测 XML 文档格式是否是良好的。

1.4 DTD

文档类型定义 DTD（Document Type Definition）是早期出现的一种 XML 约束模式语言，按照 DTD 语法创建的文件称为 DTD 文件。通过比较 XML 文档和 DTD 文件，就可以判断 XML 文档是否符合某种约束规范，以及其中的元素和标签使用是否正确。

1.4.1 对 DTD 的初步认识

一个 DTD 文件中可以包含元素和元素之间的关系的定义，元素的属性定义，以及实体和符号的定义。DTD 文件是文本格式的文件，后缀名通常为.dtd。假设一个 XML 实例文档的内容如例程 1-4 所示。

例程 1-4 book.xml

```
<?xml version="1.0" ?>
<书架>
    <书>
        <书名>Java 就业培训教程</书名>
        <作者>张孝祥</作者>
        <售价>39.00 元</售价>
    </书>
    <书>
        <书名>JavaScript 网页开发</书名>
        <作者>张孝祥</作者>
        <售价>28.00 元</售价>
    </书>
</书架>
```

对于例程 1-4 所示的 book.xml 文件，可以使用如例程 1-5 所示的 DTD 文件来定义它的约束规则。

例程 1-5 book.dtd

```
<!ELEMENT 书架 (书+)>
<!ELEMENT 书 (书名,作者,售价)>
<!ELEMENT 书名 (#PCDATA)>
<!ELEMENT 作者 (#PCDATA)>
<!ELEMENT 售价 (#PCDATA)>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

book.dtd 是一个非常简单的 DTD 文件，它定义了 book.xml 文件中的各个元素和它们的相互关系。其中的每一条<!ELEMENT ...>语句用于定义一个元素，<!ELEMENT ...>语句的通用语法格式为：

```
<!ELEMENT 元素名称 使用规则>
```

例如，“<!ELEMENT 书架（书名,作者,售价）>”语句中的“书架”为元素名称，“（书名,作者,售价）”为使用规则。

元素的使用规则定义了元素中包含的组成成分，以及每种组成成分出现的次数、次序，还可以是某些成分进行某种关系组合后出现的次数、次序。例如，在 book.dtd 中，（#PCDATA）用于表示元素中嵌套的内容是普通文本字符串；（书名,作者,售价）用于表示元素中要嵌套书名、作者、售价等子元素，并且这些子元素要按顺序依次出现；在元素的使用规则中可以通过正则表达式来定义子元素出现的次数，（书+）中的元字符“+”用于表示它修饰的成分必须出现一次或多次。

DTD 文件应该使用 UTF-8 或 Unicode 编码。如果 DTD 文件中的文本内容全部是英文字符，它们本身就符合 UTF-8 编码，不需要进行特殊的转换；如果 DTD 文件中的文本内容包含中文字符，那么就需要将它们转换成 UTF-8 编码后保存。

1.4.2 在 XML 文档中引入外部 DTD 文件

如果一个 XML 文档想说明自己是一个有效的 XML 文档，它除了要严格遵循某个 DTD 文件中定义的规则外，还必须指明其所遵循的是哪个 DTD 文件，以便解析器程序能够对它进行有效性校验。如何把 XIM 文件和一个 DTD 文件联系起来呢？XML 文档通过使用 DOCTYPE 声明语句（文档类型定义语句）来指明它所遵循的 DTD 文件。DOCTYPE 声明语句紧跟在 XML 文档声明语句后面，有两种格式：

（1）<!DOCTYPE 文档类型名称 SYSTEM "DTD 文件的 URL">

（2）<!DOCTYPE 文档类型名称 PUBLIC "DTD 名称" "DTD 文件的 URL">

各部分的意义和作用如下：

- **文档类型名称**可以由 XML 文档编写者自己定义，一个通用的习惯是使用 XML 文档的根元素名称来作为文档类型名称。
- **关键字 SYSTEM（第一种格式）**表明 XML 文件所遵循的是一个本地或组织内部所编写和使用的 DTD 文件；关键字 PUBLIC（第二种格式）表明该 XML 文件所遵循的是一个由权威机构制定的、公开提供给特定行业或公众使用的 DTD 文件，而不是某个组织内部的规范文件。
- **"DTD 名称"**用于指定该 DTD 文件的标识名称，它只在使用关键字 PUBLIC 的 DOCTYPE 声明语句中出现。DTD 标识名称需要用双引号（"）括起来，例如，用于 Java Web 应用程序的配置文件的 DTD 文件的标识名称为"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"。DTD 标识名称应符合一些标准的规定，对于

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

ISO 标准的 DTD 以“ISO”三个字母开头；被改进的非 ISO 标准的 DTD 以加号“+”开头；未被改进的非 ISO 标准的 DTD 以减号“–”开头。紧跟着开始部分后面的是双斜杠“//”及 DTD 所有者的名称，在这个名称之后又是双斜杠“//”，之后是 DTD 所描述的文件的说明，最后在双斜杠“//”之后是语言的种类。虽然标识名称看上去比较复杂，但这完全是由 DTD 文件发布者去考虑的事情，XML 文件的编写者只要把 DTD 文件发布者事先定义好的 DTD 标识名称复制到相应的位置就可以了。

- **"DTD 文件的 URL"**部分指定该 DTD 文件所在的位置，需要用双引号（"）括起来。对于使用 PUBLIC 属性的 DOCTYPE 语句，"DTD 文件的 URL"指定该 DTD 文件在 Internet 上的绝对 URL，例如，用于 Java Web 应用程序的配置文件的 DTD 文件的位置为"http://java.sun.com/dtd/Web-app_2_3.dtd"。解析器在对 XML 文档进行有效性确认时，通常都需要从该指定的 URL 下载 DTD 文件，但是，对于一些已经制定成为了行业标准的 DTD 文件，一些相应的解释器程序可能已经将它们内嵌进去，所以在解析时并不一定要到指定的 URL 进行下载。对于使用 SYSTEM 属性的 DOCTYPE 语句，"DTD 文件的 URL"除了可以是 Internet 上的一个绝对 URL 外，它还可以是一个本地文件系统的相对路径，例如，<!DOCTYPE 书架 SYSTEM "book.dtd">，它说明 book.dtd 文件和引用它的 XML 文件在同一个目录下。

如果想明确要求解析器程序必须对 XML 文档进行有效性验证时，XML 声明语句中的 standalone 属性应设置为"no"，以说明这个 XML 文档不是自成一体的，必须和其他文件（这里是 DTD 文件）一起使用。要想在例程 1-4 所示的 book.xml 文件中引入本地的例程 1-5 所示的 book.dtd 文件作为它的约束模式，应该使用 SYSTEM 属性的 DOCTYPE 声明语句，修改后的 book.xml 文件内容如例程 1-6 所示。

例程 1-6　book.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE 书架 SYSTEM "book.dtd">
<书架>
    <书>
        <书名>Java 就业培训教程</书名>
        <作者>张孝祥</作者>
        <售价>39.00 元</售价>
    </书>
    <书>
        <书名>JavaScript 网页开发</书名>
        <作者>张孝祥</作者>
        <售价>28.00 元</售价>
    </书>
</书架>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

在一份合同的开始处，人们通常都要写上“依据国家某某法，特签订如下合同”，而 XML 文档的 DOCTYPE 声明则好比“依据某某 DTD 规范，特书写如下 XML 文档”的含义。如果把一个 XML 文件比喻成人们起草的一份合同文件，那么 DOCTYPE 声明就相当于合同中所引述的“依据国家某某法”这样的说明，DTD 文件就是“国家某某法”的定义文件。

一般情况下，使用 DOCTYPE 声明语句后，即使在 XML 声明语句中省略了 standalone 属性，解析器也会自动对 XML 文档进行有效性验证。目前的 IE 浏览器只能验证 XML 文档的格式是否良好，它并不具有对 XML 文档进行合法性校验的功能。如果你想体验一下 XML 文档的 DTD 校验效果，那么应该使用专门的 XML 工具软件，例如 XML SPY。

1.4.3 在 XML 文档中直接嵌入 DTD 定义语句

在 XML 文档中，还可以使用另外一种形式的 DOCTYPE 声明语句来直接包含 DTD 定义语句，其格式为：

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE 根元素名 [
    DTD 定义语句
    ……
]>
```

如果要在例程 1-6 所示的 book.xml 文件中直接包含 book.dtd 文件中的 DTD 定义语句，可以将它修改成例程 1-7 所示的形式。

例程 1-7 book.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE 书架 [
    <!ELEMENT 书架 (书+)>
    <!ELEMENT 书 (书名,作者,售价)>
    <!ELEMENT 书名 (#PCDATA)>
    <!ELEMENT 作者 (#PCDATA)>
    <!ELEMENT 售价 (#PCDATA)>
]>
<书架>
    <书>
        <书名>Java 就业培训教程</书名>
        <作者>张孝祥</作者>
        <售价>39.00 元</售价>
    </书>
    <书>
        <书名>JavaScript 网页开发</书名>
        <作者>张孝祥</作者>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
<售价>28.00 元</售价>
    </书>
</书架>
```

一个 DTD 定义通常都要被多个 XML 文件引用。也就是说，人们可以按照一个 DTD 定义来创建出多个遵循它的语法定义的 XML 文件。在 XML 文档中直接包含 DTD 定义语句的做法要为每一个 XML 文件都加入一段相同的 DTD 定义语句，不利于 DTD 定义的统一管理和使用。所以，最好采用上一节的方式，将 DTD 定义语句放在一个单独的 DTD 文件中进行定义，然后由各个 XML 文件引用这个外部 DTD 文件。这样非常便于 DTD 定义的管理和维护，以及被多个 XML 文件所共享。

在使用 DOCTYPE 声明语句引入一个外部的 DTD 文件的同时，还可以在这个 DOCTYPE 声明语句嵌入 DTD 定义语句，如下所示：

```
<!DOCTYPE Web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/Web-app_2_3.dtd"
[
    !ENTITY copyright SYSTEM "http://www.it315.org/copyright.xml"
]>
```

到这里为止，我们可以归纳总结一下 XML 文档的各个组成部分了。从整体结构上来看，一个完整的 XML 文档包括如下一些组成部分：

- XML 声明（declaration）
- DOCTYPE 声明语句
- 处理指令（processing instruction）
- 元素（element）
- 注释（comment）
- CDATA 区

但请注意的是，并不是每个 XML 文档都必须具有所有这些组成部分。

1.5 DTD 的语法细节

一般的用户无须自己编写 DTD 文件和掌握 DTD 的完整语法，只要建立对 DTD 文件的直观认识和概念就可以了。因此，本节内容不需作为重点学习，但是读者还应先通读一下本节的内容，留个初步的印象和概念，以便日后有需要时能够想起来并来此详细查阅。

1.5.1 元素定义

元素是 XML 文档的基本组成部分，在 XML 文档中使用的每个元素及其属性都要在 DTD 文件中进行定义。在 DTD 定义中，每一条<!ELEMENT ...>语句用于定义一个元素，

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

<!ELEMENT ...>语句的通用语法格式为：

```
<!ELEMENT 元素名称 使用规则>
```

例如，“<!ELEMENT 书架（书名,作者,售价）>”语句中的“书架”为元素名称，“(书名,作者,售价)”为使用规则。

元素的使用规则定义了元素中包含的组成成分，以及每种组成成分出现的次数、次序，还可以是某些成分进行某种关系组合后出现的次数、次序。DTD 通过定义元素的使用规则来描述整个 XML 文件的文档结构，并对文档结构中的每一个细节进行说明。DTD 不但说明 XML 文件中可能存在的元素，而且还说明元素的具体类型。一个 XML 元素可以为空，也可以是一段纯文本，还可以有若干子元素，而所有这一切都是通过元素的使用规则说明的。在 DTD 的元素定义规则中，可以有以下一些形式：

- （#PCDATA）表示元素中嵌套的内容是普通文本字符串，其中，关键字 PCDATA 是 Parsed Character Data（被解析的字符数据）的简写，因此，在元素内容中出现的特殊字符需要用相应的预定义实体表示。
- 用一对圆括号()将元素中要嵌套的一组子元素括起来，例如，（书名,作者,售价）表示元素中要嵌套书名、作者、售价等子元素。
- 元素中也可以嵌套文本字符串与其他子元素的混合，例如，（书名, #PCDATA）。
- 关键字 EMPTY 表示元素中不包含任何子元素和普通文本字符串，这种情况用于定义 XML 文件中的空元素，例如，<!ELEMENT HR EMPTY>语句在 XML 文件中所定义的元素形式为<HR />。
- 关键字 ANY 表示元素中可以有任何类型的子元素和普通文本字符串，以及它们的混合，还可以不包含任何内容。<!ELEMENT 联系人列表 ANY>语句表示在 XML 文件中定义的<联系人列表>元素中可以嵌套任何形式的内容。除了根元素外，在定义其他元素时使用关键字 ANY 都是不好的习惯，这样就失去了 DTD 对 XML 文档的约束效果。

对于 DTD 文件中的元素定义，还有下面的一些补充细节说明：

（1）XML DTD 使用与 XML 文档同样的注释方式，即采用如下语法格式：

```
<!-- 注释内容 -->
```

（2）每条元素定义语句的顺序是无关紧要的。

（3）在 XML 文档结构中具有不同用途的元素不能使用相同的元素名，这样会引起文件各个元素的混淆，使文件的可读性变差。

（4）一个元素的各个组成成分可以以任意顺序出现，也可以强制遵循一定的顺序。如果元素的使用规则中的各个组成成分用空白符分隔，那么它们的出现顺序没有严格要求。例如，<!ELEMENT MYFILE (TITLE AUTHOR EMAIL)>语句表示 MYFILE 元素中可以按任意顺序包含 TITILE、AUTHOR、EMAIL 这三个子元素。如果元素的使用规则中的各个组成成分用逗号（,）分隔，那么它们在 XML 文档中的出现顺序必须与它们的排列顺序一致。例如，<!ELEMENT MYFILE (TITLE,AUTHOR,EMAIL)>语句表示 MYFILE 元素中必

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

须依次包含 TITILE、AUTHOR、EMAIL 这三个子元素。

（5）如果一个元素的各个组成成分以竖杠字符（|）分隔，那么在 XML 文档中只能出现它们之中的任何一个，也就是竖杠字符（|）用于表示各个组成成分之间的“或者”关系。例如，<!ELEMENT MYFILE (TITLE|AUTHOR|EMAIL)>语句表示 MYFILE 元素中必须且只能包含 TITLE、AUTHOR、EMAIL 这三个子元素中的任何一个。

（6）在元素的使用规则中可以通过正则表达式来定义子元素出现的次数。元字符“+”用于表示它修饰的成分必须出现一次或多次，例如，“（书+）”中的元字符“+”表示它修饰的成分必须出现一次或多次；元字符“?”用于表示它修饰的成分只能出现一次或不出现；元字符“*”用于表示它修饰的成分可以不出现，也可以出现一次或多次；不使用任何元字符则表示该成分必须且只能出现一次。

（7）一对圆括号()可用于将括在其中的内容组合成一个可统一操作的分组，分组中可以嵌套更小的分组。例如，对于下面的 DTD 定义语句：

```
<!ELEMENT MYFILE((TITLE*, AUTHOR?, EMAIL)* | COMMENT)>
```

在 XML 文档中出现的 MYFILE 元素要么包含一个 COMMENT 元素，要么包含零个、一个或多个(TITLE*, AUTHOR?, EMAIL)所表示的分组。分组里包含零个、一个或多个 TITLE 子元素，接着是零个或一个 AUTHOR 子元素，再接着是一个 EMAIL 子元素。

1.5.2 属性定义

在 XML 文档中可以为元素设置属性，一个元素到底可以具有怎样的属性设置，这也需要在 DTD 文件中进行定义，语法格式如下：

```
<!ATTLIST 元素名
    属性名 1 属性类型 设置说明
    属性名 2 属性类型 设置说明
    ……
>
```

ATTLIST 用于定义一个元素的所有属性设置信息，其中可以定义一个或多个属性。例如，有一个名为<商品>的元素，它在 XML 文档中的定义如下：

```
<商品 类别="服装" 颜色="黄色">…</商品>
```

显然，<商品>元素具有两个属性，即“类别”和“颜色”。在 DTD 中，可以使用如下形式的语句来定义<商品>元素的属性信息：

```
<!ATTLIST 商品
    类别 CDATA #REQUIRED
    颜色 CDATA #IMPLIED
>
```

其中，CDATA 为一种属性类型，表示属性值为普通文本字符串；#REQUIRED 和

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

#IMPLIED 为属性的设置说明信息，#REQUIRED 表示必须设置该属性，#IMPLIED 表示该属性可有可无。

在 DTD 中定义元素的属性时，可以有下面几种设置说明：

1. #REQUIRED

关键字#REQUIRED 说明必须设置该属性。

2. #IMPLIED

关键字#IMPLIED 说明可以设置，也可以不设置该属性。

3. #FIXED

关键字#FIXED 说明该属性的取值固定为一个默认值，在 XML 文件中不能将该属性设置为其他值。使用#FIXED 关键字时，还需要为该属性提供一个默认值。

4. 直接使用默认值

如果在属性的设置说明部分没有使用上面任何一种关键字，而是直接指定一个默认值的话，那么在 XML 文件中可以设置，也可以不设置该属性。如果 XML 文件中没有设置该属性，该属性将被自动设置，设置值为 DTD 中定义的默认值；如果 XML 文件中设置了该属性，可以使用新的属性值来覆盖 DTD 中定义的默认值。

对于一个属性究竟使用哪种设置说明，需要根据实际情况来决定。下面是一个综合了上面几种属性设置说明的例子：

```
<!ATTLIST 页面作者
    姓名 CDATA #IMPLIED
    年龄 CDATA #IMPLIED
    联系信息 CDATA #REQUIRED
    网站职务 CDATA #FIXED "页面作者"
    个人爱好 CDATA "上网"
>
```

在 DTD 中定义元素的属性时，可以将属性设置成下面 10 种类型。

1. CDATA

使用关键字 CDATA 作为属性类型时，它表示属性值为普通文本字符串。当然，在属性设置值中出现的特殊字符，也需要使用其转义字符序列来表示，例如，用&表示字符（&），用<表示字符（<），用"表示字符（"）。

2. ENUMERATED（枚举类型）

属性的类型可以是一组取值的列表，在 XML 文件中设置的属性值只能是这个列表中的某个值，这种特殊的属性类型被称之为 ENUMERATED（枚举类型）。注意：在 DTD 定

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

义中并不会出现关键字 ENUMERATED。例程 1-8 是一个定义 ENUMERATED 类型的属性的例子程序。

例程 1-8 enum.xml

```
<?xml version = "1.0" encoding="GB2312" standalone="yes"?>
<!DOCTYPE 购物篮 [
    <!ELEMENT 购物篮 ANY>
    <!ELEMENT 肉 EMPTY>
    <!ATTLIST 肉 品种 ( 鸡肉 | 牛肉 | 猪肉 | 鱼肉 ) "鸡肉">
]>
<购物篮>
    <肉 品种="鱼肉"/>
    <肉 品种="牛肉"/>
    <肉/>
</购物篮>
```

在上面这个例子中，给“品种”这个属性定义的默认值是“鸡肉”，所以，即使<购物篮>元素中的第三个子元素没有显式定义“品种”这个属性，但它实际也具有“品种”这个属性，且该属性的取值为“鸡肉”。

3. ID

使用关键字 ID 作为属性类型时，它表示属性的设置值将用于惟一标识一个 XML 文件中的某个元素，这种 ID 类型的属性通常由 XML 文件的解析处理程序或脚本语言使用。

ID 属性的值必须符合一定的名称规范，即只由字母、数字或下划线开始，其中不能出现空白字符（例如，空格和制表符）。不要给 ID 类型的属性指定默认值，这很容易造成不同的元素具有相同标识名的情况，更不能使用#FIXED 作为 ID 类型的属性的设置说明。ID 类型的属性通常使用#REQUIRED 作为设置说明，如果应用并不要求每个元素都有自己的标识时，那也可以使用#IMPLIED 作为属性的设置说明。例程 1-9 是一个定义 ID 类型的属性的例子程序。

例程 1-9 id.xml

```
<?xml version = "1.0" encoding="GB2312" standalone = "yes"?>
<!DOCTYPE 联系人列表[
    <!ELEMENT 联系人列表 ANY>
    <!ELEMENT 联系人(姓名,EMAIL)>
    <!ELEMENT 姓名(#PCDATA)>
    <!ELEMENT EMAIL(#PCDATA)>
    <!ATTLIST 联系人 编号 ID #REQUIRED>
]>
<联系人列表>
    <联系人 编号="1">
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        <姓名>张三</姓名>
        <EMAIL>zhang@it315.org</EMAIL>
    </联系人>
    <联系人 编号="2">
        <姓名>李四</姓名>
        <EMAIL>li@it315.org</EMAIL>
    </联系人>
</联系人列表>
```

4. IDREF 和 IDREFS

使用关键字 IDREF 作为属性类型时，一个元素的 IDREF 类型的属性设置值将是同一个 XML 文件中的另一个元素的 ID 类型的属性的设置值。例程 1-10 是一个定义 IDREF 类型的属性的例子程序。

例程 1-10　idref.xml

```
<?xml version = "1.0" encoding="GB2312" standalone = "yes"?>
<!DOCTYPE 联系人列表[
    <!ELEMENT 联系人列表 ANY>
    <!ELEMENT 联系人(姓名,EMAIL)>
    <!ELEMENT 姓名(#PCDATA)>
    <!ELEMENT EMAIL(#PCDATA)>
    <!ATTLIST 联系人 编号 ID #REQUIRED>
    <!ATTLIST 联系人 上司 IDREF #IMPLIED>
]>
<联系人列表>
    <联系人 编号="2">
        <姓名>张三</姓名>
        <EMAIL>zhang@it315.org</EMAIL>
    </联系人>
    <联系人 编号="1" 上司="2">
        <姓名>李四</姓名>
        <EMAIL>li@it315.org</EMAIL>
    </联系人>
</联系人列表>
```

IDREFS 关键字用于表示 IDREF 的列表类型，一个元素的 IDREFS 类型的属性设置值可以是同一个 XML 文件中的另外多个元素的 ID 类型的属性的设置值，每个 ID 属性值之间用空格分隔。

5. NMTOKEN 和 NMTOKENS

NMTOKEN 是 Name Token 的简写，中文意思就是名称记号，它表示由一个或多个字

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

母、数字、句点（.）、连字号（-）或底线（_）所组成的一个名称。除了第一个字符位置外，NMTOKEN 属性的设置值中也可以包含冒号（:）。NMTOKENS 关键字用于表示一种列表类型，一个元素的 NMTOKENS 类型的属性设置值可以是同一个 XML 文件中的另外多个元素的 NMTOKEN 类型的属性的设置值，每个 NMTOKEN 属性值之间用空格分隔。例如，对于如下的 DTD 定义语句：

```
<!ELEMENT 用户 EMPTY >
<!ATTLIST 用户 姓名 NMTOKEN #REQUIRED>
<!ELEMENT 数据 (#PCDATA)>
<!ATTLIST 数据 授权用户 NMTOKENS #IMPLIED>
```

其中，<用户>元素的“姓名”属性指定为 NMTOKEN 类型，<数据>元素的“授权用户”属性指定为 NMTOKENS，与这段 DTD 定义语句对应的 XML 数据片断如下：

```
<用户 姓名="zhangsan" />
<用户 姓名="lisi" />
<数据 授权用户 = "zhangsan lisi">
    这里是一些授权访问的数据
</数据>
```

6. NOTATION

现实世界中存在着很多无法或不易用 XML 格式组织的数据，例如图像、声音、影像，等等。对于这些数据，XML 应用程序常常并不提供直接的应用支持，但可以通过设置 NOTATION 类型的属性来让一个外部应用程序进行处理。NOTATION 类型的属性值为在 DTD 中使用<!NOTATION ...>语句定义的一个 notation（符号），这对使用非 XML 格式的数据非常有用。notation 定义语句分为两种情况：

```
<!NOTATION 符号名 SYSTEM "MIME 类型">
<!NOTATION 符号名 SYSTEM "URL 路径名">
```

第一种指定数据的 MIME 类型，第二种指定处理程序的 URL 路径。当使用 NOTATION 类型作为属性的类型时，首先要在 DTD 中使用<!NOTATION ...>语句定义相应的 notation。例程 1-11 是一个使用 NOTATION 属性和<!NOTATION ...>语句的例子程序。

例程 1-11　notation.xml

```
<?xml version = "1.0" encoding="GB2312" standalone="yes"?>
<!DOCTYPE 文件[
    <!NOTATION mp SYSTEM "movPlayer.exe">
    <!NOTATION gif SYSTEM "Image/gif">
    <!ELEMENT 文件 ANY>
    <!ELEMENT 电影 EMPTY>
    <!ATTLIST 电影 演示设备 NOTATION ( mp | gif ) #REQUIRED>
]>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
<文件>
    <电影 演示设备="mp"/>
</文件>
```

在例程 1-11 中，为<电影>元素指定了两种可选的演示设备：一种是 movPlayer.exe，另一种则用来绘制 GIF 图像的应用程序。

7. ENTITY 和 ENTITYS

ENTITY 对应的中文意思为实体，关于实体定义的细节，请参看 1.5.3 节。使用关键字 ENTITY 作为属性的类型时，表明其属性值必须为在 DTD 中使用<!ENTITY ...>语句定义的一个实体（entity）的引用。例如，对于如下的 DTD 定义语句：

```
<!ENTITY it315 "IT 资讯交流网，www.it315.org">
<!ELEMENT 电影 EMPTY>
<!ATTLIST 电影 来源 ENTITY #REQUIRED>
```

与这段 DTD 定义语句对应的 XML 数据片断如下：

```
<电影 来源="&it315;" />
```

只有引用实体才可以作为 ENTITY 类型的属性的设置值，参数实体不能用做 ENTITY 类型的属性的设置值。ENTITYS 关键字用于表示一种列表类型，一个元素的 ENTITYS 类型的属性设置值可以是多个实体的引用，每个实体的引用用空格分隔。

1.5.3 实体定义

在 DTD 中可以预先定义一个或多个实体，实体的根本作用是为一段文本内容创建一个别名，以后在 XML 文档中就可以多次引用这个别名，XML 解析器程序将把 XML 文档中出现的别名引用转变成其所对应的文本内容。如果有多个文档或一个文档中的多处都要书写一段相同的内容时，就可以将这段内容定义成一个实体，然后在各个文档中引用这个实体名来替代它所表示的文本内容。如果要修改所有文档中对这个实体所代表的那部分内容时，只要在 DTD 文件中将实体所对应的内容修改一遍就可以了。这样非常便于对多个文档中的相同内容进行统一维护和修改，可以减少差错。

在 DTD 定义中，一条<!ENTITY ...>语句用于定义一个实体。实体可分为两种类型：引用实体和参数实体。

1. 引用实体

引用实体两种语法定义格式：

```
<!ENTITY 实体名称 "实体内容">
<!ENTITY 实体名称 SYSTEM "外部 XML 文档的 URL">
```

引用实体主要在 XML 文档中被引用，其引用方式为：

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

&实体名称;

如果实体名称是通过第一种语法格式定义的，它将被直接转变成实体内容。如果实体名称是通过第二种语法格式定义的，它将被替换成外部XML文档中的内容，即在当前XML文档中的实体名称引用处插入外部 XML 文档中的内容。例如，在一个 XML 文档中要引入另外一个包含版权信息的 XML 文档，可以采用如下语法：

```
<!DOCTYPE copyright [
    <!ENTITY copyright SYSTEM "http://www.it315.org/copyright.xml">
]>
……
&copyright;
```

加深对引用实体的了解

01 使用记事本程序编写如下两个文件：例程 1-12 和例程 1-13。

例程 1-12　book2.dtd

```
<!ENTITY it315 "IT 资讯交流网，www.it315.org">
<!ELEMENT 书架 (书+)>
<!ELEMENT 书 (书名,作者,售价)>
<!ELEMENT 书名 (#PCDATA)>
<!ELEMENT 作者 (#PCDATA)>
<!ELEMENT 售价 (#PCDATA)>
```

例程 1-13　book2.xml

```
<?xml version="1.0" encoding="GB2312" standalone="yes"?>
<!DOCTYPE 书架 SYSTEM "book2.dtd">
<书架>
    <书>
        <书名>Java 就业培训教程</书名>
        <作者>&it315;</作者>
        <售价>39.00 元</售价>
    </书>
    <书>
        <书名>JavaScript 网页开发</书名>
        <作者>张孝祥</作者>
        <售价>28.00 元</售价>
    </书>
</书架>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

02 用 IE 浏览器打开 book2.xml 文件，看到的结果如图 1.10 所示。

错误提示的信息是“文本内容中发现无效字符。”这是因为 book2.dtd 文件使用的是本地字符集编码，即 GB2312 编码，而 DTD 文件应该使用 UTF-8 或 Unicode 编码。

03 按照图 1.4 所示的方式将 book2.dtd 以 UTF-8 编码保存后，用 IE 浏览器重新打开 book2.xml 文件或直接在图 1.10 中单击工具栏中的“刷新”按钮，浏览器中显示的结果如图 1.11 所示。

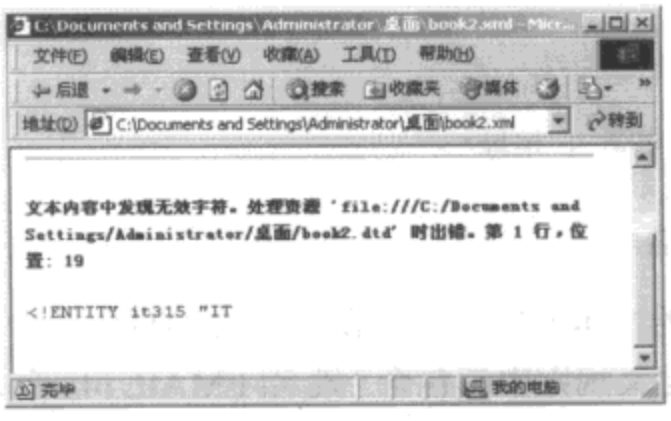


图 1.10

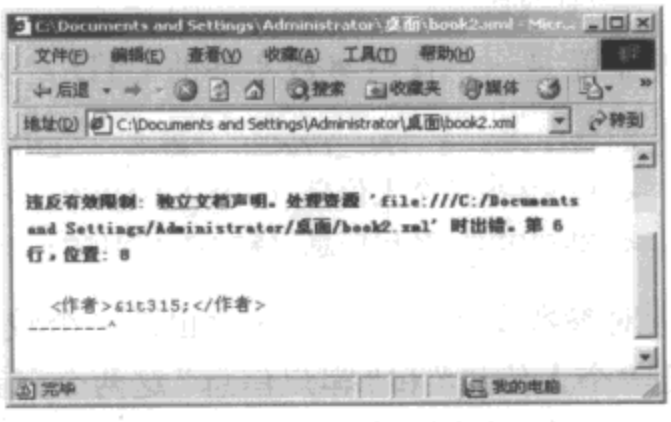


图 1.11

这个错误是因为 book2.xml 文件中的 XML 声明语句的 standalone 属性被设置成了"yes"，如下所示：

```
<?xml version="1.0" encoding="GB2312" standalone="yes"?>
```

book2.xml 文件中又引用了 book2.dtd 文件中定义的实体，显然 book2.xml 不能再独立使用，这与上面的 XML 声明语句产生了矛盾。

04 将 book2.xml 文件中的 XML 声明语句的 standalone 属性被设置成了"no"，或者干脆删除 standalone 属性设置部分（因为使用 DOCTYPE 声明语句后，即使在 XML 声明语句中省略了 standalone 属性，如果有必要，解析器也会自动从 DTD 文件中获取信息）。用 IE 浏览器重新打开 book2.xml 文件或直接在图 1.11 中单击工具栏中的“刷新”按钮，浏览器中显示的结果如图 1.12 所示。

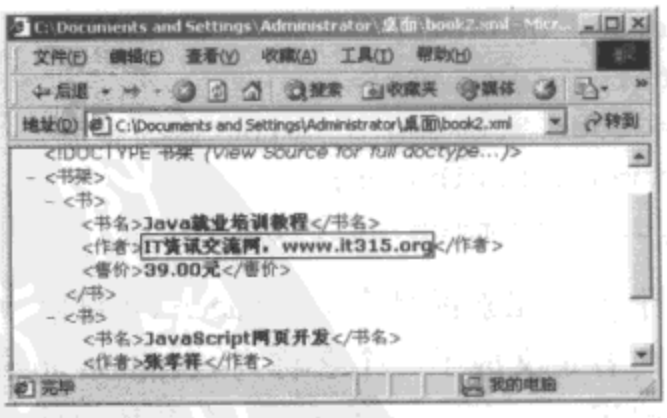


图 1.12

可见，book2.xml 文件中的“&it315;”被显示成了“IT 资讯交流网，www.it315.org”。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

2. 参数实体

参数实体被 DTD 文件自身使用，它的定义语法格式为：

```
<!ENTITY % 实体名称 "实体内容">
```

对参数实体进行引用的方式为：

```
%实体名称;
```

下面是一个使用参数实体的 DTD 定义：

```
<!ENTITY % TAG_NAMES "姓名 | EMAIL | 电话 | 地址">
<!ELEMENT 个人信息 (%TAG_NAMES; | 生日)>
<!ELEMENT 客户信息 (%TAG_NAMES; | 公司名)>
```

对参数实体的引用实际上也是一个较严格的文本替换过程。在上面的 DTD 定义中，由于“个人信息”和“客户信息”这两个元素的定义规则中都包含“姓名| EMAIL |电话 | 地址”这一相同的部分，所以，可以将这部分内容定义成一个名称为 TAG_NAMES 的参数实体，然后将“个人信息”和“客户信息”这两个元素的定义规则中的“姓名| EMAIL | 电话 | 地址”部分替换成对 TAG_NAMES 这个参数实体的引用。

参数实体主要用于在 DTD 中创建可重复使用部件，便于重复使用元素和属性定义中的相同部分。使用参数实体可以构造灵活的 DTD 定义，当 DTD 的元素和属性定义中要出现大量相同内容时，使用参数实体就是一种非常不错的手段。和引用实体一样，更改参数实体的值会更改对所有引用了该参数实体的元素或属性的定义。下面是重复使用属性定义的情况：

```
<!ENTITY % common.attributes
    'id ID #IMPLIED
    account CDATA #REQUIRED'
>
<!ELEMENT purchaseOrder (item+, manufacturer)>
<!ELEMENT item (price, quantity)>
<!ELEMENT manufacturer (#PCDATA)>
<!ATTLIST purchaseOrder %common.attributes;>
<!ATTLIST item %common.attributes;>
<!ATTLIST manufacturer %common.attributes;>
```

在上面的代码中，由于多个元素都具有 id 和 account 这两个属性的相同定义，所以，将定义这两个属性的文本内容定义成了一个名称为 common.attributes 的参数实体，然后在每个元素的属性定义中都引用 common.attributes 这个参数实体，该参数实体将被转换成对 id 和 account 这两个属性进行定义的文本内容。如果 DTD 定义中的数个约束具有相同的基本内容，但后来要对这个基本内容进行修改，这通常需要对每个约束都进行重复相同的修改过程。如果使用参数实体对这个基本内容进行定义，然后在每个约束中反复使用该参数

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

实体，当要对这个基本内容进行修改时，只需在参数实体的定义处进行修改即可。可见，参数实体可以实现模块化的 DTD，即模块化的约束模式。

最后需要提醒读者，无论人们还是计算机程序软件，要想理解 DTD 文档中的各个元素或属性的语义，仅仅能够看懂 DTD 文档的语法还不够，还必须掌握相关的领域知识。因为 DTD 定义了 XML 文档必须遵循什么样的结构，规定了 XML 文件中可出现的元素名、属性名，以及元素之间的嵌套关系，但是 DTD 并没有表达元素的语义，即元素所代表的具体意义和作用。元素的语义存在于文档之外，它存在于文档作者和读者的心中或者存在于生成和读取文档的计算机程序软件中。例如，对于下面的 DTD 定义语句：

```
<!ELEMENT filter-mapping (filter-name, (url-pattern | servlet-name))>
```

你很容易看懂这条语句的 DTD 含义，它表示在 XML 文档里可以出现的名称为 filter-mapping 的元素，该元素中必须嵌套两个子元素，第一个子元素必须是 filter-name，第二个子元素必须是 url-pattern 和 servlet-name 中的一个。但是，你知道 filter-name 元素代表的是什么东西吗？url-pattern 和 servlet-name 元素代表的又是什么东西呢？显然，只有那些具备相关领域知识的人们参看该 DTD 文档制定者提供的说明书后，才能明白。用于处理 XML 文档的程序软件也需要知道每个元素所代表的意义，才能知道在程序中该如何使用所读取到的各个元素。

1.6 XML Schema

如同 DTD 一样，XML Schema 也是一种用于定义和描述 XML 文档结构与内容的模式语言，其出现的目的是为了克服 DTD 的局限性，比如 XML DTD 并不能很好地实现应用程序的不同模块间的相互协调，缺乏对文档结构、属性、数据类型等约束的足够描述等，并为 XML 文档提供更丰富的语法结构。另外，XML DTD 不使用 XML 语法，采用的完全是另外一种专用的表示方法，并且对类型和名称空间仅提供有限的支持。W3C 希望以 XML Schema 来作为 XML 模式描述语言的主流，并逐渐代替 XML DTD。

1.6.1 XML Schema 与 DTD 的比较

XML Schema 相比 XML DTD，具有如下一些显著优点：

（1）由于 XML Schema 符合 XML 语法结构，XML Schema 文件本身也是一种 XML 文档，所以，XML 用户在使用 XML Schema 时能很容易地理解 XML Schema 的语法格式，无须重新学习。

（2）一些常用的 XML API，例如 DOM 和 SAX，它们只能处理 XML 实例文档，并没有提供解析 XML DTD 文档内容的功能，无法通过 DOM 或 SAX 来得到 DTD 中的描述信息。如果一些应用程序需要得到 DTD 本身的描述内容和结构，例如使用关系数据库来存储 XML 文档中的数据时，就涉及如何将 XML DTD 映射为关系模式的问题。由于没有创建和访问 DTD 的标准编程接口，无法使用标准的编程方式来操作 DTD 文档；为了实现对

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

XML DTD 的解读，研究人员必须为 XML DTD 开发新的接口或者专用工具。由于 XML Schema 作为 XML 的一个应用，XML Schema 本身就是一个 XML 文档，可以使用通用的 XML 解析器解析它，例如，使用 DOM、SAX 或 JDOM 等 XML API 就可以很容易解析出 XML Schema 文档中的内容。所以，XML 的编辑工具、API 开发包及语法分析器都可以直接用于处理 XML Schema 文件。

（3）XML 有非常高的合法性要求，XML DTD 和 XML Schema 都用于对 XML 文档进行描述，都被用做验证 XML 合法性的基础。但是，XML DTD 本身的合法性的验证必须采用另外一套机制，XML Schema 则采用与 XML 文档同样的合法性验证机制。

（4）XML Schema 对名称空间支持得非常好，而 XML DTD 几乎不支持名称空间。

（5）DTD 支持的数据类型非常有限，在大多数应用环境下都显得能力不足。例如，DTD 可以指定一个元素中必须包含字符文本（PCDATA），但无法指定一个元素中必须包含非负整数（nonNegativeInteger）之类的类型；而 XML Schema 比 XML DTD 支持更多的数据类型，并支持用户自定义新的数据类型。

（6）DTD 定义约束的能力也非常有限，无法对 XML 实例文档做出更细致的语义限制，例如，无法很好地指定一个元素中的某个子元素必须出现 7 到 12 次；而 XML Schema 定义约束的能力非常强大，可以对 XML 实例文档做出细致的语义限制。

（7）在将关系数据表示为 XML 数据和实现基于关系数据库的 XML 数据存储、查询和更新等问题上，XML Schema 基本上满足关系模式在数据描述上的需要，比 XML DTD 更适合描述关系数据。

当然，XML Schema 也有一些缺陷，例如，不能像 DTD 一样定义实体，XML Schema 比 DTD 更复杂。一般的用户无须自己编写 XML Schema 文档，作为通常的学习和应用需要，读者只需要知道 XML Schema 与 XML DTD 所要实现的基本功能和目的是一样的，而不用去考虑两者的语法格式的显著区别。

1.6.2 一个 XML Schema 例子

XML Schema 文件称为模式文档，而采用某个 XML Schema 文档作为约束模式的 XML 文档称为实例文档，XML Schema 模式文档的扩展名通常为.xsd。例程 1-14 是一个采用 XML Schema 作为其模式文档的 XML 实例文档，对于其中目前还没有讲解的细节，读者暂且不用理会。

例程 1-14 xmlbook.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<书架 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="xmlbook.xsd">
    <书>
        <书名>JavaScript 网页开发</书名>
        <作者>张孝祥</作者>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        <售价>28.00 元</售价>
    </书>
</书架>
```

从例程 1-14 的 XML 实例文档中的第三行上，很容易猜测出它所引用的模式文档名称为 xmlbook.xsd，该模式文档的内容如例程 1-15 所示。

例程 1-15　xmlbook.xsd

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name='书架'>
        <xs:complexType>
            <xs:sequence maxOccurs='unbounded'>
                <xs:element name='书'>
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name='书名' type='xs:string'/>
                            <xs:element name='作者' type='xs:string'/>
                            <xs:element name='售价' type='xs:string'/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

尽管前面的 XML 实例文档非常简单，但是与之对应的 XML Schema 文件 xmlbook.xsd 看起来却有点复杂，下面是对 xmlbook.xsd 文件内容的解释：

（1）<schema>元素为所有 XML Schema 模式文档的根元素，它用于声明该 XML 文档是一个 XML Schema 模式文档。

（2）由于 XML Schema 模式文档本身也是一个 XML 文档，它们里面的词汇都是有特定含义和作用的，这些词汇来自另外一个统一规定的模式文档，该模式文档的名称空间为 "http://www.w3.org/2001/XMLSchema"。在 xmlbook.xsd 中，将该名称空间定义成使用 xs 前缀名来代表，所以，有关元素前都增加了 xs 前缀。关于名称空间的细节，稍后将进行详细讲解。

（3）模式文档中的每一个<element>元素用于定义实例文档中的一个元素，<element>元素可以是不包含任何子元素的空元素的形式，也可以是包含子元素的成对标签的形式。<element>元素定义的实例文档中的每个元素都对应一种类型，这个类型通过<element>元素的 type 属性来指定。

（4）如果模式文档中的<element>元素所定义的实例文档中的元素包含子元素或者具有

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

属性，那么，这个元素的类型即为复杂类型。在模式文档中必须使用<complexType>元素来定义复杂类型，复杂类型的定义可以直接嵌套在相关的<element>元素中，也可以在<element>元素的外部进行定义，然后在<element>元素中使用 type 属性来进行引用。在<complexType>元素中可以为它所定义的元素类型指定子元素或属性，其中用于声明子元素的语句必须嵌套在<sequence>、<choice>等标签内。<sequence>、<choice>等标签用于表示子元素的各种组合关系，例如，<sequence>标签用于说明各个子元素在实例文档中的出现顺序必须与它们在模式文档中声明时的排列顺序一致，<choice>用于说明在实例文档内只需出现其中声明的各个子元素之中的任何一个。注意，即使<complexType>元素中定义的子元素只有一个，那也必须将声明子元素的语句嵌套在<sequence>、<choice>等标签内。

（5）<sequence> 元素中的 maxOccurs='unbounded' 属性设置用于说明嵌套在<sequence> 元素中的子元素定义可以在 XML 实例文档中出现多次，上不封顶。

1.7 名称空间

1.7.1 名称空间的概念

因为 XML 标记可以由任何组织和个人自行定义，因此在不同的约束模式文档中，出现表示不同含义的相同标记名称是完全有可能的。例如，有一个约束模式文档中使用<table>表示桌子，而在另一个约束模式文档中使用<table>表示表格，如果要在一个 XML 文件中同时引用这两个约束模式文档，就会出现名称冲突问题。XML 从两个方面来解决这种名称冲突的问题：

（1）使用名称空间这个概念来区分每个约束模式文档，每个约束模式文档被赋以一个惟一的名称空间，每个名称空间都用一个惟一的 URI（Uniform Resource Identifier，统一资源标识符）表示。

（2）在 XML 实例文档中为元素增加前缀部分，元素名称前增加的前缀名称代表该元素所属于的模式文档的名称空间，来自不同模式文档的元素前增加的前缀部分是不同的，应用程序通过前缀部分来判断一个元素来自于哪一个约束模式文档。

1.6.2 节中的 xmlbook.xsd 文件是一个 XML Schema 模式文档，但它本身也是一个 XML 格式的文档，它里面使用的各个元素名就是在另一个 XML Schema 模式文档中定义的，所以，xmlbook.xsd 文件中定义了一个前缀名来代表那个 XML Schema 模式文档的名称空间，并在每个元素前都加上这个前缀名。例如，在 xmlbook.xsd 文件的开始部分有如下两句内容：

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name='书架'>
    ……
```

上面的语句定义 xs 前缀名来代表名称空间"http://www.w3.org/2001/XMLSchema"，schema 和 element 元素前都增加了 xs 前缀来说明它们是来自"http://www.w3.org/2001/XMLSchema"名称空间所对应的模式文档中的词汇。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

URI 比 URL（Uniform Resource Locator，统一资源定位符）表示的概念更大，它不仅限于描述 Internet 资源的地址，还包括数字对象标识符（DOI）和国际标准书号（ISBN）等标识名称。URI 更为注重资源而不太注重位置，从理论上说，URI 可以找出文档的镜像副本或是找出已经从一个站点移动到另一站点的文档。URI 仍然处于进一步的研究之中，被当前的软件所惟一支持的一种 URI 就是 URL。将域名作为名称空间的 URI 的一部分，可以很好地避免各个组织发布的名称空间之间的冲突，实际上作为名称空间使用的 URI 基本上都符合 HTTP 的 URL 语法。所以，在一般情况下，读者可以简单地等同 URI 和 URL。在 XML 实例文档中引入名称空间，就是以“前缀部分：本地部分”的形式作为元素或属性的完整名称，前缀部分是一个能惟一标识某个约束模式文档的 URI 的临时简写，本地部分则是元素或属性在该名称空间内定义的名字。

制订 XML 名称空间标准的目的就是为了解决 XML 文档中的名称冲突问题，但在制订 DTD 标准时并没有考虑到名称空间的问题。所以，XML DTD 对于名称空间支持得非常牵强，在业界一直就存着许多争论，甚至有人主张说 XML DTD 并不能支持名称空间。而 XML Schema 对名称空间支持得非常好，可以说它与 XML 名称空间是形影相随、密不可分的。

1.7.2 名称空间声明

名称空间声明就是在 XML 实例文档中为某个模式文档的名称空间指定一个临时的简写名称，这个简写名称将作为该名称空间的元素或属性的前缀名。名称空间声明与元素的属性定义非常类似，可以位于任何一个元素的开始标记中，并且在一个元素中可以声明多个名称空间。名称空间声明的基本格式为：xmlns:前缀名称="URI"，其中的前缀名称是名称空间的一个临时缩写，URI 需要使用引号（单引号或双引号）引起来。例程 1-16 是一个使用名称空间的例子程序。

例程 1-16 xmlbook.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<it315:书架 xmlns:it315="http://www.it315.org/xmlbook/schema">
    <it315:书>
        <it315:书名>JavaScript 网页开发</it315:书名>
        <it315:作者>张孝祥</it315:作者>
        <it315:售价>28.00 元</it315:售价>
    </it315:书>
</it315:书架>
```

在例程 1-16 中，it315 被作为多个元素名称的前缀部分，必须通过名称空间声明将它关联到惟一标识某个名称空间的 URI 上，xmlns:it315="http://www.it315.org/ xmlbook/schema"语句就是将前缀名 it315 关联到名称空间"http://www.it315.org/xmlbook/schema"上。可见，名称空间的应用就是将一个前缀（如 it315）绑定到代表某个名称空间的 URI（如 http://www.it315.org/xmlbook/schema）上，然后将前缀添加到元素名称的前面（例如，it315:

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

书）来说明该元素的领域范畴。

在任何元素中声明的名称空间，只对该元素及其中嵌套的所有子孙元素有效。一个名称空间声明可以作用到的那些元素和属性，就是该名称空间声明的作用域。如果在里层元素中声明的名称空间前缀与外层元素中声明的另外一个名称空间的前缀名相同，那么在里层元素及其子孙元素范围内，该前缀名对应的是里层元素中声明的名称空间。从<it315:书架 xmlns:it315="http://www.it315.org/xmlbook/schema">这条语句上可以看到，在声明名称空间的元素自身上也可以使用代表该名称空间的前缀，即“it315:书架”。在解析 XML 文档时，语法解析器将把名称中的前缀部分 it315 与相应的名称空间声明中的 URI 相关联，所以，上面的“书架”、“书”、“书名”、“作者”、“售价”等元素都是在"http://www.it315.org/xmlbook/schema"这个 URI 所标识的词汇表中的元素。这种用冒号（:）把前缀与本地部分进行分隔的名称，有一个专门的术语，即 QName（Qualified Name，限定名），它表示被限定在某个名称空间中的名称。

1.7.3 默认名称空间

在名称空间的声明中也可以省略前缀部分，其格式为：xmlns="URI"。以这种方式声明的名称空间将作为其作用域内的所有元素的默认名称空间，即该作用域内的所有没有指定前缀部分的元素都属于这个名称空间。例程 1-16 的 XML 文件改为默认名称空间的形式如例程 1-17 所示。

例程 1-17 xmlbook.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<书架 xmlns="http://www.it315.org/xmlbook/schema">
    <书>
        <书名>JavaScript 网页开发</书名>
        <作者>张孝祥</作者>
        <售价>28.00 元</售价>
    </书>
</书架>
```

在这个简化的 XML 文件中，虽然“书架”、“书”、“书名”、“作者”、“售价”等元素名称前面没有前缀，但是由于"http://www.it315.org/xmlbook/schema"被设置成了默认的名称空间，所以，它们仍然是"http://www.it315.org/xmlbook/schema"这个 URI 所标识的词汇表中的元素。虽然使用默认名称空间会减少一些书写工作量，而使用带前缀的非默认名称空间会增加一些书写工作量，但命名合理的前缀却可以为人们浏览 XML 文档提供便利，例如，book:title 很容易让人联想到 title 是 book 词汇表中定义的元素。

1.7.4 属性的名称空间

当在一个模式文档的词汇表中定义了某个元素及其属性，在 XML 实例文档中通常只需要将元素限定于该模式文档的名称空间中，而不需要将该元素的属性限定于名称空间

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

中。例如下面的 XML 语句：

```
<xi:include xmlns:xi="http://www.w3.org/2001/XInclude"
   href="http://example.com/std/defs" parse="xml" />
```

虽然 include 元素的 href 和 parse 属性也是在名称空间"http://www.w3.org/2001/ XInclude"中定义的，但是，在 XML 实例文档中不用为它们指定名称空间，处理程序必须能对此做出正确的解释。

一个元素中可以包含另外一个名称空间中定义的属性，例如，<movie:title xml: lang="fr">语句中使用了一个不是来自 movie 名称空间的属性。如果属性名称中有一个前缀部分，该属性将被限定在该前缀所指定的名称空间中。如果属性名称中没有前缀部分，那么该属性就不属于任何名称空间。属性不受默认名称空间的影响，即使在一个没有前缀部分的属性出现之前已经指定默认名称空间，该属性也不属于默认名称空间，XML 建议书是使用下面的示例来说明这一点的。

```
<x xmlns="http://www.w3.org" xmlns:n1="http://www.w3.org">
    <good a="1" n1:a="2" />
</x>
```

因为“http://www.w3.org/”被声明成默认名称空间，且默认名称空间对没有前缀的元素起作用，所以 x 和 good 都属于“http://www.w3.org/”这一名称空间。由于 n1 指向了名称空间“http://www.w3.org/”，所以属性 n1:a 中的 a 属性属于名称空间 http://www.w3.org/；由于无前缀的 a 属性不属于任何名称空间，所以，在同一个元素中定义这两个 a 属性也不会发生冲突。

1.7.5 xml:space 和 xml:lang 属性

在 XML 的规范中，定义了两个内置的属性，即 xml:space 和 xml:lang。在每一个 XML 实例文档中，前缀名“xml”都已经被隐式地绑定到了一个名称空间上，内置属性名称以 xml:作为前缀，自定义的属性名不能以 xml:作为前缀，否则在解析时将发生错误。

由于空格等字符在 XML 文件是有意义的，XML 解析程序会将它们一个个如实地交给下游程序去处理，xml:space 属性则是用于设置下游应用程序应该如何处理解析器传递的空格等字符。xml:space 是一个 Enumerated 类型的属性，它的设置值只能是"default"或"preserve"："default"表示应用程序可以自行随意处理空格等字符；"preserve"则表示应用程序应把空格等字符当做普通文本字符处理。如果没有设置 xml:space 属性，解析器默认采用"default"作为其设置值。下面的语句采用 xml:space 内置属性来明确告诉 XML 处理程序，把元素中的空格当做正式的普通字符与其他字符合并处理：

```
<网址 xml:space="preserve">    www.it315.org    </网址>
```

xml:lang 属性用于设置元素的本地化语言信息，ISO-639 规范中规定了代表各个国家和地区的本地化语言的名称，例如"en" 表示英文、"la"表示拉丁文、"zh"表示中文、"zh-CN"

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

表示中文（中国）、"zh-TW" 表示中文（中国台湾地区）。xml:lang 属性的默认值为"en"。不同的国家地区不仅使用的文字互不相同，而且人们使用货币、时间、数字等的表现格式也各不相同，例如，在美国要表示一个日期类型的属性，它采用的语句如下：

```
<product release-date="8/18/2002"/>
```

而在中国，相应的语句应该采用如下形式：

```
<product release-date="2002-08-18"/>
```

如果 XML 文档的处理程序把"2002-08-18"当做了美国的日期来解析，也就是按“月月/日日/年年年年”的格式来解析，那将无法得到正确的结果。所以，对于 XML 文档的处理程序来说，它必须知道应将"8/18/2002"和"2002-08-18"这样的文本串当做哪个国家或地区的日期来解析。xml:lang 属性正是用于解决这种国际化的问题，以帮助 XML 文档支持国际化的应用。只要将上面的语句修改成如下形式：

```
<product xml:lang="zh-CN" release-date="2002-08-18"/>
```

XML 文档的处理程序就知道应该将"2002-08-18"这个字符串当做中国的一个日期来解析，也就是按“年年年年-月月-日日”的格式来解析，它就可以得到 2002 年 8 月 18 日的正确结果。

如果在某个元素中设置了 xml:space 和 xml:lang 属性，那么，嵌套在该元素中的子孙元素都将沿袭其设置结果。如果在子孙元素重新设置 xml:space 和 xml:lang 属性，那么在子孙元素的范围内采用它们自己的设置结果。

1.8 引入 XML Schema 文档

1.8.1 使用名称空间引入 XML Schema 文档

如果应用程序要使用某个 XML Schema 文档来校验一个 XML 实例文档时，它首先必须获得该 XML Schema 文档的内容信息。声明 XML 实例文档遵循某个 XML schema 文档的方式之一就是使用名称空间，但是，表示名称空间的 URI 并不代表相应的 XML Schema 文件的存储位置，所以，应用程序不能根据 XML 实例文档中的名称空间声明部分的 URI 来直接获取 XML Schema 文档。例如下面的语句：

```
<it315:书架 xmlns:it315="http://www.it315.org/xmlbook/schema">
```

应用程序只知道"http://www.it315.org/xmlbook/schema"是代表某个名称空间的 URI，但它根据这个 URI 无法推测出该名称空间所对应的 XML Schema 文档的访问地址，自然也就无法获取 XML Schema 文档中的内容和对 XML 实例文档进行校验。所以，要使用 XML Schema 文档对 XML 实例文档进行校验，除了要声明名称空间外，还必须指定该名称空间所对应的 XML Schema 文档的存储位置。在声明名称空间的元素中可以设置一个 schemaLocation 属性来指定名称空间所对应的 XML Schema 文档的存储位置。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

schemaLocation 属性的设置值中包含两个部分：第一个部分就是名称空间的 URI；第二个部分就是该名称空间所标识的 XML Schema 文件的位置或 URL 地址，这两个部分之间用空格分隔。

假设"http://www.it315.org/xmlbook/schema"所标识的 XML Schema 文件的 URL 是"http://www.it315.org/xmlbook.xsd"，在 XML 实例文档中引入此名称空间及 XML Schema 文档的情况如例程 1-18 所示。

例程 1-18 xmlbook.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<书架 xmlns="http://www.it315.org/xmlbook/schema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.it315.org/xmlbook/schema
        http://www.it315.org/xmlbook.xsd">
    <书>
        <书名>JavaScript 网页开发</书名>
        <作者>张孝祥</作者>
        <售价>28.00 元</售价>
    </书>
</书架>
```

shcemaLocation 属性的作用就是用于指定名称空间所对应的 XML Schema 模式文档在哪里，这个属性必须与它所设置的名称空间的声明语句位于同一元素中，不能在声明名称空间的元素之后的元素中进行设置。schemaLocation 属性本身又是在另外一个公认的标准名称空间中定义的，这个公认的标准名称空间的标识 URI 为"http://www.w3.org/2001/XMLSchema-instance"。由于 schemaLocation 属性与使用它的那个元素不在同一名称空间中，所以，在使用 schemaLocation 属性时，必须在其前面加上表示其名称空间的前缀，并要声明该前缀所对应的名称空间。

XML 解析程序都应内嵌有那些公认的标准名称空间的 XML Schema 文档的内容信息，因此在使用公认的标准名称空间时，不用去寻找它们所对应的 XML Schema 文档的内容，自然也就不用指定 XML Schema 文档的存储位置。由于"http://www.w3.org/2001/XMLSchema-instance"属于公认的标准名称空间，因此不需要在 schemaLocation 属性中指定它所标识的 XML Schema 文档的位置。公认的标准名称空间与自定义名称空间的区别，就好比“中华人民共和国主席”与“it315 网站站长”的区别一样。“中华人民共和国主席”是一个名称空间，要与“中华人民共和国主席”进行联系时，还应找到“中华人民共和国主席”所对应的具体人物，但是人人都应该知道“中华人民共和国主席”是谁，所以不用再专门说明谁是“中华人民共和国主席”。“it315 网站站长”也是一个名称空间，要与“it315 网站站长”进行联系时，还应找到“it315 网站站长”所对应的具体人物，但“it315 网站站长”是谁，显然还需要专门进行说明。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

一个 XML 实例文档可能引用多个名称空间的名称，对于这种情况，schemaLocation 属性的设置值中可以包含多对名称空间与它们所对应的 XML Schema 文档的存储位置，每一对名称空间的设置信息之间也采用空格分隔。例程 1-19 是在同一个元素中引入多个名称空间及它们所对应的 XML Schema 文档的 XML 文档实例。

例程 1-19　xmlbook.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<书架 xmlns="http://www.it315.org/xmlbook/schema"
    xmlns:demo="http://www.it315.org/demo/schema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.it315.org/xmlbook/schema
                        http://www.it315.org/xmlbook.xsd
            http://www.it315.org/demo/schema http://www.it315.org/demo.xsd">
    <书>
        <书名>JavaScript 网页开发</书名>
        <作者>张孝祥</作者>
        <售价 demo:币种="人民币">28.00 元</售价>
    </书>
</书架>
```

1.8.2　不使用名称空间引入 XML Schema 文档

一个没有使用名称空间的 XML 实例文档也可以声明它遵循某个 XML schema 文档，但它只可以引入一个 XML Schema 文件对其进行约束。XML Schema 文档的位置通过 noNamespaceSchemaLocation 属性直接指定，noNamespaceSchemaLocation 属性也是在标准名称空间"http://www.w3.org/2001/XMLSchema-instance"中定义的。例程 1-20 是一个使用 noNamespaceSchemaLocation 属性来指定 XML Schema 文档的例子程序。

例程 1-20　xmlbook.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<书架 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="xmlbook.xsd">
    <书>
        <书名>JavaScript 网页开发</书名>
        <作者>张孝祥</作者>
        <售价>28.00 元</售价>
    </书>
</书架>
```

在例程 1-20 中，模式文档 xmlbook.xsd 与引用它的实例文档位于同一目录中。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

1.9 思考与实践

1. 用你自己的理解简述 HTML 标记与 XML 标记的区别。

2. 简述 XML 文档声明语句中的 encoding 属性的作用。

3. 对于如下的一段 XML 内容：

```
<element> <![CDATA[&lt;]]> </element>
```

它在浏览器中的显示结果是哪种情况？

A. <element> < </element>

B. <element> < </element>

C. 两个都不是

4. 简述 XML 约束模式的作用。简述什么是格式良好的 XML 文档和有效的 XML 文档。一个 XML 文档必须是有效的吗？

5. 当用 IE 浏览器打开一个使用了 DOCTYPE 声明语句的 XML 文档时，IE 会查找 DOCTYPE 声明语句中指定的 DTD 文件，但并不使用该 DTD 文件对当前 XML 文档进行有效性校验。请在 1.4.2 节的 book.xml 文件的基础上用一个实验来说明这种情况，然后使用 XML SPY（请自己从网上下载）这个工具软件体验一下使用 DTD 文件对 XML 文档进行有效性校验的过程。要求写出详细的操作步骤和实验细节，然后将完成的结果提交给 www.it315.org 站点评审，表现优异者将获得 www.it315.org 站点的奖励和专家的指点。

6. 请设计一个 XML 格式的例子文件，该文件能够表达出一个国家中的每个省及省长名称，每个省下面的每个市及市长名称，每个市下面的每个镇及镇长名称。

7. 对于如下的一段 XML 文档内容：

```
<书架 xmlns="http://www.it315.org/xmlbook/schema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.it315.org/xmlbook/schema
      http://www.it315.org/xmlbook.xsd">
 …
```

请描述清楚如下问题：

（1）xmlns="..."片断的作用是什么？

（2）xmlns:xsi="..."片断的作用是什么？

（3）xsi:schemaLocation="..."片断的作用是什么？

（4）哪些名称空间所对应的 XML Schema 文档的位置不用显示指定，为什么？

（5）元素<书架>自身属于哪个名称空间？

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

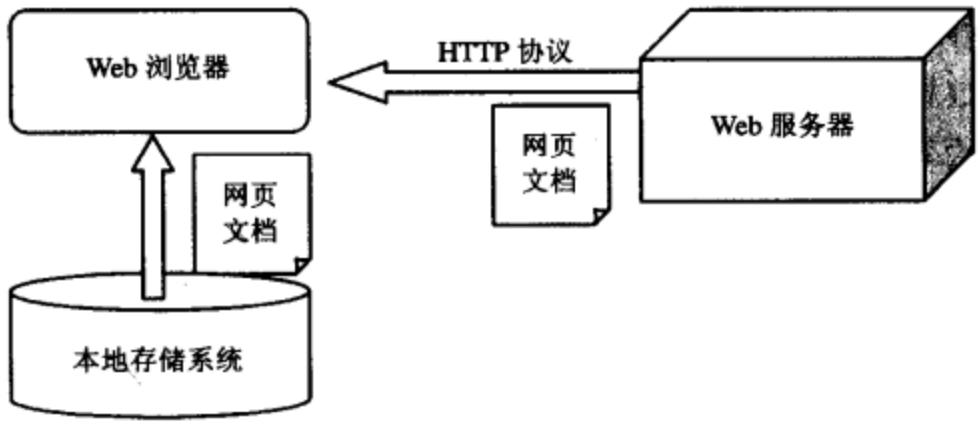
第 2 章　用 Tomcat 构建 Web 站点

2.1　Web 应用的相关知识

对于学习本书的读者来说，虽然网站、Web 浏览器和网页不应该是陌生的概念，但是还是必须对 Web 应用一些相关的知识有所了解。

2.1.1　HTTP 协议

Web 浏览器除了可以从本地硬盘上打开网页文档外，还可以使用 HTTP 网络协议从网络上的 Web 服务器（也叫 WWW 服务器）上获取网页文档的内容，如图 2.1 所示。



Web 浏览器与 Web 服务器分别代表着 TCP 网络程序的客户机和服务器。可以把 Web 服务器想象成在网络上放置的一块硬盘，网页文档就放在那个硬盘上，Web 浏览器要从远程网络上的硬盘中读取数据远比直接从本地硬盘上读取数据复杂，因为它与 Web 服务器之间需要一问一答的交互过程。

就像两个国家元首的会晤过程得遵守一定的外交礼节一样，Web 浏览器与 Web 服务器的交互过程也得遵循一定的规则，这个规则就是 HTTP 协议。Web 浏览器与 Web 服务器建立 TCP 网络连接后，Web 浏览器按照 HTTP 协议的规定向 Web 服务器发出请求信息，Web 服务器接收到浏览器的请求后，再按照 HTTP 协议的要求将结果发送给 Web 浏览器。在正常情况下，Web 服务器返回的结果中包含 Web 浏览器请求的网页文档内容，Web 浏览器接收到这些内容后，再进行解释和显示。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

2.1.2 URL

放置在 Internet 上的 Web 服务器中的每一个网页文件都应该有一个访问标记符，用于惟一地标示它的访问位置，以便 Web 浏览器能够定位到它。这个访问标记符称为 URL，即 Uniform Resource Locator 的英文简写，翻译成中文就是统一资源定位符。URL 地址中包含 Web 服务器主机名或 IP 地址、文件（或其他资源）路径名称，以及使用的网络协议和端口号，例如，http://www.it315.org:80/java/index.html 就表示一个 URL。Web 浏览器向 Web 服务器请求某个网页文件时，必须在浏览器地址栏中指定该网页文件的 URL 地址。关于 URL 的详细信息，请参看笔者的《JavaScript 网页开发》一书。

2.1.3 主要的 Web 服务器软件

无论网站规模的大小，最起码的工作就是要在用做 Web 服务器的计算机上安装某种 Web 服务器软件。目前主要的两种 Web 服务器软件是 Microsoft Internet Information Server（简称 IIS）和 Apache，但是它们都不直接支持本书所讲解的 Servlet 和 JSP 程序。要想在这些 Web 服务器软件中运行 Servlet 和 JSP，还必须将它们与某种 Servlet 和 JSP 容器（引擎）进行集成。

Tomcat 是 Sun 公司推荐的运行 Servlet 和 JSP 的容器，它可以被集成到 IIS 和 Apache 中。另外，Tomcat 也具有 Web 服务器的一些基本功能，对于一般的应用情况，也可以直接将 Tomcat 作为 Web 服务器软件运行。因本书主要是讲解基于 Java 的 Web 站点开发，所以使用 Tomcat 作为 Servlet 和 JSP 的容器，顺便也将使用 Tomcat 作为独立的 Web 服务器运行软件。

2.1.4 网站系统的组成

只要在一台计算机上安装了 Web 服务器软件，从功能上讲，这台计算机就可以称为 Web 服务器。一个网站的规模可大可小，功能可多可少，最简单的网站只需要一台 Web 服务器即可对外提供网页浏览服务。复杂的网站包括多台 Web 服务器组成的集群系统、负载均衡设备、具有缓存功能的代理服务器（可以有多级，甚至包括放置在服务器端的缓存系统）、数据库系统等，如图 2.2 所示。

www.sina.com 网站系统目前采用的基本上就是图 2.2 所示的架构，不同地区的人们在访问 www.sina.com 站点时，浏览器实际上所访问的服务器是不一样的，例如，吉林省的用户访问的服务器实际是 Sina 放在吉林地区的代理服务器，湖北省的用户访问的服务器实际是 Sina 放在湖北地区的代理服务器。各地区的浏览器访问 www.sina.com 站点的过程如图 2.3 所示。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

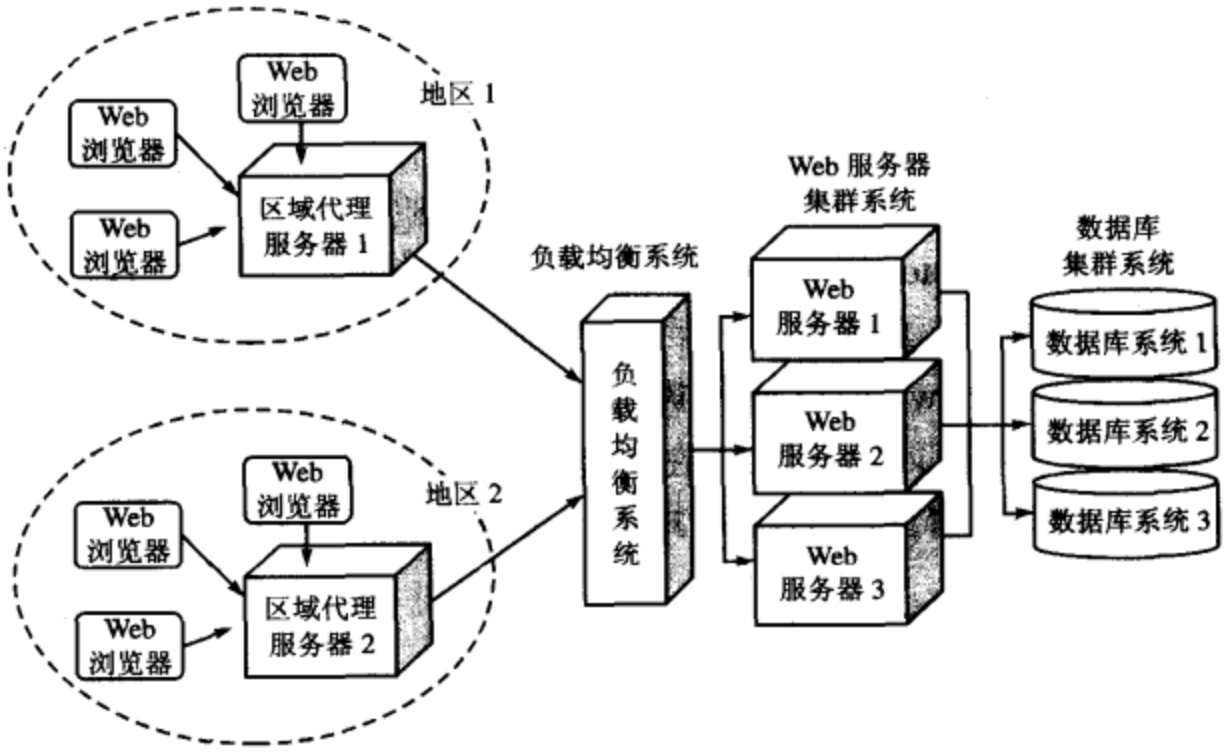


图 2.2

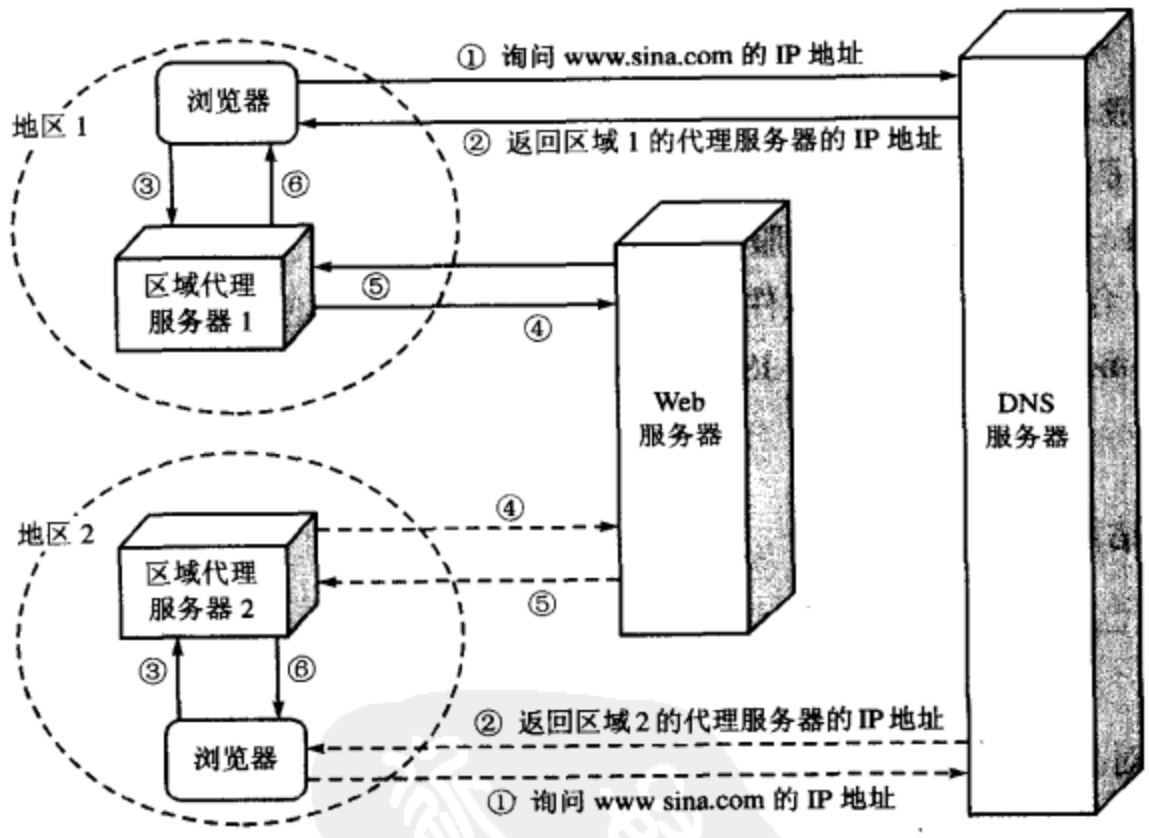


图 2.3

为了能够让浏览器透明地访问到 Web 站点，让用户感觉不到是在访问区域代理服务器，在 DNS 系统中需要将 www.sina.com 主机名指向所有区域的代理服务器的 IP 地址。在浏览器访问 www.sina.com 站点中的页面而向 DNS 服务器请求解析 www.sina.com 主机名时，DNS 服务器根据访问者的地理位置信息返回他附近的区域代理服务器的 IP 地址，这样，浏览器的访问请求将发送给该区域代理服务器。只有当区域代理服务器中没有浏览器要访问的页面时，区域代理服务器才去从真正的 www.sina.com 站点服务器上获取该页面

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

并进行缓存，以后该区域的其他浏览器就都可以就近从区域代理服务器中访问到该页面了，从而大大提高访问效率且减少骨干网络的流量。

2.2 安装 Tomcat

2.2.1 Tomcat 简介

Tomcat 是 Apache 组织的 Jakarta 项目中的一个重要子项目，它是 Sun 公司推荐的运行 Servlet 和 JSP 的容器（引擎），其源代码是完全公开的。另外，Tomcat 还具有 Web 服务器的基本功能，提供数据库连接池、SSL、Proxy 等许多通用组件功能。Tomcat 的应用已经非常广泛，越来越多的软件公司和开发人员都使用它作为运行 Servlet 和 JSP 的平台。Tomcat 也可以作为独立的 Web 服务器软件运行，但它处理静态 HTML 文件的速度比不上 Apache 和 IIS 等专业的 Web 服务器，且其作为 Web 服务器软件的功能也不如 Apache 和 IIS 强大。如果有必要，可以将 Tomcat 与 Apache 或 IIS 集成起来，让 Apache 或 IIS 处理静态 HTML 文件，Tomcat 则专门用于处理用 Servlet 与 JSP 等编写的“动态网页”。

Tomcat 的版本在不断地升级，功能也不断地完善与增强。笔者经历了从 Tomcat 3.x 到 Tomcat 5.x 等一系列的版本，发现每当 Tomcat 的主要版本号升级后，它的配置和使用等细节都有较大的变化。读者学习和工作中使用的 Tomcat 版本和一些书籍上所讲解的版本很难正好一致，因此，不能完全参照书籍去配置和使用 Tomcat，而是必须掌握一些基本的方法和思想。Tomcat 4.x 中的最新版本为 Tomcat 4.1.31，在本书编写时的 Tomcat 5.x 中的最新版本为 tomcat 5.5.12。Tomcat 4.x 实现了 Servlet 2.3 和 JSP 1.2 规范，Tomcat 5.x 实现了 Servlet 2.4 和 JSP 2.0 规范，通过后面相关内容的讲解，读者可以了解到这些规范之间的具体区别。虽然 Tomcat 5.x 对 Tomcat 4.x 进行了较大改变，但是由于 Tomcat 4.x 仍然被广泛使用和更适合于初学者入门学习，所以，本书将对这两个版本都进行讲解。

2.2.2 获取 Tomcat 安装程序包

进入 Apache 的 jakarta 项目的主页（http://jakarta.apache.org），可以看到 jakarta 项目的二进制格式的下载超链接。点击进入下载页面后，可以找到各种版本的 Tomcat 安装包，用途分别如下所述：

- jakarta-tomcat-4.1.31.tar.gz 和 jakarta-tomcat-5.5.12.tar.gz 文件是 Linux 操作系统下的安装版本。
- jakarta-tomcat-4.1.31.exe 和 jakarta-tomcat-5.5.12.exe 文件是 Windows 系统下的安装版本。
- jakarta-tomcat-4.1.31.zip 和 jakarta-tomcat-5.5.12.zip 文件是 Windows 系统下的压缩版本。

Tomcat 完全是由 Java 写成的，在使用前必须先安装合适的 JDK 版本。Tomcat 4.x 需要安装 JDK 1.2 以上版本，而 Tomcat 5.5.12 则需要安装 JDK 5.0 以上的版本。注意，Sun

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

公司并没有将 J2SE 1.4 以后的版本命名为 J2SE 1.5，而是命名成了 J2SE 5.0，相应的开发工具包也就被称之为 JDK 5.0 了。

2.2.3 使用 zip 和 tar.gz 包安装 Tomcat

在 Windows 上安装 Tomcat 的最简单方式就是将其安装程序的 zip 压缩文件解压到本地计算机上的一个目录中，jakarta-tomcat-4.1.31.zip 文件解压缩以后的目录结构如图 2.4 所示。

图 2.4

UNIX/Linux 下的安装方式与 Windows 下的这种解压安装方式差不多，只不过前者是对 tar.gz 格式的压缩包进行解压。其实，这种解压安装方式与直接从另外一台计算机上复制 Tomcat 安装后的整个根目录的效果差不多。下面是对图 2.4 中安装的 Tomcat 根目录下的一些主要子目录的介绍：

- bin　用于放置 Tomcat 的可执行文件和脚本执行文件。
- conf　用于放置 Tomcat 的配置文件。
- logs　用于放置 Tomcat 的日志记录文件。
- webapps　Web 应用程序的主要发布目录。
- work　Tomcat 的工作目录，JSP 文件翻译成的 Servlet 源文件和 class 文件放置在这里。

jakarta-tomcat-5.5.12.zip 文件解压缩以后的目录结构如图 2.5 所示。

从 Tomcat 4.1.31 和 Tomcat 5.5.12 安装后的结果目录上来看，其基本目录结构并没有太大改变，但目录中的内容有了一些改变。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

图 2.5

2.2.4 使用 exe 程序安装 Tomcat

jakarta-tomcat-4.1.31.exe 是一个普通的 Windows 安装程序，在使用它安装 Tomcat 之前，必须安装 JDK1.2 以上版本，最好是 JDK1.4.0 以上的版本。双击启动 jakarta-tomcat-4.1.31.exe 安装程序后，它会自动寻找 JDK 的主目录位置，如果没有找到，则提示你设置 JAVA_HOME 环境属性为 JDK 的主目录后重新进行安装。注意，如果要为这个安装程序的正常运行而设置 JAVA_HOME 环境属性，应该在 Windows 系统特性的“环境变量”对话框中进行设置，而不能是在某个命令行窗口中进行设置，因为后者设置的是一种临时环境变量，只对当前的命令行窗口有效，并不会影响到其他的命令行窗口和 Windows 程序。

使用 jakarta-tomcat-4.1.31.exe 进行安装将可增加一些启动和管理 Tomcat 的快捷方式，以及帮助用户修改 Tomcat 管理员账户和网络监听端口的默认设置，它所提供的安装选项如图 2.6 所示。

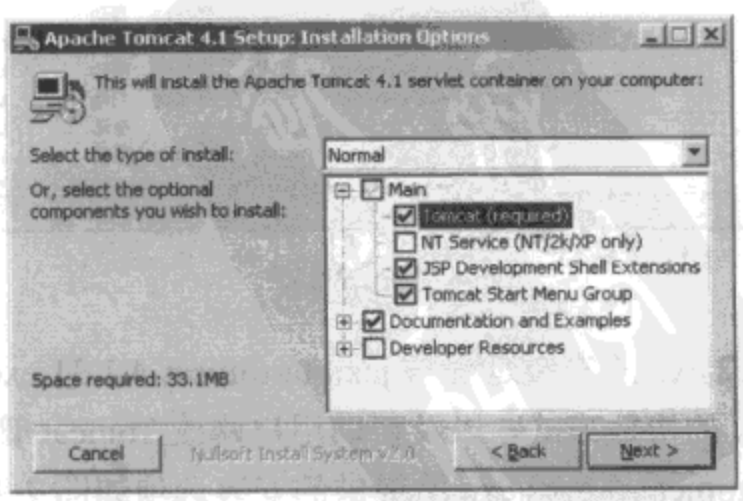


图 2.6

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

在图 2.6 中，如果选择“NT Service”项，Tomcat 将作为 NT 服务程序安装。这样，在每次启动操作系统和用户尚未登录进操作系统时，Tomcat 就可以自动启动并运行。如果选择“JSP Development Shell Extensions”项，以.jsp 为扩展名的文件将关联成双击时用 Windows 记事本程序打开。如果选择“Tomcat Start Menu Group”项，在 Windows 程序菜单中将添加启动和管理 Tomcat 的一些快捷菜单，如图 2.7 所示。

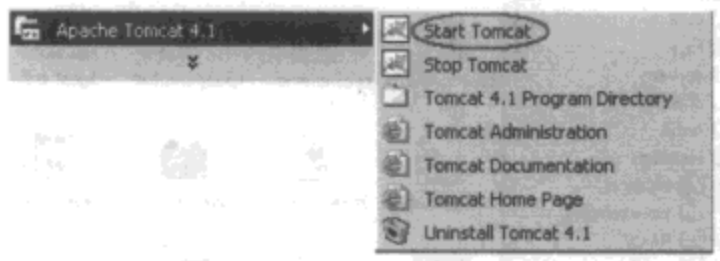


图 2.7

Web 服务器作为一个 TCP 服务器程序，必须绑定一个网络监听端口号，这个端口号通常都是可以配置的。jakarta-tomcat-4.1.31.exe 安装程序也会提示用户修改 Tomcat 管理员账户和网络监听端口的默认设置，如无特殊需要，一般不用对此进行修改。

如果在图 2.6 中不选择“Developer Resources”项（选择这项将安装 Tomcat 系统自身的源程序），那么，使用 jakarta-tomcat-4.1.31.exe 安装后得到的 Tomcat 目录结构与直接解压缩 jakarta-tomcat-4.1.31.zip 文件所得到的目录结构完全一样。

如果选择将 Tomcat 作为 NT Service 安装后，Tomcat 将可以作为 Windows 2000 Server 的服务程序在后台自动启动运行，没有图形用户界面或命令行窗口。单击 Windows 2000 Server 下的“开始”→“程序”→“管理工具”→“服务”菜单，在打开 Windows 2000 Server 的服务管理控制台中可以看到一个显示名称为“Apache Tomcat 4.1”的列表项，它对应的就是 Tomcat 4.1.31 服务程序，如图 2.8 所示。

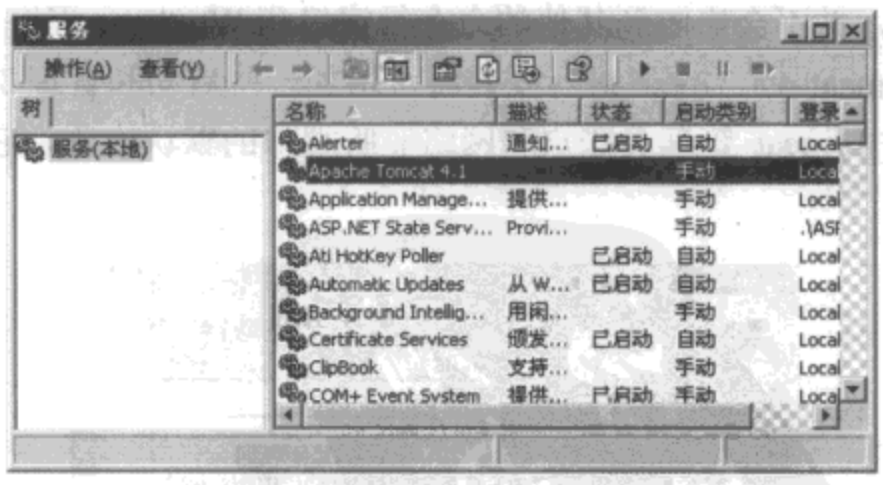


图 2.8

jakarta-tomcat-5.5.12.exe 的安装过程与 jakarta-tomcat-4.1.31.exe 差不多，同样可以增加一些启动和管理 Tomcat 的快捷方式，以及帮助用户修改 Tomcat 管理员账户和网络监听端口的默认设置。jakarta-tomcat-5.5.12.exe 所提供的安装选项如图 2.9 所示，将鼠标移动到某个选项上，可以看到每个选项的描述信息。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

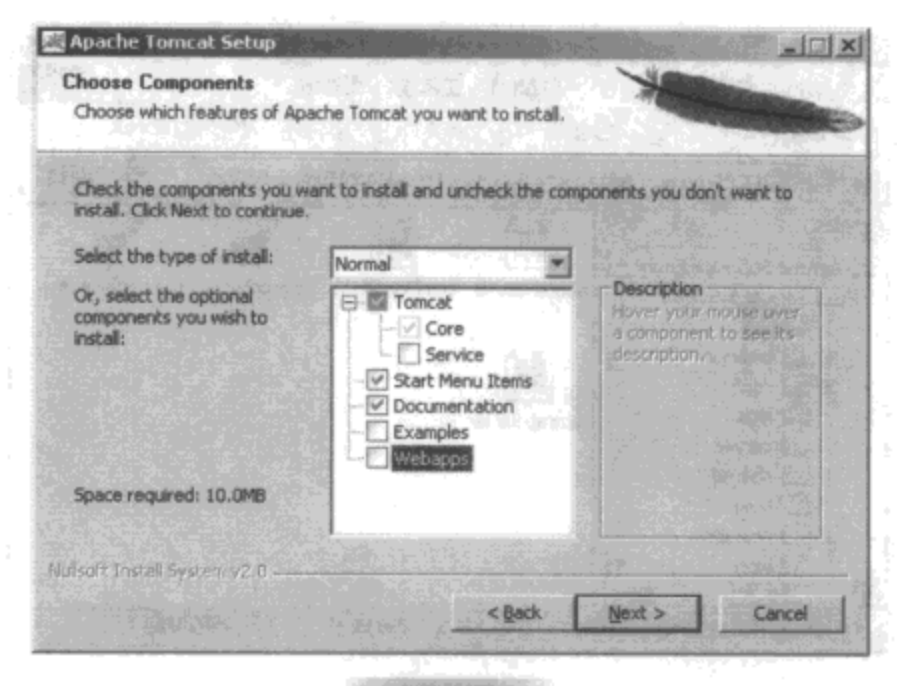


图 2.9

使用 jakarta-tomcat-5.5.12.exe 安装 Tomcat 时，计算机上至少必须安装有 J2SE 5.0 的 JRE（Java Runtime Environment，Java 运行时环境）部分。在 jakarta-tomcat-5.5.12.exe 程序的安装过程中，它会自动寻找 J2SE 5.0 的 JRE 主目录位置，并提示用户进行确认，如图 2.10 所示。

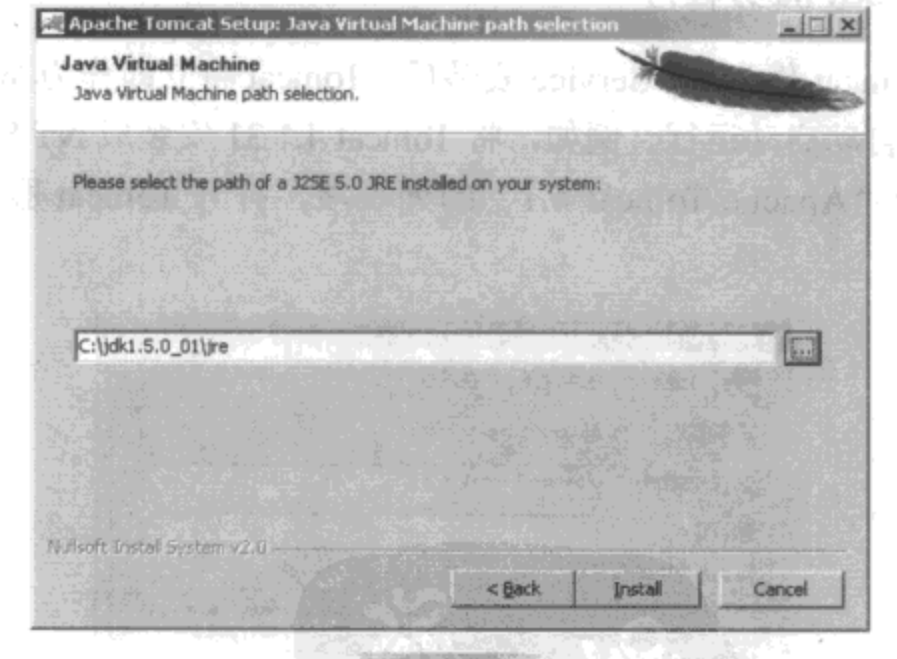


图 2.10

注意 在图 2.10 中填写的是 JRE 的主目录，而不是 JDK 的主目录。因为 Tomcat 5.5.12 默认使用 Eclipse JDT Java 编译器来编译 JSP 页面，通过这种 exe 程序安装的 Tomcat 运行时，只需要计算机上安装有 JRE 5.0，而不必安装完整的 JDK 5.0（JDK 中自带有 JRE 部分）。使用 jakarta-tomcat-5.5.12.exe 安装的 Tomcat 的 bin 子目录中的文件要直接解压缩 jakarta-tomcat-5.5.12.zip 后所看到的文件少得多，如图 2.11 所示。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

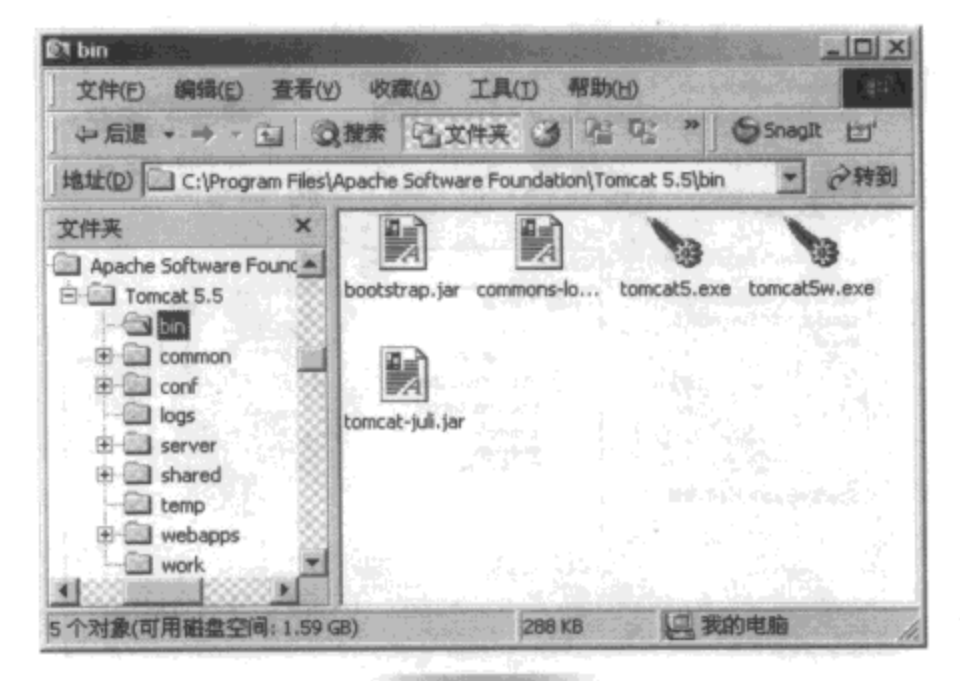


图 2.11

其中，tomcat5.exe 用于启动 Tomcat，tomcat5w.exe 用于配置 Tomcat 的运行参数。

2.3 Tomcat 的启动与停止

2.3.1 使用 Tomcat 服务程序

如果选择将 Tomcat 作为 NT Service 安装后，Tomcat 将立即作为 Windows 2000 Server 的服务程序在后台自动启动运行。例如，将 Tomcat 4.1.31 安装成 NT Service 后，在图 2.8 中双击显示名称为“Apache Tomcat 4.1”的列表项，打开 Tomcat 服务属性对话框，如图 2.12 所示。

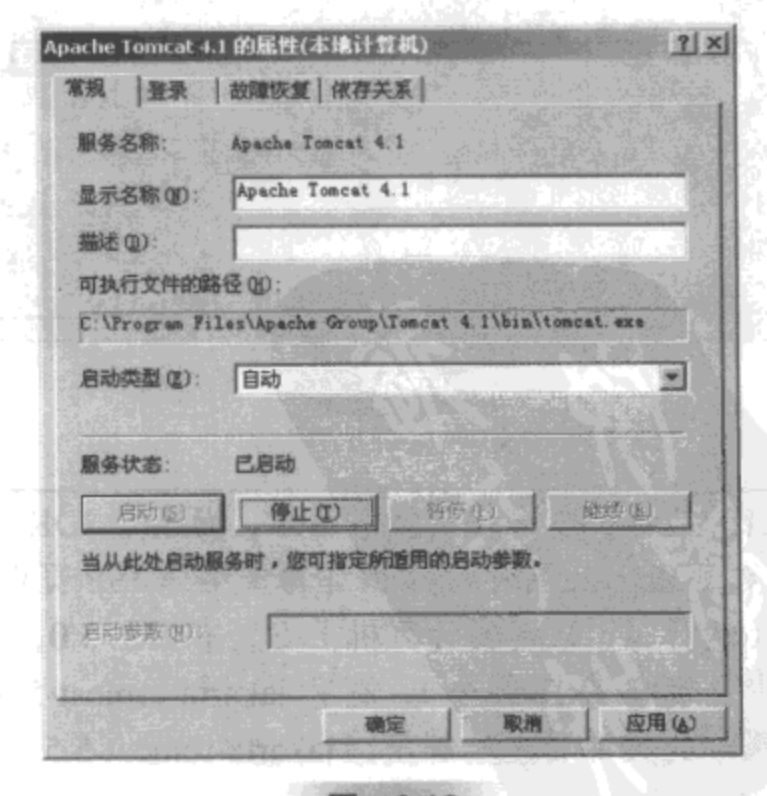


图 2.12

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

在图 2.12 中，Tomcat 服务程序目前处于启动运行状态，单击“停止”按钮，可停止该服务程序的运行。在图 2.12 中，还可以修改 Tomcat 服务程序的启动类型，设置为自动时，在每次启动 Windows 2000 Server 后，该服务程序自动启动运行；设置为手动时，需要每次启动 Windows 2000 Server 后，再单击图 2.12 中的“启动”按钮来启动服务程序。因为启动的服务程序越多，占用的系统资源（主要是内存和 CPU）就越多，整个系统的运行效率就越低。所以，一般总是将需要运行的服务程序设置为自动启动方式，偶尔需要运行的服务程序设置为手动方式，避免让计算机上安装的所有服务程序都随 Windows 2000 Server 的启动而自动启动运行。

如果知道某个服务程序的服务名称，也可以用命令方式来启动和停止该服务。以 Tomcat 服务程序为例，从图 2.12 所示的属性对话框中看到，Tomcat 服务程序的名称为“Apache Tomcat 4.1”，所以可以在命令行窗口中执行 `net start "Apache Tomcat 4.1"` 命令来启动 Tomcat 服务程序，执行 `net stop "Apache Tomcat 4.1"` 命令来停止 Tomcat 服务程序。由于 Tomcat 服务程序的服务名称中有空格，所以在命令行中应使用双引号将它们括起来。

正确安装 TCP/IP 协议后，计算机上都会有一个称为本地回路的 IP 地址：127.0.0.1，一个网络程序可以使用这个 IP 地址来访问同一台计算机上的其他网络程序，以及测试和使用本地计算机上的网络程序。另外，操作系统已将计算机名称 localhost 映射成 127.0.0.1，使用 localhost 这个名称访问计算机时，它将被转换成 127.0.0.1 这个 IP 地址。在默认情况下，Tomcat 的 Web 服务监听端口为 8080。

为了查看 Tomcat 服务程序启动运行后的效果，可以在 Tomcat 安装的本地计算机的浏览器地址栏中输入 http://localhost:8080 这个访问地址，如果修改过 Tomcat 的 Web 服务监听端口，则应将 8080 修改成相应的端口号。如果 Tomcat 服务程序正常启动，那么在浏览器中就可以显示出 Tomcat 返回的首页，如图 2.13 所示。

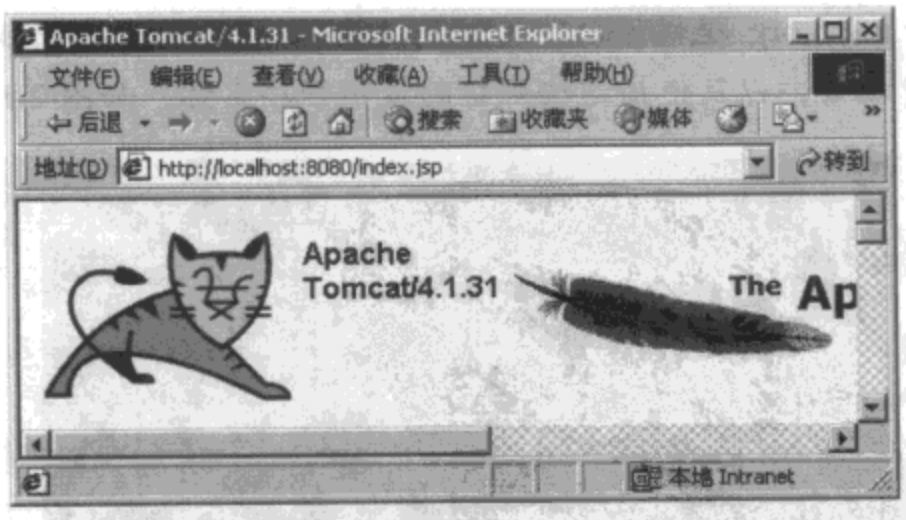
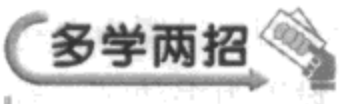


图 2.13

Tomcat 返回的首页左上角是 Tomcat 的版本信息，正文是关于这个版本的 Tomcat 的一些基本信息的介绍，包括该版本的 Tomcat 所实现的 Servlet 和 JSP 规范的版本信息。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享



查找网络服务程序监听异常的问题

如果确认图 2.12 中看到的 Tomcat 服务程序处于启动状态，但浏览器提示“该页无法显示”或者返回的不是 Tomcat 提供的首页，这很可能是因为 Tomcat 服务程序所使用的网络监听端口号已被其他网络服务程序或 Web 服务程序占用，导致 Tomcat 服务程序并没有真正正常启动运行。要想验证这种情况，先在图 2.12 中停止 Tomcat 服务程序，再进入命令行窗口中，执行 netstat-na 命令，查看 TCP 监听端口列表中是否包含 Tomcat 的 Web 服务绑定的监听端口，如图 2.14 所示。

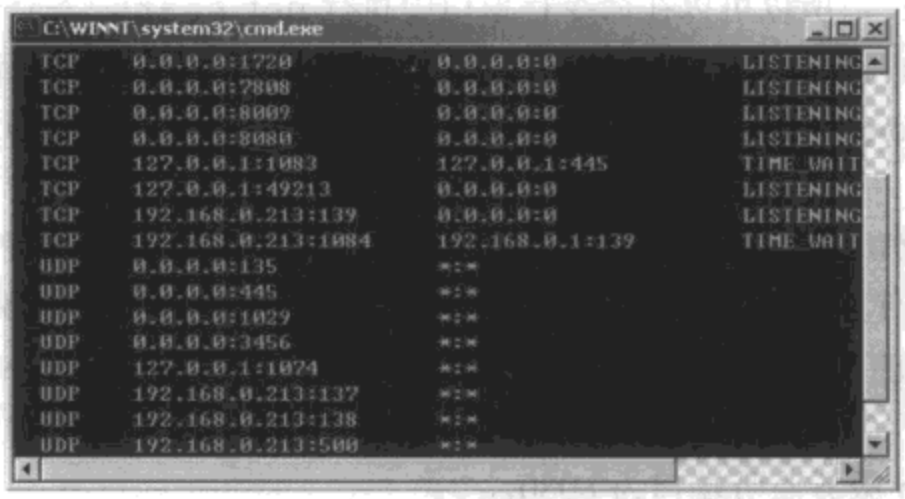


图 2.14

如果在图 2.14 中看见了为 Tomcat 的 Web 服务绑定的监听端口，这就说明该端口已经被其他网络程序所使用。使用 netstat -na 命令只能看到哪些网络端口号正在被使用，而无法查看到某个端口号是在被哪个应用程序所使用。使用一个名为 Fport 的工具软件（http://www.it315.org 提供该软件的下载链接），可以查看到本地计算机上所有打开的 TCP 和 UDP 端口，以及使用这些端口的应用程序名称，如图 2.15 所示。

图 2.15

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

可见，使用 netstat-na 命令或 Fport 工具可以查看哪些端口号在被使用，从而可以帮助查找某个网络程序的通信问题。

对于 Tomcat 5.5.12 来说，将其安装成 NT Service 后，只要在 Windows 2000 Server 的服务管理控制台中找到显示名称为“Apache Tomcat”的列表项，就可以按照 Tomcat 4.1.31 一样的方式进行操作和使用了。

2.3.2 使用 Tomcat 程序组中的快捷菜单

1. Tomcat4.1.31 的快捷启动菜单

如果使用 jakarta-tomcat-4.1.31.exe 安装 Tomcat 时选择 Tomcat Start Menu Group 项，那么也可以使用如图 2.7 所示的 Windows 程序菜单组中的“Apache Tomcat 4.1”组里面的“Start Tomcat”快捷菜单项来启动 Tomcat，单击这个菜单项后 Tomcat 将在一个新打开的命令行窗口中开始启动运行，如图 2.16 所示。

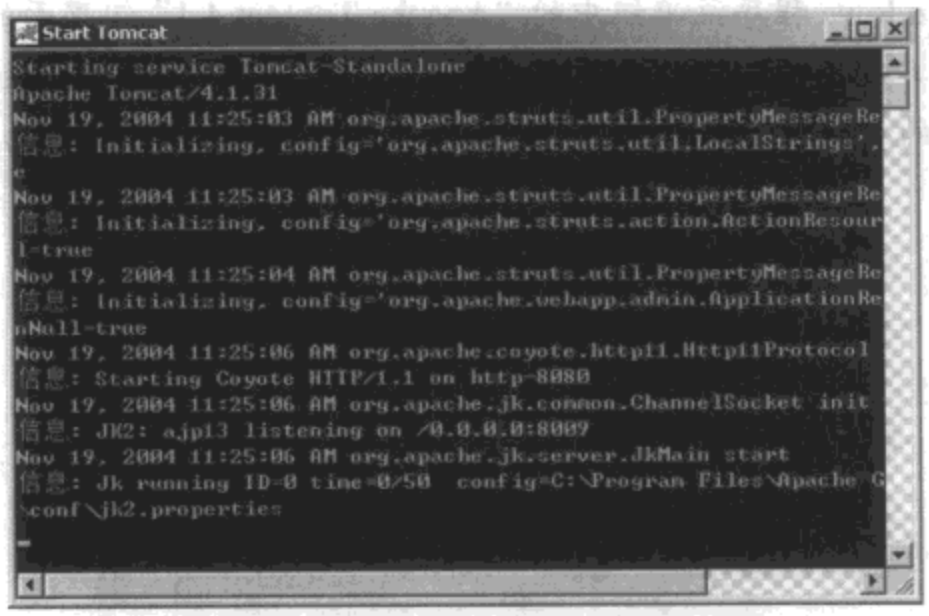


图 2.16

使用这种方式启动 Tomcat 的一个最常见问题就是作为 Windows 2000 Server 的服务程序安装的 Tomcat 已在后台启动运行，由于一个端口号不能同时被多个实例进程使用，所以使用这种方式再去启动 Tomcat 的第二个实例进程时，第二个实例进程就无法再使用 Tomcat 的监听端口号，从而发生异常而退出运行。只要停止与 Tomcat 使用相同的监听端口号的其他网络服务器程序的运行，通过这种快捷方式应能正常启动 Tomcat。如果在上面的启动过程中出现异常，图 2.16 所示的命令行窗口将会很快消失，人们根本无法查看到异常的提示信息，这样就很难找到解决问题的头绪了。对于解决这种启动时出现的问题的办法，在后面使用其他方式启动 Tomcat 的讲解中将进行详细介绍。

可以使用如图 2.7 所示的 Windows 程序菜单组中的“Apache Tomcat 4.1”组里面的“Stop Tomcat”快捷菜单项来停止已启动的 Tomcat 程序，也可以直接在图 2.16 的命令行窗口中按下 Ctrl+C 键停止 Tomcat 程序。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

多学两招

了解 Tomcat 4.1.31 的快捷启动菜单的工作原理

Tomcat 是一个完全用 Java 语言编写的程序，所以它应该可以使用如下的命令形式来启动：

```
java <各种命令选项> <Tomcat 的 Java 启动类>
```

另外，也可以在 Java 程序的启动类所位于的 Jar 包的 MANIFEST.MF 文件中，增加一个名称为 Main-Class 的属性，将它的属性值设置为 Java 程序的启动类，然后使用如下的命令形式来启动 Java 程序：

```
java -jar <其他各种命令选项> <启动类所在的 Jar 包>
```

下面来分析一下 Tomcat 4.1.3 的快捷启动菜单所使用的具体命令。将鼠标移动到图 2.7 所示的 Windows 程序菜单组中的“Apache Tomcat 4.1”组里面的“Start Tomcat”快捷菜单项上，单击鼠标右键，从弹出的快捷菜单中选择“属性”菜单项，在打开的“属性”对话框的“快捷方式”选项卡中可以看到快捷菜单所对应的具体命令，如图 2.17 所示。

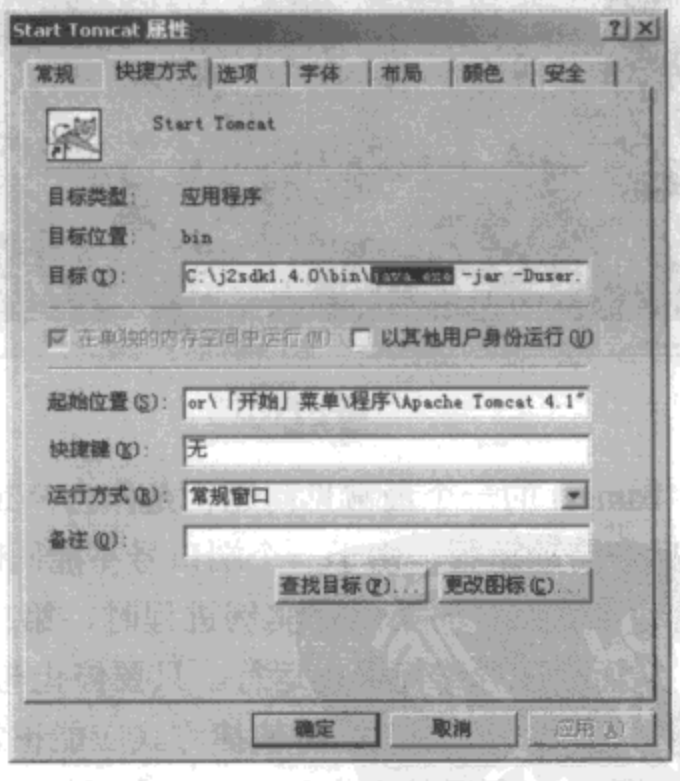


图 2.17

从图 2.17 中可以看到 Tomcat 4.1.31 的快捷启动菜单的完整命令为：

```
C:\j2sdk1.4.0\bin\java.exe -jar -Duser.dir="C:\Program Files\Apache Group\Tomcat 4.1"
-Djava.endorsed.dirs="C:\Program Files\Apache Group\Tomcat 4.1\common\endorsed"
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
"C:\Program Files\Apache Group\Tomcat 4.1\bin\bootstrap.jar" start
```

为了便于读者理解，笔者将上面的命令简化成如下形式：

```
java -jar bootstrap.jar start
```

其中，bootstrap.jar 是 Tomcat 启动类所在的 Jar 包，start 是传递给启动类中的 main 方法的参数。进入 Tomcat 主目录的 bin 目录中，用 WinRAR 软件打开 bootstrap.jar 文件，单击其中的 META-INF 目录，查看 MANIFEST.MF 文件的内容，如图 2.18 所示。

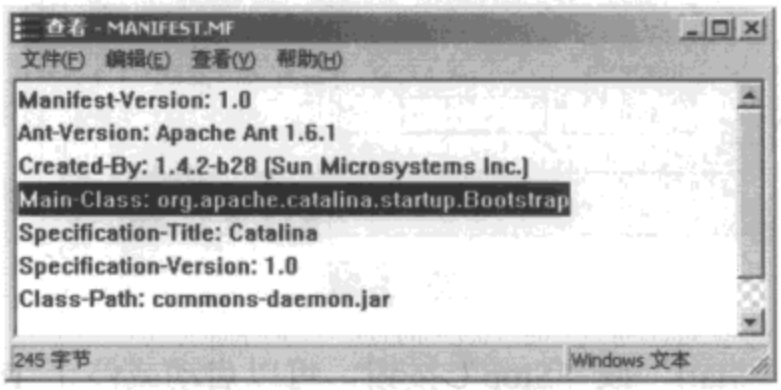


图 2.18

从图 2.18 中可以看到，Tomcat 的启动类名为 org.apache.catalina.startup.Bootstrap。

2. Tomcat 5.5.12 的快捷菜单

如果在安装 jakarta-tomcat-5.5.12.exe 程序时选择“Start Menu Items”项，在 Windows 程序菜单中将添加如图 2.19 所示的快捷菜单。

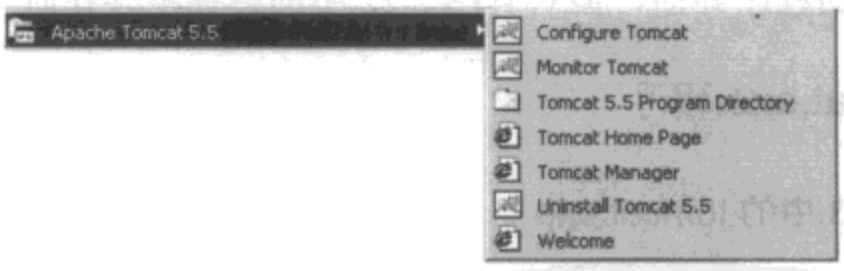


图 2.19

在图 2.19 所示的“Apache Tomcat 5.5”菜单组中，并没有包含直接启动 Tomcat 的快捷菜单项，其中，“Monitor Tomcat”菜单项用于监视 Tomcat 的运行状态，“Configure Tomcat”菜单项用于对 Tomcat 进行配置。单击“Monitor Tomcat”菜单，Windows 底部的任务栏的托盘区中将增加一个图标，如图 2.20 所示。

图 2.20

双击图 2.20 中的这个图标，将打开如图 2.21 所示的程序界面。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

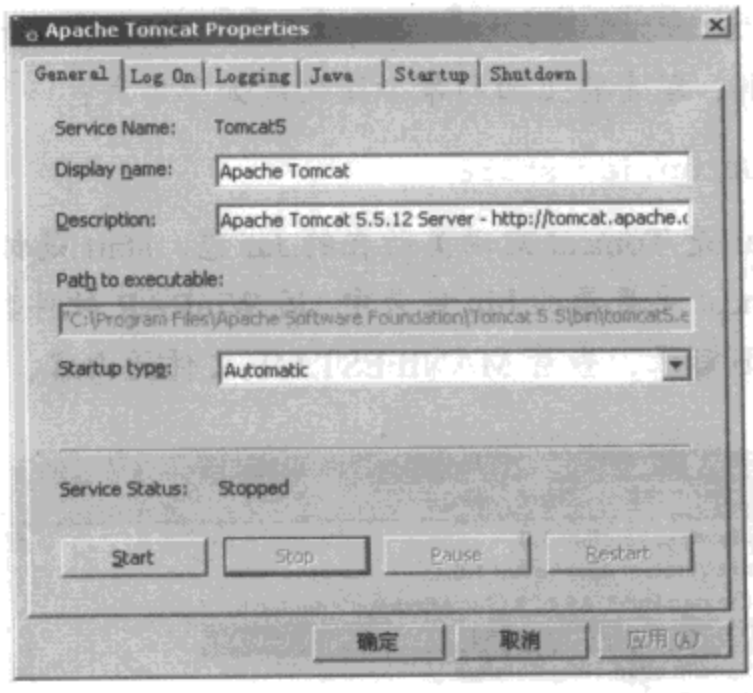


图 2.21

单击图 2.21 中的“Start”或“Stop”按钮，可以启动或停止 Tomcat 5.5.12 安装成的 NT Service 程序。如果 Tomcat 5.5.12 安装成的 NT Service 程序启动后，图 2.20 中的图标中间将变成一个绿色的指示箭头，如图 2.22 所示。

图 2.22

查看图 2.19 中的“Monitor Tomcat”和“Configure Tomcat”菜单项所使用的具体命令，可以看到这两个菜单都是指向图 2.11 中的 Tomcat5w.exe 程序，只是程序名称后面跟的选项参数不一样。Tomcat5w.exe 用于设置 Tomcat 的相关信息，直接运行 Tomcat5w.exe 将打开图 2.21 所示的程序界面。“Configure Tomcat”菜单的作用也是打开图 2.21 所示的程序界面，但只有图 2.22 中的图标不存在时才可以执行。

2.3.3 使用 tomcat.exe 程序

1. Tomcat 4.1.3 中的 tomcat.exe

从图 2.12 所示的属性对话框中看到，Tomcat 服务程序的可执行文件路径为<Tomcat 主目录>\bin\tomcat.exe，但直接执行这个程序并不能启动 Tomcat。tomcat.exe 是一个启动 org.apache.catalina.startup.Bootstrap 类的 Windows 外壳包装程序，并可将自身注册为 Windows 操作系统的服务程序和取消已注册的服务程序。在命令行窗口中执行 `tomcat -help |more` 命令可以看到 tomcat 的命令选项的介绍，例如，执行 `tomcat -uninstall "Apache Tomcat 4.1"` 可以删除服务程序的注册安装。也可以将 Tomcat 重新注册为 Windows 的服务程序，但命令的参数选项比较烦琐，以笔者的计算机环境为例，执行的命令是如下一长串文本：

```
tomcat -install "Apache Tomcat 4.1" " C:\j2sdk1.4.0\jre\bin\server\jvm.dll"
    -Djava.class.path="C:\jakarta-tomcat-4.1.31\bin\bootstrap.jar"
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
-Dcatalina.home="C:\jakarta-tomcat-4.1.31"
-Djava.endorsed.dirs="C:\jakarta-tomcat-4.1.31\common\endorsed"
-start  org.apache.catalina.startup.BootstrapService -params start
-stop org.apache.catalina.startup.BootstrapService -params stop
-out "C:\jakarta-tomcat-4.1.31\logs\stdout.log"
-err "C:\jakarta-tomcat-4.1.31\logs\stderr.log"
```

2. Tomcat 5.5.12 中的 tomcat5.exe

将 Tomcat 5.5.12 安装成 NT Service 后，在 Windows 2000 Server 的服务管理控制台中查看该服务程序的属性，可以看到 Tomcat 5.5.12 服务程序的可执行文件路径为图 2.11 中的 tomcat5.exe。使用 jakarta-tomcat-5.5.12.exe 安装 Tomcat 后，也可以直接执行 tomcat5.exe 来启动 Tomcat，但这样启动的 Tomcat 将在一个命令行窗口中运行，而不是以 NT Service 的方式运行的，如图 2.23 所示。

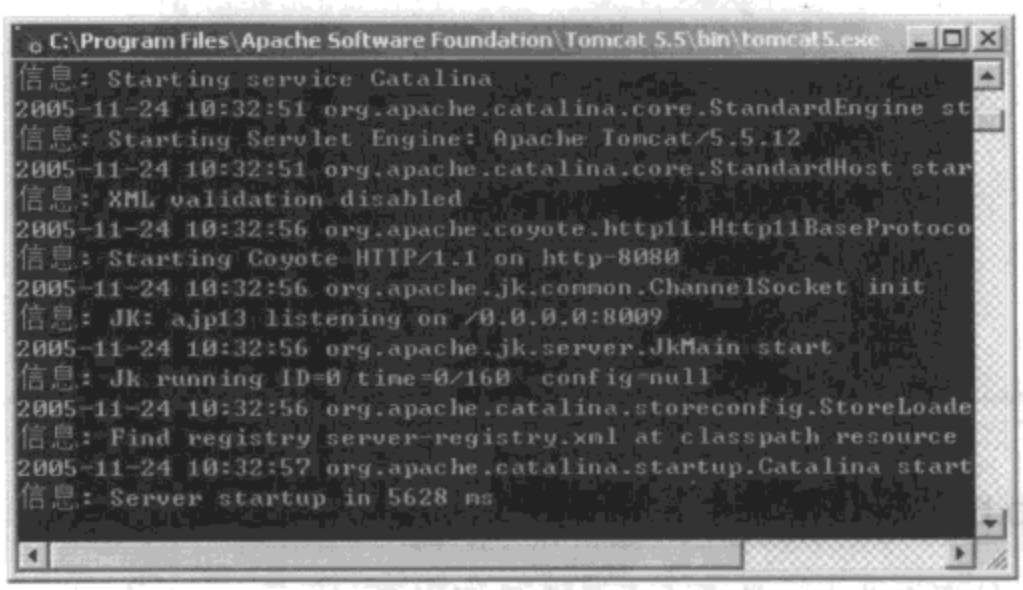


图 2.23

指点迷津

tomcat5.exe 启动的 Tomcat 程序不一定是 tomcat5.exe 自身所在的目录

tomcat5.exe 只是一个启动 org.apache.catalina.startup.Bootstrap 类的 Windows 外壳包装程序。如果同一台计算机上安装了多个 Tomcat 5.x 程序，一个 Tomcat 5.x 程序目录内部的 tomcat5.exe 启动的不一定是该 Tomcat 5.x 程序目录。也就是说，tomcat5.exe 到底启动哪个 tomcat 5.x 程序目录，与 tomcat5.exe 自身所处的目录没有关系，而由 windows 的注册表中的信息决定。例如，假设 jakarta-tomcat-5.5.12.zip 解压缩后的主目录为 C:\tomcat5，使用 jakarta-tomcat-5.5.12.exe 在同一台计算机上安装的另外一个 Tomcat 程序的主目录为 C:\Program Files\Apache Software Foundation\Tomcat 5.5，那么，C:\tomcat5\bin\tomcat5.exe 启动的将不是以 C:\tomcat5 为主目录的 Tomcat，而是以 C:\Program

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

Files\Apache Software Foundation\Tomcat 5.5 为主目录的 tomcat。这样的结果将导致你在 C:\tomcat5 中进行设置后，想当然地使用 C:\tomcat5\bin\tomcat5.exe 来启动它，结果启动的 tomcat 根本没有按照 C:\tomcat5 中的设置信息来运行，从而可能花费你很多时间去思考和查找这个问题。tomcat5.exe 启动的到底是哪个目录内的 tomcat，只要将运行 tomcat5w.exe 所打开的属性对话框切换到 Java 选项卡，查看一下相关的设置信息就知道了，如图 2.24 所示。

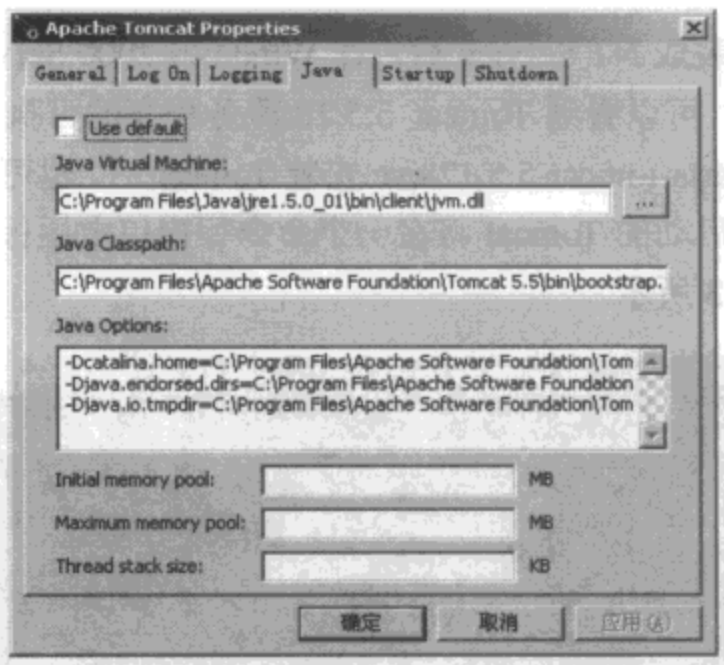


图 2.24

2.3.4 startup.bat 批处理文件

可以在一个扩展名为 bat 的文本文件中编写多条 Windows 命令和一些条件判断、跳转语句，以后只要执行一下这个文本文件，它里面的多条命令就会按照预设的流程去执行。这种用于批量执行多条 Windows 命令的文本文件称为 Windows 命令脚本文件或批处理文件。

无论 Tomcat 4.x，还是 Tomcat 5.x，采用解压缩 zip 安装包后得到的 Tomcat 主目录的 Bin 目录中有一个名为 startup.bat 的脚本文件，它也可以用于启动 Tomcat。Tomcat 4.x 和 Tomcat 5.x 中的 startup.bat 批处理原理和内容都很相似，下面以 Tomcat 4.1.31 内的 startup.bat 为例进行分析讲解。在图 2.4 所示的 Windows 资源管理器窗口中，选择 startup.bat 文件，单击鼠标右键，从弹出的快捷菜单中选择“编辑”菜单项，可以查看和修改 startup.bat 文件中的内容。想读懂 startup.bat 文件中的内容并不难，但需要有一定的耐心且掌握批处理文件中的一些命令和语句。下面结合 startup.bat 文件中的内容来解释批处理文件中的一些命令和语句，为了便于讲解，对 startup.bat 文件中的内容进行了适当裁减，并在每行前加上行号，如例程 2-1 所示。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

例程 2-1 startup.bat

```
01 @echo off
02 if "%OS%" == "Windows_NT" setlocal
03 rem Guess CATALINA_HOME if not defined
04 if not "%CATALINA_HOME%" == "" goto gotHome
05 set CATALINA_HOME=.
06 if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome
07 set CATALINA_HOME=..
08 :gotHome
09 if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome
10 echo The CATALINA_HOME environment variable is not defined correctly
11 goto end
12 :okHome
13 set EXECUTABLE=%CATALINA_HOME%\bin\catalina.bat
14 set CMD_LINE_ARGS=
15 :setArgs
16 if ""%1""=="""" goto doneSetArgs
17 set CMD_LINE_ARGS=%CMD_LINE_ARGS% %1
18 shift
19 goto setArgs
20 :doneSetArgs
21 call "%EXECUTABLE%" start %CMD_LINE_ARGS%
22 :end
```

第 01 行：@echo off

echo off 命令用于让其后的所有命令在执行时都不在命令行窗口中显示命令自身的文本内容。如果在某个命令的最前面加上@，这个命令执行时将不在命令行窗口中显示其自身的文本内容。显然，第 01 行的作用就是让 startup.bat 批处理文件中的所有命令在执行时，都不在命令行窗口中显示命令自身的文本内容。如果删除第一行，那么在执行 startup.bat 文件时，命令行窗口中就会逐一显示出 startup.bat 文件中的每行内容。

第 02 行：if "%OS%" == "Windows_NT" setlocal

if 语句判断某个条件是否成立来决定是否执行命令，它一共有三种语法格式，读者可以通过执行 help if 命令来查看 if 语句的所有语法格式。第 02 行使用的格式为：

```
if "参数或环境变量" == "字符串" 命令
```

如果参数或环境变量的值等于（用两个等号来比较）指定的字符串，则条件成立，语句中的命令执行后再继续往下执行，否则就不执行语句中的命令而直接往下执行。第 02 行检查 OS 环境变量的值是否为 Windows_NT，如果是则运行 setlocal 命令。可以在命

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

令行窗口中执行 set OS 命令来查看 OS 环境变量的设置情况，执行 help setlocal 命令来查看 setlocal 命令的帮助说明。

第 03 行：rem Guess CATALINA_HOME if not defined

rem 命令用于注释一行文本，批处理文件执行时将忽略用 rem 注释的文本行。rem 命令可用于暂时取消某条命令的执行而不用删除它们，也可用于对其他语句的作用和功能进行解释或提示应该注意的问题，便于自己或别人日后阅读和维护。

第 04 行：if not "%CATALINA_HOME%" == "" goto gotHome

goto 命令用于让批处理跳转到它后面指定的标号处开始往下执行。标号在一个单独的行中定义，格式为标号名称前面加上一个冒号（:），如第 08 行所示，标号所在行不被执行。第 04 行检查是否定义过 CATALINA_HOME 环境变量，如果该环境变量的值不等于空字符串，则表示该环境变量被定义过，批处理跳转到 gotHome 标号所在行（第 08 行）开始往下执行；如果没有定义过 CATALINA_HOME 环境变量，则顺序往下执行第 05 行。

第 05 行：set CATALINA_HOME=.

set 命令用于设置环境变量。只有第 04 行检查到 CATALINA_HOME 环境变量还没有定义过，第 05 才会被执行，它将 CATALINA_HOME 环境变量的值设置为当前目录后，开始执行第 06 行。

第 06 行：if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome

第 06 行使用 if 语句的另外一种格式来判断是否存在指定的文件：

```
if exist 文件名 命令
```

如果指定的文件存在，则条件成立，语句中的命令执行后再继续往下执行，否则就不执行语句中的命令而直接往下执行。CATALINA_HOME 环境变量的值必须设置为 Tomcat 的主目录。Tomcat 通过检查该目录下是否存在一个含有 catalina.bat 文件的 bin 子目录，作为判断 CATALINA_HOME 的设置值是否正确的依据，结合图 2.4 所示的目录结构，这个道理应不难理解。

第 05 行将 CATALINA_HOME 环境变量的值设置成执行 startup.bat 批处理文件时的当前目录，如果是在图 2.4 中的 bin 目录中直接执行 startup.bat，那么 CATALINA_HOME 环境变量的值就是这个 bin 目录，显然在这个 bin 目录自身中没有一个名称也为 bin 的子目录。在这种情况下，第 06 行的条件（检查 CATALINA_HOME 环境变量表示的目录下是否存在一个含有 catalina.bat 文件的 bin 子目录）将不成立，批处理将顺序往下执行第 07 行。第 07 行将 CATALINA_HOME 环境变量的值设置成执行 startup.bat 批处理文件的当前目录的父目录，第 09 行再次对 CATALINA_HOME 环境变量的设置值进行验证，如果条件成立，就跳转到 okHome 标号所在行（第 12 行）开始往下执行，否则，批处理将顺序往下执行第 10 行。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

如果是在图 2.4 中的 bin 目录中直接执行 startup.bat，第 07 行设置的 CATALINA_HOME 环境变量就等于 bin 目录的父目录，也就是 Tomcat 的主目录。显然 bin 目录的父目录中有一个名为 bin 的子目录，这就好比张三的父亲有一个名称为张三的孩子，所以，第 09 行的条件将会成立，程序会跳转到第 12 行开始往下执行。如果在第 04 行之前已经设置过 CATALINA_HOME 环境变量，那么第 04 行将直接跳转到 09 行来验证该 CATALINA_HOME 环境变量。

如果在命令行窗口中进入 Tomcat 的主目录，然后执行 bin\startup 命令。此时，当前目录为 Tomcat 的主目录，那么第 05 行的设置就可以满足 CATALINA_HOME 环境变量的设置要求了，第 06 行的条件将成立。批处理将从第 06 行直接跳转到 okHome 标号所在行（第 12 行）开始往下执行。

第 10 行：echo The CATALINA_HOME environment variable is not defined correctly

echo 命令用于在命令行窗口中显示一行文本，便于人们了解和跟踪批处理文件的执行情况。第 11 行的作用是跳转到批处理文件的结尾处，即结束批处理的执行，第 10 行显示程序被提前结束的原因。

第 13 行：set EXECUTABLE=%CATALINA_HOME%\bin\catalina.bat

第 14 行：set CMD_LINE_ARGS=

第 13 行设置一个名为 EXECUTABLE 的环境变量，并让其值为 catalina.bat 的完整路径名。第 14 行将 CMD_LINE_ARGS 环境变量的值清空，相当于取消这个环境变量的设置。

第 16 行：if ""%1""=="""" goto doneSetArgs

在执行批处理文件时，可以在批处理文件名后面追加若干用空格分隔的文本字符串。这些文本串就是传递给批处理文件的参数，每个用空格分隔的文本字符串分别代表一个参数。例如，在命令行窗口中执行 catalina.bat start，start 就是一个传递给 catalina.bat 批处理文件的参数。批处理文件中可以使用%0 到%9 这样的 10 个变量名，其中，%0 的值为正在执行的批处理文件名，%1 到%9 依次为传递给这个批处理文件的前 9 个参数。在批处理文件中只能使用%1 到%9 这 9 个变量名来获取前 9 个参数，如果传递给批处理文件的参数超过 9 个，而在批处理文件中要获取所有的参数，那该怎么办？第 16、17、18、19 行的作用就是获得所有参数，并将所有的参数部分赋值给 CMD_LINE_ARGS 环境变量。

第 16 行检查执行 startup.bat 批处理文件时是否给它传递了参数，如果第一个参数为空字符串，就表示没有给 statup.bat 传递参数，批处理跳转到 doneSetArgs 标号所在行（第 20 行）开始往下执行。如果第一个参数不为空，那么就顺序往下执行第 17、18、19 行。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

第 17 行：set CMD_LINE_ARGS=%CMD_LINE_ARGS% %1

第 18 行：shift

第 19 行：goto setArgs

shift 命令用于将传递给批处理文件的参数依次向前移位，使用 shift 命令移位后，%1 的新值等于原来%2 的值，%2 的新值等于原来%3 的值，依次类推，直到%9 等于第 10 个参数。可见，多次使用 shift 命令，就可以在批处理文件中使用%1 到%9 这 9 个变量，甚至只使用%1 这一个变量来获取传递给批处理文件的 9 个以上的参数。

第 17～19 行与前面的第 15、16 行共同形成一个循环执行流程。第 17 行将%1 所对应的参数追加到 CMD_LINE_ARGS 环境变量的设置值后面，由于在第 14 行对 CMD_LINE_ARGS 环境变量的值进行了初始化清空，所以在第一次执行第 17 行命令时，CMD_LINE_ARGS 环境变量的值为空，这行命令执行完以后，CMD_LINE_ARGS 环境变量的值为一个空格后面加上第一个参数。

第 18 行对所有参数进行移位后，第 19 行跳转到第 setArgs 标号所在的第 15 行开始往下执行重复执行第 16 行，再次判断%1 所对应的参数是否为空字符串，如果是，则表示没有更多的参数要处理了，也就是已经将所有的参数都追加到了 CMD_LINE_ARGS 环境变量中，批处理跳转到 doneSetArgs 标号所在行（第 20 行）开始往下执行。如果%1 所对应的参数还不为空字符串，那么又开始重复上面的一次循环执行过程，如此往复，直到将所有的参数都追加到了 CMD_LINE_ARGS 环境变量中。

第 21 行：call "%EXECUTABLE%" start %CMD_LINE_ARGS%

call 命令用于调用另一个批处理文件，另外那个批处理文件执行完后回到当前批处理文件中接着执行 call 后面的命令行。注意，在一个批处理文件中也可以不使用 call 命令而直接调用另外一个批处理文件，这种方式是跳转到另外那个批处理文件中去执行，另外那个批处理文件执行完后将不会回到当前批处理文件中去执行后面的命令。

第 21 行调用 EXECUTABLE 环境变量中设置的 catalina.bat 批处理文件，并将 start 作为第一个参数传递 catalina.bat，将原来传递给 startup.bat 的参数（已存储在 CMD_LINE_ARGS 环境变量中）放在 start 参数后面继续传递给 catalina.bat。真正启动 Tomcat 的命令是在 catalina.bat 文件中编写的：如果 catalina.bat 接收的第一个参数为 start，则启动 Tomcat 服务器；如果 catalina.bat 接收的第一个参数为 stop，则停止 Tomcat 服务器。

2.3.5 catalina.bat 批处理文件

按照打开 startup.bat 文件的方式打开 catalina.bat 文件，并查看里面的内容。在 catalina.bat 文件的开始部分，使用多条 rem 命令对这个文件中使用的各个环境变量的作用进行注释说明，其中只有 CATALINA_HOME 和 JAVA_HOME 这两个环境变量不是可选

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

的，而是必须正确设置的。尽管 CATALINA_HOME 环境变量已经在 startup.bat 文件中进行了检查和设置，但 catalina.bat 文件中还是将 startup.bat 文件中检查和设置 CATALINA_HOME 环境变量的过程又重复了一遍，这主要是为保证 catalina.bat 作为一个独立模块的完整性。下面对 catalina.bat 文件进行裁减，只挑选出其中一些重要的和有必要了解的行来进行讲解，并在这些行之前加上行号，如例程 2-2 所示。

例程 2-2 catalina.bat

```
01 if exist "%CATALINA_HOME%\bin\setenv.bat" call "%CATALINA_HOME%\bin\
   setenv.bat"
02 if exist "%CATALINA_HOME%\bin\setclasspath.bat" goto okSetclasspath
03 echo Cannot find %CATALINA_HOME%\bin\setclasspath.bat
04 goto end
05 :okSetclasspath
06 set BASEDIR=%CATALINA_HOME%
07 call "%CATALINA_HOME%\bin\setclasspath.bat"
08 set CLASSPATH=%CLASSPATH%;%CATALINA_HOME%\bin\bootstrap.jar
09 if not "%CATALINA_BASE%" == "" goto gotBase
10 set CATALINA_BASE=%CATALINA_HOME%
11 :gotBase
12 if not "%CATALINA_TMPDIR%" == "" goto gotTmpdir
13 set CATALINA_TMPDIR=%CATALINA_BASE%\temp
14 :gotTmpdir
15 echo Using CATALINA_BASE:   %CATALINA_BASE%
16 echo Using CATALINA_HOME:   %CATALINA_HOME%
17 echo Using CATALINA_TMPDIR: %CATALINA_TMPDIR%
18 echo Using JAVA_HOME:       %JAVA_HOME%
19 set _EXECJAVA=%_RUNJAVA%
20 set MAINCLASS=org.apache.catalina.startup.Bootstrap
21 set ACTION=start
22 if ""%1"" == ""start"" goto doStart
23 if ""%1"" == ""stop"" goto doStop
24 :doStart
25 if not "%OS%" == "Windows_NT" goto noTitle
26 set _EXECJAVA=start "Tomcat" %_RUNJAVA%
27 goto gotTitle
28 :noTitle
29 set _EXECJAVA=start %_RUNJAVA%
30 :gotTitle
31 goto execCmd
32 :doStop
33 set ACTION=stop
34 :execCmd
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
35 %_EXECJAVA% %JAVA_OPTS% %CATALINA_OPTS% %DEBUG_OPTS%
   -Djava.endorsed.dirs="%JAVA_ENDORSED_DIRS%" -classpath "%CLASSPATH%"
   -Dcatalina.base="%CATALINA_BASE%" -Dcatalina.home="%CATALINA_HOME%"
   -Djava.io.tmpdir="%CATALINA_TMPDIR%" %MAINCLASS%
   %CMD_LINE_ARGS% %ACTION%
36 :end
```

第 01 行：if exist "%CATALINA_HOME%\bin\setenv.bat" call "%CATALINA_HOME%\bin\setenv.bat"

第 01 行检查 bin 目录中是否存在 setenv.bat 批处理文件，如果存在则调用该批处理文件。

第 02 行：if exist "%CATALINA_HOME%\bin\setclasspath.bat" goto okSetclasspath

第 02 行检查 bin 目录中是否存在 setclasspath.bat 批处理文件，如果不存在，则执行下面的第 03、04 行。第 04 行的作用是跳转到批处理文件的结尾处，即结束批处理的执行，第 03 行则是显示程序被提前结束的原因。如果 bin 目录中存在 setclasspath.bat 批处理文件，批处理跳转到 okSetclasspath 标号所在行（第 05 行）开始往下执行，第 06 行在第 07 行执行调用 setclasspath.bat 批处理文件的命令之前，先设置好 BASEDIR 环境变量，以便 setclasspath.bat 文件使用。例程 2-3 是从 setclasspath.bat 文件中挑选出来的一些主要内容。

例程 2-3 book.xml

```
if not "%JAVA_HOME%" == "" goto gotJavaHome
echo The JAVA_HOME environment variable is not defined
echo This environment variable is needed to run this program
goto end
:gotJavaHome
if not exist "%JAVA_HOME%\bin\java.exe" goto noJavaHome
goto okJavaHome
:noJavaHome
echo The JAVA_HOME environment variable is not defined correctly
goto end
:okJavaHome
if not "%BASEDIR%" == "" goto gotBasedir
echo The BASEDIR environment variable is not defined
goto end
:gotBasedir
if exist "%BASEDIR%\bin\setclasspath.bat" goto okBasedir
echo The BASEDIR environment variable is not defined correctly
goto end
:okBasedir
set JAVA_ENDORSED_DIRS=%BASEDIR%\common\endorsed
set CLASSPATH=%JAVA_HOME%\lib\tools.jar
set _RUNJAVA="%JAVA_HOME%\bin\java"
:end
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

由于 setclasspath.bat 批处理文件中需要使用 JAVA_HOME 环境变量，所以必须在 setclasspath.bat 批处理文件第一次使用 JAVA_HOME 环境变量之前正确地设置好 JAVA_HOME 环境变量，否则，在命令行窗口中执行 startup.bat 文件时，将出现如图 2.25 所示的异常效果。

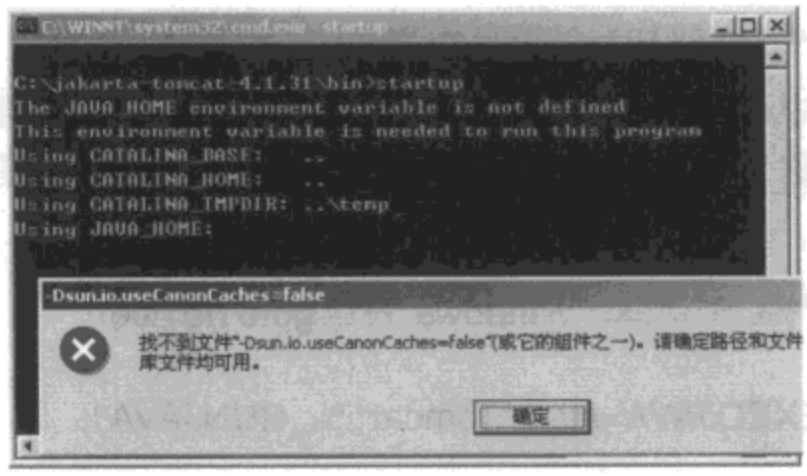


图 2.25

注意命令行窗口中的提示信息，可以发现这个错误是由于没有设置 JAVA_HOME 环境变量而引起的。弹出对话框中显示的错误信息，也是由于没有设置 JAVA_HOME 环境变量而产生的连锁错误，因此不用考虑。如果直接在 Windows 资源管理器窗口中双击 startup.bat 文件，命令行窗口将会很快消失，根本无法查看到错误的提示信息，所以在查找 Tomcat 的启动问题时不要使用双击 startup.bat 文件的方式来启动 Tomcat。

第 08 行：set CLASSPATH=%CLASSPATH%;%CATALINA_HOME%\bin\bootstrap.jar

第 08 行在 CLASSPATH 环境变量原来的设置值中增加 bootstrap.Jar 包，在执行这行命令之前，setclasspath.bat 批处理文件还将%JAVA_HOME%\lib\tools.jar 加入到 CLASSPATH 环境变量之中，这是必不可少的。因为 Tomcat 需要将由 JSP 页面生成的 Servlet 源文件编译成 Java 字节码后运行，而那些用于编译 Java 源文件的 API 类位于 tools.jar 中。

第 09～14 行：设置 CATALINA_BASE 和 CATALINA_TMPDIR 环境变量。

第 15～18 行：在命令行窗口中显示几个重要环境变量的设置值。

第 19 行：set _EXECJAVA=%_RUNJAVA%

第 20 行：set MAINCLASS=org.apache.catalina.startup.Bootstrap

第 21 行：set ACTION=start

第 19～21 行对第 35 行中要用到的几个主要环境变量进行了设置，第 35 行的最简形式为：

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
%_EXECJAVA% %MAINCLASS% %ACTION%
```

第 19 行将_RUNJAVA 环境变量的值再赋值给_EXECJAVA 环境变量，_RUNJAVA 环境变量在 setclasspath.bat 文件中被设置成%JAVA_HOME%\bin\java，所以，将这些环境变量用相应的值替换后的第 35 行的最简形式为：

```
java org.apache.catalina.startup.Bootstrap start
```

从上面这行内容中可以看出，Tomcat 是一个需要由 Java 解释器解释执行的 Java 程序，启动运行类的完整名称为 org.apache.catalina.startup.Bootstrap，start 是传递给这个启动运行类的参数。

第 25 行：if not "%OS%" == "Windows_NT" goto noTitle

第 26 行：set _EXECJAVA=start "Tomcat" %_RUNJAVA%

第 27 行：goto gotTitle

第 28 行：:noTitle

第 29 行：set _EXECJAVA=start %_RUNJAVA%

start 命令将启动一个新的命令行窗口，并在新窗口中执行某个命令，该命令结束后新窗口自动关闭。如果当前操作系统的 OS 环境属性为 Windows_NT，那么在 start 命令中还可以设置新启动的命令行窗口的标题。

第 25～29 行重新设置_EXECJAVA，在 java 命令前加上 start 命令。如果当前操作系统的 OS 环境属性为 Windows_NT，那么执行第 26 行将新启动的命令行窗口标题设置为 Tomcat，否则执行第 29 行。

第 23、32、33 行

在 catalina.bat 文件中加入用于停止 Tomcat 的功能，因为 catalina.bat 文件不仅要被启动 Tomcat 的 startup.bat 文件调用，还要被关闭 Tomcat 的 shutdown.bat 文件调用。

2.3.6 使用批处理程序及查找 Tomcat 的启动问题

通过上面对 startup.bat 和 catalina.bat 文件的分析，得到的结论如下：

只要在 setclasspath.bat 批处理文件第一次使用 JAVA_HOME 环境变量之前的任何地方，例如在 Windows 系统特性的“环境变量”对话框中，或者在 startup.bat、catalina.bat、setclasspath.bat 等任意一个文件的开始位置处，将 JAVA_HOME 环境变量设置为 JDK 的安装主目录，就可以使用 startup.bat 文件启动 Tomcat 了。

只要 Tomcat 不是以 Windows 2000 Server 服务程序的方式启动运行，就可以使用 shutdown.bat 文件或“Apache Tomcat 4.1”程序组里面的“Stop Tomcat”快捷菜单项、或

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

者直接在命令行窗口中按下 Ctrl+C 键结束 Tomcat 的运行。对于 Tomcat 5.x 的批处理文件，除了需要将 JAVA_HOME 环境变量设置为 JDK 5.0 的主目录外，其他细节与 Tomcat 4.1.31 的批处理文件相似。

在使用 jakarta-tomcat-4.1.31.exe 安装 Tomcat 时，Tomcat 服务程序或安装的快捷菜单将记住所使用的 JDK 主目录的绝对路径，它们根本不使用这些批处理文件和 JAVA_HOME 环境属性，所以即使没有设置 JAVA_HOME 环境变量，也可以使用它们启动 Tomcat。

Linux 版本和 Windows 版本的 Tomcat 的配置和运行方式没有多大区别，只要将 Linux 操作系统下的 Tomcat 安装包解压到 Linux 下的一个目录中，然后按照类似 Windows 操作系统下的批处理文件方式进行配置即可。Linux 下的 Tomcat 启动脚本文件为 startup.sh，关闭脚本文件为 shutdown.sh。本书虽然只使用 Windows 操作系统下的 Tomcat 进行讲解，但对使用 Linux 操作系统的读者来说，只要注意 Linux 区分大小写即可，例如，JAVA_HOME 和 CLASSPATH 等环境变量名称中的每个字母都要大写，其他部分几乎不受影响。

如果使用 Windows 2000 Server 的 Tomcat 服务程序、安装的快捷菜单或者直接双击 startup.bat 等方式启动 Tomcat 时失败，这些方式要么不报告错误信息，要么报告错误信息的窗口立即被自动关闭，人们都无法查看到错误的提示信息。如果先启动一个命令行窗口，然后在这个命令行窗口中执行 startup.bat 命令，在 Tomcat 启动失败而导致 startup.bat 命令结束时，该命令行窗口不会被自动关闭。但是，startup.bat 命令在执行过程中启动一个新的命令行窗口，Tomcat 打印的提示信息显示在这个新启动的窗口中，而这个新启动的窗口在 Tomcat 启动失败时也将立即被自动关闭，仍然无法看到错误的提示信息。

如果能想办法让 startup 批处理的内部不要产生新的命令行窗口，那么就可以让 Tomcat 打印的提示信息显示在执行 startup 命令的命令行窗口中，根据这些提示的错误信息就很容易知道错误的具体原因了。只要将 2.3.5 节分析的 catalina.bat 文件中的第 26 行里的位于%_RUNJAVA%前面的 start "Tomcat" 部分删除，或者是将第 29 行里的位于%_RUNJAVA%前面的 start 部分删除，就可以满足这个要求。至于删除第 26 行，还是删除第 29 行，需要根据 OS 环境属性来确定，更简单的办法是对两行中的内容都进行删除。按照这种分析对 catalina.bat 文件进行修改后，在命令行窗口中执行 startup.bat 命令就可以查看到 Tomcat 启动时的错误信息了。

练习查找 Tomcat 的启动错误

01 先采用任何一种方式正常启动 Tomcat 的一个运行实例（这将导致下一次启动 Tomcat 时发生错误），然后双击 startup.bat 文件再次启动 Tomcat，看到启动的窗口立即消失了，根本来不及查看错误的原因。

02 启动一个命令行窗口，用 cd 命令进入 Tomcat 主目录下的 bin 子目录。执行 startup 命令，可以看见这个命令启动一个新的命令行窗口后，这个新的命令行窗口很快就消失了，

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

无法查看到错误的提示信息。

03 将 catalina.bat 文件中的位于%_RUNJAVA%前面的 start "Tomcat"部分删除后，重新执行 startup 命令，在命令行窗口中看到了一些异常信息，但是由于显示的信息内容过多，造成前面的信息溢出了命令行窗口的显示缓冲区而无法看到。只有 Tomcat 打印出的提示信息的前面部分才是描述根本问题的内容，而后面部分通常都是由根本问题所引发的其他连锁问题的描述。要想看到根本问题的描述，需要增大命令行窗口缓冲区，让 Tomcat 打印出的提示信息的前面部分不要溢出命令行窗口的显示缓冲区。

04 在命令行窗口顶部的标题栏上单击鼠标右键，从弹出的快捷菜单中选择“属性”，将打开的“属性”窗口切换到“布局”选项卡，如图 2.26 所示。

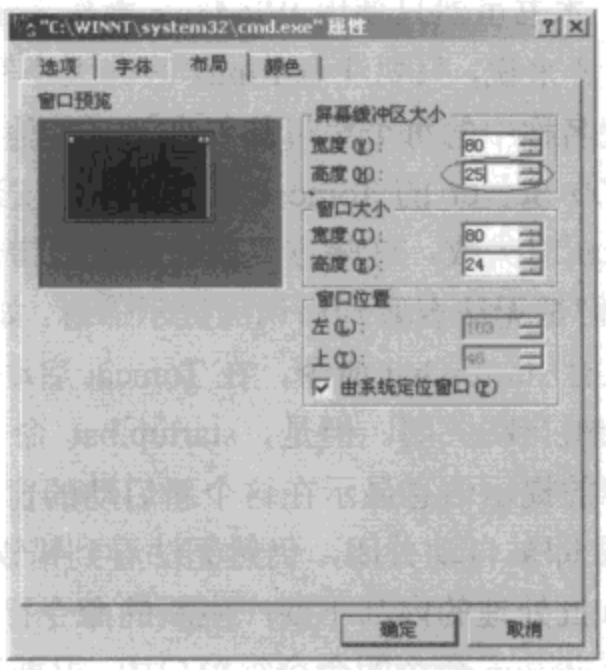


图 2.26

将图 2.26 中的屏幕缓冲区大小的高度设置为 250 或更大后，再次执行 startup 命令，这时候通过上下拖动命令行窗口的垂直滚动条中的滚动滑块就可以看到错误发生的根本原因了，如图 2.27 所示。

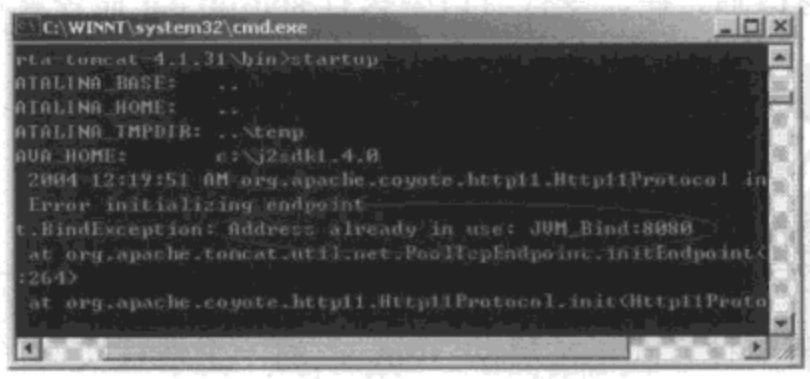


图 2.27

从错误的提示信息中可以看到，引起错误的原因是这个 Tomcat 实例所要绑定的 8080 号端口已经被其他网络程序占用了。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

05 Tomcat 的另外一个常见的启动问题就是因配置文件有错而引起的，例如，用记事本程序或 UltraEdit 工具软件打开<Tomcat 主目录>/conf/server.xml 文件，删除最后的第二行文本，即内容为</Service>的那行。关闭 Tomcat 的所有运行实例后重新执行 startup 命令，Tomcat 启动失败，上下拖动命令行窗口的垂直滚动条中的滚动滑块即可以看到错误的具体信息（包括配置文件中发生错误的行号），如图 2.28 所示。

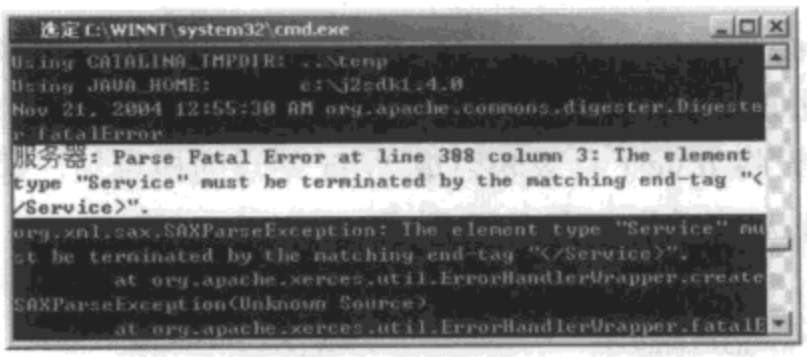


图 2.28

按照上面的实验所采用的方法，可以很好地解决 Tomcat 的绝大多数启动问题。另外，如果问题是由于 startup.bat 或 catalina.bat 文件自身编写有误造成的，可以在这两个文件中增加一些 pause 和 echo 命令。pause 命令可以暂停批处理文件的执行流程，并在命令行窗口中显示“Press any key to continue...”信息，等待用户按任意键后继续按原来的流程往下执行。用 echo 命令显示环境变量的值，然后用 pause 命令暂停批处理文件的执行流程，以便按步跟踪和查看批处理的执行线索。

2.4 配置 Web 站点的虚拟目录

一个 Web 资源的 URL 中的资源路径部分，例如 http://www.it315.org/training/index.html 中的/training/index.html，就是由 Web 服务器（www.it315.org）的某个本地目录中的资源所映射成的虚拟路径。“/training/index.html”中的第一个正斜杠“/”表示站点的根目录，它对应到 Web 服务器的本地文件系统中的一个目录。“/training/”表示站点根目录下面的一个名为“training”的子目录，“/training/index.html”则表示“training”子目录中的一个名为“index.html”的文件。站点的“/training/”子目录也要对应到 Web 服务器的本地文件系统中的一个目录上，这个目录可以直接是站点根目录所对应的本地文件系统目录中的一个名为“training”的子目录，也可以是与站点根目录所对应的本地文件系统目录没有任何隶属关系的另外一个目录。

要想让某台计算机上的一个目录内容被外面的 Web 浏览器通过 HTTP 协议访问，除了在这台计算机上必须安装和启动一个 Web 服务器软件外，还要将该目录映射成一个虚拟的 Web 站点目录。将 Web 服务器的本地文件系统中的某个目录映射成一个虚拟 Web 目录的过程叫 Web 发布（更通俗一点称呼为网页发布）。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

注意　为了达到最佳学习效果和满足后面章节的实验需要，2.4 节和 2.5 节的实验必须在 Tomcat 4.1.31 环境中进行，请不要使用 Tomcat 5.x 进行实验。

2.4.1　虚拟 Web 目录的映射原理

一个 Web 站点上必须有且只能有一个虚拟根目录，其他虚拟目录都以根目录的子孙目录形式出现，根目录用正斜杠“/”表示，根目录与该目录下的子目录或资源文件也用正斜杠“/”进行分隔。假设将 d:\test 目录映射成 Web 站点的根目录，那么浏览器访问的资源路径与本地文件系统的对应关系如图 2.29 所示。

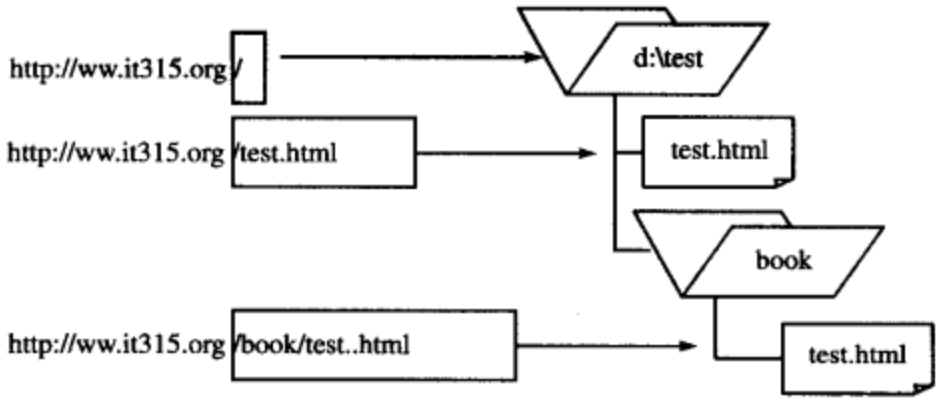


图 2.29

一个 Web 站点除了可以设置一个虚拟根目录外，还可以为其设置若干个虚拟子目录，如图 2.30 所示。

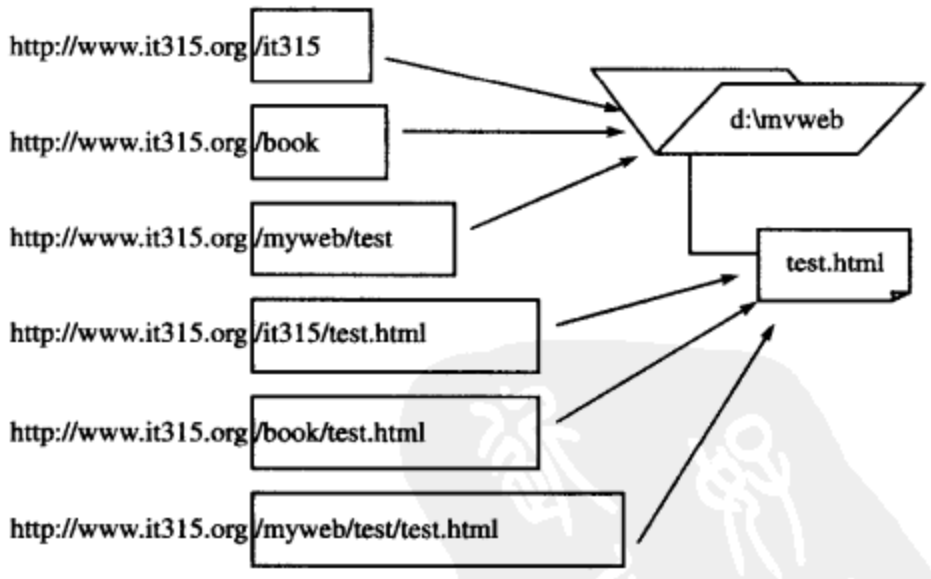


图 2.30

从图 2.30 中可以看到，一个文件系统目录可以被映射成一个或多个虚拟 Web 目录，但一个虚拟 Web 目录只能对应一个本地文件系统目录。虚拟 Web 子目录名称可以是多级目录结构的形式，例如，本地文件系统上的 d:\myweb 目录可以映射成名为/myweb/test 的虚拟 Web 子目录。当浏览器请求某个路径的资源时，Tomcat 按照最长路径匹配

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

原则进行处理。它首先以请求路径中的最深子目录作为一个虚拟子目录名称，查看是否存在这个虚拟子目录：如果不存在这个虚拟子目录，则查找是否存在上一级目录名称的虚拟子目录；如果找到匹配的虚拟目录后，则进入该虚拟目录中去查找指定名称的资源。

显然，当虚拟子目录的名称与根目录下的某个子目录名称相同时，Web 浏览器无法再访问到根目录下的该名称的子目录，只能访问同名的虚拟子目录。例如，将 d:\myweb 映射成的一个名称为/book 的虚拟 Web 子目录，而已映射成 Web 站点根目录的 d:\test 目录中也有一个名称为 book 的子目录，在本地计算机上的 IE 浏览器地址栏中输入 http://127.0.0.1:8080/book/test.html，浏览器访问到的将是 d:\myweb 目录中的 test.html 文件，而不是 d:\test\book 目录中的 test.html 文件。

2.4.2 设置 Web 站点的根目录

要在 Tomcat 中为 Web 站点设置根目录，可以使用下面的两种方式之一：

（1）<Tomcat 主目录>/webapps/ROOT 目录可以自动映射成为 Web 站点的虚拟根目录，Tomcat 初始安装后的虚拟根目录就是以这种方式创建的，注意目录名 ROOT 中的每个字母都必须是大写的。在建立自己的 Web 站点时，可以先删除 Tomcat 提供的 ROOT 目录中的原始内容，然后将自己想提供给外部访问的资源复制到 ROOT 目录中。

（2）用 UltraEdit 打开<Tomcat 主目录>/conf/server.xml 文件，使用“查找”菜单查找 root 关键字，可以看到如下的内容：

```
<!--
<Context path="" docBase="ROOT" debug="0"/>
-->
```

server.xml 文件中的一个<Context>元素用于将本地文件系统中的一个目录映射成一个可供 Web 浏览器访问的虚拟 Web 目录。其中，path 属性指定虚拟 Web 目录的路径，当其设置为空字符串时，则表示的是整个 Web 站点的根目录；docBase 属性指定该虚拟 Web 目录所映射到的本地文件系统目录，可以使用绝对路径或相对<Tomcat 主目录>/webapps 的相对目录。上面看到的 Web 根目录配置信息被注释掉了，如果要通过这种方式来配置 Web 站点的根目录，应删除起始和结束注释符，例如，要将 d:\test 目录配置成 Web 站点的根目录，应将上面看到的这部分内容修改成如下形式：

```
<Context path="" docBase="d:\test" debug="0"/>
```

其实，由于一个 Web 站点的根目录只有一个，直接以 ROOT 目录作为 Web 站点的根目录就可以了，一般不必使用 server.xml 文件中的<Context>元素来设置 Web 站点的根目录。将 Web 服务器上的某个文件系统目录映射成 Web 站点的根目录后，就可以使用浏览器来访问这个目录及其子目录下的资源了。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

修改 Web 站点的默认根目录

01 在 ROOT 目录下创建一个 test.html 文件，在 test.html 文件中写上如下一行内容：

```
这是ROOT目录中的test.html文件
```

在 Web 服务器的本地计算机上的 IE 浏览器地址栏中输入 http://127.0.0.1:8080/test.html，可以查看到浏览器中显示的网页文档内容为刚才新建的 test.html 文件中的内容。从这个实验中，可以看到 Web 服务器上的本地文件系统目录与 Web 站点根目录的默认映射关系。

02 在 d:\盘中创建一个 test 目录，在该 test 目录下创建一个 test.html 文件，在这个 test.html 文件中写上如下一行内容：

```
这是d:\test目录中的test.html文件
```

接着修改<Tomcat 主目录>/conf/server.xml 文件，取消其中用于设置 Web 站点根目录的语句的注释，将 d:\test 设置为 Web 站点的根目录：

```
<Context path="" docBase="d:\test" debug="0"/>
```

注意，要想让修改后的 server.xml 文件发生作用，必须重新启动 Tomcat。重新启动 Tomcat 后，刷新步骤 **01** 中的 IE 浏览器窗口，可以查看浏览器中重新显示的网页文档内容为 d:\test\test.html 文件中的内容。这个实验说明，Web 站点根目录已经被映射到了 d:\test 目录。

03 在 d:\test 目录中新建一个 book 子目录，在 book 子目录下新建一个 test.html 文件，在这个 test.html 文件中写上如下一行内容：

```
这是d:\test\book目录中的test.html文件
```

在 IE 浏览器地址栏中输入 http://127.0.0.1:8080/book/test.html，浏览器又显示 book 目录中的 test.html 网页文件的内容。这个实验说明，将某个本地文件系统目录映射成 Web 站点目录后，它里面的子目录也可以作为该 Web 站点目录的子目录进行访问。

04 在其他计算机上打开 IE 浏览器，重复第 **02** 和第 **03** 步的访问，但应将访问 URL 中的 IP 地址从 127.0.0.1 改为 Web 服务器的真实 IP 地址。

2.4.3 设置 Web 站点的虚拟子目录

在 Tomcat 中设置虚拟子目录的方式有多种，具体细节可能会随着 Tomcat 版本的升级而变化。一般来说，可以通过如下方式查看这些方式的说明：访问 Tomcat 的首页，通过其左侧导航栏中的“Tomcat Documentation”超链接进入 Tomcat 的帮助文档页面，在帮助文档页面的“Reference”导航栏中可以看到“Apache Tomcat Configuration”超链接，进入

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

该超链接指向的页面后，再通过该页面左侧的“Context”超链接进入 Context 元素的帮助页面，就可以看到设置虚拟子目录的各种方式的介绍。

1. 基本方式

在 Tomcat 中设置虚拟子目录的最基本的方式就是在<Tomcat 主目录>/conf/server.xml 文件中设置<Context>元素，在 2.4.2 节中已对<Context>元素进行了基本的介绍。在 server.xml 文件中，<Context>元素必须嵌套在<Host>元素之中，一个<Host>元素表示一个 Web 站点，其中可以包含多个<Context>子元素，每个<Context>子元素分别对应该站点下的一个虚拟 Web 目录。<Context>元素中指定的虚拟 Web 子目录名称与本地文件系统的目录名称没有必然的关联，两者的名称可以不一致，例如，只要在<Tomcat 主目录>/conf/server.xml 文件的<Host>元素中增加如下一条语句：

```
<Context path="/it315" docBase="d:\myweb" debug="0"/>
```

即可将本地计算机文件系统上的 d:\myweb 目录映射成 Web 站点的/it315 虚拟子目录。

一个虚拟目录就是一个独立的 Web 应用程序。为了便于 Web 应用程序的发布，通常将一个 Web 应用程序所对应的整个目录打包成一个 jar 格式的压缩文件，这个文件称之为 war（Web Application Archive）文件。为了见名知意，war 文件通常以.war 作为扩展名，但它实际上就是一个 jar 文件。使用 war 文件发布 Web 应用程序相比直接使用目录的好处是非常明显的，它让 Web 应用程序显得更加紧凑，发布更加方便。server.xml 文件中的<Context>元素的 docBase 除了可以指向一个目录外，它还可以指向一个 war 文件。

2. 其他方式

除了可以在 server.xml 文件中使用<Context>元素来设置 Web 应用程序的虚拟目录外，还可以将 server.xml 文件中的<Host>元素的 autoDeploy 属性设置为 TRUE，然后让 Tomcat 4.1.31 在启动时自动按如下几种方式来创建 Web 应用程序的虚拟目录。

（1）对于 Tomcat 4.x，如果<Host>元素指定的 appBase 目录（默认为<Tomcat 主目录>/webapps 目录）中的 XML 文件包含<Context>元素设置，这些<Context>元素与直接包含在 server.xml 文件中<Context>元素具有同样的作用效果，例如，Tomcat 4.1.31 的<Tomcat 主目录>/webapps 目录下的 admin.xml 和 manager.xml 文件。对于 Tomcat 5.x，如果<Tomcat 主目录>/conf/<引擎名>/<主机名>目录中的 XML 文件中包含<Context>元素设置，这些<Context>元素与直接包含在 server.xml 文件中<Context>元素具有同样的作用效果，例如，Tomcat 5.5.12 的<Tomcat 主目录>/conf/Catalina/localhost 目录中的 manager.xml 文件。

（2）如果<Host>元素指定的 appBase 目录下的子目录中包含 Web-INF/web.xml 文件，这些子目录将被自动设置成各自独立的 Web 应用程序，其虚拟路径就是在该子目录的名称前面加上“/”。

（3）如果<Host>元素指定的 appBase 目录中包含 war 文件，这些 war 文件将被自动设置成各个独立的 Web 应用程序，其虚拟路径就是该文件名称（不含.war 后缀部分）前面加

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

上“/”。如果<Host>元素的 unpackWARs 属性设置为了 true，这些 war 文件在第一次使用时将被自动展开成同名的目录（不含.war 后缀部分）形式；如果以后更新了 war 文件，一定要记住将以前展开的同名目录删除掉，否则更新不会生效。

设置 Web 站点的虚拟子目录

01 在 d:\盘下创建一个 myweb 目录，接着在 myweb 目录下新建一个 test.html 文件，并在这个 test.html 文件中写上如下一行内容：

```
这是d:\myweb目录中的test.html文件
```

02 将本地计算机文件系统的 d:\myweb 目录映射成 Web 站点的/it315 虚拟子目录。用 UltraEdit 打开<Tomcat 主目录>/conf/server.xml 文件，查找前面设置根目录的那行内容，紧接着那行内容的下面，增加如下一行内容：

```
<Context path="/it315" docBase="d:\myweb" debug="0"/>
```

重新启动 Tomcat 后，在本地计算机上的 IE 浏览器地址栏中输入 http://127.0.0.1:8080/it315/test.html，可以查看到浏览器中显示的网页文档内容为 d:\myweb\test.html 文件中的内容。这个实验说明，文件系统的 d:\myweb 目录已经成功地映射成 Web 站点的/it315 虚拟子目录。

03 在 server.xml 文件中紧接着上面新增加的内容下面，再增加如下一行内容：

```
<Context path="/myweb/test" docBase="d:\myweb" debug="0"/>
```

重新启动 Tomcat 后，在本地计算机上的 IE 浏览器地址栏中输入 http://127.0.0.1:8080/myweb/test/test.html，在浏览器中也能显示出 d:\myweb\test.html 文件中的内容。这个实验说明，文件系统的 d:\myweb 目录被同时映射成 Web 站点的多个虚拟子目录，并且虚拟子目录名称可以是多级目录结构的形式。

04 在前面的实验中，已经将 d:\test 目录映射成 Web 站点的根目录，因此，d:\test 目录中的 book 子目录对应的 URL 地址为 http://127.0.0.1:8080/book。在 server.xml 文件中紧接着上面新增加的内容下面，再增加如下一行内容：

```
<Context path="/book" docBase="d:\myweb" debug="0"/>
```

即将 d:\myweb 映射成的一个名称正好是/book 的虚拟 Web 子目录。当使用 IE 浏览器访问 http://127.0.0.1/book/test.html 时，浏览器访问的是文件系统中的哪个目录下的 test.html 文件呢？重新启动 Tomcat 后，在本地计算机上的 IE 浏览器地址栏中输入 http://127.0.0.1:8080/book/test.html，可以看到浏览器中显示的是 d:\myweb 目录中的 test.html 文件内容，而不是 d:\test\book 目录中的 test.html 文件内容。

05 按下面的步骤将 d:\myweb 目录打包成 d:\myweb.jar 文件。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

1）在命令行窗口中，使用 cd 命令进入 d:\myweb 目录；

2）执行命令下面的命令：

```
jar -cvf mywebwar.war.
```

其中的“.”代表当前目录，上面的命令表示要将 d:\myweb 目录中的所有内容压缩到 mywebwar.war 文件中，但不包含 myweb 目录本身。

3）将 mywebwar.war 文件移动到 d:\中。

然后在 server.xml 文件中紧接着上面新增加的内容下面，再增加如下一行内容：

```
<Context path="/testwar" docBase="d:\mywebwar.war" debug="0"/>
```

重新启动 Tomcat，在本地计算机上的 IE 浏览器地址栏中输入 http://127.0.0.1:8080/testwar/test.html，浏览器中显示 d:\mywebwar.war 压缩包中的 test.html 文件的内容。

06 将 d:\ myweb 目录及其中的所有内容复制到<Tomcat 主目录>/webapps 目录中，重新启动 Tomcat 后，在本地计算机上的 IE 浏览器地址栏中输入 http://127.0.0.1:8080/myweb/test.html，浏览器提示找不到网页文档的错误信息。在<Tomcat 主目录>/webapps/myweb 目录中创建一个名为 Web-INF 的子目录后，重新启动 Tomcat，接着在本地计算机上的 IE 浏览器地址栏中还是输入 http://127.0.0.1:8080/myweb/test.html，浏览器这次显示 test.html 文件的内容。这说明只有<Tomcat 主目录>/webapps 目录下的子目录中包含 Web-INF 子目录（如果 Web-INF 目录中不存在 web.xml 文件，Tomcat 将使用默认的 web.xml 文件内容）。更确切地说，在包含 Web-INF/web.xml 文件时，该子目录才会被自动设置成一个 Web 应用程序，其虚拟路径就是在该子目录的名称前面加上“/”。

07 将 d:\mywebwar.war 文件复制到<Tomcat 主目录>/webapps 目录中，稍后即可看到 tomcat 将该 war 文件解压成了一个名为 mywebwar 的目录，接着在本地计算机上的 IE 浏览器地址栏中还是输入 http://127.0.0.1:8080/mywebwar/test.html，浏览器显示 mywebwar.war 压缩包中的 test.html 文件的内容。这说明放入<Tomcat 主目录>/webapps 目录下的 war 文件将被自动设置成一个 Web 应用程序，其虚拟路径就是该 war 文件名称（不带扩展名）的前面加上“/”。在这种情况下，war 文件对应的目录中可以不包含 Web-INF 子目录和 Web-INF/web.xml 文件。

war 文件的内部目录结构

在使用 jar 命令将 d:\myweb 目录打包成 war 文件时，如果命令行窗口中的当前工作目录为 d:\，那么不能够使用下面的命令来创建 d:\myweb.war 文件：

```
jar -cvf  myweb.war  myweb
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

因为这种方式所创建的 myweb.war 展开后的结果包含 myweb 这个目录自身，而 war 文件要求其展开后的结果直接包含 myweb 目录下面的所有内容，而不要包含 myweb 目录自身。如果命令行窗口中的当前工作目录为 d:\，则应该使用下面的命令来创建 d:\myweb.war 文件：

```
jar -cvf  myweb.war  -C  myweb
```

其中的–C myweb 选项的作用是改变所要压缩文件源的相对基准目录为 d:\myweb。只要用 WinRar 软件打开上面两种方式所创建的 myweb.war 文件，就可以看到它们在目录结构上的差别。

2.4.4 部署描述符与目录的默认网页

一个 Web 虚拟目录就是一个独立的 Web 应用程序，每个 Web 应用程序都有各自独立的配置信息。每个 Web 应用程序的配置信息都保存在各自所映射的文件系统目录下的 Web-INF/web.xml 文件中。web.xml 文件用于描述 Web 应用程序的部署信息，因此，web.xml 文件也称为应用程序的部署描述符。由于每个 Web 应用程序都有许多相同的配置特性，为了免除在每个 Web 应用程序的 web.xml 文件中都重复设置这些相同的特性，Tomcat 在<Tomcat 主目录>/conf 目录中也提供一个 web.xml 文件，该文件中的设置信息对所有的 Web 应用程序都起作用，因此，<Tomcat 主目录>/conf/web.xml 文件里面的设置信息也就是每个 Web 应用程序的默认设置。每个 Web 应用程序内部的 web.xml 文件用于在<Tomcat 主目录>/conf/web.xml 文件的设置基础上增加或覆盖一些设置信息，一个 Web 应用程序内部的 web.xml 中所设置的信息只对该 Web 应用程序自己有效。

如果客户端访问的 URL 指向的是一个目录名，而不是一个文件名，那么，Tomcat 将向客户端返回怎样的内容呢？在这种情况下，Tomcat 有可能返回目录下的内容列表，也可能返回该目录下的一个默认网页文件。其实，大家经常听说的站点首页就是站点根目录的默认网页。目录的默认网页文件是在 web.xml 文件内的<welcome-file-list>元素中进行设置的，其格式如下：

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

<welcome-file-list>元素用于设置 Web 目录的默认网页文档列表，其中可以嵌套多个<welcome-file>子元素，每个<welcome-file>子元素中指定一个网页文件。设置 Web 目录的默认网页文档后，当浏览器访问指向目录的 URL 地址时，Tomcat 按照<welcome-file-list>元素指定的默认网页文档的顺序，依次在该目录中查找这些默认网页文档，将最先找到的默认网页文件回传给 Web 浏览器，并停止查找后面的默认网页文件。如果<welcome-file-

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

list>元素指定的所有默认网页文档在浏览器访问的目录中都不存在，Tomcat 将该目录中的文件和子目录的超链接列表传送给 Web 浏览器。

设置目录的默认网页

01 在前面已经映射成 Web 站点根目录的 d:\test 目录中新建一个名称为 Web-INF 的子目录，注意必须保证 Web-INF 这个目录名称中的每个字母全部是大写的。然后在 IE 浏览器地址栏中输入 http://127.0.0.1:8080/。对于指向目录的 URL 地址，目录名最后的正斜杠（/）可以省略，所以，在浏览器地址栏中也可以输入 http://127.0.0.1:8080，浏览器中显示的结果如图 2.31 所示。

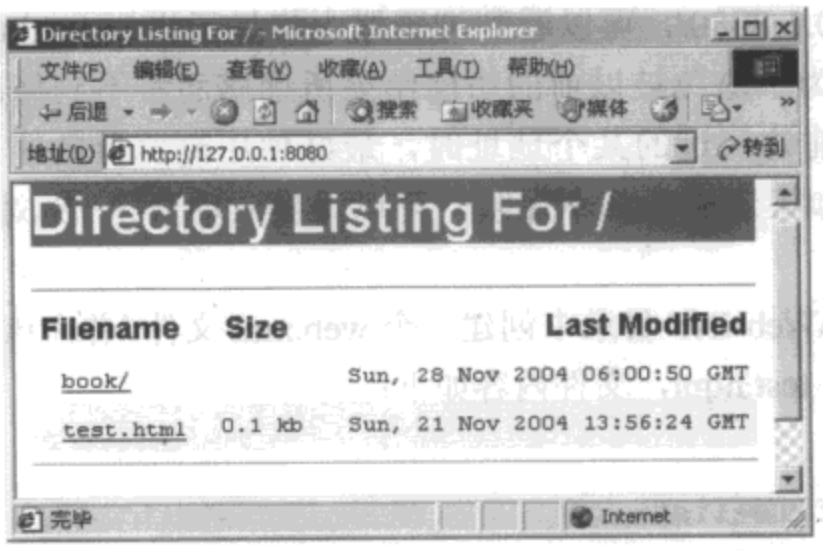


图 2.31

从图 2.31 的结果中可以看到，如果在 IE 浏览器中输入的 URL 是一个目录名，浏览器将显示该目录中的除了名称为 Web-INF 的子目录之外的所有其他子目录和文件名称的超链接列表。一个 Web 应用程序下的 Web-INF 目录是保留给系统使用的目录，浏览器无法访问到该目录，所以，该目录不会出现在显示列表中。另外，逻辑上属于某个虚拟目录的虚拟子目录也不会显示在上面的目录列表中，例如，/it315 和/myweb/test 都没有显示在图 2.31 所示的根目录列表中。要访问虚拟子目录，用户必须在浏览器地址栏中输入含有虚拟子目录名称的 URL 地址，而不可能使用如图 2.31 中所列出的超链接方式进入。

02 用 UltraEdit 打开<Tomcat 主目录>/conf/web.xml 文件，拖动滚动条中的滚动滑块，在 web.xml 文件的最后可以看到如下的内容：

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


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

这说明当浏览器访问指向目录的 URL 地址时，Tomcat 将依次在该目录中查找 index.html、index.htm 和 index.jsp 文件。为了验证这种效果，在 Web 站点根目录所映射的 d:\test 目录中创建两个名称分别为 index.html 和 index.htm 的文件，在 index.html 文件中写入如下一行内容：

这是d:\test目录中的index.html文件

在 index.htm 文件中写入如下一行内容：

这是d:\test目录中的index.htm文件

刷新步骤 **01** 中打开的 IE 浏览器窗口，或为了防止浏览器缓存影响查看结果，在一个新启动的 IE 浏览器地址栏中输入 http://127.0.0.1:8080/，可以看到浏览器中显示 index.html 文件的内容。删除 index.html 文件后，在一个新启动的 IE 浏览器地址栏中再次输入 http://127.0.0.1:8080/，可以看到浏览器中将显示出 index.htm 文件的内容。由于 IE 浏览器有缓存，为了避免被以前的访问结果所迷惑而看不到修改后的效果，使用 IE 浏览器重新访问以前访问过的某个地址时，最好使用浏览器的刷新功能，而不是在地址栏上按回车键。如果连刷新都不保险，可以重新启动一个 IE 浏览器程序来重新访问这个地址。

03 在 d:\test\Web-INF 目录中创建一个 web.xml 文件，并在该文件设置 Web 应用程序的默认网页文档为 test.html，文件内容如下：

```
<web-app>
    <welcome-file-list>
        <welcome-file>test.html</welcome-file>
    </welcome-file-list>
</web-app>
```

其中，<web-app>为整个 web.xml 文件的根元素。在编写 web.xml 文件时，可以参照<Tomcat 主目录>/conf/web.xml 文件的相应部分，为减少一些手误，最好是使用“复制”和“粘贴”的方式。注意，要想让修改后的 web.xml 文件发生作用，必须重新启动 Tomcat。重新启动 Tomcat 后，在一个新启动的 IE 浏览器地址栏中输入 http://127.0.0.1:8080/，可以看到浏览器中将显示 test.html 文件的内容。

04 将 d:\test 目录下的 test.html 文件改名为 testx.html，然后再次访问 http://127.0.0.1:8080/。这时，浏览器中没有显示 d:\test 目录下的 index.htm 文件的内容，而又显示该目录中的除了名称为 Web-INF 的子目录之外的所有其他子目录和文件名称的超链接列表。这说明在 Web 应用程序内部的 web.xml 文件中设置的<welcome-file-list>元素将覆盖<Tomcat 主目录>/conf/web.xml 文件中设置的<welcome-file-list>元素信息，而不是对此进行增加。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

2.5 配置虚拟主机

2.5.1 浏览器访问 Web 资源的过程

Web 服务器上的资源的通用 URL 格式为："http://IP 地址（或主机名）：端口号/目录名称/网页文件名称"，如果 Web 服务器使用的是 HTTP 协议的默认端口号（80），则可以省略其中的端口号。例如，对于 http://www.it315.org/book/java.html 这个 URL，所用的协议为 HTTP，提供 Web 服务的主机名是 www.it315.org，因为主机名后面没有指定端口号，则表示 Web 服务程序监听的端口号为 HTTP 协议的默认端口号 80，资源名为 /book/java.html。

浏览器访问一个 Web 资源时，根据 URL 中的协议名、IP 地址（或主机名）、端口号等信息与 Web 服务器建立网络连接。由于主机名比 IP 地址更容易记忆，人们通常在 URL 中使用的是主机名，而不是 IP 地址。如果 URL 中指定的是主机名，Web 浏览器会先查询该主机名对应的 IP 地址，然后再使用这个 IP 地址去与 Web 服务器建立连接。查询主机名所对应的 IP 地址，一般都是通过 DNS（Domain Name System，域名系统）服务器完成的。如果 Web 浏览器不能使用主机名访问 Internet 上的网站，但能够通过主机的 IP 地址访问，这就是浏览器客户机上没有正确设置 DNS 服务器或 DNS 服务器停止服务所致。有关 DNS 的知识，感兴趣的读者可以阅读专门讲解网络的书籍。Web 浏览器与 Web 服务器建立连接后，通过这个连接将 URL 后半部分的资源路径发送给 Web 服务器，请求服务器返回这个路径名所指定的资源。

用 Telnet 程序分析浏览器访问 Web 资源的过程

Windows 提供的 Telnet 程序是一个简单和通用的 TCP 客户端程序，它可以按指定的 IP 地址和端口号去与某个服务器程序建立 TCP 连接。连接建立后，Telnet 程序窗口中会显示服务器程序发送过来的数据，在 Telnet 程序窗口中用键盘输入的内容也会发送到服务器程序。

01 启动 Tomcat Web 服务器程序，然后打开一个新的命令行窗口，并在这个命令行窗口中执行如下命令：

```
telnet 127.0.0.1 8080
```

如果 Tomcat Web 服务器没有启动，Telnet 程序将显示连接失败的提示信息；如果连接成功，Telnet 程序将启动一个新的命令行窗口，该命令行窗口可以将客户端键盘输入的数据发送给服务器程序和显示从服务器程序接收的数据。

02 在 Telnet 程序启动的命令行窗口中，输入如下内容：

```
GET /test.html HTTP/1.1<回车>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

Host:<空格><回车>
<回车>

上面的第一行表示要获取该 Web 站点的/test.html 网页文档，期望使用 HTTP 1.1 版本的协议来交换数据，其中的<回车>代表按下回车键，GET 和 HTTP 中的每个字母必须全部大写。第二行表示发送的 Host 头信息为空，表示要访问该 Web 服务器上的默认站点，其中的<空格>和<回车>分别代表按下空格键和回车键。输入上面的内容后，Telnet 程序命令窗口中将显示出 Tomcat Web 服务器回送的内容，如图 2.32 所示。

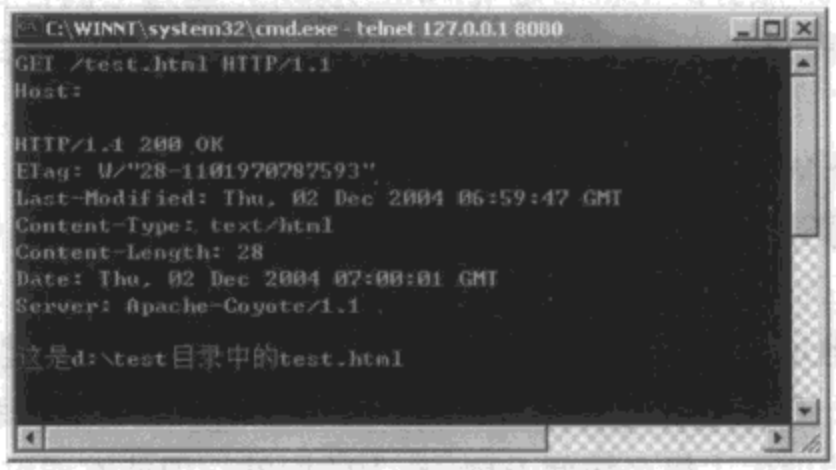


图 2.32

从图 2.32 中可以看到，Tomcat Web 服务器程序除了将 d:\test\test.html 文件中的内容作为其正文部分返回外，它还在正文前面返回其他一些状态信息。当在 IE 浏览器地址栏中输入 http://127.0.0.1:8080/test.html 时，浏览器将生成类似上述的请求信息发送给 Web 服务器，服务器程序则回送给浏览器类似上述的响应信息，只是浏览器获取服务器程序返回的内容以后，会对这些内容做更进一步的处理，例如，处理各种状态信息和按 HTML 语法格式将其中属于正文的部分（即 d:\test\test.html 文件中的内容）显示在浏览器窗口中。

多学两招

打开 Telnet 程序的本地回显功能

Windows 自带的 Telnet 程序是一个通用的 TCP 客户端程序，它可以访问各种 TCP 服务器，因此常被用做测试 TCP 服务器程序功能的工具。在默认情况下，Telnet 程序连接上 TCP 服务器后将不进行本地回显（即不显示键盘输入的内容），我们只能用盲打的方式键入字符，很不方便。如果要让 Telnet 程序对键入的字符进行本地回显，可以按如下步骤操作：

01 在命令行窗口中运行不带任何参数的 Telnet 命令，接着执行 Telnet 的 help 子命令查看所有命令的帮助信息。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

02 根据帮助信息的提示，接着再查看 set 命令的帮助，可以看到打开 Telnet 程序的本地回显功能的具体命令。不同操作系统中的命令的拼写可能会有一些细微差别，对于 Windows 2000 来说，该命令为：

```
set LOCAL_ECHO
```

03 执行 set LOCAL_ECHO 命令后，接着执行 quit 命令退出 Telnet 程序，以后在连接上 TCP 服务器的 Telnet 程序窗口中的输入的字符就可以在本地回显了。

但是，在有些版本的 Windows XP 系统中，即使按照上面的流程和 Telnet 程序的帮助信息进行操作后，Telnet 程序在有些情况下仍然会出现不能本地回显的问题，这是 Windows XP 系统中的 Telnet 程序的一个缺陷。如果读者遇到这种情况，可以参照以下步骤来解决。

01 重新启动一个 Telnet 程序连接到 TCP 服务器。

02 在连接上 TCP 服务器的 Telnet 程序窗口中同时按下“Ctrl”和“]”键，接着在出现的提示符窗口中执行 set localecho 命令，如图 2.33 所示。

图 2.33

执行 set localecho 命令后，**再次按下回车键退回 Telnet 程序与 TCP 服务器的连接窗口中**，此后在连接窗口中输入的内容就可以回显出来了。

注意，上面的方法只对当前的 Telnet 连接窗口有效，在以后新连接的 Telnet 程序窗口中仍然需要重复上面的过程来进行本地回显。

2.5.2 虚拟主机及实现原理

在一台计算机上可以使用 Tomcat 创建多个 Web 站点，并为每个 Web 站点设置不同的虚拟根目录和虚拟子目录，每个 Web 站点作为各自独立的网站分配给不同的公司或部门。通过这种方式，多个公司或部门的网站就可以共用同一台计算机，而用户感觉每个公司和部门都有各自独立的网站。多个没有实力在 Internet 上架设自己专用服务器的中小公司可以联合租用一台 Web 服务器，对外提供各自的 Web 服务而互不影响，这就是称之为虚拟主机的解决方案。

因为多个虚拟主机的 Web 站点是创建在同一台计算机上的，不管 Web 浏览器访问其中的哪一个站点，请求都会发送到这台计算机上。那么，这台计算机通过什么方式来区分 Web 浏览器要找的到底是哪一个站点呢？这个问题需要从两个方面来配合解决：一是 Web

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

服务器上的每个 Web 站点必须设置不同的标识信息；二是 Web 浏览器发出的连接和请求信息中必须包含 Web 站点的标识信息。

HTTP 协议的 Host 请求头用于指定客户端所要访问的 Web 站点的主机名，根据建立每个 TCP 服务器程序的两个要素（IP 地址和端口号）及 HTTP 协议的 Host 请求头的作用，一个 Web 站点可以通过三个部分来与其他 Web 站点相区别，即每个 Web 站点的标识信息由三个部分组成：IP 地址、端口号和主机名。在一台 Web 服务器上，可以使用不同的端口号、IP 地址或主机名，以及这三个要素的不同组合来创建多个 Web 站点。根据 Web 浏览器建立连接和发出请求消息时所指定的这三部分信息，Web 服务器就能够确定 Web 浏览器要访问的到底是这台 Web 服务器上的哪个站点。

2.5.3 基于主机名的虚拟主机

Web 浏览器与 Web 服务器建立连接后，除了将请求 URL 中的资源路径发送给 Web 服务器外，还会将 URL 中的主机名部分作为 HTTP 请求消息的 Host 头发送给 Web 服务器。例如，在浏览器地址栏中输入 http://www.it315.org，浏览器发送给 www.it315.org 主机上的 Web 服务器的请求消息内容如下：

```
GET  /  HTTP/1.1<回车>
Host:  www.it315.org<回车>
<回车>
```

如果一个 Web 服务器上建立的多个站点采用不同的主机名来作为各自的标识信息，当 Web 服务器接收到类似上面的访问请求消息后，根据 Host 头字段中所设置的主机名，就知道该选择哪个 Web 站点来进行响应。

Tomcat 的 server.xml 配置文件中有一个<Host>元素，一个<Host>元素用于建立一个 Web 站点，<Host>元素的父级元素为<Engine>元素。一个<Engine>元素构建一个处理客户端请求的引擎，它接受 Tomcat 的连接器传递来的访问请求，进行具体的处理后将结果返回连接器。在一个<Engine>元素中可以使用多个<Host>元素来建立多个 Web 站点，即一个引擎可以管理多个 Web 站点。<Host>元素的 name 属性用于指定其所对应的 Web 站点的主机名称，嵌套在同一个<Engine>元素中的多个<Host>元素的 name 属性不能相同。Tomcat 的 server.xml 配置文件中初始设置的<Host>元素内容如下：

```
<Host name="localhost" appBase="webapps" …>
…
</Host>
```

<Host>元素中的 appBase 属性指定一个路径，该路径将作为嵌套在它里面的<Context>元素的 docBase 属性中设置的相对路径的基准路径。

当 Tomcat 的连接器将接收到的访问请求交给关联的处理引擎时，处理引擎将比较请求消息中的 Host 头字段的值与相应的<Engine>元素中的所有<Host>元素的 name 属性值，

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

并选择匹配的<Host>元素所对应的 Web 站点下的内容来进行响应。如果相应的<Engine>元素中没有与请求消息的 Host 头字段匹配的<Host>元素，处理引擎将选择默认的 Web 站点来进行响应。只要<Engine>元素的 defaultHost 属性设置为嵌套在它里面的某个<Host>元素的 name 属性值，该<Host>元素所创建的 Web 站点就成该引擎的默认 Web 站点。例如，Tomcat 的 server.xml 文件中的<Engine>元素的默认设置如下：

```
<Engine name="Standalone" defaultHost="localhost" debug="0">
…
<Host name="localhost" appBase="webapps" …>
…
</Host>
…
</Engine>
```

上面的这段配置信息说明，该引擎的默认 Web 站点为嵌套在<Engine>元素中的 name 属性为“localhost”的<Host>元素所创建的 Web 站点。

在同一台计算机上建立多个基于主机名的虚拟主机后，Web 浏览器要访问其中的某个虚拟主机的资源时，在访问 URL 中必须采用主机名，而不能采用 IP 地址。这是因为 Web 浏览器要将 URL 中的主机名部分作为 HTTP 请求消息的 Host 头发送给 Web 服务器，如果 URL 中的主机名部分使用的是 IP 地址，那么，浏览器发出的请求消息中的 Host 头字段的值就是这个 IP 地址，而在同一台计算机上建立的多个基于主机名的虚拟主机共享同一个 IP 地址，在 Host 头字段使用 IP 地址根本就无法区分出不同的站点。

即使在 URL 中指定主机名，Web 浏览器也要先获得该主机名所对应的 IP 地址，再使用这个 IP 去连接 Web 服务器。所以，在建立基于主机名的虚拟主机时，除了要在 Tomcat 的 server.xml 文件中进行设置外，还需要在整个网络系统中建立主机名与 IP 地址的映射关系，即必须将主机名添加到名称解析系统，以便 Web 浏览器从名称解析系统中查询出主机名所对应的 IP 地址。建立主机名与 IP 地址的映射关系的惯用方式有两种：一是使用客户机本地的 Hosts 文件，二是使用 DNS（Domain Name System，域名系统）服务器。Hosts 文件和 DNS 的作用都是允许用户使用“友好”的、文本格式的主机名称，而不是数字格式的 IP 地址来访问网络中的计算机。Hosts 文件可用于小型的 Intranet（企业内部网），网络中的所有计算机上都需要在各自的 Hosts 文件中建立其所要访问的计算机的主机名称与 IP 地址的对应关系。DNS 通常用于大型的网络，特别是 Internet 上对外提供服务的计算机都要通过 DNS 来建立其主机名与 IP 地址的映射关系，网络上的每台计算机不用分别保存其所要访问的计算机的主机名称与 IP 地址的对应关系。客户机首先在本地的 Hosts 文件中查找主机名称所映射的 IP 地址，如果没有找到，再去查询 DNS 服务器。为了简单起见，这里仅介绍 Hosts 文件。对于 Windows 2000，Hosts 文件位于操作系统根目录（取决操作系统所在的分区，通常是 c:\winnt）下的 System32\Drivers\Etc 子目录中，默认情况下，该文件中有如下一行内容：

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
127.0.0.1       localhost
```

这行文本的作用就是将 IP 地址（127.0.0.1）映射成主机名（localhost），这也就是在 IE 浏览器地址栏中可以使用 localhost 访问本地 Web 服务器的原因。如果要增加更多的主机名与 IP 地址的映射，可以在 Hosts 文件中增加更多的行，然后参照上面这行内容的格式在每行中填写 IP 地址和相应的主机名。

使用 Tomcat 建立基于主机名的虚拟主机

01 在 d:盘下创建名称为 VirtualHost1 和 VirtualHost2 两个目录，并在这两个目录中分别创建一个名为 test.html 的文件，在 d:\VirtualHost1\test.html 文件中写入如下一行内容：

```
这是d:\VirtualHost1目录中的test.html文件
```

在 d:\VirtualHost2\test.html 文件中写入如下一行内容：

```
这是d:\VirtualHost2目录中的test.html文件
```

02 用 UltraEdit 打开<Tomcat 主目录>/conf 目录下的 server.xml 文件，使用“查找”菜单查找内容为“</Host>”的行，紧接该行下面增加一对<Host></Host>标签。参照前面的<Host>标签的属性设置情况，设置新增的<Host>标签的属性，并在它里面嵌套一个设置该 Web 站点根目录的<Context>元素，最终的内容如下：

```
<Host name="site1" debug="0" appBase="d:\VirtualHost1">
    <Context path="" docBase="." debug="0"/>
</Host>
```

这样，将创建一个新的 Web 站点。上面的<Context>元素的 docBase 属性值被设置为一个点（.），即表示使用<Host>元素的 appBase 属性中所设置的路径作为这个<Context>所映射的目录。

03 在上面新增的<Host></Host>标签对中再增加一对<Host></Host>标签，并将它设置为如下形式：

```
<Host name="site2" debug="0" appBase="d:\VirtualHost2">
    <Context path="" docBase="." debug="0"/>
</Host>
```

这又创建了一个新的 Web 站点，该站点的主机名称为 site2，根目录对应的本地文件系统目录为 d:\VirtualHost2。

04 保存修改后的 server.xml 文件，重新启动 Tomcat Web 服务器程序。打开一个新的命令行窗口，并在这个命令行窗口中执行如下命令：

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
telnet 127.0.0.1 8080
```

接着在连接成功的 Telnet 程序命令窗口中，输入如下内容：

```
GET /test.html HTTP/1.1<回车>
Host:<空格>site1<回车>
<回车>
```

这时在 Telnet 程序窗口中可以看到，Web 服务器返回内容的正文部分为 d:\VirtualHost1\test.html 文件中的内容。接着 Web 服务器返回内容的下边，输入如下内容：

```
GET /test.html HTTP/1.1<回车>
Host:<空格>site2<回车>
<回车>
```

在 Telnet 程序窗口中又可以看到，Web 服务器这次返回内容的正文部分为 d:\VirtualHost2\test.html 文件中的内容。接着 Web 服务器返回内容的下边，输入如下内容：

```
GET /test.html HTTP/1.1<回车>
Host:<空格><回车>
<回车>
```

在 Telnet 程序窗口中又可以看到，Web 服务器这次返回内容的正文部分为 d:\test\test.html 文件中的内容。接着 Web 服务器返回内容的下边，再次输入如下内容：

```
GET /test.html HTTP/1.1<回车>
Host:<空格>xxx<回车>
<回车>
```

在 Telnet 程序窗口中又可以看到，Web 服务器这次返回内容的正文部分仍然为 d:\test\test.html 文件中的内容。

上面的整个交互过程如图 2.34 所示。Tomcat 根据第 1 次和第 2 次请求中所指定 Host 头的值，查找 server.xml 文件中与之相匹配的<Host>元素的 name 属性值，并以匹配的<Host>元素所设置的 Web 站点来响应；在第 3 次和第 4 次请求中所指定 Host 头的值，在 server.xml 文件中没有与之对应的<Host>元素的 name 属性值，Tomcat 将以默认的 Web 站点来响应。

05 在 Windows 2000 系统中，用记事本程序打开 c:\winnt\System32\Drivers\Etc\Hosts 文件，可以看到有如下一行内容：

```
127.0.0.1       localhost
```

紧接这行文本的下面，用它复制出两行文本，并将这两行文本中的 localhost 分别修改成 site1 和 site2，这样，就使用 Hosts 文件为当前计算机设置了多个主机名。如果要用 site1 和 site2 这个两主机名访问其他计算机上的 Web 站点，则应将 127.0.0.1 修改成其他计算机的实际 IP 地址。保存 Hosts 文件后，在命令行窗口中执行“ping site1”和“ping site2”命令，查

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

看该主机名是否被正确解析到相应的 IP 地址上。在建立 site1 和 site2 与计算机 IP 地址的映射后，在 IE 浏览器地址栏中输入 http://localhost:8080/test.html、http://site1:8080/test.html 和 http://site2:8080/test.html，可以看到浏览器将显出各自站点中的 test.html 网页文件的内容。

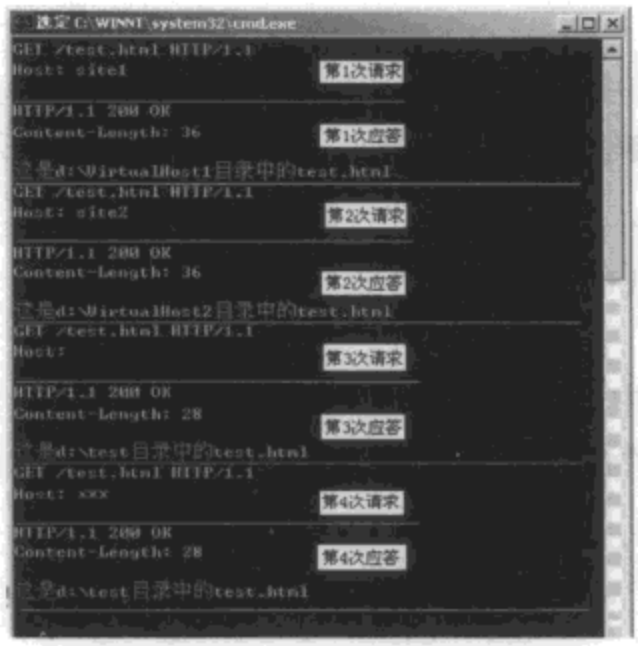


图 2.34

可见，使用主机名的方式在同一台 Web 服务器上创建多个虚拟主机后，在 Web 浏览器中使用主机名访问 Web 服务器时，Web 服务器将选择与该主机名关联的 Web 站点进行响应。通过这种方式，多个 Web 站点可以共享同一个 IP 地址和相同的端口号，惟一不足的就是 Web 浏览不能通过 IP 地址去访问这些 Web 站点。基于主机名的虚拟主机是目前 Internet 上的大多数虚拟主机业务提供商所通常采用的方式。只要找出几个小型公司或个人的网站，如果在 Web 浏览器地址栏中输入“http://主机名”可以访问到该 Web 站点，接着在命令行窗口中执行“ping 主机名”，查看该域名对应的 IP 地址，然后在 Web 浏览器地址栏中输入“http://IP 地址”时，却无法访问这个 Web 站点了，那么，这个 Web 站点就是一个基于主机名的虚拟主机，它与其他一些 Web 站点共享一台 Web 服务器，而不是自己独享一台 Web 服务器。

注意 当使用安全套接字层（SSL）时，不能使用主机头字段来实现虚拟主机，这是因为使用 SSL 的 HTTP 请求有加密保护。主机头字段是加密内容的一部分，不能被解释和路由到正确的站点。

2.5.4 基于端口号的虚拟主机

通过使用不同的监听端口号，Web 服务器可以在一个 IP 地址或主机名上维护多个 Web 站点。为各个 Web 站点指定不同的端口号后，Web 服务器接收到浏览器的访问请求时，根

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

据浏览器连接的端口号的不同，就知道该选择哪个 Web 站点来进行响应。

在 Tomcat 的 server.xml 文件中，一个<Connector>元素用于配置一个对外的连接器，对于 Web 服务器上的每一个用于接收访问请求的监听端口号，都要使用一个单独的<Connector>元素。一个<Connector>元素对应的连接器必须有一个相关的<Engine>元素去处理它所接收到的访问请求，将一个或多个<Connector>元素与某个<Engine>元素相关联的方式就是将它们嵌套在同一个<Service>元素中。<Service>元素的基本结构如下：

```
<Service name="…">
    <Connector port="…" …/>
    <Engine name="…" >
        <Host name="…" >
            <Context …>
            </Context>
        </Host>
    </Engine>
</Service>
```

要想使用 Tomcat 建立基于端口号的多个虚拟主机，需要在 server.xml 文件中创建多个<service>元素，并且每个<service>元素中的<Connector>元素的 port 属性的值不同。在 server.xml 文件中，可以查找到一个 port 属性值为 8080 的<Connector>元素，如果将该元素的 port 属性值修改为 80，那么，客户端访问与该<Connector>元素相关的站点时的端口号就应为 80，由于 80 是 HTTP 协议的默认端口号，浏览器访问这些站点时可以省略 URL 中的端口号。

虽然可以在一台 Web 服务器上用不同的端口号创建多个 Web 站点，但绝大多数 Internet 用户都不乐意去专门用心记住某个站点所使用的特殊端口号，每个公司也都希望自己的站点使用默认端口号 80。使用不同的端口号来创建多个 Web 站点的情况在 Internet 上并不常见，这种情况通常出现在企业内部网和一些需要在 Internet 上使用的临时性个人网站。一些个人网站依附在别的网站上，它没有申请自己的域名，无法建立基于主机名的虚拟主机，仍然要使用主网站的域名或 IP 地址，这时才需要与主网站通过使用不同的端口号进行区别。

使用 Tomcat 建立基于端口号的虚拟主机

01 用 UltraEdit 打开<Tomcat 主目录>/conf 目录下的 server.xml 文件，使用“查找”菜单查找内容为“</Service>”的行，紧接该行下面增加如下内容：

```
<Service name="Tomcat-Second">
    <Connector className="org.apache.coyote.tomcat4.CoyoteConnector"
        port="8081" minProcessors="5" maxProcessors="75"
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        enableLookups="true" redirectPort="8443"
        acceptCount="100" debug="0" connectionTimeout="20000"
        useURIValidationHack="false" disableUploadTimeout="true" />
    <Engine name="Second" debug="0">
        <Host name="site1" debug="0" appBase="d:\VirtualHost1">
            <Context path="" docBase="." debug="0"/>
        </Host>
        <Host name="site2" debug="0" appBase="d:\VirtualHost2">
            <Context path="" docBase="." debug="0"/>
        </Host>
    </Engine>
</Service>
```

注意 这里的<Service>元素的 name 属性值的设置应与前面的<Service>元素的 name 属性值不同。在编写<Connector>元素时，应直接复制原有的<Service>元素中的 port 属性值为 8080 的<Connector>元素内容，然后将其 port 属性值设置为 8081。在编写<Engine>元素时，应将在 2.5.3 节中增加的两对<Host>标签及其中嵌套的所有内容复制下来，然后插入进<Engine></Engine>标签对中。

02 保存 server.xml 文件后，重新启动 Tomcat Web 服务器程序，在 IE 浏览器地址栏中输入 http://site1:8081/test.html 和 http://site2:8081/test.html，可以看到浏览器将显示这两个站点中的 test.html 网页文件的内容。

2.5.5 基于 IP 地址的虚拟主机

一台计算机上可以设置多个 IP 地址，使用其中任何一个 IP 地址都可以连接上 Web 服务器程序，Web 服务器也可以检测出浏览器所连接的本地 IP 地址。因此，在同一台计算机上可以使用多个 IP 地址来维护多个 Web 站点，Web 服务器根据浏览器所连接的本地 IP 地址的不同，就知道该选择哪个 Web 站点来进行响应。由于 HTTP 1.0 不支持 Host 头字段，在保证所有站点统一使用端口号 80 的情况下，人们早期只能采用不同的 IP 地址来建立虚拟主机。使用基于 IP 地址建立虚拟主机的方式，要求为每个站点分配一个独立的 IP 地址。在 Internet IP 地址稀缺的情况下，自从有 HTTP 1.1 可以使用不同的主机名来建立多个 Web 站点后，虚拟主机业务提供商一般不再建立基于 IP 地址的虚拟主机了。正因为建立基于 IP 地址的虚拟主机的意义不大，所以 Tomcat 没有提供建立基于 IP 地址的虚拟主机方式。

2.6 与专用 Web 服务器的集成

虽然 Tomcat 也可以作为独立的 Web 服务器软件运行，但其处理静态 HTML 文件的速度比不上 Apache 和 IIS 等专业的 Web 服务器，并且其作为 Web 服务器的功能也不如 Apache

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

和 IIS 强大。如果一个 Web 站点既要支持 Servlet 和 JSP，又要有较高的普通 HTML 页面处理速度和更多的 Web 站点功能，那么可以将 Tomcat 与 Apache 或 IIS 集成起来使用，让 Apache 和 IIS 处理静态 HTML 文件和完成其他 Web 功能，让 Tomcat 专门处理与 Servlet 和 JSP 相关的内容。实现 Tomcat 与 Apache 和 IIS 之间的集成，需要在 Apache 和 IIS 中安装由 Tomcat 开发人员专门为此开发的模块，这些模块称为 Tomcat 的连接器。不同版本的 Tomcat 的连接器的配置细节不尽相同，请使用 Tomcat 5.x 学习本节的讲解。

2.6.1 集成的考虑因素

将 Tomcat 与其他 Web 服务器进行集成的理由有很多，但不用将 Tomcat 与其他 Web 服务器进行集成的理由也有很多。特别是 Tomcat 5.x 以后的版本的性能有了很大提升，将 Tomcat 与其他 Web 服务器进行集成到底能带来多大程度的性能改善，已经是一件很难说清楚的事情了。读者应根据自己的实际情况和需要，决定你的系统是否有必要将 Tomcat 与其他 Web 服务器进行集成，或者干脆单独使用 Tomcat 来架设你的站点。Apache 网站上列举了一些是否要将 Tomcat 与其他 Web 服务器进行集成的考虑因素，可以作为参考。

访问 Apache 站点内的 Tomcat 首页，可以看到其左侧有一个名称为“FAQ”的超链接，如图 2.35 所示。

单击“FAQ”超链接，在进入的页面中可以看到一个关于连接器（Connectors）的超链接，如图 2.36 所示。

单击“Connectors”超链接，进入 http://tomcat.apache.org/faq/connectors.html 页面，该页面中介绍 Tomcat 与其他 Web 服务器集成的一些考虑因素。

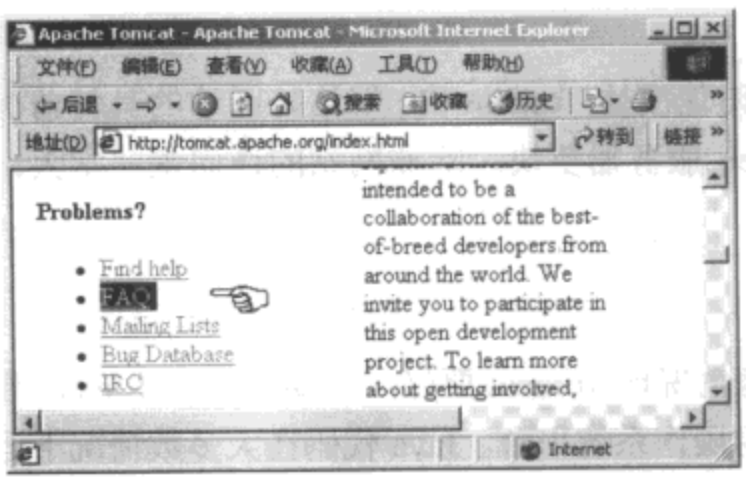


图 2.35

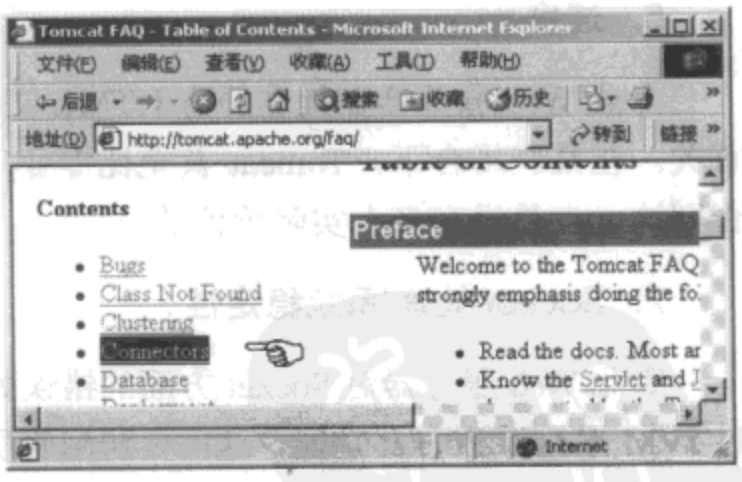


图 2.36

1. 集群

让专业 Web 服务器作为多个提供相同内容的 Tomcat 服务器的前端系统，客户端的访问请求先交给专业 Web 服务器，专业 Web 服务器再将访问请求均衡地分配给这些 Tomcat 服务器去处理。如果你的系统已经使用一个硬件负载均衡设备，或者已经使用 Tomcat 的

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

集群功能，那么，这一因素就不必再考虑了。说明：集群系统就是联合多台具有相同功能的计算机对外提供服务的系统，负载均衡则是指把所有的访问请求均衡地分配到集群系统中的每台计算机上。

2. 集群/安全

让专业 Web 服务器作为多个 Tomcat 服务器的前端系统，将不同的 URL 名称空间（例如，/app1/、/app2/、/app3/或虚拟主机）的访问分配给不同的 Tomcat 服务器去处理。每个 Tomcat 服务器位于专业 Web 服务器的后端，它们位于被保护的区域内，用户不能够直接访问它们，所以，系统管理人员只需要考虑专业 Web 服务器的安全问题，而不用考虑所有的 Tomcat 服务器的安全问题。

3. 安全

安全是一个令人拿不定主意的问题，尽管 Java 有安全管理器，但专业 Web 服务器对于安全问题有更多的考虑和处理技巧。

4. 附加功能

在专业服务器中集成 CGI、Perl、PHP 等功能比较容易，而在 Tomcat 中要集成这些功能非常麻烦。

5. 特殊功能

使用专业 Web 服务器作为 Tomcat 的前端系统，能够执行很多 Tomcat 不支持或者现在还不能很好支持的特殊功能。

6. 速度

专业 Web 服务器在处理静态 HTML 文件时速度比 Tomcat 快，但这与网站的具体情况有关。在某些场合下，Tomcat 甚至比专业 Web 服务器更快。除非网站的访问量非常大，否则这一点是没有多大实际价值的。

7. Socket 处理/系统稳定性

专业 Web 服务器对 Socket 方面的错误处理情况比 Tomcat 要好，主要原因是 Tomcat 通过 JVM 来执行 Socket 功能，这需要跨越底层的操作系统平台。Java 代码在大多数情况下运行得都很好，但是当服务器遭受来自无效 IP 地址发出的半连接、非法数据包或无效请求的猛烈攻击时，专业 Web 服务器在应对这些情况时，比基于 JVM 的程序做得要好得多。

2.6.2 集成技术的选择

Apache 组织先后推出了 mod_webapp、mod_jserv、JK 和 JK2、mod_proxy 等一系列的技术来实现 Tomcat 与其他 Web 服务器之间的集成。在 2.6.1 节中进入的 http://tomcat.apache.

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

org/faq/connectors.html 页面中也有一段关于这些技术的描述，内容如下：

（1）mod_webapp 已经过时，因缺少对它感兴趣的开发者而不再被 Tomcat 支持。

（2）mod_jserv 是早期使用 AJP 协议的连接器，目前也不再被支持和使用。

说明 AJP 是 Tomcat 与其他 Web 服务器之间通信的协议，能提供较高的通信速度和效率。在配置 Tomcat 与 Web 服务器集成时，读者可以不必关心 AJP 协议的细节。

（3）JK 是目前集成 Tomcat 与其他 Web 服务器的首选技术，它与 Tomcat 服务器之间采用 AJP 协议通信，在实际应用中非常稳定。

（4）JK2 本来是在 JK 基础上改造成的连接器，虽然 JK2 曾经是比 JK 更新的技术，但因其配置过程更复杂，导致缺少对 JK2 感兴趣的开发人员和使用人员，所以，JK2 小组停止了对 JK2 的继续开发。JK2 目前也不被支持了，它的许多功能都被移植进了新版本的 JK 中。

（5）mod_proxy 是比 JK 的配置过程简单得多的连接器，但它不能实现有会话状态的负载均衡，如果不需要此方面的功能，可以使用 mod_proxy 代替 JK 来实现 Tomcat 与其他 Web 服务器的集成。

（6）mod_proxy_ajp 是在 mod_proxy 基础上改写的连接器，mod_proxy_ajp 支持负载均衡，但它要在 Apache 2.2 版本中才能使用，并且是随着 Apache 服务器程序一起发布的。

2.6.3 JK 插件

IIS 和 Apache 等专业 Web 服务器都支持插件技术，它们可以将符合某种条件的请求交给插件去处理，IIS 和 Apache 能够与 Tomcat 进行集成的关键原理就是这种插件技术。JK 是 Tomcat 开发小组提供给其他 Web 服务器和 Tomcat 集成的插件，它是目前集成 Tomcat 与其他 Web 服务器的首选技术，需要作为插件安装到其他 Web 服务器上（此处及本节后面的 Web 服务器指 IIS 和 Apache 等专业服务器）。当在 Web 服务器上安装和配置好 JK 插件后，Web 服务器会根据为 JK 插件预先配置好的 URL 映射信息，将接收到的这些 URL 访问请求交给 JK 插件来处理，JK 插件再把 Web 服务器转交来的访问请求转发给 Tomcat 服务器处理。假定在为 JK 插件预先配置好的 URL 映射信息中，所有“/*.jsp”形式的 URL 都由 Tomcat 服务器来处理，那么，当用户请求访问“/index.jsp”时，Web 服务器将调用 JK 插件来处理，JK 插件又把这个请求转发给 Tomcat 服务器处理，Tomcat 服务器最后将响应结果经 JK 插件传给 Web 服务器。

JK 插件是供给 Web 服务器使用的，不同的 Web 服务器下的插件形式是不同的，Tomcat 开发小组为不同的 Web 服务器提供相应的 JK 插件。另外，不同版本的 Tomcat 与其他 Web 服务器的各个版本集成时的 JK 插件也可能不同，具体的信息需要查看某个版本的 JK 插件的说明和自己动手实验，也可以在 Internet 上查阅别人讨论的经验。笔者实验用的 Web 服

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

务器为 IIS 5.0 和 Apache 2.0.54，Tomcat 服务器为 Tomcat 5.5.12，在笔者编写本书时，Apache 网站上提供的 JK 插件的最新版本是 JK 1.2.15，但笔者使用这个版本的 JK 插件将 Web 服务器与 Tomcat 进行集成时，实验未能成功！最后，笔者使用 JK 1.2.14，将 Tomcat 与 IIS 和 Apache 都成功地进行了集成，并且非常稳定。

下载 JK 插件

01 进入 http://tomcat.apache.org 站点首页，可以看到其左侧的 download 栏目中列出了“Tomcat Connectors”的超链接，单击该超链接进入 Tomcat 连接器的下载页面，在其中可以看到 JK 1.2 的二进制发行版的超链接，如图 2.37 所示。

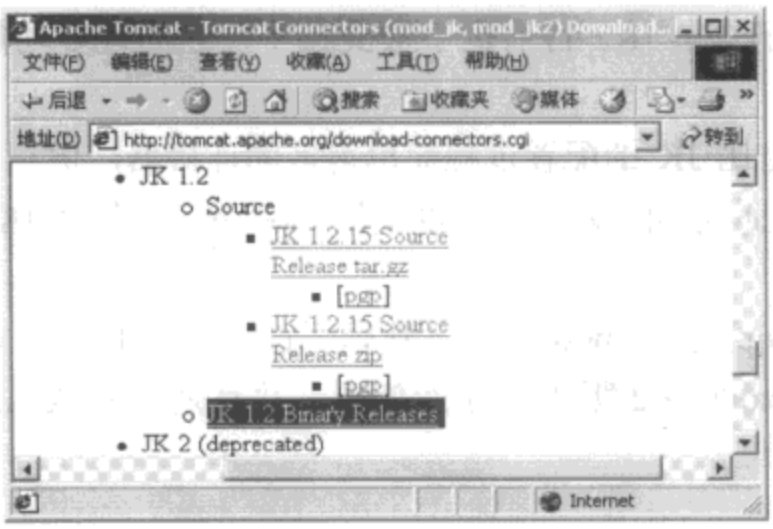


图 2.37

02 进入“JK 1.2 Binary Releases”超链接指向的页面，可以看到一个名为“win32/”的超链接，如图 2.38 所示。

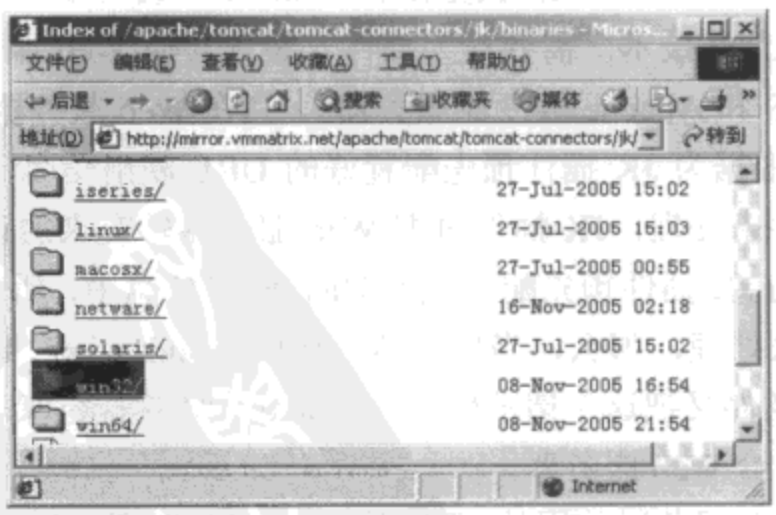


图 2.38

03 进入“win32/”超链接指向的页面，可以看到其中列出了 JK 插件的三个版本的下载超链接，如图 2.39 所示。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

04 进入“jk-1.2.14/”超链接所指向的页面，可以看到用于各种 Web 服务器的 JK 插件的下载超链接，如图 2.40 所示。

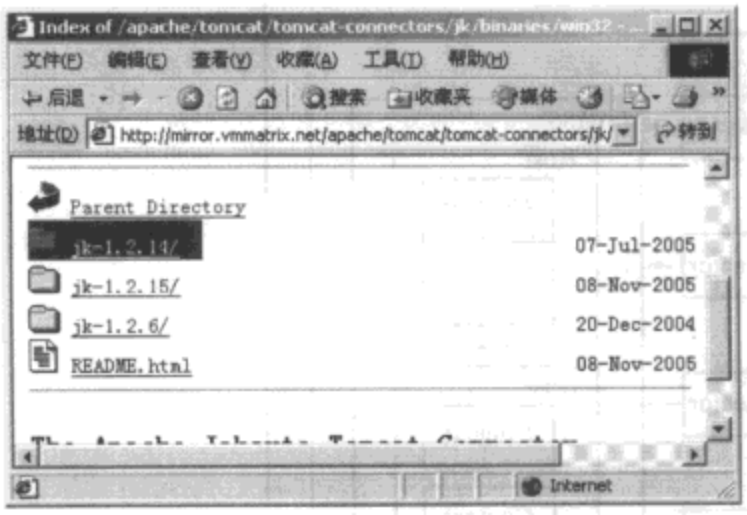


图 2.39

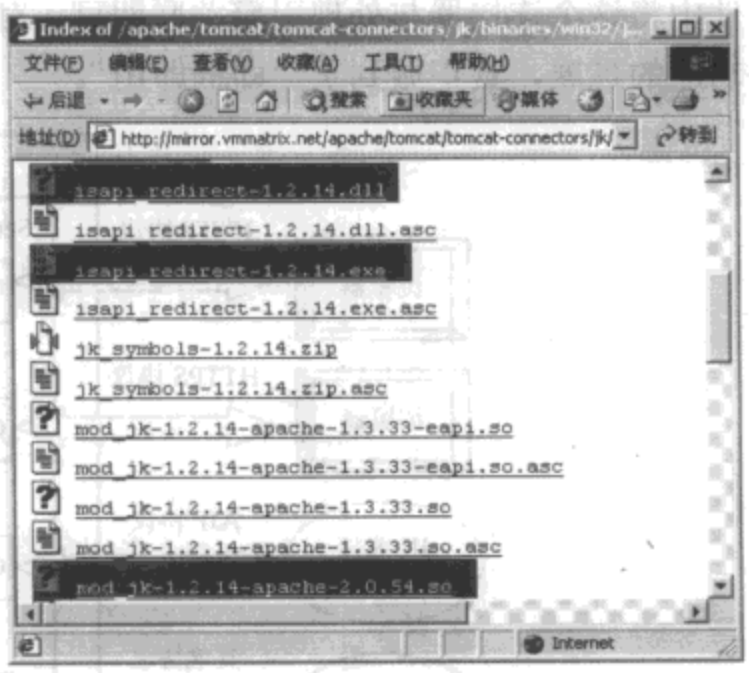


图 2.40

在图 2.40 所示的页面中，用于 IIS 的 JK 插件提供了两种发行方式：isapi_redirect-1.2.14.dll 文件和 isapi_redirect-1.2.14.exe 文件，下载其中任何一个都可以，但这里要求读者下载 isapi_redirect-1.2.14.exe 文件，因为在该 exe 文件安装后的目录中可以找到配置文件的范例。针对 Apache 服务器的不同版本，需要使用不同的 JK 插件，图 2.40 所示的页面中提供了各个 Apache 版本的 JK 插件，这里要求下载 Apache 2.0.54 的 JK 插件，即 mod_jk1.2.14-apache-2.0.54.so 文件。

提示 读者在自己下载 JK 插件时，可以选择试用最新版本的 JK 插件，如果最新版本不能很好工作时，再选择稍旧一点的版本。

2.6.4 Tomcat 体系结构与 JK 插件的工作原理

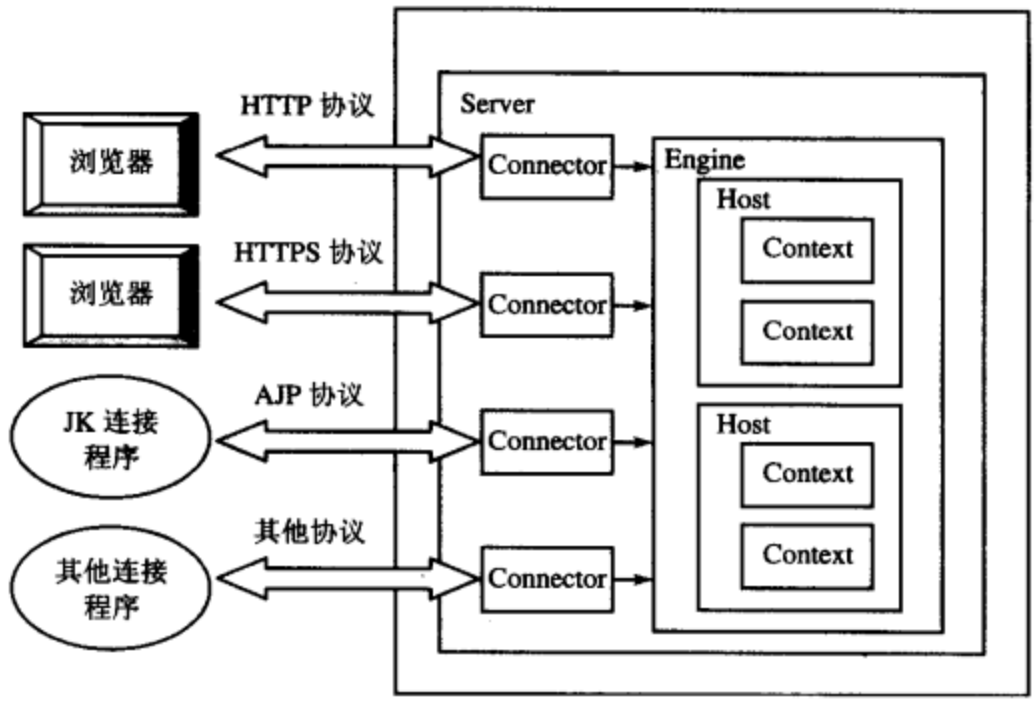
从 Tomcat 的 server.xml 文件的文档结构中可以看出 Tomcat 的体系结构如图 2.41 所示。

其中，一个<Connector>元素用于配置一个对外的连接器。Tomcat 提供几种连接器，它们分别用于接收某种通信协议的外部访问请求，例如，采用 HTTP 协议的连接器用于接收外部的 HTTP 访问请求，采用 HTTPS 协议的连接器用于接收外部的 HTTPS 访问请求。server.xml 文件中的一个<Engine>元素用于配置一个 Tomcat 的处理引擎，一个<Engine>元素可以与多个<Connector>元素相关联，也就是多个连接器可以共用一个处理引擎。每个连接器接收到不同协议和格式的访问请求后，将它们转换成统一的格式后再交给与它们关联

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

的引擎去处理。Tomcat 的这种设计架构具有非常大的灵活性，如果要让 Tomcat 支持某种新的协议和格式的访问请求时，只需要开发与之相应的连接器，然后在 server.xml 配置文件中将这个连接器与处理引擎关联即可，对原有系统不需要进行任何改动，就实现了系统功能的扩展，真正做到了“即插即用”。



Tomcat 与 JK 插件的连接采用 AJP 协议进行通信，Tomcat 要接受 JK 插件的访问请求，需要在 server.xml 文件中配置一个相应<Connector>元素。其实，在 Tomcat 提供的 server.xml 文件中，已经默认设置了与 JK 插件之间的连接器，例如，在 Tomcat 5.5.12 的 server.xml 中，有如下一段连接器配置信息：

```
<Connector port="8009"
enableLookups="false" redirectPort="8443" protocol="AJP/1.3" />
```

这段信息说明该 Tomcat 服务器在 8009 端口上接收采用 AJP 1.3 协议的访问请求，JK 插件应该连接到该服务器的 8009 端口上，并采用 AJP 1.3 协议与该 Tomcat 服务器通信。可以修改上面的<Connector>元素中的 port 属性值，例如，将 8009 修改为 8007，那么在 JK 插件的属性设置中，也应该将它要连接的 Tomcat 服务器的端口号修改为 8007。

要想让 JK 插件正常工作，需要解决如下一些问题：

（1）在 Web 服务器中正确安装 JK 插件。

（2）JK 插件需要知道把请求转发给哪个 Tomcat 服务器上的哪个连接器，也就是需要知道该连接器的通信协议和地址。

（3）JK 插件需要知道它负责处理的 URL，以便通知 Web 服务器把这些 URL 转交给它处理；另外，一个 JK 插件可以向多个 Tomcat 服务器转发请求，它需要知道哪些 URL 应转发给哪个 Tomcat 服务器。

（4）JK 本身的一些配置信息，例如，记录日志的文件路径、记录日志的级别等信息，

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

如果第（2）和第（3）个问题都在各自的一个属性文件中进行设置，那么，还需要告诉这两个属性文件的位置路径。

JK 插件的配置和工作细节如图 2.42 所示，专业 Web 服务器接收到浏览器的访问请求后，将符合某种条件的请求交给 JK 插件，JK 插件再根据配置文件的设置将请求交给相应的 Tomcat 去处理。其中，workers.properties 文件用于设置与 JK 插件进行通信的 Tomcat 服务器上的连接器的通信协议和地址；uriworkermap.properties 文件用于设置 JK 插件负责处理的 URL 和具体处理各个 URL 的 Tomcat 服务器；isapi_redirect.properties 文件用于设置 JK 本身的配置信息，以及 workers.properties 文件和 uriworkermap.properties 文件的存储位置。

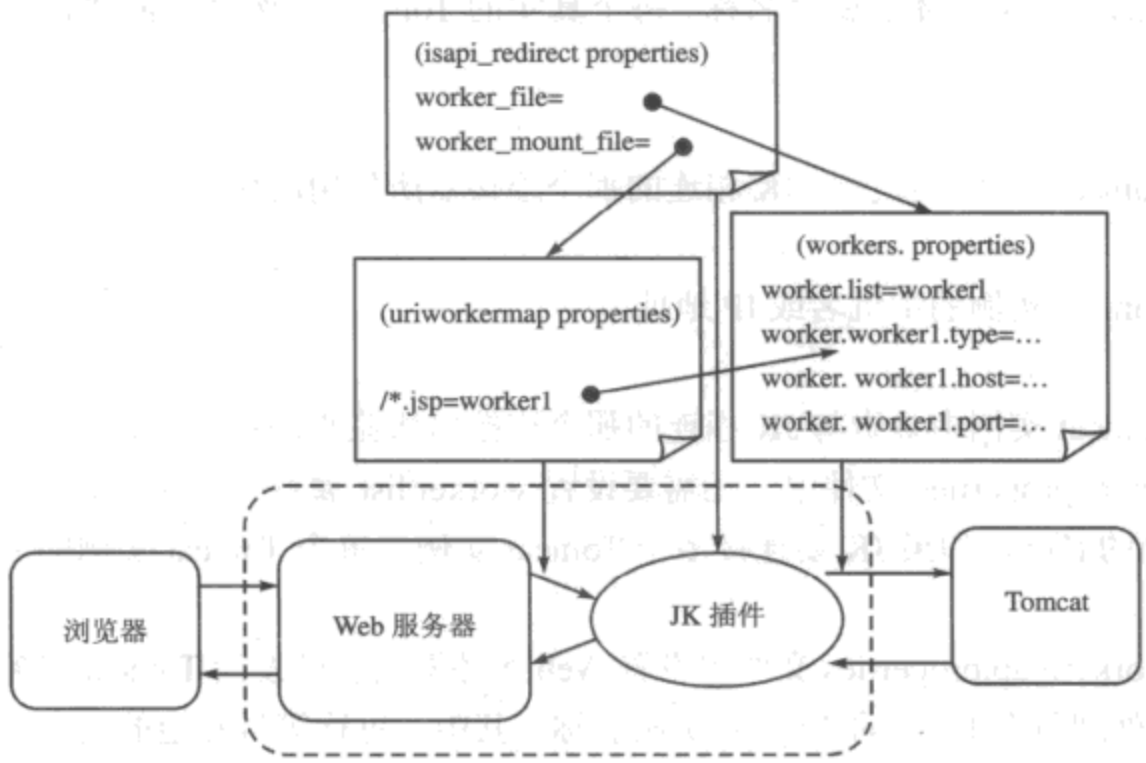


图 2.42

下载并安装 isapi_redirect-1.2.14.exe 程序后，在安装主目录的 conf 子目录中可以找到 workers.properties 和 uriworkermap.properties 的例子文件，读者可以通读一下这两个例子文件（包括其中的注释）。为了便于讲解，下面列出作者自己编写的 workers.properties 和 uriworkermap.properties 文件，如例程 2-4 和例程 2-5 所示。

例程 2-4 workers.properties

```
worker.list=worker1,worker2
worker.worker1.type=ajp13
worker.worker1.host=localhost
worker.worker1.port=8009
worker.worker2.type=ajp13
worker.worker2.host=192.168.0.3
worker.worker2.port=8009
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

例程 2-5 uriworkermap.properties

```
/*.jsp=worker1
/myweb/*=worker1
/jsp-examples/*=worker2
```

在 workers.properties 文件中设置了与 JK 插件进行连接的每个 Tomcat 实例的属性。每个 Tomcat 实例的属性名由“.”符号分隔成的三部分组成，例如，worker.worker1.host，其中，第一部分固定为“worker”；第二部分是为要连接的一个 Tomcat 实例的名称，它是由 workers.properties 文件编写者自行定义的一个名称，可以包含 [A~Z][a~z][0~9] 中任意的字符；第三部分就是具体的属性名称。每个真实的 Tomcat 实例至少需要设置下面的三个属性：

- **type**

  指定 Tomcat 实例中负责与 JK 相连的那个连接器所使用的协议类型。

- **host**

  指定 Tomcat 实例的主机名或 IP 地址。

- **port**

  指定 Tomcat 实例中负责与 JK 相连的那个连接器所监听的端口。

在 workers.properties 文件中，还需要设置 worker.list 属性来指定 JK 插件所要连接的 Tomcat 实例的名称，如果 JK 要连接多个 Tomcat 实例，每个 Tomcat 实例的名称之间使用逗号分隔。

在 uriworkermap.properties 文件中设置 Web 服务器需要转发给 Tomcat 服务器去处理的 URL，以及处理每个 URL 的 Tomcat 实例名称。其中，每行等号左边的内容表示要转发给 Tomcat 服务器处理的 URL，可以使用“*”通配符来匹配一组 URL，等号右边是处理该组 URL 的 Tomcat 实例的名称，该名称必须是在 workers.properties 文件中定义的名称。

在 workers.properties 文件中还可以将某个 Tomcat 实例的 type 属性设置为 lb，表示这个 Tomcat 实例并不指向真正的 Tomcat 服务器，而是代表一个实现负载均衡（Load Balance）的系统。代表负载均衡系统的 Tomcat 实例是一种虚拟的 Tomcat 实例，它没有 host 和 port 属性，而是需要设置一个 balance_workers 属性。balance_workers 属性中列出了组成该负载均衡系统的所有其他真实的 Tomcat 实例的名称。参与了一个负载均衡系统的每个 Tomcat 实例的名称不能出现在 worker.list 属性中，在 worker.list 属性中只能出现代表整个负载均衡系统的 Tomcat 实例的名称，JK 插件通过代表整个负载均衡系统的 Tomcat 实例去管理组成该负载均衡系统的每个真实的 Tomcat 实例。isapi_redirect-1.2.14.exe 程序提供的例子 workers.properties 文件采用的就是这种设置，如下所示：

```
worker.list=wlb
worker.ajp13w.type=ajp13
worker.ajp13w.host=localhost
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
worker.ajp13w.port=8009
worker.wlb.type=lb
worker.wlb.balance_workers=ajp13w
```

上面的设置信息定义一个名为 wlb 的负载均衡系统，该负载均衡系统中只加入一个名为 ajp13w 的真实 Tomcat 实例。

查阅 JK 连接器的帮助文档

01 进入 http://tomcat.apache.org 站点首页，可以看到其左侧的 Documentation 栏目中列出了“Tomcat Connectors”的超链接，如图 2.43 所示。

02 单击“Tomcat Connectors”超链接，进入 Tomcat 连接器的文档页面，如图 2.44 所示。

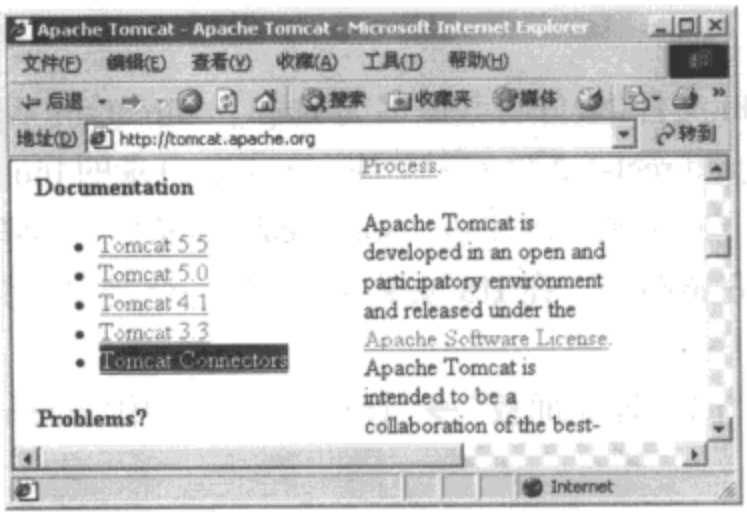


图 2.43

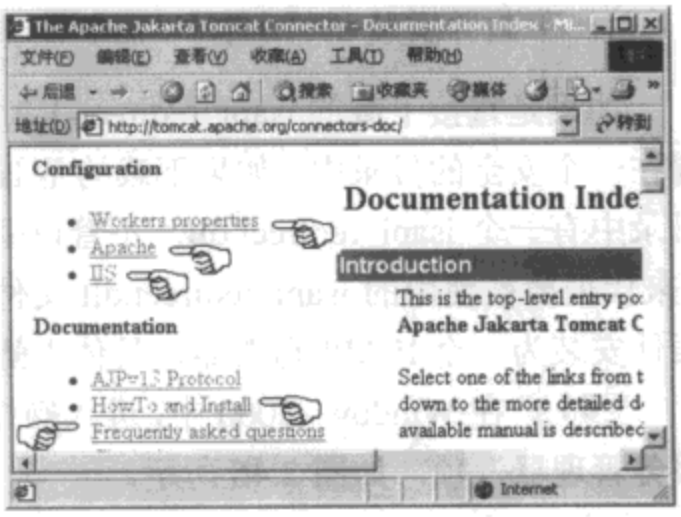


图 2.44

在 Tomcat 连接器的文档页面中，提供在各种 Web 服务器中安装和配置 JK 插件的相关信息的超链接，其中笔者用手指符号标注的超链接，是读者遇到配置 JK 插件的棘手问题时应重点查阅的地方。在 Tomcat 连接器的文档页面中的 Headlines 部分，还可以看到有关 JK2 的描述信息，如图 2.45 所示。

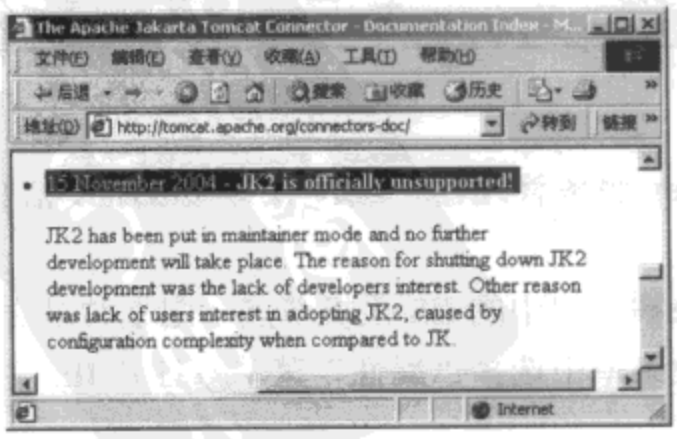


图 2.45

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

图 2.45 中的这段描述信息的大致意思是：因为 JK2 的配置过程比 JK 更复杂，导致缺少对 JK2 感兴趣的开发人员和使用人员，所以，Apache 小组停止了对 JK2 的继续开发，JK2 已经进入维护状态。

03 分别进入图 2.44 中的每个手指所指向的超链接，粗略浏览所有信息，等到有需要时再有针对性地详细阅读。

2.6.5 Tomcat 与 IIS 服务器集成

IIS（Internet Information Service）服务器是微软开发的在 Windows 上运行的 Web 服务器，功能强大，且直接支持 ASP 和 ASP.NET。把 Tomcat 与 IIS 服务器集成，可以扩展 IIS 的功能，使它支持 Java Web 应用。

假定计算机上已经安装 Tomcat 和 IIS 服务器，它们都能独立运行，并且在同时运行时没有端口号冲突问题，那么，请按以下步骤将 Tomcat 与 IIS 集成。

1. 下载并安装 JK 插件

如果是直接下载的 isapi_redirect-1.2.14.dll 文件，最好将其改名为 isapi_redirect.dll 后放置在一个安全的目录中。如果下载的是 isapi_redirect-1.2.14.exe 文件，在安装目录的 bin 子目录中有一个 isapi_redirect.dll。不管以哪种方式下载，这里以*<isapi_redirect.dll 文件路径>*来代表要被使用的 isapi_redirect.dll 文件的完整路径。在 IIS 安装 JK 插件，就是要把 JK 插件安装为一个 ISAPI 筛选器，具体步骤如下：

01 在 Windows 2000 中，进入操作系统的“控制面板”→“管理工具”，选中“Internet 服务管理器”项，如图 2.46 所示。

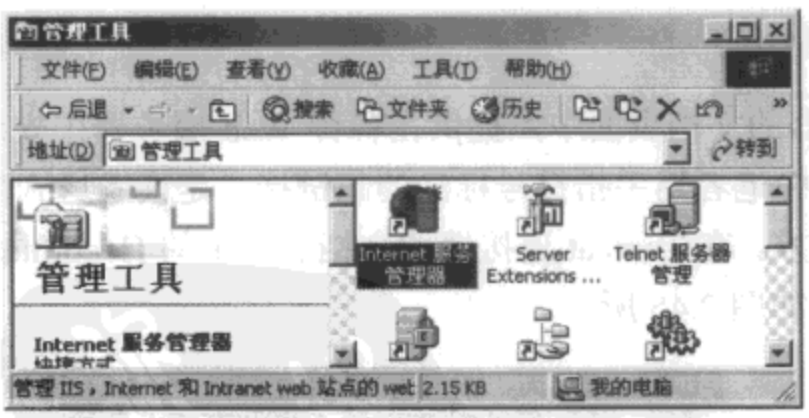


图 2.46

02 双击“Internet 服务管理器”项，打开 Internet 信息服务管理器，如图 2.47 所示。

03 选中左边目录树窗口中的 IIS 主机节点（图 2.47 中的“*it315-zxx”项），单击鼠标右键，在弹出的下拉菜单中选择“属性”菜单，打开 IIS 主机的属性窗口，如图 2.48 所示。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

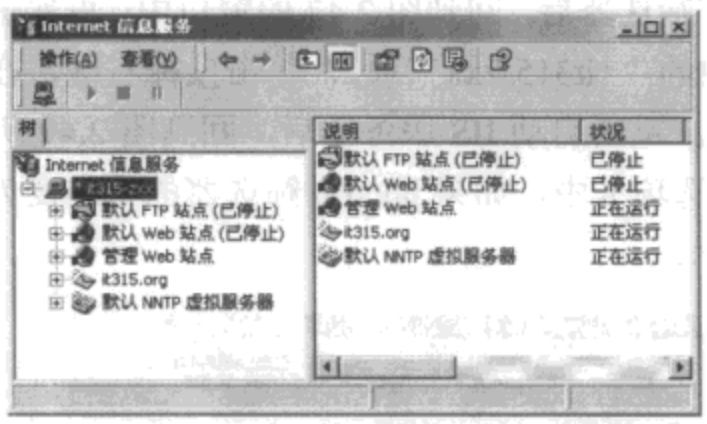


图 2.47

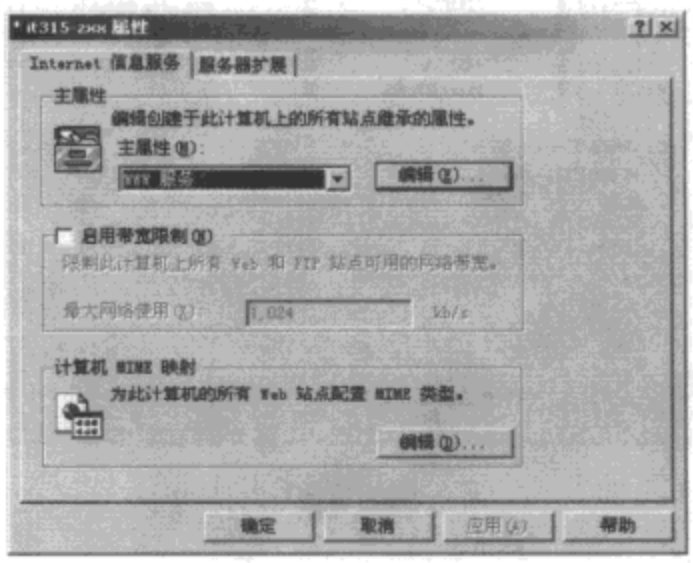


图 2.48

04 在“Internet 信息服务”选项卡的“主属性”栏中单击“WWW 服务”右边的“编辑”按钮，在弹出的“WWW 服务主属性”窗口中添加 ISAPI 筛选器，筛选器名称为“jakarta”，可执行文件为*<isapi_redirect.dll 文件路径>*，如图 2.49 所示。

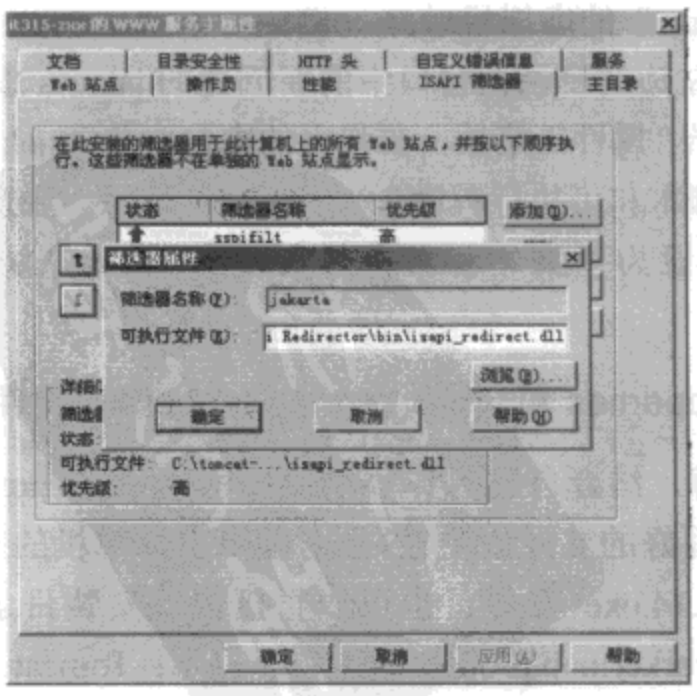


图 2.49

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

05 添加完 ISAPI 筛选器后，回到图 2.47 的窗口中，再次选中左边目录树窗口中的 IIS 主机节点（图 2.47 中的“*it315-zxx”项），单击鼠标右键，在弹出的下拉菜单中选择“重新启动 IIS...”菜单项。重新启动 IIS 服务器后，回到图 2.49 所示的“WWW 服务主属性”的“ISAPI 筛选器”选项卡中，如果 jakarta 筛选器的状态变为绿色向上的箭头，表示配置正常，如图 2.50 所示。

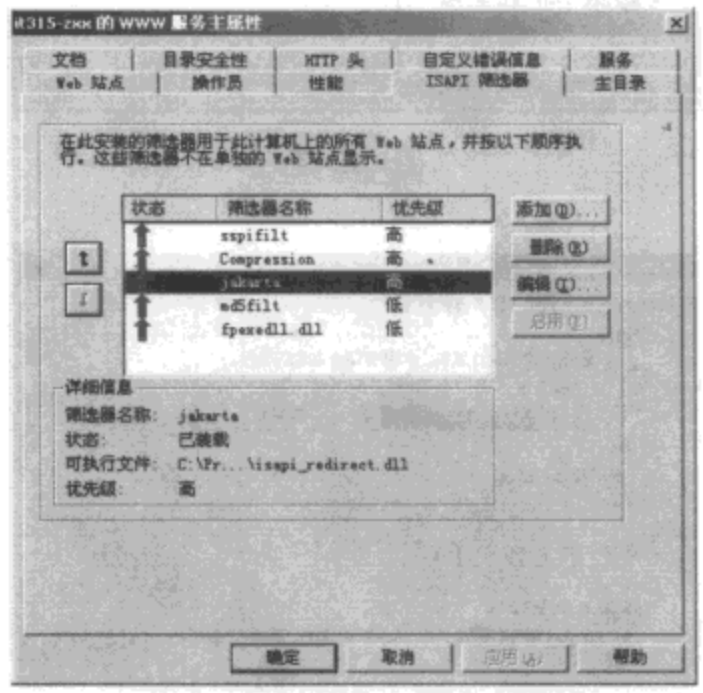


图 2.50

06 在安装完 ISAPI 筛选器后，还必须将 JK 插件所在的目录设置为某个站点下的一个具有执行许可权限的虚拟目录。选中图 2.47 左边的目录树窗口中的“默认 Web 站点”，单击鼠标右键，在弹出的下拉菜单中选择“新建”→“虚拟目录”，按照虚拟目录创建向导的提示完成创建虚拟目录的操作，其中，虚拟目录的别名为“jakarta”，本地路径为 JK 插件所在的目录。isapi_redirect-1.2.14.exe 程序在安装时，将自动把安装的 JK 插件所在的目录设置成名称为“jakarta”的虚拟目录。

07 选中图 2.47 左边的目录树窗口中的新创建的 jakarta 虚拟目录，单击鼠标右键，在弹出的下拉菜单中选择“属性”菜单，在弹出的窗口中修改 jakarta 虚拟目录的属性，将其执行许可权限设为“脚本和可执行程序”，如图 2.51 所示。注意：如果没有将 jakarta 虚拟目录的执行许可权限设为“脚本和可执行程序”，将无法访问 Tomcat 服务器中的 Servlet/JSP 组件。

2. 编写 workers.properties 和 uriworkermap.properties 文件

按照 2.6.4 节的讲解，结合自己的实际环境编写 workers.properties 和 uriworkermap.properties 文件，并将编写好的文件放在 IIS 服务器所在计算机的某个目录中。如果下载和安装了 isapi_redirect-1.2.14.exe 文件，也可以直接修改安装目录下的 conf 子目录中的 workers.properties 和 uriworkermap.properties 文件。提示：Tomcat 与 IIS 可以位于不同的计算机上，并且一个 IIS 服务器可以与多个不同计算机上的 Tomcat 服务器进行集成，JK 插

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

件及相关的配置文件都应该放置在 IIS 服务器所在的计算机上，而不用放置在任何 Tomcat 服务器上。

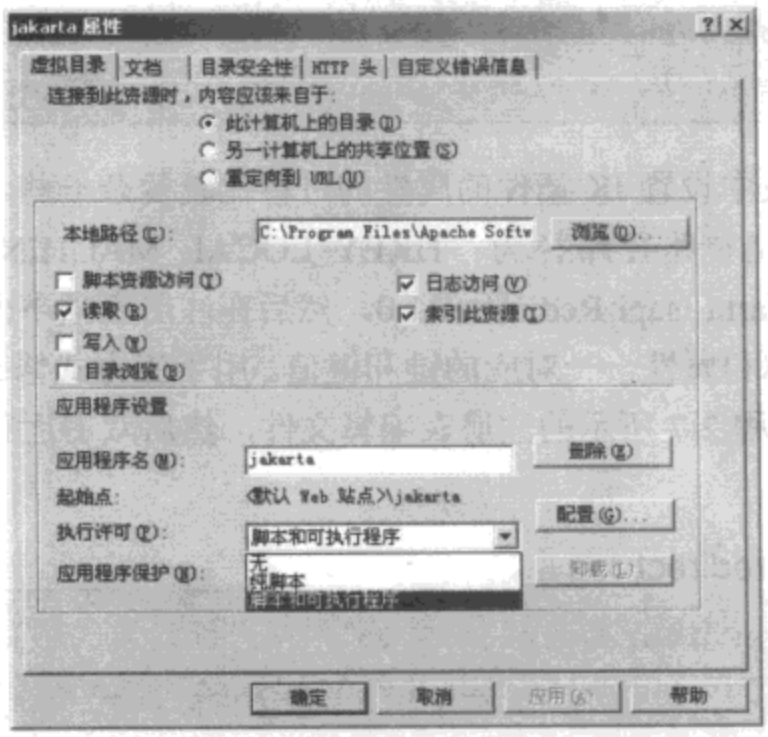


图 2.51

3. 设置 JK 插件的属性

JK 插件需要知道 workers.properties 和 uriworkermap.properties 文件的位置路径，此外，JK 自身有一些运行时的属性也需要设置，例如，记录日志的文件路径、记录日志的级别等信息。JK 插件的这些属性信息可以在一个属性文件中设置，也可以在 Windows 注册表中设置。JK 插件在初始化时优先读取属性文件中的设置信息，如果属性文件不存在，才从 Windows 注册表中读取设置信息。使用属性文件可以让同一台 IIS 服务器上安装多个 JK 插件，每个 JK 插件都具有自己独立的设置信息。使用注册表是为了向后兼容，属性文件简单方便，建议读者使用属性文件。

JK 插件的属性文件必须与 JK 插件放置在同一个目录中，并且文件名必须为 isapi_redirect.properties（即与 JK 插件同名，但扩展名为 properties）。例程 2-6 是一个 JK 插件的例子属性文件，它仅包含一些基本的设置信息，其中以“#”字符开头的行为注释行，读者应根据自己的实际情况，对例程 2-6 中的日志文件、workers.properties 文件和 uriworkermap.properties 文件的路径进行修改。

例程 2-6 isapi_redirect.properties

```
# JK插件在IIS服务器中虚拟路径，应与IIS中创建的虚拟目录名称保持一致
extension_uri=/jakarta/isapi_redirect.dll
# JK插件生成的日志文件的完整路径
log_file=C:\Jakarta_Isapi_Redirector\log\isapi_redirect.log
# 日志级别，其有效值为debug、info、warn、error或trace
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
log_level=debug
# workers.properties文件的完整路径
worker_file=C:\Jakarta_Isapi_Redirector\conf\workers.properties
# uriworkermap.properties文件的完整路径
worker_mount_file=C:\Jakarta_Isapi_Redirector\conf\uriworkermap.properties
```

在 Windows 注册表中设置 JK 插件的属性的过程比较复杂一些，它要求在注册表中新建一个注册表项。其完整路径路径为：HKEY_LOCAL_MACHINE\SOFTWARE\Apache Software Foundation\Jakarta Isapi Redirector\1.0，然后在注册表项下面增加与 isapi_redirect.properties 文件中的名称和属性一一对应的键和键值。用手工方式修改注册表比较麻烦，读者可以先创建一个如例程 2-7 所示的注册表编辑文件，然后双击运行这个文件，将其中的内容导入进注册表。

例程 2-7　isapi_redirect.reg

```
Windows Registry Editor Version 5.00
[HKEY_LOCAL_MACHINE\SOFTWARE\Apache Software Foundation\Jakarta Isapi
Redirector\1.0]
"extension_uri"="/jakarta/isapi_redirect.dll"
"log_file"="C:\\Jakarta_Isapi_Redirector\\logs\\isapi_redirect.log"
"log_level"="info"
"worker_file"="C:\\Jakarta_Isapi_Redirector\\conf\\workers.properties"
"worker_mount_file"="C:\\Jakarta_Isapi_Redirector\\conf\\uriworkermap.
properties"
```

将 isapi_redirect.reg 文件中的内容导入注册表后，单击 Windows 操作系统的“开始”→“运行”菜单，在弹出的对话框中输入“regedit”命令，打开注册表编辑器，可以看到新增的注册表项的内容如图 2.52 所示。

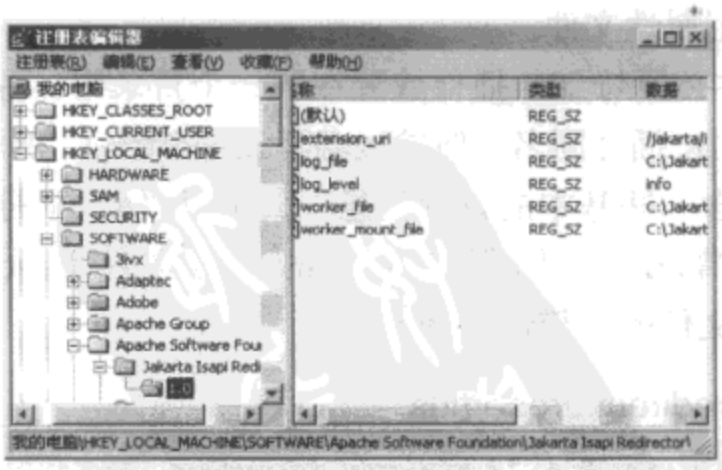


图 2.52

4. 测试配置是否成功

重新启动 Tomcat 服务器和 IIS 服务器，如果在 uriworkermap.properties 文件中将“/*.jsp”

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

映射给 Tomcat 处理，那么在浏览器地址栏输入：http://localhost/index.jsp，应该出现 Tomcat 的默认首页。否则，说明配置没有成功，请仔细检查整个配置过程的细节。

2.6.6 Tomcat 与 Windows 系统下的 Apache 服务器集成

Apache 服务器是 Apache 软件组织提供的开放源代码的专业 Web 服务器软件，它为网络管理员提供丰富多彩的 Web 管理功能，包括目录索引、目录别名、内容协商、可配置的 HTTP 错误报告、CGI 程序的 SetUID 执行、子进程资源管理、服务器端图像映射、重写 URL 和 URL 拼写检查等。Apache 服务器本身没有提供 Servlet/JSP 容器，在实际应用中，可以集成 Tomcat 与 Apache 来建立具有实用价值的商业化平台。

下面介绍在 Windows 2000 系统中如何集成 Tomcat 5.5.12 与 Apache 2.0.55 服务器。

1. 下载并安装 Apache 服务器

访问网址 http://httpd.apache.org/download.cgi，下载最新版本的 Apache 服务器软件，笔者使用的版本是 2.0.55。运行下载到的 apache_2.0.55-win32-x86-no_ssl.msi 安装文件，就启动了 Apache 服务器的安装过程，按照安装向导提示的默认设置进行安装即可，Apache 服务器的默认端口号是 80。如果安装成功，在 Windows 2000 Server 的服务管理控制台中可以看到 Apache 服务项，如图 2.53 所示。

安装成功后，在 Windows 程序菜单中将添加“Apache Http Server2.0.55”菜单组（Windows 系统的“开始”→“程序”→“Apache Http Server2.0.55”菜单组），这个菜单组中包含启动和停止 Apache 服务器的一些快捷菜单。启动 Apache 服务器后（应关闭其他占用 80 端口的网络程序，例如 IIS），可以在浏览器地址栏中输入 http://localhost 来测试 Apache 服务器是否成功启动。如果出现如图 2.54 所示的页面，则说明安装成功。

图 2.53

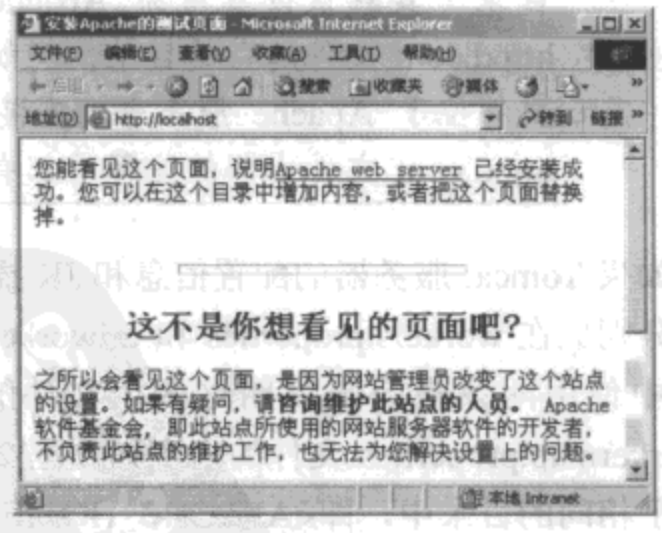


图 2.54

2. 下载并安装 JK 插件

Apache 服务器的插件一般都应放置在<APACHE_HOME>\modules 目录中，<APACHE_HOME>是 Apache 服务器的安装根目录。在 Apache 服务器下安装 JK 插件的过程比较简

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

单，只要将下载的供 Apache 服务器的 2.0.54 版本使用的 JK 插件文件 mod_jk1.2.14-apache-2.0.54.so 改名为 mod_jk.so 后，复制到<APACHE_HOME>\modules 目录中，然后在 Apache 的 httpd.conf 配置文件中增加一条 loadMoudle 指令即可。

3. 设置配置信息

Tomcat 与 Apache 服务器集成的所有设置信息，都可以在 Apache 的 httpd.conf 配置文件中直接设置，在 httpd.conf 文件末尾加入以下内容即可完成配置：

```
#加载JK插件模块
LoadModule jk_module modules/mod_jk.so
#配置Tomcat服务器的信息
JkWorkerProperty worker.list=worker1
JkWorkerProperty worker.worker1.port=8009
JkWorkerProperty worker.worker1.host=localhost
JkWorkerProperty worker.worker1.type=ajp13
#指定生成JK插件的日志文件的目录
JkLogFile logs/mod_jk.log
#指定JK插件的日志文件的级别
JkLogLevel debug
#设置JK插件的路径映射信息
JkMount /*.jsp worker1
JkMount /myweb/* worker1
```

注意 httpd.conf 文件位于<APACHE_HOME>\conf 目录中。即使对于 Windows 系统，该文件内的路径分隔符也必须使用正斜杠（/），而不能使用反斜杠（\）。对于 httpd.conf 文件中使用的相对目录，它们都是以 ServerRoot 指令所设置的目录（其默认设置值为 Apache 的安装根目录）为基准的，而不是相对于 httpd.conf 文件所在的目录，例如，上面的 logs/mod_jk.log 表示 JK 插件的日志文件位于 Apache 服务器的安装根目录下的 logs 子目录中。这些注意信息在 httpd.conf 文件内部的注释中都有明确说明。

如果 Tomcat 服务器的配置信息和 JK 插件的 URL 路径映射信息比较复杂，也可以将它们分别放在 workers.properties 和 uriworkermap.properties 文件中编写，然后在 httpd.conf 文件中使用两条指令来说明这两个文件的位置路径。假设 workers.properties 文件和 uriworkermap.properties 文件的内容如例程 2-8 和例程 2-9 所示，并且它们与 httpd.conf 文件位于相同的目录中，即<APACHE_HOME>\conf 目录。

例程 2-8 workers.properties

```
worker.list=worker1
worker.worker1.port=8009
worker.worker1.host=localhost
worker.worker1.type=ajp13
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

例程 2-9 uriworkermap.properties

```
/*.jsp=worker1
/myweb/*=worker1
```

那么，在 httpd.conf 文件末尾加入的内容可修改成如下形式：

```
LoadModule jk_module modules/mod_jk.so
JkWorkersFile conf/workers.properties
JkLogFile logs/mod_jk.log
JkLogLevel debug
JkMountFile conf/uriworkermap.properties
```

4. 测试配置是否成功

重新启动 Tomcat 服务器和 Apache 服务器，在浏览器地址栏中输入：http://localhost/index.jsp，应该出现 Tomcat 的默认首页。否则，说明配置没有成功，请仔细检查整个配置过程的细节。

2.6.7 与专业 Web 服务器集成的现实问题

在作者编写本章内容时，Tomcat 服务器的最高版本是 Tomcat5.5.12，随着 Tomcat 服务器版本的不断升级，JK 插件也在不断更新和演变，一点不用奇怪，当你隔周再来 Apache 站点查阅这方面的最新资料时，它们可能又做了相当大的改变。在六个月的时间内，人们可以公布、开发、运用、批评以及放弃一种技术。对于那些非常专用和其生命期限不是很长的技术，读者要现用现查、不能死记硬背。

由于一本好书的编写到出版的过程需要相当长的一段时间，至少都有一年的期限，而书在市面上的流通则又是若干年，也许等到读者看到书的内容时，书中讲解的专用技术已经被新的技术所取代，而读者却不知道，还在按照书中的步骤和内容进行学习与研究，这样反而会误导读者。笔者认为，对于这样的技术，书籍只能是提及一下其作用和目的何在，能解决什么问题，至于具体的实现细节，则有可能变化，建议读者最好去查阅相关文档资料，学习怎样去查阅这些资料。读者只需要知道有这样的一种技术，以及了解一些粗略的术语和学习方向后，在日后有具体应用需求时，再去 Internet 上和相关站点上查找有关于这方面的最新资料，再进行深入学习和研究，而不是按书上所讲的已经过时的方式来工作。

2.7 思考与实践

1. 请亲自从 Apache 的 jakarta 站点上下载 Tomcat 安装程序包的最新版本进行安装，并确保 Tomcat 可以正常启动运行。

2. 请用 Telnet 程序访问 www.it315.org 站点和 www.sina.com.cn 站点的首页，用 Alt+Print 键对 Telnet 程序窗口访问两个站点的过程进行抓图，再把抓取的图像复制到

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

Windows 自带的画图程序中，保存为不同的文件名后提交给 www.it315.org 网站。要求对每个网站的访问过程都抓取两幅图像，第一幅反映你输入的请求消息的完整内容，第二幅反映你得到了两个网站回复的结果。如果有什么疑问，请向 www.it315.org 网站咨询。

3．请使用 Tomcat 配置两个基于主机名的虚拟主机，并用浏览器对这两个站点进行访问测试，将你的配置文件提交给 www.it315.org 站点检查。

4．如果将 IIS 或 APACHE 与 Tomcat 成功集成后，且将“/*.jsp”映射给 Tomcat 处理，在浏览器地址栏中输入 http://localhost/index.jsp 时，Tomcat 的默认首页中原有的图片为什么显示不出了？

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

第 3 章　HTTP 协议详解

3.1　了解 HTTP

3.1.1　HTTP 简介

就像两个国家元首的会晤过程得遵守一定的外交礼节一样，浏览器与 Web 服务器之间的一问一答的交互过程也得遵循一定的规则，这个规则就是 HTTP 协议。HTTP 是 HyperText Transfer Protocol（超文本传输协议）的英文简写，它是 TCP/IP 协议集中的一个应用层协议，用于定义浏览器与 Web 服务器之间交换数据的过程以及数据本身的格式，大家平常通过浏览器访问 Internet 上的某一个网页的过程就是借助 HTTP 协议来完成的。HTTP 协议在不断地完善和扩展，因此有不同的版本，现在被广泛使用的是 HTTP1.1，相对 HTTP1.0 而言，HTTP1.1 的最大特点是支持持续连接。另外，HTTP-NG（Next Generation of HTTP）的建议已经提出，它将会提供更好的安全性、更快的速度，其改进要点为：模块化强、网络效率高、安全性更好、结构更简单。

对于从事 Web 站点规划、建设与管理的人员，特别是要从事网站服务器端程序的开发人员，必须对 HTTP 协议有一个比较深入的了解。对 HTTP 协议的透彻理解，并不需要具有多么高深的理论知识，只是一个实践问题。深入理解 HTTP 协议，对管理和维护复杂的 Web 站点、开发具有特殊用途的 Web 服务器程序具有直接影响。

本章将接着第 2 章的实验所保留的环境，再做一些实验来模拟浏览器与 Web 服务器的交互过程，并结合实验讲解 HTTP 协议的若干细节，以帮助读者深入了解 HTTP 协议。

3.1.2　HTTP 1.0 的会话方式

基于 HTTP 1.0 协议的客户机与服务器的信息交换过程包括四个步骤：建立连接；发送请求信息；回送响应信息；关闭连接，如图 3.1 所示。

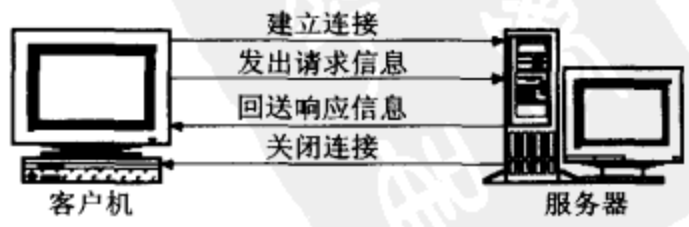


图 3.1

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

一个 Web 客户机与 Web 服务器建立连接后，给服务器发送一个请求。服务器接受客户机的请求，进行处理后再将响应结果回送给客户机，然后断开连接。浏览器与 Web 服务器的连接过程是短暂的，从浏览器发出请求到 Web 服务器返回结果后即告终结。浏览器与 Web 服务器的连接过程的持续时间与浏览器窗口保持打开的时间无关，它不是持续到浏览器程序关闭。

每次连接只处理一个请求和响应，即使是对同一个网站的每一个页面的访问，浏览器与 Web 服务器都要建立一次单独的连接。当浏览器获得一个页面访问的结果后，浏览器断开与服务器的连接，当浏览器要访问服务器上的另一个网页时，又会重新建立与服务器的连接。大家平常在一个网站上浏览信息时，浏览器其实与服务器建立了很多次的连接，每一次鼠标单击都是一次连接请求，这种通信过程总是通过浏览器发送请求来启动的，Web 服务器则是被动地对这些请求做出响应。

浏览器到 Web 服务器之间的所有通信都是完全独立分开的请求和响应对，当浏览器向服务器发出了一个访问请求，Web 服务器返回 HTML 文档给浏览器后，浏览器与 Web 服务器不再有任何关系。即使拔掉客户机的网线，先前获得的 HTML 文档在浏览器窗口中也会照样正常显示，重新插上网线，单击浏览器窗口中显示的一个超链接向 Web 服务器发送新的请求，Web 服务器接收到新的请求时，它根本就不知道客户机曾经被拔掉过网线。可见，HTTP 协议是无状态的，上一次请求的响应结果并不会影响下一次请求的响应过程，不管在什么样的情况下，只要浏览器发出的请求信息完全一样，Web 服务器的处理过程也就完全一样。

在浏览器与 Web 服务器之间还可以加入一个或多个中间代理服务器，Web 浏览器、代理服务器、Web 服务器三者之间的拓扑关系如图 3.2 所示。

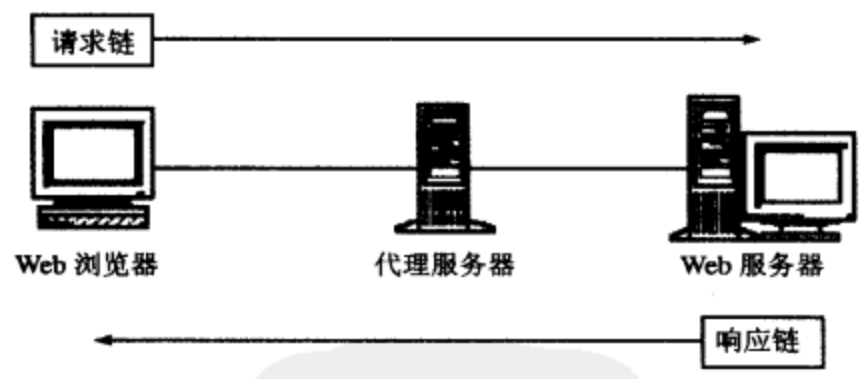


图 3.2

代理服务器接收浏览器的请求后，如果其本地缓存中没有浏览器请求的资源，代理服务器随即向 Web 服务器请求此资源后返回给浏览器，并将资源内容保存在本地缓存中，以后再有浏览器访问该资源时，代理服务器将直接从本地缓存中取出保存的副本进行响应。相对浏览器而言，代理服务器充当了服务器的角色。相对 Web 服务器而言，代理服务器就像普通浏览器一样充当了客户机的角色。在 Web 服务器前端使用多个代理服务器与浏览器进行通信，可以有效地减少 Web 服务器的访问负载，但应充分考虑网页的有效性，即代理服务器中缓存的页面在 Web 服务器端是最新的、没有再被更改的。HTTP 协议也定义了一

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

些与代理服务器有关的内容，譬如，浏览器可以要求代理服务器必须重新到 Web 服务器上获取请求的资源，而不是使用代理服务器本地缓存中的副本进行响应。

3.1.3 HTTP 1.1 与 HTTP 1.0 的比较

一个 Web 站点每天可能要接收到上百万的用户请求，为了提高系统的效率，HTTP 1.0 规定浏览器与服务器只保持短暂的连接，浏览器的每次请求都需要与服务器建立一个 TCP 连接，服务器完成请求处理后立即断开 TCP 连接，服务器不跟踪每个客户机，也不记录过去的请求。但是，这也造成了一些性能上的缺陷，例如，一个包含许多图像的网页文件中并没有包含真正的图像数据内容，而只是指明了这些图像的 URL 地址，当浏览器访问这个网页文件时，浏览器首先要发出针对该网页文件的请求，当浏览器解析 Web 服务器返回的该网页文档中的 HTML 内容时，发现其中的<img>图像标签后，浏览器将根据<img>标签中的 src 属性所指定的 URL 地址再次向服务器发出下载图像数据的请求，如图 3.3 所示。

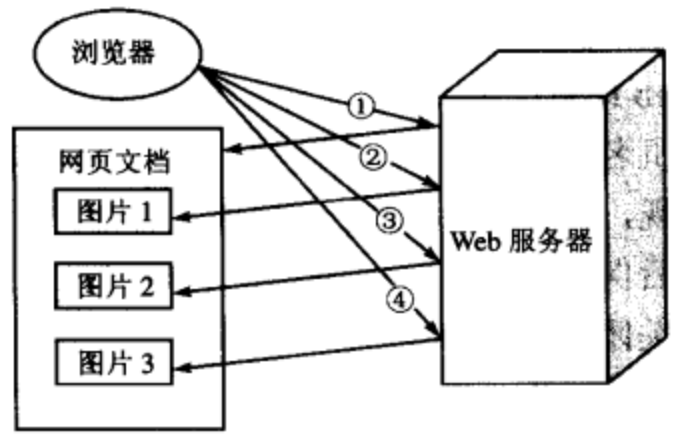


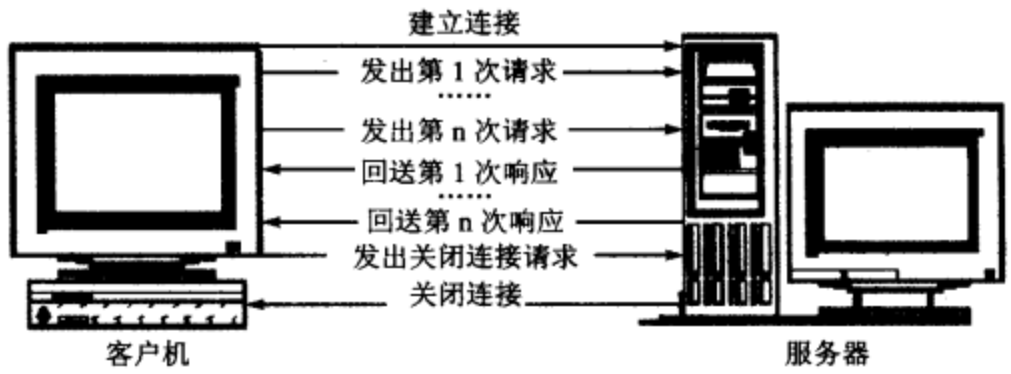
图 3.3

显然，浏览器访问一个包含有许多图像的网页文件的整个过程需要多次请求和响应。在 HTTP 1.0 中，每次请求和响应都需要建立一个单独的连接，每次连接只是传输一个文档和图像，上一次和下一次请求完全分离，这就导致浏览器要获得一个包含许多图像的网页文件时需要与 Web 服务器建立多次连接。即使图像文件都很小，但是客户端和服务器端每次建立和关闭连接也是一个相对比较费时的过程，并且会严重影响客户机和服务器的性能。当一个网页文件中包含 Applet、JavaScript 文件、CSS 文件等内容时，也会出现类似上述的情况。

为了克服 HTTP 1.0 的这个缺陷，HTTP 1.1 支持持久连接，在一个 TCP 连接上可以传送多个 HTTP 请求和响应，减少了建立和关闭连接的消耗和延迟。一个包含许多图像的网页文件的多个请求和应答可以在一个连接中传输，但每个单独的网页文件的请求和应答仍然需要使用各自的连接。HTTP 1.1 还允许客户端不用等待上一次请求结果返回，就可以发出下一次请求，但服务器端必须按照接收到客户端请求的先后顺序依次回送响应结果，以保证客户端能够区分出每次请求的响应内容，这样也显著地减少了整个下载过程所需要的时间，这个功能称之为 Pipeline（管道线）。基于 HTTP 1.1 协议的客户机与服务器的信息交换过程，如图 3.4 所示。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享



可见，HTTP 1.1 在继承了 HTTP 1.0 优点的基础上，也解决了 HTTP 1.0 的性能问题。不仅如此，HTTP 1.1 还通过增加更多的请求头和响应头来改进和扩充 HTTP 1.0 的功能。例如，由于 HTTP 1.0 不支持 Host 请求头字段，浏览器无法使用主机名来明确表示要访问服务器上的哪个 Web 站点，这样就无法使用 Web 服务器在同一个 IP 地址和端口号上配置多个虚拟 Web 站点。在 HTTP 1.1 中增加 Host 请求头字段后，浏览器可以使用主机名来明确表示要访问服务器上的哪个 Web 站点，这才实现了在一台 Web 服务器上可以在同一个 IP 地址和端口号上使用不同的主机名来创建多个虚拟 Web 站点。HTTP 1.1 的持续连接，也需要增加新的请求头来帮助实现，例如，Connection 请求头的值为 Keep-Alive 时，客户端通知服务器返回本次请求结果后保持连接；Connection 请求头的值为 close 时，客户端通知服务器返回本次请求结果后关闭连接。HTTP 1.1 还提供了与身份认证、状态管理和 Cache 缓存等机制相关的请求头和响应头。

3.1.4 HTTP 消息的格式

当用户在浏览器地址栏中输入某个 URL 地址，或单击网页上一个超链接，或提交网页上的 Form 表单后，浏览器将生成请求消息发送给服务器。服务器收到请求后，将生成响应消息回送给浏览器。浏览器发出的请求信息和 Web 服务器回送的响应信息都叫 HTTP 消息，HTTP 消息是有一定严格规定的格式。浏览器提交给 Web 服务器的 FORM 表单内容和从 Web 服务器上获取的网页内容仅仅是 HTTP 消息中的一部分数据，浏览器与 Web 服务器传递的信息中还包含一般用户所看不到的一些其他“隐藏”信息。

一个完整的请求消息包括：一个请求行、若干消息头，以及实体内容，其中的一些消息头和实体内容都是可选的，消息头和实体内容之间要用空行隔开（即只有回车和换行两个字符的一个单独行）。下面是一个 HTTP 请求消息的内容：

```
GET /books/java.html HTTP/1.1
Accept: */*
Accept-Language: en-us
Connection: Keep-Alive
Host: localhost
Referer: http://localhost/links.asp
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)
Accept-Encoding: gzip, deflate
```

上面的请求消息中的第一行为请求行，请求行后面的内容都属于消息头部分。虽然该请求消息中没有实体内容（正文），但紧接着消息头部分之后的一个空行（只有回车和换行符的行）必不可少，这个空行用于表示消息头部分已经结束。当用户在浏览器地址栏中直接输入某个 URL 地址或单击网页上一个超链接时，浏览器将使用 GET 方式发送类似上面的不包含实体内容的请求消息；只有使用 POST、PUT 和 DELETE 方式的请求消息中才可以包含实体内容，譬如，当浏览器使用 POST 请求方式提交网页中的 FORM 表单时，FORM 中的各个元素信息将作为请求消息的实体内容。

一个完整的响应消息包括一个状态行、若干消息头，以及实体内容。与请求消息一样，响应消息中的一些消息头和实体内容也都是可选的，消息头和实体内容之间也要用空行隔开。下面是一个 HTTP 响应消息的内容：

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Thu, 13 Jul 2000 05:46:53 GMT
Content-Length: 2291
Content-Type: text/html
Set-Cookie: ASPSESSIONIDQQGGGNCG=LKLDFFKCINFLDMFHCBCBMFLJ; path=/
Cache-control: private

<HTML>
<BODY>
......
```

上面的响应消息中的第一行为状态行，紧跟状态行后面的内容是消息头部分，接着是一个空行，然后是实体内容（这里只显示了头两行）。在通常情况下，响应消息中都包含实体内容，响应消息的实体内容就是网页文档的内容，也就是在浏览器中使用查看源文件的方式所看到的内容。对于 HTTP 1.1 来说，如果 HTTP 消息中包括实体内容，且没有采用 chunked 传输编码方式，那么消息头部分必须包含实体内容长度的字段，否则，客户端和服务程序就无法知道实体内容何时结束。

在 HTTP 协议中，还可以使用只包含请求行的请求消息，这种只有请求行的请求消息称为简单请求消息。简单请求消息只能用于 GET 方式，且请求行中不用指定 HTTP 版本号。对于简单请求消息，服务器返回的响应消息只包含实体内容，这种只包含实体内容的响应消息称为简单响应消息。对于能接受完全响应消息的浏览器来说，它不应向 Web 服务器发送简单请求消息，所以，简单请求消息在实际应用中并不常见。

3.1.5 HTTP 消息头

HTTP 请求和 HTTP 响应都使用消息头来描述 HTTP 消息本身的信息，这种描述信息

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

的信息称为元信息。使用消息头，可以实现 HTTP 客户机与服务器之间的条件请求和应答。消息头相当于服务器和浏览器之间的一些暗号指令，服务器可以使用消息头来通知浏览器采取一些特殊动作，例如，隔多长时间刷新网页内容，用哪种字符集显示网页文档内容，等等；浏览器也可以使用消息头来向服务器传递一些特殊信息，例如，当前请求是通过哪个网页上的超链接引导进来的，浏览器的国家语言版本信息，等等。

浏览器除了显示网页文件内容外，还要按照消息头所表示的指令采取行动。在目前版本的 IE 浏览器中只能查看到实体内容，而看不到状态行和消息头部分，所以，一般用户是不知道浏览器与服务器之间的这些数据传送细节的。

消息头部分由一系列的行组成，每行都包含一个头字段名称，然后依次是冒号、空格、值、回车和换行符。消息头字段名是不区分大小写的，但习惯上将每个单词的第一个字母大写。整个消息头部分中的各行消息头可按任何顺序排列。

按照其作用分类，消息头又可以分为通用信息头、请求头、响应头、实体头四类。

- **通用信息头** 既能用于请求消息中，也能用于响应消息中，但与被传输的实体内容没有关系的常用消息头，如 Date、Pragma。
- **请求头** 用于在请求消息中向服务器传递附加信息，主要包括客户机可以接受的数据类型、压缩方法、语言，以及客户计算机上保留的 Cookie 信息和发出该请求的超链接源地址等。
- **响应头** 用于在响应消息中向客户端传递附加信息，包括服务程序的名称、要求客户端进行认证的方式、请求的资源已移动到的新地址等。
- **实体头** 用做实体内容的元信息，描述了实体内容的属性，包括实体信息的类型、长度、压缩方法、最后一次修改的时间和数据的有效期等。

请求消息头中可包括通用信息头、请求头和实体头；响应消息头中可包括通用信息头、响应头和实体头。某些头字段既能用于请求头也能用于响应头，而另一些头字段只能用于其中之一。

许多请求头字段都允许客户端在值部分指定多个可接受的选项，有时甚至可以对这些选项的首选项进行排名。多个项之间以逗号分隔，例如，客户端可以发送包含“Accept-Encoding: gzip, compress”的请求头，表示可以接受 gzip 和 compress 这两种压缩类型。如果服务器的响应实体内容（正文）使用 gzip 进行了压缩，其响应头中将包含“Content-Encoding: gzip”。有些头字段可以出现多次，例如，响应消息中可以包含有多个“Warning”头字段。

3.2 请求行与状态行

3.2.1 请求行与状态行

1. 请求行

请求消息的请求行中包括三个部分：请求方式、资源路径，以及所使用的 HTTP 协议

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

版本，各个部分用空格分隔，语法格式为：

请求方式 资源路径 HTTP 版本号<CRLF>

其中的<CRLF>表示回车和换行这两个字符的组合。HTTP 的请求方式包括 POST、HEAD、OPTIONS、DELETE、TRACE 和 PUT，名称全部大写。每种请求方式规定了对服务器上的资源所要执行的动作，通常的应用只使用其中的 GET 和 POST 两种方式。3.2.2 节将对 POST 和 GET 方式进行详细讲解，关于其他几种方式的作用，请参阅 Internet Engineering Task Force (IETF) 文档 RFC 2616。Internet 上的许多 Web 站点都有正式的 RFC 文档，使用 www.google.com 搜索“RFC 2616”关键字，能够找到该文档的许多站点列表。直接访问 http://www.w3.org/Protocols，可以看到更多的与 WWW 有关的各种 RFC 文档。

2. 状态行

响应消息的状态行中包括三个部分：HTTP 协议的版本号、一个表示成功或错误的整数代码（状态码）和对状态码进行描述的文本信息，各部分用空格分隔，语法格式为：

HTTP 版本号 状态码 原因叙述<CRLF>

借助 Telnet 程序直观地体验请求行和状态行

01 在本地 Tomcat 服务器的默认站点根目录下创建一个 test.html 文件，在这个文件中随便写上一些便于自己识别的内容。启动 Tomcat Web 服务器程序后，使用 telnet 命令连接到 Tomcat Web 服务器监听的 8080 端口，即在本地计算机的命令行窗口中执行如下命令：

```
telnet localhost 8080
```

在成功连接后启动的 Telnet 程序窗口中，输入如下一行内容：

```
GET /test.html HTTP/1.0
```

注意 不能弄错大小写，也不允许写错一个字符后再进行更正（具体原因请参看笔者的《Java 就业培训教程》一书中的网络编程案例分析）。在这行文本后面再次用回车键输入一个空行，也就是接着又输入了回车（\r）和换行（\n）这两个字符，Telnet 程序窗口中将显示出 Web 服务器返回的响应信息，如图 3.5 所示。

在 Telnet 窗口中输入的那一行内容就是请求行，Web 服务器返回的响应信息中的第一行就是响应状态行。在上面的请求行中，请求的方式为 GET，资源路径为/test.html，请求信息所用的 HTTP 协议版本号为 1.0。在上面响应状态行中，Web 服务器返回的信息所用的 HTTP 协议版本号为 1.1，状态码为 200，状态码的简要描述信息为 OK。在命令行窗口

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

中显示的换行和空行，实际上就是回车（\r）和换行（\n）这两个字符的显示效果。

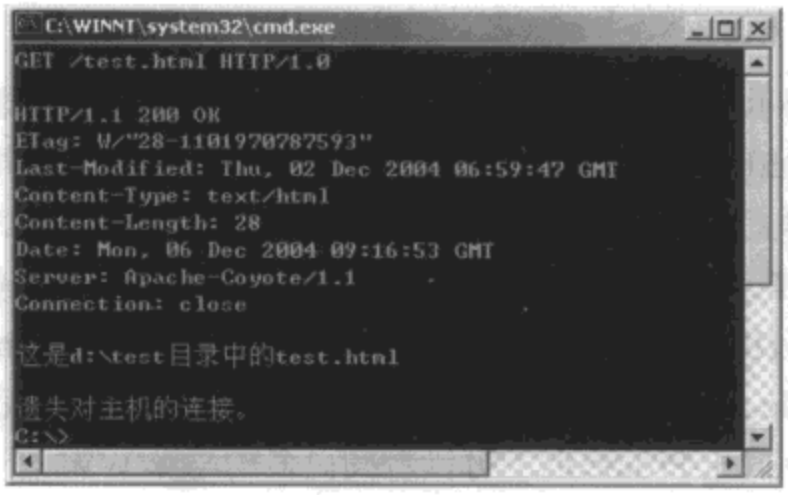


图 3.5

02 在本地计算机上使用 Telnet 命令再次连接到 Tomcat Web 服务器监听的 8080 端口，在成功连接后启动的 Telnet 程序窗口中，输入如下一行内容：

```
GET /test.html HTTP/1.1
```

这次将请求信息所用的 HTTP 协议版本号更改为了 1.1，在这行文本后面再次用回车键输入一个空行后，Telnet 程序窗口中显示出 Web 服务器返回的响应信息，如图 3.6 所示。

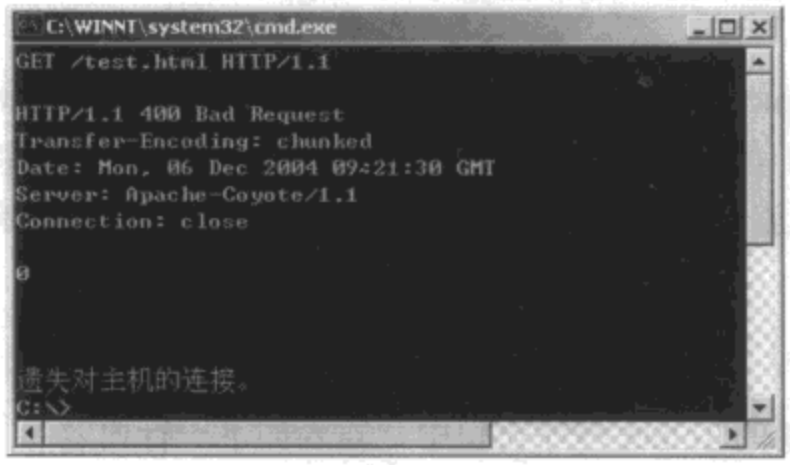


图 3.6

这次响应状态行中的状态码变成了 400，状态码的简要描述信息为 Bad Request，即请求消息的格式是错误的，这是因为 HTTP 1.1 要求请求消息中至少应该有一个 Host 头信息。可见，HTTP 使用不同的响应状态码来表示服务器对请求的各种不同处理结果和状态。在图 3.6 中看到实体内容为字符 0，这是由响应头中包含了值为 chunked 的 Transfer-Encoding 头字段所造成的。

03 用 Telnet 程序重新连接到 Tomcat Web 服务器，在 Telnet 程序窗口中，输入如下一行内容：

```
GET /test.html
```

上面的内容在请求行中没有指定 HTTP 版本号，它就代表一个简单的请求消息，因此 Web

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

服务器返回的结果将是一个简单的响应信息，即只有实体内容部分，如图 3.7 所示。

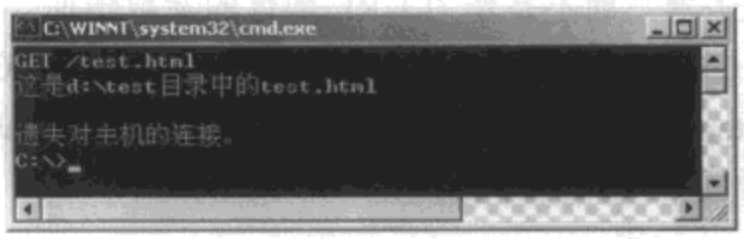


图 3.7

3.2.2 使用 GET 和 POST 方式传递参数

在 URL 地址后面可以附加一些参数，每个参数由参数名和参数值组成，参数名与参数值之间用等号（=）分隔，各个参数之间用&分隔，URL 地址与整个参数部分之间用问号（?）分隔，如下所示：

```
http://www.it315.org/servlet/ParamsServlet?param1=abc&param2=xyz
```

在一个 URL 地址后面附加参数有很大的用处，这使得 URL 地址所指向的 Web 资源可以根据参数值的不同返回不同的结果内容，例如，我们可以通过访问某一个 URL 地址来分别读取 BBS 中的各个帖子，那就需要将帖子的编号作为参数分别附加到这个 URL 地址后面。

1. GET 方式

当用户在浏览器地址栏中直接输入某个 URL 地址或单击网页上一个超链接时，浏览器将使用 GET 方式发送请求。如果将网页上的<form>表单的 method 属性设置为“GET”或没有设置 method 属性（<form>表单的默认值为“GET”），当用户提交表单时，浏览器也将使用 GET 方式发送请求。

如果浏览器请求的 URL 中有参数部分，在浏览器生成的请求消息中，参数部分将附加在请求行中的资源路径后面。例如，对于如下的 URL 地址：

```
http://www.it315.org/servlet/ParamsServlet?param1=abc&param2=xyz
```

浏览器发送给 www.it315.org 服务器的请求消息如下所示：

```
GET /servlet/ParamsServlet?param1=abc&param2=xyz HTTP/1.1
Host: www.it315.org
```

当使用 GET 方式提交表单内容时，浏览器将各个表单字段元素及其数据按照 URL 参数的格式附加在请求行中的资源路径后面。使用 GET 方式传送的数据量是有限制的，一般限制在 1KB 以下。

2. POST 方式

如果网页上的<form>元素的 method 属性设置为“POST”，当用户提交表单请求时，

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

浏览器将使用 POST 方式提交表单内容，并把各表单字段元素及其数据作为 HTTP 消息的实体内容发送给 Web 服务器，而不是作为 URL 地址的参数传递，因此，使用 POST 方式传送的数据量要比使用 GET 方式传送的数据量大得多。在使用 POST 方式向服务器传递数据时，还必须将 Content-Type 消息头设置为“application/x-www-form-urlencoded”，将 Content-Length 消息头设置为实体内容的长度，如下所示：

```
POST /servlet/ParamsServlet HTTP/1.1
Host:
Content-Type: application/x-www-form-urlencoded
Content-Length: 28

param1=abc&param2=xyz
```

对于这种 POST 方式传递的请求消息，Web 服务器端的程序可以采用获取 URL 后面的参数一样的方式来获取各表单字段元素的数据。

使用 GET 与 POST 方式传递参数

01 Tomcat 提供了一个获取浏览器使用 POST 或 GET 方式传递的参数的例子程序，直接在浏览器地址栏中输入 http://localhost:8080/examples/servlet/RequestParamExample，就可以运行这个例子程序，如图 3.8 所示。

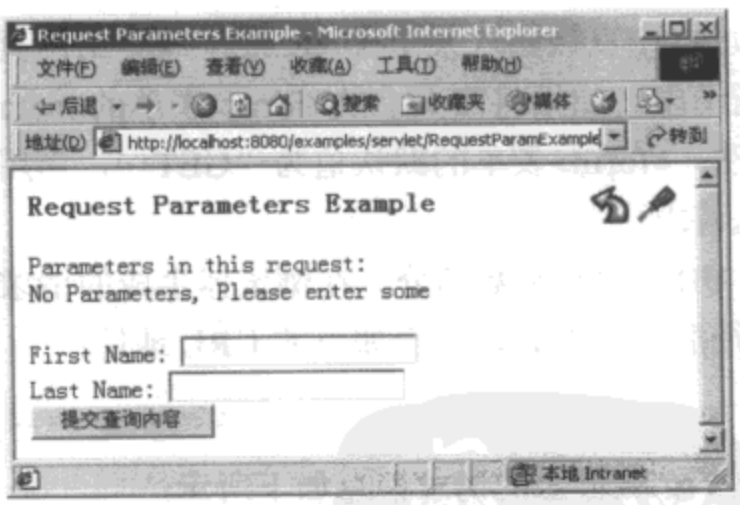


图 3.8

要记住 Tomcat 的每个例子程序的 URL 地址非常困难，并且输入这么长的 URL 信息也容易出错，所以，要访问 Tomcat 的各个例子程序的最好方式是访问 Tomcat Web 服务器的默认站点的首页（如果修改过 Tomcat Web 服务器的默认站点的根目录设置，应修改 Server.xml 配置文件，将其恢复为安装时的初始根目录），在其中找到关于 Servlet 例子的入口超链接，如图 3.9 所示。

尽管不同版本的 Tomcat 服务器的首页布局和组织结构不尽相同，但读者应该能够找

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

到关于 Servlet 例子的入口超链接。单击这个超链接，进入 Tomcat 提供的 Servlet 例子页面，找到执行 RequestParamExample 这个 Servlet 程序的超链接，如图 3.10 所示。单击这个超链接，即可进入如图 3.8 所示的页面。

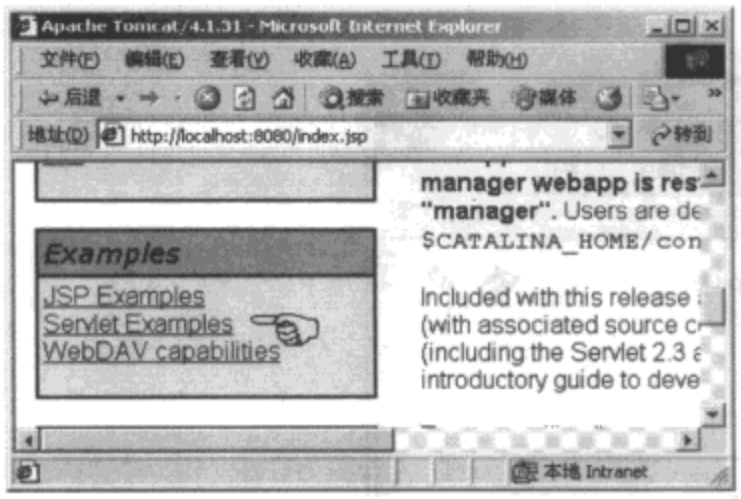


图 3.9

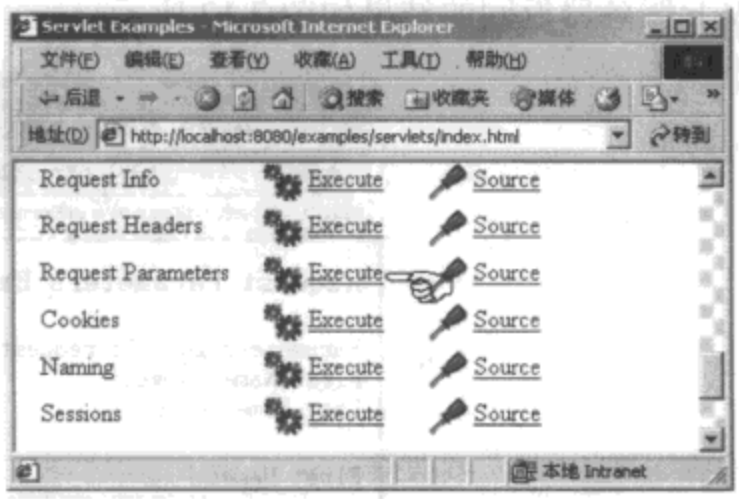


图 3.10

02 在图 3.8 所示页面的“First Name”文本输入框中输入“zhang”，在“Last Name”文本输入框中输入“san”，单击“提交查询内容”按钮，这时候浏览器内部将使用 POST 方式向 Web 服务器传递参数，Web 服务器返回的结果如图 3.11 所示。

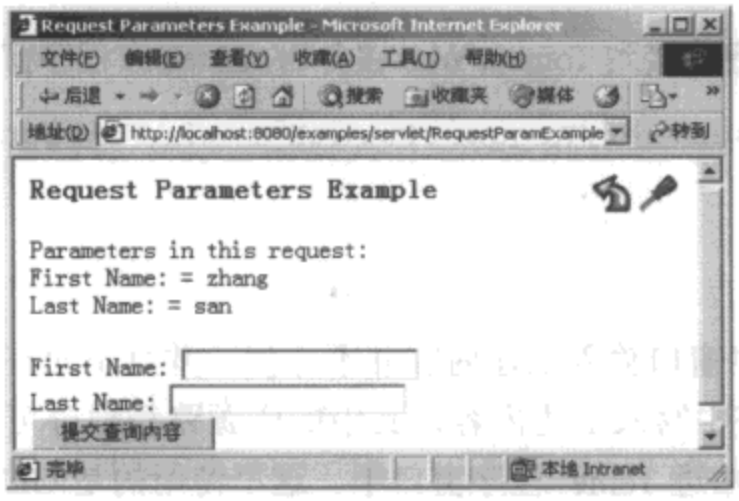


图 3.11

Web 服务器返回的页面中显示出了上次在两个文本输入框中填写的内容，这表明浏览器已将 FORM 表单中填写的数据提交给了 Web 服务器，Web 服务器程序也获得了这些数据，并在响应消息中将这些数据回送给用户查看。在图 3.11 中的两个文本输入框内填写新的内容，单击“提交查询内容”按钮后，Web 服务器返回的页面中又将显示出新填写的内容。

03 在图 3.8 或图 3.11 中，单击浏览器的“查看”→“源文件”菜单，可以看到两个文本输入框的名称（由其 name 属性指定）分别为 firstname 和 lastname。在图 3.8 或图 3.11 所示的浏览器地址栏中的 URL 地址后面增加?firstname=wang&lastname=wu，这样将直接在 URL 中向 Web 服务器传递参数，增加参数后的 URL 的完整内容为：

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
http://localhost:8080/examples/servlet/RequestParamExample?firstname=wa
ng&lastname=wu
```

输入完上面的URL后按回车键，此时浏览器将使用GET方式向Web服务器传递参数，Web服务器返回的结果如图3.12所示。

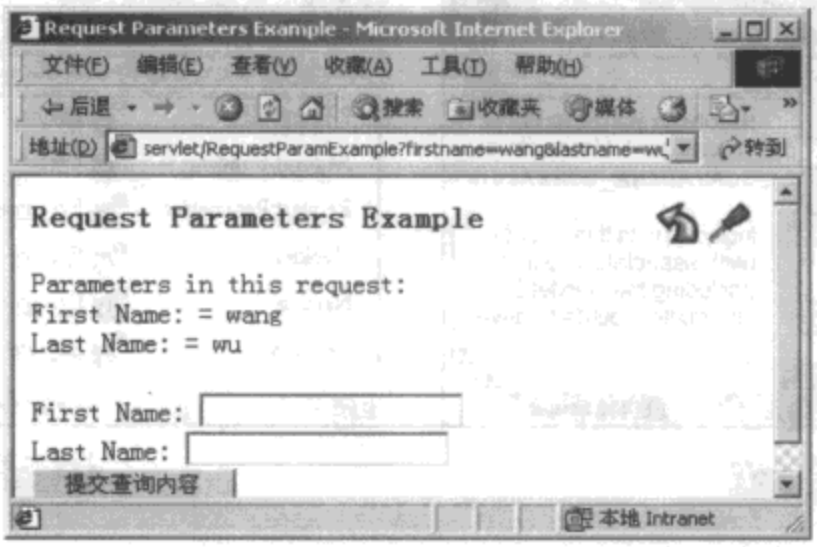


图 3.12

从图3.12中可以看到，这个Servlet例子程序也可以获得浏览器使用GET方式在URL中传递的参数。

04 用Telnet程序连接到Tomcat Web服务器，在telnet程序窗口中输入如下几行内容：

```
GET /examples/servlet/RequestParamExample?firstname=wang&lastname=
wu HTTP/1.1
Host:
<空行>
```

直接在Telnet程序窗口输入上述内容并不方便，并且不允许出错后修改，可以先在Windows的记事本程序中写入上述内容，其中的资源路径部分可以从浏览器地址栏中复制，然后将这些内容粘贴进Telnet程序窗口中。这里使用的是GET方式向Web服务器传递参数，Telnet程序窗口中显示出了Web服务器返回的内容，如图3.13所示，其中的实体内容部分就是图3.12中显示的网页的源文件内容。

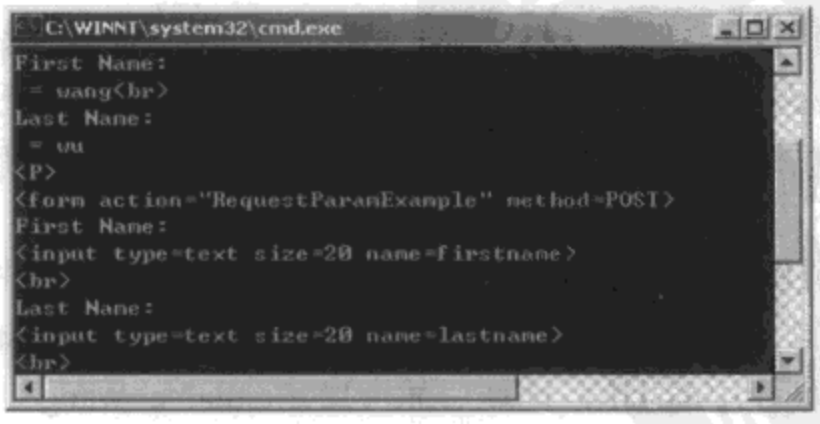


图 3.13

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

05 用 Telnet 程序重新连接到 Tomcat Web 服务器，按照上面的方式在 Telnet 程序窗口中输入如下几行内容：

```
POST /examples/servlet/RequestParamExample HTTP/1.1
Host:
Content-Type: application/x-www-form-urlencoded
Content-Length: 28

firstname=zhang&lastname=san
```

输入上述内容后，Telnet 程序窗口中将显示出 Web 服务器返回的内容，其中的实体内容部分就是图 3.11 中显示的网页的源文件内容。注意：上面设置的 Content-length 头字段的值一定要和实体内容的长度一致，读者可以将上面的 Content-length 头字段的值分别设置为 26 和 30，看看实验效果怎样？想想为什么？

3.2.3 响应状态码

响应状态码用于表示服务器对请求的各种不同处理结果和状态，它是一个三位的十进制数。响应状态码可归为 5 种类别，使用最高位为 1 到 5 来进行分类，如下所示：

（1）100～199

表示成功接收请求，要求客户端继续提交下一次请求才能完成整个处理过程。

（2）200～299

表示成功接收请求并已完成整个处理过程。

（3）300～399

为完成请求，客户端需进一步细化请求。例如，请求的资源已经移动一个新地址。

（4）400～499

客户端的请求有错误。

（5）500～599

服务器端出现错误。

HTTP 1.1 比 HTTP 1.0 增加了一些对应 HTTP 1.1 新特性的状态码，表 3.1 到表 3.5 分别列出了 HTTP 1.1 的 5 种类别的状态码及其含义。

表 3.1 （100～199）

编　号	说　明
100（继续）	如果客户机发送一个值等于 100-continue 的 Expect 请求头，询问是否可以在后面的请求中发送一个附加文档。在此情形下，服务器使用 100 响应状态码告诉客户机可以继续，或者用 417（Expectation Failed）告诉浏览器它不能接收下次请求中附加的文档
101（切换协议）	如果客户机发送的请求要求使用另外一种协议与服务器进行对话，服务器发送 101 响应状态码表示自己将遵从客户机的请求转换到另外一种协议

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

表 3.2 （200～299）

编　号	说　明
200（正常）	表示一切正常，返回的是正常请求结果
201（已创建）	服务器已经创建了文档，Location 头给出了这个文档的 URL
202（已接受）	告诉客户机请求已被接受，但处理目前尚未完成
203（非权威性的信息）	文档已经正常地返回，但一些应答头可能不正确，因为使用的是文档的缓存信息，而不是最原始的（在计算机术语中，最原始的常称为权威性的）信息
204（无内容）	规定浏览器应该继续显示前面缓存的文档，因为新的文档得不到。如果用户刷新某个页面时，并且服务器能够确定客户端当前显示的页面已经是最新的，这种功能就很有用，不用继续向客户端传送文档内容，节省了网络流量和服务器处理时间
205（重置内容）	表示没有新的文档，浏览器仍应显示原来的文档，但要重置文档的内容，例如，清除表单字段中已经输入的内容
206（部分内容）	客户端发送了一个带有 Range 头（要求服务器只返回文档中的部分内容）的 GET 请求，服务器按要求完成了这个请求

表 3.3 （300～399）

编　号	说　明
300（多项选择）	客户端请求的文档可以在多个位置找到，这些位置已经在返回的文档内列出。如果服务器要提出优先选择，则应该在 Location 响应头中指明
301（永久移动）	指出被请求的文档已被移动到别处，此文档的新的 URL 在 Location 响应头中给出，浏览器以后对该文档的访问应该自动使用新的 URL
302（找到）	类似于 301，但 Location 头中给出的 URL 应该被作为一个临时的、而非永久的地址。实际上，多数浏览器把 301 和 302 等同对待。严格地说，只有当原来的请求是 GET 时浏览器才会自动重定向
303（参见其他）	类似于 301 和 302，但如果最初的请求方式为 POST，客户端应该接着用 GET 方式请求 Location 头中给出的新文档
304（未修改）	如果客户机有一个缓存文档，它可以通过提供一个 If-Modified-Since 头来执行一个条件请求，表示只有请求的文档自某个指定的时间以来发生过更改，客户机才要下载此文档，If-Modified-Since 头指出这个时间。状态码 304（SC_NOT_MODIFIED）表示客户机缓存的版本是最新的，客户机应该继续使用它。否则，服务器将用正常的（200）状态代码返回所请求的文档
305（使用代理服务器）	客户机请求的文档应该通过 Location 头所指明的代理服务器提取
307（临时重定向）	类似于 302 状态码。按严格规定，如果浏览器使用 POST 方式发出请求，只有响应状态码为 303 时才能重定向，但实际上许多浏览器对 302 状态码也按 303 状态码来处理。由于这个原因，HTTP 1.1 新增了 307，以便更加清楚地区分几个状态代码：当出现 303 应答时，浏览器可以重定向 GET 和 POST 请求；如果是 307 应答，则浏览器只能重定向 GET 请求。浏览器接收到 307 状态码后，不自动重定向到新的 URL 地址，在含有 307 状态码的消息中，应包含一个简单的文本实体内容，文本中有一条指向新 URL 地址的超链接，以便用户通过手工单击这个超链接进入新的 URL 地址

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

表 3.4 （400～499）

编　号	说　明
400（无效请求）	指出客户机请求中有不正确的语法格式
401（未经授权）	当客户机试图访问一个受口令和密码保护的页面，且在请求中没有使用 Authorization 请求头传递用户信息时，服务器返回 401 状态码，同时结合使用一个 WWW-Authenticate 响应头来提示客户机应重新发出一个带有 Authorization 头的请求消息
402（需要付款）	目前未使用，为以后更高的 HTTP 版本而保留
403（禁止）	服务器理解客户端的请求，但拒绝处理它。通常由于服务器上文件或目录的权限设置导致
404（找不到）	服务器上不存在客户机所请求的资源，这是一个常见的状态码
405（不允许此请求方式）	请求行中的请求方式对指定的资源不适用。例如，有的资源只能用 GET 方式访问，当使用 POST 方式访问这些资源时，服务器将返回 405 状态码
406（不能接受）	指定的资源已经找到，但它的 MIME 类型和客户端在 Accept 头中所指定的不兼容
407（需要代理服务器身份验证）	类似于 401，是由代理服务器对客户端发出的响应，表示客户端必须先经过代理服务器的授权
408（请求超时）	在服务器许可的等待时间内，客户端一直没有发出任何请求
409（冲突）	由于请求和资源的当前状态相冲突，因此请求不能成功。这个状态码通常和 PUT 请求有关，例如，要用上传的文件覆盖一个正在服务端打开的文件
410（离开）	请求的文档已经不再可用，而且服务器不知道应该重定向到哪一个地址。410 表示文档永久地离开了指定的位置，而 404 表示由于未知的原因文档不可用
411（需要长度）	客户端发送请求中应包含一个指定实体内容长度的 Content-Length 头
412（未满足前提条件）	请求头中指定的一些前提条件失败
413（请求实体过大）	请求文档的大小超过服务器当前愿意处理的大小。如果服务器认为自己能够稍后再处理该请求，则应该提供一个 Retry-After 头
414（请求 URI 过长）	URI（这里就是指 URL）太长
415（不支持的媒体类型）	请求消息中的实体内容的格式不被服务器支持
416（请求的范围不合适）	服务器不能满足客户端在请求中指定的 Range 头
417（预期失败）	客户机可以发送一个值等于 100-continue 的 Expect 请求头，询问是否可以在后面的请求中发送一个附加文档。在这种情形下，服务器可以使用 417 状态码，告诉浏览器它不接受这个文档；或者使用 100（SC_CONTINUE）告诉客户机可以接受

表 3.5 （500～599）

编　号	说　明
500（内部服务器错误）	服务器端的 CGI、ASP、JSP 等程序发生错误
501（未实现）	服务器没有实现客户端正确请求方式的功能
502（无效网关）	服务器作为网关或者代理时，为了完成请求访问下一个服务器，但该服务器返回了非法的应答
503（服务不可用）	服务器由于当前负载过大，例如，当前线程过多或数据库连接池满等原因而不能作出正常响应。对于这种情况，服务器可提供一个 Retry-After 头告诉客户机何时再试
504（网关超时）	由作为代理或网关的服务器使用，表示不能及时地从远程服务器获得应答
505（不支持的 HTTP 版本）	服务器不支持请求行中给出的 HTTP 版本

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

有关HTTP 1.1协议和状态代码定义的详细信息，请参阅Internet Engineering Task Force（IETF）文档RFC 2616。HTTP 协议的WebDAV扩展还包括以上未列出的状态代码，有关WEBDAV 的详细信息，请参阅 IETF 文档 RFC 2518（HTTP Extensions for Distributed Authoring – WEBDAV）。

3.3 通用信息头

通用信息头字段既能用于请求消息，也能用于响应消息，它包括一些与被传输的实体内容没有关系的常用消息头字段。

3.3.1 Cache-Control

如果 Cache-Control 头字段用在客户机发出的请求消息头中，它用于通知位于客户机和服务器之间的代理服务器如何使用已缓存的页面，在这种情况下，Cache-Control 的取值可以为 no-cache、no-store、max-age、max-stale、min-fresh、no-transform、only-if-cached 等。

如果 Cache-Control 头字段用在响应消息中，它用于通知客户机和代理服务器如何缓存该页面，在这种情况下，Cache-Control 的取值可以为 public、private、no-cache、no-store、no-transform、must-revalidate、proxy-revalidate、max-age、s-maxage 等。

在一个 Cache-Control 头字段中可以设置多个取值，例如，

```
Cache-Control: no-store,no-cache,must-revalidate
```

下面是其中一些取值的意义和作用：

- **public** 文档可以被任何代理和客户机缓存。
- **private** 文档只能够被保存在单个用户的私有（非共享）缓存中。
- **no-cache** 如果 no-cache 后没有指定字段名，则客户机和代理服务器不应该缓存该文档，此时的 Cache-Control 头的作用等效于 HTTP/1.0 中定义的 Pragma 头。在 no-cache 后也可以指定一个或多个其他的头字段名，这样代理服务器可以缓存该页面内容来对以后的请求进行响应，但响应消息中不能包含 no-cache 后指定的头字段。例如，如果不想让客户机和代理服务器缓存 Cookie 信息，可以使用 Cache-Control: no-cache=Set-Cookie。
- **no-store** 请求和响应的信息都不应被存储在对方的磁盘系统上，存储与缓存是有区别的，缓存是指将信息保存在内存或磁盘系统中，而存储是专指磁盘系统。
- **must-revalidate** 对于客户机的每次请求，代理服务器必须向服务器验证缓存的文档是否过时，以保证总是发送最新的文档给客户机。
- **proxy-revalidate** 除了只能用于共享缓存外，其作用与 must-revalidate 相同。
- **max-age=xxx** 应该在 xxx 秒后认为文档过时，这可以替代 Expires 头的作用，如果响应头中同时给出 Cache-Control 头的 max-age 设置值和 Expires 头，则以 Cache-Control 头的 max-age 设置值为准。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

- **s-max-age=xxx** 在代理服务器中缓存的文档（通常称之为共享缓存）在 xxx 秒后过时。

3.3.2 Connection

Connection 头字段用于指定处理完本次请求/响应后，客户端与服务器是否还要继续保持连接。当 Connection 请求头的值为 Keep-Alive 时，客户端通知服务器返回本次请求结果后继续保持连接；当 Connection 请求头的值为 close 时，客户端通知服务器返回本次请求结果后关闭连接。HTTP 1.1 默认采用持久连接，在客户端发出的请求消息中没有指定 Connection 头字段时，等效于 Connection 值为 Keep-Alive 的情况。

Connection 头字段的作用

01 启动 Tomcat Web 服务器程序后，在本地计算机的命令行窗口中执行如下命令：

```
telnet localhost 8080
```

在成功连接后启动的 telnet 程序窗口中，输入如下内容：

```
GET /test.html HTTP/1.1<回车>
Host:<空格><回车>
Connection:<空格>close<回车>
<回车>
```

输入上面的内容后，可以看到 Web 服务器返回响应内容后即断开了与 Telnet 程序的连接。

02 再次使用 telnet 命令连接上 Tomcat Web 服务器后，在 Telnet 程序命令窗口中输入不包括 Connection 头字段的上述请求消息的其他部分，这时将看到 Web 服务器返回响应内容后并不会立即断开与 Telnet 程序的连接，而是等待客户端继续发送新的请求。只有等待一段时间以后，服务器还没有收到客户端发送的数据，服务器才强制断开与 Telnet 程序的连接，这就是 Web 服务器的连接超时处理。

03 趁服务器还没有强制断开与 Telnet 程序之间的连接，重复输入第 **02** 步中输入的请求消息内容，可以看到 Web 服务器又返回了响应内容，这就是 HTTP 的持久连接特性，即在一次 TCP 连接上可以包含多次请求和响应过程。

3.3.3 Date

Date 头字段用于表示 HTTP 消息产生的当前时间。服务器回送的正常响应消息中，应该总是设置 Date 头字段。如果客户机发送的请求消息中包含有实体内容，也需要设置 Date

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

头字段。Date 头字段的设置值必须为 GMT 格式，例如，

```
Date: Tue, 11 Jul 2000 18:23:51 GMT
```

3.3.4 Pragma

Pragma 头字段的设置值只能固定为 no-cache，即 Pragma: no-cache。当 Pragma 头字段用于响应消息时，指示 HTTP 1.0 客户机不要缓存文档；当用于请求消息时，指示代理服务器必须返回一个最新的文档，而不能用缓存的文档。一些 HTTP 1.0 的浏览器对这个头字段的支持不是非常可靠，因此，人们过去常常使用取值为 0 的 Expires 头字段来实现同样功能。在 HTTP 1.1 中，使用 Cache-Control 头字段的 no-cache 设置值是一种比 Pragma 更为可靠的实现。

3.3.5 Trailer

一些头字段可以放置在整个 HTTP 消息的尾部，也就是可以在实体内容部分之后出现。对于放置在尾部的头字段，需要在消息头中使用 Trailer 字段说明。例如，

```
Trailer: Date
```

3.3.6 Transfer-Encoding

如果 HTTP 消息的实体内容部分采用了某种传输编码方式，Transfer-Encoding 头字段指定该传输编码方式，目前标准的设置值只有 chunked，用于把整个消息体分成一连串的分段后再进行传输。例如，

```
Transfer-Encoding: chunked
```

每个分段的开始部分都是一个十六进制格式的数字，用来表示这个分段的大小。最后一个分段的大小必须是 0，用于表示整个 chunked 编码数据结束。使用 telnet 命令连接上 Tomcat Web 服务器后，在 Telnet 程序命令窗口中输入如下内容：

```
GET /examples/index.jsp HTTP/1.1<回车>
Host:<空格><回车>
<回车>
```

从 Tomcat Web 服务器返回的结果中，可以看到 Transfer-Encoding 头字段的应用情况。在响应消息中使用设置值为 chunked 的 Transfer-Encoding 头字段后，实体内容被分成了多个段，每段内容都有自己的大小标识信息，所以，这种情况下不必用 Content-Length 头字段来指定实体内容的大小。

3.3.7 Upgrade

Upgrade 头字段允许客户机指定它所支持并希望将当前协议切换到的通信协议，对于

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

101（切换协议）响应码，服务器必须使用 Upgrade 头字段指定切换的协议，例如，

```
Upgrade: HTTP/2.0, SHTTP/1.3
```

3.3.8 Via

Via 头字段用于指定 HTTP 消息所途径的中介代理服务器名称和所使用的协议，这个头字段由代理服务器产生，每个代理服务器必须把它的信息追加到 Via 字段的最后，以反映 HTTP 消息途径的多个代理服务器的顺序，例如，

```
Via: HTTP/1.1 Proxy1, HTTP/1.1 Proxy2
```

3.3.9 Warning

Warning 头字段主要用于说明其他头字段和状态码不能说明的一些附加警告信息，例如，返回的实体内容可能已经过时。

3.4 请 求 头

请求头字段用于客户端在请求消息中向服务器传递附加信息，主要包括客户端可以接受的数据类型、压缩方法、语言，以及发出请求的超链接所属页面的 URL 地址等信息。

3.4.1 Accept

Accept 头字段用于指出客户端程序（通常是浏览器）能够处理的 MIME（Multipurpose Internet Mail Extension ，多用途 Internet 邮件扩展）类型。例如，PNG 格式的图像具有某种优于 GIF 格式图像的压缩优点，但只有少数浏览器支持 PNG 格式。如果服务器同时提供了这两种格式的图像文件，并且浏览器能支持 PNG 格式，浏览器可以发送包含 image/png 的 Accept 头字段，服务器检查到 Accept 头中包含 image/png 这种 MIME 类型，则在它动态生成的网页中的 IMG 元素中使用 xxx.png 的文件，否则，它将使用 xxx.gif 的文件。在 Accept 头字段中可以指定多个以逗号分隔的 MIME 类型，例如，

```
Accept: text/html,image/*
```

3.4.2 Accept-Charset

Accept-Charset 头字段用于指出客户端程序可以使用的字符集，关于各种字符集的名称信息，可以参阅 RFC 1700。在 Accept-Charset 头字段中也可以指定多个以逗号分隔的字符集，例如，

```
Accept-Charset: ISO-8859-1,unicode-1-1
```

3.4.3 Accept-Encoding

Accept-Encoding 头字段用于指定客户机能够进行解码的数据编码方式，这里的编码方

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

式通常是指某种压缩方式。在传输较大的实体内容之前，对其进行压缩，可以节省网络带宽和传送时间，最常见的压缩编码方式是 gzip 和 compress 这两种格式。如果服务器接收到这个请求头，它通过使用其中指定的一种格式对原始文档内容进行压缩编码后再作为响应消息的实体内容，并在 Content-Encoding 响应头中指出实体内容所使用的压缩编码格式，浏览器在接收到这样的实体内容之后，需要对其进行反向解压缩。Accept 请求头中指定的 MIME 类型是指解压缩后的实体内容类型。在 Accept-Encoding 头字段中可以指定多个以逗号分隔的数据编码方式，例如，

```
Accept-Encoding: gzip,compress
```

3.4.4 Accept-Language

Accept-Language 头字段用于指定客户机期望服务器返回哪个国家语言的文档，可以指定多个以逗号分隔的国家语言。对于一些 Web 站点，可以根据用户所使用的浏览器国家语言版本的不同，分别返回该国家语言的网页文档。例如，使用中文的 IE 浏览器访问 www.google.com 站点时，服务器返回的是使用中文语言的网页文件。单击 IE 浏览器的“工具”→“Internet 选项”菜单，在打开的“Internet 选项”对话框中，单击“语言”按钮，在打开的“语言首选项”对话框中添加英语，并将其上移到语言列表中的第一个位置，如图 3.14 所示。

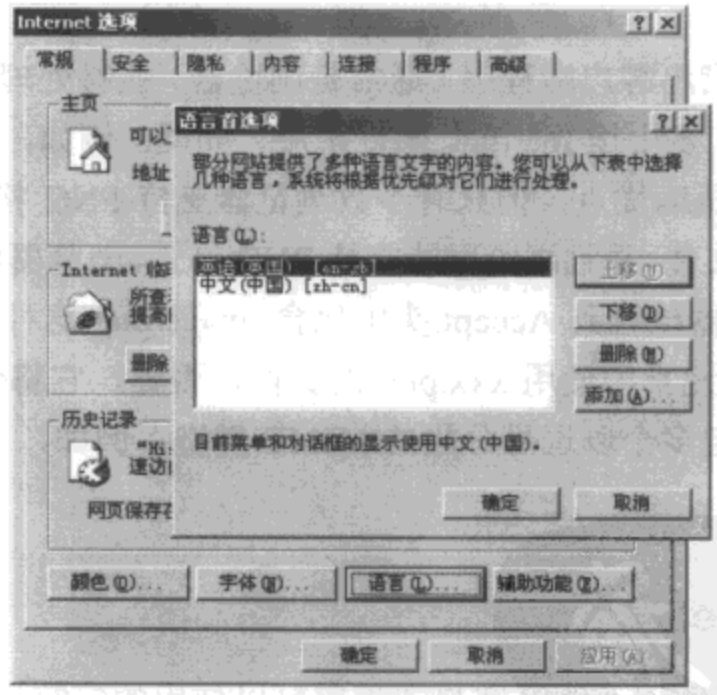


图 3.14

设置好语言首选项后，再次访问 www.google.com 站点，这时服务器返回的是使用英语的网页文件。上面的实验结果很可能会受到 Google 站点的策略调整和 Cookie 的影响，例如，作者在编写本书的初稿时，按照上面的操作，看到了预期的效果，但等到作者几个月后再次校稿时，即使将英语设置成了语言首先项，但所看到的结果却仍然是中文网页。这可能是由于英语已成为了国际通用语言的原因，Google 对语言首选项设置成英语的情况

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

进行了忽略吧！如果把语言首选项设置成日语后，服务器可以返回日语的网页文件。

在 Web 服务器上实现上述功能的原理非常简单，因为，IE 浏览器会根据“语言首选项”对话框中的语言列表，生成相应的 Accept-Language 请求头，例如，

```
Accept-Language: en-gb,zh-cn
```

Web 服务器只要检查 Accept-Language 请求头中的信息，按照其中设置的国家语言的先后顺序，就知道应该返回哪种国家语言的网页文档给浏览器。如果 Web 服务器不能返回 Accept-Language 设置值中的位于前面的国家语言的网页文档时，Web 服务器将返回后面的国家语言的网页文档。关于各个国家语言的名称信息，例如，中国是 zh-cn，可以参阅 RFC 1766，其实，只要在图 3.14 中添加进某个国家的语言，显示在中括号里面的内容就是这个国家语言的名称。

Accept-Language 头字段的作用

01 在命令行窗口中执行 telnet www.google.com 80，在成功连接后启动的 Telnet 程序窗口中，输入如下几行内容：

```
GET / HTTP/1.1
Host:
Accept-Language: en-us,zh-cn
<空行>
```

可以看到 Google 返回的正文部分是英文字符的网页文档。省略上面的 Accept-Language 字段部分，Google 默认返回的正文部分也是英文字符的网页文档。

02 重新连接上 Google 站点，在成功连接后启动的 Telnet 程序窗口中，输入如下几行内容：

```
GET / HTTP/1.1
Host:
Accept-Language: zh-cn
<空行>
```

可以看到 Google 返回的正文部分是中文字符的网页文档。

在作者编写本书初稿时，实验结果就是本书所讲述的情形。但是，后来受到政治因素的影响，Google 调整了其处理策略，对于来自中国大陆地区的访问请求，不管 Accept-Language 头字段的设置值如何，Google 总是返回 302 状态码和 Location 头字段来将浏览器重定向到另外一个专门为中国大陆地区准备的站点。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

3.4.5 Authorization

当客户端访问受口令保护的网页文件时，Web 服务器会发送 401（Unauthorized）响应状态码和 WWW-Authenticate 响应头，要求客户机使用 Authorization 请求头来应答。根据 WWW-Authenticate 响应头中指定的认证方式的不同，Authorization 请求头中的内容格式也不一样。有两种认证方式，分别是 BASIC 和 DIGEST。对于 BASIC 验证方式，客户端需要把用冒号（:）分隔的用户名和密码进行 Base64 编码之后传送给 Web 服务器。例如，要发送用户名为“zxx”和密码为“123456”的认证信息，使用的请求头字段内容应该如下：

```
Authorization: Basic enh4OjEyMzQ1Ng==
```

使用 Base64 编码的数据很容易被解码，这实际上相当于是一种未加密的明文传送方式，装备了网络监视工具的计算机截获到此信息后，很容易知道用户名和密码。

如果使用 DIGEST 验证方式，服务器首先向浏览器发送一些将用于验证过程的信息及附加信息，浏览器将这些信息与用户名和密码以及某些其他信息进行混合后，再执行 MD5 散列算法，并将散列算法的结果和附加信息一起以明文文本通过网络发送到服务器。服务器也使用与客户端同样的信息和附加信息验证过程，对它所保存的客户端密码执行散列算法，然后，将它的计算结果与客户端的计算机结果比较，只有这两个数字完全相同时才允许访问。使用 DIGEST 验证方式，即使装备了网络监视工具的计算机截获了网络传输信息，也不可能将密码从散列结果中破译出来。在这个过程中使用附加的信息，可以防止他人复制散列值结果并再次使用。DIGEST 认证方式的细节比较复杂，要对此进行深入研究的读者，请参阅 RFC 2617。

3.4.6 Expect

Expect 头字段用于指定客户机请求服务器采取的特殊行动，目前设置值只有 100-continue，用于询问服务器是否可以在后面的请求中发送一个附加文档。

3.4.7 From

From 头字段用于指定请求发送者的 E-mail 地址，它只被一些特殊的 Web 客户端程序使用，浏览器不会使用到它。

3.4.8 Host

Host 头字段用于指定资源所在的主机名和端口号，格式与资源的完整 URL 中的主机名和端口号部分一样，例如，

```
Host: www.it315.org:80
```

如果端口号等于连接服务器时所使用的端口号，则可以省略。在 HTTP 1.1 中，浏览

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

器和其他客户端发送的每个请求消息中必须包含 Host 请求头字段，以便 Web 服务器能够根据 Host 头字段中的主机名来区分客户端所要访问的虚拟 Web 站点。即使 Web 服务器没有使用主机头字段创建虚拟 Web 站点，客户端也要发送一个空值的 Host 头字段。如果使用 HTTP/1.1 协议的请求消息中没有 Host 头字段，服务器将回应 400（Bad Request)响应状态码。在使用 IE 浏览器访问 Web 站点时，IE 浏览器能根据地址栏中的 URL 地址自动生成相应的 Host 请求头。

3.4.9 If-Match

浏览器以及位于浏览器和服务器之间的代理服务器，都可以缓存 Web 服务器回应的网页文档。当用户再次访问已缓存的页面时，只有网页内容已被更新，服务器才需把该页面的内容重新传递到客户机，以减少不必要的网络传输流量。可以通过定义各种条件来判断网页内容是否已更新，满足某些条件时则被视为网页内容已更新。服务器在传送给客户机网页文件的内容时，可以传送一些代表实体内容特征的头字段，这些头字段被称为实体标签。当客户机再次向服务器请求这个网页文件的内容时，可以使用 If-Match 头字段附带以前缓存的实体标签内容，这个请求将被视为一个条件请求。例如，

```
If-Match: "xyzzy", "r2d2xxxx", "c3piozzzz"
```

其中，"xyzzy"等内容是客户机上次访问该页面时，服务器使用如下字段传送的实体标签：

```
ETag: "xyzzy"
```

如果客户机使用的是 GET 请求方式，服务器检索 If-Match 头中的实体标签内容，并与服务器端的代表当前网页内容特征的实体标签内容进行比较。如果两者相同，则表示网页内容没有更改，Web 服务器不返回网页内容，让客户机仍然使用以前缓存的网页内容。否则，服务器返回新的网页文件内容和新的实体标签内容头字段。如果客户机使用的是 PUT 请求方式，即表示要更新服务器上的网页内容，服务器检索 If-Match 头中的实体标签内容，并与服务器端的代表当前网页内容特征的实体标签内容进行比较。如果两者相同，则表示网页内容没有更改，客户机可以更新网页内容。否则，说明网页内容已经被修改过，不允许 PUT 方法成功执行，以防止在多个人维护一个网页内容的情况下，一个人覆盖掉另外一个人对网页所做的修改。

3.4.10 If-Modified-Since

当用户再次访问已缓存的页面时，只有该文档自某个指定的时间以来发生过更改，客户机才有必要下载此文档。If-Modified-Since 请求头用于指定这个时间，其设置值为 GMT 格式。使用 If-Modified-Since 的请求将被视为一个条件请求，只有文档的修改时间比 If-Modified-Since 请求头指定的时间新时，服务器才会返回文档内容。如果自从 If-Modified-Since 指定的时间以来，网页内容没有发生修改，服务器将返回一个 304（Not Modified）状态码来表示浏览器缓存的版本是最新的，而不向浏览器返回文档内容，浏览器则继续使

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

用以前缓存的内容。通过这种方式，可以在一定程度上减少浏览器与服务器之间的通信数据量，从而提高了通信效率。

3.4.11 If-None-Match

If-None-Match 头字段的作用与 If-Match 头字段的作用正好相反，详细的信息请参看 If-Match 头字段的讲解。

3.4.12 If-Range

If-Range 头字段只能伴随着 Range 头字段一起使用，其设置值可以是实体标签或 HTTP 的 GMT 格式的时间。如果设置值为实体标签，且该标签内容与服务器端的代表当前网页内容特征的实体标签内容相同，则服务器按 Range 头的要求返回网页中的部分内容，否则，服务器返回当前网页的所有内容。如果设置值为 HTTP 的 GMT 格式的时间，只有自从这个时间以来，服务器上保存的网页文件没有发生修改，服务器按 Range 头的要求返回网页中的部分内容，否则，服务器返回当前网页的所有内容。

3.4.13 If-Unmodified-Since

If-Unmodified-Since 头字段一般只用于 PUT 方式，其设置值为 HTTP 的 GMT 格式，作用与 If-Modified-Since 正好相反。它指出只有在服务器上保存的网页文件的修改时间比给出的时间还早时，PUT 操作应该才能继续。If-Modified-Since 用于 GET 请求，表示只有服务器上的文档比浏览器所缓存的版本新时，服务器才返回这个文档给浏览器。而 If-Unmodified-Since 用于 PUT 请求，表示只有服务器上的文档比浏览器所要上传的文档旧时，服务器才更新此文档。

3.4.14 Max-Forwards

Max-Forwads 头字段指定当前请求可以途径的代理服务器数量，每经过一个代理服务器，此数值就减一。如果当此数值为 0 时，请求还只是达到了某个代理服务器，而没有到达最终的 Web 服务器的话，那个代理服务器应该终止转发这个请求，由它来完成对客户机的最终响应。

3.4.15 Proxy-Authorization

除了是针对代理服务器的用户信息认证外，Proxy-Authorization 头字段的作用与用法完全与 Authorization 头字段相同。

3.4.16 Range

Range 头字段用于指定服务器只需返回文档中的部分内容及内容范围，这对较大文档

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

的断点续传非常有用。如果客户机在一次请求中只接收到服务器返回的部分内容就中断了，可以在第二次请求中，使用Range头字段要求服务器只返回中断位置以后的内容。Range头有以下几种使用格式：

（1）Range: bytes=100-599

（2）Range: bytes=100-

（3）Range: bytes=-100

其中，第一种格式请求服务器返回文档中的第100个字节到第599个字节之间的内容，包括位置为第100个和第599个的字节，字节顺序从0开始计算；第二种格式请求服务器返回文档中的第100个字节以后的所有内容；第三种格式请求服务器返回文档中的最后100个字节的内容。

3.4.17 Referer

浏览器向服务器发出的请求，可能是直接在浏览器地址栏中输入 URL 而发出，也可能是单击另一个网页上的超链接而发出的。对于第二种情况，浏览器使用 Referer 头字段指定发出请求的超链接源（超链接所在网页）的 URL。对于第一种直接在浏览器地址栏中输入 URL 的情况，浏览器不应发送 Referer 请求头。浏览器发送 Referer 请求头，可以让服务器了解和追踪发出本次请求的起源 URL 地址。很多网站管理员和公司负责人都非常想知道自己网站的访问者是如何导航进来的，以便了解在哪些网站上投放的广告链接是有效的，这可以通过在服务器端记录每个访问请求的 Referer 头字段来解决。虽然 Referer 头字段很有用，但不要过分依赖和相信它，因为用户可以使用类似 Telnet 这样的客户端程序，轻易地伪造出一个 Referer 头。注意，由于 HTTP 协议原作者的拼写错误，这个头字段的拼写是 Referer，而不是 Referrer。

3.4.18 TE

TE 头字段用于指定客户机可接受的扩展传输编码类型（除 chunked 之外的类型）和使用 chunked 传输编码类型时是否允许使用 Trailer 头字段，即每个 chunck 分段内容的后面是否允许有头字段。可以组合使用“trailers”和允许的扩展传输编码类型，相互之间用逗号（,）分隔，例如，

```
TE: trailers,deflate
```

3.4.19 User-Agent

User-Agent 头字段用于指定浏览器或其他客户端程序的类型和名称，以便服务器针对不同类型的浏览器而返回不同的内容。例如，服务器可以检查 User-Agent 头，如果发现客户机是一种无线手持终端，就返回一个 WML 文件；如果发现客户端是一种普通的浏览器，则返回通常的 HTML 文件。Internet Explorer 在生成的 User-Agent 请求头中首先会列出一

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

个“Mozilla”（Netscape）版本，然后才附带列出实际的浏览器版本，例如，

```
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)
```

这是为了与 JavaScript 兼容，JavaScript 有时要用 User-Agent 头字段的值来确定支持哪些功能。

3.5 响 应 头

响应头字段用于服务器在响应消息中向客户端传递附加信息，包括服务程序名、被请求资源需要的认证方式、被请求资源已移动到的新地址等信息。

3.5.1 Accept-Range

Accept-Range 头字段用于说明当前 Web 服务器是否接受 Range 请求和 Range 请求中指定的数据的单位。如果服务器想告诉客户机不要使用 Range 请求，则应使用下面的头信息：

```
Accept-Range: none
```

如果服务器要告诉客户机可以使用以 byte 为单位的 Range 请求，则应使用下面的头信息：

```
Accept-Range: bytes
```

3.5.2 Age

Age 头字段用于指出当前网页文档可以在客户机或代理服务器中缓存的有效时间，设置值是一个以秒为单位的时间数，例如，

```
Age: 315315315
```

客户机再次访问已缓存的某个网页文档内容时，先使用当前的时间值减去服务器上次返回该网页时所设置的 Date 头字段的时间值，如果结果值小于服务器上次返回该网页时所设置的 Age 头字段的时间值，客户机直接使用缓存中的网页内容。否则，客户机才向服务器发出针对该页面的访问请求。

3.5.3 Etag

Etag 头字段用于向客户机传送代表实体内容特征的标记信息，这些标记信息称为实体标签，用于比较在不同时间获得的同一资源路径下的实体内容是否相等。例如，

```
Etag: b38b9-17dd-367c5dcd
```

3.5.4 Location

Location 头字段用于通知客户机应该到哪个新的地址去获取文档。状态码为 300～399 的响应消息都应该使用 Location 头字段将新的文档地址告诉给客户机，以便客户机自动重

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

新连接到新地址并检索新文档。由于当前响应并没有直接返回内容给客户机，所以使用 Location 头的 HTTP 消息不应该有实体内容，由此可见，在 HTTP 消息头中不能同时出现 Location 和 Content-Type 这两个头字段。

Location 头字段的作用

在 Tomcat Web 服务器中为目录设置默认网页文档的机制，就是通过使用 Location 头字段的重定向功能来实现的。使用 telnet 命令连接上 Tomcat Web 服务器后，在 Telnet 程序命令窗口中，输入如下内容：

```
GET / HTTP/1.1<回车>
Host:<空格><回车>
<回车>
```

Web 服务器返回的内容如图 3.15 所示，它使用 302 响应状态码和 Location 头字段指示客户机应该自动去重新访问当前计算机的默认站点的根目录下的 index.jsp 文件。

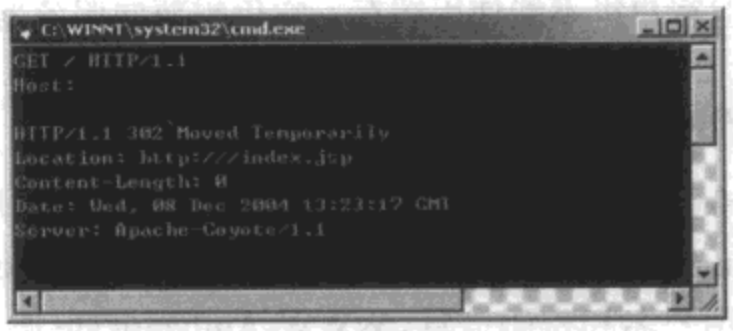


图 3.15

笔者认为 Tomcat 使用这种机制来实现目录的默认网页文档，考虑得有点欠妥，因为浏览器每次访问某个站点或站点下的目录时，总是要发出两次请求才能获得该站点的首页和目录的默认网页文档，这不如在浏览器的第一次请求响应中就直接返回目录的默认网页文档。

3.5.5 Proxy-Authenticate

除了是针对代理服务器的用户信息认证外，Proxy-Authenticate 头字段的作用与用法与 WWW-Authenticate 头字段完全相同。

3.5.6 Retry-After

Retry-After 头字段用于告诉客户机可以在什么时间重复发出刚才的请求，需要与 503（Service Unavailable）响应状态码结合使用。它设置值为 HTTP 的 GMT 格式，例如，

```
Retry-After: Tue, 11 Jul 2000 18:23:51 GMT
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

3.5.7 Server

Server 头字段用于指定服务器软件的产品名称，例如，

```
Server: Microsoft-IIS/5.0
```

3.5.8 Vary

Vary 头字段用于指定影响了服务器所生成的响应内容的那些请求头字段名，例如，

```
Vary: Accept-Language
```

上面的字段设置说明服务器响应的内容受到了客户机发送的 Accept-Language 请求头的影响，服务器根据 Accept-Language 请求头的设置值的不同，返回相应国家语言的网页内容。即使客户机已经缓存了这次请求返回的内容，当客户机再次请求这个资源路径时，需要检查 Vary 头字段中指定的请求头字段，看这些请求头字段的这次设置与上次的设置是否相同，以此作为是否使用缓存的条件。例如，如果上次的 Accept-Language 设置值为 en-gb，而这次的 Accept-Language 设置值为 zh-cn，即使客户机使用请求资源路径的本地缓存的其他条件都成立，但客户机不能使用缓存，它仍需要向服务器发出访问请求。

3.5.9 WWW-Authenticate

当客户端访问受口令保护的网页文件时，服务器需要结合采用 401（Unauthorized）响应状态码和 WWW-Authenticate 响应头，指示客户端应该在 Authorization 请求头中以什么样的认证方式提供用户名和密码信息。WWW-Authenticate 响应头中可以指定两种认证方式：BASIC 和 DIGEST，如果要求客户端采用 BASIC 方式传送认证信息，语法格式如下：

```
WWW-Authenticate: BASIC realm="it315"
```

其中，realm 属性用于指定当前资源所属的域，域定义了同一个主机内的一个受保护区间（一组需要保护的资源），它可以是任意字符串。同一台主机上可以有多个域，相同域内的所有资源都共享相同的访问账户。如果某个账户具有访问某个资源的权限，那么该账户应该能访问同一个域中的其他资源。根据 HTTP 验证的规范，与某一资源具有相同的目录路径或位于其目录路径的子目录中的资源，与该资源使用相同的域。

DIGEST 认证方式的细节比较复杂，要对此进行深入研究的读者，请参阅 RFC 2617。

3.6 实 体 头

实体头是实体内容的元信息，描述了实体内容的属性，包括实体信息类型、长度、压缩方法、最后一次修改时间、数据有效期等。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

3.6.1 Allow

Allow 头字段用于指定客户端请求的资源所支持的请求方法（如 GET、POST 等），它通常伴随着 405（Method Not Allowed）响应状态码一起使用。

3.6.2 Content-Encoding

Content-Encoding 头字段用于指定实体内容的压缩编码方式。Content-Type 响应头字段中指定的 MIME 类型是指实体内容压缩编码前的类型。关于对文档内容进行压缩编码的细节，请参阅 Accept-Encoding 请求头。

3.6.3 Content-Language

Content-Language 头字段用于指定返回的网页文档的国家语言类型，其设置值应该是 zh-cn、en-us、ja 等国家语言的标准名称。由于同一个字符在不同的国家语言中的样式和意义上可能会有略微区别，例如，不同国家的标记为引用的文本的显示样式可能就不一样，如果一些客户端软件正好要对字符文本按不同的国家语言进行不同处理时，Content-Language 头字段就比较重要了。对于简体中文，繁体中文，日文，韩文来说，如果它们的网页文档使用 Unicode 编码，从编码本身上就无法区分出是哪一个国家的文字，那么使用国家语言信息就可以帮助一些发声客户端软件来决定如何发音，帮助一些视觉客户端软件决定以哪种字体显示网页中的文本。图 3.16 中演示了 Mozilla 浏览器对同一个字符在不同国家语言下的显示效果。

图 3.16

3.6.4 Content-Length

Content-Length 头字段用于表示实体内容的长度（字节数），浏览器与 Web 服务器之间使用持久（keep-alive）的 HTTP 连接时，这个头字段在非 chunked 传输编码的响应消息中是必不可少的。HTTP/1.1 支持在一个 TCP 连接上发送多个请求，Web 服务器必须在每

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

个应答中发送一个 Content-Length 的响应头来表示各个实体内容的长度，以便客户端能够分辨出上一个响应内容的结束位置，而不会将上一个响应和下一个响应的内容混为一谈。对于请求消息，只有 POST、PUT 和 DELETE 等方式具有实体内容，所以，Content-Length 头只适用于这几种请求方式。

3.6.5 Content-Location

Content-Location 头字段用于指定响应消息中所封装的实体内容的实际位置路径，例如，

```
Content-Location: http://www.it315.org/java_cn.html
```

不能简单地认为响应消息中的实体内容的位置路径就是请求资源的路径，当一个请求资源路径对应有多种实体内容形式时，例如，同一请求资源可能有多个国家语言的版本，每个国家语言的版本都有自己的位置，这种情况下的请求资源路径与响应的实体内容所在的路径是不同的。Content-Location 的设置值可以是绝对路径，也可以是相对路径。如果是相对路径，则是相对请求资源路径而言的。

3.6.6 Content-MD5

Content-MD5 头字段用于指定实体内容的 MD5 算法算出的数字摘要的 base64 编码结果，以提供实体内容的完整性检查。md5 算法可用于对一段信息进行不可逆的变换运算，产生一个 128 比特位的数字摘要。如果对输入信息做了任何形式的改变，对改变后的信息再次进行 MD5 运算所产生的数字摘要，与改变前的内容所运算出的 MD5 数字摘要都是不同的。Cotent-MD5 可用于对实体内容在网络传输过程中的意外损坏检查，但并不能防止恶意的黑客攻击。例如，黑客篡改网页内容后，然后用篡改后的内容运算出 MD5 数字摘要，并将这个摘要作为 Content-MD5 头的设置值，对于这种情况，是无法做出正确的检测和判断的。

3.6.7 Content-Range

Content-Range 头字段用于指定服务器返回的部分实体内容的位置信息。只有客户机使用了 Range 请求头要求服务器返回实体的部分内容时，服务器的响应头中才应包含 Content-Range 头。例如，

```
Content-Range: bytes 2543-4532/7898
```

其中，bytes 说明后面的数据以 byte 为单位，2543-4532 说明返回的内容从第 2543 个字节开始到第 4532 个字节结束，/7898 说明整个实体内容的大小为 7898 个字节。

3.6.8 Content-Type

不管浏览器接收到什么格式的实体内容，实际上都是一堆由 0 和 1 组成的二进制数据，浏览器并不知道这些数据的具体含义，也就是说，浏览器无法知道它所接收到的数据是普

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

通网页、图片，还是 excel 文档，因此，Web 服务器必须通过某种方式告诉浏览器所接收到的实体内容代表的是哪种格式的数据，浏览器才能进行正确的处理。计算机应用中有多种通用的数据格式，人们为每一种通用的数据格式都定义了一个名称，这个名称被称之为 MIME（Multipurpose Internet Mail Extension，多用途 Internet 邮件扩展）类型。MIME 最初是用于解决邮件传输系统中的数据格式问题的，因为通过 HTTP 协议传输的数据也有各种类型，所以，HTTP 也采用 MIME 来标识不同的数据类型。

Content-Type 头字段用于指出实体内容的 MIME 类型，客户端通过检查服务器响应消息的 Content-Type 头字段中的 MIME 类型，就能知道实体内容的数据格式和知道以何种方式来进行处理了。在很多时候，Web 服务器回送给浏览器的数据内容都是直接从服务器本地硬盘中的某个文件中读取到的，Web 服务器读取本地硬盘中的某个文件时，它又是怎么知道该文件中的数据内容的 MIME 类型的呢？大多数 Web 服务器都可以设置文件扩展名与 MIME 类型的映射关系，从而可以根据请求资源的扩展名，自动确定 Content-Type 实体头中的 MIME 类型。Tomcat 可以通过在 web.xml 文件中添加<mime-mapping>元素来实现这种设置，如下所示：

```
<mime-mapping>
    <extension>abs</extension>
    <mime-type>audio/x-mpeg</mime-type>
</mime-mapping>
```

web.xml 文件中的每一个<mime-mapping>元素都用于设置一种静态资源的 MIME 类型，其中的<extension>子元素用于指定一个文件扩展名，<mime-type>子元素用于指定该文件扩展名所代表的静态资源的 MIME 类型。在<Tomcat 主目录>/conf 目录下的 web.xml 文件中的设置 MIME 映射将对所有的 Web 应用程序都起作用，而在某个 Web 应用程序的 web.xml 文件中的设置 MIME 映射只对该 Web 应用程序起作用。

MIME 类型包含主类型和子类型，两者之间用“/”分隔。例如，主类型为 image，子类型为 jpg 的 MIME 类型表示形式为：image/jpg。使用“*/*”代表所有的 MIME 类型，使用“image/*”则代表所有 image 的子类型。如果子类型以“x-”开头，则表示目前还属于实验性的类型。关于 MIME 更多的信息，请参阅 RFC 1521 和 RFC 1522。Content-Type 头中的 MIME 类型后面还可以指定响应内容所使用的字符集编码，两者之间用分号（;）和空格隔开，例如，Content-Type: text/html; charset=GB2312。如果 Content-Type 头中没有指定字符集编码类型，则默认使用 ISO-8859_1（Latin）。

Tomcat 中的文件扩展名与 MIME 的映射关系

01 用 UltraEdit 打开<Tomcat 主目录>/conf 目录下的 web.xml 文件，拖动滚动条中的滚动滑块，可以看到类似下面的许多<mime-mapping>元素：

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
<mime-mapping>
    <extension>abs</extension>
    <mime-type>audio/x-mpeg</mime-type>
</mime-mapping>
<mime-mapping>
    <extension>ai</extension>
    <mime-type>application/postscript</mime-type>
</mime-mapping>
……
```

在<Tomcat 主目录>/conf 目录下的 web.xml 文件已经建立了各种常见的文件扩展名与 MIME 类型的映射关系，我们可以在这个 web.xml 文件，或在每个 Web 应用程序自己的 web.xml 文件中，添加新的 MIME 映射，删除和编辑已有的 MIME 映射。

02 从上面列出的<mime-mapping>元素中可以看到，扩展名为 abs 的文件的所映射的 MIME 为 audio/x-mpeg。在 Tomcat 的默认站点的根目录中，创建一个内容为空的文本文件，将其名称设置为 test.abs。注意确保修改了文件的扩展名为“.abs”，如果 Windows 资源管理器的“隐藏已经类型的文件扩展名”选项处于选中状态，文件的实际扩展名有可能是“.txt”，而不是看到的“.abs”。

03 使用 telnet 命令连接上 Tomcat Web 服务器后，在 Telnet 程序命令窗口中，输入如下内容：

```
GET /test.abs HTTP/1.1<回车>
Host:<空格><回车>
<回车>
```

Web 服务器返回的内容如图 3.17 所示，可以看到其中的 Contenet-Type 头字段的值为 audio/x-mpeg，这正是 web.xml 文件中设置的情形。

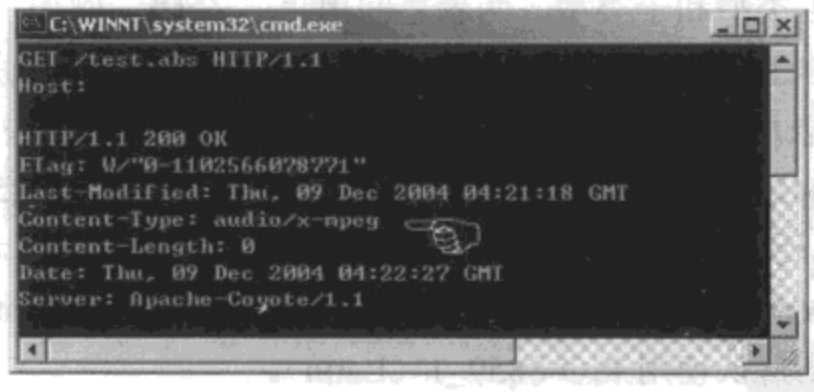


图 3.17

04 如果使用浏览器访问 test.abs 文件，浏览器也将获得图 3.17 中的返回内容，但它会根据 Content-Type 头中的 MIME 类型做出更进一步的处理。在 IE 浏览器地址栏中输入如下地址：http://localhost:8080/test.abs

浏览器检查到 Content-Type 头的设置值为 audio/x-mpeg 后，它把实体部分的内容当做这种格式的音频数据，从而启动本地计算机上安装和注册的相应音频播放程序来

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

播放。因为 test.abs 中并没有真正包含相应格式的数据，音频播放程序将报告格式不正确的错误。

3.6.9 Expires

Expires 头字段用于指定当前文档应该在什么时候被认为过期，浏览器到那个时候以后不能再继续使用本地缓存，而是在有需要时应该向服务器发出新的访问请求。Expires 头字段的设置值必须为 GMT 格式，例如，

```
Expires: Tue, 11 Jul 2000 18:23:51 GMT
```

如果 Expires 头字段的设置值格式不正确，特别是干脆设置为“0”时，客户端应该认为当前网页已过期，不用缓存。因为有的旧浏览器对 Pragma 头字段的支持不可靠，并且根本就不能识别 Cache-Control 头字段，所以，服务器端常常用 Expires 头字段来阻止浏览器进行缓存。

3.6.10 Last-Modified

Last-Modified 头字段用于指定文档的最后更改时间，设置值为 GMT 格式。

当客户机缓存一个文档后，如果它知道文档的最后更新时间，它将在以后的请求消息中发送 If-Modified-Since 请求头来指出缓存文档的最后更新时间，If-Modified-Since 请求头中填写的时间，就是客户机从上次的响应消息中接收到的 Last-Modified 头字段指定的时间。

3.7 扩 展 头

在 HTTP 消息中，也可以使用一些在 HTTP 1.1 正式规范里没有定义的头字段，这些头字段统称为自定义的 HTTP 头或扩展头，它们通常被当做是一种实体头处理。由于扩展头并没有在标准规范中定义，所以，从原理上讲，不能保证接收程序都能正确理解扩展头所表达的意义。有关各种 HTTP 扩展信息头的文档资料及其说明，请参见 http://www.w3.org/Protocols，现在流行的浏览器实际上都支持 Cookie、Set-Cookie、Refresh 和 Content-Disposition 等几个常用的扩展头字段。下面只介绍 Refresh 和 Content-Disposition 这两个头字段，Cookie 和 Set-Cookie 头字段在第 7 章进行专门讲解。

3.7.1 Refresh

Refresh 头字段用于告诉浏览器过多少秒后自动刷新页面，例如，下面的字段设置告诉浏览器过一秒钟后刷新此页面的访问。

```
Refresh: 1
```

在一些聊天网站中，就是通过使用 Refresh 头字段来自动刷新聊天记录的显示网页。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

在 Refresh 头字段的时间值后面还可以增加一个 URL 参数，时间值与 URL 之间用分号（;）分隔，这将让浏览器在指定的时间值后跳转到其他网页，而不是刷新当前页面。例如，下面的字段设置告诉浏览器过一秒钟后跳转到 www.it315.org 站点的主页。

```
Refresh: 1;url=http://www.it315.org
```

3.7.2 Content-Disposition

假如服务器希望浏览器不是直接处理响应的实体内容，而是可以让用户选择将响应的实体内容保存到一个文件中，这就需要使用 Content-Disposition 头字段。Content-Disposition 头字段是从 RFC 1806（已被 RFC 2183 替代）中关于 Content-Disposition 的定义中借鉴过来的，在 RFC 2183 的定义中，Content-Disposition 指定了接收程序处理数据内容的方式，有 inline 和 attachment 两种标准方式，inline 表示直接处理，而 attachment 则要求用户干预并控制接收程序如何处理数据内容。在 HTTP 应用中，只有 attachment 是 Content-Disposition 的标准方式。如果将 Content-Disposition 设置为 attachment，在其后还可以指定 filename 参数。filename 参数值是服务器建议浏览器将实体内容保存到文件中时的文件名称，浏览器应该忽略 filename 参数值中的目录部分，只取参数中的最后部分作为文件名。在设置 Content-Disposition 头字段之前，一定要设置指定 Content-Type 头字段，例如，

```
Content-Type: application/octet-stream
Content-Disposition: attachment; filename=aaa.zip
```

3.8 思考与实践

1. 请按自己的理解简述 HTTP 1.1 与 HTTP 1.0 的区别。
2. 请描述 HTTP 请求消息和 HTTP 响应消息的组成结构，并各举一例进行说明。
3. 浏览器分别在哪些情况下使用 GET 方式和 POST 方式访问 Web 服务器？如果浏览器传递给 Web 服务器的参数内容超过 1KB，应该使用哪种方式发送请求消息？
4. 请描述 200、206、302、304、401、404 和 500 等响应状态码所表示的意义。
5. 请列举三种禁止浏览器缓存的头字段，并写出相应的设置值。
6. 如果 Web 服务器端程序想根据访问者所使用的浏览器国家语言版本来返回该国家语言的网页文档内容，应该检查浏览器发送的哪个请求头字段来识别其国家语言信息？
7. 请根据自己的理解描述 Referer 头字段的作用。
8. 请根据自己的理解描述 Content-Type 头字段的作用。如果将一个 Word 文档（文件扩展名为.doc）放置在 Tomcat 服务器的根目录中，要让 IE 浏览器访问这个 Word 文档时自动打开 Word 程序，在 IE 浏览器所在的客户机和 Tomcat 服务器上分别应具备什么条件。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

第 4 章　Servlet 开发基础

本章的实验假设读者已经按 2.4.3 节的讲解，在/it315 虚拟目录所映射的本地文件系统目录 d:\myWeb 中创建了 Web-INF 子目录和在 Web-INF 子目录中创建了 Web.xml 文件。另外，**本章的实验还需要在 Web-INF 目录中事先创建好一个名为 classes 的子目录**，编写的所有 Servlet 程序都应发布到这个 classes 子目录中。

提示　为了能够参照本章的实验步骤进行操作，建议读者使用 Tomcat 4.1.31 作为实验环境，以便与作者讲解的实验环境相一致。

4.1　Servlet 的有关概念与前置知识

4.1.1　什么是动态网页

如果浏览器在不同时刻或不同条件下访问 Web 服务器上的某个页面，浏览器所获得的页面内容可以发生变化，那么这个页面就称之为动态网页。动态网页与静态网页最明显的区别就是针对页面的访问请求，Web 服务器在不同条件下返回给客户端的网页内容是否总是相同。例如，当浏览器访问 Web 服务器上的某一资源路径时，如果 Web 服务器返回的网页内容中总是要包含 Web 服务器的当前时间，因为时间是在不断变化的，所以服务器在不同时刻返回给客户端的网页内容也是在不断变化的，客户端从服务器端得到的响应结果就属于动态网页。要想让浏览器访问某一 Web 页面时，能够查询出一些动态变化的实时信息，例如，某日某次列车的车票剩余状况，浏览器在不同时刻所获得的 Web 页面的内容必须随着车票的剩余状况进行变化，这就必须采用动态网页来实现。

不管浏览器是从本地硬盘上打开网页文件，还是从 Web 服务器上获取 Web 页面，浏览器最终的显示效果总是由其所获得的内容来决定。当浏览器访问 Web 服务器上的某个 Web 页面时，它所接收到的内容就是 Web 服务器通过网络传送过来的一个连续字符流，浏览器并不关心 Web 服务器是如何创建出这个字符流的，它仅仅是把这个字符流当做一个网页文档的内容来处理。由于浏览器只关心如何显示和处理 Web 服务器所返回的内容，它并不知道返回的内容在服务器端是如何产生的，它只知道从 Web 服务器端获得了一个网页文档，所以，浏览器处理动态网页的方式与它处理静态网页的方式完全相同、并无差异。

Web 服务器传送给浏览器的字符流内容可以从某个 HTML 文件中原封不动地读取，

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

也可以由一个程序动态创建。如果 Web 服务器传送给浏览器的内容是在用户访问时由程序临时动态产生的，那么这些内容就可以随时变化，从而让用户浏览到一些实时动态变化的信息。一般来说，Web 服务器发送的静态内容直接来自于文本或数据文件，例如 HTML 或 JPEG 文件，而动态内容需要依靠一个程序来临时产生。

指点迷津

区分动态网页与动态 HTML 页面的概念

要注意区分 Web 服务器端创建的动态网页与使用客户端脚本程序实现的具有动态视觉效果的网页及 Flash 动画网页的区别。虽然后两者在浏览器中也可以显示出视觉上的动态效果，并能与用户交互，但这种动态视觉效果是浏览器执行的结果，它并不是网页的源文件内容改变后的结果。例如，例程 4-1 是一个利用 JavaScript 代码来产生动态 HTML 效果的页面。

例程 4-1　dynamic.html

```
现在的时间是:
<script type="text/javascript">
document.write(new Date());
</script>
```

将 dynamic.html 文件发布到 Web 服务器上，然后用浏览器访问这个文件，每隔一段时间刷新一下访问，可以看到浏览器窗口中显示的时间都在不断改变。但是，IE 浏览器每次从 Web 服务器上获得这个页面后，单击浏览器上的“查看”→“源文件”菜单，看到的网页源文件内容总是一成不变的，这种情况就不属于动态网页。

这里所说的动态网页，是指 Web 服务器在不同时刻和不同的条件下返回给客户端的网页文档内容会发生变化的网页。对于动态网页，浏览器每次从 Web 服务器上获得这个页面后，单击 IE 浏览器上的“查看”→“源文件”菜单，看到的网页源文件内容是有可能变化的。

4.1.2　动态网页程序与引擎

Web 服务器的作用在于实时接收浏览器的请求，并将响应结果回送给浏览器。对于静态 HTML 文件内容的访问请求，Web 服务器可以直接从文件系统中读取文件内容后返回给浏览器。但对于要根据某些条件去动态创建内容的访问请求，Web 服务器本身不具备处理这种请求的能力，而是需要专门的 Web 服务器程序模块来处理。因此，动态网页内容的创建过程包含两个步骤：首先使用某种编程语言编写出相应的动态网页程序；然后由一个专门的 Web 服务器程序模块来解释执行该动态网页程序。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

动态网页程序可以完成各种程序逻辑控制、获取客户端传递的数据、对数据库进行操作、创建输出内容等功能，Web 程序开发就是指使用某种编程语言来编写完成上述功能的动态网页程序。动态网页程序需要有一个专门的 Web 服务器程序模块来解释执行，专门解释和执行某种动态网页程序的 Web 服务器程序模块习惯上被称为引擎，譬如 ASP 引擎、Servlet 引擎、JSP 引擎等等。

只有引擎才会与浏览器直接进行信息交换，动态网页程序不直接与浏览器进行信息交换，动态网页程序只与引擎进行信息交换，由引擎将浏览器的信息传递给动态网页程序，并将动态网页程序生成的结果回送给浏览器。引擎以 Web 服务器的扩展模块的形式提供，通常都是由那些提供动态网页解决方案的公司开发的。引擎要与动态网页程序进行通信，必须要提供一些 API（编程接口）给动态动态网页程序调用，这就好比电台总部要与某个人进行通信时，必须给那个人提供对讲机，引擎提供的 API 就相当于电台提供的对讲机，动态网页程序只有借助 API 才能与引擎之间交换信息。引擎提供的 API 有两个基本的作用：将访问请求的相关信息传递给动态网页程序，例如，访问者的 IP 地址和请求消息头；将动态网页程序产生的结果传递给引擎。浏览器、引擎、API 和动态网页程序之间的工作关系如图 4.1 所示。

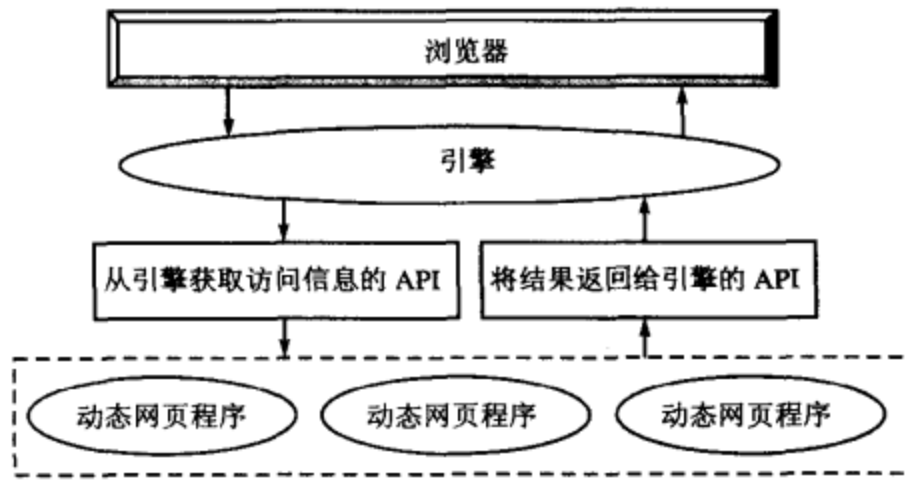


图 4.1

客户端向 Web 服务器请求的 URL 资源路径，不仅可以指向服务器端的一个静态 HTML 文件，还可以指向服务器端的某种可被解释运行的程序文件。如果客户端请求的 URL 后缀名为.htm 或.html，通常由 Web 服务器直接在本地文件系统中查找该 URL 所映射的 HTML 文件，并从该文件中读取内容后原封不动地传递到客户端，这样产生的网页文档内容总是相同的。如果客户端请求的 URL 后缀名不是.htm 或.html，而是符合某种引擎所处理的资源名的特征，Web 服务器将把请求转交给相应的引擎去处理，然后由引擎去调用某个程序文件，并将程序动态产生的内容返回给客户端。

事实上，如果 Web 服务器将浏览器的请求交给一个程序文件来处理，这个程序文件中的代码不仅可以创建出用于返回给浏览器的动态变化的内容，还可以实现对数据库的插入、修改、删除和查询等操作，从而让网站可以实现论坛交流、产品订购、问卷调查，甚至是企业的整个管理信息系统等功能。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

目前有不少公司提供了动态网页的解决方案，称之服务器端的 Web 编程技术，常见的技术有 CGI、ISAPI、ASP、ASP.NET、Servlet/JSP、PHP 等，这些技术分别使用不同的编程语言，本书主要讲解基于 Java 语言的 Servlet/JSP 技术。

4.1.3 Servlet 技术简介

Servlet 技术是 Sun 公司提供的一种实现动态网页的解决方案，它是基于 Java 编程语言的 Web 服务器端编程技术，主要用于在 Web 服务器端获得客户端的访问请求信息和动态生成对客户端的响应消息。Servlet 技术也是 JSP 技术（另外一种动态网页开发技术）的基础。**一个 Servlet 程序就是一个实现了特殊接口的 Java 类，它由支持 Servlet 的 Web 服务器（具有 Servlet 引擎）调用和启动运行。一个 Servlet 程序负责处理它所对应的一个或一组 URL 地址的访问请求，并用于接收客户端发出的访问请求信息和产生响应内容。**

Applet 是用于浏览器端的 Java 小程序，在浏览器端被解释执行，用于在 HTML 网页中实现一些桌面应用程序的功能，称为“小应用程序”。Servlet 是用于 Web 服务器端的 Java 小程序，它在 Web 服务器端被解释执行，用于处理客户端的请求和产生动态网页内容。源于 Applet 的命名，这种 Web 服务器端的 Java 小程序就被命名为了 Servlet。与 Applet 相对应，Servlet 可以被称之为“小服务程序”。

Servlet 与普通 Java 程序相比，只是输入信息的来源和输出结果的目标不一样，所以，普通 Java 程序所能完成的大多数任务，Servlet 程序都可以完成。Servlet 程序具有如下的一些基本功能：

- 获取客户端通过 HTML 的 FORM 表单递交的数据和 URL 后面的参数信息。
- 创建对客户端的响应消息内容。
- 访问服务器端的文件系统。
- 连接数据库并开发基于数据库的应用。
- 调用其他 Java 类。

4.1.4 B/S 系统架构与 C/S 系统架构

企业信息管理软件系统的一个重要特点就是：要使用多台计算机作为操作终端对系统进行操作，而系统的数据信息集中存放在网络上的数据库服务器中。在开发企业信息管理软件时，人们通常需要在两种系统架构之间进行选择，即 C/S 架构和 B/S 架构。C/S 架构是 Client/Server 的简写，也就是客户机/服务器架构，它是早期出现的一种分布式架构。B/S 架构是 Browser/Server 的简写，也就是浏览器/服务器架构，它是随着 Internet 技术的兴起，对 C/S 架构的一种变化和改进的架构。

1. C/S 系统架构

最简单的 C/S 架构只采用了两层软件组件：一层就是客户端程序；另外一层就是服务器端的数据库。对于 Java 语言来说，如果编写一个 AWT 或 Swing 的 GUI 程序，这个程序

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

要处理用户的数据输入，并对用户输入的数据进行格式检查后插入一个服务器数据库中，这个程序还要从数据库中检索数据，并对检索到的数据进行各种业务逻辑处理或者采用各种方式进行显示，那么，这个 GUI 程序和服务器数据库就组成了一个典型的 C/S 架构的系统。作为客户端的 GUI 程序可以安装在多台客户机上运行，作为服务器的数据库系统则安装在一台服务器计算机上运行，多个客户端程序对同一个数据库服务器进行访问，采用两层软件组件的 C/S 架构如图 4.2 所示。

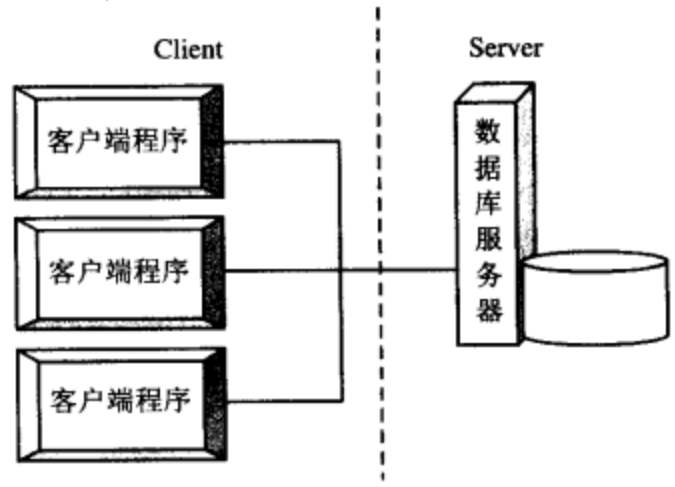


图 4.2

采用两层软件组件的 C/S 架构有如下一些缺陷：

（1）如果每个客户端程序都与数据库保持持久的连接，而数据库系统支持的并发连接数有限，那么这就限制了同时运行的客户端程序的数目。如果考虑到客户端程序并不是一直在与数据库系统进行交互，它们的交互过程相对于客户端程序的整个运行过程的比率并不高，因此将客户端程序设计成不要与数据库保持持久连接，而是在每次要传输数据时才建立连接，数据传输完毕后立即断开连接，这样可以实现多个客户端程序与数据库轮流建立连接，从而增加同时运行的客户端程序的数目。但是这种方式给程序的事务处理带来极大困难，并且客户端每次与数据库建立连接的过程需要相对较长的时间，所以，这种方式并不适合于需要频繁与数据库进行交互的客户端程序，而只适合于偶尔与数据库交互一下的 Web 应用。

（2）业务逻辑处理和界面显示都由客户端程序负责处理，没有对业务逻辑和显示界面进行分离，一旦业务逻辑或者显示界面要发生变化，则需要对整个客户端程序进行修改，不利于软件的维护和功能扩展。

（3）由于在每个客户机上都需要安装客户端程序，对于大一点的企业来说，如果客户端程序发生了修改，则需要系统管理人员到几百甚至上千台客户机上去重新安装客户端程序，这种工作量之大和效率之低是可想而知的。

笔者曾经遇到过一个在某总公司的信息管理部门工作的学员，该总公司开发了一套汇总和统计 50 多个分公司销售情况的 C/S 架构的软件系统，各个分公司需要通过系统的客户端程序将他们每天的销售信息报告给总公司进行汇总和统计，这个学员的工作就是维护这套系统。一个系统刚推出时不可能完美无缺，特别是在试运行期间经常都会发现一些问

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

题和需要进行修正，一旦客户端程序进行了修正，他就需要亲自上门去给每个分公司安装新版本的客户端程序，在测试和确认客户端程序可以正常工作后才可离开。虽然总公司和所有分公司都在同一个城市，但这个学员每天只能骑着自行车去为各个分公司安装客户端程序，即使平均一天跑两家分公司，要想让所有分公司都安装上新版本的客户端程序，至少也需要一个月的时间。由于这套系统还处于试运行期间，没过多久，客户端程序又发生了修改，这个学员又要去为各个分公司安装更新版本的客户端程序。另外，因为各个分公司人员的计算机应用水平比较低，如果客户端计算机出现了任何一点问题，他也需要上门去解决问题，特别是有时候根本就不是客户端程序的问题，而是分公司的操作人员使用不当和其他程序的问题。所有这些工作都令这个学员感到非常苦恼，还没等到这套系统完全通过测试和正式运行，他就辞职不干了，据说这套系统最后没有使用多久就被废弃了。

由于 C/S 架构中的客户端程序就是计算机上的一个桌面程序，因此它也存在着一些优点，功能可以做得非常强大，例如，在数据库服务器不可用时仍可以独立运行，可以将事务记录临时保存在本地桌面数据库中，然后再集中提交给数据库服务器，很容易对数据库中的数据进行实时监控和实现报警功能。

2. B/S 系统架构

B/S 架构与 C/S 架构最大的区别就是将软件系统的用户操作界面以网页的形式在客户机的 Web 浏览器中展现给用户，在客户端计算机上不需要安装应用软件的客户端程序，而 Web 浏览器则是每台客户机的操作系统自带的工具软件，也不需要单独安装。通俗地讲，B/S 架构就是一个能通过浏览器与之进行交互的网站系统，通过该网站系统可以实现企业的信息管理。最简单的 B/S 架构如图 4.3 所示。

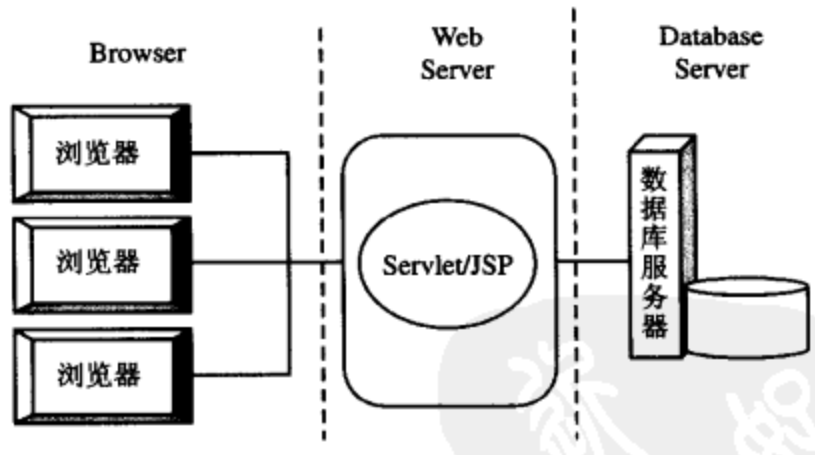


图 4.3

B/S 架构将程序中的界面显示和业务逻辑处理都移动到了 Web 服务器中来实现，其应用全部集中到了 Web 服务器端，客户端只需要具有浏览器就可以作为 B/S 架构的操作终端，而不用安装和部署任何程序。

因为 B/S 架构中的用户操作界面是由 Web 服务器创建的，当要修改系统提供的用户操作界面时，只需要在 Web 服务器端修改相应的网页文档，用户再次使用浏览器访问系统时，所得到的操作界面就自动随之发生了更改，所以，只要在 Web 服务器端这一个地方进行更

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

新即可完成整个系统的更新。整个系统的更新部署不需要在客户机上进行任何操作和设置，在用户的不知不觉中就迅速完成，可以做到快速服务响应。另外，由于客户端不直接与数据库建立连接，而是只有 Web 服务器端的程序需要与数据库建立连接，所以数据库并发连接数量有限的问题也得到了解决。

因为 B/S 架构相对于 C/S 架构有诸多优点，所以，B/S 架构应作为各类信息管理系统的首选体系架构，它基本上已全面取代了 C/S 架构。Servlet 和 JSP 就是使用 Java 语言开发 B/S 架构的应用软件的核心技术，随着 B/S 架构的流行，Servlet 和 JSP 的应用已经非常广泛。

4.1.5 Servlet 开发涉及的相关知识

（1）Servlet 是 Java 程序语言的一种应用，在深入学习本书之前，读者必须具备良好的 Java 编程基础和排查程序错误的基本能力。本书不讲解 Java 的基础语法，没有 Java 编程基础的读者可以学习笔者编写的《Java 就业培训教程》一书。

（2）Servlet 是用于开发 Web 站点的技术，除了要了解本书前面所讲的 XML、HTTP 协议和 Web 站点管理方面的知识外，读者还应对 HTML 语言、CSS 和 JavaScript 有所了解，对于这些内容，读者可以参看笔者编写的《JavaScript 网页开发——体验式学习教程》一书。

（3）几乎所有支持动态网页功能的 Web 站点都是与数据库系统紧密相连的，例如，动态网页中的一些信息需要从数据库中检索，HTML 的 FORM 表单提交的数据需要保存在数据库中。所以，绝大多数要使用 Servlet 与 JSP 从事实际项目开发的人员，需要有一定的数据库基础知识和 Java 数据库编程知识，其中熟练掌握各种 SQL 语句是至关重要的，能对所使用的数据库系统进行基本的管理和操作也是必要的。

4.2 Servlet 开发入门

4.2.1 Servlet API 介绍

Sun 公司定义了一套专门用于开发 Servlet 程序的 Java 类和接口，这些类和接口提供 Servlet 程序开发中所涉及的各种功能，它们统称为 Servlet API（Servlet Application Programming Interface）。Servlet 引擎与 Servlet 程序之间采用 Servlet API 进行通信，因此，Servlet 引擎与 Servlet 程序都需要用到 Servlet API，事实上，一个 Servlet 程序就是一个在 Web 服务器端运行的调用了 Servlet API 的 Java 类。

早期包装 Servlet API 的 Jar 包被称之为 JSDK（Java Servlet Development Kit，即 Java Servlet 开发工具包），最新版本的 Java Servlet 开发工具包已经被集成到了 Sun 公司的 Java EE（J2EE 升级后的名称，即 Java 企业级版本）开发工具包中，这些开发工具包都可以从 http://java.sun.com 站点上下载到。由于支持 Servlet 的 Web 服务器软件都会自带 Servlet API 的 Jar 包，所以一般不用去专门下载 Java Servlet 开发工具包。各种支持 Java Servlet 的 Web 服务器所提供的用于包装 Servlet API 的 Jar 包的文件名称可能各不一样，例如，在 Tomcat 4.x 中，其名称为<tomcat 的安装目录>\common\lib\servlet.jar，但在 Tomcat 5.x 中，其名称

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

就变成了<tomcat 的安装目录>\common\lib\servlet-api.jar。

Tomcat 中也附带 Sun 公司提供的 Servlet API 帮助文档，与学习和使用其他 API 的方式一样，读者应结合 Servlet API 帮助文档来学习和应用 Servlet API，边学边查、边用边查。虽然有了 Servlet API 帮助文档可以参考，但是，仅仅是能够看懂帮助文档中的英文单词是没有多大作用的，读者必须了解 HTTP 协议和 Web 服务器的内部运行方式，以及 Servlet 引擎的一些实现机制后，才能够在看懂帮助文档中的英文单词的基础上去了解和掌握一些 Servlet API 的用途与用法，才能够使用 Servlet 来实现一些特殊的 Web 功能。

查看 Tomcat 提供的 Servlet API 帮助文档

01 启动 Tomcat 4.1.31 服务器程序后，使用 IE 浏览器进入 Tomcat 安装的默认站点首页。

02 在默认站点首页左侧的链接导航列表中，单击 Tomcat Documentation 链接，进入 tomcat-docs 子目录的默认页面。

03 在 tomcat-docs 子目录的默认页面左侧的链接导航列表中，单击 Servlet/JSP Javadocs 链接，进入 Servlet API 和 JSP API 的帮助文档的起始页面，如图 4.4 所示。

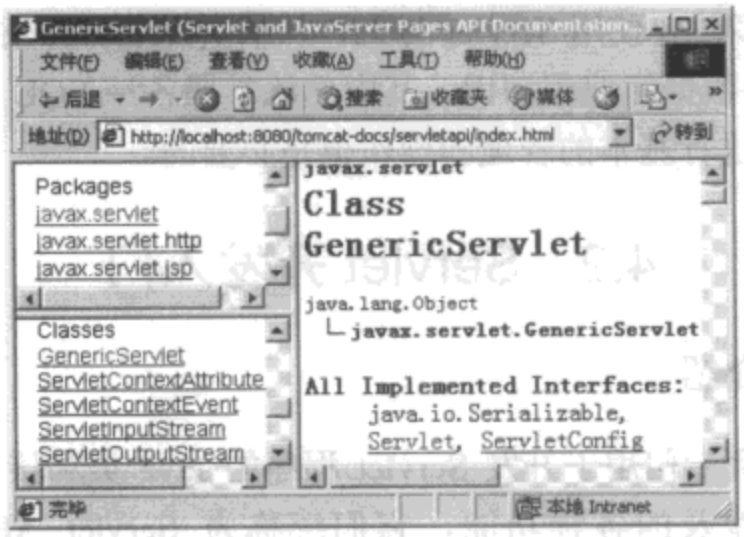


图 4.4

在图 4.4 所示的 Servlet API 帮助文档中，可以查看到所有 Servlet API 类及其方法的用途与用法，一些通用的 Servlet API 类和接口包含在 javax.servlet 包中，专用于 HTTP 协议的 Servlet API 类和接口包含在 javax.servlet.http 包中。在图 4.4 中，还列出了 javax.servlet.jsp 与 javax.servlet.jsp.tagext 这两个包，这两个包中包含了 JSP 相关的 API。

04 找到图 4.4 所示的页面在本地文件系统中的位置，然后用浏览器直接打开它，这样，不用启动 Tomcat Web 服务器程序就可以查看帮助文档。如果要经常使用这个帮助文档，建议读者在桌面创建帮助文档起始页面文件的快捷方式。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

4.2.2 编写与编译 Servlet 程序

一个 Servlet 程序就是一个在 Web 服务器端运行的特殊 Java 类，这个特殊的 Java 类必须实现 javax.servlet.Servlet 接口，Servlet 接口定义了 Servlet 引擎与 Servlet 程序之间通信的协议约定。为了简化 Servlet 程序的编写，Servlet API 中也提供了一个实现 Servlet 接口的最简单的 Servlet 类，其完整名称为 javax.servlet.GenericServlet，这个类实现了 Servlet 程序的基本特征和功能。

Servlet API 中还提供了一个专用于 HTTP 协议的 Servlet 类，其名称是 javax.servlet.http.HttpServlet，它是 GenericServlet 的子类，在 GenericServlet 类的基础上进行了一些针对 HTTP 特点的扩充。显然，一个 Java 类只要继承了 GenericServlet 或 HttpServlet，它就是一个 Servlet。反过来说，要编写一个 Servlet 类，这个类只要继承 GenericServlet 类或 HttpServlet 类就可以了。为了充分利用 HTTP 协议的功能，在一般情况下，都应让自己编写的 Servlet 类继承 HttpServlet 类，而不是继承 GenericServlet 类。

查看 HttpServlet 类的帮助文档，可以看到其中有一个名为 service 的方法，当客户端每次访问一个 Servlet 程序时，Servlet 引擎都将调用这个方法来进行处理，我们自己编写的 Servlet 程序通常只需要在 HttpServlet 类的基础上覆盖这个方法。service 方法接受两个参数：一个是用于封装 HTTP 请求消息的对象，其类型为 HttpServletRequest；另一个是代表 HTTP 响应消息的对象，其类型为 HttpServletResponse。调用 HttpServletResponse 对象的 getWriter 方法可以获得一个文本输出流对象，向这个流对象中写入的数据将作为 HTTP 响应消息的实体内容部分发送给客户端。

指点迷津

如何快速入门新技术

软件开发人员在学习任何一门与编程相关的新技术时，在初始阶段需要先对该新技术的运用原理和理论有一定的认识和了解，然后就应该通过一个相关的最简单的应用例子，完全用手工的方式把程序的编写、编译、配置和运行等过程完整地走一遍，从而了解这门新技术的开发和运用流程。只要有了一个简单程序的开发和运用经验后，就算对这门新技术入门了，以后就很容易在这个简单程序的基础上来逐步学习和体验更多的技术细节与复杂功能。

对一门新技术的入门是最关键、也是最令人感到棘手的阶段，许多人在这一阶段往往都不知所措和不知从哪里下手。其实，任何一门新技术的开发厂家都会提供一些针对这门技术的应用例子，其中往往都包含有一个最简单的入门例子，以帮助初学者入门这门技术。初学者只要找到这个用于入门的例子程序，然后参照这个例子程序的源代码和配置信息，自己编写一个同样简单的程序，再把该程序的编译、配置和运行等过程完整

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

地走一遍，就可以快速入门了。

完全用手工的方式把程序的编写、编译、配置和运行等过程完整地走一遍，其目的是为了体验程序的开发和运用流程，但这并不要求手工逐一输入源程序和配置文件中的每个字符，而是可以在工作每进行到一个小的步骤时，都去例子程序中复制相应的内容后再进行修改。以 Servlet 技术为例，在 Tomcat 4.x 中就提供了的一些 Servlet 例子程序，它们存储在<tomcat 的安装目录>\Webapps\examples\Web-INF\classes 目录中，其中最简单的例子为 HelloWorldExample.java 文件，在编写下面的第一个 Servlet 程序时，最好的方式是从 HelloWorldExample.java 文件中进行复制和修改。

编写与编译 Servlet 程序的过程

01 编写一个继承了 HttpServlet 类的 HelloServlet 类，它对 HttpServlet 类中的 service 方法进行了覆盖，如例程 4-2 所示。

例程 4-2 HelloServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloServlet extends HttpServlet
{
    //从 HttpServlet 类的帮助文档中复制 service 方法的声明部分可避免书写错误
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException
    {
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<font size=30 color=red>www.it315.org</font><br>");
        out.println("<marquee>" + new java.util.Date() + "</marquee>");
        out.println("</html>");
    }
}
```

02 使用 javac 命令编译这个源文件，通常情况下，第一次学习编写 Servlet 程序的人都会遇到类似如下错误：

```
HelloWorld.java:2: package javax.servlet does not exist
import javax.servlet.*;
^
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
HelloWorld.java:3: package javax.servlet.http does not exist
import javax.servlet.http.*;
^
......
```

这些错误信息提示 javac 编译器找不到 javax.servlet 和 javax.servlet.http 这两个包以及其中的一些 Servlet API 类，这是因为包含 Servlet API 类的 jar 文件还没有被加入到 javac 所使用的 CLASSPATH 环境变量中。

03 在执行 javac 的命令行窗口中，将包含有 Servlet API 的 jar 文件加入到 CLASSPATH 环境变量的路径列表中，例如，对于 Tomcat 4.x，应使用如下的命令设置：

```
set CLASSPATH=<tomcat 的安装目录>\common\lib\servlet.jar;%CLASSPATH%
```

设置完 CLASSPATH 环境变量后，重新编译 HelloServlet.java 文件，如果源程序中没有书写错误的话，这时候就可以编译成功了。

笔者在《Java 就业培训教程》中曾经讲过，在命令行窗口中设置的 CLASSPATH 环境变量是临时的，只对这个命令行窗口起作用，对其他窗口和应用程序不起作用，因此在每次新启动的命令行窗口中都要重新设置。如果想让设置一直保持有效，应该使用对整个系统都有效的环境变量的设置方法，例如，在 Windows 2000 的系统特性的“环境变量”对话框中进行设置，在 Windows 98 的开机自动批处理文件 Autoexec.bat 中设置，在 Linux 的 Profile 文件中设置。

4.2.3 Servlet 的注册与运行

HTML 文件可以直接用浏览器打开并查看运行效果，但是，Servlet 程序必须通过 Web 服务器和 Servlet 引擎来启动运行。Servlet 程序的存储目录有特殊要求，通常需要存储在<Web 应用程序目录>\Web-INF\classes\目录中。另外，Servlet 程序必须在 Web 应用程序的 Web.xml 文件中进行注册和映射其访问路径，才可以被 Servlet 引擎加载和被外界访问。

1. 注册 Servlet

在 Web.xml 文件中，一个<servlet>元素用于注册一个 Servlet，<servlet>元素中包含有两个主要的子元素：<servlet-name>和<servlet-class>，它们分别用于设置 Servlet 的注册名称和指定 Servlet 的完整类名，如下所示：

```
<servlet>
    <servlet-name>AnyName</servlet-name>
    <servlet-class>HelloServlet</servlet-class>
</servlet>
```

其中，<servlet-class>子元素指定了当前注册的 Servlet 的完整类名为 HelloServlet，<servlet-name>子元素指定了 HelloServlet 的注册名为 AnyName。Servlet 的注册名可以是一

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

个任意的合法友好的名称，但不能与其他 Servlet 的注册名相冲突。如果 HelloServlet 是在包 org.it315 中定义的，那么它在“d\myWeb\Web-INF\classes\”目录下要有与包名相对应的目录结构存储，即它的完整文件路径应该为“d\myWeb\Web-INF\classes\org\it315\HelloServlet.class”，同时，<servlet-class>元素的设置也应该修改成如下形式：

```
<servlet-class>org.it315.HelloServlet</servlet-class>
```

2. 映射 Servlet

在 Web.xml 文件中，一个<servlet-mapping>元素用于映射一个已注册的 Servlet 的一个对外访问路径，客户端将使用 Servlet 所映射的对外访问路径来访问 Servlet，而不是使用 Servlet 名称来访问 Servlet。<servlet-mapping >元素中包含有两个子元素：<servlet-name>和<url-pattern>，它们分别用于指定 Servlet 的注册名称和设置 Servlet 的对外访问路径，如下所示：

```
<servlet-mapping>
    <servlet-name>AnyName</servlet-name>
    <url-pattern>/demo/hello.html</url-pattern>
</servlet-mapping>
```

其中，<servlet-name>子元素的设置值是在前面某个<servlet>元素的<servlet-name>子元素中设置的某个 Servlet 的注册名，<url-pattern>子元素中的访问路径必须以正斜杠（/）开头，这个正斜杠（/）表示当前 Web 应用程序的根目录，而不是整个 Web 站点的根目录。路径名可以是多级目录的形式，例如上面的“/demo/hello.html”。当客户端访问地址“http://localhost:8080/it315/demo/hello.html”时，即使/it315 这个 Web 应用程序所映射的本地文件系统目录“d:\myWeb”中存在一个“demo\hello.html”文件，Tomcat 也首先查找/it315 这个 Web 应用程序中是否存在映射到“/demo/hello.html”路径上的 Servlet，如果存在，则调用相应的 Servlet 进行响应，只有不存在与该路径匹配的 Servlet 映射时，Tomcat 才会读取“d:\myWeb\demo\hello.html”文件内容来进行响应。

同一个 Servlet 可以被映射到多个 URL 上，即多个<servlet-mapping>元素的<servlet-name>子元素的设置值可以是同一个 Servlet 的注册名。在 Servlet 映射到的 URL 中也可以使用*通配符，但是只能有两种固定的格式：一种格式是“*.扩展名”，在*前面不能有目录分隔符“/”，例如，“*.do”表示匹配以“.do”结尾的所有 URL；另一种格式是以正斜杠（/）开头并以“/*”结尾，例如，“/*”表示匹配当前 Web 应用程序下的所有 URL，“/action/*”表示匹配当前 Web 应用程序下的“/action”子路径下的所有 URL。

当 Servlet 引擎接收到一个访问请求时，它将采用最具体匹配原则来查找请求 URL 的映射，其中，“*.扩展名”的匹配优先级最低。例如，对于如下的一些映射关系：

- /abc/*　映射到　Servlet1
- /*　映射到　Servlet2
- /abc　映射到　Servlet3
- *.do　映射到　Servlet4

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

将发生如下一些行为：

- 当请求 URL 为“/abc/a.html”，“/abc/*”和“/*”都可以匹配这个 URL，但 Servlet 引擎将调用 Servlet1。
- 当请求 URL 为“/abc”时，“/abc/*”和“/abc” 都可以匹配这个 URL，但 Servlet 引擎将调用 Servlet3。
- 当请求 URL 为“/abc/a.do”时，“/abc/*”和“*.do”都可以匹配这个 URL，但 Servlet 引擎将调用 Servlet1。
- 当请求 URL 为“/a.do”时，“/*”和“*.do”都可以匹配这个 URL，但 Servlet 引擎将调用 Servlet2。
- 当请求 URL 为“/xxx/yyy/a.do”时，“/*”和“*.do”都可以匹配这个 URL，但 Servlet 引擎将调用 Servlet2。

注册与访问 Servlet 程序

下面把上面编写和编译的 HelloServlet 程序发布到 2.4.3 节中创建的/it315 虚拟目录中，并通过一系列的实验步骤来讲解 Servlet 程序的配置和运行。

01 在/it315 虚拟目录所映射的本地文件系统目录 d:\myWeb 中创建一个名称为 Web-INF 的子目录，然后在该子目录中创建一个名称为 classes 的子目录，将 HelloServlet.class 文件复制到 classes 目录中。

02 在 d:\myWeb\Web-INF 目录中创建一个 Web.xml 文件，并在该文件中注册 HelloServlet 和映射其访问路径，内容如下：

```
<Web-app>
    <servlet>
        <servlet-name>
            AnyName
        </servlet-name>
        <servlet-class>
            HelloServlet
        </servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>
            AnyName
        </servlet-name>
        <url-pattern>
            /demo/hello.html
        </url-pattern>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
    </servlet-mapping>
</Web-app>
```

照猫画虎

如何编写 Web.xml 文件

一个快速有效的学习方法就是先参看别人的例子，进行模仿。在编写上面的 Web.xml 文件时，可以参照 Tomcat 自带的 examples 应用程序中的 Web.xml 文件的相应部分。为减少一些手误，最好是使用“复制”和“粘贴”的方式来复制一些同类性质的内容，然后对复制的内容进行适当的修改。如果是在一个原有许多内容的 Web.xml 文件中插入新的内容，要注意插入的位置，不能破坏其他标签的完整性和各个标签出现的层次关系和先后顺序，应该遵循 Web.xml 采用的 Servlet 规范的版本。一般来说，只要找到与所要插入的内容有同类性质的段落，在这个段落的前后进行插入就不会有太大问题。如果原来的 Web.xml 文件中没有与所要插入的内容具有同类性质的部分，那么可以参考 Tomcat 自带的一些 Web.xml 配置文件，通过对比来确定所要插入内容的位置。

03 为了让 Web.xml 文件中新增或修改的内容发生作用，重新启动 Tomcat。在浏览器地址栏中输入 http://localhost:8080/it315/demo/hello.html，浏览器中将显示出 HelloServlet 的响应结果，如图 4.5 所示。

图 4.5

如果看不到正确的访问结果，在上面的操作中应注意以下三个问题：

- 确保虚拟目录的配置正确，这可以通过在虚拟目录中创建一个简单的 HTML 文件来检验。例如，在/it315 下面已有一个 test.html 文件，如果无法访问 http://127.0.0.1:8080/it315/test.html，则说明虚拟目录的设置出了问题。
- Tomcat 本身的 Servlet 引擎是否有问题，可以通过访问 Tomcat 自带的 Servlet 例子程序进行判断，如果这些例子程序都无法正常访问，那么就应考虑重装 Tomcat。
- 确保 URL 地址中的每个字母的大小写严格一致，弄错一个字符的大小写都会导致浏览器报告找不到请求资源的错误。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

每隔一段时间刷新一下浏览器的访问，浏览器中每次显示的时间值都不相同，每次访问后单击浏览器上的“查看”→“源文件”菜单，每次看到的网页源文件内容也都不相同，这种情况就属于服务器端程序创建出的动态网页效果。

04 在上面创建的 Web.xml 文件中的<servlet-mapping>元素下面再增加两个内容如下的<servlet-mapping>元素：

```
<servlet-mapping>
    <servlet-name>
        AnyName
    </servlet-name>
    <url-pattern>
        /action/*
    </url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>
        AnyName
    </servlet-name>
    <url-pattern>
        *.do
    </url-pattern>
</servlet-mapping>
```

重新启动 Tomcat 后，在浏览器地址栏中分别输入 http://localhost:8080/it315/ test.do 和 http://localhost:8080/it315/action/test.html，浏览器中都能显示出 HelloServlet 的响应结果。如果将上面的内容为“/action/*”的<url-pattern>子元素中的内容修改为“/action/*.xx”，重新启动 Tomcat 后，在浏览器地址栏中输入 http://localhost:8080/it315/action/test.xx，浏览器将提示找不到资源的错误信息，这是因为在“*.扩展名”格式中的“*”前面不能有目录分隔符“/”，所以“/action/*.xx”是一个无效的映射路径。

05 Servlet 程序不能被设置为 Web 应用程序的目录默认网页文档。由于目录默认网页文档是对各级目录都有效的，所以，在<welcome-file>元素中设置的默认网页文档名称中不能出现目录分隔符“/”，例如，将“/demo/hello.html”设置为目录默认网页文档显然是不合理的。由于 HelloServlet 有一个形式为“*.do”的 URL 映射，我们不妨将 test.do 设置为目录默认网页文档，查看访问/it315 根目录时 Tomcat 能否调用 HelloServlet 进行响应。在上面创建的 Web.xml 文件中的最后一个<servlet-mapping>元素下面增加如下一段内容：

```
<welcome-file-list>
    <welcome-file>test.do</welcome-file>
</welcome-file-list>
```

重新启动 Tomcat 后，在浏览器地址栏中分别输入 http://localhost:8080/it315，浏览器中没有显示出 HelloServlet 的响应结果，而是显示出了/it315 目录的内容列表。可见，Servlet

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

程序不可能被设置为 Web 应用程序的目录默认网页文档。

4.2.4 Servlet 激活器

在第 3 章的实验中已经访问过 Tomcat 提供的一些 Servlet 例子程序，它们存储在<tomcat 的安装目录>\Webapps\examples\Web-INF\classes 目录中，但是，打开 examples 这个 Web 应用程序的 Web.xml 文件（即<tomcat 的安装目录>\Webapps\examples\Web-INF\Web.xml），在其中根本就找不到这些例子 Servlet 的注册信息，那么，客户端是以什么样的路径来访问这些 Servlet 例子程序的呢？Servlet 引擎又是如何处理针对这些例子 Servlet 的访问请求的呢？

只要在图 3.9 所示的页面中，将鼠标移动到某个 Servlet 例子的 Execute 超链接上，在浏览器底部的状态栏上就会显示出这个超链接的目标 URL，这就是访问该 Servlet 时的完整 URL。在浏览器窗口中用鼠标右键单击某个超链接，然后单击弹出快捷菜单中的“属性”菜单项，在打开的属性对话框中也可以看到和复制这个超链接的目标 URL。按照这种方式查看到的 Tomcat 提供的第一个标题为 Hello World 的 Servlet 例子程序的 URL 如下：

```
http://localhost:8080/examples/servlet/HelloWorldExample
```

其中，URL 中最后一部分就是所要访问的 Servlet 名称。查看所有 Servlet 例子程序的访问路径，可以总结出它们的访问路径格式如下：

```
http://localhost:8080/examples/servlet/<Servlet 名称>
```

接着查看 examples 这个 Web 应用程序的 Web.xml 文件，可以发现其中包含如下一段内容：

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

这段内容说明：当访问/examples 应用程序下面的任何以“/servlet/”开头的资源名时，都将去调用一个注册名称为 invoker 的 Servlet。可是，在 examples 这个 Web 应用程序的 Web.xml 文件中根本就找不到将一个 Servlet 的名称注册为 invoker 的<servlet>元素设置，这是怎么回事呢？在本书第 2 章曾经讲过，一个 Web 应用程序的部署描述符除了包含该应用程序内部的 Web.xml 文件的设置信息外，还包含<Tomcat 主目录>/conf 目录下的 Web.xml 文件中的全局设置信息，打开<tomcat 的安装目录>\conf\Web.xml 文件，可以看到其中包含如下一段内容：

```
<servlet>
    <servlet-name>invoker</servlet-name>
    <servlet-class>
        org.apache.catalina.servlets.InvokerServlet
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
    </servlet-class>
    <load-on-startup>2</load-on-startup>
</servlet>
```

可见，注册名称为 invoker 的 Servlet 是在全局 Web.xml 文件中设置的，这样做的好处在于：如果有多个 Web 应用程序要使用这个 Servlet，我们不必在每个 Web 应用程序中都单独注册这个 Servlet，而只需在想使用这个 Servlet 的 Web 应用程序中映射其访问路径就可以了。注册名称为 invoker 的 Servlet 的作用就是去激活和调用任何其他 Servlet，故被称为 Servlet 激活器，它被设置为随 Tomcat 服务器起动时加载（通过<load-on-startup>元素）。由于 Servlet 激活器将把请求 URL 中的额外路径信息（映射 URL 中的“*”部分所匹配的内容）当作某个 Serlvet 的类名去进行调用，所以，在每个 Web 应用程序中为 Servlet 激活器所映射的访问路径可以各不相同，但是必须以“/*”结尾。经过这样的设置以后，即使某个 Servlet 程序没有在 Web.xml 文件中进行注册，我们只需要将 Servlet 激活器所映射的访问路径中的通配符（*）替换为这个 Servlet 的完整类名，就可以通过 Servlet 激活器来调用这个 Servlet 程序。

使用 Servlet 激活器

01 在/it315 应用程序的 Web.xml 文件中的合适位置处，增加下面的内容：

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

02 重新启动 Tomcat，在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/servlet/HelloServlet
```

浏览器中显示出了 HelloServlet 的响应结果。

为了节约篇幅，本章后面部分所讲解的 Servlet 程序基本上都采用 Servlet 激活器方式进行访问和调用。

4.2.5 缺省 Servlet

如果某个 Servlet 的映射路径仅仅为一个正斜杠（/），那么这个 Servlet 就成为当前 Web 应用程序的缺省 Servlet。凡是在 Web.xml 文件中找不到匹配的<servlet-mapping>元素的 URL，它们的访问请求都将交给缺省 Servlet 处理，也就是说，缺省 Servlet 用于处理所有其他 Servlet 都不处理的访问请求。在<tomcat 的安装目录>\conf\Web.xml 文件中，注册了一个名称为 org.apache.catalina.servlets.DefaultServlet 的 Servlet，并将这个 Servlet 设置为了

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

缺省 Servlet。由于<tomcat 的安装目录>\conf\Web.xml 文件的设置信息对该服务器上的所有 Web 应用程序都起作用，所以，服务器上的所有 Web 应用程序的缺省 Servlet 都是 org.apache.catalina.servlets.DefaultServlet。

当访问 Tomcat 服务器中的某个静态 HTML 文件和图片时，实际上是在访问这个缺省 Servlet，而这个缺省 Servlet 的处理方式通常是把静态资源中的内容按字节原封不动地读出来，然后再按字节流原封不动传递给客户端，并且生成一些响应消息头字段，例如，根据静态资源的扩展名所映射的 MIME 类型生成 Content-Type 头字段，根据静态资源的大小生成 Content-Length 头字段。

缺省 Servlet 的作用

01 在<tomcat 的安装目录>\conf\Web.xml 文件中注释掉设置缺省 Servlet 的<servlet-mapping>元素，如下所示：

```
<!--
<servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
-->
```

02 重新启动 Tomcat 后，用浏览器访问前面曾经成功访问过的静态 HTML 页面，这时浏览器再也无法得到正常的静态 HTML 页面内容了。但是，浏览器仍然可以成功访问映射到其他 Servlet 的 URL 地址，例如，在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/demo/hello.html
```

仍然可以获得 HelloServlet 的响应结果。

4.2.6 Servlet 的类装载器

1. 类装载器

Java 虚拟机使用每一个类的第一件事情就是将该类的字节码装载进来，装载类字节码的功能是由类装载器完成的，类装载器负责根据一个类的名称来定位和生成类的字节码数据后返回给 Java 虚拟机。最常见的类装载器是将要加载的类名转换成一个.class 文件名，然后从文件系统中找到该文件并读取其中的内容，这种类装载器也不是直接将.class 文件中的内容原封不动地返回给 Java 虚拟机，它需要将.class 文件中的内容转换成 Java 虚拟机使用的类字节码，譬如，Java 程序中的字符串编译成.class 文件后是以 UTF-8 编码存在的，而装载进 Java 虚拟机后要被转换成 Unicode 编码。类装载器本身也是一个 Java 类，Java

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

虚拟机也允许开发人员编写自己的类装载器，以便通过其他各种特殊方式来产生类字节码，例如，从数据库中装载并建造某个类的字节码，甚至根据一个文本文件中的描述信息来产生一个类的字节码。我们可以对一些.class 文件进行加密处理来防止反编译，但这就需要使用特殊的类装载器从已被加密处理的.class 文件中还原出正常的字节码内容。不管类装载器采用什么方式，只要能够在内存中制造出给 Java 虚拟机调用类字节码即可，所以，把类装载器描述为类字节码的制造器更容易让人理解。

当一个类被加载后，Java 虚拟机将其编译为可执行代码存储在内存中，并将索引信息存储进一个 HashTable 中，其索引关键字就是这个类的完整名称。当 Java 虚拟机需要用到某个类时，它先使用类名作为关键字在 HashTable 中查找相应的信息，如果该可执行代码已经存在，Java 虚拟机直接从内存里调用该可执行代码，反之则调用类装载器加载并进行加载和编译。

一个 Java 类用来描述现实生活中有形或无形的事物，甚至可以是一种思想，一种概念。不可否认，Java 程序中的类本身也是一种事物，它也可以用一个 Java 类描述，这个特殊的类名就叫 Class。Class 类用于描述 Java 程序语言中使用的一个类的有关信息，它里面定义了对其所描述的类进行各种操作的方法，例如，Class 类的 getName 方法可以获得它所描述的类的类名。可以认为，类装载器装载某个类的字节码的过程实际上就是在创建 Class 类的一个实例对象，这个 Class 类的实例对象封装的内容正好是当前加载的类的字节码数据，也就是 Java 虚拟机对当前加载类编译后存储在内存中的可执行代码。要想在程序中获得代表某个类的字节码数据的 Class 实例对象，可以采用下面三种方式：

- 类名.class，例如，System.class
- 对象.getClass()，例如，new Date().getClass()
- Class.forName("类名")，例如，Class.forName("java.util.Date");

类装载器本身也是一个 Java 类，Java 类库中提供了一个 java.lang.ClassLoader 来作为类装载器的基类，Java 虚拟机和程序都调用 ClassLoader 类的 loadClass 方法来加载类，ClassLoader 是一个抽象类，真正的类装载器必须是 ClassLoader 的子类。Class 类中定义了一个 getClassLoader 方法，用于返回它所描述的类的类装载器对象，这个返回对象的类型就是 ClassLoader。

在一个 Java 虚拟机中可以有多个类装载器，当 Java 虚拟机要装载一个类时，它通过以下一些方式来选择类装载器：

（1）一个类装载器本身也是一个 Java 类，所以，类装载器自身也需要被另外一个类装载器装载，这就出现了类似人类的第一位母亲是如何产生出来的问题，虽然人类的这个问题至今没有确切的答案，但 Java 中的类装载器的这个问题却很容易解决。Java 虚拟机中内嵌了一个称为 Bootstrap 的类装载器，它是用特定于操作系统的本地代码实现的，属于 Java 虚拟机的内核，这个 Bootstrap 类装载器不用专门的类装载器去进行装载。Bootstrap 类装载器负责加载 Java 核心包中的类（即 rt.jar 文件中的类），这些类的 Class.getClassLoader 方法返回值为 null，即表示是 Bootstrap 类装载器。Java 核心包中有另外两个类装载器：

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

ExtClassLoader 和 AppClassLoader，它们都是用 Java 语言编写的 Java 类，其中 ExtClassLoader 类装载器负责加载存放在<JAVA_HOME>/jre/lib/ext 目录下的 Jar 包中的类，AppClassLoader 负责加载应用程序的启动执行类，即当使用 java 命令去启动执行一个类时，Java 虚拟机使用 AppClassLoader 加载这个类。在编译和运行 Java 程序时，都会通过 ExtClassLoader 类装载器去<JDK 安装主目录>\jre\lib\ext 目录下的 Jar 包中搜索要加载的类，所以，如果将包含 Serlet API 的 jar 文件复制到该目录下，在编译 Servlet 程序时，就不必在 CLASSPATH 环境变量中增加包含 Serlet API 的 jar 文件。

（2）一个 Java 虚拟机中的所有类装载器采用具有父子关系的树形结构进行组织，在实例化每个类装载器对象时，需要为其指定一个父级类装载器对象。如果没有指定的话，则以 ClassLoader.getSystemClassLoader()方法返回的系统类装载器作为其父级类装载器对象。系统类装载器通常被设置为启动应用程序的 AppClassLoader，它是在 getSystemClassLoader()方法第一次被调用时设置的，而 getSystemClassLoader()方法的第一次调用发生在应用程序启动的早期阶段，可以通过 java.system.class.loader 系统属性来将系统类装载器设置为其他类装载器。ExtClassLoader 是 AppClassLoader 的父级类装载器，ExtClassLoader 没有父级类装载器。

每个 ClassLoader 本身只能分别加载特定位置和目录中的类，但是，ClassLoader 被设计成了一种委托模式，使得某一个 ClassLoader 可以委托它的父级类装载器去加载类，从而让应用程序可以借助某一个子级的 ClassLoader 去多个位置和目录中进行类的加载。这就好比“儿子”除了可以花自己的钱，他还可以花“父亲”的钱，“父亲”又可以花“父亲的父亲”的钱，所以，最终能通过“儿子”花出去的钱包括他历代前辈的钱。当要加载一个类时，调用的是 ClassLoader 的 loadClass 方法，loadClass 方法先查找这个类是否已被加载，如果没有加载则委托其父级类装载器去加载这个类，如果父级的类装载器无法装载这个类，子级类装载器才调用自己内部的 findClass 方法去进行真正的加载。父级类装载器调用 loadClass 方法去装载一个类时，它也是先查找其父级类装载器，这样一直追溯到没有父级的类装载器时（例如 ExtClassLoader），则使用 Java 虚拟机内嵌的 BootStrap 类装载器进行装载，当 BootStrap 无法加载当前所要加载的类时，然后才一级级回退到子孙类装载器去进行真正的加载。当回退到最初的类装载器时，如果它自己也不能完成类的装载，那就应报告 ClassNotFoundException 异常。一个类装载器不能委托它的子孙类装载器去加载类，这就好比只有“儿子”能花“父亲”的钱，“父亲”不能花“儿子”的钱，当“父亲”没钱时，即使“儿子”有钱，“父亲”最终还是没钱花。

一个类装载器只能创建某个类的一份字节码数据，即只能为某个类创建一个与之对应的 Class 实例对象，而不能为同样的一个类创建多个 Class 实例对象。在一个 Java 虚拟机中可以存在多个类装载器，每个类装载器都拥有自己的名称空间，对于同一个类，每个类装载器都可以创建出它的一个 Class 实例对象，即每个类装载器都可以分别创建出某个类的一份字节码数据。两个类装载器分别创建的同一个类的字节码数据属于两个完全不同的对象，相互之间没有任何关联，例如，在某个类中定义了一个静态成员变量，它在不同的

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

类装载器之间是不可以实现数据共享的。采用委托模式给类的加载管理带来了明显的好处，当父级的类装载器加载了某个类，那么子级的类装载器就不要再去加载这个类，这样就可以避免一个 Java 虚拟机中的多个类装载器为同一个类创建多份字节码数据的情况。只要开发人员自定义的类装载器不覆盖 ClassLoader 的 loadClass 方法，而是覆盖其 findClass 方法，这样就可以继续采用委托模式。

（3）如果在类 A 中使用 new 关键字创建类 B，Java 虚拟机将使用加载类 A 的类装载器来加载类 B。如果在一个类中调用 Class.forName 方法来动态加载另外一个类，可以通过传递给 Class.forName 方法的一个参数来指定另外那个类的类装载器，如果没有指定该参数，则使用加载当前类的类装载器。

读者应牢牢记住笔者总结出的下面一段话：依据一个类的存放位置，这个类最终只能由一个特定的类装载器装载。对于一个已被父级类装载器装载的类来说，Java 虚拟机默认也使用这个父级类装载器去装载它所调用的其他类，由于父级类装载器不会委托子级类装载器去装载类，所以，在一般情况下，一个已被父级类装载器装载的类无法调用那些只能被子级类装载器发现和装载的其他类。

（4）每个运行中的线程都有一个关联的上下文类装载器，可以使用 Thread.setContext ClassLoader()方法设置线程的上下文类装载器。每个线程默认的上下文类装载器是其父线程的上下文类装载器，而主线程的类装载器初始被设置为 ClassLoader.getSystem ClassLoader()方法返回的系统类装载器。当线程中运行的代码需要使用某个类时，它使用上下文类装载器来装载这个类，上下文类装载器首先会委托它的父级类装载器来装载这个类，如果父级的类装载器无法装载时，上下文类装载器才自己进行装载。

2. Tomcat 中的类装载器

当 Tomcat 4.x 启动后，它产生了一套类装载器，这些类装载器被组织成如图 4.6 所示的父子关系。

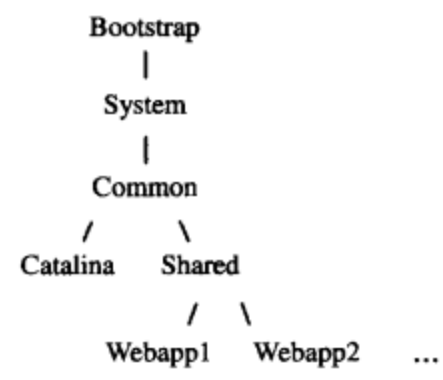


图 4.6

下面是每个类装载器的功能定义：

（1）这里的 Bootstrap 为 Java 虚拟机内嵌的类装载器与 ExtClassLoader 的总称，负责加载 Java 核心包中的类，以及存放在<JAVA_HOME>/jre/lib/ext 目录下的类。

（2）System 即系统类装载器，通常情况下就是 AppClassLoader，负责加载 CLASSPATH

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

环境变量设置的目录中的类。查看 Tomcat 4.x 的启动脚本（catalina.bat），它已将 CLASSPATH 环境变量原来的设置值完全清除，也就是说，Tomcat 4.x 不会继承操作系统上原来设置好的 CLASSPATH 环境变量的内容，而是将 CLASSPATH 环境变量重新设置为仅包含如下两个 Jar 包：

- *<CATALINA_HOME>/bin/bootstrap.jar*
- *<JAVA_HOME>/lib/tools.jar*

（3）Common 类装载器负责从<CATALINA_HOME>/common/classes 中的.class 类文件和<CATALINA_HOME>/common/lib 中的 Jar 包加载类。Common 类装载器加载的类对 Tomcat 服务器内核和每个 Web 应用程序都可见，例如，servlet.jar 中包含的类既要被 Tomcat 服务器内核使用，又要被每个 Web 应用程序使用，所以，它需要放置在 CATALINA_HOME/common/lib 中。

（4）Catalina 类装载器负责从<CATALINA_HOME>/server/classes 中的.class 类文件和<CATALINA_HOME>/server/lib 中的 Jar 包加载类。Catalina 类装载器加载的类只对 Tomcat 服务器内核可见，对每个 Web 应用程序完全不可见，所以，只想让 Tomcat 服务器内核使用而不想让 Web 应用程序使用的类应放置在 Catalina 类装载器的搜索目录中。对于运行 Tomcat 内核的线程，它的上下文类装载器就是 Catalina 类装载器。

（5）Shared 类装载器负责从<CATALINA_HOME>/share/classes 中的.class 类文件和<CATALINA_HOME>/share/lib 中的 Jar 包加载类。Catalina 类装载器加载的类只对所有的 Web 应用程序可见，对 Tomcat 服务器内核完全不可见。

（6）WebappX 类装载器负责从当前 Web 应用程序的/Web-INF/classes 中的.class 类文件和/Web-INF/lib 中的 Jar 包加载类。WebappX 类装载器加载的类只对当前 Web 应用程序可见，对其他 Web 应用程序不可见。对于运行每个 Web 应用程序的线程，它们的上下文类装载器就是它们各自的 WebappX 类装载器。Sun 公司在 Servlet 规范中建议 WebappX 类装载器不应按照标准委托模式来设计，而是应该由 WebappX 类装载器自己先加载某个类，只有自己加载不了时，才委托父级的类装载器进行加载，除了 WebappX 类装载器之外，Servlet 引擎中的其他类装载器都遵循标准委托模式。但是，许多 Servlet 引擎开发商认为这个建议没有什么意义，所以，他们并没有采纳 Servlet 规范的建议，而是仍然按照标准委托模式设计了他们的 WebappX 类装载器，例如，笔者通过实验发现 Tomcat 4.x 中的 WebappX 类装载器采用的就是标准委托模式。

Tomcat 如何完成类的加载

01 将 4.2.3 节创建的 d:\myWeb\Web-INF\classes 目录中的 HelloServlet.class 文件移动（注意不是复制）到 d:\myWeb\目录中，重新启动 Tomcat，在浏览器地址栏中输入 http://localhost:8080/it315/demo/hello.html，浏览器将显示出找不到请求资源的响应信息。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

02 打开<tomcat 的安装目录>\bin\setclasspath.bat 文件，找到其中设置 CLASSPATH 环境变量的语句，将 d:\myWeb 目录增加到 CLASSPATH 环境变量中，如下所示：

```
set CLASSPATH=%JAVA_HOME%\lib\tools.jar;d:\myWeb;
```

为了保证上面设置的 CLASSPATH 环境变量起作用，应使用 startup.bat 程序重新启动 Tomcat，再次访问 http://localhost:8080/it315/demo/hello.html，浏览器中显示出如图 4.7 所示的错误信息。

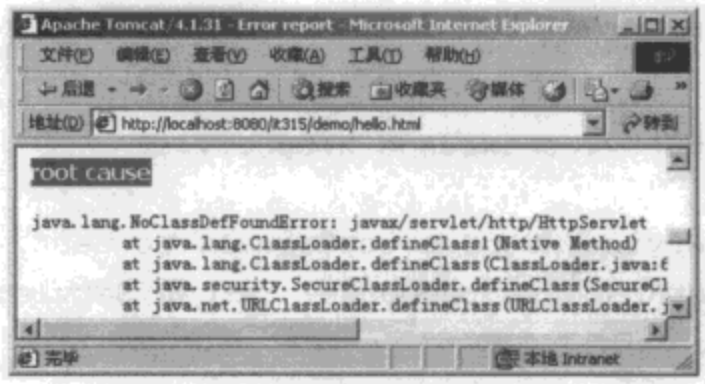


图 4.7

错误提示类装载器找不到 javax.servlet.http.HttpServlet 类，这说明 Servlet 引擎已找到并成功加载了 HelloServlet 类，只是没有找到 HelloServlet 类中引用（继承）的 HttpServlet 类。因为每个 Web 应用程序的线程上下文类装载器就是它们各自的 WebappX 类装载器，所以，当客户端访问映射到 HelloServlet 的 URL 时，首先由 WebappX 类装载器加载 HelloServlet 类，WebappX 类装载器在它所负责的目录中没有找到 HelloServlet 类，然后将装载请求按照 WebappX→share→common→system→bootstrap 路线传递到了 bootstrap，bootstrap 也不能装载 HelloServlet，装载请求又回给了 system，system 在 CLASSPATH 环境变量指定的目录中找到了 HelloServlet.class 文件，并成功对它进行加载。此时，由于装载 HelloServlet 的类装载器不再是 WebappX，而是 system，所以 HelloServlet 类中引用到的其他类（例如 HttpServlet）也将由 system 加载。system 加载 HttpServlet 类时，先将加载请求委托给 Bootstrap，显然，Bootstrap 不能加载 HttpServlet，加载请求回到 system。由于 system 只在 CLASSPATH 环境变量指定的目录中查找它所要加载的类，而不会去查找 <CATALINA_HOME>/common/lib 目录下的 Jar 包（HelloServlet 类中引用的 HttpServlet 等类位于该目录下的 servlet.Jar 包中），当然就找不到 javax.servlet.http.HttpServlet 这个类了，由于 system 是加载 HttpServlet 类的初始类装载器，它不会将此加载任务传递给子级的 Common 类装载器，于是就报告了 ClassNotFoundException 异常。

03 将 servlet.jar 文件复制到<JAVA_HOME>/jre/lib/ext 目录中，或者修改 setclasspath.bat 文件，将 servlet.jar 文件的路径增加到 CLASSPATH 环境变量中，如下所示：

```
set CLASSPATH=%JAVA_HOME%\lib\tools.jar;d:\myWeb;%BASEDIR%\common\lib\
servlet.jar;
```

这样，system 类装载器就可以找到 HelloServlet 类中引用的 HttpServlet 等类了。重新

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

启动 Tomcat，再次访问 http://localhost:8080/it315/demo/hello.html，浏览器中显示出了正常的响应结果。

04 修改 HelloServlet.java 源文件，在其中增加一条打印出当前类装载器名称的语句，如例程 4-3 所示。

例程 4-3 HelloServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException
    {
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<font size=30 color=red>www.it315.org</font><br>");
        out.println("<marquee>" + new java.util.Date() + "</marquee>");
        out.println(this.getClass().getClassLoader().getClass().
        getName());
        out.println("</html>");
    }
}
```

重新编译 HelloServlet.java 源文件，并确保编译结果覆盖了 d:\myWeb 目录中原来的 HelloServlet.class 文件。重新启动 Tomcat，再次访问 http://localhost:8080/it315/demo/hello.html，浏览器中显示的结果如图 4.8 所示。

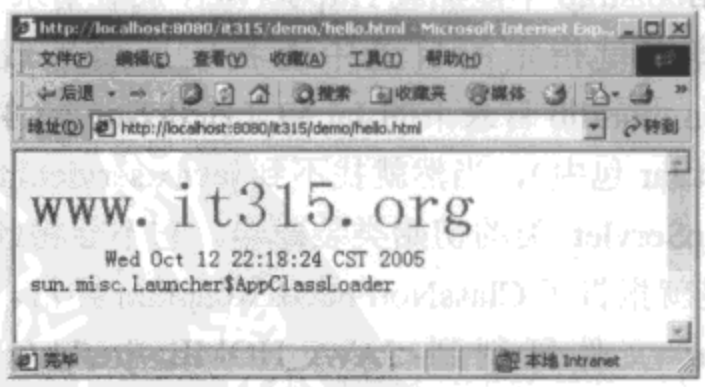


图 4.8

从图 4.8 中可以看出，HelloServlet 当前是由 AppClassLoader 类装载器进行加载的。

05 将上一步新编译成的 HelloServlet.class 文件移动到 d:\myWeb\Web-INF\classes

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

目录中，重新启动 Tomcat，再次访问 http://localhost:8080/it315/demo/hello.html，浏览器中显示的结果如图 4.9 所示。

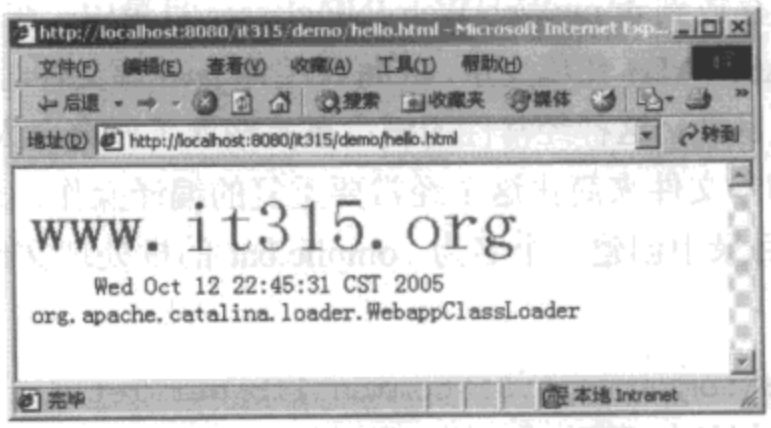


图 4.9

从图 4.9 中可以看出，HelloServlet 此时是由 WebappClassLoader 类装载器所加载。

指点迷津

区分 Servlet 的编译和运行环境

编译和运行 Servlet 的环境是完全不同的两个环境。解释执行 Servlet 的过程是在 Tomcat 服务器程序中进行的，Servlet 的运行环境就是 Tomcat 服务器程序所设置的环境；编译 Servlet 是由编程人员使用 javac 命令完成的，Servlet 的编译环境是某个命令行窗口或开发工具所设置的环境。即使 Servlet 程序在运行时能够成功装载某个其他类，但在编译时也不一定能够找到这个类，例如，Servlet 在运行时可以找到 Servlet API 类，但是如果运行 Javac 命令时的 CLASSPATH 环境变量中没有包含 Servlet API 所在的 jar 文件，编译时就会出现找不到 Servlet API 类的错误。另外，即使 Servlet 程序的编译能够成功通过，但该 Servlet 在 Tomcat 中运行时也可能出现找不到其他类的情况。

4.2.7 编写一个自动编译工具

在编译 HelloServlet.java，有以下两个特殊要求：

（1）必须将包含有 Servlet API 的 jar 文件加入到 CLASSPATH 环境变量的路径列表中，例如，对于 Tomcat 4.x，应使用如下的命令设置：

```
set CLASSPATH=<tomcat 的安装目录>\common\lib\servlet.jar;%CLASSPATH%
```

（2）编译生成的.class 文件必须放置在 Web 应用程序的 Web-INF\classes 目录中，并且要有与包名对应的子目录结构。例如，上面的 HelloServlet.java 文件的编译结果必须放置在 d:\myWeb\Web-INF\classes 目录中，则应使用如下命令进行编译：

```
javac -d d:\myWeb\Web-INF\classes HelloServlet.java
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

如果要经常要在某个目录下编写 Java 源文件，并且总是要将编译生成的 class 文件保存到另外一个目录中去，例如，本章的所有例子程序都是在 d:\javaWeb\lesson4 目录中编写的，而编译后的结果文件应存放到 d:\myWeb\Web-INF\classes 目录中。在每天开始工作时都要按照上面的要求设置 CLASSPATH 环境变量，在每次编译 java 源文件时都使用 javac 的-d 选项来指定编译结果的存储目录，这种经常性的键盘重复输入工作会令人感到烦琐和抱怨。

笔者编写了一个脚本文件来简化这个经常要重复的编译操作，运用起来非常的方便。在 d:\javaWeb\lesson4 目录中创建一个名为 compile.bat 的批处理文件，文件中包含类似如下的三行内容：

```
set CLASSPATH=c:\tomcat-4.1.31\common\lib\servlet.jar;%CLASSPATH%
javac -d d:\myWeb\Web-INF\classes %1
pause
```

以后只要在 Windows 的资源管理器窗口中，将某个要编译的 java 源文件拖动到这个批处理文件上，即可完成编译。compile.bat 的中的%1 表示传递给这个批处理文件的第一个参数，这里就是拖动到它上面的 java 源文件名。compile.bat 运行后会自动关闭它自己启动的命令行窗口，这样将无法查看到编译中出现的问题，pause 语句的作用就是让 compile.bat 执行完编译命令后暂停，等待人们按任意键后才继续运行，便于查看编译结果。读者应根据自己的实际情况，修改上面的 compile.bat 批处理文件中的目录位置。

4.3 Servlet 的特点及运行过程

有了一个 Servlet 的编写、配置和运行的经历后，理解 Serlvet 的特点和运行过程就要容易得多了。

4.3.1 Servlet 的特点

Servlet 是一个供其他 Java 程序（Servlet 引擎）调用的 Java 类，它不能独立运行。Servlet 与 Servlet 引擎之间的关系就如同 Applet 与 Web 浏览器之间的关系一样，Servlet 的运行完全由 Servlet 引擎来控制和调度。Servlet 引擎是一种容器程序，它负责管理和维护所有 Servlet 对象的生命周期，因此也被称之为 Servlet 容器或 Web 容器。Servlet 的加载、执行流程，以及如何接收客户端发送的数据和如何将数据传输到客户端等具体的底层事务，都是由 Servlet 引擎来实现的。Servlet 引擎负责将客户端的请求信息转交给 Servlet 和将 Servlet 生成的响应信息返回给客户端。Servlet 属于一种插件，它是一个提供了一些约定方法供容器去调用的类，它只负责在自身的方法中接受并处理容器传递进来的数据，以及生成并返回给容器去使用的数据和状态信息。容器好比房子的基础结构部分，留出了容纳门窗的空间，而插件则好比要嵌入到房子结构中去的门窗。

Servlet 的最常见应用在于读取 Web 浏览器传递给 Web 服务器的参数和生成 Web 服务器返回给 Web 浏览器的动态网页文档内容；Serlvet 也能获取 Web 浏览器发送的 HTTP 请

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

求消息中的各个请求头和请求行信息，以及生成用于 Web 服务器发送的 HTTP 响应消息中的状态行和响应头信息；Serlvet 还能获取 Web 服务器和 Servlet 引擎自身的一些环境和状态信息。Servlet 程序的运行过程就是它与 Servlet 引擎的交互过程，Servlet 程序从 Servlet 引擎那里获得访问请求信息和一些运行环境信息，并生成响应消息内容给 Servlet 引擎。Servlet 程序只与 Servlet 引擎打交道，它并不与 Web 服务器和客户端进行任何直接的交互。

Servlet 本身就是 Java 语言的一个应用，具有 Java 语言的所有优势，能完成普通 Java 程序所能完成的所有功能，例如，可以访问本地文件系统、数据库、调用其他 Java 类。如果需要，在 Servlet 中还可以将 HTTP 请求/响应的状态信息保存到 Servlet 引擎中，从而实现 Web 浏览器与 Web 服务器之间的有状态会话。显然，使用 Servlet 可以极大地扩展 Web 服务器的能力，例如，获取浏览器提交的一个订单网页中的 FORM 表单数据，然后将这些数据增加到公司的订单数据库中。

Web 服务器上可以布置多个功能各不相同的 Servlet，每个 Servlet 都应宣称它可以处理何种样式的 URL，当符合样式的 URL 请求到达时，Web 服务器调用 Servlet 引擎来装载和解释执行相应的 Servlet 程序，Servlet 进行处理后生成响应内容给 Servlet 引擎，然后由 Web 服务器发送给客户端。Web 服务器返回给客户端的内容是 Servlet 的运行结果，Servlet 自身的程序代码并不会传递到客户端。图 4.10 是通过 Servlet 程序查询数据库的工作过程。

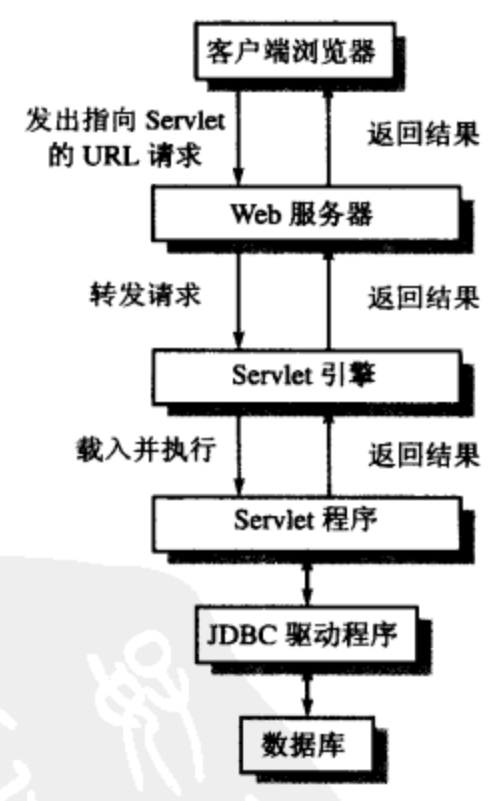


图 4.10

4.3.2 Servlet 的运行过程

Servlet 引擎按照下面的过程来调用一个 Servlet 程序：

01 接收到访问某个 Servlet 的 HTTP 请求之后，Servlet 引擎首先检查是否已经装

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

载并创建了该 Servlet 的实例对象。如果已经装载并创建了该 Servlet 的实例对象，Servlet 引擎则直接执行第 04 步，否则，执行第 02 步。

02 装载并创建 Servlet 的一个实例对象。

03 调用 Servlet 实例对象的 init()方法，以便执行 Servlet 的一些初始化工作。

04 创建一个用于封装 HTTP 请求消息的 HttpServletRequest 对象和一个代表 HTTP 响应消息的 HttpServletResponse 对象，然后调用 Servlet 的 service()方法并将请求和响应对象作为参数传递进去。

05 在一个 Web 应用程序被停止或重新启动之前，Servlet 引擎将卸载其中运行的 Servlet。在卸载 Servlet 之前，Servlet 引擎将调用 Servlet 的 destroy()方法，以便在这个方法中执行 Servlet 的一些清尾工作，例如释放一些被该 Servlet 占用的资源。Servlet 引擎卸载某个 Servlet 以后，该 Servlet 实例对象变成垃圾，等待 Java 虚拟机的垃圾收集器将其彻底从内存中清除。

补充说明：

01 上面的第 02 步和第 03 步过程一般是在 Web 服务器重新启动后，针对该 Servlet 的第一次访问时完成的，但也可以让 Web 应用程序启动时就完成这两步过程。在 4.2.4 节中，我们已经看到在 Web.xml 文件中定义的<servlet>元素中可以嵌套一个名为<load-on-startup>的子元素，这个子元素用于指定 Servlet 被装载的时机和顺序。如果<load-on-startup>元素中的内容被设置为 0 或一个正整数，它指定该 Servlet 应该在 Web 应用程序启动时就被实例化和调用它的 init()方法，且这个数字越小，Servlet 被装载的时间也越早，相同数字的 Servlet 之间的装载顺序由 Servlet 引擎自己决定。如果<load-on-startup>元素中的内容被设置成了一个负整数，则由 Servlet 引擎自己决定在什么时候装载该 Servlet。<load-on-startup>元素中的数值大小可用于解决两个 Servlet 之间的依赖关系，如果一个 Servlet 必须先于另一个 Servlet 装载，这个数值就非常有意义了。

02 因为在 Servlet 的整个生命周期内，Servlet 只被初始化一次，而对一个 Servlet 的每次访问请求都导致 Servlet 引擎调用一次 servlet 的 service 方法，所以，在 Servlet 的整个生命周期内，它的 init 方法只被调用一次，但它的 service 方法可能被调用多次。对于每次访问请求，Servlet 引擎都会创建一个新的 HttpServletRequest 请求对象和 HttpServletResponse 响应对象，然后将这两个对象作为参数传递给它调用的 Servlet 的 service()方法。在 service()方法内部首先从请求对象中获得请求信息，接着处理请求和访问其他资源以获得需要返回的信息，然后调用响应对象的方法将响应内容写入到 Servlet 引擎的缓冲区中，再由 Web 服务器发送给客户端。

03 如果编程人员对某个已被装载的 Servlet 进行了修改，除非重新启动 Tomcat，否则，客户端以后访问到的 Servlet 实例对象还是修改前的 Servlet 版本。这是因为 Servlet 在服务器的运行期间只会被加载一次，尽管在硬盘上已经覆盖掉了原来的 Servlet 文件，但服务器内存中运行的还是原来的 Servlet 程序代码。在调试一个新编写的 Servlet 时经常都会发生一些修改，如果每次修改 Servlet 后，都需要重新启动 Tomcat 才能看到修改后的运

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

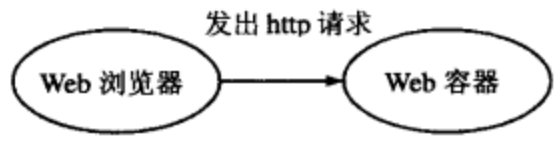
行效果，这显然是很不方便的事情，因此，Tomcat 也提供了是否自动重新装载被修改的 Servlet 的配置选项。在<Tomcat 安装主目录>/conf/server.xml 文件中，可以使用<Context>元素来设置每个独立的 Web 应用程序的配置信息，<Context>元素有一个 reloadable 属性，当它的设置值为 true 时，Tomcat 将监视该 Web 应用程序的/Web-INF/classes 和/Web-INF/lib 目录下的类是否发生了改变，然后自动重新装载那些发生了改变的类，并在 Tomcat 所运行的命令行窗口中显示出重新装载的有关提示信息。reloadable 属性的默认设置值为 false，如果想让以前设置的/it315 这个 Web 应用程序能够重新装载发生了修改的 Servlet，应将 server.xml 文件中的相应语句修改成如下形式：

```
<Context path="/it315" docBase="d:\myWeb" debug="0" reloadable="true"/>
```

因为开发阶段要经常反复地调试和修改程序，所以在开发阶段进行这样的设置是非常有用的，省得每次修改 Servlet 后都需重新启动 Tomcat 才能查看结果。即使设置了 reloadable 属性，Tomcat 也是每隔一段时间才执行一次 Servlet 的修改检查，如果修改 Servlet 后立即刷新浏览器对它的访问，看到的仍然可能是该 Servlet 修改前的执行效果，在这种情况下，应该多刷新几次或稍等片刻后再访问。让 Tomcat 不断地执行这样的检测工作，显然需要额外的运行开销，从而降低了系统的运行效率，所以在程序作为产品真正上线运行时，不要将 Web 应用程序的 reloadable 属性设置为 true。如果 Web 应用程序作为产品正在上线运行，偶尔对某个 Servlet 程序进行了修改，可以使用 Tomcat 自带的管理程序（也是以 Web 应用程序的形式提供的，在 Tomcat 默认首页上具有进入该程序的超链接）来重新装载某个 Web 应用程序的所有内容。

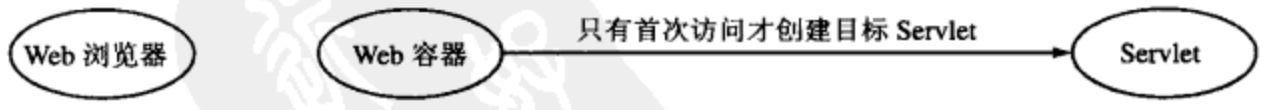
下面是浏览器访问 Servlet 的过程示意图，其中的 Web 容器泛指 Web 服务器和 Servlet 引擎。

01



浏览器根据用户的请求生成 http 请求消息，并将其发送给 Web 容器。

02



Web 容器检查内存中是否存在负责处理当前请求的 Servlet 程序的实例对象，如果不存在，则加载和创建该 Servlet 对象。

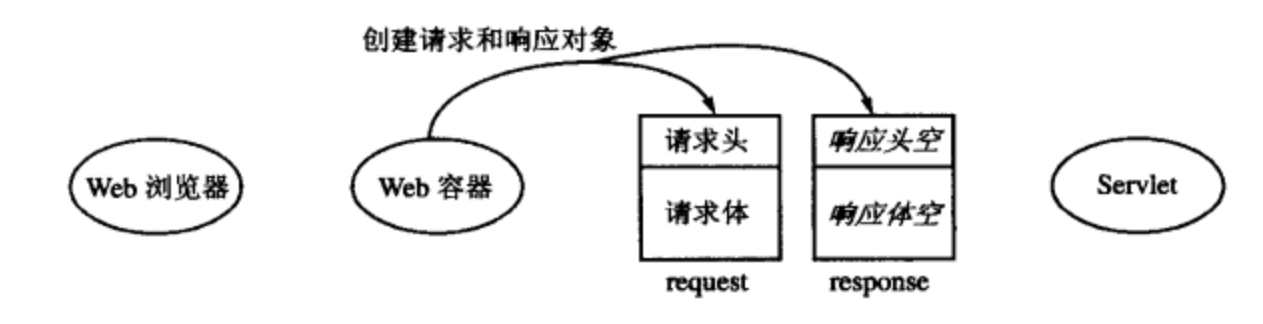
03

Web 容器创建针对该次访问的请求对象（request）和响应对象（response）。请求对象中包含了 http 的请求信息，从它里面可以获取 HTTP 请求消息的内容；响应对象用于封装将要回送给浏览器的 http 响应信息，响应对象中的初始内容为空，但是以后可以调用它的各种方法来生成 HTTP 响应消息的各个部分。HttpServletRequest 对象和 HttpServletResponse

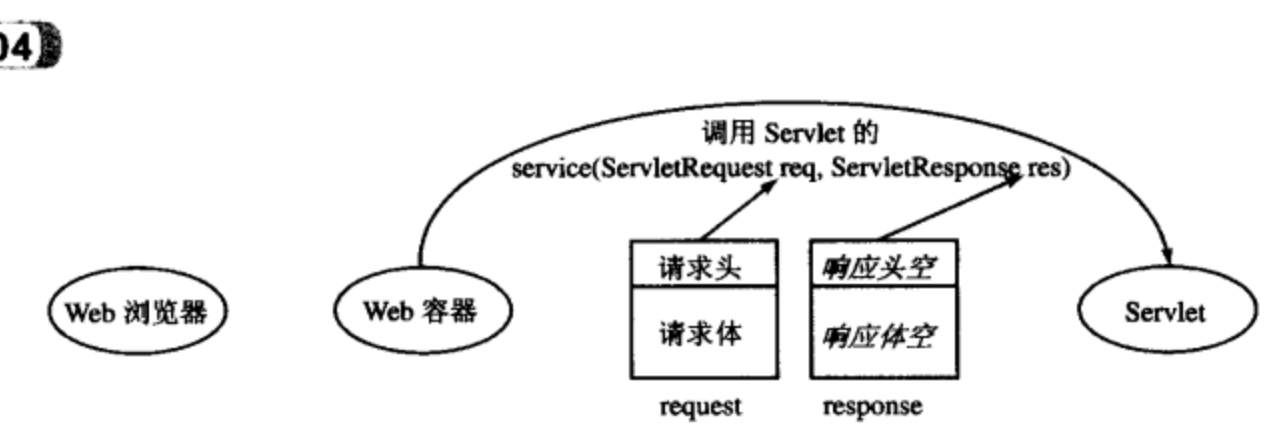
http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

对象是 Servlet 引擎与 Servlet 程序进行通信的纽带，它们好比朋友 A 拜访朋友 B 时，朋友 A 给朋友 B 带来的两个盒子，一个盒子中装的是 A 送给 B 的礼品，另外一个则是用于装 B 回赠给 A 的礼品的空盒子。B 可以从第一个盒子中取出 A 送来的礼品，同时，只要将回赠的礼品放入第二个盒子中，A 就可以从第二个盒子中得到返回的礼品。HttpServletRequest 对象相当于装有 A 送给 B 的礼品的盒子，HttpServletResponse 对象则相当于 B 用来装回赠给 A 的礼品的空盒子。

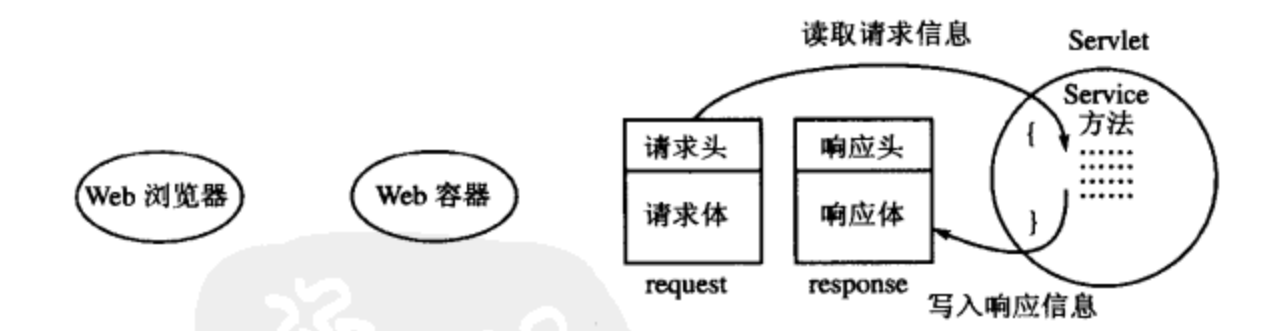


04



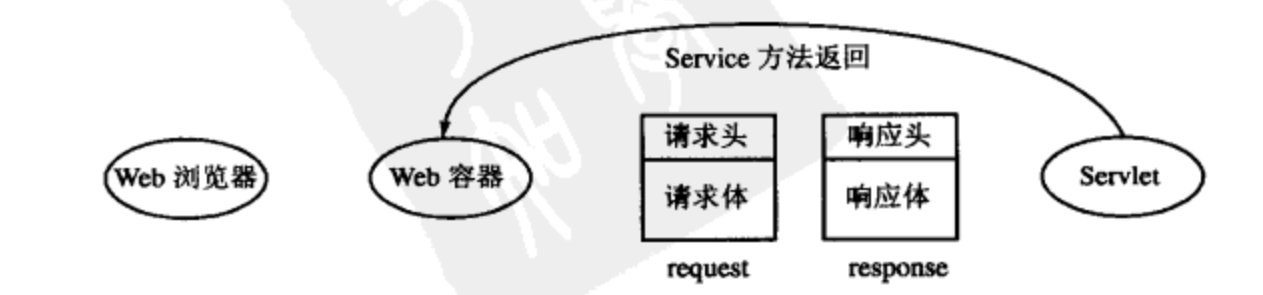
Web 容器调用 Servlet 的 service（ServletRequest request，ServletResponse response）方法，并将请求对象和响应对象传递给该方法。

05



Servlet 对象的 Service 方法从请求对象中读取请求信息，并向响应对象中写入响应头和响应体信息。

06

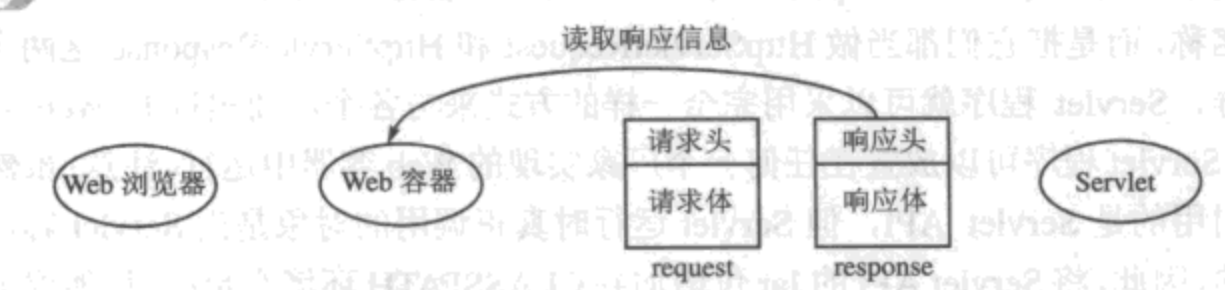


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

Service 方法执行完毕后返回。

07



Web 容器从响应对象中读取响应信息。

Web 容器与 Servlet 之间的通信方式不是“一手交钱、一手交货”的直接方式，而是类似电影中的两个地下党交换情报的间接方式，一个地下党不是把情报直接交到另一方的手中，而是先把情报放在公园的某个树洞中，另一方则在以后从树洞中取走情报。这种情报交换方式可能会遇到一个问题：敌人可能提前从树洞中截获情报，然后在树洞中放置一个假情报，当另外一个地下党来取情报时，他取走的就是假情报了。读者在以后学习 Filter 编程时，只要明白了 Web 容器与 Servlet 之间不是直接交换信息，而是通过 HttpServletRequest 对象和 HttpServletResponse 对象来交换信息，再借助这个情报交换的比喻，把 Filter 程序想象成是截获情报的敌人，就会很容易理解 Filter 程序是如何截获并修改请求消息和响应消息的原理了。

08

Web 容器将响应信息发送给浏览器处理和显示，一次请求响应过程完全结束，request 和 response 变成垃圾，等待垃圾收集器将其彻底从内存中清除。

从上面的过程示意图中可以看到，Web 容器与 Servlet 的交互主要是通过 request 和 response 对象来完成的，Web 容器负责创建这些对象并传递给 Servlet 程序，Servlet 程序则使用这些对象，并调用它们的方法来与 Web 容器进行通信。Servlet 程序要使用这些对象，必须知道这些对象所提供的方法约定，按照面向对象的松耦合模块化设计思想，只要让创建这些对象的类实现某一接口，Servlet 程序就不需要知道该类的具体名称，而只需知道创建这些对象的类所实现的接口，即可知道这些对象所具有的方法并能对这些方法进行调用。Servlet API 的主要作用就是定义了这些接口类，Servlet、Servlet 容器与 Servlet API 三者之间的关系如图 4.11 所示。

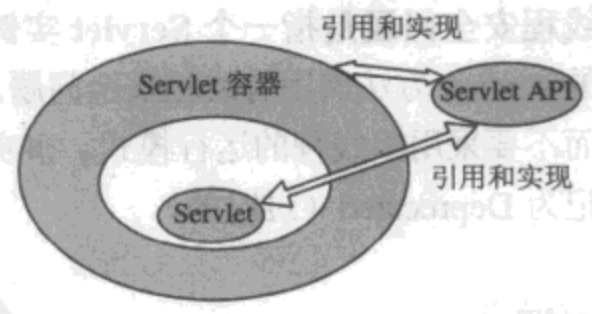


图 4.11

Servlet API 中并没有定义用于创建 request 和 response 对象的具体类，它里面只定义了创建这些对象的类所需要实现的接口，例如，HttpServletRequest 和 HttpServletResponse 实

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

际上都是接口类型，它们的实现类则由各个 Web 容器厂商去实现。不管每个 Web 容器厂商设计的用于创建 request 和 response 对象的类叫什么名称，Servlet 程序并不需要知道它们的具体名称，而是把它们都当做 HttpServletRequest 和 HttpServletResponse 这两个名称来对待，这样，Servlet 程序就可以采用完全一样的方式来与各个厂商实现的 Web 容器进行交互，一个 Servlet 程序可以放置在任何一个厂家实现的 Web 容器中运行。注意：虽然 Servlet 源程序中引用的是 Servlet API，但 Servlet 运行时真正调用的对象是由 Servlet 容器中的实现类创建的，因此，将 Servlet API 的 Jar 包增加到 CLASSPATH 环境变量中，只能保证 Servlet 程序可以被成功编译，但不能让 Servlet 程序离开 Servlet 容器运行。

4.3.3 Servlet 的线程安全问题

Servlet 引擎采用多线程模式运行，它为并发的每个访问请求都预备一个独立的线程来进行响应。Servlet 引擎采用多线程模式可以提高访问性能，但也带来了线程安全的问题。例如，如果并发的多个请求访问的是同一个 Servlet，那么将有多个线程来并发调用该 Servlet 的 service 方法。

Servlet API 中定义了一个 javax.servlet.SingleThreadModel 接口，如果某个 Servlet 实现了 SingleThreadModel 接口，那么 Servlet 引擎将以单线程模式来调用其 service 方法。SingleThreadModel 接口中没有定义任何方法，只要在 Servlet 类的定义中增加实现 SingleThreadModel 接口的声明即可。对于实现了 SingleThreadModel 接口的 Servlet，Servlet 引擎将保证其单个实例对象的 service 方法不会同时被两个线程调用。但是，Servlet 引擎仍然支持对该 Servlet 的多线程并发访问，其采用的方式是产生多个 Servlet 实例对象，并发的每个线程分别调用一个独立的 Servlet 实例对象，这样，可以同时有多个 Servlet 实例对象并发运行，但每一个 Servlet 实例对象的 service 方法都不会被多个线程并发调用。实现了 Single 章 ThreadModel 接口的 Servlet 的多个实例对象共同组成了一个 Servlet 对象池，当 Web 容器接收到针对该 Servlet 的访问请求时，将从 Servlet 对象池中挑选一个空闲的 Servlet 实例对象进行响应。实现 SingleThreadModel 接口并不能真正解决 Servlet 的线程安全问题，因为 Servlet 引擎会创建多个 Servlet 实例对象，多个并发的请求分别由不同的 Servlet 实例对象处理，或者被单个 Servlet 实例对象逐个依次排队处理，而**真正意义上的多线程安全问题是指一个 Servlet 实例对象被多个线程同时调用的问题**。所以，如果多个并发的 service 方法调用有线程安全问题，最好是自己在 Service 方法中编写多线程同步控制代码，而不要采用单线程的运行模式。事实上，在 Servlet API 2.4 中，已经将 SingleThreadModel 标记为 Deprecated（过时的）。

Servlet 的并发访问与线程安全问题

01 按例程 4-4 编写一个名为 ThreadServlet 的 Servlet 程序，在 ThreadServlet 类中定

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

义一个用于统计访问次数的成员变量 count，在 service 方法每次被调用时将 count 加 1，然后调用 System.out.println 方法打印出 count 变量的值，就可以知道当前是 Servlet 的第几次调用。在修改 count 变量的语句和打印访问次数的程序代码之间增加一段延时，以便在 ThreadServlet 还没有完成第 1 次响应的期间，客户机可以发出针对它的第 2 次、第 3 次、…、第 *n* 次访问请求，从而看到多个线程并发访问同一个 Servlet 程序的运行效果。另外，在调用 System.out.println 方法打印 count 变量时，顺便也打印出当前线程的名称，以便可以通过打印在 Tomcat 所运行的命令行窗口中的结果来了解 service 方法每次被调用时的线程情况。

例程 4-4 ThreadServlet.java

```
package org.it315;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ThreadServlet extends HttpServlet
{
    private int count = 0;
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,IOException
    {
        count++;
        try
        {
            Thread.sleep(500);
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
        }
        System.out.println("这是第" + count + "次访问" +
            "，这次访问的线程名称为：" + Thread.currentThread().getName());
    }
}
```

在 Windows 的资源管理器窗口中，将 ThreadServlet.java 源文件拖动到前面编写的 compile.bat 批处理文件上进行编译，确保编译结果存放在 d:\myWeb\Web-INF\classes 目录中，并且在 d:\myWeb\Web-INF\classes 目录中应该有与包名 org.it315 对应的子目录层次结构，ThreadServlet.class 文件位于该子目录中。

02 启动 Tomcat 后，在本地浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/servlet/org.it315.ThreadServlet
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

然后快速单击几次浏览器的“刷新”按钮，在 Tomcat 所运行的命令行窗口中看到打印出如图 4.12 中的白色区域所示的内容。

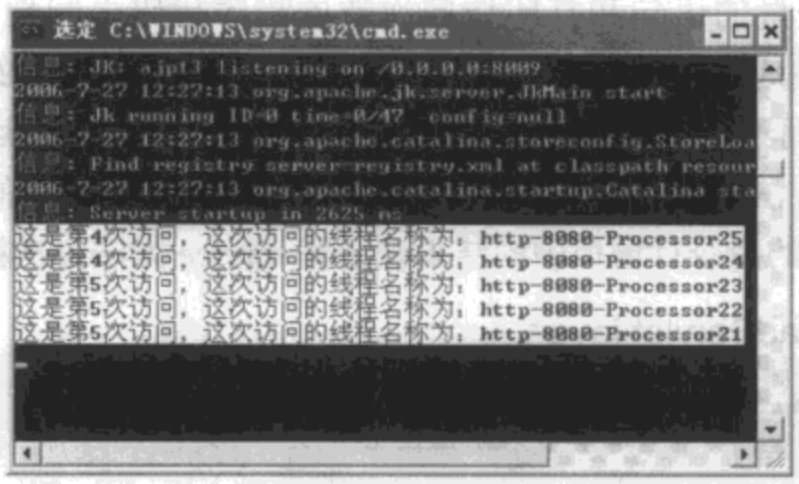


图 4.12

从图 4.12 中的打印结果上可以看到，如果 Servlet 引擎在一个线程中执行某个 Servlet 的 service 方法期间，接收到了对该 Servlet 的新访问请求，Servlet 引擎将使用另外的一个线程来再次调用该 Servlet 的 service 方法。图 4.12 中打印出了多行“这是第 5 次访问”的内容，这说明多个线程并发执行同一个 Servlet 的 service 方法时，出现了线程安全的问题。

这里使用三个线程并发执行的情况为例来分析这个例子程序中的问题。第一线程将把 count 变量变成 1，第二线程将把 count 变量变成 2，第三线程将把 count 变量变成 3。计算机的 CPU 在一个小的时间段内只能执行一个线程，当第一个线程将 count 变量变成 1 后，还没有打印出此时的 count 变量值，CPU 就切换到了其他线程上去执行，第一个线程处于暂停状态。计算机的 CPU 执行其他线程时，其他线程对 count 进行了修改，等到 CPU 再回到第一个线程上继续执行时，count 变量已经变成了 3，所以第一个线程打印出的结果就是“第 3 次访问”，而不是预想的“第 1 次访问”。关于线程序安全问题的解决办法，不属于本书所要讲解的问题，如果读者以前没有这方面的概念，请参看笔者编写的《Java 就业培训教程》中的有关多线程的讲解，这里就不再赘述了。

注意 如果在 Tomcat 所运行的命令行窗口中用鼠标选中了任何一块区域，如图 4.12 所示，那么使用浏览器再次访问 Web 服务器时，浏览器端看到的现象将处于一种停滞不动的响应状态。这是因为在 Windows 2000 的命令行窗口中只要用鼠标选中了任何一块区域，就会暂停它里面运行的程序。只要在命令行窗口按动任意一个键，就可以放弃选中，从而恢复程序的运行。

03 将 ThreadServlet 类的定义部分修改成如下形式：

```
public class ThreadServlet extends HttpServlet implements SingleThreadModel
{
    ...
}
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

重新编译并等待 Tomcat 重新装载 ThreadServlet 后，重复第 02 步的访问操作，在 Tomcat 所运行的命令行窗口中打印出如图 4.13 所示的白色区域的内容。

图 4.13

图 4.13 中显示出了多个线程名称，这说明有多个线程在同时运行。图 4.13 中打印出了三行“这是第 1 次访问”的内容，这显然是三个各不相干的 count 成员变量的结果，这说明一共产生了三个 ThreadServlet 实例对象。图 4.13 中也打印出了一行“这是第 2 次访问”的内容，这说明其中一个 ThreadServlet 实例对象被调用了两次。从这个实验结果中可以看到，Servlet 引擎创建了多个 Servlet 实例对象，多个并发的请求分别由不同的 Servlet 实例对象处理。

指点迷津

哪些情况下要考虑线程安全问题

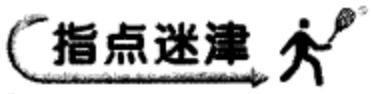
在 Servlet 程序中除了访问成员变量时要注意线程安全问题，访问其他共享资源时也应注意线程安全问题，例如，Web 应用程序对象，Session 对象，数据库等都可能涉及多个 Servlet 线程同时访问的问题。如果 Servlet 中没有定义成员变量和 service 方法中没有访问无同步访问控制的共享数据时，即使多个线程并发调用 service 方法，也不会存在任何线程安全问题。

4.4 ServletConfig 接口

Servlet 程序是发布到 Web 应用程序中运行的，此 Web 应用程序就称之为 Servlet 容器。Servlet 是在 Servlet 容器内运行的程序，它在有些情况下可能需要访问 Servlet 容器或借助 Servlet 容器访问外部的资源，所以，Serlvet 引擎需要将表示 Servlet 容器的对象传递给 Servlet。另外，在 Web.xml 文件中为某个 Servlet 设置的友好名称和初始化参数等信息也需要传递给该 Servlet。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享



Servlet 的初始化参数的作用

假设某个 Servlet 要被多个公司去使用，这个 Servlet 运行时需要显示出正在使用该 Servlet 的公司的名称，显然在编写 Servlet 程序时无法确定它将被哪个（或哪些）公司使用，所以，公司名称不能被硬编码进 Servlet 程序中。如果在某个公司使用这个 Servlet 时，再在配置文件通过参数的形式将公司名传递给 Servlet，这个问题就迎刃而解了。在 web.xml 配置文件中为 Servlet 设置参数的情况如下所示：

```
<servlet>
   <servlet-name>ConfigTest</servlet-name>
   <servlet-class>ConfigTestServlet</servlet-class>
   <init-param>
       <param-name>Corporation</param-name>
       <param-value>北京传智播客教育科技有限公司</param-value>
   </init-param>
</servlet>
```

Servlet 引擎将代表 Servlet 容器的对象和 Servlet 的配置参数信息一并封装到一个称为 ServletConfig 的对象中，并在初始化 Servlet 实例对象时传递给该 Servlet。ServletConfig 接口则用于定义 ServletConfig 对象需要对外提供的方法，以便在 Servlet 程序中可以调用这些方法来获取有关信息，例如，获取代表 Servlet 容器的对象、获取在 Web.xml 文件中为 Servlet 设置的友好名称和初始化参数等等。

Servlet 引擎装载并创建一个 Servlet 的实例对象后，接着调用该实例对象的 init (ServletConfig config)方法将 ServletConfig 对象传递给 Servlet。Servlet.getServletConfig()方法必须返回 init(ServletConfig config)方法传递进来的这个 ServletConfig 对象的引用。

Servlet 的 init 方法与 getServletConfig 方法的关系与作用

初学人员要想能很好地理解 Servlet.init(ServletConfig config)方法和 Servlet.getServlet Config()方法的关系与作用，首先必须理解 Servlet 引擎、Servlet 程序和 ServletConfig 对象之间的关系，并要在脑海中树立一种新的编程观念：你写的 Servlet 程序只是相当于整个系统程序中的一个零部件，整个系统程序的主体框架（这里是 Servlet 引擎）则已经由别人写好，你写的 Servlet 程序是专供别人写的主体框架程序去调用的。别人编写的主体框架程序负责读取配置文件，根据配置文件中的信息生成 ServletConfig 对象，接着创建你写的 Servlet 类的实例对象，然后调用该 Servlet 实例对

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

象的 init 方法把 ServletConfig 对象传递给该 Servlet 实例对象。你编写的 Servlet 程序内部应该使用一个私有成员来引用这个 ServletConfig 对象，以便在 init 方法之外的其他地方可以使用这个 ServletConfig 对象。另外，为了能够在 Servlet 程序中其他的地方获得这个 ServletConfig 对象的引用，最好是提供一个 getServletConfig 方法来暴露这个 ServletConfig 对象的引用，特别是针对子类不能访问父类中的私有成员这一情况，作为基类的 Servlet 更应该提供 getServletConfig 方法来让它所派生的 Servlet 子类可以获得它所保存的 ServletConfig 对象的引用。

4.4.1 getInitParameterNames 方法

在 web.xml 文件中可以为 Servlet 设置若干个初始化参数，getInitParameterNames 方法用于返回一个 Enumeration 集合对象，该集合对象中包含在 Web.xml 文件中为当前 Servlet 设置的所有初始化参数的名称。如果在 Web.xml 文件中对一个名为 ConfigTestServlet 的 Servlet 进行了如下设置：

```
<servlet>
    <servlet-name>ConfigTest</servlet-name>
    <servlet-class>ConfigTestServlet</servlet-class>
    <init-param>
        <param-name>firstname</param-name>
        <param-value>zhang</param-value>
    </init-param>
    <init-param>
        <param-name>lastname</param-name>
        <param-value>san</param-value>
    </init-param>
</servlet>
```

那么，在 ConfigTestServlet 中调用 ServletConfig.getInitParameterNames 方法返回的 Enumeration 集合对象中将包含两个元素，即内容分别为“firstname”和“lastname”的两个字符串对象。

4.4.2 getInitParameter 方法

getInitParameter 方法用于返回在 Web.xml 文件中为 Servlet 所设置的某个名称的初始化参数的值，如果指定名称的初始化参数不存在，则返回值为 NULL。例如，对于上面的 ConfigTestServlet 这个 Servlet 的设置情况，将“firstname”作为参数传递给 getInitParameter 方法时，得到的返回结果为“zhang”。

4.4.3 getServletName 方法

getServletName 方法用于返回 Servlet 在 Web.xml 文件中的注册名称，例如，对于上面

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

的 ConfigTestServlet 这个 Servlet 的设置情况，getServletName 方法返回的结果为“ConfigTest”。对于没有在 web.xml 文件中注册的 Servlet，getServletName 方法返回的结果为该 Servlet 的类名。

4.4.4 getServletContext 方法

在 Servlet 程序中，每个 Web 应用程序（Servlet 容器）都用一个各自的 ServletContext 对象来表示，ServletConfig 对象中包含了 ServletContext 对象的引用，getServletContext 方法用于返回 ServletConfig 对象中所包含的 ServletContext 对象的引用。

4.4.5 GenericServlet 类实现 ServletConfig 接口的目的

在 Servlet 接口中定义了一个 getServletConfig 方法，该方法必须返回 Servlet 容器调用 Servlet.init(ServletConfig config)方法时传递进来的那个 ServletConfig 对象的引用，GenericServlet 类已经按此要求实现了 getServletConfig 方法。如果要在 Servlet 程序中调用 ServletConfig 对象的方法，首先应调用 getServletConfig()方法得到 ServletConfig 对象的引用，然后再通过这个引用来调用 ServletConfig 对象的各个方法，例如，在 Servlet 程序内部可以使用如下语句来得到 Servlet 的名称：

```
String servletName = getServletConfig().getServletName();
```

虽然上面的语句并不复杂，但是，GenericServlet 类还是提供了一些简化的手段。GenericServlet 类除了实现了 Servlet 接口中的方法之外，它还实现了 ServletConfig 接口中的方法，在实现的 ServletConfig 接口方法内部，它们调用 ServletConfig 对象的同名方法。例如，GenericServlet 类的 getServletName 方法的定义如下：

```
public String getServletName()
{
    return getServletConfig().getServletName();
}
```

这样，如果要在 Servlet 中调用 ServletConfig 对象的方法，就不必先调用 getServletConfig()方法返回 ServletConfig 对象的引用之后，再通过这个引用来调用 ServletConfig 对象的方法，而是直接调用 GenericServlet 类中实现的同名方法。例如，在 Servlet 程序内部不必再调用如下语句来得到 Servlet 的名称：

```
String servletName = getServletConfig().getServletName();
```

而是可以简写成如下形式：

```
String servletName = getServletName();
```

4.4.6 ServletConfig 的应用举例

下面编写一个调用 ServletConfig 中的所有方法的例子程序，并通过一些操作实验来查

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

看其运行效果。

ServletConfig 对象的应用

01 按例程 4-5 编写一个名为 ConfigTestServlet 的 Servlet 程序。

例程 4-5 ConfigTestServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Enumeration;

public class ConfigTestServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException
    {
        //下面这一句的作用可以参看第 5 章的中文输出问题，
        //建议读者暂且不要关心这个问题。
        response.setContentType("text/html;charset=GB2312");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        //取消下面语句中的注释符号，程序可能更容易理解
        out.println("Servlet 名称为：" + /*this.*/getServletName() + "<br>");
        Enumeration e = /*getServletConfig().*/getInitParameterNames();
        out.println("下面是为 Servlet 设置的初始化参数：" + "<br>");
        while (e.hasMoreElements())
        {
            String key = (String)e.nextElement();
            String value = getInitParameter(key);
            out.println("  " + key + "=" + value + "<br>");
        }
        ServletContext context = getServletContext();
        String path = context.getRealPath("/");
        out.println("当前 Web 应用程序的本地目录为：" + path + "<br>");
        out.println("</html>");
    }
}
```

编译 ConfigTestServlet.java 文件，确保生成的 class 文件放置进 d:\myWeb\Web-INF\classes 目录中。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

02 在 d:\myWeb\Web-INF\Web.xml 文件中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>ConfigTest</servlet-name>
    <servlet-class>ConfigTestServlet</servlet-class>
    <init-param>
        <param-name>firstname</param-name>
        <param-value>zhang</param-value>
    </init-param>
    <init-param>
        <param-name>lastname</param-name>
        <param-value>san</param-value>
    </init-param>
</servlet>

……
<servlet-mapping>
    <servlet-name>
        ConfigTest
    </servlet-name>
    <url-pattern>
        /demo/config.html
    </url-pattern>
</servlet-mapping>
```

03 启动或重新启动 Tomcat，在浏览器地址栏中输入如下地址：

http://localhost:8080/it315/demo/config.html

浏览器中显示的结果如图 4.14 所示。

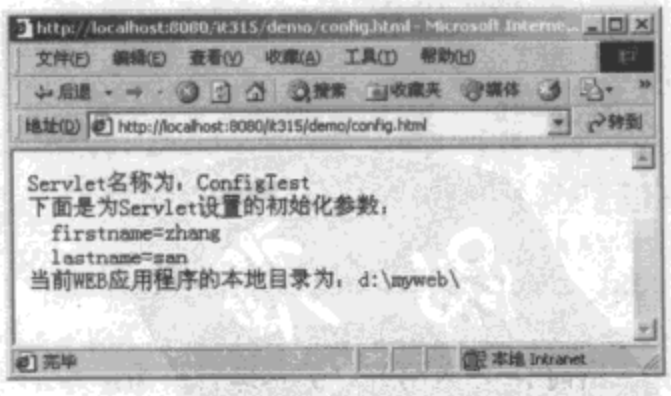


图 4.14

注意 要获取 Serlvet 程序在 Web.xml 文件中的设置信息，必须使用该 Serlvet 在 web.xml 文件中所映射的 URL 路径来访问它，而不能使用激活器的方式进行访问。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

4.5 GenericServlet 与 HttpServlet 类

GenericServlet 是一个实现了 Servlet 的基本特征和功能的基类，其完整名称为 javax.servlet.GenericServlet。HttpServlet 是 GenericServlet 子类，其完整名称为 javax.servlet.http.HttpServlet，它提供了处理 HTTP 协议的基本架构。如果一个 Servlet 要充分利用 HTTP 协议的功能，就应该继承 HttpServlet。HttpServlet 类继承了 GenericServlet 类，而 GenericServlet 类又实现了 Servlet 和 ServletConfig 这两个接口类，所以，在 HttpServlet 类及其子类中，除了可以调用 HttpServlet 类内部新定义的方法外，还可以调用包括 Servlet、ServletConfig 接口和 GenericServlet 类中定义的一些方法。

4.5.1 区分回调方法与工具方法

一般情况下，人们编写的程序都习惯于调用系统工具类的方法来帮助自己完成某个功能，例如，调用 java.util.Random 类的 nextInt()方法获得一个随机整数。但是，对于用作组件或插件的类则需要编写一些供系统去调用的方法，这些专门用于被系统调用的方法被称之为回调方法，也就是回过来被系统调用的方法。

Servlet 是一个要被容器调用的组件类，它的运行过程由 Servlet 容器控制和调度。Servlet 接口中定义的所有方法都是回调方法，这些方法定义了 Servlet 容器与 Servlet 程序之间通信的协议约定，专门供 Servlet 容器调用。通常情况下，Servlet 编程人员只负责编写这些方法，而自己不对这些方法进行调用。容器与回调方法的工作关系，就像好莱坞电影里的一段精彩对话："你站着别动，我在有需要的时候来找你"。例如，Servlet 接口中定义的 service 方法是一个回调方法，它处于一种被动等待状态，等待 Servlet 容器在适当时机（客户端发出请求时）来调用并接受容器传递进来的参数信息，以及生成并返回给容器去使用的数据和状态信息。

GenericServlet 和 HttpServlet 类中的回调方法是供容器调用的方法，可以在子类中进行覆盖，而不能在程序代码中直接调用。GenericServlet 和 HttpServlet 类中除了包含有回调方法外，也包含有一些供 Servlet 编程人员调用的工具方法，编程人员可以在程序代码中直接调用这些工具方法去完成某种任务，例如，可以调用 GenericServlet 类中的 log()方法将一条信息记录到日志文件中。在学习和应用 GenericServlet 和 HttpServlet 类时，读者应该区分清楚哪些是需要自己来实现或覆盖的回调方法，哪些是可以直接调用的工具方法。

4.5.2 init 方法

init 方法在 Servlet 的生命期中仅执行一次，Servlet 引擎创建 Servlet 实例对象后立即调用该方法，Servlet 开发人员可以在这个方法中完成与构造方法类似的初始化功能。构造方法是由 Java 系统自动调用的，总是伴随着 Java 对象的创建而发生，这是 Java 语言的一个

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

特性；init 方法是在 Servlet 对象被创建后，再由 Servlet 引擎调用的方法，init 方法的调用位于构造方法之后。Servlet 引擎在调用 init 方法时，会传递一个包含 Servlet 的配置和运行环境信息的 ServletConfig 对象，Servlet 接口中定义的 init 方法的语法格式为：

```
public void init(ServletConfig config) throws ServletException
```

可见，如果初始化代码中要使用到 ServletConfig 对象，那么，这些初始化代码就只能在 Servlet 的 init 方法中编写，而不能在构造方法中编写。如果 init 方法抛出了异常，Servlet 引擎将卸载 Serlvet。

在 GenericServlet 中还定义了一个无参数的 init 方法，如下所示：

```
public void init() throws ServletException
```

GenericServlet 中的这个无参数的 init()方法的方法体中没有编写任何程序语句，是一个空函数。GenericServlet 实现的 init(ServletConfig config)方法中调用了这个无参数的 init()方法，所以，对于继承 GenericServlet 类的 Servlet 程序，也可以覆盖这个无参数的 init()方法来编写初始化代码，并且覆盖这个无参数的 init()方法还有一个好处，那就是省去了覆盖 init(ServletConfig config)方法时总是要编写 super.init(config)语句的麻烦。

指点迷津

覆盖无参数的 init()方法来编写初始化代码的好处

在 GenericServlet 实现的 init(ServletConfig config)方法中，使用了一个私有成员变量将作为参数传递进来的 ServletConfig 对象的引用存储起来，以便 GenericServlet 中的其他方法使用，例如，getServletConfig()方法需要返回这个 ServletConfig 对象，getServletContext()等方法内部要调用这个 ServletConfig 对象。GenericServlet 类中的有关方法的示意代码如下：

```
public class GenericServlet
{
   private ServletConfig config = null;

   public void init(ServletConfig config) throws ServletException
   {
       this.config = config;
   }

   public ServletConfig getServletConfig()
   {
       return config;
   }
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
    public ServletContext getServletContext()
    {
        return config.getServletContext();
    }

    ......
}
```

可见，GenericServlet 中的一些方法依赖于 GenericServlet 的 init(ServletConfig config)方法处理后的结果。显然，如果在执行其他方法前，没有执行 init(ServletConfig config)方法内的代码，那么，其他方法将会出现问题！

如果要在自己编写的 GenericServlet 的子孙类的 init(ServletConfig config)方法中增加一些额外的初始化功能，那么就必须覆盖 GenericServlet 实现的 init(ServletConfig config)方法。由于子类覆盖 GenericServlet 的 init(ServletConfig config)方法后，GenericServlet 的 init(ServletConfig config)方法将不再被调用，为了不破坏 GenericServlet 的其他方法的功能，GenericServlet 子孙类中的 init(ServletConfig config)方法首先必须完成 GenericServlet 类的 init(ServletConfig config)方法的功能，即应该使用 super.init(config)语句调用 GenericServlet 的 init(ServletConfig config)方法，然后才能编写自己的扩展初始化代码。但是，并不是每个 Servlet 开发人员都能意识到和总能想到这一问题，为了省去子类覆盖 init(ServletConfig config)方法时总是要编写 super.init(config)语句的麻烦，GenericServlet 的设计者采用了一种巧妙的方式来解决这一问题。他们在 GenericServlet 中定义了一个无参数的 init()方法，如下所示：

```
public void init() throws ServletException
```

这个无参数的 init()方法的方法体中没有编写任何程序语句，是一个空函数。在 GenericServlet 的 init(ServletConfig config)方法中除了编写原有的初始化代码外，在最后还调用了这个无参数的 init 方法，如下所示：

```
public void init(ServletConfig config) throws ServletException
{
    this.config = config;
    init();
}
```

注意 由于 Servlet 容器与 Servlet 程序是按 Servlet 接口中约定的方法进行通信的，对于 Servlet 程序中的多个重载形式的 init 方法，Servlet 容器始终只会去调用 Servlet 接口中定义的那个 init 方法，而其他形式的 init 方法则相当于 Servlet 程序内部定义的普通方法，Servlet 容器根本就不知道它的存在。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

GenericServlet 类进行这样的设计后，如果 GenericServlet 的子孙类要增加一些额外的初始化功能，它只需覆盖无参数的 init()方法，而不必覆盖有参数的 init(Servlet Config config)方法。这样，Serlvet 容器在初始化由 GenericServlet 派生出的子类 Servlet 程序时，调用的仍然是 GenericServlet 的有参数的 init(Servlet Config config)方法，但是由于该方法最后调用了无参数的 init()方法，根据面向对象的多态性，这里调用的无参数的 init()方法不再是 GenericServlet 自身的无参数的 init()方法，而实际是子类中的 init()方法。因为子类的无参数 init（）方法执行前，Servlet 容器已经调用了 GenericServlet 的有参数的 init(Servlet Config config)方法，所以，在子类中覆盖的无参数的 init()方法不再需要调用 super.init(config)语句来完成父类原来的初始化功能，子类从 GenericServlet 类中继承的其他方法的功能仍然会保持完好无损，例如，getServletConfig()方法仍然可以返回 ServletConfig 对象。

4.5.3 service 方法

service 方法是 Servlet 的核心方法，每当针对某个 Servlet 的访问请求到达时，Servlet 引擎就会调用该 Servlet 实例对象的 service 方法来进行响应。Servlet 接口中定义的 service 方法的语法格式为：

```
public void service(ServletRequest req,ServletResponse res)
    throws ServletException,java.io.IOException
```

GenericServlet 类没有对这个方法进行实现，HttpServlet 类实现了这个方法。由于 Servlet 接口中定义的 service 方法接受的参数类型分别为 ServletRequest 和 ServletResponse，对于采用 HTTP 协议的访问请求，如果要在这个 service 方法内部使用 HTTP 消息的特有功能，也就是要调用 HttpServletRequest 和 HttpServletResponse 类中定义的方法时，还需要将请求和响应对象的类型分别转换成 HttpServletRequest 和 HttpServletResponse，如下所示：

```
public void service(ServletRequest req,ServletResponse res)
    throws ServletException,java.io.IOException
{
    HttpServletRequest request = (HttpServletRequest)req;
    HttpServletResponse response = (HttpServletResponse)res;
    ……
    request.HttpServletRequest 中定义的方法();
    response.HttpServletResponse 中定义的方法();
    ……
}
```

为了简化这一转换过程，HttpServlet 类实现的 service 方法内部调用了另外一个重载形式的 service 方法，重载的 service 方法的定义语法为：

```
protected void service(HttpServletRequest req,HttpServletResponse resp)
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
    throws ServletException,java.io.IOException
```

在 HttpServlet 所实现的 Servlet 接口中定义的 service 方法内部，将请求和响应对象的类型转换成 HttpServletRequest 和 HttpServletResponse 类型后，再将它们作为参数传递给这个重载的 service 方法，如下所示：

```
public void service(ServletRequest req,ServletResponse res)
    throws ServletException,java.io.IOException
{
    HttpServletRequest request = (HttpServletRequest)req;
    HttpServletResponse response = (HttpServletResponse)res;
    service(request,response);//这里调用的是那个重载的 service 方法
}
```

如果在子类中覆盖这个重载的 service 方法，其接收的参数类型即为 HttpServletRequest 和 HttpServletResponse，那么，在子类覆盖的 service 方法中就不再需要对这两个参数进行类型转换，如下所示：

```
public void service(HttpServletRequest req,HttpServletResponse res)
    throws ServletException,java.io.IOException
{
    ……
    req.HttpServletRequest 中定义的方法();
    res.HttpServletResponse 中定义的方法();
    ……
}
```

由于 Servlet 引擎与 Servlet 程序是按 Servlet 接口中约定的方法进行通信的，对于 Servlet 程序中的多个重载形式的 service 方法，Servlet 引擎只会去调用 Servlet 接口中定义的那个 service 方法，而其他形式的 service 方法则相当于 Servlet 程序内部定义的普通方法，Servlet 引擎根本就不知道它们的存在。

4.5.4 destroy 方法

destroy 方法在 Web 容器卸装 Servlet 之前被调用，显然，该方法在 Servlet 的生命期中也仅执行一次。可以通过覆盖 destroy 方法来完成与 init 方法相反的功能，释放被该 Servlet 打开的资源，例如，关闭数据库连接和 IO 流。GenericServlet 类实现的 destroy 方法已经满足通常的需要了，子类 Servlet 一般不必覆盖这个方法。

4.5.5 getServletConfig 方法

GetServletConfig 方法用于返回 Servlet 引擎通过 init 方法传递进来的那个 ServletConfig 对象的引用。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

4.5.6 getServletInfo 方法

GetServletInfo 方法用于返回 Servlet 的描述信息，Servlet 引擎要获得某个 Servlet 的描述信息时将调用这个方法。如果有必要，可以对这个方法进行覆盖，以便返回 Servlet 的作者、版本、版权等有关信息。

4.5.7 doXxx 方法

客户端可以用 HTTP 协议中规定的各种请求方式来访问 Servlet，对于不同的请求方式，Servlet 程序可能需要采用不同的方式进行处理。不管客户端以哪种请求方式访问 Servlet，Servlet 引擎都将调用 Servlet 的 service 方法，service 方法是处理所有请求方式的总入口。

显然，如果要在 service 方法中对不同的请求方式进行不同的处理，那么首先必须判断客户端的请求方式，然后通过选择语句对每种方式分别进行处理。HttpServlet 类为每一种 HTTP 请求方式都定义了一个对应的 doXxx 方法，例如，与 GET 请求方式对应的是 doGet 方法，与 POST 请求方式对应的是 doPost 方法。HttpServlet 中重载的 service 方法根据客户端的请求方式，分别调用与之对应的 doXxx 方法来完成具体的处理和响应细节，并将它接受的两个参数传递给该 doXxx 方法，例如，如果 HTTP 请求方式为 GET，则调用 doGet 方法。

下面是 HttpServlet 中定义的 doXxx 方法：

- doGet　用于处理 GET 请求
- doPost　用于处理 POST 请求
- doHead　用于处理 Head 请求
- doPut　用于处理 PUT 请求
- doDelete　用于处理 DELETE 请求
- doTrace　用于处理 TRACE 请求
- doOptions　用于处理 OPTIONS 请求

HttpServlet 中的 doXxx 方法分别用于对客户端的相应请求方式进行处理，一个 doXxx 方法是专用于处理某种请求方式的分岔入口，如果 Servlet 只想对某种或某几种请求方式进行处理，只需要覆盖这些方式各自对应的 doXxx 方法，而不用覆盖 HttpServlet 的 service 方法。HttpServlet 中的 doXxx 处理方式是回应给客户端一条不支持该请求方式的信息，这就好比一个人虽然接听了你的电话，但他回话的内容却是“不认识你”。因为 HEAD 请求方式除了不要求 Web 服务器返回实体正文部分外，它的操作与 GET 方式完全一样，所以只要覆盖了 doGet 方法，即使没有覆盖 doHead 方法，Servlet 也可以支持 HEAD 请求方式，但直接覆盖 doHead 方法对 HEAD 请求方式的响应效率会高些。在一般情况下，Servlet 只需处理 GET 和 POST 这两种请求方式，也就是只需覆盖 doGet 和 doPost 这两个方法。

如果自己编写的 Servlet 程序要对客户端的所有请求方式进行相同的处理，可以覆盖 HttpServlet 的 service 方法，并在其中编写相应的程序代码。注意，如果我们自己编写的 Servlet 程序覆盖了父类 HttpServlet 的 service 方法，Servlet 引擎将调用我们自己编写的这

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

个 Servlet 程序中的 service 方法，而不再调用父类 HttpServlet 中的 service 方法（除非在子类的 service 方法中使用 super.service()语句显式调用父类的 service 方法），所以，doXxx 方法也将不再被调用。

doGet 方法与 doPost 方法的作用

01 将例程 4-2 所示的 HelloServlet.java 中的 service 方法名修改成 doGet，修改后的程序代码如例程 4-6 所示。

例程 4-6 HelloServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloServlet extends HttpServlet
{
    //只是在下面将 service 方法名修改成了 doGet
    public void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException
    {
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<font size=30 color=red>www.it315.org</font><br>");
        out.println("<marquee>" + new java.util.Date() + "</marquee>");
        out.println("</html>");
    }
}
```

02 重新编译 HelloServlet.java 文件，并确保用编译后的结果覆盖了 d:\myWeb\确良Web-INF\classes 目录中的 HelloServlet.class 文件。在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/demo/hello.html
```

可以看到浏览器中显示出正常的响应结果。

03 在例程 4-6 所示的 HelloServlet 程序中，只覆盖了 doGet 方法，没有覆盖 doPost 方法，所以，客户端不能用 Post 方式访问 HelloServlet 映射的 URL 地址。在 d:\myWeb 目录中编写一个名为 TestHelloServlet.html 的网页文件，在这个网页中加入一个 FORM 表单，表单的提交方式设置为 POST，表单提交的目标地址为 HelloServlet 所映射的 URL，如例程 4-7 所示。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

例程 4-7 TestHelloServlet.html

```
<form action="demo/hello.html" method="post">
<input type=submit>
</form>
```

在浏览器地址栏中输入访问这个网页的地址：

http://localhost:8080/it315/TestHelloServlet.html

单击网页上显示的“提交查询内容”按钮，Web 服务器返回的结果如图 4.15 所示。

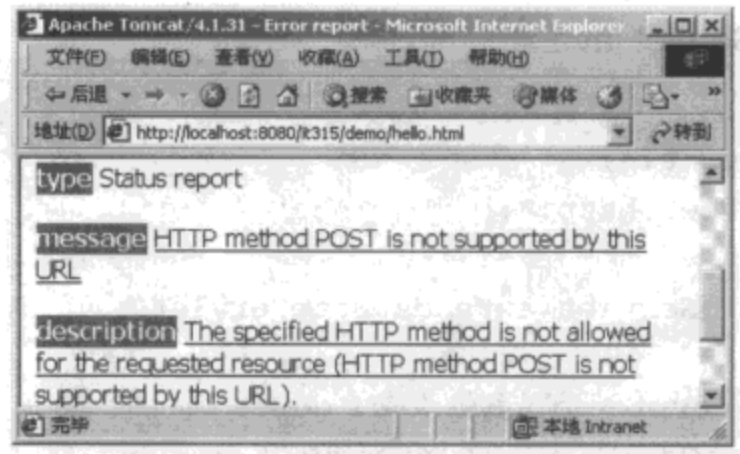


图 4.15

图 4.15 中提示的错误非常明显，即 HelloServlet 不支持客户端的 Post 请求方式，这就是因为 HelloServlet 类中没有覆盖 doPost 方法而导致的结果。

04 要想让 HelloServlet 既支持 GET 方式，也支持 POST 方式，还必须对 doPost 方法进行覆盖。如果 doPost 方法中的处理方式与 doGet 一样，那么就可以在 doPost 方法中直接调用 doGet 方法，修改后的程序代码如例程 4-8 所示。

例程 4-8 HelloServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException
    {
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<font size=30 color=red>www.it315.org</font><br>");
        out.println("<marquee>" + new java.util.Date() + "</marquee>");
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        out.println("</html>");
    }

    //下面是新增的 doPost 方法
    public void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,IOException
    {
        doGet(request, response);
    }
}
```

重新编译 HelloServlet.java 文件，并确保编译后的结果覆盖了 d:\myWeb\Web-INF\classes 目录中的 HelloServlet.class 文件。重复第 03 步中提交表单的操作过程，或者直接在图 4.15 中单击工具栏中的“刷新”按钮来重复上次的表单提交操作，浏览器中显示出了与 GET 请求方式一样的响应结果。

指点迷津

如何理解“在自己的程序代码中不能调用回调方法”

有人问：“不是说过在自己的程序代码中不能调用回调方法吗？上面的例子程序中的 doGet 方法就是一个回调用方法，可上面的程序代码却在 doPost 方法中调用了 doGet 方法，这是怎么回事啊？”

一个方法是否是回调方法，是从它的调用形式上来说的，而不是在定义时就被限定为只能作为回调方法使用。doGet 方法毕竟也是 java 语言中的一个方法，只要是一个方法，它就可以被同一个类中的其他方法调用，只是从这种调用方式上来讲，它不是一个回调方法，而是充当了一个普通方法。4.5.7 节中的 doGet 方法在被 Servlet 引擎调用时充当了一个回调方法，而在 doPost 方法中被调用时则充当了一个普通方法。

4.5.8 浏览器缓存内幕与 getLastModified 方法

在第 3 章中曾经讲解过，响应消息中的 Last-Modified 头字段可用于指定响应内容的最后更新时间，当客户机缓存此文档内容后，它在以后的请求消息中将根据 Last-Modified 头字段指定的时间来生成 If-Modified-Since 请求头字段，以指出缓存文档的最后更新时间。只有文档的修改时间比 If-Modified-Since 请求头指定的时间新时，服务器才会返回文档内容。如果自从 If-Modified-Since 指定的时间以来，网页内容没有发生修改，服务器将返回一个 304（Not Modified）状态码来表示浏览器缓存的版本是最新的，而不会向浏览器返回文档内容，浏览器则继续使用以前缓存的内容。通过这种方式，可以在一定程度上减少浏

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

览器与服务器之间的通信数据量，从而提高了通信效率。

一般情况下，浏览器都会缓存已经访问过的页面内容，Web 应用程序的开发人员必须对浏览器的缓存机制有所了解。单击 IE 浏览器的“工具”→“Internet 选项”菜单，打开“Internet 选项”对话框，接着再单击“常规”选项卡中的“Internet 临时文件”栏中的“设置”按钮，打开如图 4.16 所示的“设置”对话框。

图 4.16

在图 4.16 所示的“设置”对话框的“Internet 临时文件夹”栏中，可以看到浏览器保存所有缓存页面内容的文件夹的完整目录名称，对于 Administrator 用户来说，其默认的设置为“C:\Documents and Settings\Administrator\Local Settings\Temporary Internet Files”。单击“Internet 临时文件夹”栏中的“查看文件”按钮，可以打开这个文件夹，如图 4.17 所示。

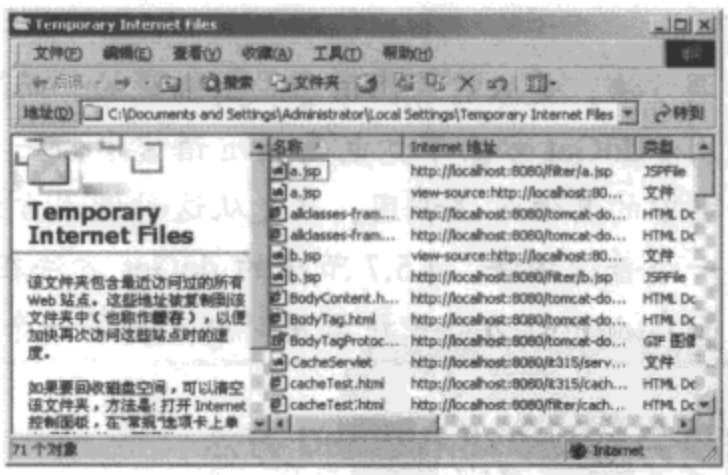


图 4.17

从图 4.16 中还可以看到，其“检查所存网页的较新版本”功能项的设置值有 4 个选项，只要先单击“设置”对话框标题栏中的问号按钮，然后再单击相应的选项，就可以看到每个选项的作用和意义。

- “每次访问此页时检查”选项表示浏览器每次访问一个页面时，不管浏览器是否缓存过此页面，都要向服务器发出访问请求。这种设置的优点是实时性很强，肯定能

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

够访问到网页的最新内容，但是如果网页内容很少更新，这种设置的访问效率就比较低了。

- “每次启动 Internet Explorer 时检查”选项表示在浏览器的每次启动运行期间，在第一次访问一个页面时，不管浏览器是否缓存过此页面，都要向服务器发出访问请求，但是在浏览器的本次启动运行期间对该页面的后续访问，浏览器将不再向服务器发出访问请求，而是直接使用缓存中的内容。这种设置具有较高的访问效率，同时也兼顾了较好的实时性，它可以保证每次启动浏览器后看到的都是最新的网页内容。
- “自动”选项与“每次启动 Internet Explorer 时检查”选项的功能相似，只是对图像的访问有所不同，如果随着时间的推移，浏览器发现网页上的图像更新并不频繁，这样，即使浏览器在对某个已缓存的图像执行本次启动运行以来的第一次访问时，它也不一定会向服务器发出访问请求，而是干脆直接使用缓存中的内容。“自动”选项是浏览器的默认设置，所以，几乎所有人的浏览器都是按照这种方式工作的，这个选项的作用和意义应该成为读者熟悉的重点。
- “不检查”选项表示浏览器不管在什么情况下访问一个页面时，只要能够在本地找到此页面的缓存信息，浏览器就不会向服务器发出访问请求，而是直接使用缓存的内容。这种设置的优点是访问效率很高，但是如果服务器端的网页内容更新后，浏览器看到的内容很可能是过期的内容。

在浏览器的“检查所存网页的较新版本”的功能项采用默认的“自动”设置项的情况下，如果浏览器刚刚访问过一个网页，服务器端就更新了这个网页的内容，当浏览器在关闭前又重新访问这个页面时，用户看到的将不是更新的网页内容，而是过期的网页内容。为了提高浏览效率，在访问静态的网页内容时，这么一点小概率的过期信息还是应该允许的，并且这些过期信息也不会造成什么不好的后果，就像你偶尔有一次看到了前一天发生的新闻，而不是当天的新闻，这又有什么问题呢？

可是，如果浏览器访问的是一个动态网页，这就要求浏览器在其整个运行期间的每次访问都能看到最新的内容，例如，销售一件商品后再回到商品库存的显示页面时，看到的就应该是更新的库存数据，而不应该是先前看到的内容。仅仅根据被访问页面的资源名称，浏览器是无法知道商品库存的显示页面是属于动态内容，还是属于静态内容。对于这种情况，浏览器将根据响应消息中是否包含 Last-Modified 头字段来进行处理，如果响应消息中没有包含 Last-Modified 头字段，它将在每次访问此页面时都向服务器发出访问请求，否则，它仅在每次启动运行后的第一次访问此页面时才向服务器发出访问请求，而在启动运行期间对此页面的后续访问都不再向服务器发出访问请求。

HttpServlet 类为 If-Modified-Since 请求头和 Last-Modified 头字段的这种应用提供了处理机制。在 HttpServlet 类中定义了一个 getLastModified 方法，getLastModified 方法的返回值可以影响浏览器如何处理和利用缓存内容，该方法的完整语法定义如下：

```
protected long getLastModified(HttpServletRequest req)
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

其中的返回值代表 Servlet 当前输出的响应内容的修改时间，是自 1970 年 1 月 1 日的 0 点 0 分 0 秒开始计算的一个毫秒数。getLastModified 方法是一个回调方法，由 HttpServlet 类的 service 方法调用，HttpServlet 类的 service 方法可以根据这个返回值在响应消息中自动生成 Last-Modified 头字段。HttpServlet 类中定义的 getLastModified 方法总是返回一个负数，在 HttpServlet 子类中可以对这个方法进行覆盖。

当继承了 HttpServlet 类的 Servlet 程序接收到一个 GET 方式的访问请求时，HttpServlet 中重载的 service 方法在调用 doGet 方法之前，它还将先调用 getLastModified 方法，并根据 getLastModified 方法的返回值来决定是否调用 doGet 方法和在响应消息中是否生成 Last-Modified 头字段，具体规则如下：

- 当 getLastModified 方法返回一个负数时，不管请求消息中的情况怎样，service 方法都将直接调用 doGet 方法来生成响应内容，这正是 HTTPServlet 类中定义的 getLastModified 方法的行为；
- 当 getLastModified 方法返回一个正数，且请求消息中没有包含 If-Modified-Since 请求头时（这往往出现在对某个资源的第一次访问时），或者请求消息中包含的 If-Modified-Since 请求头中的时间值比 getLastModified 方法返回的时间值旧时，service 方法将根据 getLastModified 方法的返回值生成一个 Last-Modified 头字段，然后调用 doGet 方法生成响应内容；
- 当 getLastModified 方法返回一个正数时，且请求消息中包含的 If-Modified-Since 请求头中的时间值比 getLastModified 方法返回的时间值新或者与之相同时，service 方法将不调用 doGet 方法，而是向浏览器返回一个 304（Not Modified）状态码来通知浏览器继续使用以前缓存的内容。

揭开浏览器缓存的奥秘

01 编写一个名为 CacheServlet 的 Servlet 程序，在其 doGet 方法中向浏览器和 Tomcat 的命令行窗口中都打印出当前的时间值，getLastModified 方法也是向 Tomcat 的命令行窗口中打印出当前的时间值和返回当前时间值，这里先将 getLastModified 方法注释掉，如例程 4-9 所示。

例程 4-9 CacheServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class CacheServlet extends HttpServlet
{
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
    public void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        PrintWriter out = response.getWriter();
        long now = System.currentTimeMillis();
        out.println("doGet:" + now);
        System.out.println("doGet:" + now);
    }

    /*protected long getLastModified(HttpServletRequest req)
    {
        long now = System.currentTimeMillis();
        System.out.println("getLastModified:" + now);
        return now;
    }*/
}
```

编译 CacheServlet.java 文件，确保编译后生成的 class 文件放置进了 d:\myWeb\Web-INF\classes 目录中。

02 在 d:\myWeb 目录中编写一个名称为 CacheTest.html 的网页文件，如例程 4-10 所示。

例程 4-10 CacheTest.html

```
<a href="servlet/CacheServlet">缓存测试</a>
```

03 为了便于查看浏览器如何生成缓存内容，最好是先删除掉 Internet 临时文件夹中保存的所有缓存内容。单击 IE 浏览器的“工具”→“Internet 选项”菜单，打开如图 4.16 所示的“Internet 选项”对话框，接着再单击“常规”选项卡中的“Internet 临时文件”栏中的“删除文件”按钮，这就删除了 Internet 临时文件夹中保存的所有缓存内容。单击“常规”选项卡中的“Internet 临时文件”栏中的“设置”按钮，在打开的“设置”对话框框中单击“Internet 临时文件夹”栏中的“查看文件”按钮，打开如图 4.17 所示的 Internet 临时文件夹，可以看到其中已经不再有任何缓存的文件。另外，在“设置”对话框中还需要将“检查所存网页的较新版本”的功能项恢复为默认的“自动”。

在浏览器地址栏中输入如下地址：

http://localhost:8080/it315/CacheTest.html

在浏览器窗口中显示的结果页面中，单击“缓存测试”超链接访问 CacheServlet。再次打开 Internet 临时文件夹，这时可以看到其中生成了刚才访问过的 CacheTest.html 和 CacheServlet 这两个页面的缓存文件，如图 4.18 所示。选中其中的 CacheServlet 缓存文件，从 Windows 资源管理器窗口中显示出的摘要信息中可以看到，CacheServlet 缓存文件中记

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

录的上次修改时间为“无”。

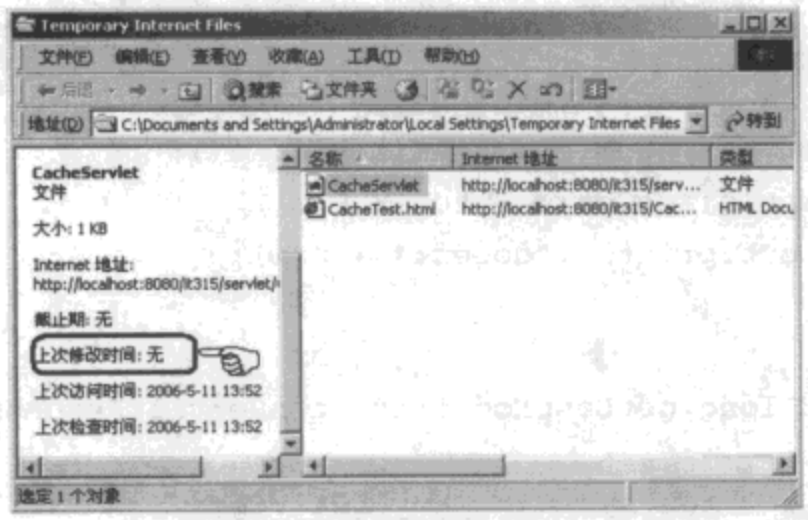


图 4.18

04 在命令行窗口中执行 telnet 127.0.0.1 8080 命令，连接上 Tomcat Web 服务器后，接着在 Telnet 程序命令窗口中，输入如下内容：

```
GET /it315/servlet/CacheServlet HTTP/1.1<回车>
Host:<空格><回车>
<回车>
```

Telnet 窗口中显示出的结果如图 4.19 所示。

图 4.19

从图 4.19 中可以看到，CacheServlet 返回的响应消息中没有 Last-Modified 头字段，这正是图 4.18 中显示的 CacheServlet 缓存文件的上次修改时间为“无”的原因。

05 回到浏览器窗口中，简要记住浏览器窗口和 Tomcat 的命令行窗口中显示出的时间值，然后反复单击浏览器工具栏中的“后退”和“前进”按钮，可以看到每次显示出的 CacheServlet 页面内容都没有变化，Tomcat 的命令行窗口中也没有打印出新的信息。直接在浏览器地址栏中输入 CacheServlet 的访问地址，结果也是如此。这说明用浏览器的“后退”或“前进”方式访问已缓存的页面时，或者直接在浏览器地址栏中访问已缓存的页面时，浏览器将直接调用缓存的内容，而不向服务器发出新的访问请求。

在图 4.18 所示的 Windows 资源管理器窗口中删除 CacheServlet 页面的缓存文件，再次使用“后退”和“前进”按钮访问 CacheServlet，或者直接在浏览器地址栏中输入

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

CacheServlet 的访问地址，这时候浏览器窗口和 Tomcat 的命令行窗口中都将显示出新的时间值。这说明，在缓存文件不再存在的情况下，使用“后退”和“前进”按钮访问 CacheServlet 时，或者直接在浏览器地址栏中输入 CacheServlet 的访问地址，浏览器将向服务器发出新的访问请求。

通过浏览器工具栏中的“后退”或“前进”按钮回到 CacheTest.html 页面，然后单击其中的超链接来访问 CacheServlet，这时可以看到浏览器窗口和 Tomcat 的命令行窗口中显示出了新的时间值。重复这个过程，可以看到 CacheServlet 页面每次都能显示出新的时间值，这说明通过其他网页文档中的超链接来调用 CacheServlet 时，浏览器将向服务器发出新的访问请求，而不是调用已缓存的内容。

06 修改 CacheServlet.java 源文件，取消对 getLastModified 方法的注释，重新编译 CacheServlet.java 源文件，等待 Tomcat 重新装载 CacheServlet 后，在浏览器窗口中刷新对 CacheServlet 的访问。打开 Internet 临时文件夹（如果该文件夹原来已经打开，则按 F5 键刷新一下），选中其中的 CacheServlet 缓存文件，从 Windows 资源管理器窗口中显示出的摘要信息中可以看到，在这次的 CacheServlet 缓存文件中有了一个上次修改时间的记录信息，如图 4.20 所示。

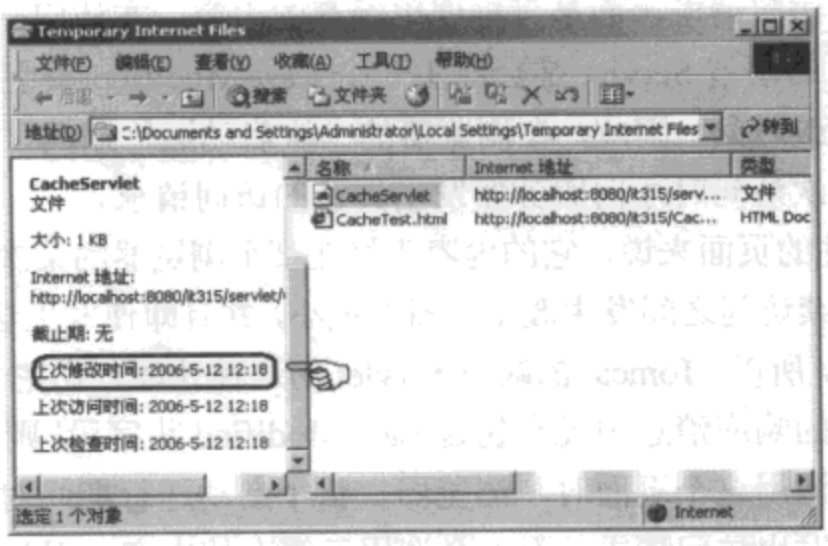


图 4.20

07 重复第 **04** 步的操作，此时，Telnet 窗口中显示出的结果如图 4.21 所示。

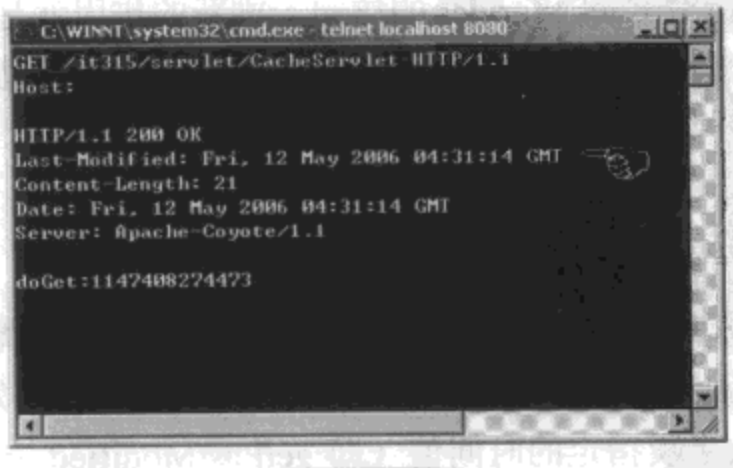


图 4.21

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

从图 4.21 中可以看到，CacheServlet 这次返回的响应消息中多了一个 Last-Modified 头字段，这正是图 4.20 中显示的 CacheServlet 缓存文件有了一个上次修改时间的记录信息的原因。

08 回到浏览器窗口中，简要记住一下浏览器窗口和 Tomcat 的命令行窗口中显示出的时间值，然后反复单击浏览器工具栏中的“后退”和“前进”按钮，可以看到每次显示出的 CacheServlet 页面内容都没有变化，Tomcat 的命令行窗口中也没有打印出新的信息。直接在浏览器地址栏中输入 CacheServlet 的访问地址，结果也是如此。这种情况与在步骤 **05** 中看到的实验效果完全一样。

通过浏览器工具栏中的“后退”或“前进”按钮回到 CacheTest.html 页面，单击其中的超链接来访问 CacheServlet，这时看到浏览器窗口和 Tomcat 的命令行窗口中显示出的内容仍然没有变化，多次重复这个过程，看到的结果还是没有变化。这种情况与步骤 **05** 中的看到的实验效果截然不同，通过两者的实验效果的对比，我们可以得出如下结论：

- 如果某个页面的响应消息中包含 Last-Modified 头字段，则当通过其他网页文档中的超链接来调用这个页面时，浏览器只在每次启动后的第一次访问这个页面时才向服务器发出访问请求，对于在这次启动运行期间对该页面的后续访问，浏览器将不再向服务器发出访问请求，而是直接调用缓存的内容。在访问一个普通的 HTML 文件时，Tomcat 的缺省 Servlet 都会产生一个 Last-Modified 头字段来说明这个 HTML 文件的最新修改时间，因此，在浏览器的一次启动运行期间，它只在对某个 HTML 文件进行第一次访问时才向服务器发出真正的访问请求。
- 对 HTML 文件的页面来说，它的内容正好在某个浏览器的某次运行期间的对该页面的第一次和后续访问之间发生改变的概率很小，并且即使发生这种情况，也不会出现什么大的问题，所以，Tomcat 的缺省 Servlet 的这种处理方式完全是合理的和应该的。
- 如果某个页面的响应消息中没有包含 Last-Modified 头字段，则当通过其他网页文档中的超链接来调用这个页面时，浏览器在整个启动运行期间对该页面的每次访问，都将向服务器发出访问请求，而不会调用已缓存的内容。因此，只要动态网页程序中没有产生 Last-Modified 头字段，就不用担心通过超链接再次访问该动态网页程序时会出现浏览器不向服务器发生访问请求的问题。

在浏览器窗口中刷新对 CacheServlet 的访问，刷新的作用就是让浏览器务必向服务器发出访问请求，这时候浏览器窗口中显示出了新的内容。查看启动 Tomcat 的命令行窗口，可以看到其中打印出类似如下的两行新信息：

```
getLastModified:1147408274473
doGet:1147408274473
```

这说明 CacheServlet 调用了 getLastModified 方法后，接着又调用了 doGet 方法。因为 getLastModified 方法返回的是当前时间，它肯定比浏览器发出的请求消息中的 If-Modified-Since 请求头字段指定的时间值（即 getLastModified 方法上次被调用时返回的“当前时间”）要新，所以，CacheServlet 调用完 getLastModified 方法后，接着又调用了 doGet

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

方法，于是在浏览器中就看到了新的内容。

09 修改 CacheServlet.java 源文件，让 getLastModified 方法中的最后那条 return 语句返回一个固定的时间值，如下所示：

```
protected long getLastModified(HttpServletRequest req)
{
    long now = System.currentTimeMillis();
    System.out.println("getLastModified:" + now);
    return /*now*/1234;
}
```

重新编译 CacheServlet.java 源文件，等待 Tomcat 重新装载 CacheServlet 后，新启动一个浏览器程序访问 CacheTest.html 页面，然后单击 CacheTest.html 页面中的“缓存测试”超链接访问 CacheServlet，这时显示出来的 CacheServlet 页面的内容还是在步骤 **08** 中的最后看到的内容，没有发生改变！查看启动 Tomcat 的命令行窗口，可以看到其中仅打印出了一行类似如下的新信息：

```
getLastModified:1147414361586
```

这个现象说明，浏览器刚才确实向服务器发出了访问请求。由于浏览器发出的请求消息中带有一个 If-Modified-Since 请求头字段，其设置值为上次的 Last-Modified 头字段所指定的时间，而 CacheServlet 中的 getLastModified 方法这次返回的时间值比 Last-Modified 头字段指定的时间值更旧（在一般情况下应该是相等，这里为了简化程序的编写，仅仅是返回 1234 这个比较陈旧的时间值，但不影响实验效果），所以，CacheServlet 没有调用 doGet 方法（Tomcat 的命令行窗口中没有打印出 doGet 方法被调用的提示信息），而是向浏览器返回一个 304（Not Modified）状态码来表示浏览器缓存的版本是最新的，从而导致浏览器仍旧显示出了上次缓存的内容。

在浏览器窗口中刷新对 CacheServlet 的访问，这时浏览器中显示的内容依然如故，查看启动 Tomcat 的命令行窗口，可以看到其中又只是打印出了一行 getLastModified 方法被调用的提示信息。可见，如果浏览器缓存的某个页面的响应消息中包含有 Last-Modified 头字段，当浏览器再次向服务器发出针对该页面的访问请求时，只要处理这个页面的 Servlet 程序的 getLastModified 方法返回的时间值比上次的 Last-Modified 头字段的时间值旧或者与之相同时，服务器就不会返回新的内容。

10 重复步骤 **07** 的 telnet 访问操作，等待服务器返回响应结果后，接着输入如下内容：

```
GET /it315/servlet/CacheServlet HTTP/1.1<回车>
Host:<空格><回车>
If-Modified-Since:<空格>Thu, 01 Jan 1970 00:00:01 GMT
<回车>
```

其中的 If-Modified-Since 头字段中的值是从刚才返回的 Last-Modified 头字段中复制而

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

得到的，这时候可以看到服务器只返回了一个 304（Not Modified）状态码，而没有返回任何实体内容。接着输入如下内容：

```
GET /it315/servlet/CacheServlet HTTP/1.1<回车>
Host:<空格><回车>
If-Modified-Since:<空格>Thu, 01 Jan 1970 00:00:00 GMT
<回车>
```

注意 其中的 If-Modified-Since 头字段中的值比上一次的值小了 1 秒，这时候可以看到服务器返回了 200 状态码和新的实体内容，如图 4.22 所示。

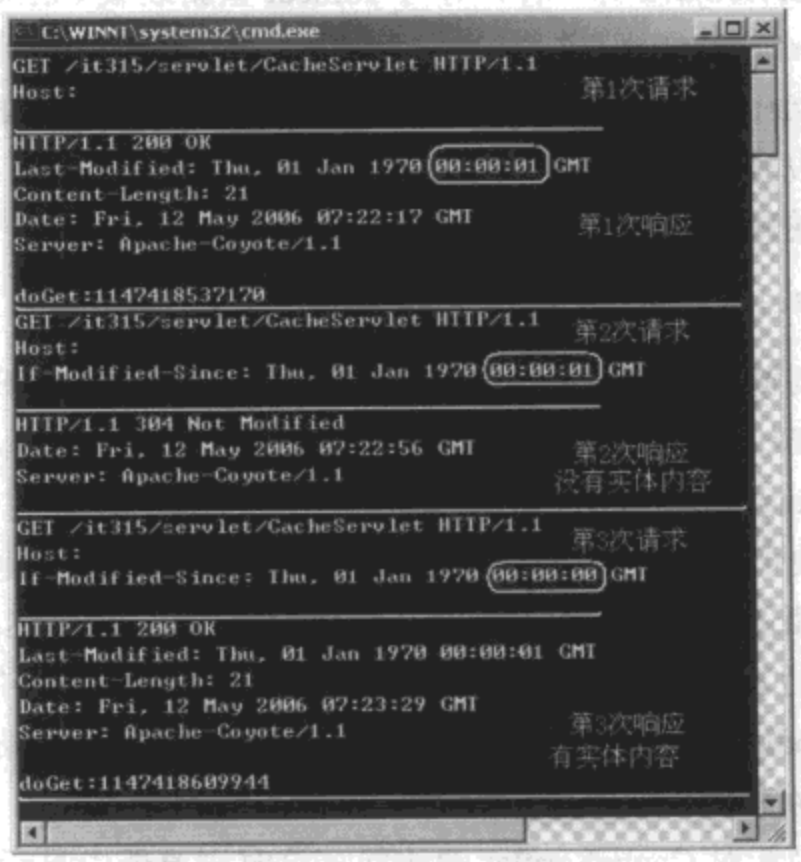


图 4.22

4.6 ServletContext 接口

每个 Web 应用程序都是一个独立的 Servlet 容器，每个 Web 应用程序分别用一个 ServletContext 对象来表示，ServletContext 接口定义了 ServletContext 对象需要对外提供的方法，Servlet 程序通过这些方法与 Servlet 容器进行通信。Servlet 引擎为每个 Web 应用程序都创建一个对应的 ServletContext 对象，ServletContext 对象被包含在 ServletConfig 对象中，调用 ServletConfig.getServletContext 方法可以返回 ServletContext 对象的引用。在 Servlet 容器初始化 Servlet 对象时，ServletContext 对象随着 ServletConfig 对象提供给 Servlet。与 Servlet API 中的其他接口一样，ServletContext 接口的实现类也是由 Servlet 引擎提供的。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

4.6.1 获取 Web 应用程序的初始化参数

在 server.xml 文件和 Web.xml 文件中都可以为某个 Web 应用程序设置若干初始化参数，ServletContext 接口中定义了获得这些初始化参数信息的一些方法。为 Web 应用程序设置初始化参数的好处在于不需要修改 Servlet 源程序，就可以改变整个 Web 应用程序范围内的一些参数信息，例如，一个 Web 应用程序中的多个 Servlet 程序都要输出当前站点的公司名称，而这个 Web 应用程序可能会交给多个公司去使用，如果将公司名称作为 Web 应用程序的初始化参数进行设置，那么，只需要在 Web.xml 文件中修改公司名称，所有的 Servlet 程序输出的公司名称就都会随之更改。ServletContext.getInitParameterNames 方法用于返回一个包含 Web 应用程序的所有初始化参数名称的 Enumeration 集合对象；ServletContext.getInitParameter 方法用于返回某个指定名称的初始化参数值，如果不存在指定名称的参数，返回值为 null。

如果要在 server.xml 文件中为某个 Web 应用程序设置初始化参数，需要在该 Web 应用程序所对应的<Context>元素中增加<Parameter>子元素，如下所示：

```
<Context path="/it315" docBase="d:\myWeb" debug="0" reloadable="true">
    <Parameter name="companyName" value="it315" override="false"/>
</Context>
```

<Context>元素中的每个<Parameter>子元素对应一个初始化参数，其中的 override 属性用于指定在 Web 应用程序的 Web.xml 文件中设置的同名初始化参数是否覆盖这里的设置，当该属性值为 false 时表示不允许覆盖，默认值为 true，即允许覆盖。

如果要在 Web 应用程序的 Web.xml 文件中设置初始化参数，则需要在根元素<Web-app>中增加<context-param>子元素，如下所示：

```
<context-param>
    <param-name>companyName</param-name>
    <param-value>it315_new</param-value>
</context-param>
```

Web.xml 文件中的一个<context-param>元素设置一个初始化参数，其中的<param-name>和<param-value>子元素分别用于设置参数值和参数名。

Web 应用程序的初始化参数的设置与应用

01 用 UltraEdit 软件打开 tomcat 4.1.31 版本的<Tomcat 主目录>/conf 目录下的 server.xml 文件，将/it315 这个 Web 应用程序所对应的<Context>元素修改成如下形式：

```
<Context path="/it315" docBase="d:\myWeb" debug="0" reloadable="true">
    <Parameter name="companyName" value="it315" override="false"/>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
</Context>
```

02 在 d:\myWeb\Web-INF\Web.xml 文件中的根元素<Web-app>下面，插入如下一段内容：

```
<context-param>
    <param-name>companyName</param-name>
    <param-value>it315_new</param-value>
</context-param>
<context-param>
    <param-name>WebSite</param-name>
    <param-value>http://www.it315.org</param-value>
</context-param>
```

注意，这里故意设置了一个已在 server.xml 文件设置过的同名参数，即名称为 companyName 的参数，该参数在两处的设置值不同。

03 按例程 4-11 编写一个名为 ContextParameterServlet 的 Servlet 程序。

例程 4-11 ContextParameterServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Enumeration;

public class ContextParameterServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        PrintWriter out = response.getWriter();
        out.println("<html>");
        Enumeration e = getServletContext().getInitParameterNames();
        out.println("Context Parameter:" + "<br>");
        while (e.hasMoreElements())
        {
            String key = (String)e.nextElement();
            String value = getServletContext().getInitParameter(key);
            out.println("  " + key + "=" + value + "<br>");
        }
        out.println("</html>");
    }
}
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

编译 ContextParameterServlet.java 文件，确保生成的 class 文件放置进了 d:\myWeb\Web-INF\classes 目录中。

04 启动或重新启动 Tomcat，在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/servlet/ContextParameterServlet
```

浏览器中显示的结果如图 4.23 所示。

图 4.23

从图 4.23 中可以看到，由于在 server.xml 文件中设置了<Parameter>参数的 override 属性为 false，即使在 Web.xml 文件有同名的初始化参数设置，SerlvetContext. getInitParameter 方法所返回的结果仍为 server.xml 文件中的设置值。

4.6.2 记录日志

在一个完善的应用系统中，通常都应包含日志功能，即将程序运行过程中发生的一些特殊事件、重要信息和异常错误记录到一个或多个文件中，以便系统管理人员和程序维护人员能够随时查看和了解到这些信息。在产品开发阶段，开发人员也经常要使用日志记录来跟踪程序的执行流程、查找程序的错误和分析程序的性能。

ServletContext 接口中定义了两个重载的 log 方法来记录日志，它们的定义如下：

- public void log(java.lang.String msg)
- public void log(java.lang.String message,java.lang.Throwable t)

这两个 log 方法用于向 Web 服务器的日志文件中写入信息。第一个 log 方法只接受一个 String 类型的参数，它通常用于记录一些特殊事件和重要信息。第二个 log 方法除了接受一个 String 类型的参数外，它还接受一个 Throwable 类型（Exception 类的父类）的参数，这个方法通常用于记录程序运行中的异常信息。GenericServlet 类中也定义了两个 log 方法，它们分别调用 ServletContext 对象中对应的 log 方法，这样，在 Servlet 中就可以简化对 log 方法的调用，即不用先得到 ServletContext 对象的引用后，再调用它的 log 方法，而是直接调用 Servlet 自己的 log 方法。

log 方法记录的日志文件名称和存储路径因 Web 服务器而异，Tomcat 中的日志文件的存储路径和名称在 servet.xml 文件中进行设置。每个 Web 应用程序都可以设置其单独的日

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

志文件，log 方法将日志信息写入到当前 Web 应用程序的日志文件中。在 server.xml 文件中，使用<logger>元素来设置日志文件的相关信息，如下所示：

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

其中，className 属性用于指定具体负责日志记录操作的 Java 类。directory 属性用于指定所创建的日志文件的存储目录，如果指定的是一个相对路径，则其基准路径为 $CATALINA_HOME 环境变量所设置的目录，directory 的默认值为相对目录“logs”。prefix 属性用于指定所创建的日志文件的名称的前缀部分，suffix 属性用于指定日志文件的名称的后缀部分，如果不想要前缀或后缀部分，则将相应的属性设置为空字符串即可。timestamp 属性指定记录的每条日志信息中是否自动加上一个时间戳，其默认值是 false。

<logger>元素可以嵌套在<Engine>、<Host>和<Context>元素之中，如果里层元素中没有单独设置<logger>元素，它将继承外层元素中设置的<logger>元素，否则，里层元素采用自己设置的<logger>元素，即在里层元素的范围内覆盖外层元素中设置的<logger>元素。

Web 应用程序的日志记录功能

01 用 UltraEdit 软件打开 tomcat 4.1.31 版本的<Tomcat 主目录>/conf 目录下的 server.xml 文件，可以查找到多个<logger>元素，其中一个<logger>元素的设置如下：

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

参照这个<logger>元素的设置，将/it315 这个 Web 应用程序所对应的<Context>元素修改成如下形式：

```
<Context path="/it315" docBase="d:\myWeb" debug="0" reloadable="true">
    <Parameter name="companyName" value="it315" override="false"/>
    <Logger className="org.apache.catalina.logger.FileLogger"
        directory="logs\it315"  prefix="" suffix=".txt" timestamp="true"/>
</Context>
```

读者在编写其中的 logger 元素部分时，应该先复制前面查找到的 logger 元素，然后在此基础上进行修改。保存 server.xml 文件后，重新启动 Tomcat，可以看到<Tomcat 主目录>/logs 目录下自动增加了一个名为 it315 的子目录，在这个目录中创建了一个以当前日期为名称的.txt 文件，并且这个文件中记录了/it315 这个 Web 应用程序的一些启动加载信息。

02 按例程 4-12 编写一个名为 LogTestServlet 的 Servlet 程序，该程序中调用了 log 方法的两个重载形式。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

例程 4-12 LogTestServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class LogTestServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        /*getServletContext().*/log("LogTestServlet is running.");
        try
        {
            //刻意营造一个异常
            int x = 3/0;
        }
        catch(Exception e)
        {
            /*this.*/log("LogTestServlet happen exception",e);
        }
    }
}
```

编译 LogTestServlet.java 文件，确保生成的 class 文件放置进了 d:\myWeb\Web-INF\classes 目录中。

03 在浏览器地址栏中输入如下地址：

http://localhost:8080/it315/servlet/LogTestServlet

再次查看<Tomcat 主目录>/logs/it315 目录下的日志文件内容，可以看到两个 log 方法所记录的信息。

多学两招

JDK 中提供的日志记录模块

JDK 中提供了一个功能非常强大的日志记录模块，它允许系统人员通过配置文件来指定输出哪些级别的日志信息，以什么格式输出日志信息，以及将日志信息输出到哪个目标设备，如控制台、文件、管理员邮箱等。

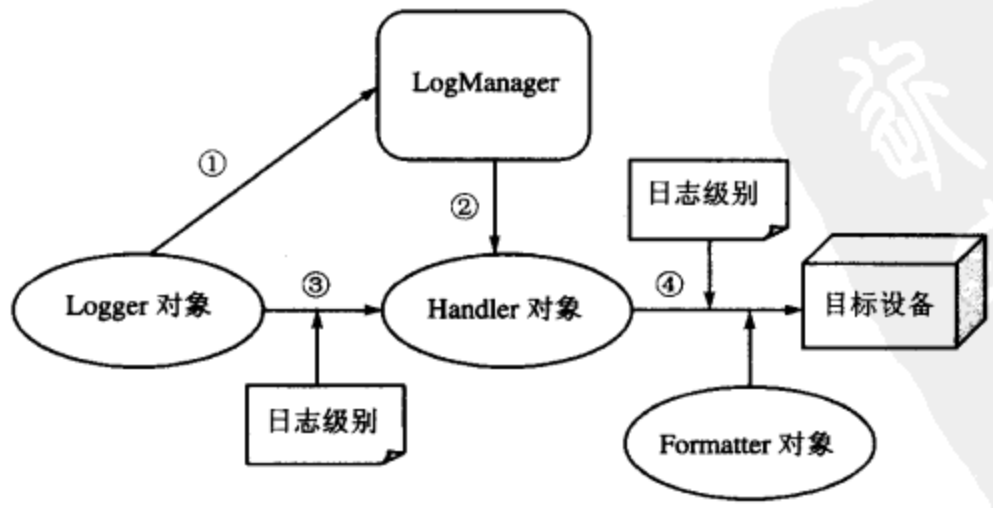
JDK 中的日志记录模块的所有类都位于 java.util.logging 包中，其中，只有

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

工作或扩展日志记录功能时才需要使用。JDK 中的日志记录模块将日志信息定义成了 7 个级别：FINEST、FINER、FINE、CONFIG、INFO、WARNING 和 SEVERE，它们的级别按这里排列的顺序依次递增。FINEST 级别最低，可用于记录一些非常普通的调试信息；SEVERE 级别最高，用于记录一些非常严重和错误的事件。logger 类中定义了多组方法来分别生成这 7 个级别的日志信息，其中最简单的就是与这些级别同名的一组方法，例如，finest()方法用于生成 FINEST 级别的日志信息，finer()方法用于生成 FINER 级别的日志信息，……关于 logger 类中的其他生成日志信息的方法，请参看 logger 类的帮助文档。

一个 logger 对象用于记录一个特定的子系统或程序部件的日志信息，一个应用程序中可以有多个 logger 对象。每个 logger 对象都有一个名称，其名称通常采用包名的形式，用点（.）对名称进行层次划分。根据 logger 对象的名称，logger 对象可以在逻辑上形成一种父子关系，例如，名称为“org.it315.test”的 logger 对象就是名称为“org.it315”的 logger 对象的子级。最顶级的 logger 对象称为根 logger 对象，用空字符串（""）表示。调用 Logger.getLogger 方法可以获得一个 logger 对象，logger 对象的名称通过传递给 getLogger 方法的参数指定，如果指定名称的 logger 对象已经存在，getLogger 方法返回已经存在的 logger 对象，否则，getLogger 方法创建一个新的 logger 对象。在应用程序中创建的每个 logger 对象存储在一个 LogManager 对象中，一个 JVM 中只能有一个 LogManager 对象。这个全局惟一的 LogManager 实例对象在 LogManager 类加载时被创建，logger.getLogger 方法内部首先调用 LogManager.getLogManager()方法获取 LogManager 实例对象，然后从中检索指定名称的 logger 对象。如果 logManager 对象中存在指定名称的 logger 对象，logger.getLogger 方法返回该 logger 对象；如果 logManager 对象中不存在指定名称的 logger 对象，logger.getLogger 方法内部则创建一个新的 logger 对象，并以指定的名称将该 logger 对象注册到 logManager 对象中后返回。实际上，应用程序开发人员不用关心这些细节，也不用接触 LogManager 对象，只需知道调用 Logger.getLogger 方法来获得一个 logger 对象即可。JDK 中的日志记录模块的运行原理如图 4.24 所示。



http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

logger 对象产生的日志信息需要交给一个或多个 Handler（处理器）对象去处理，Handler 对象的作用就是把日志信息写入到一个真正的外部设备中。显然，要想将某个 logger 对象记录的日志信息写入到某种设备中去，需要把与设备对应的 Handler 对象注册到该 logger 对象上，调用某个 logger 对象的 addHandler 方法可以完成这一任务。在默认情况下，一个 logger 对象也会把日志信息交给父级 logger 对象的 Handler 去处理，并且是一直追溯到根 logger 对象的 Handler。另外，还可以调用 logger 对象的 setLevel 方法，来指定把该 logger 对象生成的哪个级别的日志信息交给 Handler 去处理，当指定某个日志级别后，只有这个级别及比这个级别更高的日志信息才交给 Handler 去处理。如果为某个 logger 对象指定的日志级别为 null，它将继承它父级 logger 对象的级别设置。

除了可以调用 logger 对象的方法来为其指定 Handler 对象和要记录的日志级别外，还可以通过属性文件来动态配置。属性文件可以通过两种方式指定：一种方式是使用默认的<JRE_HOME>/lib/log.properties 文件；另一种方式则是为 JVM 设置 java.util.logging.config.file 系统属性，即采用如下方式：

```
java -Djava.util.logging.config.file=<属性文件名> <类名>
```

其中，属性文件必须位于 CLASSPATH 环境变量包含的某个目录中。如果使用了第二种方式，那么默认的<JRE_HOME>/lib/log.properties 文件将不再起作用。例程 4-13 是一个简单的 log.properties 文件。

例程 4-13 log.properties

```
handlers=java.util.logging.ConsoleHandler
.level=INFO
org.it315.xxx.level=FINER
java.util.logging.ConsoleHandler.level=CONFIG
java.util.logging.ConsoleHandler.formatter=java.util.logging.
SimpleFormatter
```

在 log.properties 属性文件的 handlers 属性中可以设置若干 Handler 类名，每个 Handler 类名之间用空格分隔，logManager 对象将为 handlers 属性中设置的每个 Handler 类创建一个实例对象，并将这些实例对象都注册为根 logger 对象的 Handler 处理对象。属性文件中的以“.level”为后缀的属性名用于设置相应 logger 对象的日志级别，例如，名称为“org.it315.level”的属性用于设置名称为“org.it315”的 logger 对象的日志级别，名称为“.level”的属性用于设置根 logger 对象的日志级别。

Handler 对象将每条日志信息写入到设备中时，需要采用一定的格式，这个格式通过为 Handler 对象指定的一个 Formatter 对象来控制。另外，Handler 对象将日志信息写入到目标设备中时，也可以调用 Handler 对象的 setLevel 方法来指定将哪个级别的日志信息写到设备中去，当指定某个级别后，只有这个级别及比这个级别更高的日志信息才

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

会被真正记录到该设备上。Handler 对象的这些信息也可以通过上面所讲的属性文件设置，其属性名都是以 Handler 的完整类名为前缀，不同 Handler 类有不同的信息设置，具体信息需要分别参看各个 Handler 的帮助文档。

例程 4-14 是一个调用 JDK 中的日志记录模块的例子程序。

例程 4-14 LogTest.java

```
package org.it315;
import java.util.logging.Logger;

public class LogTest {
   private static Logger log = Logger.getLogger("org.it315.xxx");
   public static void main(String args[])
   {
       log.finest("the finest message");
       log.finer("finer message");
       log.fine("a fine message");
       log.config("some configuration message");
       log.info("a little bit of information");
       log.warning("a warning message");
       log.severe("a severe message");
   }
}
```

编译 LogTest.java 后，执行如下命令：

```
java org.it315.LogTest
```

程序在控制台窗口中打印出了如下所示的信息：

```
2005-12-15 17:32:16 org.it315.LogTest main
信息: a little bit of information
2005-12-15 17:32:17 org.it315.LogTest main
警告: a warning message
2005-12-15 17:32:17 org.it315.LogTest main
严重: a severe message
```

从打印结果中可以看到，并不是所有的记录信息都显示在了控制台中，这是因为在 <JRE_HOME>/lib/log.properties 文件中设置了如下一些信息：

```
handlers= java.util.logging.ConsoleHandler
.level= INFO
java.util.logging.ConsoleHandler.level = INFO
```

可见，logger 对象和 Handler 对象的日志级别都设置成了 INFO，所以，LogTest 程序中只输出了 INFO、WARNING 和 SEVERE 等级别的日志信息。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

将例程 4-13 所示的 log.properties 文件放置进 CLASSPATH 环境变量包含的某个目录中，这里将 log.properties 文件放置进 org.it315 包名的顶级包目录 org 的父目录中，即让 log.properties 文件与 org 目录位于同一目录，接着执行如下命令：

```
java -Djava.util.logging.config.file=log.properties org.it315.LogTest
```

程序在控制台窗口中打印出了如下所示的信息：

```
2005-12-15 17:55:55 org.it315.LogTest main
配置: some configuration message
2005-12-15 17:55:55 org.it315.LogTest main
信息: a little bit of information
2005-12-15 17:55:55 org.it315.LogTest main
警告: a warning message
2005-12-15 17:55:55 org.it315.LogTest main
严重: a severe message
```

尽管 log.properties 文件中设置的 logger 对象的日志级别为 FINER，但是由于 Handler 对象的日志级别为 CONFIG，所以，最终也只输出了 CONFIG 及更高级别的日志信息。注释掉 log.properties 文件中的 org.it315.xxx.level 属性设置，即将相应行修改成如下形式：

```
#org.it315.xxx.level=FINER
```

接着重复执行上次的命令，程序在控制台窗口中打印出了如下所示的信息：

```
2005-12-15 18:04:43 org.it315.LogTest main
信息: a little bit of information
2005-12-15 18:04:43 org.it315.LogTest main
警告: a warning message
2005-12-15 18:04:43 org.it315.LogTest main
严重: a severe message
```

这是由于 log.properties 文件中没有为 org.it315.xxx 这个 logger 对象指定日志级别，它将继承根 logger 对象的级别设置，即 org.it315.xxx 这个 logger 对象当前采用的日志级别为 INFO，这样，logger 对象交给 Handler 对象去处理的日志信息仅为 INFO 及更高级别的日志信息，所以，尽管 Handler 对象的日志级别为 CONFIG，但最终也只输出了 INFO 及更高级别的日志信息。

4.6.3 application 域范围的属性

由于一个 Web 应用程序中的所有 Servlet 都共享同一个 ServletContext 对象，所以，ServletContext 对象被称之为 application 对象（Web 应用程序对象）。application 对象（ServletContext 对象）内部有一个哈希表集合对象，存储进 application 对象内的哈希表集

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

合对象中的每对关键字/值被称为 application 对象的属性。由于一个 Web 应用程序中的所有 Servlet 程序都共享同一个 application 对象，所以 application 对象的属性可以被该 Web 应用程序内的所有 Servlet 程序访问，存储在 application 对象中的属性也被称之为 application 域范围的属性，application 域范围的属性可以被当做该 Web 应用程序范围内的全局变量使用。ServletContext 接口中定义了 4 个分别用于增加、删除、访问 application 域范围的属性的方法。

- getAttributeNames 方法

用于返回一个 Enumeration 集合对象，该 Enumeration 集合对象中包含了 application 对象中的所有属性的名称。

- getAttribute 方法

用于返回某个属性的值。

- removeAttribute 方法

用于删除某个属性。

- setAttribute 方法

用于增加一个属性，设置的属性名最好遵循与包名同样的命名习惯。如果该名称的属性已经存在，则用新的设置值替换原来的设置值；如果属性设置值为 null，则等效于使用 removeAttribute 方法。

一个 Servlet 程序可以发布到多个 Web 应用程序中运行，在每个 Web 应用程序中都会为该 Servlet 程序创建单独的实例对象，每个 Servlet 实例对象所使用的 application 对象是不同的，它们分别代表不同的 Web 应用程序。关于 application 域范围的属性的应用，将在 7.4.6 节中与 Session 域范围的属性进行对比讲解。

4.6.4 访问资源文件

ServletContext 接口中还定义了一些用于访问 Web 应用程序的内部资源文件的方法，这些资源文件是指 Web 应用程序内部的各种形式的文件，包括 Web-INF 目录中不能被外界访问的文件。如果将整个 Web 应用程序打包进一个 WAR 文件中，WAR 文件中将会包含一个名为 META-INF 的目录，这个目录中的内容不能被客户端直接访问，但是它里面的内容可以被 ServletContext 接口中定义的这些方法访问。

- getResourcePaths 方法

返回一个 java.util.Set 集合对象，该 Set 集合对象中包含某个资源目录中的所有子目录和文件的路径名称，每个路径名称都以相对于整个 Web 应用程序的根目录的形式表示，即都用“/”开头，其中的每个目录路径的最后也都有一个“/”结尾。如果某个 Web 应用程序中包含下面的目录和文件：

```
/welcome.html
/Web-INF/Web.xml
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

那么，getResourcePaths("/")返回的结果是["/welcome.html","/Web-INF/"]，getResource Paths ("/Web-INF/")返回的结果是["/Web-INF/Web.xml"]。

- getResource 方法

返回映射到某个资源上的 URL 对象，传递给 getResource 方法的参数必须是一个以"/"开头的资源路径名，开头的"/"表示当前 Web 应用程序的根目录。

- getResourceAsStream 方法

返回连接到某个资源上的 InputStream 对象，这个方法实际上是打开了 getResource 方法返回的 URL 对象的输入流，并返回这个输入流对象供程序直接去读取数据，它的参数传递规则与 getResource 方法完全一样。

一个重要的编程习惯和经验是将程序中要用到的某些参数值设置在一个文件中，而不是以硬编码的形式写在程序代码中，以便不用修改源程序就可以对参数进行调整，例如，TCP 服务器程序监听的端口号，连接数据库时的数据库名称、用户名和密码等信息。虽然在 Servlet 程序中也可以使用 FileInputStream 类来直接访问文件，但是由于下面两个原因，导致这种访问方式无法很好地应用于实际工作中。

（1）一个 Web 应用程序在本地文件系统中的安装位置是可以变化的，不同的公司或个人可能将它安装在不同的位置上，所以 Servlet 程序不应使用绝对路径的形式来访问 Web 应用程序中的某个文件。

（2）在某个 java 类中使用的相对路径是相对于当前的工作目录而言的，这个工作目录通常是执行 java 命令的目录，而不是当前正在执行的 java 类所在的目录，这一特性导致在 Java 程序中很难直接使用相对目录。

针对在 Java 程序中使用 FileInputStream 类和相对路径访问资源文件时会出现的问题，JDK 中的 ClassLoader 类专门提供了 getResource 等方法去装载资源文件，它们使用与查找 Java 类文件同样的方式去查找资源文件，即在类装载器所搜索的目录中查找。为了简化程序的编写，Class 类中也定义了几个访问资源文件的方法，这些方法内部调用了 ClassLoader 类中的同名方法去完成相应的功能。为了防止外部使用浏览器访问到资源文件，Web 应用程序中的资源文件通常应放置在 Web-INF 目录或其子目录中。由于 Web 应用程序的类装载器会搜索 Web-INF/classes 目录，所以，ClassLoader.getResourceAsStream 方法也可以访问该目录中的资源文件，但是，该方法不能访问 Web 应用程序内的其他目录中的资源。ServletContext 类中的访问资源的方法是通过 Servlet 容器来获得资源文件的，它使得 Servlet 程序可以访问 Web 应用程序内部的任意位置的文件。

在 Servlet 程序中使用 FileInputStream 类的问题

01 在 d:\myWeb\Web-INF\classes 目录中创建一个名为 it315.properties 的文件，文件中的内容格式为 java.util.Properties 类的存储格式，如例程 4-15 所示。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

例程 4-15 it315.properties

```
database=it315
username=zxx
password=123
```

02 按例程 4-16 编写一个名为 FileAccessServlet 的 Servlet 程序，它使用 FileInputStream 类和相对路径的形式访问 it315.properties 文件，并读取其中的各个属性设置值。

例程 4-16 FileAccessServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Properties;

public class FileAccessServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
    PrintWriter out = response.getWriter();
        FileInputStream fis = new FileInputStream("it315.properties");
        Properties props = new Properties();
        props.load(fis);
        out.println("database=" + props.getProperty("database"));
        out.println("username=" + props.getProperty("username"));
        out.println("password=" + props.getProperty("password"));
        fis.close();
    }
}
```

编译 FileAccessServlet.java 文件，确保编译后生成的 class 文件放置进了 d:\myWeb\Web-INF\classes 目录中后，在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/servlet/FileAccessServlet
```

尽管 FileAccessServlet.class 与 it315.properties 位于同一目录中，但浏览器中返回的显示内容为系统找不到 it315.properties 文件的异常信息。

（3）如果 Tomcat 是通过 startup.bat 文件启动的，那么，FileAccessServlet.java 文件中使用的相对路径就是相对于 startup.bat 所在的目录而言的，将 it315.properties 移动到该目录中，刷新浏览器刚才的访问，这时就可以看到正常的访问结果了。注意，如果

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

使用如图 2.6 所示的 Windows 程序菜单组中的“Apache Tomcat 4.1”组里面的“Start Tomcat”快捷菜单项来启动 Tomcat，在图 2.16 中可以看到，这个快捷命令所设置的起始位置为"C:\Documents and Settings\Administrator\「开始」菜单\程序\Apache Tomcat 4.1"，在这种情况下必须将 it315.properties 移动该起始位置目录中，才可以看到正常的访问结果。

（4）在第（3）步实验成功的基础上，注释掉最后的 fis.close()语句，重新编译和访问更新的 ResourceServlet 后，尝试删除 it315.properties 文件，可以看到该文件不能被删除。这就是因为在 ResourceServlet 中创建的连接到该文件的底层输入流资源没有被释放的缘故，由此可见使用 close 方法关闭底层 IO 流的作用和重要性。

使用 ServletContext 接口和 ClassLoader 类中的方法访问资源文件

01 按例程 4-17 编写一个名为 ResourceServlet 的 Servlet 程序，它分别使用 ServletContext 类或 ClassLoader 类的方法访问在上面的实验中编写的 it315.properties 文件，并读取其中的各个属性设置值。

例程 4-17 ResourceServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Properties;
import java.net.URL;

public class ResourceServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
     PrintWriter out = response.getWriter();

        URL urlIt315 = getServletContext().getResource(
                        "/Web-INF/classes/it315.properties");
        out.println(urlIt315.toString());

        //下面两条语句可以完成同样的功能
        InputStream ips = getServletContext().getResourceAsStream(
                            "/Web-INF/classes/it315.properties");
        /*InputStream ips = getClass().getResourceAsStream(
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
                                "/it315.properties");*/
        Properties props = new Properties();
        props.load(ips);
        out.println("database=" + props.getProperty("database"));
        out.println("username=" + props.getProperty("username"));
        out.println("password=" + props.getProperty("password"));
        ips.close();
    }
}
```

编译 ResourceServlet.java 文件，确保编译后生成的 class 文件放置进了 d:\myWeb\Web-INF\classes 目录中，并且将 it315.properties 文件也移动到该目录中，然后在浏览器地址栏中输入如下地址：

http://localhost:8080/it315/servlet/ResourceServlet

浏览器中显示的结果如图 4.25 所示。

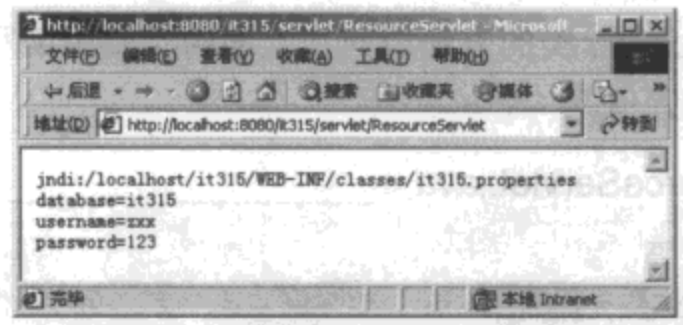


图 4.25

02 修改 ResourceServlet.java 文件，注释掉 ServletContext.getResourceAsStream 调用语句，并取消 Class.getResourceAsStream 调用语句的注释，重新编译 ResourceServlet.java 文件。等待 Tomcat 所运行的命令行窗口中显示出重新加载了 ResourceServlet 类的提示信息后，刷新浏览器刚才的访问请求，浏览器中显示的结果与刚才的访问结果完全一样。尽管使用 ClassLoader.getResourceAsStream 方法也可以访问资源文件，但是它只能加载位于 ClassLoader 搜索目录中的资源文件，如果将 it315.properties 文件放置在 Web-INF 目录或 Web-INF 的其他子目录中，那么就只能使用 ServletContext.getResourceAsStream 方法访问资源文件了。

4.6.5 获取虚拟路径所映射的本地路径

ServletContext 接口中定义了一个 getRealPath 方法，用于返回某个虚拟路径所映射的本地文件系统路径。如果传递给 getRealPath 方法的路径字符串以“/”作为起始字符，这个“/”代表的是当前 Web 应用程序的根目录。要返回 Web 应用程序的根目录所映射的本

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

地文件系统路径，传递给 getRealPath 方法的路径字符串中可以只有一个“/” 字符，也可以是一个空字符串。

4.6.6 Web 应用程序之间的访问

由于传递给 ServletContext.getRealPath 方法的路径字符串必须以“/”作为起始字符，并且这个“/”代表的是当前 Web 应用程序的根目录，这就说明 ServletContext. getRealPath 方法只能获得某个 Web 应用程序范围内的虚拟路径所映射的本地实际路径。如果要获得其他 Web 应用程序范围内的虚拟路径所映射的本地实际路径，该怎么办呢？

要实现这个功能，首先都要获得代表其他 Web 应用程序的 ServletContext 对象，然后再调用这个 ServletContext 对象的 getRealPath 方法。只要获得了代表其他 Web 应用程序的 ServletContext 对象，就可以实现诸如访问其他 Web 应用程序内的全局变量和跳转到其他 Web 应用程序的资源等功能。

在某个 Web 应用程序中能否访问代表其他 Web 应用程序的 ServletContext 对象呢？查看 Servlet API 帮助文档，可以看到 ServletContext 接口中定义了一个 getContext 方法来获得某个 URL 所对应的 ServletContext 对象，传递给 getContext 方法的路径字符串必须以“/”作为起始字符，这个“/”代表的是整个 Web 服务器的根目录，而不是某个 Web 应用程序的根目录，这说明在一个 Web 应用程序中可以获得代表其他 Web 应用程序的 ServletContext 对象。如果要让某个 Web 应用程序内部能够使用 ServletContext.getContext 方法返回代表其他 Web 应用程序的 ServletContext 对象，必须将<Tomcat 安装主目录>/conf/server.xml 文件中与该 Web 应用程序对应的<Context>元素的 crossContext 属性设置为 true。只有将<Context>元素的 crossContext 属性设置为 true 时，它所对应的 Web 应用程序内部才能使用 ServletContext.getContext 方法返回代表其他 Web 应用程序的 ServletContext 对象。例如，如果想让/it315 这个 Web 应用程序能够访问其他的 Web 应用程序，应将 server.xml 文件中的与/it315 这个 Web 应用程序对应的<Context>元素的起始标签修改成如下形式：

```
<Context path="/it315" docBase="d:\myWeb" debug="0" reloadable="true"
          crossContext="true">
```

crossContext 属性的默认设置值为 false，此时，ServletContext.getContext 方法的返回值为 null。

访问其他 Web 应用程序

01 按例程 4-18 编写一个名为 OtherContextServlet 的 Servlet 程序，它显示出整个 Web 站点的根目录所映射的本地文件系统路径。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

例程 4-18 OtherContextServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class OtherContextServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        PrintWriter out = response.getWriter();
        out.println(getServletContext().getContext("/").getRealPath(""));
    }
}
```

编译 OtherContextServlet.java 文件，确保生成的 class 文件位于 d:\myWeb\Web- INF\classes 目录中。

02 在浏览器地址栏中输入如下地址：

http://localhost:8080/it315/servlet/OtherContextServlet

浏览器中显示的结果如图 4.26 所示。

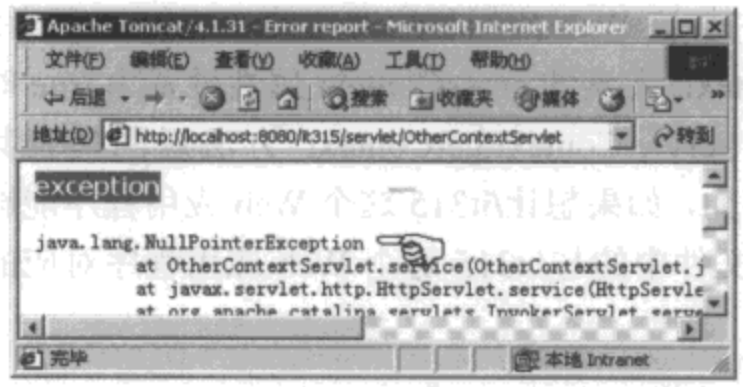


图 4.26

许多初学者在学习 Servlet 和 JSP 的过程中，都会经常遇到这个 NullPointerException 异常错误，因为他们通常只考虑一个方法调用的预期结果，总是想当然地认为方法调用的结果就是所期望的结果，而忽略方法也可能返回非预期结果的情况。虽然 Servlet Context.getContext 方法的作用和目的就是返回与某个 URL 对应的 ServletContext 对象，但它也可能返回 null 值。

上面的异常是因为 getContext 方法返回的结果正好是 null，然而程序没有对此进行判断，把该结果想当然地看做是一个 ServletContext 对象，接着就调用这个“对象”的 getRealPath 方法而造成的。一条重要的软件编码规范：对于可以有多个返回结果的方法调用，应该先对返回值的几种情况进行判断和检查后再继续使用，切勿用“就是”、“应该是”

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

的思想来对待每行程序代码的编写。

03 按照上面的规范编写重新编写 OtherContextServlet，如例程 4-19 所示。

例程 4-19 OtherContextServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class OtherContextServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        PrintWriter out = response.getWriter();
        ServletContext rootContext = getServletContext().getContext("/");
        if(rootContext != null)
        {
            out.println(rootContext.getRealPath(""));
        }
        else
        {
            out.println("getContext failed!");
        }
    }
}
```

可见，修改后的程序代码与修改前的程序代码的最大区别就是增加了 if...else 语句，显得比较臃肿，读者要花更多的时间去阅读和理解程序，但是在实际的软件项目开发中，这种“臃肿”的代码是必需的。笔者在本书中编写的许多例子程序中，并没有严格地去遵循这个规范，主要是为了节省篇幅和简化程序的复杂性，不让这些内容干扰读者的学习视线，以便读者把注意力集中在所讲的核心问题上。

重新编译 OtherContextServlet.java 文件，等待 Tomcat 所运行的命令行窗口中显示出重新加载了 OtherContextServlet 类的提示信息后，刷新浏览器刚才的访问请求。可以看到浏览器中显示出了“getContext failed!”信息。

04 将 server.xml 文件中的与/it315 这个 Web 应用程序对应的<Context>元素的起始标签修改成如下形式：

```
<Context path="/it315" docBase="d:\myWeb" debug="0" reloadable="true"
        crossContext="true">
```

重新启动 Tomcat，刷新浏览器刚才的访问请求，可以看到浏览器窗口中显示了整个

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

Web 站点的根目录所映射的本地文件系统路径。

4.6.7 ServletContext 接口中的其他方法

ServletContext 中还定义了一些不常用的方法，读者浏览一下就可以了。

- getMajorVersion 方法

返回 Servlet 容器所支持的 Servlet API 的主要版本号，例如，对于 Servlet API 2.3，它返回整数 2。

- getMinorVersion 方法

返回 Servlet 容器所支持的 Servlet API 的次版本号，例如，对于 Servlet API 2.3，它返回整数 3。

- getMimeType 方法

返回某个文件名所对应的 MIME 类型。一个文件名的 MIME 类型是在 Web 应用程序的部署描述符中定义的，这已经在 3.6.8 节中进行过详细讲解。

- getServerInfo 方法

返回 Servlet 容器的名称和版本信息，也就是 Tomcat 安装后建立的默认首页的左顶部显示的文本信息。

4.7 思考与实践

1. Servlet 程序的编译和运行过程的环境是完全不同的，如果 Servlet 程序中用到了第三方的 Jar 包，例如数据库驱动程序，在编译时需要设置 CLASSPATH 环境变量来包含该 Jar 包，在运行时也还需要用到该 Jar 包。请编写一个 Servlet 程序，该 Servlet 程序调用另外一个自己编写的普通类，将该普通类打包成一个 jar 文件后删除普通类的 class 文件，然后编译 Servlet 程序，该怎样设置才能让 Servlet 程序的编译通过？将 Servlet 放置在 Web 应用程序内的 Web-INF/classes 目录中运行，要想 Servlet 能够访问到该普通类，包含普通类的 jar 文件可以放置在下面的哪些目录中？

A. <JAVA_HOME>/jre/lib/ext

B. <CATALINA_HOME>/common/lib

C. <CATALINA_HOME>/server/lib

D. <CATALINA_HOME>/share/lib

E. <当前 Web 应用程序的目录>/Web-INF/lib

F. 将 jar 文件路径增加到 Tomcat 设置的 CLASSPATH 环境变量中

如果按照“自己的东西放在自己的抽屉里”的原则，你觉得最好的方式是哪种？请将答案和实验用的程序代码提交给 www.it315.org 站点，认真完成作业的读者可获得作者的标准答案。

2. 假设在两个 Web 应用程序中都要调用某一个 Servlet，如果将这个 Servlet 类文件放

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

置在<CATALINA_HOME>/share/classes 目录中，然后在这两个 Web 应用程序中分别访问该 Servlet 时，Servlet 的构造方法或 init 方法将被调用几次？

如果在每个 Web 应用程序的 Web-INF/classes 目录中都放置这个 Servlet 类文件，然后在这两个 Web 应用程序中分别访问该 Servlet 时，Servlet 的构造方法或 init 方法将被调用几次？

另外，请在这个 Servlet 程序中增加一条静态代码，用这条静态代码打印出一条提示信息，看看在上面的两种运行情况下，这条静态代码被执行了几次。请分析原因并写出详细的实验步骤和结果，然后提交给 www.it315.org 站点，认真完成作业的读者可获得作者的标准答案。

提示

上面的两种情况分别由 share 类装载器和两个 Web 应用程序各自的 Webapp 类装载器加载该 Servlet，可以在该 Servlet 的构造方法或 init 方法中将一个初始值为 0 的静态成员变量加 1 后打印出来，根据打印结果验证自己的想法。

3. 如果在 4.4.5 节的 ConfigTestServlet 例子程序中，覆盖 GenericServlet 的有参数的 init(ServletConfig config)方法，在其方法体中什么也不写，即在 ConfigTestServlet.java 增加如下一个方法：

```
public void init(ServletConfig config) throws ServletException
{
}
```

请问 ConfigTestServlet 将是怎样的访问运行效果？如果将这个方法定义修改为覆盖 GenericServlet 的无参数的 init()方法，即修改成如下形式：

```
public void init() throws ServletException
{
}
```

这时，ConfigTestServlet 又将是怎样的访问运行效果？请通过实验来验证你的想法，参加 it315 函授学习的读者对实验结果进行抓图后发送给 www.it315.org 站点检查。

4. 在 4.5.7 节中有这样一段描述："因为 HEAD 请求方式除了不要求 Web 服务器返回实体正文部分外，它的操作与 GET 方式完全一样，所以只要覆盖了 doGet 方法，即使没有覆盖 doHead 方法，Servlet 也可以支持 HEAD 请求方式。"请针对 4.5.7 节中编写和部署的例子程序，使用 Telnet 命令做一个 HEAD 请求方式的访问实验，对实验过程抓图后发送给 www.it315.org 站点。

5. 请参看 java.util.logging.FileHandler 类的帮助文档和<JRE_HOME>/lib/log. properties 文件的设置，做一个将日志信息同时记录到控制台和文件的实验，要求写出详细的实验步骤和结果，并对 FileHandler 类的每个设置信息进行详细解释。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

第5章 HttpServletResponse的应用

5.1 准备实验环境

由于 Servlet 2.4 和 JSP 2.0 规范与 Servlet 2.3 和 JSP 1.2 规范相比有较大的变动，从本章开始将按照 Servlet 2.4 和 JSP 2.0 规范进行讲解，其中的一些实验在 Tomcat 4.1.31 中将无法正常运行，因此，从本章开始的实验不再使用 Tomcat 4.1.31，而是全部使用 Tomcat 5.5.12 作为运行环境。

准备实验环境的具体过程如下所述。

01 在 Tomcat 5.5.12 的<tomcat 的安装目录>\Webapps 目录中创建一个名为 it315 的子目录，并在 it315 目录中创建一个名为 test.html 的网页文件，在该文件中写上“<s>中文测试页面</s>”这几个字符。启动 Tomcat，在本地计算机的浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/test.html
```

浏览器提示找不到该网页文档。

02 在<tomcat 的安装目录>\Webapps\it315 目录中创建一个名为 Web-INF 的子目录，稍后在本地计算机的浏览器地址栏中再次输入上面的访问地址或刷新含有该地址的浏览器窗口，就可以看到 test.html 网页文档中的内容了。这说明只要<tomcat 的安装目录>\Webapps 目录下的某个子目录中包含 Web-INF 子目录，这个子目录就会被部署为一个独立的 Web 应用程序。进入启动 Tomcat 的命令行窗口中，也可以看到其中打印出了一条类似如下的信息：

```
Missing application Web.xml, using defaults only StandardEngine[Catalina].
StandardHost[localhost].StandardContext[/it315]
```

可见，在 Tomcat 5.x 运行时，将一个 Web 应用程序放置到<tomcat 的安装目录>\Webapps 目录下，容器可以自动检测和部署这个应用程序。Tomcat 4.x 容器不能自动检测和部署在运行时放置到<tomcat 的安装目录>\Webapps 目录下的应用程序，除非重新启动 Tocmat 服务程序。

03 为了便于对 Servlet 源文件进行编译，在保存 Servlet 源文件的目录中编写一个 compile.bat 批处理文件，例如，本章的所有例子程序都是在 d:\javaWeb\lesson5 目录中编写的，那么就是在 d:\javaWeb\lesson5 目录中创建一个名为 compile.bat 的批处理文件。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

compile.bat 文件中的内容如下：

```
set PATH=C:\jdk1.5.0_01\bin;%path%
set CLASSPATH=C:\tomcat-5.5.12\common\lib\servlet-api.jar;%CLASSPATH%
javac -d C:\tomcat-5.5.12\Webapps\it315\Web-INF\classes %1
pause
```

以后只要在 Windows 资源管理器窗口中将 Servlet 源文件拖动到这个批处理文件上，就可以完成 Servlet 源程序的编译了。

04 为了让/it315 这个 Web 应用程序能自动重新装载发生了修改的 Servlet 程序，需要修改 Tomcat 的 server.xml 文件，在该文件的<Host>元素中增加如下一个<Context>子元素：

```
<Context path="/it315" docBase="it315" reloadable="true"/>
```

保存 server.xml 文件后，重新启动 Tomcat。

5.2 HttpServletResponse 简介

Web 服务器回送给 Web 客户端的 HTTP 响应消息分为三个部分：状态行、响应消息头、消息正文（也叫实体内容）。Servlet API 中定义的 ServletResponse 接口类用于创建响应消息，Servlet 程序通过调用 ServletResponse 对象的方法可以向客户端回送基本的响应消息。HttpServletResponse 是专用于 HTTP 协议的 ServletResponse 子接口，它用于封装 HTTP 响应消息，允许操控 HTTP 协议相关数据，包括响应头和状态码，支持 Cookies 和 session 跟踪。HttpServletResponse 也定义了一系列用于描述各种 HTTP 状态码的常量。

对于一个 Servlet 的每次访问请求，Servlet 引擎都会创建一个用于封装 HTTP 请求消息的对象和一个代表 HTTP 响应消息的对象，然后调用此 Servlet 对象的 service()方法，并将这两个对象作为参数传递给该方法。其中，作为参数传递进来的响应对象就是 HttpServletResponse 对象，在 service()方法内部可调用它的各种方法来创建响应消息。

5.3 产生响应状态行

HTTP 响应消息的响应状态行可分为三个部分：HTTP 版本、状态代码和一条相关的提示信息，如下所示：

```
HTTP/1.1 200 OK
```

HttpServletResponse 中定义了几个用于产生 HTTP 消息的响应状态行的方法和对应各个响应状态码的常量。

5.3.1 状态码常量

计算机里只有数值，计算机里的一切都是用数值来表示的，数值可以代表各种意义。但是，如果在程序代码中直接使用数值，显然不如使用英文单词那么直观和容易记忆，例

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

如，现在告诉你："用 507 代表鼠标移动，用 826 代表键盘按下，用 631 代表鼠标按下，用 327 代表鼠标弹起"，过了一个星期甚至是一个小时后，你还能记住刚才所说的各个数字所代表的意义吗？当你在写程序时想对鼠标按下这个事件进行处理，你应该对哪个数值进行检测呢？当你在阅读程序时看到一个数值 631，难道你一下子就能知道它代表哪种事件吗？为此，在各种编程语言中都提供了定义和使用常量的方式，即用一个英文单词定义为一个数值，在要使用到该数值的地方，用已定义的相应英文单词来替换，这些英文单词就被称之为常量。

在一些情况下使用常量来表示数值，可以使得程序代码易读、易写。HttpServletResponse 中定义了若干与状态码数值对应的常量，每个常量的名称以前缀 SC（Status Code 的简写）开头，然后是状态码在 HTTP 1.1 规范中所表示的状态信息的英文单词的组合，每个单词之间用下划线连接，且所有字母都大写。例如，状态码 404 在 HTTP 1.1 规范中的标准状态信息为 Not Found，在 HttpServletResponse 中为它所定义的常量是 SC_NOT_FOUND。状态码 302 在 HTTP 1.0 中的提示信息为 Moved Temporarily，在 HTTP 1.1 中的提示信息为 Found，所以，在早期的 Servlet API 规范中，状态码 302 所对应的常量名称是 SC_MOVED_TEMPORARILY，在新的 Servlet API 2.4 规范中，HttpServletResponse 中增加了一个按 HTTP 1.1 的提示信息产生的 SC_FOUND 常量来对应状态码 302。状态码 307（Temporary Redirect）所对应的常量在早期的 Servlet API 规范中被 HttpServletResponse 的当初设计开发者遗忘了，在新的 Servlet API 2.4 规范中，HttpServletResponse 增加了 SC_TEMPORARY_REDIRECT 常量来对应状态码 307。通过比较 Tomcat 4.x 和 Tomcat 5.x 中各自附带的 Servlet API 帮助文档，可以查看到不同版本下的 Servlet API 的这些细微区别。

5.3.2 setStatus 方法

setStatus 方法用于设置 HTTP 响应消息的状态码，并生成响应状态行。由于响应状态行中的提示信息直接与状态代码相关，且 HTTP 版本由服务器确定，因此在使用 setStatus 方法产生 HTTP 消息的响应状态行时，只需设置响应状态代码就足够了。setStatus 方法接受一个 int 类型的参数，这个参数就是要设置的响应状态码。正常情况下的响应码为 200，Web 服务器默认自动产生这种正常情况下的响应状态行，所以，通常情况下的 Servlet 不需要调用 HttpServletResponse 的 setStatus 方法来指定状态代码和产生响应状态行。只有在 HTTP 响应消息中使用一个特殊的状态码，才需要调用 setStatus 方法。

由于 HTTP 响应消息是由状态行、一个或多个响应头、实体内容顺序组成的，所以，如果 Servlet 没有使用或不支持输出缓冲区，那么就一定要在调用其他也可产生客户端输出内容的方法前调用 setStatus 方法。即使 Servlet 使用了输出缓冲区，也必须在输出缓冲区的内容尚未被实际发送到客户端之前调用 setStatus 方法。

5.3.3 sendError 方法

sendError 方法用于发送表示错误信息的状态码（一般是 404，找不到客户机所请求的

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

资源）到客户端，并清除缓冲区中的内容。HttpServletResponse 中提供了两个重载的 sendError 方法，定义如下：

- public void sendError(int code) throws java.io.IOException
- public void sendError(int code, String message) throws java.io.IOException

第一个 sendError 方法只是发送错误信息的状态码，而第二个 sendError 方法除了发送状态码外，还可以增加一条用于提示说明的文本信息，该文本信息将出现在发送给客户端的正文内容中。例如，虽然大多数服务器会自动为 404 响应码生成“File Not Found”的文本提示信息，但调用第二个 sendError 方法可以用更合适的文本内容作为 404 响应码的提示信息。

5.4 构建响应消息头

ServletResponse 和 HttpServletResponse 接口中定义了若干用于产生 HTTP 响应消息中的头字段的方法，其中，addHeader 和 setHeader 方法是两个通用的方法，可用于设置所有的头字段。除了 addHeader 和 setHeader 方法外，HttpServletResponse 还定义了一些其他便利的方法，例如，setContentType、setContentLength、setIntHeader、setDateHeader 等方法，用来设置那些常用或有日期、整数数据的响应头。

对于状态码为 200 的正常响应消息，通过向客户端发送的各种 HTTP 响应消息头字段，可以让浏览器执行各种特殊功能，例如，让浏览器自动定时刷新网页，告诉浏览器采用何种字符集显示网页，禁止浏览器缓存页面等。对于一些特殊的响应状态码，通常也要设置一些相关联的 HTTP 响应消息头，例如，300～399 之间的所有状态代码都应伴随一个 Location 头，而状态代码 401（Unauthorized）总是应该伴随一个 WWW-Authenticate 头。要利用 HTTP 响应头字段来实现一些特殊功能，必须了解响应头字段有哪些作用及具体用法。

5.4.1 addHeader 与 setHeader 方法

addHeader 和 setHeader 方法都可用于设置 HTTP 响应消息的各种头字段，它们的语法定义如下：

- public void addHeader(java.lang.String name, java.lang.String value)
- public void setHeader(java.lang.String name, java.lang.String value)

其中的第一个参数为响应头的字段名，第二个参数为字段取值。这两个方法都是增加一个新的头字段。其中，如果已经设置过同名的响应头，setHeader 方法将用新的设置值取代原来的设置值，而 addHeader 方法则是增加一个同名的响应头。HTTP 响应消息中允许同一名称的头字段出现多次，addHeader 方法正好可用于在消息头中生成要出现多次的头字段。这两个方法的使用方法很简单，但要想充分有效地利用好这两个方法，还必须很好地掌握各个 HTTP 响应头的作用。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

5.4.2 addIntHeader 与 setIntHeader 方法

HttpServletResponse 提供了两个专门用于设置包含整数值的响应头的方法，它们的定义语法为：

- `public void addIntHeader(java.lang.String name,int value)`
- `public void setIntHeader(java.lang.String name,int value)`

这两个方法避免了一个小小的麻烦，即 addHeader 与 setHeader 方法将 int 类型的值设置给一个响应头之前必须转换为 String 类型。

5.4.3 addDateHeader 与 setDateHeader 方法

HttpServletResponse 提供了两个专门用于设置包含日期值的响应头的方法，它们的定义语法为：

- `public void addDateHeader(java.lang.String name, long date)`
- `public void setDateHeader(java.lang.String name, long date)`

HTTP 响应头中的日期通常都为 GMT 时间格式，这两个方法省掉了把自 1970 年 1 月 1 日的 0 点 0 分 0 秒开始计算的一个以毫秒为单位的长整数值(可以由 System.currentTimeMillis、Date.getTime 或 Calendar.getTimeInMillis 等方法返回）转换为 GMT 时间串的麻烦。

5.4.4 setContentLength 方法

setContentLength 方法用于设置响应消息的实体内容的大小，单位为字节。对于 HTTP 协议来说，这个方法就是设置 Content-Length 响应头字段的值。因为浏览器与 Web 服务器之间使用持久（keep-alive）的 HTTP 连接，如果 Web 服务器没有采用 chunked 传输编码方式，那么它必须在每个应答中发送一个 Content-Length 的响应头来表示各个实体内容的长度，以便客户端能够分辨出上一个响应内容的结束位置。一般来说，Servlet 程序不必调用 setContentLength 方法来设置 Content-Length 头字段，因为 Servlet 引擎在向客户端实际输出响应内容时，它可以自动设置 Content-Length 头字段或采用 chunked 传输编码方式。

5.4.5 setContentType 方法

setContentType 方法用于设置 Servlet 输出内容的 MIME 类型，对于 HTTP 协议来说，就是设置 Content-Type 响应头字段的值。例如，如果发送到客户端的内容是 jpeg 格式的图像数据，就需要将响应内容的类型设置为“image/jpeg”；如果发送到客户端的内容是 XML 格式的文本数据，就需要将响应内容的类型设置为“text/xml”。

Servlet 引擎默认设置的输出内容类型为“text/plain”，即普通的文本类型，而网页文档的类型通常为“text/html”，所以在 Servlet 程序中需要显式地调用 setContentType 方法来对此进行指定。在 MIME 类型后面还可以指定响应内容所使用的字符集类型，例如，

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

"text/html; charset=GB2312"，如果在 MIME 类型后面没有指定字符集编码类型，且使用 ServletResponse.getWriter 方法返回的 PrintWriter 对象输出文本内容时，Tomcat 5.x 将在 MIME 类型后默认增加 ISO8859-1 的字符集编码类型。

PrintWriter 对象在输出字符文本时，需要先将它们转换成某种字符集编码的字节数组形式后，再把这些字节内容写入到 Servlet 引擎提供的缓冲区中。setContentType 方法除了具有设置 Content-Type 响应头字段（即指定输出内容的 MIME 类型）的功能之外，它还具有指定 ServletResponse.getWriter 方法返回的 PrintWriter 对象输出文本时所采用的字符集编码的作用。

5.4.6 setCharacterEncoding 方法

setCharacterEncoding 是在 Servlet API 2.4 规范中新增的方法，在以前版本的 Servlet 规范中没有这个方法。setCharacterEncoding 方法用于设置输出内容的 MIME 声明中的字符集编码，对 HTTP 协议来说，就是设置 Content-Type 头字段中的字符集编码部分。如果没有设置 Content-Type 头字段，setCharacterEncoding 方法设置的字符集编码不会出现在 HTTP 消息的响应头中，但是它的设置结果仍然决定了 ServletResponse.getWriter 方法返回的 PrintWriter 对象输出文本内容时所采用的字符集编码。setCharacterEncoding 方法比 setContentType 和 setLocale 方法的优先权高，它的设置结果将覆盖 setContentType 和 setLocale 方法所设置的字符集编码。

5.4.7 setLocale 方法

setLocale 方法用于设置响应消息的本地化信息，对 HTTP 来说，它将设置 Content-Language 响应头字段和 Content-Type 头字段中的字符集编码部分。如果没有设置 Content-Type 头字段，setLocale 方法设置的字符集编码不会出现在 HTTP 消息的响应头中，但是它的设置结果仍然决定了 ServletResponse.getWriter 方法返回的 PrintWriter 对象输出文本内容时所采用的字符集编码。如果调用 setCharacterEncoding 或 setContentType 方法指定了响应内容的字符集编码，setLocale 方法将不再具有指定字符集编码的功能。setLocale 方法的语法定义如下：

```
public void setLocale(java.util.Locale loc)
```

该方法接受一个类型为 java.util.Locale 的参数。Locale 对象中只包含了语言和国家地区的信息，它并没有包含字符集编码的信息，所以，Locale 信息与字符集编码之间需要有一种映射关系。那么，这种映射关系是如何建立的呢？在部署描述符文件（Web.xml）中可以通过<locale-encoding-mapping-list>元素来设置各种 Locale 与字符集编码之间的映射关系，如下所示：

```
<locale-encoding-mapping-list>
    <locale-encoding-mapping>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        <locale>zh_CN</locale>
        <encoding>GB2312</encoding>
    </locale-encoding-mapping>
</locale-encoding-mapping-list>
```

对于<locale-encoding-mapping-list>元素中没有包含的 Locale 信息，则由各个厂商的 Servlet 引擎自己决定该 Locale 所映射的字符集编码。

5.4.8 containsHeader 方法

containsHeader 方法用于检查某个名称的头字段是否已经被设置。

5.5 响应消息头的实用案例

5.5.1 Servlet 的中文输出问题

在 Java Web 程序开发中，经常会遇到各种各样的中文乱码问题，其中以 Servlet 输出中文时所出现的乱码问题最常见，其原理和解决方式也最简单。关于字符编码的一些细节知识，请参看本书作者的《Java 就业培训教程》一书。

浏览器接收到的中文字符并不是中文符号本身，而是它的某种字符集编码数据。由于同一个中文字符可以使用多种字符编码，浏览器必须使用相应的字符集编码进行显示，才能将它所接收到的中文字符的某种编码数据显示为正确的中文文字。因此，Servlet 程序输出的中文字符在浏览器中显示为乱码的原因有两种可能：

- Servlet 程序输出给浏览器的内容不是任何一种正确的中文字符集编码；
- 浏览器网页文档时所采用的字符集编码与它所接收到的中文字符本身的字符集编码不一致。

当 Servlet 程序仅仅需要输出纯文本格式的响应正文时，通常应调用 ServletResponse 对象的 getWriter 方法返回一个 PrintWriter 对象，然后使用这个 PrintWriter 对象将文本内容写入到客户端。Java 程序中的字符文本在内存中是以 unicode 编码的形式存在的，PrintWriter 对象在输出字符文本时，需要先将它们转换成其他某种字符集编码的字节数组后，再把这些字节内容写入到 Servlet 引擎提供的缓冲区中。如果 PrintWriter 对象在转换过程中使用的字符集编码不正确，那么输出到客户端的数据就不是原始字符的正常字符集编码。例如，ServletResponse 对象的 getWriter 方法返回的 PrintWriter 对象默认使用 ISO8859-1 字符集编码进行 Unicode 字符串到字节数组的转换，由于 ISO8859-1 字符集中根本就没有包含中文字符，Unicode 编码的中文字符将被转换成无效的字符编码后输出给客户端。为此，ServletResponse 接口中定义了 setCharacterEncoding、setContentType 和 setLocale 等方法来指定 ServletResponse.getWriter 方法返回的 PrintWriter 对象所使用的字符集编码，只有在调用 ServletResponse.getWriter 方法返回 PrintWriter 对象之前调用这些方法，它们才会对 PrintWriter 对象起作用。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

浏览器显示网页文档时所采用的字符集编码可以通过手工方式设置，也可以在 HTTP 响应消息的 Content-Type 头字段中指定。作为网站开发人员，应尽量避免让用户通过手工方式来设置浏览器显示网页文档时所采用的字符集编码，这通过调用 ServletResponse 接口中定义的 setContentType 方法即可实现。

浏览器显示网页文档时怎样选择字符集编码

01 启动浏览器程序，查看浏览器的“查看”→“编码”→“自动选择”菜单是否处于选中（前面打钩）状态，是则单击该菜单去掉选中状态。单击“查看”→“编码”→“西欧(ISO)”菜单，设置浏览器采用“ISO8859-1”字符集编码来显示当前网页文档内容，如图 5.1 所示。

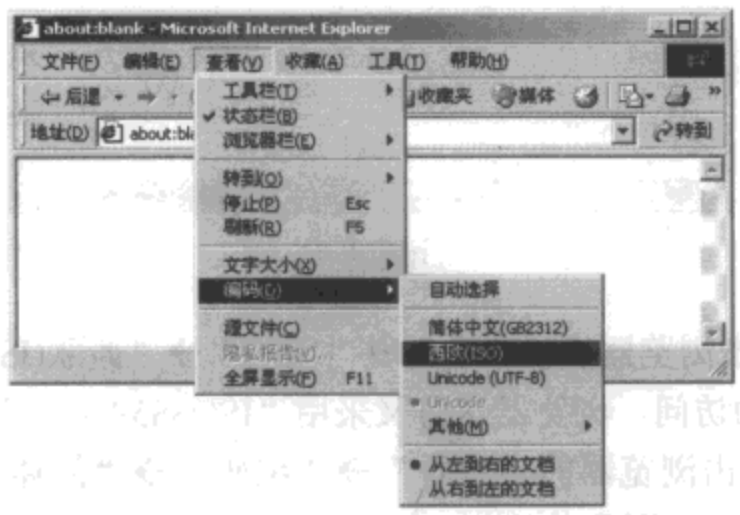


图 5.1

02 用浏览器访问在准备实验环境时所创建的 test.html 文件，即在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/test.html
```

浏览器中显示的结果如图 5.2 所示。

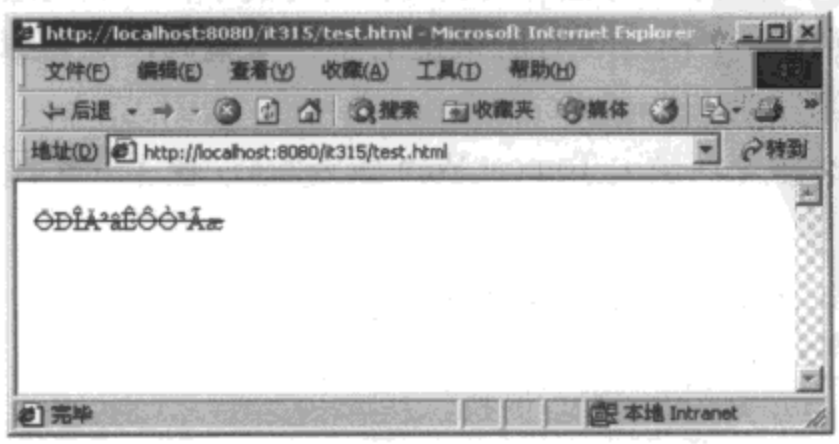


图 5.2

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

在图 5.2 所示的浏览器窗口中，单击“查看”→“源文件”菜单，可以看到其内容为“<s>中文测试页面</s>”。在浏览器显示的网页文档的源文件中看到了正常的中文字符，这说明服务器端程序已经输出了这些中文字符的正确的 GB2312 字符集编码。至于图 5.2 所示的浏览器中的中文字符显示为乱码，这是由于浏览器当前采用的是“ISO8859-1”字符集编码来显示当前网页文档内容，与网页文档中的中文字符的字符集编码不一致所造成的。

03 单击图 5.2 所示的浏览器的“查看”→“编码”→“简体中文(GB2312)”菜单，让浏览器采用“GB2312”字符集编码来显示当前网页文档内容，结果如图 5.3 所示，浏览器显示出了正常的中文字符。

图 5.3

在图 5.3 中再次单击浏览器的“查看”→“编码”→“西欧(ISO)”菜单，然后刷新浏览器对 test.html 页面的访问，浏览器这时又采用“ISO8859-1”字符集编码来显示返回的网页文档内容；再次单击浏览器的“查看”→“编码”→“简体中文(GB2312)”菜单，然后刷新浏览器对 test.html 页面的访问，浏览器这时则又采用“GB2312”字符集编码来显示返回的网页文档内容。这说明浏览器接收到 test.html 页面内容后，总是采用浏览器当前所设置的字符集编码来进行显示，当这与页面内容的实际字符集编码不一致时就会出现乱码。

显然，网站开发人员和网页设计人员不希望用户在浏览自己站点时看到乱码，更不能寄希望于用户通过手工方式来调整浏览器显示网页文档内容时所采用的字符集编码，网站开发人员和网页设计人员应该想办法将网页文档内容的字符集编码通知给浏览器，让浏览器能够自动调整显示网页文档内容时所采用的字符集编码，这可以通过 HTTP 消息的 Content-Type 头字段来完成。

04 使用 telnet 命令连接上 Tomcat Web 服务器后，在 Telnet 程序命令窗口中输入如下内容：

```
GET /it315/test.html HTTP/1.1<回车>
Host:<空格><回车>
<回车>
```

Telnet 窗口中显示的结果如图 5.4 所示。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

图 5.4

从图 5.4 中可以看出，尽管响应消息中设置了 Content-Type 头字段，但在其设置值中只指定了响应内容的 MIME 类型，而没有附加指定实体内容的字符集编码，所以，浏览器无法知道实体内容的实际字符集编码，它只能采用当前所设置的字符集编码来显示响应消息中的实体内容，而不能自动调整显示网页文档内容时所采用的字符集编码，这就有可能出现显示时的乱码现象。这里先只引出访问 test.html 页面时可能出现乱码的问题和原因，具体的解决办法在 5.5.4 节中讲解，读者应耐心学完下面的内容后才能更好地理解 5.5.4 节的内容。

Servlet 的中文输出问题及解决之道

01 按例程 5-1 编写一个名为 ChineseServlet 的 Servlet 程序。

例程 5-1 ChineseServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ChineseServlet extends HttpServlet
{
public void service(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException
{

    //response.setCharacterEncoding("GB2312");
    //response.setContentType("text/html;charset=GB2312");
    //response.setLocale(new java.util.Locale("zh","CN"));

    PrintWriter out = response.getWriter();
    out.println("<s>IT资讯交流网</s>");
}
}
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

在 Windows 的资源管理器窗口中，将 ChineseServlet.java 源文件拖动到前面编写的 compile.bat 批处理文件上进行编译，确保编译后生成的 class 文件存放进了<tomcat 的安装目录>\Webapps\it315\Web-INF\classes 目录中。

02 在<tomcat 的安装目录>\Webapps\it315\Web-INF 目录中创建一个 Web.xml 文件，内容如下：

```
<servlet>
    <servlet-name>ChineseServlet</servlet-name>
    <servlet-class>ChineseServlet</servlet-class>
</servlet>

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

保存 Web.xml 文件后，重新启动 Tomcat。

03 在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/servlet/ChineseServlet
```

浏览器中显示的结果如图 5.5 所示。

图 5.5

从图 5.5 的显示结果中可以看到如下两个问题：

- 浏览器直接显示出了 HTML 标签内容，例如，<s>和</s>没有被浏览器当做 HTML 标签来进行解释处理；
- Servlet 程序输出的中文字符全部显示为问号（?）。

浏览器在解释处理它所接收到的响应消息的实体内容时，是依据响应消息的 Content-Type 头中设置的 MIME 类型来决定如何进行处理的，只有 Content-Type 头中设置的 MIME 类型为“text/html”时，浏览器才会把它所接收到的实体内容当做 HTML 文档处理，才会去解释其中的 HTML 标签。所以，上面的第一个问题很可能是 Servlet 程序输出的响应消息中没有指定正确的 MIME 类型所造成的。使用 telnet 命令连接上 Tomcat Web 服务器后，在 Telnet 程序命令窗口中输入如下内容：

```
GET /it315/servlet/ChineseServlet HTTP/1.1<回车>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
Host:<空格><回车>
<回车>
```

Telnet 窗口中显示的结果如图 5.6 所示，响应消息中没有 Content-Type 头字段，所以，浏览器把接收到的实体内容当做了普通的文本类型（text/plain）来处理，即直接显示出其所接收到的原始内容，而不是把它们当做 HTML 内容进行解释和显示。

图 5.6

至于 Servlet 程序输出的中文字符全部显示为了问号（?），是因为 Servlet 程序输出到客户端的数据根本就不是这些中文字符的正常字符集编码，具体原因则是 Servlet 程序中的 PrintWriter 对象在输出中文字符时使用了不正确的字符集编码进行转换。

04 修改 ChineseServlet.java 源文件，将调用 setCharacterEncoding 方法或 setLocale 方法的语句前面的注释去掉，重新编译 ChineseServlet.java 源文件。等待 Tomcat 重新装载 ChineseServlet 后，使用 telnet 命令再次连接上 Tomcat Web 服务器后，在 Telnet 程序命令窗口中再次输入第 **03** 步中所输入的命令，Telnet 窗口中显示出的结果如图 5.7 所示。

图 5.7

这次在命令行窗口中显示出了正常的中文字符，这说明服务器端程序已经输出了中文字符的正确的 GB2312 字符集编码数据，但响应消息中仍然没有 Content-Type 头字段。如果命令行窗口中显示的中文字符还是图 5.6 所示的问号（?），那很可能是由于修改后的 ChineseServlet 没有被重新装载的缘故，这可以通过检查运行 Tomcat 5.x 的命令行窗口中是否出现了相关的提示信息来判断。关于如何让 Servlet 引擎自动重新加载修改过的 Servlet 程序，请参看 5.1 节中的第 **04** 步设置。

05 由于修改后的 ChineseServlet 输出的响应消息中没有包含 Content-Type 头字段，所以用浏览器访问 ChineseServlet 时，仍然会出现实体内容中 HTML 标签不被浏览器解释

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

处理和需要手工调整浏览器显示网页文档内容时所采用的字符集编码的问题。在图 5.5 所示的浏览器中，单击“查看”→“编码”→“西欧(ISO)”菜单，手工设置浏览器采用“ISO8859-1”字符集编码来显示当前网页文档内容，然后刷新对 ChineseServlet 的访问，浏览器返回的结果如图 5.8 所示，其中的中文字符仍然显示为乱码，但不再是问号（?）。

图 5.8

在图 5.8 所示的浏览器窗口中，单击“查看”→“源文件”菜单，可以看到其内容为“<s>IT 资讯交流网</s>”。图 5.8 所示的浏览器中的中文字符显示为乱码，是由于浏览器当前采用的是“ISO8859-1”字符集编码来显示当前网页文档内容，与实体内容中的中文字符本身的字符集编码不一致所造成的。单击图 5.8 所示的浏览器的“查看”→“编码”→“简体中文(GB2312)”菜单，让浏览器采用“GB2312”字符集编码来显示当前网页文档内容，结果如图 5.9 所示，浏览器显示出了正常的中文字符。

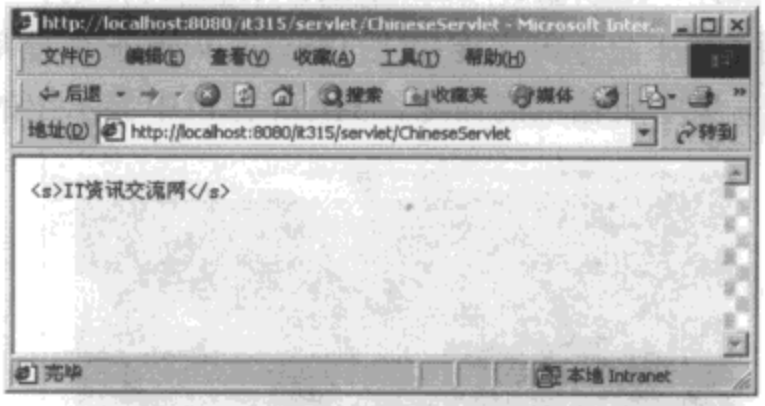


图 5.9

06 修改 ChineseServlet.java 源文件，重新注释掉调用 setCharacterEncoding 方法或调用 setLocale 方法的语句，取消调用 setContentType 方法的语句的注释，重新编译 ChineseServlet.java 源文件，等待 Tomcat 重新装载 ChineseServlet 后，在浏览器窗口中刷新对 ChineseServlet 的访问，返回结果如图 5.10 所示。

图 5.10

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

这时浏览器除了显示出正常的中文字符外，也按照 HTML 标签的语法对<s>和</s>进行了解释处理，而不是直接将它们当做普通文本显示出来。使用 telnet 命令再次连接上 Tomcat Web 服务器后，在 Telnet 程序命令窗口中再次输入第 03 步中所输入的命令，Telnet 窗口中显示出的结果如图 5.11 所示。

图 5.11

这次在响应消息中有了一个 Content-Type 头字段，它设置的 MIME 类型为“text/html”，且字符集编码为“GB2312”。正是由于这次的响应消息中有了这个 Content-Type 头字段，浏览器才把响应正文中的<s>…</s>当做 HTML 标签进行处理，才能显示出图 5.10 所示的效果。单击图 5.10 中的浏览器的“查看”→“编码”→“西欧(ISO)”菜单，然后刷新浏览器对 ChineseServlet 的访问，浏览器还是能正常显示出中文字符。单击浏览器的“查看”→“编码”菜单，可以看到经过这次访问后，浏览器当前的字符集编码状态被自动设置成了“简体中文(GB2312)”。这说明浏览器接收到使用 Content-Type 头字段指定了字符集编码的响应消息时，浏览器会自动采用 Content-Type 头字段中指定的字符集编码来显示网页文档内容，不需要用户手工设置浏览器采用何种字符集编码来显示网页文档内容。

如果传递给 setContentType 方法的参数中只指定了 MIME 类型为“text/html”，而没有指定字符集编码部分，当使用 ServletResponse.getWriter 方法返回的 PrintWriter 对象输出文本内容时，在 tomcat 5.x 环境中输出的 Content-Type 头字段的 MIME 后面会自动增加“iso8859-1”字符集编码，而在 tomcat 4.x 环境中输出的 Content-Type 头字段中则不含字符集编码部分，读者可以结合 telnet 命令做一个实验来验证。

脚下留心

注意传递给 setContentType 方法的参数值的大小写

在使用 setContentType 方法设置字符集编码时，传递给该方法的参数字符串中的“charset”的每个字符都必须小写，否则，设置的字符集编码将不起作用。但是，笔者也曾经遇到过在有的厂商提供的 Servlet 引擎中，需要将“charset”写成“charSet”，也就是只有其中的 s 大写，设置的字符集编码才能起作用，这个问题发生的原因是参照该厂商自带的例子程序后才找出来的。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

多学两招

修改某个 Servlet 将导致整个 Web 应用程序被重新加载

笔者无意中注意到每次修改并重新编译某个 Servlet 程序后，启动 Tomcat 5.x 的命令行窗口中显示的提示信息不是这个修改过的 Servlet 程序被重新装载，而是整个 Web 应用程序被重新装载，如下所示：

```
2005-10-2 20:40:02 org.apache.catalina.core.StandardContext reload
信息: Reloading this Context has started
```

对于这一新发现，可以通过什么方式来进行验证呢？读者应自己先想一想，然后再来参看笔者提供的解决方案。

笔者的方法是编写另外一个简单的 Servlet 程序，在这个 Servlet 程序中定义一条在命令行窗口中打印出一行提示信息的静态代码，首先分别访问一下这个简单的 Servlet 程序和上面的 ChineseServlet 程序，在命令行窗口中可以看到这个简单的 Servlet 程序打印出的提示信息。如果修改上面的 ChineseServlet 程序后，Tomcat 提示重新装载后，再次访问这个简单的 Servlet 程序，由于 Java 类中的静态代码在且只在字节码被装载时执行一次，如果在命令行窗口中又看到了这个简单的 Servlet 程序再次打印出提示信息，这就说明这个简单的 Servlet 程序也被重新装载了，从而可以推断出修改某一个 Servlet 程序都会导致整个 Web 应用程序被重新装载的结论，反之，这个结论不成立。另外，使用一个统计 Servlet 访问次数的成员变量也可以判断出 Servlet 是否被重新装载，如果在其他 Servlet 程序被修改并被重新装载后，这个简单 Servlet 的访问次数又回到了从 0 开始统计，这就说明这个简单 Servlet 程序也被重新装载了。合并了这两种思路的简单 Serlvet 程序代码如下：

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ReloadTestServlet extends HttpServlet
{
   private int count = 0;
   static {System.out.println("loading...");}
   public void service(HttpServletRequest request,
       HttpServletResponse response) throws ServletException, IOException
   {
       PrintWriter out = response.getWriter();
       out.println(++count);
   }
}
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

5.5.2 让浏览器定时刷新网页或跳转至其他页面

HTTP 协议中定义了一个 Refresh 头字段，用于告诉浏览器过多少秒后自动刷新页面。在 Refresh 头字段的时间设置值后面还可以用分号（;）分隔后，再指定一个 URL 地址，这将让浏览器在指定的时间值后自动去访问该 URL 地址指向的资源。

Refresh 头字段的作用效果

01 按例程 5-2 编写一个设置 Refresh 头字段的 Servlet。

例程 5-2 RefreshServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class RefreshServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
     //分别注释下面两条语句中的一条，查看各自的运行效果
        response.setHeader("Refresh","2");
        //response.setHeader("Refresh","2;URL=http://www.it315.org");

        response.getWriter().println(new java.util.Date());
    }
}
```

使用 compile.bat 编译 RefreshServlet.java 源文件，确保编译后生成的 class 文件存放在了<tomcat 的安装目录>\Webapps\it315\Web-INF\classes 目录中。

02 修改<tomcat 的安装目录>\Webapps\it315\Web-INF\Web.xml 文件，在其中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>RefreshServlet</servlet-name>
    <servlet-class>RefreshServlet</servlet-class>
</servlet>
……
……
<servlet-mapping>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


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

保存 Web.xml 文件后，重新启动 Tomcat。

03 在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/servlet/RefreshServlet
```

可以看到浏览器两秒钟后自动刷新网页文档的效果。

04 将 RefreshServlet.java 源文件中的第一条调用 setHeader 方法的语句注释掉，并取消第二条调用 setHeader 方法的语句的注释。重新编译 RefreshServlet.java 源文件并等待 Tomcat 重新装载 RefreshServlet 后，重复步骤 **03** 的操作，可以看到浏览器两秒后自动去访问 http://www.it315.org。

5.5.3 禁止浏览器缓存当前文档内容

为了提高网页的访问效率，浏览器可以在本地缓存以前访问过的页面，当浏览器再次访问那些已经缓存了的页面时，浏览器就不用再去访问远程的 Web 服务器了，而是直接从本地取出缓存的内容。缓存提高了浏览器的访问效率，但有时候也会带来负面效果，那就是服务器端的内容变化不能被实时地反映到客户端，例如，笔者接触过许多开发人员在实际项目中都遇到过动态产生的图片文件和 Javascript 脚本文件不能及时更新的问题，由于他们以前并不是非常清楚浏览器缓存的概念，所以，花费了不少时间和精力才解决他们遇到的问题。有三个 HTTP 响应头字段都可以禁止浏览器缓存当前页面，它们在 Servlet 程序中的示例代码分别如下：

- `response.setDateHeader("Expires",0);`
- `response.setHeader("Cache-Control","no-cache");`
- `response.setHeader("Pragma","no-cache");`

不是所有的浏览器都能完全支持上面的三个响应头，因此最好是同时使用上面的三个响应头，只要浏览器能支持其中任何一种形式，就能可靠地禁止浏览器缓存当前页面。

浏览器的缓存问题和解决方案

01 按例程 5-3 编写一个名为 CacheTestServlet 的 Servlet 程序，它向客户端输出一个随机值和一段用于产生 Form 表单的 HTML 文本。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

例程 5-3 CacheTestServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class CacheTestServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        response.setContentType("text/html;charset=GB2312");
        /*
        response.setDateHeader("Expires",0);
        response.setHeader("Cache-Control","no-cache");
        response.setHeader("Pragma","no-cache");
        */
        PrintWriter out = response.getWriter();
        out.println("本次响应的随机数为: " + Math.random());
        out.println("<form action='NotServlet'" + "method='POST'>" +
            "第一个参数: <input type='text' name='p1'><br>" +
            "第二个参数: <textarea name='p2'></textarea><br>" +
            "<input type='submit' value='提交'> " +
            "</form>");
    }
}
```

编译 CacheTestServlet.java 文件，确保编译后生成的 class 文件存放在了<tomcat 的安装目录>\Webapps\it315\Web-INF\classes 目录中。

02 在<tomcat 的安装目录>\Webapps\it315\Web-INF\Web.xml 文件中注册该 Servlet，并设置其映射 URL。在 Web.xml 文件中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>CacheTestServlet</servlet-name>
    <servlet-class>CacheTestServlet</servlet-class>
</servlet>
……
<servlet-mapping>
    <servlet-name>CacheTestServlet</servlet-name>
    <url-pattern>/servlet/CacheTestServlet</url-pattern>
</servlet-mapping>
```

保存 Web.xml 文件后，重新启动 Tomcat。

03 在浏览器地址栏中输入如下地址：

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://localhost:8080/it315/servlet/CacheTestServlet

填写浏览器返回页面中的表单，如图 5.12 所示。

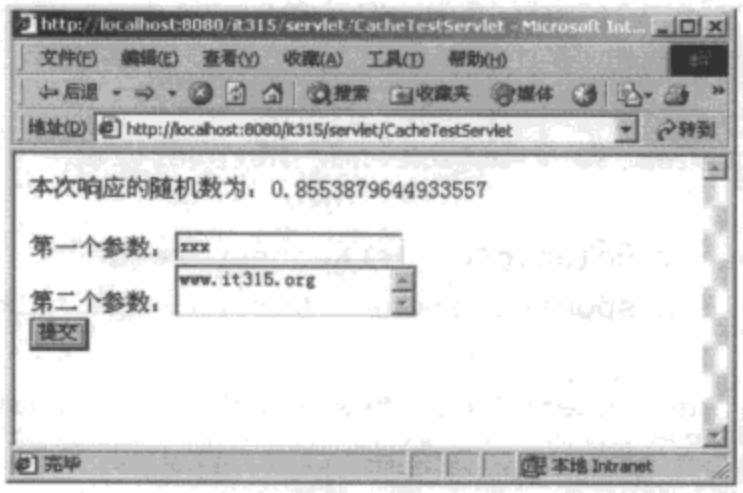


图 5.12

单击图 5.12 中的“提交”按钮后，由于表单提交的目标资源根本不存在，服务器返回了一个报告 404 错误的页面。单击浏览器工具栏中的“后退”按钮，浏览器显示的页面效果与图 5.12 完全一样，上次在 FORM 表单的文本框中填写的内容依然存在，显示的随机数也与上次相同，这说明浏览器此时没有访问服务器，而是直接从缓存中取出了先前的页面。

04 取消 CacheTestServlet.java 源文件中的注释，即增加那些禁止浏览器缓存的语句，重新编译 CacheTestServlet.java 源文件。确信 Tomcat 重新加载 CacheTestServlet 后，刷新对 CacheTestServlet 的访问，再次重复第 **04** 步的操作过程。尽管浏览器也能后退到 CacheTestServlet 页面，但后退到的页面中显示的随机数与表单提交前的页面中显示的随机数不同，FORM 表单内的文本框中也不再有上次填写的内容，这说明浏览器后退到的 CacheTestServlet 页面是重新访问服务器后得到的结果，而不是从浏览器缓存中取出的结果，从而可以推断出浏览器此时没有缓存 CacheTestServlet 输出的页面内容。

5.5.4 使用<meta>标签模拟响应消息头

利用 HTTP 消息的响应头字段，可以让浏览器完成各种有用的功能，但是，这需要通过编写 Web 服务器端的程序来实现。如果不会服务器端编程的普通 HTML 页面制作者也想借助 HTTP 消息的响应头字段来实现一些特殊功能，他们该怎么办呢？为此，HTML 语言中专门定义了<meta>标签的 http-equiv 属性来在 HTML 文档中模拟 HTTP 响应消息头，当浏览器读取到 HTML 文档中具有 http-equiv 属性的<meta>标签时，它会用与处理 Web 服务器发送的响应消息头一样的方式来进行处理。这样，静态 HTML 页面的制作者不用编写 Web 服务器端的程序，就可以在静态 HTML 页面中实现 HTTP 响应消息头的功能了。例如，如果不想让浏览器缓存一个内容经常要被更新的静态 HTML 页面，那么可以在它的<head></head>标签对之间增加如下三条<meta>语句：

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

- <meta http-equiv="Expires" content="0">
- <meta http-equiv="Cache-Control" content="no-cache">
- <meta http-equiv="Pragma" content="no-cache">

<meta>标签可以模拟 HTTP 消息响应头的这一特点往往也会被一些喜欢搞恶作剧的人利用，因为即使一些<meta>语句没有位于 HTML 文档的<head></head>标签对之间，浏览器也能对它进行解释处理。例如，在某个网页的正文部分中增加如下的一条语句：

```
<meta http-equiv="Refresh" content="0;url=http://www.it315.org">
```

浏览器打开这个网页文档后，将立即跳转到 www.it315.org 站点的首页。在一些没有限制使用 HTML 语句的留言板和论坛系统中只要发表包含上面语句的留言或帖子，当任何用户浏览包含这条留言或帖子内容的页面时，浏览器不是把这条留言或帖子原样显示出来，而是把它当做 HTML 语句来执行，从而跳转到 www.it315.org 站点的首页，用户就无法继续浏览原来想看的内容了。

<meta>标签的一个最常见的应用就是用来设置当前页面的字符集编码。在前面讲解 Servlet 的中文输出问题时，读者已经了解到：如果浏览器显示网页文档时所采用的字符集编码与实体内容中的中文字符本身的字符集编码不一致，那么中文字符就会显示为乱码，所以，服务器端程序需要使用 Content-Type 头字段指定实体内容的字符集编码，以便浏览器自动调整显示网页文档内容时所采用的字符集编码。同样的道理，对于静态的 HTML 页面，也需要通过某种方式来将它的字符集编码信息告诉给浏览器，这就可以通过将<meta>标签的 http-equiv 属性设置为“Content-Type”来解决。大家在浏览许多中文网站的静态页面时，只要查看页面的源文件内容，基本上都可以看到其开始部分包含有如下的一条<meta>语句：

```
<meta http-equiv="Content-Type" content="text/html; charset=GB2312">
```

如果一个采用 GB2312 编码的中文静态网页没有使用上面的<meta>语句，那么，当用户浏览器访问了某个英文的站点后再来访问这个页面，其中的中文字符很可能就会显示为乱码。这是因为浏览器访问某个英文站点的页面后，浏览器显示网页文档的字符集编码可能被该页面自动设置成了 ISO8859-1，在这种情况下，除非用户将浏览器显示网页文档的字符集编码手工设置为 GB2312 编码，否则，浏览器就无法正常显示 GB2312 编码的中文字符。所以，在每个静态的中文 HTML 页面的开始处都应增加一条<meta>语句来设置页面的字符集编码，已经成为了设计 HTML 页面的指导性原则和验收标准。

使用<meta>标签解决浏览器显示网页时的中文乱码问题

01 用记事本程序编写如例程 5-4 和例程 5-5 所示的两个 HTML 文档。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

例程 5-4 english.html

```
<meta http-equiv="Content-Type" content="text/html; charset=ISO8859-1">
english
```

例程 5-5 chinese.html

```
中国
```

02 查看浏览器的“查看”→“编码”→“自动选择”菜单是否处于选中（前面打勾）状态，是则单击该菜单去掉选中状态。然后用浏览器直接打开 english.html 文件，接着再打开 chinese.html 文件，可以看到浏览器中显示的中文为乱码。

03 在 chinese.html 源文件的开始处增加如下一条语句：

```
<meta http-equiv="Content-Type" content="text/html; charset=GB2312">
```

刷新浏览器对 chinese.html 页面的访问，这时浏览器显示出了正常的中文字符。

04 用记事本程序打开 chinese.html 文件，将它另存为“UTF-8”编码格式，然后刷新浏览器对 chinese.html 页面的访问。尽管 chinese.html 文件的字符集编码为“UTF-8”，而其中的<meta>标签指定的字符集编码为“GB2312”，但是，浏览器仍然正常显示出了 chinese.html 文件中的中文字符。单击浏览器的“查看”→“编码”菜单，可以看到当前的字符集编码被自动设置成了“Unicode(UTF-8)”。单击“西欧(ISO)”子菜单，然后再单击浏览器的“查看”→“编码”菜单，可以看到浏览器当前显示页面的字符集状态仍然保持为“Unicode(UTF-8)”不变。这个实验说明，如果网页文档采用 UTF-8 编码，那么，浏览器能够自动进行识别，不会再按照 Content-Type 头字段中指定的字符集编码来显示当前网页文档，而是始终采用 UTF-8 编码来显示。

多学两招

网页文档应尽量采用 UTF-8 编码

笔者曾接触过某一版本的 Tomcat，当访问服务器上的任何静态 HTML 页面时，它总是发送字符集编码为“ISO8859-1”的 Content-Type 头字段，因此，浏览器总是无法正常显示 GB2312 编码的中文网页。即使在这些页面的开始部分增加<meta>标签来指定当前页面的字符集编码为“GB2312”，浏览器也仍然无法正常显示中文字符，这是因为 Web 服务器实际发送的响应头字段比实体内容中用<meta>标签模拟的头字段的优先权高，浏览器优先采用 Web 服务器实际发送的响应头。对于这种情况，惟一的解决办法就是将所有的中文静态 HTML 页面都采用 UTF-8 编码保存。

因为所有标准的浏览器和客户机终端都支持 UTF-8 编码，所以最好是采用 UTF-8 编码来编写网页文档，这样，同一个网页文档可以被多个国家和地区的人们正常浏览并避免一些特殊情况下所造成的字符显示乱码。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

5.6 创建响应正文

ServletResponse 是 Servlet 程序与 Servlet 引擎进行通信的接口，Servlet 程序通过 ServletResponse 对象将响应正文传递给 Servlet 引擎，再由 Servlet 引擎将响应正文输出到客户端。如果两个模块之间要传递大量的数据，最好是以输入或输出流来作为这两个模块传递数据的接口方式。ServletResponse 遵循这种以 IO 流来传递大量数据的设计理念，其中没有定义直接输出响应正文的方法，而是定义了一些方法来返回 Servlet 引擎提供的输出流对象，再由 Servlet 程序通过这个输出流对象将响应正文传写入到 Servlet 引擎。

5.6.1 getOutputStream 与 getWriter 方法

为了方便 Servlet 程序灵活选择按字节或按字符的形式输出响应正文，Servlet 引擎内部提供了字节输出流和字符输出流来供 Servlet 程序选择，与此对应，ServletResponse 接口中定义了两个方法：getOutputStream 和 getWriter，getOutputStream 方法用于返回 Servlet 引擎创建的字节输出流对象，getWriter 方法用于返回 Servlet 引擎创建的字符输出流对象。getOutputStream 和 getWriter 这两个方法互相排斥，调用了其中的任何一个方法后，就不能再调用另一方法。

getOutputStream 方法返回的字节输出流对象的类型为 ServletOutputStream，ServletOutputStream 是 OutputStream 的子类，它可以直接输出字节数组中的二进制数据。如果 Servlet 要输出二进制格式的响应正文，就应该使用 getOutputStream 方法返回的 ServletOutputStream 对象。

getWriter 方法将 Servlet 引擎的数据缓冲区包装成 PrintWriter 类型的字符输出流对象后返回，PrintWriter 对象可以直接输出字符文本内容，所以，它专用于输出内容全为文本字符的网页文档。

Servlet 程序向 ServletOutputStream 或 PrintWriter 对象中写入的数据将被 Servlet 引擎获取，Servlet 引擎将这些数据当做响应消息的正文，然后再与响应状态行和各响应头组合后输出到客户端。Servlet 引擎在将响应消息输出到客户端前，可以调整响应状态行、响应头和响应正文的位置顺序，尽管如此，Servlet 开发人员最好还是在使用 PrintWriter 或 ServletOutputStream 对象之前，调用所有设置 HTTP 的响应头信息的方法，特别是要注意设置响应内容的 MIME 类型的 Content-Type 头字段。设置响应内容大小的 Content-Length 头字段不用 Servlet 开发人员关心，Servlet 容器可以自动设置和调整。

Serlvet 的 service 方法结束后，Servlet 引擎将检查 getWriter 或 getOutputStream 方法返回的输出流对象是否已经调用过 close 方法，如果没有，Servlet 引擎将调用 close 方法关闭该输出流对象。所以，即使在 Serlvet 的 service 方法中没有调用输出流对象的 close 方法，也不会发生输出流未关闭的问题，但是 Servlet 开发人员最好还是自己在 Serlvet 程序的 service 方法内调用 Printwriter 或 ServletOutputStream 对象的 close 方法，以便尽快释放相关资源。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

在创建输出响应正文时，到底是使用 ServletOutputStream 对象，还是使用 PrintWriter 对象？要想能根据具体情况而做出明智的选择，还需要理解和掌握以下一些要点：

- PrintWriter 对象输出字符文本内容时，它内部还是会先将字符串转换成某种字符集编码的字节数组再进行输出，使用 PrintWriter 对象的好处就是不用编程人员自己来完成字符串到字节数组的转换。由于大多动态产生的网页文档的内容都是字符文本，所以，PrintWriter 对象的应用比 ServletOutputStream 对象的应用更为常见和普遍。
- 使用 ServletOutputStream 对象也能输出内容全为文本字符的网页文档，但是，如果网页文档内容是在 Servlet 程序内部使用文本字符串动态拼凑和创建出来的，则需要先将字符文本转换成字节数组后输出，这时就不如使用 ServletResponse 的 getWriter 方法返回的 PrintWriter 对象直接对字符文本进行输出方便。
- 如果一个网页文档内容全部为字符文本，但是这些内容可以直接从一个字节输入流中读取出来，然后再原封不动地输出到客户端，那么就应该使用 ServletOutputStream 对象直接进行输出，而不要使用 PrintWriter 对象进行输出。因为采用 PrintWriter 对象进行输出时，还需要将这些内容以字符文本的形式读取到程序中，这样，在读取时涉及了字节到字符的转换过程，在输出时又涉及了字符到字节的转换过程，显然存在着效率低下的问题，并且由于多了一些额外的过程后，更容易出现字符编码转换的错误。

5.6.2 关于 getWriter 方法的一些相关疑问

在 Serlvet 程序中可以多次调用 HttpServetResponse 对象的 getWriter 方法吗？每次调用 getWriter 方法返回的 PrintWriter 对象的引用变量指向的是同一个 PrintWriter 对象吗？PrintWriter.println 方法与 PrintWriter.print 方法有什么区别呢？下面通过些实验步骤来回答这些问题。

关于 getWriter 方法的一些相关疑问

01 按例程 5-6 编写一个名为 TestOutServlet 的 Servlet 程序。

例程 5-6 TestOutServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class TestOutServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        IOException
    {
        response.setContentType("text/html");
        PrintWriter out1 = response.getWriter();
        PrintWriter out2 = response.getWriter();
        if(out1 == out2)
        {
            out1.print("content outputed by out1<br>");
            out2.println("content outputed by out2<br>");
        }
    }
}
```

编译 TestOutServlet.java 源文件，确保编译后生成的 class 文件存放在了<tomcat 的安装目录>\Webapps\it315\Web-INF\classes 目录中。

02 修改<tomcat 的安装目录>\Webapps\it315\Web-INF\Web.xml 文件，在其中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>TestOutServlet</servlet-name>
    <servlet-class>TestOutServlet</servlet-class>
</servlet>
……
……
<servlet-mapping>
    <servlet-name>TestOutServlet</servlet-name>
    <url-pattern>/servlet/TestOutServlet</url-pattern>
</servlet-mapping>
```

保存 Web.xml 文件后，重新启动 Tomcat。

03 在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/servlet/TestOutServlet
```

浏览器窗口中显示出了如下两行内容：

```
content outputed by out1
content outputed by out2
```

这个结果说明 out1 和 out2 引用变量指向的是同一个 PrintWriter 对象，通过 out1 和 out2 写入的数据将按调用的顺序写入到同一输出流中。同样，多次调用 getOutputStream 方法返回的也将是同一个 ServletOutputStream 对象的引用，通过这些引用变量写入的数据将按调用的顺序写入到同一输出流中。

04 单击浏览器的“查看”→“源文件”菜单，在源文件中看到的内容如下：

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
content outputed by out1<br>content outputed by out2<br>
```

在客户端接收到的源文件中，TestOutServlet 程序中的两条打印语句输出的内容之间没有换行，不便于人们阅读源文件内容。接着将 TestOutServlet.java 程序中的 out.println 语句中的 print 修改成 println，重新访问 TestOutServlet，再次查看浏览器接收到的源文件，内容如下：

```
content outputed by out1<br>
content outputed by out2<br>
```

这次 TestOutServlet 程序中的两条打印语句输出的内容之间有了换行。可见，PrintWriter.println 方法与 PrintWriter.print 方法的区别在于它们输出到客户端的内容后面是否有换行，但浏览器显示不出这个换行，这个换行主要是为了增强浏览器接收到的源文件的易读性和易理解性。

5.6.3 输出缓冲区

Servlet 程序输出的 HTTP 消息的响应正文不是直接发送到客户端，而是首先被写入到了 Servlet 引擎提供的一个输出缓冲区中，这个缓冲区就像一个临时的蓄水池，直到输出缓冲区被填满或者 Servlet 程序已经写入了所有的响应内容，缓冲区中的内容才会被 Servlet 引擎发送到客户端。使用输出缓冲区后，Servlet 引擎就可以将响应状态行、各响应头和响应正文严格按照 HTTP 消息的位置顺序进行调整后再输出到客户端，特别是可以自动对 Content-Length 头字段进行设置和调整。读者可以想想，如果没有响应正文的输出缓冲区，Servlet 引擎是无法在向客户端输出响应正文之前知道响应正文的大小，也就不可能自动设置 Content-Length 头字段了。如果在提交响应到客户端时，输出缓冲区中已经装入了所有的响应内容，Servlet 引擎将计算响应正文部分的大小并自动设置 Content-Length 头字段。如果在提交响应到客户端时，输出缓冲区中装入的内容只是全部响应内容的一部分，那么 Servlet 引擎将无法再计算 Content-Length 头字段的值，它将使用 HTTP 1.1 的 chunked 编码方式（通过设置 Transfer-Encoding 头字段来指定）传输响应内容，这样就不用设置 Content-Length 头字段了。

ServletResponse 定义了若干与缓冲区有关的方法，下面是这些方法的介绍：

1. setBufferSize 方法

setBufferSize 方法用于设置期望的输出缓冲区大小，Servlet 引擎实际使用的缓冲区的大小不一定等于该设置值，但不会小于该设置值。

2. getBufferSize 方法

getBufferSize 方法用于返回 Servlet 引擎实际使用的缓冲区大小。

3. flushBuffer 方法

flushBuffer 方法用于将输出缓冲区的内容强制输出到客户端，如果这是当前响应的第

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

一次向客户端实际输出数据，响应状态行和各个响应头也会被输出到客户端。

4. reset 方法

reset 方法用于清空输出缓冲区中的内容，以及设置的响应状态码和各个响应头。如果当前响应已经向客户端输出过部分内容，这个方法将抛出 IllegalStateException 异常。要注意区分刷新缓冲区（flushBuffer 方法）与清空缓冲区（reset 方法）的差异，前者是将缓冲区的内容提前提交给客户端后腾出缓冲区空间，而后者是将缓冲区的内容作废后腾出缓冲区空间。

5. isCommitted 方法

判断是否已经提交了部分响应内容到客户端，如果已提交了，则返回 true，否则，返回 false。

Servlet 引擎处理输出缓冲区的细节

01 按例程 5-7 编写一个名为 BufferServlet 的 Servlet 程序。

例程 5-7 BufferServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class BufferServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
    //将缓冲区的大写设置为 0，接着在后台查看实际缓冲区的大小
        response.setBufferSize(0);
        int len = response.getBufferSize();
        System.out.println(len);

        PrintWriter out = response.getWriter();
        //让输出的内容正好填满整个缓冲区
        for(int i=0; i<len/*-1*/; i++)
        {
            out.print("w");
        }
    }
}
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

编译 BufferServlet.java 源文件，确保编译后生成的 class 文件存放在了<tomcat 的安装目录>\Webapps\it315\Web-INF\classes 目录中。

02 修改<tomcat 的安装目录>\Webapps\it315\Web-INF\Web.xml 文件，在其中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>BufferServlet</servlet-name>
    <servlet-class>BufferServlet</servlet-class>
</servlet>
……
……
<servlet-mapping>
    <servlet-name>BufferServlet</servlet-name>
    <url-pattern>/servlet/BufferServlet</url-pattern>
</servlet-mapping>
```

保存 Web.xml 文件后，重新启动 Tomcat。

03 在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/servlet/BufferServlet
```

查看一下显示效果后，接着在命令行窗口中执行 telnet 127.0.0.1 8080 命令，在成功连接后启动的 Telnet 程序窗口中，输入如下两行内容：

```
GET /it315/servlet/BufferServlet HTTP/1.1
Host:
```

接着用回车键输入一个空行后，Telnet 程序窗口中显示出 Web 服务器返回的响应信息，如图 5.13 所示。

如果在命令行窗口中不能看到所有内容，请调整命令行窗口的屏幕缓冲区大小后再重复上面的操作。在图 5.13 中可以看到，Servlet 引擎没有设置 Content-Length 头字段，它采用的是 chunked 编码方式来传输 BufferServlet 的响应内容，其中的 2000 是一个十六进制的数据，对应的十进制数为 2×163，即 8092，在运行 Tomcat 的命令行窗口中打印出的输出缓冲区的大小正是这个值。因为这里写入的内容正好充满了缓冲区，Servlet 引擎需要将缓冲区中的内容发送给客户端，而它此时无法断定 BufferServlet 是否还会继续写入内容，即无法断定整个响应内容的大小，所以它只能采用 chunked 编码方式来传输缓冲区的内容。

04 在 BufferServlet.java 源文件中，将输出内容的循环次数减 1，也就是将程序中的/*-1*/部分的注释取消掉。重新编译 BufferServlet.java 源文件并等待 Tomcat 重新装载 BufferServlet 后，重复步骤 **03** 中的 telnet 操作命令，Telnet 程序窗口中显示出的 Web 服务器返回的响应信息如图 5.14 所示。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

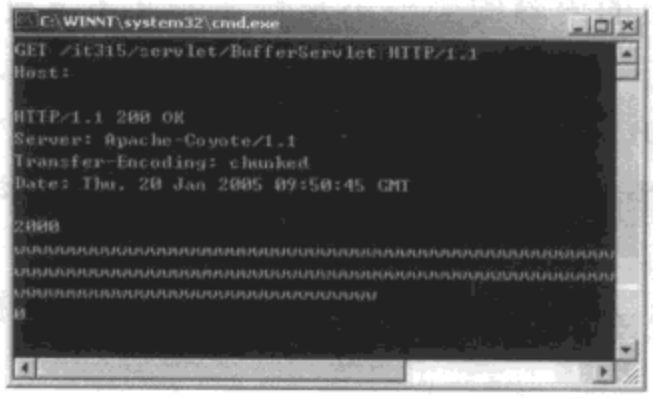


图 5.13

图 5.14

在图 5.14 中可以看到，Servlet 引擎这次没有采用 chunked 编码方式来传输 BufferServlet 的响应内容，而是设置了 Content-Length 头字段。因为 Servlet 程序运行结束后，输出缓冲区还没有被填满，Servlet 引擎在向客户端输出响应正文时，可以确定响应正文的长度。

5.6.4 实现动态文件内容的下载

在 Web 服务器上实现资源下载功能是一件非常容易的事情，只要让超链接的 URL 地址指向一个 exe 或 zip 等类型的文件，用户单击这个超链接就可以将该资源文件下载到客户端。但是如果要下载的文件并不真正存在于 Web 服务器的文件系统中，而是需要用一个 Servlet 程序临时在服务器内存中动态产生后再传送到客户端，那该如何实现呢？

例如，在 Web 站点上建立的邮件收发系统，需要让用户能通过浏览器来下载邮件附件，用户只要单击网页上的指向邮件附件的超链接，邮件附件就可以被下载到用户本地保存。一封邮件中的附件不是单独保存的，它们与正文部分采用 MIME 格式组合在一个文件中进行存储和传输，从这个文件中可以解析出邮件正文和各个附件的内容。当用户要下载某个邮件附件时，由于邮件的附件文件并没有真正地存在于 Web 服务器中，Web 服务器需要从邮件服务器上取出整封邮件的内容，然后从整封邮件内容中解析出属于该附件文件的内容部分，再将解析出的附件内容传递给浏览器。Web 浏览器、Web 服务器、邮件服务器的工作关系如图 5.15 所示。

图 5.15

要实现图 5.15 中的邮件附件下载功能，Servlet 程序仅仅将解析出来的邮件附件内容传递给浏览器是不行的，还需要设置一些特殊的响应消息头，具体要求如下：

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

（1）Servlet 程序必须告诉浏览器其所输出的内容的类型不是普通的文本文件或 HTML 文件，而是一个要保存到本地的下载文件，这就需要通过 HttpServletResponse.setContentType 方法设置 Content-Type 头字段的值为浏览器无法使用某种方式或激活某个程序来处理的 MIME 类型，例如，“application/octet-stream”或“application/x-msdownload”等。

（2）需要通过 HttpServletResponse.setHeader 方法设置 Content-Disposition 头的值为“attachment; filename =文件名”。

（3）因为附件文件可以是各种类型的文件，要将附件文件中的内容传送给客户端，其中的内容应被当做二进制处理，所以应该调用 HttpServletResponse.getOutputStream 方法返回的 ServletOutputStream 对象来向客户端写入附件文件内容，而不应使用 HttpServletResponse.getWriter 方法返回的 PrintWriter 对象。

实现动态文件内容的下载

01 按例程 5-8 编写一个模拟提供邮件附件下载功能的 Servlet 程序，邮件附件的内容不是从邮件服务器上获取和解析出来的，而是直接用一个简单的字符串生成的。

例程 5-8　AttachmentServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Attachment extends HttpServlet
{
    public void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,IOException
    {
        response.setContentType ("application/x-msdownload");
        response.addHeader("Content-Disposition","attachment;filename=
        aaa.txt");
        ServletOutputStream sos = response.getOutputStream();
        sos.write("hello www.it315.org".getBytes());
        sos.close();
    }
}
```

编译 AttachmentServlet.java 源文件，确保编译后生成的 class 文件存放在了<tomcat 的安装目录>\Webapps\it315\Web-INF\classes 目录中。

02 修改<tomcat 的安装目录>\Webapps\it315\Web-INF\Web.xml 文件，在其中的相

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>AttachmentServlet</servlet-name>
    <servlet-class>AttachmentServlet</servlet-class>
</servlet>
……
……
<servlet-mapping>
    <servlet-name>AttachmentServlet</servlet-name>
    <url-pattern>/servlet/AttachmentServlet</url-pattern>
</servlet-mapping>
```

保存 Web.xml 文件后，重新启动 Tomcat。

03 在<tomcat 安装目录>\Webapps\it315 目录中编写一个名为 attachment.html 的 html 文件，这个文件中含有一个指向 AttachmentServlet 的超链接，如例程 5-9 所示。

例程 5-9 attachment.html

```
<meta http-equiv="Content-Type" content="text/html;charset=GB2312">
<a href="servlet/AttachmentServlet">下载附件</a>
```

在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/attachment.html
```

然后单击浏览器显示页面中的“下载附件”超链接，浏览器窗口中显示出如图 5.16 所示的下载提示对话框，单击对话框上的保存按钮即可完成动态文件内容的下载。

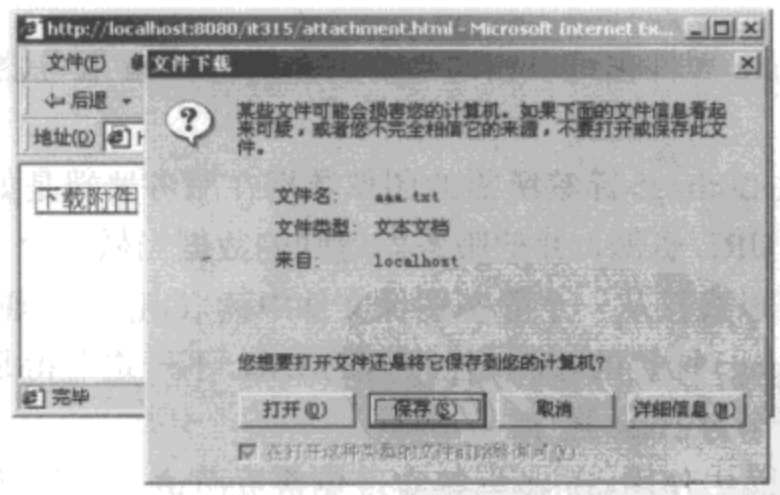


图 5.16

5.6.5 图像访问计数器

统计和显示页面的访问次数是一个常见的 Web 应用，因为在网页每次被访问时，页面的访问次数都要发生改变，所以这个功能必须通过服务器端的程序来实现。页面访问计数

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

器在服务器端的基本执行过程为：首先从数据库或其他存储系统中取出页面原来的访问次数，然后将原来的访问次数加 1 后再作为当前访问次数送给浏览器显示，同时将当前访问次数存储回数据库或其他存储系统中，以便下次访问该页面时使用。

但是，一些 Web 站点只能输出静态页面内容，没有开放运行服务器端程序的功能。例如，一些提供免费网页空间的站点，在客户端访问任何类型的文件时，服务器都会原封不动地输出给浏览器去处理，显然无法直接在这些只支持静态内容的 Web 站点上编写服务器端程序来实现页面访问次数的统计和显示功能。于是，一些具有执行服务器端程序功能的 Web 站点推出了免费的页面访问计数器，只要在位于任何站点的一个静态 HTML 页面中增加一条该站点提供的 HTML 语句，该语句就能显示出该静态页面的访问次数，从而帮助静态 HTML 页面实现了访问次数的统计和显示功能。一个站点要想能统计另外一个站点上的某个 HTML 页面的访问次数，必须让任何一个浏览器在每次访问那个 HTML 页面都通知这个站点一下，这可以通过在静态 HTML 页面中增加两种特殊的标签来实现：<img>标签和设置 src 属性的<script>标签。下面只分析使用<img>标签的应用情况，对于使用设置 src 属性的<script>标签的应用情况，则作为一道习题交给读者自己去完成。

要利用<img>标签为静态 HTML 页面实现访问次数的统计和显示功能，需要理解<img>标签的三个重要特性：

（1）一个包含有图像的网页文件中并没有包含真正的图像数据内容，而只是使用<img>标签指明了图像的 URL 地址，如下所示：

```
本网页已被浏览了<img src= "count.gif ">次
```

如果浏览器在解析网页文档的过程中遇到了<img>标签，它根据<img>标签的 src 属性所指定的 URL 地址去访问 Web 服务器，从 Web 服务器上获取图像数据后，再在<img>标签的位置处显示出来。

（2）<img>标签的 src 属性也可以指向当前页面所在 Web 服务器之外的其他 Web 服务器上的图像文件。

（3）浏览器并不关心<img>标签所需的图像数据在服务器端是如何产生的，它只知道去访问 src 属性指定的 URL 资源，并把服务器返回的数据当做一个图像的内容来显示。服务器返回的图像数据可以直接从一个静态图像文件中读取，也可以通过 Servlet 程序在内存中动态创建。所以，<img>标签的 src 属性指向的 URL 不一定非得是事先静态存在的静态图像文件，也可以是一个 Servlet 程序。

在静态 HTML 页面中使用<img>标签实现免费页面访问计数器的工作原理如图 5.17 所示，Web 站点 A 上的静态页面 count.html 内有一条<img>标签语句，<img>标签的 src 属性指向了 Web 站点 B 上的一个名为 CountServlet 的 Serlvet 程序，当浏览器显示从站点 A 上获得的 count.html 页面时，它将根据其中的<img>标签的 src 属性去访问 Web 站点 B 上的 CountServlet 程序以获得要显示的图像数据，CountServlet 则创建出具有该页面的访问次数的图像回送给浏览器显示。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

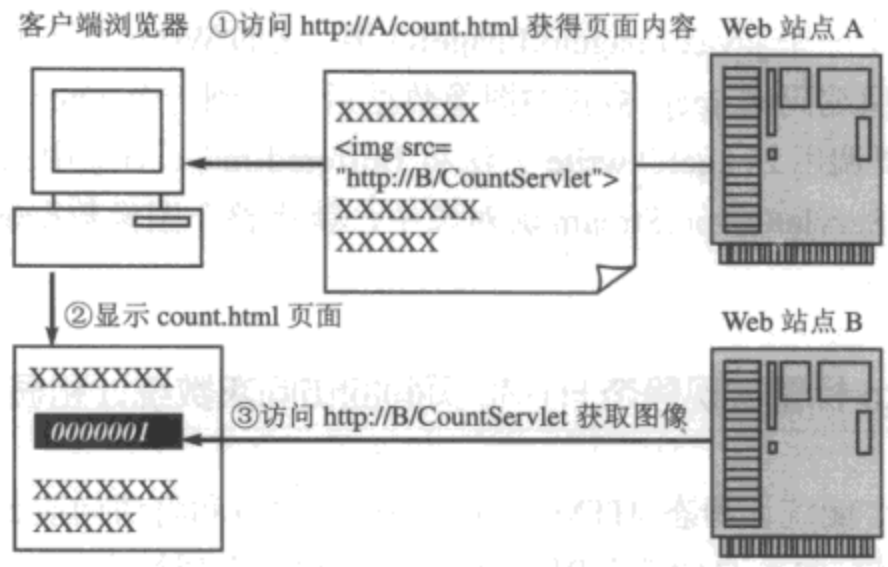


图 5.17

因为 Web 站点 B 上并不存在客户端要访问的图像文件，而是由 Servlet 程序在内存中临时动态生成图像内容后传送到客户端的，所以，尽管静态页面内的<img>标签所指向的这个 Servlet 程序的 URL 地址是固定不变的，但是，它每次被访问时所返回的图像内容却是可以不断变化的。

要编写一个用<img>标签统计和显示静态 HTML 页面的访问次数的 Servlet 程序，需要考虑以下一些技术细节：

- Servlet 程序输出的图像格式为 jpeg，所以，它应告诉浏览器其所输出的实体内容的 MIME 类型为 image/jpeg，浏览器才会将它所接收到的数据当做一个图像进行处理。
- 因为图像是二进制数据，所以应该调用 HttpServletResponse.getOutputStream 方法返回的 ServletOutputStream 对象来向客户端写入图像数据，而不应使用 HttpServletResponse.getWriter 方法返回的 PrintWriter 对象。
- java.awt.image.BufferedImage 类用于在内存中创建一幅图像，具体的图像内容则可以通过调用其图形上下文对象（java.awt.Graphics）的各种绘图方法生成，这里仅仅需要绘制一串数字，调用 Graphics.drawString 方法即可。BufferedImage 类创建的内存图像的初始背景颜色和绘图颜色都是黑色的，如果想改变背景颜色，还需要先设置相应的绘图颜色，然后调用 Graphics.fillRect 填充整个背景。
- 在内存图像中绘制访问次数时，必须限定显示的位数，如果访问次数超过七位，则用数字 9999999 显示，如果访问次数不足七位，则在前面补充相应个数的 0。
- 每个引用该 Servlet 程序的静态页面的 URL 都对应一个各自的访问次数，每个 URL 及其访问次数需要使用数据库系统来进行存储，对于简单的实验，也可以采用一个属性文件来进行存储。程序首先使用 Referer 请求头获得当前引用页面的 URL，然后去数据库或属性文件中检索该 URL 的访问次数，再将访问次数加 1 后保存进数据库或属性文件，并将访问次数绘制在内存图像中。如果属性文件中不存在该 URL 的信息，则说明该 URL 页面属于第一次统计访问。关于如何获得 Referer 请求头，

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

请参看 6.4 节的讲解。

- JDK 中提供了一个 javax.imageio.ImageIO 类，它的 Write 方法可以将 BufferedImage 对象中的图像编码成 jpeg 格式的图像数据后写入到一个 OutputStream 流对象中。因此，只需要调用 ImageIO.write 方法将 BufferedImage 对象中的图像编码成 jpeg 格式后写入到 ServletOutputStream 流对象中，就将整个图像数据输出给了客户端。

使用<img>标签实现静态 HTML 页面的访问次数统计和显示

01 编写一个能统计静态 HTML 页面的访问次数和将访问次数以图像形式返回给客户端的 Servlet 程序，每个页面的 URL 及访问次数用一个属性文件进行存储，如例程 5-10 所示。

例程 5-10 CountServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.awt.*;
import java.awt.image.*;
import javax.imageio.ImageIO;
import java.util.Properties;

public class CountServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,IOException
    {
        response.setContentType("image/jpeg");
        //设置浏览器不要缓存此图片
        response.setHeader("Pragma","No-cache");
        response.setHeader("Cache-Control","no-cache");
        response.setDateHeader("Expires", 0);

        ServletOutputStream sos = response.getOutputStream();

        BufferedImage image =
                new BufferedImage(80, 20, BufferedImage.TYPE_INT_RGB);
        Graphics g = image.getGraphics();
        /*g.setColor(Color.BLACK);
        g.fillRect(0, 0, width, height);*/
        g.setColor(Color.WHITE);
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        g.setFont(new Font(null,Font.ITALIC|Font.BOLD,18));
        String value = getAccessCount(request);
        int len = value.length();
        //最大访问次数不能超过 9999999
        if(len > 7)
        {
            value = "9999999";
        }
        else
        {
            String tmp="";
            for(int i=0;i<7-len;i++)
            {
                tmp = tmp +"0";
            }
            value = tmp + value;
        }
        g.drawString(value,0,18);
        //结束图像的绘制过程，完成图像
        g.dispose();

        ImageIO.write(image, "JPEG", sos);
        sos.close();
    }

    private String getAccessCount(HttpServletRequest request)
    {
        //得到引用页面的 URL 地址，并以此作为访问次数的检索关键字
        String pageKey = request.getHeader("referer");
        if(pageKey == null)
        {
            return "0";
        }

        Properties settings = new Properties();
        //count.txt 等内部文件最好保存在 Web-INF 目录中
        String countFilePath = getServletContext().getRealPath
         ("/count.txt");
        try
        {
            //下面的语句没有使用 close 方法关闭流，有缺陷!
            settings.load(new FileInputStream(countFilePath));
        }
        catch(Exception e){}
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        String count = "0";
        try
        {
            count = settings.getProperty(pageKey);
            if(count == null)
            {
                count = "0";
            }
            int c = Integer.parseInt(count) + 1;
            count = new Integer(c).toString();
            settings.put(pageKey,count);
            //下面的语句也没有使用 close 方法关闭流，有缺陷!
            settings.store(new FileOutputStream(countFilePath),
                    "the page is accessed:");
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
        }
        return count;
    }
}
```

编译 CountServlet.java 源文件，确保编译后生成的 class 文件存放在了<tomcat 的安装目录>\Webapps\it315\Web-INF\classes 目录中。

02 修改<tomcat 的安装目录>\Webapps\it315\Web-INF\Web.xml 文件，在其中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>CountServlet</servlet-name>
    <servlet-class>CountServlet</servlet-class>
</servlet>
……
……
<servlet-mapping>
    <servlet-name>CountServlet</servlet-name>
    <url-pattern>/servlet/CountServlet</url-pattern>
</servlet-mapping>
```

保存 Web.xml 文件后，重新启动 Tomcat。

03 在任何计算机上的任何 HTML 页面中只要加入如下语句：

```
<img src="http://可对外访问的主机地址:8080/it315/servlet/CountServlet">
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

就可以统计和显示该页面的访问次数了。在<tomcat 安装目录>\Webapps\it315 目录中编写一个名为 count.html 的 html 文件，如例程 5-11 所示。

例程 5-11 count.html

```
<meta http-equiv="Content-Type" content="text/html;charset=GB2312">
本网页已被浏览了<img src="http://localhost:8080/it315/ servlet/
CountServlet">次。
```

在浏览器地址栏中输入如下地址：

http://localhost:8080/it315/count.html

浏览器中显示的结果如图 5.18 所示。

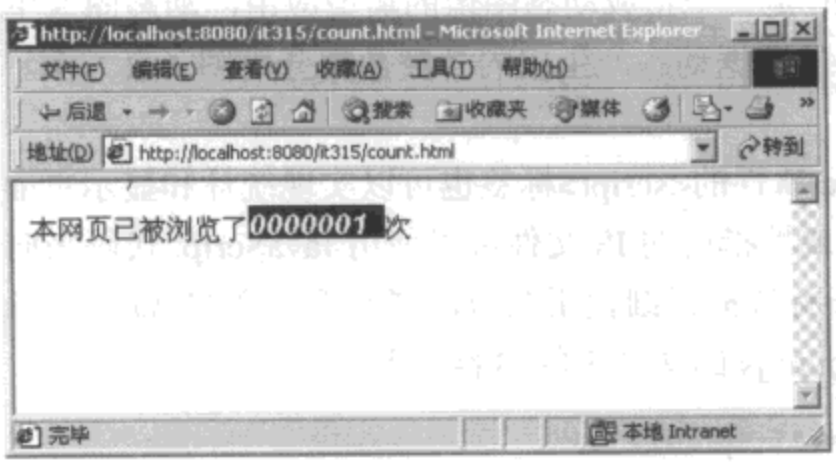


图 5.18

刷新浏览器的访问几次，可以看到图片中的数值随之增长。

下面对 CountServlet 程序实现的图像访问计数器进行更进一步的探讨和思考。

（1）在上面的 CountServlet 程序中，如果图像数据的大小没有超过 Servlet 引擎输出缓冲区的大小，Servlet 引擎将自动计算和设置响应消息的 Content-Length 头字段，否则，Servlet 引擎在响应消息中不设置 Content-Length 头字段，而是使用 chunked 编码方式传输响应内容。如果 CountServlet 希望在响应消息中总是使用 Content-Length 头字段，而不使用 chunked 编码方式，那么应该使用 ImageIO 类将 BufferedImage 对象中的图像编码成 jpeg 格式后写入到一个 ByteArrayOutputStream 流对象中，然后将 ByteArrayOutputStream 对象中的数据转换成字节数组，接着将 Content-Length 头字段设置为字节数组的大小，最后再将整个字节数组写入到 ServletOutputStream 对象中，具体的程序代码如下所示：

```
ByteArrayOutputStream bos = new ByteArrayOutputStream();
ImageIO.write(image, "JPEG", bos);
byte [] buf = bos.toByteArray();
response.setContentLength(buf.length);
bos.close();
sos.write(buf);
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

（2）采用 Referer 请求头来区分和跟踪每个静态 HTML 页面的方式并不可靠，因为同一个页面可以使用不同的 URL 形式访问，例如，http://127.0.0.1:8080/it315/count.html 和 http://localhost:8080/it315/count.html 访问的是同一个页面，但是，上面的 CountServlet 程序会把它们当做两个不同的页面分别进行统计。在实际应用中，往往采用为每个页面分配一个 id 号的方式来区分和跟踪每个静态 HTML 页面，然后在要统计和显示访问次数的页面中加入如下语句：

```
<img src=http://主机地址:8080/it315/CountServlet?id=本页面的 id 号>
```

CountServlet 根据传递的 id 参数，就知道是哪个网页被访问了，然后在数据库或文件系统中查询出该 id 号所对应的访问次数，再将访问次数加 1 后保存进数据库或文件系统，同时创建出显示访问次数的图像传递给浏览器。

（3）CountServlet 是一个生成动态图像的典型应用，动态图像生成技术非常有用，例如根据数据库系统中的数据动态产生出的各类数据分析图（直方图、饼状图、折线图等），甚至是股票走势图。

（4）使用设置 src 属性的<script>标签也可以实现统计和显示页面访问次数的功能，因为<script>标签的 src 属性指向的 JS 文件可以使用 JavaScript 代码来向当前页面插入一段文本内容。关于<script>标签的详细使用细节，请参看笔者的《JavaScript 网页开发》一书。

假设一个名为 count.js 的文件中的内容如下：

```
document.write("0000001");
```

在 count.js 文件的相同目录中有一个内容如下的 HTML 文件：

```
本网页已被浏览了<script type= "text/javascript" src= "count.js" ></script>次
```

该 HTML 文件的显示效果将如图 5.19 所示。

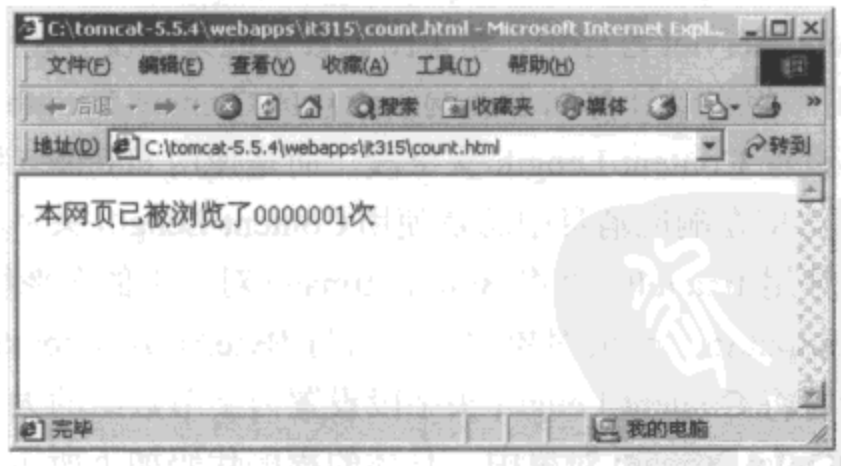


图 5.19

当浏览器在解释网页文档的过程中，遇到设置了 src 属性的<script>标签时，它将去访问 src 属性所指定的资源，并把服务器返回的内容当做一个 JavaScript 脚本文件处理。与<img>标签一样，<script>标签的 src 属性也可以指向当前页面所在服务器之外的其他服务器上的 JS 文件，并且 JS 文件的内容不一定要事先静态存在，而是可以由服务器程序动态

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

生成。如果让一个网页中的<script>标签的 src 属性指向一个 Servlet 程序，由这个 Servlet 程序来动态生成 count.js 文件内容，那么就很容易改变插入到网页中的访问计数值。使用<script>标签来帮助静态 HTML 页面实现访问次数的统计和显示功能，要比使用<img>标签简单和灵活，例如，可以让 Servlet 程序输出如下内容的 JavaScript 脚本代码：

```
document.write("<a href='www.it315.org' title='由 IT 资讯交流网统计'>" +
    "0000001</a>");
```

那么，访问计数值将以一个超链接的形式显示出来，用户单击这个超链接就可以到达提供免费的页面访问计数器的 Web 站点了，对该站点起到一定的广告宣传效果，这也是一些 Web 站点愿意提供免费的页面访问计数器的根本原因。另外，即使不用作免费的页面访问计数器，这种页面访问计数器在一个网站内部使用时也具有非常大的实用价值，例如，大家在浏览一些网站上的文章时，都可以看到每篇文章被浏览的次数，就是通过这种技术来实现的。

（5）借鉴网页访问计数器的设计思想统，www.it315.org 站点还为其他站点的页面提供了一个“显示来访者的 IP 地址和地区信息”的功能。当用户访问其他站点的一个网页时，只要该网页中使用了 www.it315.org 站点提供的一个<script>标签语句，浏览器便会根据<script>标签的 src 属性所指定的 URL 地址来访问 www.it315.org 站点的某个服务器程序。该服务器程序获取访问者的 IP 地址，并查询出该 IP 地址所在的地区（www.it315.org 站点的数据库系统中存储有各 IP 地址段与地区的对照表），然后将该 IP 地址和地区信息通过输出的 JavaScript 代码插入进用户最初访问的网页中。

多学两招

如何动态产生大小可变的图像

在一些网站中，能够根据用户的不同操作，将一幅图像以各种大小进行输出。由于 Java 语言的 JDK 中已经提供了相应的 API 类，所以在 Servlet 程序中很容易实现这种功能。

要将一幅图像以各种大小进行输出，除了需要涉及例程 5-10 中使用过的 java.awt.image.BufferedImage 和 java.awt.image.ImageIO 这两个类外，还需要使用到 java.awt.geom. AffineTransform 和 java.awt.image.AffineTransformOp 这两个类。AffineTransformOp 类用于完成图像的转换，其中定义的 filter 方法用于完成具体的转换操作，下面是其中一个 filter 方法的定义：

```
public final BufferedImage filter(BufferedImage src,BufferedImage dst)
```

可见，filter 方法转换前后的图像数据的类型都为 BufferedImage。如果要对一个图像文件进行转换，可以先调用 ImageIO.read 方法从该文件输入流中读取图像数据并生成一个 BufferedImage 对象，然后调用 AffineTransformOp.filter 方法进行转换，最后再调用 ImageIO.write 方法将转换得到的 BufferedImage 对象写入到一个文件输出流中。这个过程中还缺少一个关键的步骤：AffineTransformOp.filter 方法怎么知道该按何种方式对原

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

始图像进行转换呢？这就涉及了 AffineTransform 类。一个 AffineTransform 对象定义了一种具体的转换方式，在创建 AffineTransformOp 对象时，需要为其传递一个 AffineTransform 对象，下面是 AffineTransformOp 类的一个构造方法的定义：

```
public AffineTransformOp(AffineTransform xform,int interpolationType)
```

通过在创建 AffineTransformOp 对象时为其传递一个 AffineTransform 对象，就指定了 AffineTransformOp.filter 方法进行图像转换时的转换方式。对于 AffineTransform 对象，可以通过多种方式进行创建，如果仅仅是需要改变图像的大小，可以调用 AffineTransform.getScaleInstance 这个静态方法来创建，getScaleInstance 方法的定义如下：

```
public static AffineTransform getScaleInstance(double sx, double sy)
```

其中的两个参数分别指定了 x 和 y 方向的缩放比例。

关于 AffineTransformOp 和 AffineTransform 类的详细信息，请参看 JDK 帮助文档，由于其中涉及许多图像处理方面的专业知识，一般人都比较难于读懂。作者本人就属于读不太懂的人群，所以编写不出专业的图像处理程序，下面的讲解也仅限于在例程 5-10 的基础上动态改变输出图像的大小，开阔一下读者的视野，详细的实验步骤如下：

01 在例程 5-10 的开始处增加如下两条 import 语句：

```
import java.awt.geom.AffineTransform;
import java.awt.image.AffineTransformOp;
```

02 在例程 5-10 中的 ImageIO.write(image, "JPEG", sos);语句前增加如下内容：

```
double ratio = 1;
try
{
   ratio = Double.parseDouble(request.getParameter("ratio"));
}
catch(Exception e){}
AffineTransform atf = AffineTransform.getScaleInstance(ratio,ratio);
AffineTransformOp atf_op = new AffineTransformOp(
       atf,AffineTransformOp.TYPE_NEAREST_NEIGHBOR);
image = atf_op.filter(image,null);
```

上面的语句根据一个名为 ratio 的请求参数来设置图像的 x 和 y 方向的缩放比例，最后将转换后生成的 BufferedImage 对象又赋值给了 image 变量，以便原程序中的 ImageIO.write(image, "JPEG", sos);语句仍然可以使用。

03 重新编译 CountServlet.java 源文件，然后在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/servlet/CountServlet?ratio=2
```

看到浏览器中显示出的图像效果后，再将 ratio 参数改为其他值进行测试，比较每次输出的图像的大小。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

5.7 请求重定向与请求转发

在 Servlet 程序中，有时需要调用另外一个资源来对浏览器的请求进行响应，这可以通过两种方式来实现：其中一种就是调用 RequestDispatcher.forward 方法实现的请求转发；另外一种则是调用 HttpServletResponse.sendRedirect 方法实现的请求重定向。

5.7.1 RequestDispatcher 接口

Servlet API 中定义了一个 RequestDispatcher 接口，它定义了 RequestDispatcher 实例对象的方法。RequestDispatcher 实例对象是由 Servlet 引擎创建的，它用于包装一个要被其他资源调用的资源（例如，Servlet、HTML 文件、JSP 文件等），并可以通过其中的方法将客户端的请求转发给所包装的资源。

RequestDispatcher 接口中定义了两个方法：forward 方法和 include 方法，它们分别用于将请求转发到 RequestDispatcher 对象封装的资源和将 RequestDispatcher 对象封装的资源作为当前响应内容的一部分包含进来。forward 和 include 方法接收的两个参数必须是传递给当前 Servlet 的 service 方法的那两个 ServletRequest 和 ServletResponse 对象，或者是对它们进行了包装的 ServletRequestWrapper 或 ServletResponseWrapper 对象，以便调用和被调用的两个资源共享相同的 ServletRequest 和 ServletResponse 对象。关于 ServletRequestWrapper 和 ServletResponseWrapper 类的应用，请参看本书作者编写的《深入体验 Java Web 开发内幕——高级特性》一书中的关于过滤器的讲解。

ServletContext 接口中定义了两个用于获取 RequestDispatcher 对象的方法：

- getRequestDispatcher 方法
- 返回包装了某个路径所指定的资源的 RequestDispatcher 对象，传递给该方法的路径字符串必须以“/”开头，这个“/”代表的是当前 Web 应用程序的根目录。Web-INF 目录中的内容对 RequestDispatcher 对象是可见的，所以，传递给 getRequestDispatcher 方法的资源可以是 Web-INF 目录中不能被外界访问的文件。
- getNamedDispatcher 方法

返回包装了某个 Servlet 或 JSP 文件的 RequestDispatcher 对象，传递给该方法的参数是在 Web 应用程序部署描述符中为 Servlet 或 JSP 文件指定的友好名称。

在 ServletRequest 接口中也定义了一个 getRequestDispatcher 方法来获得 RequestDispatcher 对象，它与 ServletContext.getRequestDispatcher 方法的区别在于：传递给这个方法的参数除了可以采用以“/”开头的路径字符串，还可以采用非“/”开头的相对路径。RequestDispatcher 对象只能包装当前 Web 应用程序中的资源，所以，forward 方法和 include 方法只能在同一个 Web 应用程序内的资源之间转发请求和实现资源包含。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

5.7.2 用 include 方法实现资源包含

RequestDispatcher.include 方法用于将 RequestDispatcher 对象封装的资源内容作为当前响应内容的一部分包含进来，从而实现可编程的服务器端包含功能。被包含的 Servlet 程序不能改变响应消息的状态码和响应头，如果它里面存在这样的语句，这些语句的执行结果将被忽略。在调用 RequestDispatcher.include 方法时，Servlet 容器不会去调整 HttpServletRequest 对象中的信息，HttpServletRequest 对象仍然保持其初始的 URL 路径和参数信息，也就是说，在被调用者程序中检索当前访问路径时，得到的结果是调用者程序的 URL 路径，而不是被调用者程序自己的 URL 路径。

RequestDispatcher.include 方法的应用细节

01 编写两个 Servlet 程序：IncludingServlet 和 IncludedServlet，在 IncludingServlet 中通过 RequestDispatcher.include 方法调用 IncludedServlet，如例程 5-12 和例程 5-13 所示。

例程 5-12 IncludingServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class IncludingServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        //response.setContentType("text/html;charset=gb2312");
        PrintWriter out = response.getWriter();
        String china = "中国";
        RequestDispatcher rd = getServletContext().getRequestDispatcher
            ("/servlet/IncludedServlet?p1=" + china);
        out.println("before including" + "<br>");
        rd.include(request,response);
        out.println("after including" + "<br>");
    }
}
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

例程 5-13 IncludedServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class IncludedServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        response.setContentType("text/html;charset=GB2312");
        response.setCharacterEncoding("GB2312");
        PrintWriter out = response.getWriter();
        out.println("中国" + "<br>");
        out.println("URI:" + request.getRequestURI() + "<br>");
        out.println("QueryString:" + request.getQueryString() + "<br>");
        out.println("parameter p1:" + request.getParameter("p1") + "<br>");
    }
}
```

编译这两个 Servlet 源文件，确保编译后生成的 class 文件存放在了<tomcat 的安装目录>\Webapps\it315\Web-INF\classes 目录中。

02 在<tomcat 的安装目录>\Webapps\it315\Web-INF\Web.xml 文件中注册这两个 Servlet，并设置其映射 URL。在 Web.xml 文件中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>IncludingServlet</servlet-name>
    <servlet-class>IncludingServlet</servlet-class>
</servlet>
<servlet>
    <servlet-name>IncludedServlet</servlet-name>
    <servlet-class>IncludedServlet</servlet-class>
</servlet>

……
<servlet-mapping>
    <servlet-name>IncludingServlet</servlet-name>
    <url-pattern>/servlet/IncludingServlet</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>IncludedServlet</servlet-name>
    <url-pattern>/servlet/IncludedServlet</url-pattern>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
</servlet-mapping>
```

保存 Web.xml 文件后，重新启动 Tomcat。

03 在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/servlet/IncludingServlet
```

浏览器返回的结果如图 5.20 所示。

图 5.20

从图 5.20 中可以看到，被包含的 IncludedServlet 程序输出到客户端的中文字符显示为乱码，且 HTML 标记被当做普通文本内容直接显示出来，这说明 IncludedServlet 程序中的如下两条语句没起作用：

```
response.setContentType("text/html;charset=GB2312");
response.setCharacterEncoding("GB2312");
```

这是因为被包含的 Servlet 程序不能改变响应消息的状态码和响应头，如果它里面存在这样的语句，这些语句的执行结果将被忽略。

04 修改 IncludingServlet.java 源文件，取消其中调用 setContentType 方法的语句的注释，重新编译 IncludingServlet.java 源文件，等待 Tomcat 重新装载 IncludingServlet 后，在浏览器窗口中刷新对 IncludingServlet 的访问，返回结果如图 5.21 所示。

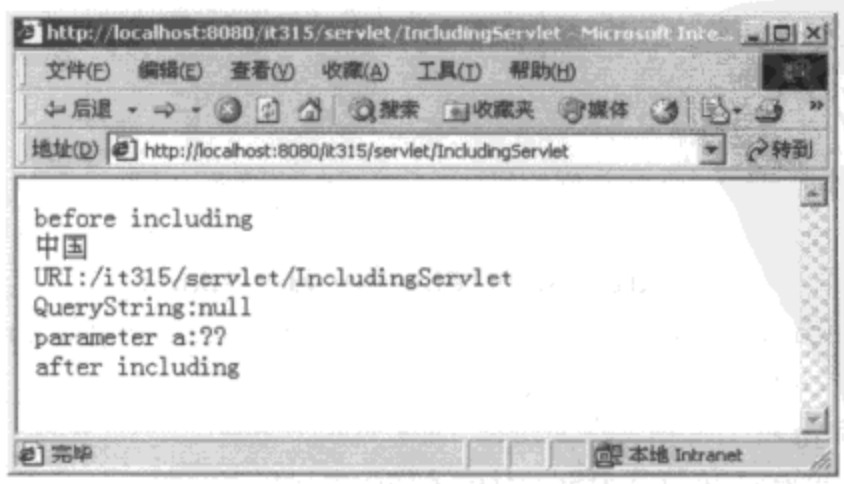


图 5.21

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

从图 5.21 中可以看到，这次被包含的 IncludedServlet 程序自己输出到客户端的中文字符显示正常了。这是因为在调用 RequestDispatcher.include 方法时，Servlet 容器继续保持调用者程序当前已写入到 HttpServletResponse 对象中的信息不变，由于在 IncludingServlet 中已经调用 setContentType 设置了输出的字符集编码为 GB2312，所以，IncludedServlet 根据该设置信息将其中的字符文本按 GB2312 编码输出。

从图 5.21 中还可以看到，在被调用者 IncludedServlet 程序中获得的 URI 仍为调用者 IncludingServlet 的路径，而不是被调用者 IncludedServlet 程序自己的 URI。虽然在调用 IncludedServlet 时向它传递了一个参数，但是浏览器访问 IncludingServlet 时没有传递参数，所以在 IncludedServlet 程序中获得的整个查询参数为 null。尽管从图 5.21 中看到 HttpServletRequest.getParameter 方法获得的参数值部分是乱码，但是，这也说明调用 HttpServletRequest.getParameter 方法可以获得传递给被调用者 IncludedServlet 的参数。关于检索 URL 中的附加参数时所出现的中文乱码问题，将在第 6.9.5 节进行详细讲解。

05 RequestDispatcher 对象中封装的资源也可以是一个静态 HTML 文件，如果这个静态 HTML 文件中包含有中文字符，调用者程序调用 RequestDispatcher.include 方法包含这个静态 HTML 文件时，浏览器会接收到这个静态 HTML 文件中的原始中文内容吗？浏览器中的显示效果与直接访问这个静态 HTML 文件时的显示效果一样吗？修改 IncludingServlet.java 程序，将其调用的目标资源修改成在准备实验环境时所创建的 test.html 文件，如例程 5-14 所示。

例程 5-14 IncludingServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class IncludingServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        //response.setContentType("text/html;charset=gb2312");
        //PrintWriter out = response.getWriter();
        RequestDispatcher rd = request.getRequestDispatcher(
            "/test.html");
        rd.include(request,response);
    }
}
```

重新编译 IncludingServlet.java 源文件，等待 Tomcat 重新装载 IncludingServlet 后，在浏览器窗口中刷新对 IncludingServlet 的访问，返回结果如图 5.22 所示。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

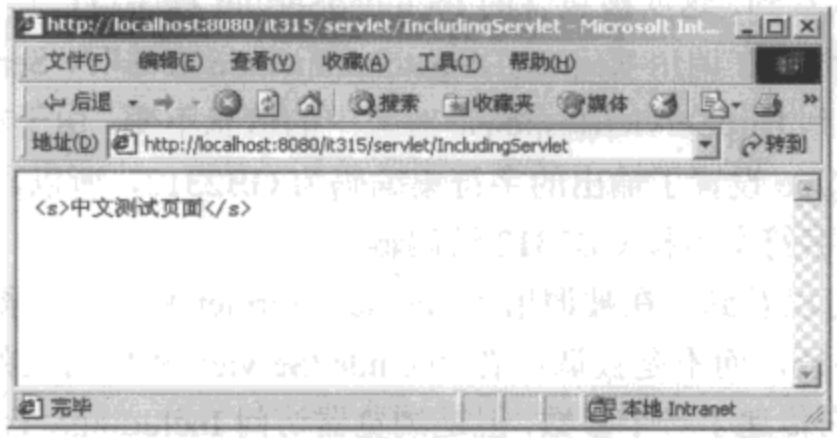


图 5.22

从图 5.22 中可以看到，test.html 文件中的<s>和</s>没有被浏览器当做 HTML 标签来解释处理，这与直接访问 http://localhost:8080/it315/test.html 页面时所看到的效果（如图 5.3 所示）是不同的。图 5.22 中的问题在前面已经学习过，这是由于浏览器接收到的响应消息中没有 Content-Type 头字段造成的，这可以通过使用 telnet 命令访问 IncludingServlet 来验证。

那么为什么直接访问 http://localhost:8080/it315/test.html 页面时所得到的响应消息中包含有 Content-Type 头字段（如图 5.4 所示），而通过 IncludingServlet 包含 test.html 页面时则得不到 Content-Type 头字段了呢？凡是在 Web.xml 文件中找不到匹配<servlet-mapping>元素的 URL 访问请求，也就是所有其他 Servlet 都不处理的访问请求，都将交给 Tomcat 中设置的一个缺省 Servlet 处理，客户端对静态 HTML 文件和图片的访问其实都是由缺省 Servlet 来完成响应的。在一个 Servlet 中包含静态 HTML 文件实际上就是在调用缺省 Servlet，所以，调用静态 HTML 文件与调用另外一个 Servlet 程序在本质上没有什么区别。

在这里，因为作为被调用者的缺省 Servlet 程序不能改变响应消息的状态码和响应头，它里面的 response.setContentType 语句将不再起作用，而调用者程序中又没有调用 response.setContentType 语句来设置响应内容的 MIME，所以，发送给客户端的响应消息中就没有 Content-Type 头字段。

取消 IncludingServlet.java 程序中的 response.setContentType("text/html;charset=gb2312"); 语句前面的注释符，重新编译 IncludingServlet.java 源文件，等待 Tomcat 重新装载 IncludingServlet 后，在浏览器窗口中刷新对 IncludingServlet 的访问，浏览器就可以显示出完全正常的结果了。

06 取消 IncludingServlet.java 程序中的 PrintWriter out = response.getWriter();语句前面的注释符，重新注释掉 response.setContentType("text/html;charset=gb2312");语句，再次编译 IncludingServlet.java 源文件，等待 Tomcat 重新装载 IncludingServlet 后，在浏览器窗口中刷新对 IncludingServlet 的访问，返回结果如图 5.23 所示。

在图 5.23 中看到的内容不再是 test.html 文件中的原始内容，中文都变成了问号（?），这是怎么回事呢？下面分析这种现象的发生原因。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

图 5.23

Tomcat 的缺省 Servlet 首先检查当前 HttpServletResponse 对象是否已经调用了 getWriter 方法返回过 PrintWriter 对象，如果已经调用，则使用 getWriter 方法返回的 PrintWriter 对象来输出静态 HTML 文件中的内容，否则，它将调用 getOutputStream 方法返回的 ServletOutputStream 对象将静态 HTML 文件中的内容按字节流的形式原封不动地写入到客户端。Tomcat 的缺省 Servlet 中与此相关的示意代码如下：

```
ServletOutputStream ostream = null;
PrintWriter writer = null;
......
try
{
    /*
    如果抛出了异常，说明前面已经调用过 getWriter 方法，
    则在异常处理代码中再次调用 getWriter 方法对 writer 变量进行赋值。
    */
    ostream = response.getOutputStream();
}
catch (IllegalStateException e)
{
    /*只有那些文本内容才可以用 PrintWriter 对象进行转换输出*/
    if ( (contentType == null) || (contentType.startsWith("text")) )
    {
    writer = response.getWriter();
    }
    else
    {
        throw e;
    }
}
......
//如果已经对 writer 变量赋值，则执行 else 从句
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
if (ostream != null)
{
    //将资源中的内容按字节流原封不动地输出到客户端
    copy(cacheEntry, renderResult, ostream);
}
else
{
    //将资源中的内容转换成字符文本后再由 PrintWriter 对象转换输出
    copy(cacheEntry, renderResult, writer);
}
```

在这里，由于调用者 IncludingServlet 已经调用过当前 HttpServletResponse 对象的 getWriter 方法，所以，Tomcat 的缺省 Servlet 将使用 getWriter 方法返回的 PrintWriter 对象来输出静态 HTML 文件中的内容，而不是把 test.html 文件中的内容按字节流原封不动地读出来，再按字节流原封不动地写入到 Servlet 引擎的缓冲区中。由于 IncludingServlet 在调用 getWriter 方法之前，没有通过任何方式指定它所获得的 PrintWriter 对象所采用的字符集编码，该 PrintWriter 对象将 test.html 文件中的内容按默认的 ISO8859-1 编码进行转换后输出到客户端，所以其中的中文字符就变成了问号（?）。

07 在步骤 **06** 的基础上，将程序中的 PrintWriter out = response.getWriter();语句移动到 rd.include(request,response);语句的后面，重新编译 IncludingServlet.java 源文件，等待 Tomcat 重新装载 IncludingServlet 后，在浏览器窗口中刷新对 IncludingServlet 的访问，返回结果如图 5.24 所示。

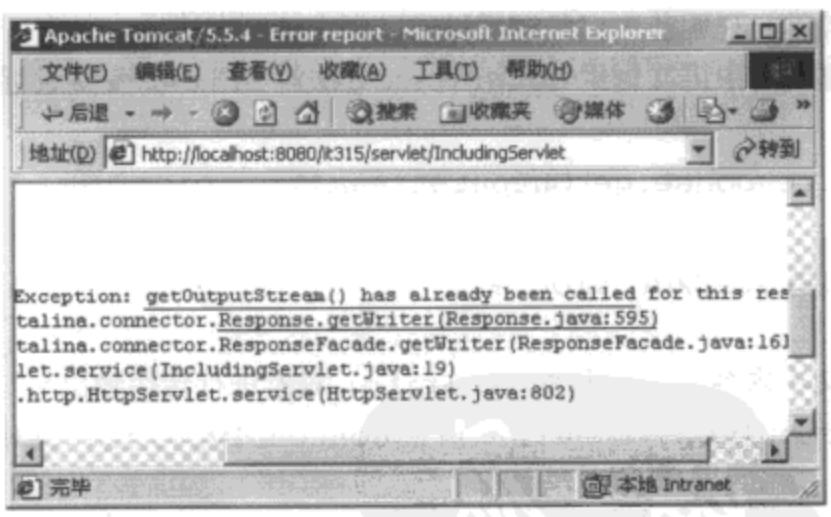


图 5.24

由于调用者 IncludingServlet 在引入 test.html 文件之前没有调用过当前 HttpServletResponse 对象的 getWriter 方法，Tomcat 的缺省 Servlet 将调用 getOutputStream 方法返回 ServletOutputStream 对象来输出静态 HTML 文件中的内容，而在引入 test.html 文件的语句之后，IncludingServlet 又去调用 getWriter 方法，所以就发生了图 5.24 所示的异常。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

5.7.3 用 forward 方法实现请求转发

forward 方法用于将请求转发到 RequestDispatcher 对象封装的资源，Servlet 程序在调用这个方法进行转发之前可以对请求进行一些前期的预处理。在后面要讲解的 MVC 架构中，forward 方法是一个核心方法，控制器组件（Controller）就是使用该方法来跳转到视图组件（Viewer），让视图组件产生响应内容返回给浏览器显示。使用 forward 方法时，应注意下面一些问题：

（1）如果在调用 forward 方法之前，在 Servlet 程序中写入的部分内容已经被真正地传送到了客户端，forward 方法将抛出 IllegalStateException 异常。由于 IllegalStateException 是 RuntimeException 的子类，属于系统级的异常，所以在程序代码中并不强制要求使用 try…catch 语句进行处理。

（2）调用 RequestDispatcher.forward 方法时，Servlet 容器将根据目标资源路径对当前 HttpServletRequest 对象中的请求路径和参数信息进行调整。

（3）如果在调用 forward 方法之前向 Servlet 引擎的缓冲区中写入了内容，只要写入到缓冲区中的内容还没有被真正输出到客户端，forward 方法就可以被正常执行，原来写入到输出缓冲区中的内容将被清空。在调用 forward 方法之后，如果调用者程序继续向 Servlet 引擎的缓冲区中执行写入操作，这些写入操作的执行结果将被忽略。

（4）在调用者程序中设置的响应状态码和响应头不会被忽略，在被调用者程序中设置的响应状态码和响应头也不会被忽略。注意：对于 RequestDispatcher.include 方法调用，只有调用者程序设置的响应状态码和响应头才会有效，在被调用者程序中设置的响应状态码和响应头将被忽略。

（5）如果调用者与被调用者的访问 URL 不属于同一个目录，当被调用者输出的内容中包含使用相对 URL 的访问路径时，原来相对被调用者的 URL 将变成相对于调用者的 URL。这是因为浏览器只知道当前访问的是调用者程序的 URL，所以对响应内容中的相对路径都以调用者程序的访问 URL 作为基准进行计算，这样就导致被调用者程序输出的相对路径不再是相对于被调用者的路径了。

RequestDispatcher.forward 方法的应用细节

01 编写一个名为 ForwardTestServlet 的 Servlet 程序，在它里面调用 RequestDispatcher.include 方法将请求转发给在准备实验环境时创建的 test.html 文件，如例程 5-15 所示。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

例程 5-15 ForwardTestServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ForwardTestServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        //禁止浏览器缓存，以免影响实验效果
        response.setDateHeader("Expires",0);
        response.setHeader("Cache-Control","no-cache");
        response.setHeader("Pragma","no-cache");

        RequestDispatcher rd =
            getServletContext().getRequestDispatcher("/test.html");
        rd.forward(request,response);
    }
}
```

编译 ForwardTestServlet.java 源文件，确保编译后生成的 class 文件存放在了<tomcat 的安装目录>\Webapps\it315\Web-INF\classes 目录中。

02 在<tomcat 的安装目录>\Webapps\it315\Web-INF\Web.xml 文件中注册这个 Servlet，并设置其映射 URL。在 Web.xml 文件中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>ForwardTestServlet</servlet-name>
    <servlet-class>ForwardTestServlet</servlet-class>
</servlet>
……
<servlet-mapping>
    <servlet-name>ForwardTestServlet</servlet-name>
    <url-pattern>/servlet/ForwardTestServlet</url-pattern>
</servlet-mapping>
```

保存 Web.xml 文件后，重新启动 Tomcat。

03 设置浏览器显示网页文档所采用的字符集为 GB2312 后，在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/servlet/ForwardTestServlet
```

浏览器中显示出了 test.html 文件中的内容，并且与直接访问 http://localhost:8080/it315/

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

test.html 页面时所看到的效果一样，只是浏览器地址栏保持为初始的 URL 地址不变，即还是显示 ForwardTestServlet 的 URL 路径。

04 使用 telnet 命令连接上 Tomcat Web 服务器后，在 Telnet 程序命令窗口中，输入如下内容：

```
GET /it315/servlet/ForwardTestServlet HTTP/1.1<回车>
Host:<空格><回车>
<回车>
```

Telnet 窗口中显示出的结果如图 5.25 所示。

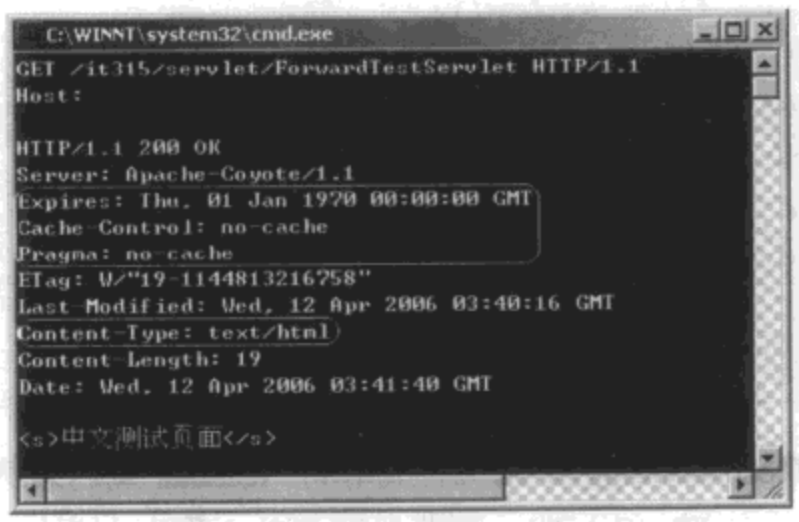


图 5.25

从图 5.25 中可以看到如下两个重要信息：

- 响应消息包含在调用者程序（ForwardTestServlet）中设置的响应头，例如那些禁止浏览器缓存的响应头字段。
- 响应消息中也包含被调用者程序设置的响应头，例如 Content-Type 头字段。当 forward 方法将请求转发给 test.html 页面时，实际上在调用 Tomcat 的缺省 Servlet，图 5.25 中的 Content-Type 头字段就是由 Tomcat 的缺省 Servlet 设置的。

05 按例程 5-16 修改 ForwardTestServlet.java 程序，在其中加入一些输出响应正文的程序代码。

例程 5-16 ForwardTestServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ForwardTestServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        response.setContentType("text/html;charset=gb2312");

        //禁止浏览器缓存，以免影响实验效果
        response.setDateHeader("Expires",0);
        response.setHeader("Cache-Control","no-cache");
        response.setHeader("Pragma","no-cache");

        PrintWriter out = response.getWriter();
        out.println("before forward");
        //用于将缓冲区中的内容强制刷新到客户端
        //response.flushBuffer();
        RequestDispatcher rd =
            getServletContext().getRequestDispatcher("/test.html");
        rd.forward(request,response);
        out.println("after forwarding");
        System.out.println("after forwarding");
    }
}
```

重新编译 ForwardTestServlet.java 源文件并等待 Tomcat 重新装载后，再次用浏览器和 Telnet 程序分别访问 ForwardTestServlet，浏览器和 Telnet 程序中都仅仅是显示出了 test.html 页面中的内容，而没有显示出 ForwardTestServlet 程序向 Servlet 引擎的缓冲区中写入的内容。

这里请读者思考一个问题：在 ForwardTestServlet 程序的 service 方法中调用 forward 方法将请求转发给另外一个资源后，service 方法中的位于 forward 方法调用语句后面的程序代码是否还会被执行呢？不管 forward 方法完成的功能是什么，调用 forward 方法的语句毕竟只是 Java 程序代码中的一个方法调用语句，它不可能违背 Java 语言中的方法调用的基本执行规则，当 forward 方法返回后，service 方法将按照通常的流程继续执行其后程序代码。只要查看 Tomcat 所运行的命令行窗口，就可以看到最后那条语句执行后的打印结果，这说明 ForwardTestServlet 程序中的调用 fowrard 方法的语句与调用普通方法的语句的执行流程是完全一样的，位于这个语句后面的程序代码仍然会被执行，只是向 Servlet 引擎的缓冲区中进行写入的那些语句的执行结果将被忽略。

06 注释掉例程5-16中的`response.setContentType("text/html;charset=gb2312");`语句，重新编译 ForwardTestServlet.java 源文件并等待 Tomcat 重新装载后，用浏览器再次访问 ForwardTestServlet，这时候在浏览器中看到的 test.html 文件中的中文内容都变成了问号（?），如图 5.26 所示。

这个问题的原因在前面讲解 RequestDispatcher.include 方法时已经分析过，由于 Tomcat 的缺省 Servlet 首先检查当前 HttpServletResponse 对象是否已经调用过 getWriter 方法返回了 PrintWriter 对象，如果已经调用，则使用 getWriter 方法返回的 PrintWriter 对象输出静态 HTML 文件中的内容，由于在得到该 PrintWriter 对象之前没有指定其采用的字符集编码，

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

所以该 PrintWriter 对象将静态 HTML 文件中的内容按默认的 ISO8859-1 编码进行转换后输出到客户端。

图 5.26

在步骤 **05** 中没有注释掉 response.setContentType("text/html;charset=gb2312");语句的情况下，浏览器能够正常显示出 test.html 文件中的中文内容，这是由于在请求转发之前已经调用过了当前 HttpServletResponse 对象的 getWriter 方法，并且在这之前已经使用 HttpServletResponse.setContentType 方法指定了 getWriter 方法返回的 PrintWriter 对象所采用的字符集编码为 GB2312，Tomcat 的缺省 Servlet 将使用该 PrintWriter 对象把静态 HTML 文件中的内容按 GB2312 编码进行转换后再输出到客户端。如果将 setContentType 方法中设置的输出字符集编码由 GB2312 修改成 UTF-8，如下所示：

```
response.setContentType("text/html;charset=UTF-8");
```

这时候浏览器中还能正常显示出 test.html 文件中的中文内容吗？因为中文字符可以采用 GB2312 编码表示，也可采用 UTF-8 编码表示，只要浏览器支持该字符集编码且输出到浏览器的数据确实是中文字符的该字符集编码，浏览器就能正常显示出该中文字符。Tomcat 的缺省 Servlet 可以将静态 HTML 文件中的内容按 GB2312 编码进行转换后输出到客户端，当然也可以按照 UTF-8 编码进行转换后输出到客户端。

07 取消 ForwardTestServlet 程序中强制刷新缓冲区的语句前面的注释符，重新编译并等待 ForwardTestServlet 被重新加载后，刷新浏览器刚才的访问请求，可以看到浏览器中显示出了 forward 语句前面写入的内容，而没有显示出 test.html 文件中的内容。查看 Tomcat 所运行的命令行窗口，看不到最后那条语句执行后的打印结果，而是看到了 forward 方法抛出异常导致 service 方法提前结束的相关信息。异常信息的中一些关键内容如下所示：

```
严重: Servlet.service() for servlet ForwardTestServlet threw exception
java.lang.IllegalStateException: Cannot forward after response has been
committed
    at ForwardTestServlet.service(ForwardTestServlet.java:23)
```

08 取消 response.setContentType("text/html;charset=gb2312");语句前的注释和重新注释掉强制刷新缓冲区的语句，重新编译 ForwardTestServlet.java 程序。接着用记事本程序

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

将 test.html 文件本身另存为 UTF-8 编码格式，然后再访问 ForwardTestServlet，这时候浏览器显示出的内容为乱码。前面讲过，Tomcat 的缺省 Servlet 可以将静态 HTML 文件中的内容按照某种字符集编码进行转换后输出到客户端，这是因为 PrintWriter 对象的输出方法接受的参数类型都是 Java 字符串，这些 Java 字符串在内存中都是以 Unicode 编码的形式存在的，所以 PrintWriter 对象在输出这些内容时要进行字符编码转换。现在出现的问题在于：这些要输出的 Java 字符串又是从静态 HTML 文件中读取到的，这个读取过程只有知道静态 HTML 文件中的内容的字符集编码，才能将它们转换成正确的 Unicode 编码的 Java 字符串，而从 HTML 文件本身的内容上无法准确知道其所采用的字符集编码，所以，Tomcat 的缺省 Servlet 只能去假定和猜测静态 HTML 文件的字符集编码。由于人们一般都按操作系统本地字符集编码来编写自己的文件，所以，Tomcat 的缺省 Servlet 假定所有静态 HTML 文件的字符集编码为操作系统本地字符集编码，对中文系统来说，即 GB2312。在通常情况下，进行这种假定都不会出什么问题，但是，一旦静态 HTML 文件的字符集编码与本地操作系统的不一致时，就会出现问题。这里的 test.html 文件本身的字符集编码为 UTF-8，但 Tomcat 的缺省 Servlet 将它假定成了 GB2312 字符集编码进行读取，所以就出现了乱码问题。

查看 Tomcat 的缺省 Servlet 的源文件或者<tomcat 的安装目录>\conf\Web.xml 文件中的关于缺省 Servlet 的注释说明，可以看到其中有一个名为 fileEncoding 初始化参数用于指定缺省 Servlet 所读入的静态 HTML 文件的字符集编码，如果设置了这个参数，Tomcat 的缺省 Servlet 对所有的文件都按照该指定的字符集编码来处理。

09 修改 ForwardTestServlet 程序，注释掉其中的 PrintWriter out = response.getWriter(); 语句以及所有产生输出的相关语句，重新编译并等待 ForwardTestServlet 被重新加载后，刷新浏览器刚才的访问请求，这时浏览器中就正常显示出了采用 UTF-8 编码的 test.html 文件的内容。这是因为 Tomcat 的缺省 Servlet 检查到对当前 HttpServletResponse 对象还没有调用过 getWriter 方法，它将调用 getOutputStream 方法返回的 ServletOutputStream 对象将静态 HTML 文件中的内容按字节流的形式原封不动地写入到客户端，所以不会发生字符编码转换的问题。

10 修改 ForwardTestServlet 程序，取消步骤**09**中对 PrintWriter out = response.getWriter();语句以及所有产生输出的相关语句所加上的注释，然后将请求转发给的目标资源修改成一个图片文件。假设<tomcat 的安装目录>\Webapps\it315 目录中存在一个名为 logo.gif 的图片文件，那么，应将 ForwardTestServlet 程序中的用于获得 RequestDispatcher 对象的语句修改成如下形式：

```
RequestDispatcher rd =
        getServletContext().getRequestDispatcher("/logo.gif");
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

重新编译并等待 ForwardTestServlet 被重新加载后，刷新浏览器刚才的访问请求，浏览器中显示出如图 5.27 所示的异常信息。

图 5.27

图 5.27 中的异常信息说明：对于图片等非文本格式的资源文件，Tomcat 的缺省 Servlet 总是调用 HttpServletResponse.getOutputStream 方法返回 ServletOutputStream 对象来将它们按字节流的形式原封不动地写入到客户端。由于当前的 HttpServletResponse 对象在 ForwardTestServlet 程序中已经调用过了 getWriter 方法，Tomcat 的缺省 Servlet 再调用 HttpServletResponse 对象的 getOutputStream 方法，就发生了如图 5.27 所示的异常。

11 修改 ForwardTestServlet 程序，注释掉其中的 PrintWriter out = response.getWriter(); 语句以及所有产生输出的相关语句，重新编译并等待 ForwardTestServlet 被重新加载后，刷新浏览器刚才的访问请求，浏览器显示出了正常的图片。

RequestDispatcher.forward 方法的相对路径问题

01 参照 5.7.2 节的两个例子程序，按例程 5-17 和例程 5-18 编写两个 Servlet 程序：ForwardingServlet 和 ForwardedServlet，在 ForwardingServlet 中通过 RequestDispatcher.foward 方法调用 ForwardedServlet。

例程 5-17 ForwardingServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ForwardingServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
    IOException
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
    {
        response.setContentType("text/html;charset=UTF-8");
        //PrintWriter out = response.getWriter();

        response.setDateHeader("Expires",0);
        response.setHeader("Cache-Control","no-cache");
        response.setHeader("Pragma","no-cache");

        String china = "中国";
        RequestDispatcher rd = getServletContext().getRequestDispatcher(
            "/ForwardedServlet?p1=" +
            java.net.URLEncoder.encode(china,"GB2312"));
        rd.forward(request,response);
    }
}
```

例程 5-18　ForwardedServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ForwardedServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        response.setContentType("text/html;charset=GB2312");
        PrintWriter out = response.getWriter();

        /*
        String basePath = request.getScheme() + "://"+ request.
         getServerName()
            + ":" + request.getServerPort() + request.getRequestURI();
        out.println("<base href=\"" + basePath + "\">");
        */

        out.println("<a href='ForwardedServlet'>访问自己</a><br>");
        out.println("URI:" + request.getRequestURI() + "<br>");
        out.println("QueryString:" + request.getQueryString() + "<br>");
        out.println("URL:" + request.getRequestURL() + "<br>");
        String p1 = request.getParameter("p1");
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        String chP1 = null;
        if(p1 != null)
        {
            chP1 = new String(p1.getBytes("iso8859-1"),"GB2312");
        }
        out.println("parameter p1:" + chP1 + "<br>");
    }
}
```

编译这两个 Servlet 源文件，确保编译后生成的 class 文件存放在了<tomcat 的安装目录>\Webapps\it315\Web-INF\classes 目录中。

02 在<tomcat 的安装目录>\Webapps\it315\Web-INF\Web.xml 文件中注册这两个 Servlet，并设置其映射 URL。在 Web.xml 文件中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>ForwardingServlet</servlet-name>
    <servlet-class>ForwarddingServlet</servlet-class>
</servlet>
<servlet>
    <servlet-name>ForwarddedServlet</servlet-name>
    <servlet-class>ForwardedServlet</servlet-class>
</servlet>

……
<servlet-mapping>
    <servlet-name>ForwardingServlet</servlet-name>
    <url-pattern>/servlet/ForwardingServlet</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>ForwardedServlet</servlet-name>
    <url-pattern>/ForwardedServlet</url-pattern>
</servlet-mapping>
```

注意 ForwardingServlet 和 ForwardedServlet 被映射到了不同的目录路径下面。保存 Web.xml 文件后，重新启动 Tomcat。

03 先用浏览器直接访问 ForwardedServlet，看看其运行效果。在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/ForwardedServlet
```

浏览器返回的结果如图 5.28 所示。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

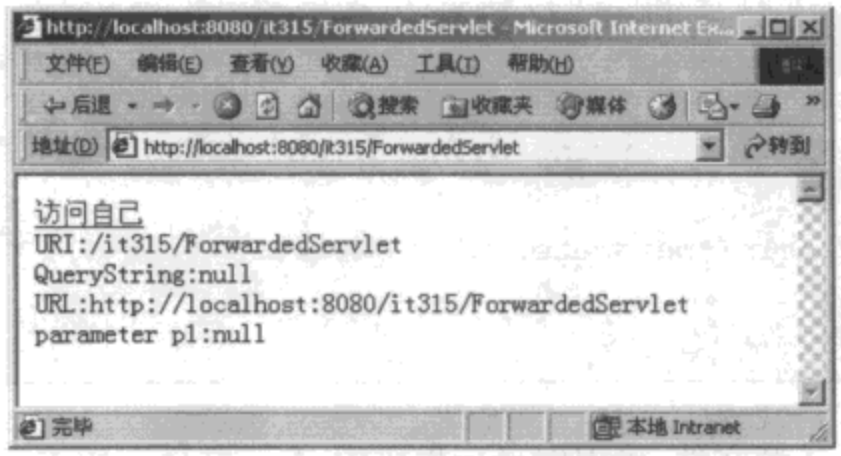


图 5.28

单击其中的“访问自己”超链接，证明这个链接确实有效无误。

04 用浏览器访问 ForwardingServlet，看看其转发请求的运行效果。在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/servlet/ForwardingServlet
```

浏览器返回的结果如图 5.29 所示。

图 5.29

从图 5.29 中可以看到，在被调用者 ForwardedServlet 程序中获得的 URI 为被调用者自己的 URI 路径，并且可以获得传递给它的整个查询参数。不过，有一点值得注意，在 ForwardedServlet 程序中调用 HttpServletRequest.getRequestURL 方法返回的路径仍然为调用者 ForwardingServlet 程序的路径。

单击图 5.29 中的浏览器的“查看”→“编码”菜单，可以看到当前的字符集编码为“简体中文(GB2312)”，这是被调用者程序 ForwardedServlet 中通过 setContentType 方法设置的字符集编码，这说明在这种情况下，被调用者程序中的设置的字符集编码将覆盖调用者程序中设置的字符集编码。

05 取消 ForwardingServlet.java 程序中的 PrintWriter out = response.getWriter();语句的注释，重新编译 ForwardingServlet.java 程序并等到 Tomcat 重新加载后，刷新图 5.29 中的浏览器的访问请求。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

然后单击浏览器的“查看”→“编码”菜单，可以看到当前的字符集编码为“Unicode (UTF-8)”，这是调用者程序 ForwardingServlet 中通过 setContentType 方法所设置的字符集编码，这说明如果调用者程序中已经调用了当前 HttpServletResponse 对象的 getWriter 方法，以后在被调用者程序中的设置的字符集编码将不再对 PrintWriter 对象起作用。这是因为只有在第一次调用当前 HttpServletResponse 对象的 getWriter 方法之前，调用那些设置字符集编码的方法才会对 PrintWriter 对象起作用，而在 getWriter 方法之后调用这些方法都将对 PrintWriter 对象无效。

06 单击图 5.29 的浏览器中“访问自己”超链接，浏览器将返回找不到“这个超链接所指向的资源”的提示信息。这是因为“访问自己”超链接使用的是相对 URL，并且原来是相对于被调用者 ForwardedServlet 而言的，虽然调用者 ForwardingServlet 程序内部调用了 ForwardedServlet 程序进行响应，但是，对浏览器来说，它只知道当前访问的是调用者 ForwardingServlet 程序的 URL，所以对响应内容中的相对路径都以调用者 ForwardingServlet 程序的访问 URL 作为基准进行计算，这样，就导致被调用者程序输出的相对路径都不再是相对于被调用者自己的路径了，在原来单独访问被调用者程序时的响应内容中有效的相对路径此时就不再有效了。由于这在服务器端的 Web 程序开发中是一个很常见的问题，HTML 规范专门为此提供了一个<base>标签来作为解决方案。<base>标签用于指定网页中的相对超链接的基准地址，详细的细节请参看本书作者的《JavaScript 网页开发》一书。

取消 ForwardedServlet 程序中被注释掉的那段语句的注释，重新编译 ForwardedServlet.java 程序并等到 Tomcat 重新加载后，重新访问 ForwardingServlet，浏览器显示的结果与图 5.29 完全一样。但是，这次再单击返回的结果页面中的“访问自己”超链接，这个超链接又变得有效无误了。查看浏览器中显示的网页文档的源文件，可以看到如下一行代码：

```
<base href="http://localhost:8080/it315/ForwardedServlet">
```

注意 这条<base>标签语句指定网页中相对路径的基准地址为“http://localhost:8080/it315/”，而不是“http://localhost:8080/it315/ForwardedServlet”，因为如果在<base>标签的 href 属性设置的 url 最后不是以路径分隔符“/”结尾的话，那么整个 url 最后部分的组件将被看做一个文件资源名，而不是目录名，这时的基准地址就是最后部分的组件的上级组件。

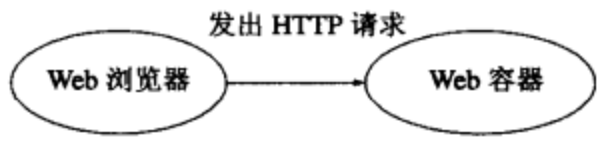
5.7.4 请求转发的运行流程

为了便于读者更好地理解使用 forward 方法实现的请求转发的工作机理，下面画出了请求转发的整个过程的示意图。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

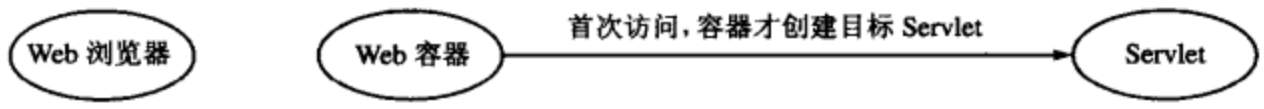
http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

步骤 1：



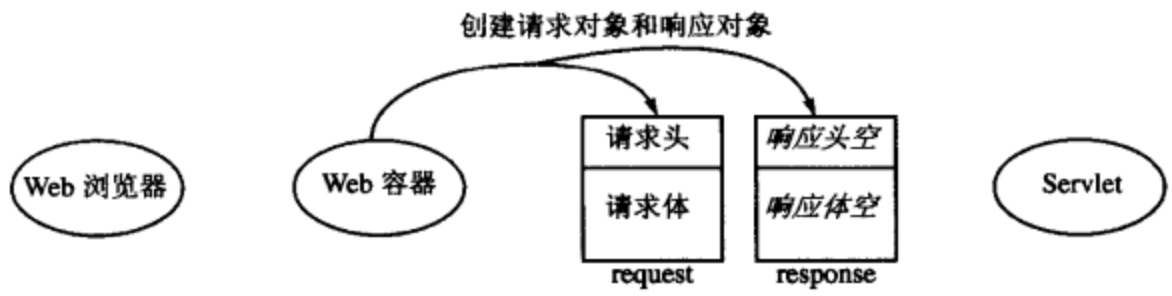
浏览器根据用户的请求生成 HTTP 请求消息，并将其发送给 Web 服务器（Web 容器）。

步骤 2：



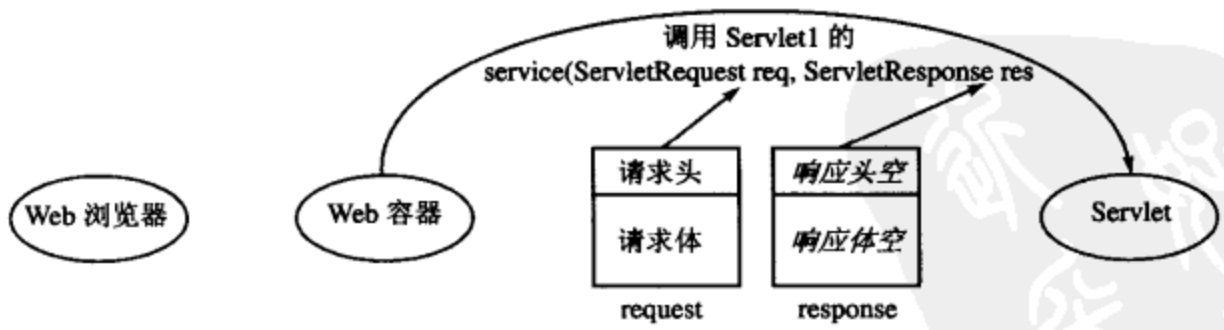
如果请求资源为 Servlet，Web 服务器（Web 容器）检测内存中是否存在该 Servlet 对象，如果不存在，则加载和创建该 Servlet 对象。

步骤 3：



Web 服务器（Web 容器）创建针对该次访问的请求对象（request）和响应对象（response）。请求对象中包含了 HTTP 的请求信息，响应对象用于封装将要发送的 HTTP 响应信息，此时的响应对象中的内容为空。

步骤 4：

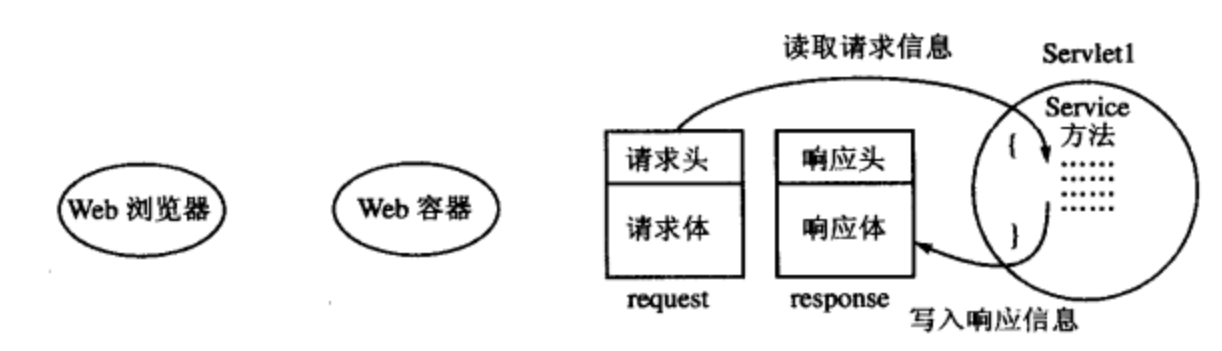


Web 容器调用 Servlet1 的 service（ServletRequest request，ServletResponse response）方法，并将请求对象和响应对象传递给该方法。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

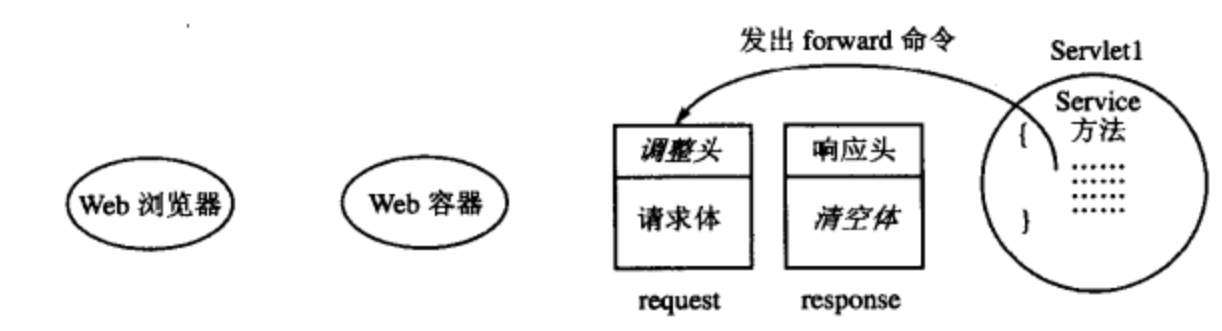
http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

步骤 5：



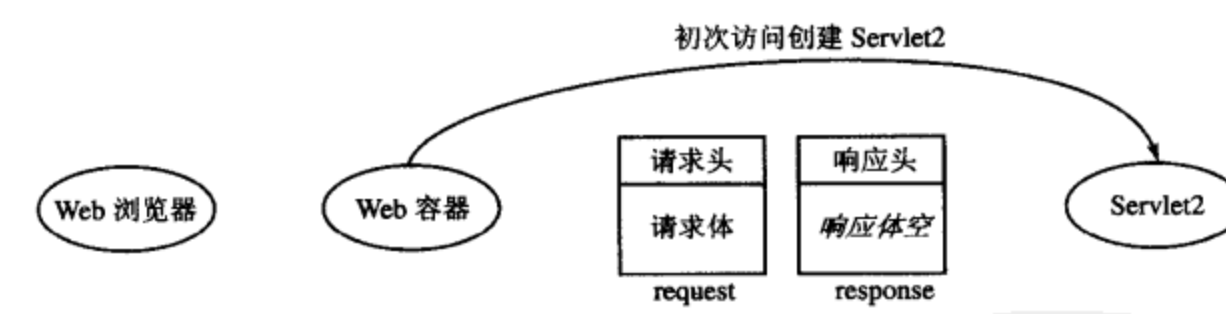
Servlet1 对象的 service 方法从请求对象中读取请求信息，这时也可以在响应对象中写入部分响应头和响应体数据。

步骤 6：



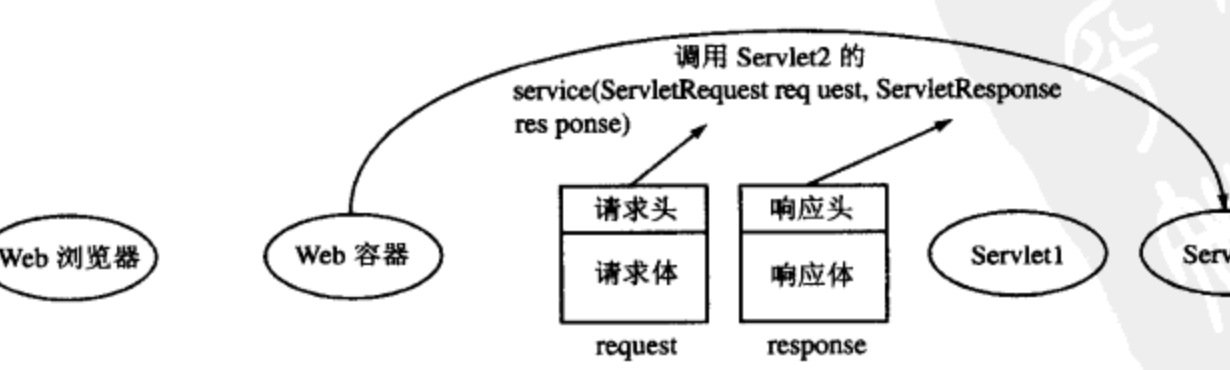
service 方法向 Web 容器发出 forward 命令表示要跳转到 Servlet2，Web 容器清空先前所写入的响应体内容，并调整请求头中的信息。

步骤 7：



Web 服务器（Web 容器）根据 forward 命令调用 Servlet2 对象，如果内存中不存在 Servlet2 对象，则加载和创建 Servlet2 对象。

步骤 8：

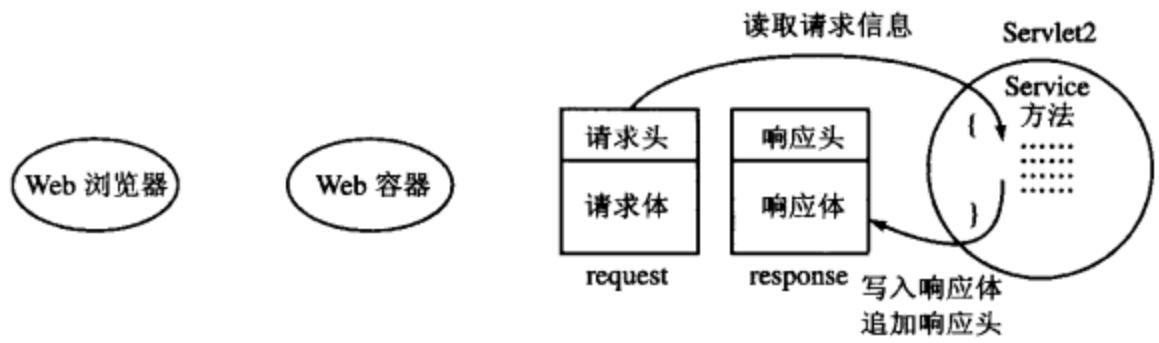


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

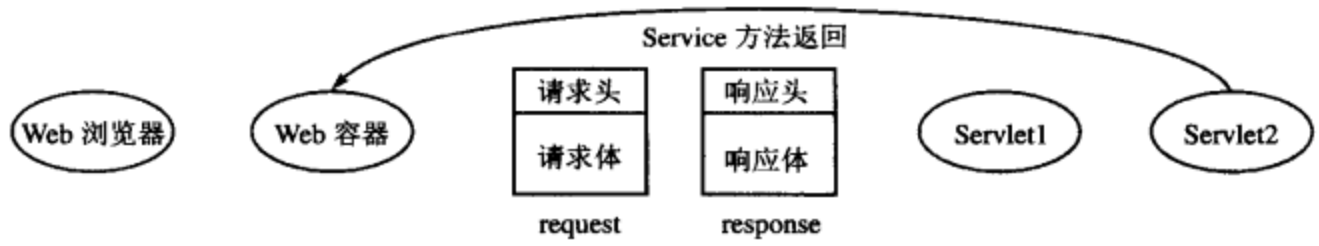
Web 容器调用 Servlet2 的 service（ServletRequest request，ServletResponse response）方法，并将请求对象和响应对象传递给该方法。注意，这里传递的还是调用 Servlet1 时所创建的 request 和 response 对象。

步骤 9：



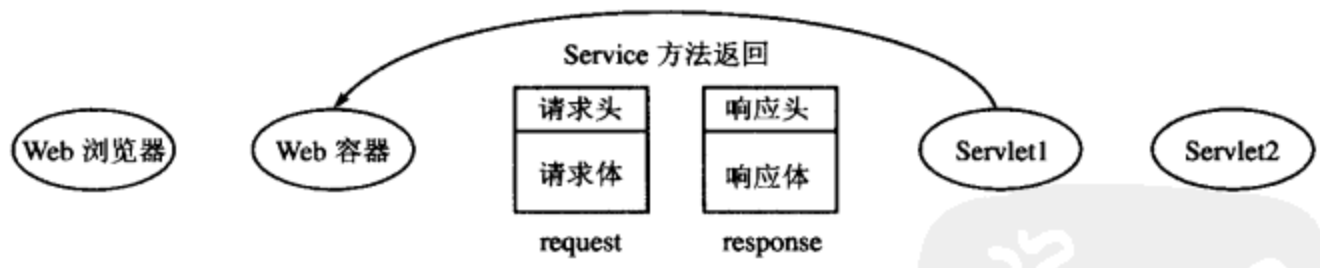
Servlet2 对象的 service 方法从请求对象中读取请求信息，并向响应对象中写入响应头和响应体信息。

步骤 10：



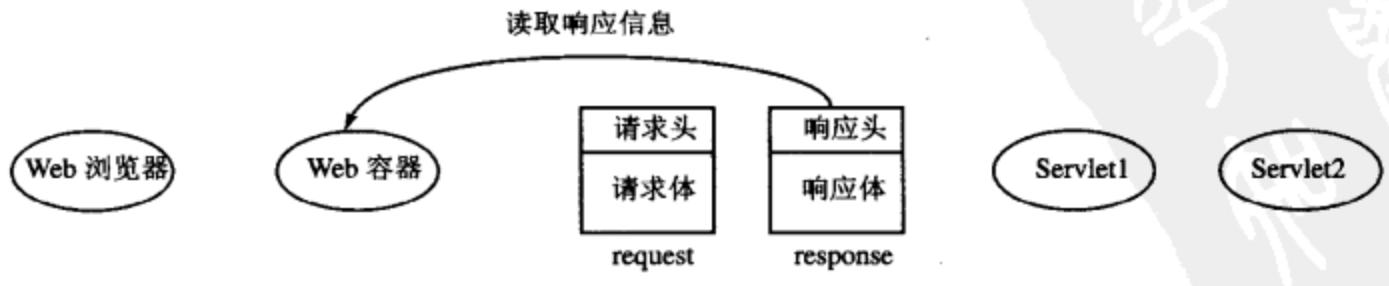
Servlet2 的 service 方法执行完毕后返回。

步骤 11：



Servlet1 的 service 方法继续执行调用 forward 方法的语句后面的代码，执行完毕后返回。

步骤 12：

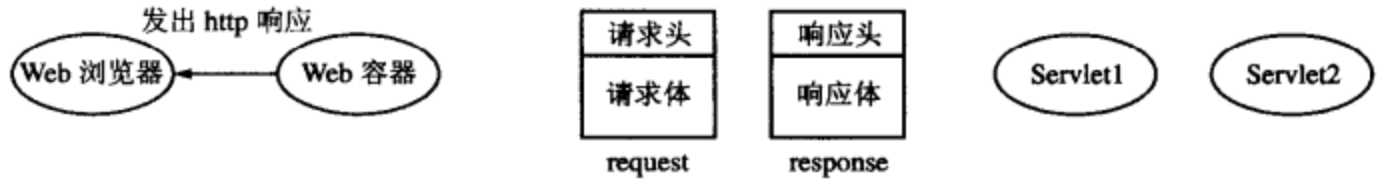


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

Web 服务器（Web 容器）从响应对象中读取响应信息。

步骤 13：



Web 服务器（Web 容器）将响应信息发送给浏览器处理和显示，一次请求响应过程完全结束，request 和 response 变成垃圾，等待垃圾收集器将其彻底从内存中清除。

5.7.5 用 sendRedirect 方法实现请求重定向

sendRedirect 方法用于生成 302 响应码和 Location 响应头，从而通知客户端去重新访问 Location 响应头中指定的 URL，其完整的定义语法如下：

```
public void sendRedirect(java.lang.String location) throws java.io. IOException
```

其中的 location 参数指定了重定向的 URL，它可以使用相对 URL，Servlet 引擎会自动将相对 URL 翻译成绝对 URL 后，再生成 Location 头字段。sendRedirect 方法与 sendError 方法一样都声明可能抛出 IOException 异常，而 setStatus 方法的声明不抛出任何异常。

使用下面两条语句也能完成 response.sendRedirect(url)语句所完成的功能：

```
response.setStatus(response.SC_MOVED_TEMPORARILY );
response.setHeader ("Location", url);
```

但是，response.setHeader ("Location", url)语句中的 url 参数必须是一个绝对的 URL。另外，sendRedirect 方法还会自动创建包含重定向 URL 的超链接的文本正文，该文本正文显示在不支持自动重定向的旧浏览器中，以便用户可以手工进入该 URL 所指向的页面。

sendRedirect 方法不仅可以重定向到当前应用程序中的其他资源，它还可以重定向到同一个站点上的其他应用程序中的资源，甚至是使用绝对 URL 重定向到其他站点的资源，而 RequestDispatcher.forward 方法只能在同一个 Web 应用程序内的资源之间转发请求。如果传递给 sendRedirect 方法的相对 URL 不是以“/”开头，则表示是相对于当前请求的 URL；如果该相对 URL 是以“/”开头，则是相对于整个 Web 站点的根目录，而不是相对于当前 Web 应用程序的根目录，例如，下面的语句表示重定向到当前站点中的 examples 应用程序根目录下的 index.html 页面：

```
response.sendRedirect("/examples/index.html");
```

为了在客户端不支持或关闭 Cookie 的情况下，当 sendRedirect 方法将请求重定向到当前 Web 应用程序中另外的一个资源时仍然能进行会话跟踪，需要调用 HttpServletResponse.encodeRedirectURL 方法对传递给 sendRedirect 方法的 URL 进行 URL 重写，详细细节请参看 7.4.8 节。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

sendRedirect 方法的应用

01 按例程 5-19 编写一个名为 RedirectServlet 的 Servlet 程序，在 service 方法中调用 HttpServletResponse.sendRedirect 方法让浏览器重定向到在准备实验环境时创建的 test.html 文件。

例程 5-19 RedirectServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class RedirectServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        PrintWriter out = response.getWriter();
        out.println("before sendRedirect");
        response.sendRedirect(request.getContextPath() + "/test.html");
        out.println("after sendRedirect");
        System.out.println("after sendRedirect");
    }
}
```

编译 RedirectServlet.java 源文件，确保编译后的结果保存到了<tomcat 的安装目录>\Webapps\it315\Web-INF\classes 目录中。

02 修改<tomcat 的安装目录>\Webapps\it315\Web-INF\Web.xml 文件，在其中的相应位置处增加如下两段内容：

```
<Web-app>
    <servlet>
        <servlet-name>RedirectServlet</servlet-name>
        <servlet-class>RedirectServlet</servlet-class>
    </servlet>
    ……
    ……
    <servlet-mapping>
        <servlet-name>RedirectServlet</servlet-name>
        <url-pattern>/servlet/RedirectServlet</url-pattern>
    </servlet-mapping>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
</Web-app>
```

保存 Web.xml 文件后，重新启动 Tomcat。

03 在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/servlet/RedirectServlet
```

浏览器中显示出了 test.html 网页文档中的内容，注意浏览器地址栏中的地址也变成了 test.html 文件的 URL。

使用 telnet 命令连接上 Tomcat Web 服务器后，在 Telnet 程序命令窗口中，输入如下如下两行内容：

```
GET /it315/servlet/RedirectServlet HTTP/1.1
Host:
```

接着用回车键输入一个空行后，Telnet 程序窗口中显示出 Web 服务器返回的响应信息，如图 5.30 所示。

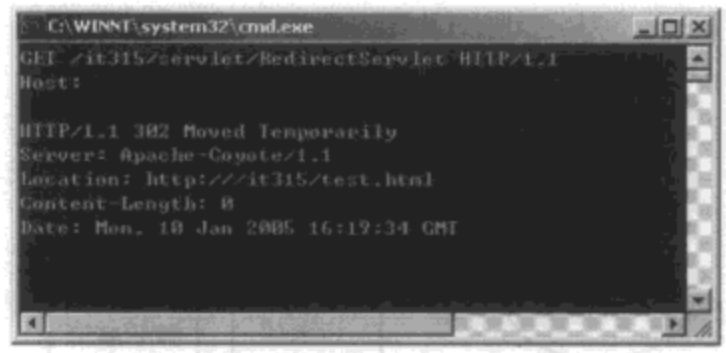


图 5.30

从图 5.30 中可以清楚地看到，HttpServletResponse.sendRedirect 方法只是生成 302 响应状态码和 Location 头字段，以此来告诉客户端去访问另外一个资源，客户端要想获得另外那个资源，必须接着发出对该资源的访问请求。

从图 5.30 中还可以看到，RedirectServlet 自身向客户端写入的内容并没有被传送给客户端，这是因为即使在调用 sendRedirect 方法之前向 Servlet 引擎的缓冲区中写入了内容，只要写入到输出缓冲区中的内容还没有被真正传送到客户端，sendRedirect 方法就可以被正常执行，并且将输出缓冲区中原来写入的内容清空。查看 Tomcat 所运行的命令行窗口，可以看到最后那条语句执行后的打印结果，这说明调用 sendRedirect 方法的语句与调用普通方法的语句的执行流程是完全一样的，位于这个语句后面的程序代码仍然会被执行，只是向 Servlet 引擎的缓冲区中进行写入的那些语句的执行结果将被忽略。

5.7.6 请求重定向的运行流程

为了便于读者更好地理解使用 sendRedirect 方法实现的请求重定向的工作机理，下面画出了请求重定向的整个过程的示意图。

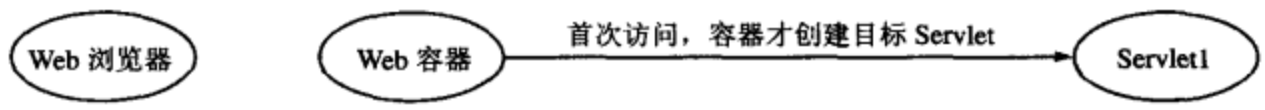
http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

步骤 1：

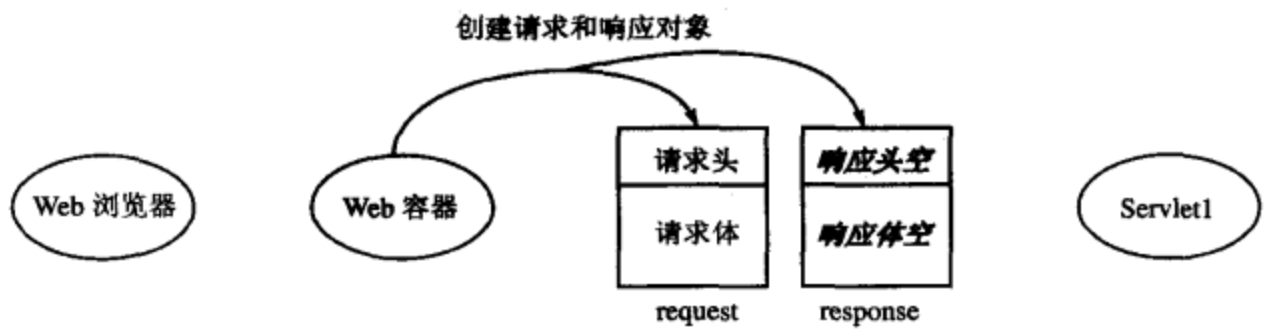
浏览器根据用户的请求生成 HTTP 请求消息，并将其发送给 Web 服务器（Web 容器）。

步骤 2：



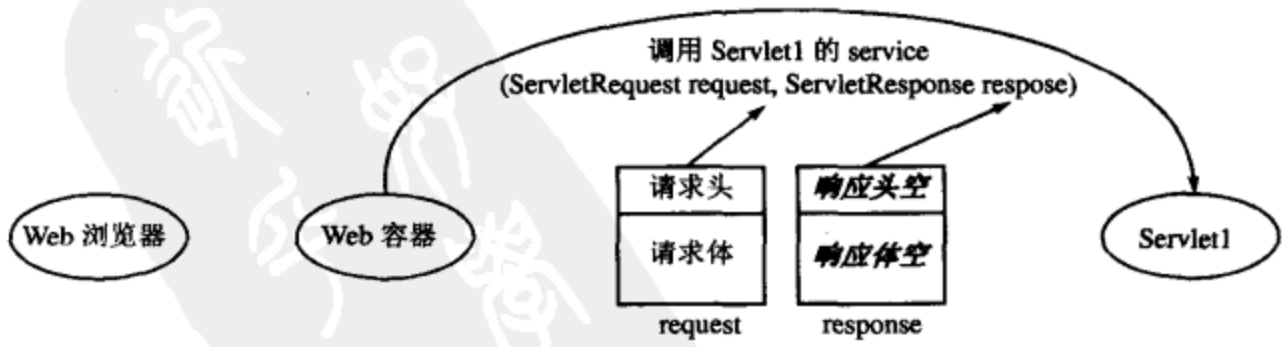
如果请求资源为 Servlet，Web 服务器（Web 容器）检查内存中是否存在该 Servlet 对象，如果不存在，则加载和创建该 Servlet 对象。

步骤 3：



Web 服务器（Web 容器）创建针对该次访问的请求对象（request）和响应对象（response）。请求对象中包含了 HTTP 的请求信息，响应对象用于封装将要发送的 HTTP 响应信息，此时的响应对象中的内容为空。

步骤 4：

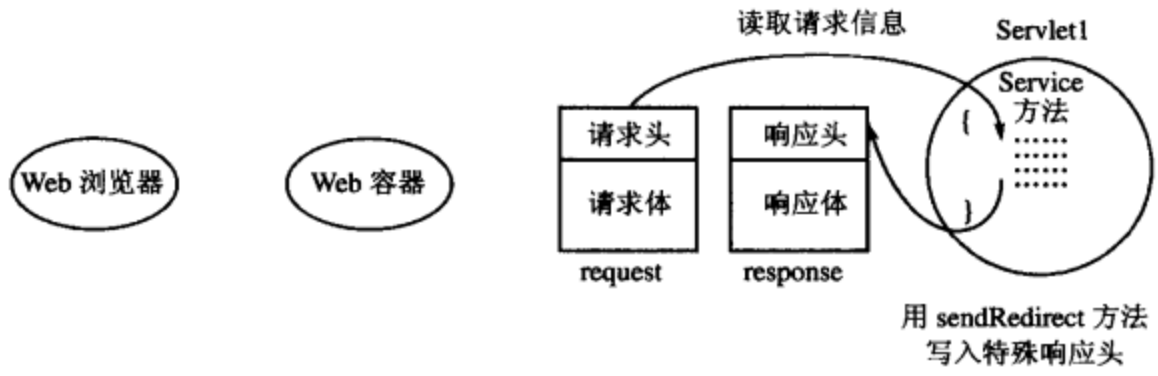


Web 容器调用 Servlet1 的 service（ServletRequest request，ServletResponse response）方法，并将请求对象和响应对象传递给该方法。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

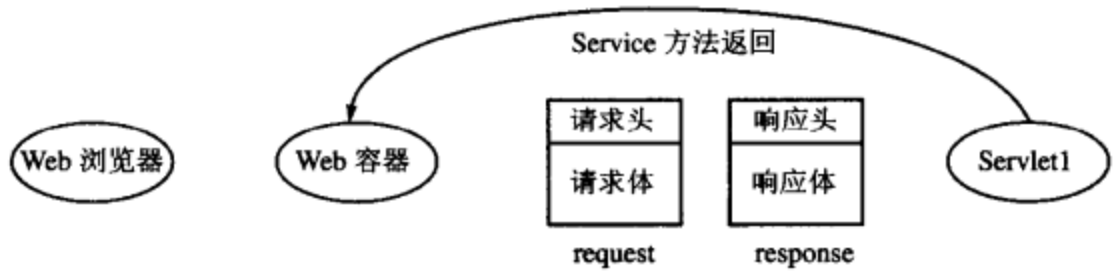
http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

步骤 5：



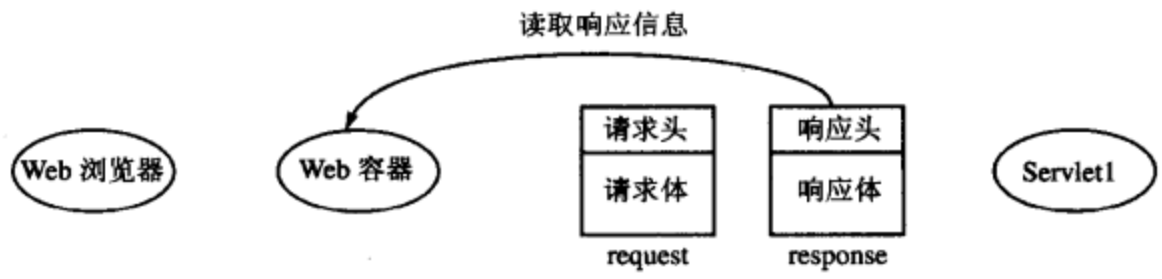
Servlet1 对象的 service 方法从请求对象中读取请求信息，并将包含重定向的响应头信息写入到响应对象中。

步骤 6：



service 方法执行完毕后返回。

步骤 7：



Web 服务器（Web 容器）从响应对象中读取响应信息。

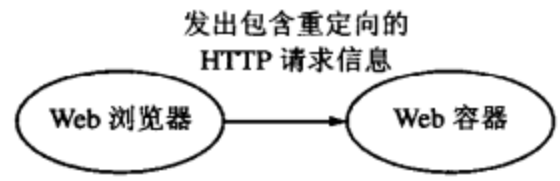
步骤 8：

Web 服务器（Web 容器）把包含重定向的响应信息发送给浏览器。一次请求响应过程完全结束，request 和 response 变成垃圾，等待垃圾收集器将其彻底从内存中清除。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

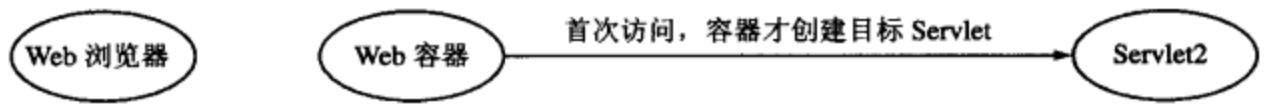
http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

步骤 9：



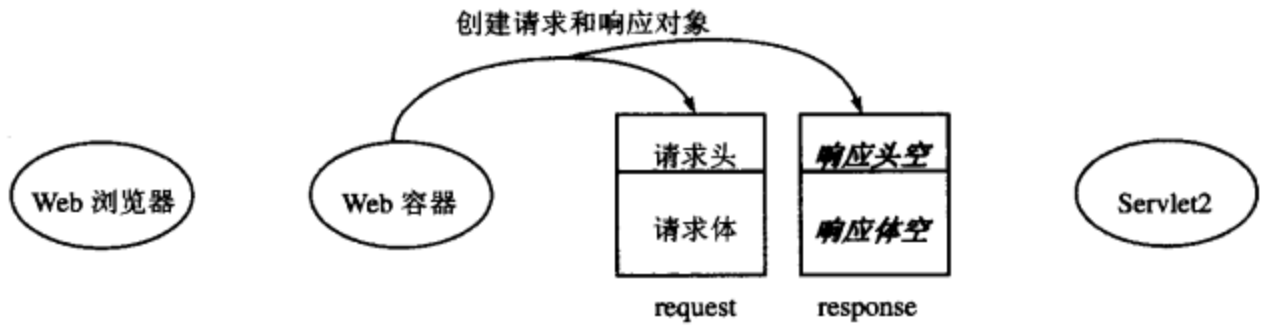
Web 浏览器根据上次返回的重定向 URL 发出新的 HTTP 请求，这个请求甚至可以指向另外一个站点，Web 浏览器又开始一次全新的请求响应过程。

步骤 10：



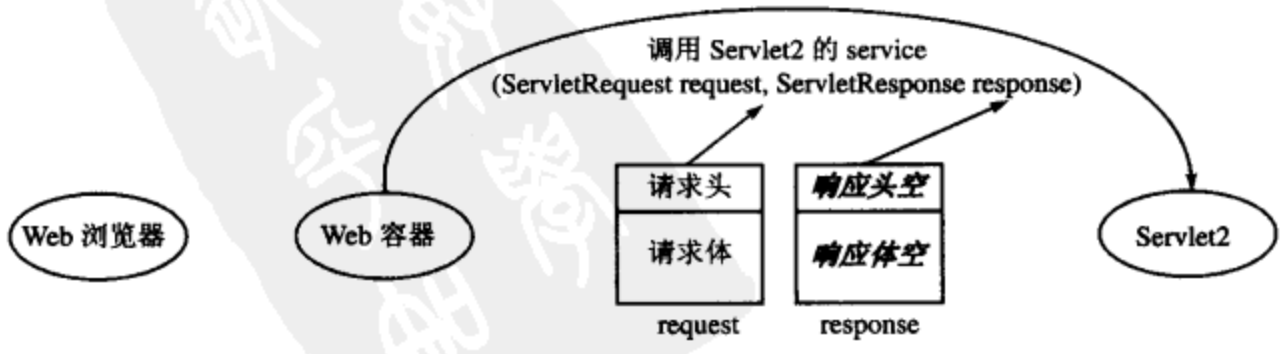
如果请求资源为 Servlet，Web 服务器（Web 容器）判断在内存中是否存在该 Servlet 对象，如果不存在，则加载和创建该 Servlet 对象。

步骤 11：



Web 服务器（Web 容器）创建针对这次访问的请求对象（request）和响应对象（response）。请求对象中包含了 HTTP 的请求信息，响应对象用于封装将要发送的 HTTP 响应信息，此时的响应对象中的内容为空。

步骤 12：

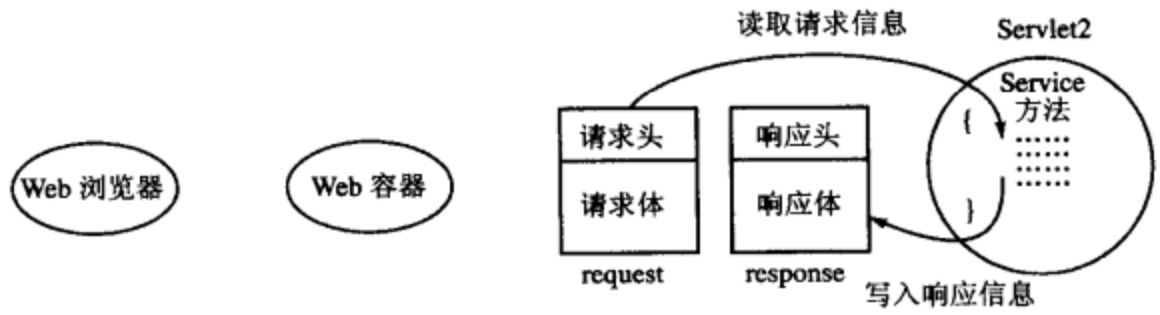


Web 容器调用 Servlet2 的 service（ServletRequest request，ServletResponse response）方法，并将这次新创建请求对象和响应对象传递给该方法。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

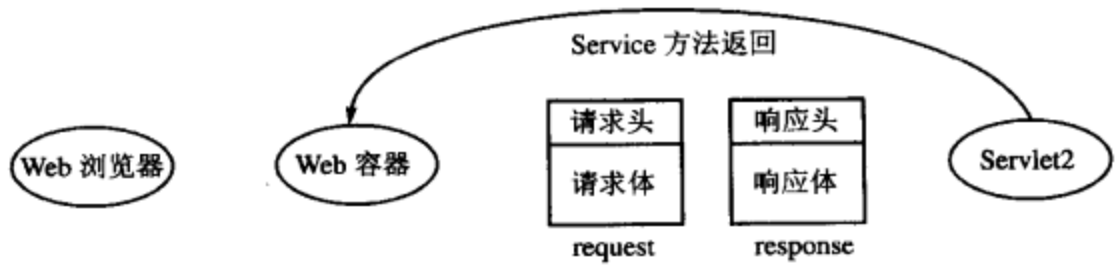
http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

步骤 13：



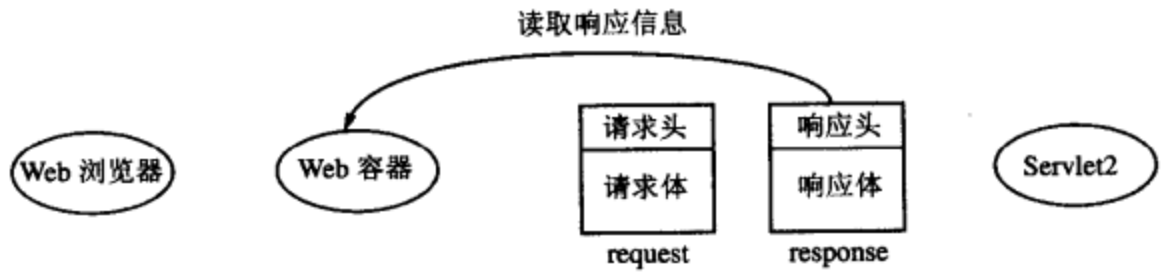
Servlet2 对象的 service 方法从请求对象中读取请求信息，并向响应对象中写入响应头和响应体信息。

步骤 14：



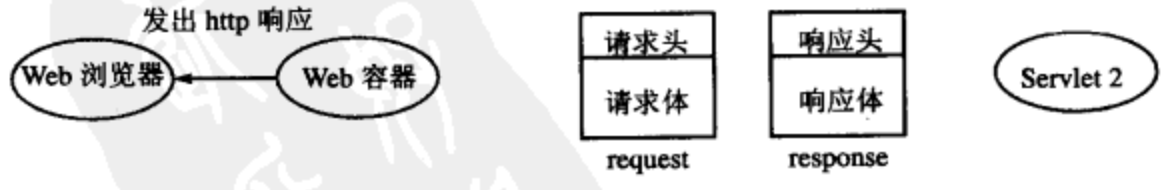
service 方法执行完毕后返回。

步骤 15：



Web 服务器（Web 容器）从响应对象中读取响应信息。

步骤 16：



Web 服务器（Web 容器）将响应信息发送给浏览器处理和显示，新的请求响应过程结束，request 和 response 变成垃圾，等待垃圾收集器将其彻底从内存中清除。

5.7.7 请求重定向与请求转发的比较

尽管 HttpServletResponse.sendRedirect 方法和 RequestDispatcher.forward 方法都可以让

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

浏览器获得另外一个 URL 所指向的资源，但两者的内部运行机制有着很大的区别。下面是 HttpServletResponse.sendRedirect 方法实现的请求重定向与 RequestDispatcher.forward 方法实现的请求转发的总结比较：

（1）RequestDispatcher.forward 方法只能将请求转发给同一个 Web 应用中的组件；而 HttpServletResponse.sendRedirect 方法不仅可以重定向到当前应用程序中的其他资源，还可以重定向到同一个站点上的其他应用程序中的资源，甚至是使用绝对 URL 重定向到其他站点的资源。如果传递给 HttpServletResponse.sendRedirect 方法的相对 URL 以“/”开头，它是相对于整个 Web 站点的根目录；如果创建 RequestDispatcher 对象时指定的相对 URL 以“/”开头，它是相对于当前 Web 应用程序的根目录。

（2）调用 HttpServletResponse.sendRedirect 方法重定向的访问过程结束后，浏览器地址栏中显示的 URL 会发生改变，由初始的 URL 地址变成重定向的目标 URL；而调用 RequestDispatcher.forward 方法的请求转发过程结束后，浏览器地址栏保持初始的 URL 地址不变。

（3）HttpServletResponse.sendRedirect 方法对浏览器的请求直接做出响应，响应的结果就是告诉浏览器去重新发出对另外一个 URL 的访问请求，这个过程好比有个绰号叫“浏览器”的人写信找张三借钱，张三回信说没有钱，让“浏览器”去找李四借，并将李四现在的通信地址告诉给了“浏览器”。于是，“浏览器”又按张三提供的通信地址给李四写信借钱，李四收到信后就把钱汇给了“浏览器”。可见，“浏览器”一共发出了两封信和收到了两次回复，“浏览器”也知道他借到的钱出自李四之手。

RequestDispatcher.forward 方法在服务器端内部将请求转发给另外一个资源，浏览器只知道发出了请求并得到了响应结果，并不知道在服务器程序内部发生了转发行为。这个过程好比绰号叫“浏览器”的人写信找张三借钱，张三没有钱，于是张三找李四借了一些钱，甚至还可以加上自己的一些钱，然后再将这些钱汇给了“浏览器”。可见，“浏览器”只发出了一封信和收到了一次回复，他只知道从张三那里借到了钱，并不知道有一部分钱出自李四之手。

（4）RequestDispatcher.forward 方法的调用者与被调用者之间共享相同的 request 对象和 response 对象，它们属于同一个访问请求和响应过程；而 HttpServletResponse.sendRedirect 方法调用者与被调用者使用各自的 request 对象和 response 对象，它们属于两个独立的访问请求和响应过程。如果一次请求的响应过程被设计成由 Web 应用程序内部的多个资源来协同完成，这就需要在同一个 Web 应用程序的内部资源之间进行跳转，并且在跳转之前往往要进行一些前期的预处理，然后再使用 HttpServletRequest.setAttribute 方法将预处理结果传递给下一个资源，这种情况下就必须使用 RequestDispatcher.forward 方法。不同 Web 应用程序之间的重定向，特别是要重定向到另外一个 Web 站点上的资源的情况，都应该使用 HttpServletResponse.sendRedirect 方法。

（5）无论是 RequestDispatcher.forward 方法，还是 HttpServletResponse.sendRedirect 方法，在调用它们之前，都不能有内容已经被实际输出到了客户端。如果缓冲区中已经有了一些内容，这些内容将被从缓冲区中清除。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

5.7.8 缺省 Servlet 的缓存问题

在实际项目中，有些 HTML 文件是在程序运行时调用一个模板文档来动态生成的，例如，一些站点的统计报表就是在用户访问某个 Servlet 时，由这个 Servlet 临时调用报表生成模块（例如，ireport）生成一个 HTML 文件，然后再将请求转发这个 HTML 文件。例程 5-20、例程 5-21 和例程 5-22 是一个模拟这种应用的程序，其中，ReportServlet.java 是为用户提供报表的 Servlet 程序，它首先调用 ReportTool.java 临时生成用做报表的 HTML 文件，然后将请求转发给这个临时生成的 HTML 文件；ReportTool.java 是一个负责生成报表 HTML 文件的辅助工具类，ReportTest.html 页面提供了分别以不同参数去访问 ReportServlet 的超链接。

例程 5-20 ReportServlet.java

```
package org.it315;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ReportServlet extends HttpServlet
{
    private int count = 0;
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        response.setContentType("text/html;charset=UTF-8");

        //禁止浏览器缓存
        response.setDateHeader("Expires",0);
        response.setHeader("Cache-Control","no-cache");
        response.setHeader("Pragma","no-cache");

        String department = request.getParameter("dept");
        synchronized(this)
        {
            System.out.println(++count + ":" + department);
            String path = getServletContext().getRealPath("/Web-INF/
            report.html");
            ReportTool rt = new ReportTool();
            rt.setCount(count);
            rt.setDepartment(department);
            rt.createReportHTML(path);
            RequestDispatcher rd = getServletContext().getRequest Dispatcher(
                                    "/Web-INF/report.html");
            rd.forward(request,response);
        }
    }
}
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

例程 5-21 ReportTool.java

```
package org.it315;
import java.io.*;

public class ReportTool
{
    private int count = 0;
    private String department = "";
    public void setCount(int count)
    {
        this.count = count;
    }
    public void setDepartment(String department)
    {
        this.department = department;
    }
    public void createReportHTML(String path) throws IOException
    {
        FileOutputStream fos = new FileOutputStream(path);
        OutputStreamWriter ops = new OutputStreamWriter(fos,"UTF-8");
        ops.write(String.valueOf(count) + ":" + department +
        "的统计信息");
        ops.close();
    }
}
```

例程 5-22 ReportTest.html

```
<a href="servlet/ReportServlet?dept=DEPT_A">查看部门 A 的统计信息</a>
<a href="servlet/ReportServlet?dept=DEPT_B">查看部门 B 的统计信息</a>
```

笔者接触过的一个学员在自己公司里发现了这种应用方式的一个重大问题：如果在 A 部门的一个员工查看了 A 部门的统计信息后的一个较短时间范围内，B 部门的员工来查看 B 部门的统计信息，结果后者看到的是 A 部门的统计信息。在程序中已经对 Servlet 的多线程并发访问的安全问题和浏览器缓存问题进行了处理，因此，可以排除多线程并发访问的安全因素和浏览器缓存的问题。这个问题困扰了这个学员和他们公司很长一段时间，在万般无奈之下，他们找到了笔者来帮助分析这个问题。笔者分析和解决这个问题后，觉得有必要与广大读者共享这一宝贵经验，于是特地写了上面的模拟程序来演示和解释这个问题。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

缺省 Servlet 的缓存问题

01 编写和编译上面的 ReportServlet.java 和 ReportTool.java 源文件，确保编译后生成的 class 文件存放在了<tomcat 的安装目录>\Webapps\it315\Web-INF\classes 目录中，注意要有与 org.it315 包名对应的目录结构。编写上面的 ReportTest.html 文件，并将它保存到<tomcat 的安装目录>\Webapps\it315 目录中。

02 在<tomcat 的安装目录>\Webapps\it315\Web-INF\Web.xml 文件中注册 ReportServlet，并设置其映射 URL。在 Web.xml 文件中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>ReportServlet</servlet-name>
    <servlet-class>org.it315.ReportServlet</servlet-class>
</servlet>

……
<servlet-mapping>
    <servlet-name>ReportServlet</servlet-name>
    <url-pattern>/servlet/ReportServlet</url-pattern>
</servlet-mapping>
```

保存 Web.xml 文件后，重新启动 Tomcat。

03 在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/ReportTest.html
```

在 ReportTest.html 页面中单击“查看部门 A 的统计信息”的超链接，可以看到返回的页面中显示出了部门 A 的信息。稍等十几秒钟后，单击浏览器工具栏中的“后退”按钮，退回到 ReportTest.html 页面，再单击“查看部门 B 的统计信息”的超链接，可以看到返回的页面中显示出了部门 B 的信息。看到部门 B 的统计信息后，快速退回到 ReportTest.html 页面，快速单击“查看部门 A 的统计信息”的超链接，注意：这时候，返回页面中显示出的仍然是上次显示的部门 B 的信息！查看 Tomcat 运行的命令行窗口，可以看到其中打印出了如下三行信息：

```
1:DEPT_A
2:DEPT_B
3:DEPT_A
```

这说明 ReportServlet 程序确实接收到了浏览器发出的第 3 次请求，并且处理了这次请求，只是 Web 服务器返回给浏览器的页面内容仍然是第 2 次访问时所生成的 report.html 文件中的内容。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

04 多次刷新浏览器的访问，每次访问都在 Tomcat 运行的命令行窗口中打印出了一行信息，并且打印出的访问计数在不断增加，这说明 ReportServlet 程序确实接收和处理了浏览器发出的每次访问请求，但浏览器中显示的内容总要间隔一段时间后才能变化。

只要知道“客户端对静态 HTML 文件和图片的访问其实都是由缺省 Servlet 来完成响应的，访问 HTML 文件实际上就是在调用缺省 Servlet”，就很容易想到上面的现象可能是由于缺省 Servlet 输出 report.html 文件时造成的，最大的可能性就是缺省 Servlet 输出某个 HTML 文件内容时会将这个 HTML 文件内容缓存起来，在后续一段时间范围内发生的针对该文件的访问请求，直接使用缓存中的内容进行响应，而不是每次都去硬盘上读取 HTML 文件。查看 Tomcat 的缺省 Servlet 的源文件及其中调用的其他类的源文件，可以看到这一猜想完全正确，并且缓存的时间为 5 秒钟，其中一些典型的代码如下：

```
// BaseDirContext.java
protected int cacheTTL = 5000; // 5s
//ProxyDirContext.java
cacheEntry.timestamp = System.currentTimeMillis() + cacheTTL;
```

找到问题的原因后，接下来就应该考虑如何解决问题了，笔者提供了以下三种方案，读者应根据自身的实际情况进行选择：

- 为每一种统计方案都生成一个名称不同的 HTML 报表文件，例如，不同部门使用不同名称的报表文件，那么就不会再出现一个统计方案使用另外一个统计方案的报表的情况了。由于统计报表一般并不要求做到实时，通常都是统计当前时间以前的某个时间段的信息，如果已经为某种统计方案生成了 HTML 报表文件，那么，以后对这种统计方案就可以一直使用已经存在的 HTML 报表文件，而不用再重新生成。所以，在这种情况下，缺省 Servlet 的缓存问题根本就不用考虑。如果统计方案比较多，生成的 HTML 报表文件的个数也就会很多，所以，这个方案比较适合只有少数几种固定的统计方案的情况，特别不适合要根据各种组合条件来生成报表的情况。
- 修改 Tomcat 的缺省 Servlet 程序，让它不要进行缓存，这种方案可能会严重影响系统的性能，一般不要采用。
- 在 ReportServlet 程序中生成 HTML 报表文件后，按字节流的形式读取该文件中的内容，然后以字节流的形式直接输出到客户端。这种方案可以生成实时的统计报表，也适合根据各种组合条件来生成报表的情况，所以，推荐采用这种方案。

5.8 思考与实践

友情提示：请认真地完成以下思考与实践，并将完成的结果以及操作步骤提交给 www.it315.org 站点评审，表现优异者将获得 www.it315.org 站点的奖励和专家的特别指导。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

1. 请描述 ServletResponse.setContentType 方法的作用，然后参照本书编写实验步骤的方法设计一个详细的实验步骤来解释其作用。

2. 如果一个网页文档内的超链接指向的资源是一个 html、jpeg 或 txt 等类型的文档，用户单击该超链接将得不到提示下载的信息，而是直接在浏览器窗口中打开了该文件。要想下载这种类型的文件，用户需要在超链接上单击鼠标右键，然后单击弹出菜单的“目标另存为”菜单。可是，一些网站可能需要这样的功能：列出某个目录下的所有文件和子目录列表，每个文件和子目录都作为一个超链接的文本；如果用户单击指向某个文件的超链接，不管文件类型怎样，都对文件进行下载，而不是直接打开这个文件；如果用户单击指向某个子目录的超链接，则列出该目录下的所有文件和子目录列表。要实现这样的功能，不能让超链接直接指向要下载文件的URL，而是应参照5.6.4节的例子程序编写一个Servlet程序，让 Servlet 程序读取要下载的文件内容，然后再传递给浏览器。请编写一个完成这种功能的程序。

提示

首先让 Servlet 程序显示出某个目录下的所有文件和子目录列表，列表中的所有文件和子目录的超链接又都指向该 Servlet 程序，要下载文件名或要打开的目录名作为超链接附带的一个参数传递给 Servlet 程序，关于如何向 Servlet 程序传递参数以及 Servlet 程序如何获取参数，请参看 6.6 节的讲解。

3. 参照 5.6.2 节讲解的图像访问计数器的例子程序，编写一个使用<script>标签来统计和显示页面访问次数的程序。

4. 在使用 RequestDispatcher.include 方法时，如果被调用者输出的内容中包含有使用相对 URL 的访问路径，当调用者与被调用者的访问 URL 不属于同一个目录时，这个相对 URL 是相对于调用者的 URL，还是相对于被调用者的 URL 呢？在不允许使用绝对路径的情况下，该如何解决这个问题？请编写一个实验程序和详细的操作步骤。

5. 有人为了禁止浏览器缓存某个普通的 HTML 页面，他采用的方式是让浏览器不要直接访问这个 HTML 页面，而是访问一个前端 Servlet，在这个前端 Servlet 中增加那些禁止浏览器缓存的语句后，再跳转到普通 HTML 页面。请问，跳转到普通 HTML 页面时应该采用哪种方式？

A. HttpServletResponse.sendRedirect

B. RequestDispatcher.forward

C. A 和 B 都可以

6. 请编写一个 Servlet 程序，并在部署描述符文件（Web.xml）中采用通配符（*）的形式将多个 URL 映射到该 Servlet 上。如果在这个 Servlet 程序中使用 RequestDispatcher.forward 方法跳转到某个 URL，而跳转到的 URL 也匹配该 Servlet 所映射的 URL（即仍由这个 Servlet 处理），这将会出现怎样的运行效果？。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

第 6 章　HttpServletRequest 的应用

6.1　HttpServletRequest 简介

Web 客户端发送给 Web 服务器的 HTTP 请求消息可以分为三个部分：请求行、请求消息头、消息正文（也叫实体内容）。消息正文只在以 POST、PUT 和 DELETE 等方式发出的请求消息中才出现，譬如，HTML 的 FORM 表单中的各个元素信息通常都以 POST 请求方式的消息正文传递给 Web 服务器。HTTP 和 HTML 这两个 Web 应用的协议都提供了对表单提交数据的支持，但数据接收则由 Web 服务器自行处理。Servlet API 中定义了一个 ServletRequest 接口类来作为获取客户端请求信息的基本接口，Servlet 程序通过调用 ServletRequest 对象的方法可以获知客户端的请求信息，以及客户机和服务器的网络环境信息，例如，获取客户发送的请求消息的各个部分和客户端 IP 地址。HttpServletRequest 是专用于 HTTP 协议的 ServletRequest 子接口，它用于封装 HTTP 请求消息，增加了获取 HTTP 协议专有的头信息的方法，支持 Cookies 和 session 跟踪，以及获取 HTTP 请求消息参数的功能。

读者在前面章节中已经了解到：客户端对一个 Servlet 的每次访问请求，Servlet 引擎都会创建一个 HttpServletRequest 请求对象和一个 HttpServletResponse 响应对象，然后调用此 Servlet 的 service()方法，并将这两个对象作为参数传递给它。所以，对 HttpServletRequest 的编程应用，也就是在 service()方法内部调用作为参数传递进来的 HttpServletRequest 请求对象的各种方法。

6.2　获取请求行的相关信息

HTTP 请求消息的请求行可以分为三个部分：请求方式、资源路径和 HTTP 协议版本，如下所示：

```
GET /it315/servlet/RequestURI?param1=a&param2=b HTTP/1.1
```

HttpServletRequest 接口中定义了若干获取请求行中的各个部分的方法。

6.2.1　getMethod 方法

getMethod 方法返回 HTTP 请求消息中的请求方式（如 GET、POST，HEAD、PUT 和

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

DELETE 等），也就是请求行中的第一部分的内容。

6.2.2 getRequestURI 方法

getRequestURI 方法返回请求行中的资源名部分，也就是位于 URL 的主机和端口之后、参数部分之前的那部分内容。例如，对于如下的 URL：

```
http://www.it315.org:80/servlet/RequestURI?param1=a&param2=b
```

getRequestURI 方法返回“/servlet/RequestURI”。

6.2.3 getQueryString 方法

getQueryString 方法返回请求行中的参数部分，也就是资源路径后面的问号（?）以后的所有内容。返回结果没有被进行 URL 解码，而是保持其在 HTTP 消息的请求行中的原始面貌。例如，对于如下的 URL：

```
http://www.it315.org:80/servlet/RequestURI?param1=a&param2=b
```

getQueryString 方法返回“Param1=a&param2=b”。如果资源路径后本身就没有参数部分，getQueryString 返回 null。

6.2.4 getProtocol 方法

getProtocol 方法返回请求行中的协议名和版本，即请求行的第三部分，它一般是 HTTP/1.0 或 HTTP/1.1。如果有必要，可以在设置响应头前先调用 getProtocol 方法检查一下请求消息中的 HTTP 版本号，然后只使用与该 HTTP 版本对应的状态码和响应消息头，不要使用比该版本更高的 HTTP 所特有的状态码和响应消息头，从而避免使用对当前客户端无效的状态码和响应消息头。

6.2.5 getContextPath 方法

getContextPath 方法返回请求 URL 所属于的 Web 应用程序的路径，这个路径以“/”开头，表示相对于整个 Web 站点的根目录，路径结尾不含“/”。如果请求 URL 属于 Web 站点的根目录，那么返回结果为空字符串（""）。

6.2.6 getPathInfo 方法

getPathInfo 方法返回请求 URL 中的额外路径信息。额外路径信息是请求 URL 中的位于 Servlet 的路径之后和查询参数之前的内容，它以“/”开头。如果请求 URL 中没有额外路径信息部分，getPathInfo 返回 null。

额外路径可用于向 Servlet 传递信息，但它传递的信息通常是指 Web 服务器上的其他资源名，例如，配置文件、临时文件或者其他 Servlet、JSP 页面。在 MVC 设计模式中，客户端的所有请求都指向一个作为中央控制器的 Servlet，其他各个 JSP 页面则作为请求

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

URL 中的额外路径部分，由这个 Servlet 根据额外路径信息去调用其他各个 JSP 页面。只要在 web.xml 文件中将某个 Servlet 映射成一个通配符形式的路径，例如，“/controller/*”，然后就可以使用“/controller/one.jsp”、“/controller/two.jsp”等多个路径来访问这个 Servlet，其中的“/one.jsp”和“/two.jsp”等即为额外路径信息，可以让这些额外路径信息正好是 Web 服务器上的其他资源名称。

6.2.7 getPathTranslated 方法

getPathTranslated 方法返回 URL 中的额外路径信息所对应的资源的真实路径。假设“/controller/one.jsp”中的“/one.jsp”为额外路径信息，getPathTranslated 即为“/one.jsp”所对应的资源文件的真实路径。

6.2.8 getServletPath 方法

getServletPath 方法返回 Servlet 的名称或 Servlet 所映射的路径。

6.2.9 获取请求行信息的综合实例

下面编写一个 Servlet 程序，在其 service 方法中调用上面讲解的与请求行相关的各个方法，并将这些方法的返回结果输出到浏览器中查看。

与请求行信息相关的各个方法的作用

01 按例程 6-1 编写一个名为 RequestLineServlet 的 Servlet 程序。

例程 6-1 RequestLineServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class RequestLineServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        response.setContentType("text/html ;charset=gb2312");
        PrintWriter out = response.getWriter();
        out.println("getMethod : " + request.getMethod() + "<br>");
        out.println("getRequestURI : " + request.getRequestURI() + "<br>");
        out.println("getQueryString : " + request.getQueryString()
        + "<br>");
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        out.println("getProtocol : " + request.getProtocol() + "<br>");
        out.println("getContextPath : " + request.getContextPath()
        + "<br>");
        out.println("getPathInfo : " + request.getPathInfo() + "<br>");
        out.println("getPathTranslated : " + request.getPathTranslated()
        + "<br>");
        out.println("getServletPath : " + request.getServletPath()
        + "<br>");
    }
}
```

编译 RequestLineServlet.java 文件，确保编译后生成的 class 文件存放在了<tomcat 的安装目录>\webapps\it315\Web-INF\classes 目录中。

02 在<tomcat 的安装目录>\webapps\it315\Web-INF\web.xml 文件中注册该 Servlet，并设置其映射 URL。在 web.xml 文件中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>RequestLineServlet</servlet-name>
    <servlet-class>RequestLineServlet</servlet-class>
</servlet>
……
<servlet-mapping>
    <servlet-name>RequestLineServlet</servlet-name>
    <url-pattern>/controller/*</url-pattern>
</servlet-mapping>
```

保存 web.xml 文件后，重新启动 Tomcat。

03 在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/controller/one.jsp?param1=abc
```

浏览器中显示的结果如图 6.1 所示，从中可以很清楚地了解到各个方法的作用。

图 6.1

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

6.3 获取网络连接信息

Web 服务器与客户端的通信是建立在 TCP 连接上的，服务器端程序经常需要知道客户端的一些网络连接信息，为此，ServletRequest 接口中定义了若干获取客户端和服务器的网络连接信息和访问特性的方法。

6.3.1 getRemoteAddr 方法

getRemoteAddr 方法返回发出请求的客户机的 IP 地址，其格式为“192.168.0.3”这种形式的字符文本。

6.3.2 getRemoteHost 方法

getRemoteHost 方法返回发出请求的客户机的完整主机名，即“pc1.it315.org”这种格式，如果 Servlet 引擎不能解析出客户机的完整主机名，那么该方法就返回客户机的 IP 地址。

6.3.3 getRemotePort 方法

getRemotePort 方法返回发出请求的客户机所使用的网络接口的端口号。

6.3.4 getLocalAddr 方法

getLocalAddr 方法返回 Web 服务器上接收当前请求的网络接口的 IP 地址。

6.3.5 getLocalName 方法

getLocalName 方法返回 Web 服务器上接收当前请求的网络接口的 IP 地址所对应的主机名。

6.3.6 getLocalPort 方法

getLocalPort 方法返回 Web 服务器上接收当前请求的网络接口的端口号。

6.3.7 getServerName 方法

getServerName 方法返回当前请求所指向的主机名。如果 HTTP 请求消息的 Host 头字段中包含有主机名部分，那么返回值即为该主机名部分。

6.3.8 getServerPort 方法

getServerPort 方法返回当前请求所连接的服务器端口号。如果 HTTP 请求消息的 Host

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

头字段中包含有端口号部分，那么返回值即为该端口号部分。

6.3.9 getScheme 方法

getScheme 方法返回请求的协议名，例如 http、https 或 ftp。

6.3.10 getRequestURL 方法

getRequestURL 方法返回客户端发出请求时的完整 URL，包括协议、服务器名、端口号、资源路径等信息，但不包括后面的查询参数部分。注意，getRequestURL 方法返回的结果是 StringBuffer 类型，而不是 String 类型，这样更便于对结果进行修改，例如，在其后面追加查询参数。

6.4 获取请求头信息

ServletRequest 和 HttpServletRequest 接口中定义了若干用于读取 HTTP 请求消息中的头字段的方法，其中的 getHeader 方法是一个通用的方法，可用于读取所有的头字段。除了 getHeader 方法外，HttpServletRequest 还定义了一些便利的方法，例如，getContentType、getContentLength、getIntHeader、getDateHeader 等方法，用来方便地读取那些常用的请求头和读取包含有日期、整数数据的请求头。

通过利用客户端发送的 HTTP 请求消息的头字段，可以在 Servlet 中实现丰富多样的功能，例如，依据浏览器的国家语言版本信息输出相应的国家语言的网页内容，获取发出当前访问请求的超链接所在页面的 URL 地址等信息。要实现这些特殊功能，首先需要了解有哪些作用的请求头字段及其具体用法。

6.4.1 getHeader 方法

getHeader 方法用于返回一个指定名称的头字段的值，其完整的语法定义如下：

```
public java.lang.String getHeader(java.lang.String name)
```

如果请求消息中没有包含指定名称的头字段，getHeader 方法返回 null；如果请求消息中包含有多个该指定名称的头字段，getHeader 方法返回其中的第一个头字段的值。由于头字段名对大小写不敏感，因此，调用 request.getHeader("Connection")和调用 request.getHeader ("connection")都可以返回 Connection 头字段的值。

6.4.2 getHeaders 方法

getHeaders 方法用于返回一个 Enumeration 集合对象，该集合对象由请求消息中出现的某个指定名称的所有头字段值组成。getHeaders 方法的完整语法定义如下：

```
public java.util.Enumeration getHeaders(java.lang.String name)
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

在大多数情形下，一个头字段名在请求消息中只出现一次，但有时候可能会出现多次，例如，请求消息中可以有多个 Accept-Language 头字段，每个 Accept-Language 头字段分别单独设置一个不同的值。

6.4.3 getHeaderNames 方法

getHeaderNames 方法用于返回一个包含请求消息中的所有头字段名的 Enumeration 对象，在此基础上，可以对请求消息中的所有头字段进行遍历处理。getHeaderNames 方法的完整语法定义如下：

```
public java.util.Enumeration getHeaderNames()
```

6.4.4 getIntHeader 方法

getIntHeader 方法用于获取一个指定名称的头字段的值并将其转换成 int 类型后返回，如果指定名称的头字段不存在，返回值为-1；如果获得的头字段值不能转换成整数，则抛出 NumberFormatException 异常。getIntHeader 方法的完整语法定义如下：

```
public int getIntHeader(java.lang.String name)
```

6.4.5 getDateHeader 方法

getDateHeader 方法用于获取一个指定名称的头字段的值并将其按 GMT 时间格式转换成一个代表日期/时间的长整数后返回，这个长整数是自 1970 年 1 月 1 日 0 点 0 分 0 秒算起的以毫秒为单位的时间值。getDateHeader 方法可用于读取含有日期的头字段，例如 If-Modified-Since，如果指定名称的头字段不存在，返回值为-1；如果获得的头字段值不能转换成整数，则抛出 NumberFormatException 异常。getDateHeader 方法的完整语法定义如下：

```
public long getDateHeader(java.lang.String name)
```

6.4.6 getContentType 方法

getContentType 方法可用于直接返回 Content-Type 头字段的值，结果为 String 类型。

6.4.7 getContentLength 方法

getContentLength 方法可用于直接返回 Content-Length 头字段的值，结果为 int 类型。

6.4.8 getCharacterEncoding 方法

getCharacterEncoding 方法用于返回请求消息的实体部分的字符集编码，通常从 Content-Type 头字段中进行提取。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

6.4.9 获取所有请求头的编程实例

获取请求消息中的所有请求头信息

01 按例程 6-2 编写一个名为 RequestHeadersServlet 的 Servlet 程序，在其 service 方法中打印出请求消息中包含的所有请求头信息。

例程 6-2 RequestHeadersServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Enumeration;

public class RequestHeadersServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        response.setContentType("text/html ;charset=gb2312");
        PrintWriter out = response.getWriter();
        Enumeration headerNames = request.getHeaderNames();
        while(headerNames.hasMoreElements())
        {
            String headerName = (String)headerNames.nextElement();
            out.print(headerName + " : " + request.getHeader(headerName)
            + "<br>");
            /*如果要考虑同一个请求头名可能出现多次，
            那么应该用下面的代码段代替上面一行程序代码*/
            /*Enumeration values = request.getHeaders(headerName);
            while(values.hasMoreElements())
            {
                out.print(headerName + " : " + (String)values.nextElement()
                + "<br>");
            }*/
        }
    }
}
```

编译 RequestHeadersServlet.java 文件，确保编译后生成的 class 文件存放在了<tomcat 的安装目录>\webapps\it315\Web-INF\classes 目录中。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

02 在<tomcat 的安装目录>\webapps\it315\Web-INF\web.xml 文件中注册该 Servlet，并设置其映射 URL。在 web.xml 文件中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>RequestHeadersServlet</servlet-name>
    <servlet-class>RequestHeadersServlet</servlet-class>
</servlet>
……
<servlet-mapping>
    <servlet-name>RequestHeadersServlet</servlet-name>
    <url-pattern>/servlet/RequestHeadersServlet</url-pattern>
</servlet-mapping>
```

保存 web.xml 文件后，重新启动 Tomcat。

03 在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/servlet/RequestHeadersServlet
```

浏览器中显示的结果如图 6.2 所示。

04 在<tomcat 的安装目录>\webapps\it315 目录中编写一个名称为 referrer.html 的网页文件，内容如下：

```
<a href="servlet/RequestHeadersServlet">referrer test</a>
```

接着在浏览器中访问 http://localhost:8080/it315/referrer.html 页面，然后单击其中的超链接，浏览器中显示的结果如图 6.3 所示。

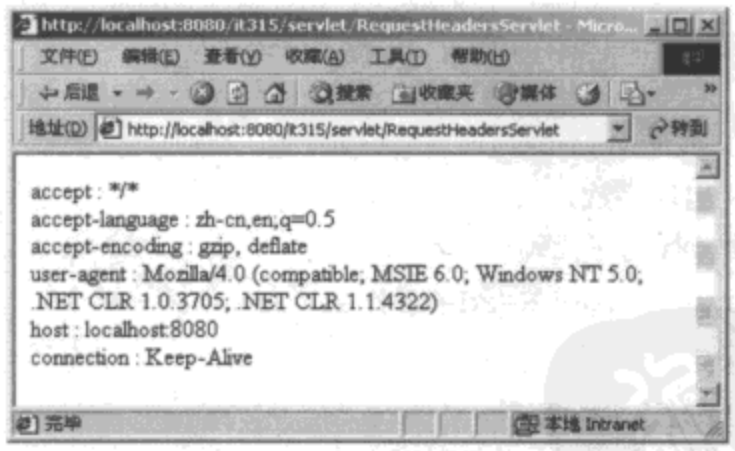


图 6.2

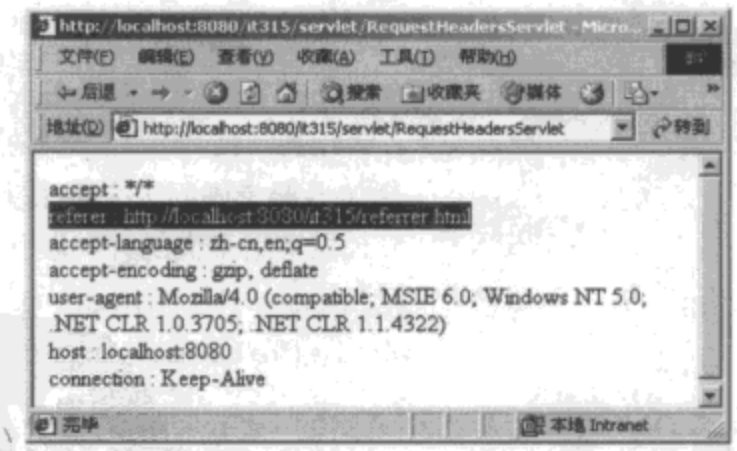


图 6.3

在图 6.3 中，读取到了一个名为 referer 的头字段，它的值是发出请求的超链接源（超链接所在网页）的 URL，对于上面的情况就是 referrer.html 页面的 URL。如果直接在浏览器地址栏中输入 RequestHeadersServlet 的访问地址，浏览器不会发送 Referer 请求头。注意，由于 HTTP 协议原作者的拼写错误，这个头字段的拼写是 Referer，而不是 Referrer。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

6.4.10 利用 Referer 请求头阻止“盗链”

有一些站点自身没有提供下载空间，但是为了吸引人气和提高站点的访问量，他们也提供了各种软件的下载页面，并让下载的超链接指向其他站点上的资源。另外一些真正提供了下载空间的站点为了防止这种“盗链”，需要检查请求的来路，只接受本站内的页面链接进来的下载请求，而阻止其他站点的页面链接进来的下载请求。要实现这样的功能，就需要检查请求消息的 referer 头字段是否与本站匹配。

利用 Referer 请求头阻止“盗链”

01 按例程 6-3 编写一个名为 DownManagerServlet 的 Servlet 程序，这个 Servlet 程序负责提供下载内容，但它要求下载请求必须通过本站的下载页面链接进来，否则将请求转发给本站的下载说明页。

例程 6-3 DownManagerServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DownManagerServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        response.setContentType("text/html ;charset=gb2312");
        PrintWriter out = response.getWriter();
        String referrer = request.getHeader("referer");
        String sitePart = "http://" + request.getServerName();
        if(referrer!=null && referrer.startsWith(sitePart))
        {
            //处理正当的下载请求，这里只进行示意
            out.println("dealing download ...");
        }
        else
        {
            //非法下载请求跳转到本站的下载说明页
            RequestDispatcher rd = request.getRequestDispatcher("/down
            .html");
            rd.forward(request,response);
        }
    }
}
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

编译 DownManagerServlet.java 文件，确保编译后生成的 class 文件存放在了<tomcat 的安装目录>\webapps\it315\Web-INF\classes 目录中。

02 在<tomcat 的安装目录>\webapps\it315\Web-INF\web.xml 文件中注册该 Servlet，并设置其映射 URL。在 web.xml 文件中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>DownManagerServlet</servlet-name>
    <servlet-class>DownManagerServlet</servlet-class>
</servlet>
……
<servlet-mapping>
    <servlet-name>DownManagerServlet</servlet-name>
    <url-pattern>/servlet/DownManagerServlet</url-pattern>
</servlet-mapping>
```

保存 web.xml 文件后，重新启动 Tomcat。

03 在<tomcat 的安装目录>\webapps\it315 目录中编写一个名称为 down.html 的网页文件，如例程 6-4 所示。

例程 6-4　down.html

```
<base href="http://localhost:8080/it315/down.html" />
<a href="servlet/DownManagerServlet">down</a>
```

接着在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/servlet/DownManagerServlet
```

由于这是直接在浏览器地址栏中输入的访问地址，请求消息中不含 referer 请求头，DownManagerServlet 将 down.html 页面转发给浏览器，浏览器中显示的结果如图 6.4 所示。

单击图 6.4 中的超链接再次访问 DownManagerServlet，由于这时的请求消息中包含有 Referer 请求头且其值与 DownManagerServlet 位于同一 Web 站点，DownManagerServlet 接受下载请求，浏览器中显示的结果如图 6.5 所示。

图 6.4

图 6.5

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

6.4.11 利用 Referer 请求头隐藏 JavaScript 源代码

利用 JavaScript 可以实现许多网页特效功能，但是，访问者可以轻松地查看到所有的 JavaScript 源代码，即使是将 JavaScript 代码放在单独的 js 文件中编写也无法保密。如果让 js 文件通过一个服务器端程序动态创建，结合使用 Referer 请求头字段就可以防止他人查看到 JavaScript 源码。

利用 Referer 请求头隐藏 JavaScript 源代码

01 在<tomcat 的安装目录>\webapps\it315 目录中编写如例程 6-5 和例程 6-6 所示的两个文件。

例程 6-5 js_call.html

```
<script type="text/javascript" src="test.js">
</script>
```

例程 6-6 test.js

```
document.write("假设这是很多重要的 Javascript 代码的执行结果");
```

02 在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/js_call.html
```

看到浏览器窗口中的返回结果后，单击“查看”→“源文件”菜单，可以看到 js_call.html 文件的原始内容。从查看到的 js_call.html 文件内容中，可以推断出浏览器窗口中的显示结果是由 test.js 文件中的 javascript 代码完成的，并且可以推断出 test.js 文件的访问路径为 http://localhost:8080/it315/test.js。在浏览器地址栏中直接输入 test.js 文件的访问路径，就可以下载或查看到 test.js 文件中的内容了。

03 为了阻止别人在浏览器地址栏中直接输入 test.js 文件的访问路径来查看其中的 JavaScript 代码，可以用一个 Servlet 程序来输出 test.js 文件中的内容，即用一个 Servlet 程序来代替 test.js 文件。如果直接在浏览器地址栏中输入 Servlet 程序的访问路径，那么 Servlet 程序就什么也不输出，只有通过其他网页来引用这个 Servlet 程序时，Servlet 程序才输出 test.js 文件中的 JavaScript 代码。按例程 6-7 编写一个名为 HideJSServlet 的 Servlet 程序，在其 service 方法中检查 referer 请求头是否来自本站，并决定是否输出 javascript 代码。

例程 6-7 HideJSServlet.java

```
import java.io.*;
import javax.servlet.*;
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


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

public class HideJSServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        response.setContentType("text/javascript;charset=GB2312");
        PrintWriter out = response.getWriter();
        String referrer = request.getHeader("referer");
        String sitePart = "http://" + request.getServerName();
        if(referrer!=null && referrer.startsWith(sitePart))
        {
            //向客户端输出 javascript 的 document.write(...)语句
            out.println(
                "document.write('假设这是很多重要的 Javascript 代码的执行
                结果');");
        }
    }
}
```

编译 HideJSServlet.java 文件，确保编译后生成的 class 文件存放在了<tomcat 的安装目录>\webapps\it315\Web-INF\classes 目录中。

04 在<tomcat 的安装目录>\webapps\it315\Web-INF\web.xml 文件中注册该 Servlet，并设置其映射 URL。在 web.xml 文件中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>HideJSServlet</servlet-name>
    <servlet-class>HideJSServlet</servlet-class>
</servlet>
……
<servlet-mapping>
    <servlet-name>HideJSServlet</servlet-name>
    <url-pattern>/servlet/HideJSServlet</url-pattern>
</servlet-mapping>
```

保存 web.xml 文件后，重新启动 Tomcat。

05 修改 js_call.html 文件中的内容，将<script>标签的 src 属性由"test.js"修改成"servlet/HideJSServlet"，结果如下：

```
<script type="text/javascript" src="servlet/HideJSServlet">
</script>
```

接着在浏览器地址栏中再次输入如下地址：

```
http://localhost:8080/it315/js_call.html
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

可以看到浏览器窗口中的执行结果和以前完全一样。单击“查看”→“源文件”菜单，看到其中的<script>标签的 src 属性值为"servlet/HideJSServlet"，因此可以推断出 js 文件的访问路径为 http://localhost:8080/it315/servlet/HideJSServlet。在浏览器地址栏中直接输入这个地址，针对不同的浏览器设置，看到的返回结果可能不一样。有的浏览器看到的返回结果为一个空白网页文档，这似乎基本上实现了对 JavaScript 源码进行隐藏的效果，但有的浏览器却能看到 HideJSServlet 输出的 JavaScript 代码。单击 IE 浏览器的“工具”→“Internet 选项”菜单，打开“Internet 选项”对话框，接着再单击“常规”选项卡中的“Internet 临时文件”栏中的“设置”按钮，打开如图 6.6 所示的“设置”对话框。

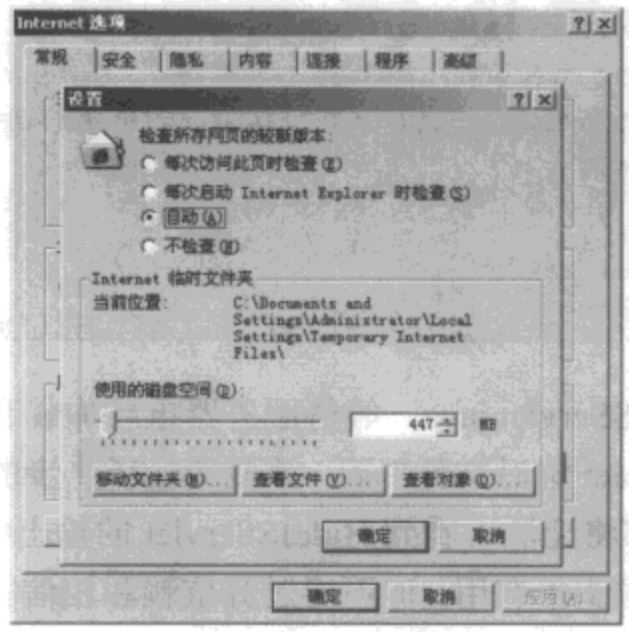


图 6.6

如果图 6.6 中的“检查所存网页的较新版本”功能项的设置为“每次访问此页时检查”，那么就看不到 HideJSServlet 输出的 JavaScript 代码。如果该功能项的设置为“自动”，则能看到 HideJSServlet 输出的 JavaScript 代码，这是因为在这种设置下，浏览器在通过 js_call.html 页面引用 HideJSServlet 时，会将 HideJSServlet 的输出缓存起来，而下次访问 HideJSServlet 时，浏览器则直接从缓存中取出 HideJSServlet 的输出内容。

06 为了阻止用户利用浏览器缓存的方式来查看 HideJSServlet 输出的 JavaScript 代码，需要修改 HideJSServlet.java 程序，设置一些特殊的响应头来禁止浏览器缓存 HideJSServlet 的输出，如例程 6-8 所示。

例程 6-8 HideJSServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HideJSServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        response.setContentType("text/javascript;charset=GB2312");
        response.setDateHeader("Expires",0);
        response.setHeader("Cache-Control","no-cache");
        response.setHeader("Pragma","no-cache");

        PrintWriter out = response.getWriter();
        String referrer = request.getHeader("referer");
        String sitePart = "http://" + request.getServerName();
        if(referrer!=null && referrer.startsWith(sitePart))
        {
            //向客户端输出 javascript 的 document.write(...)语句
            out.println(
                "document.write('假设这是很多重要的 Javascript 代码的执行结果
                ');");
        }
    }
}
```

07 重新编译 HideJSServlet.java，等待服务器重新加载和在图 6.6 所示的“Internet 选项”对话框中删除“Internet 临时文件”后，重复第 **05** 步的访问操作过程，不管浏览器的设置情况怎样，这时都将无法再看到 HideJSServlet 的输出结果了。

08 虽然 referer 请求头很有用，但不要过分依赖和相信它，因为用户可以使用类似 telnet 这样的客户端程序，轻易地伪造出一个 referer 请求头。用 Telnet 程序连接到本地的 Tomcat Web 服务器监听的 8080 端口，在 Telnet 程序窗口中输入如下几行内容：

```
GET /it315/servlet/HideJSServlet HTTP/1.1
Host: localhost:8080
Referer: http://localhost:8080/it315/js_call.html
<空行>
```

Telnet 程序窗口中显示出了 HideJSServlet 输出的响应信息，其中包含有 JavaScript 源代码，如图 6.7 所示。

图 6.7

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

6.4.12 BASE64 编码及客户端身份认证

读者在前面讲解的 HTTP 协议中已经了解到，当客户端访问 Web 服务器时，Web 服务器可以发送 401（Unauthorized）响应状态码和 WWW-Authenticate 响应头来要求客户端进行身份认证，而客户端则可以再次发出请求，并通过 Authorization 请求头来提供用户名和密码信息，服务器接着对 Authorization 请求头中的身份信息进行认证，并根据认证结果来决定是否提供服务。关于 WWW-Authenticate 响应头和 Authorization 请求头的详细细节，请参看本书第 3 章。

服务器发送的 WWW-Authenticate 响应头中可以指定两种认证方式：BASIC 和 DIGEST。BASIC 验证要求客户端对用户名和密码进行 BASE64 编码后传送给服务器，它属于一种未加密的明文传送方式，网络上的窃听者获取这些信息后，很容易将用户名和密码解码出来。DIGEST 认证方式的细节比较复杂，其基本思想是将用户名和密码以及某些其他信息进行混合后，再执行 MD5 散列算法，并将散列算法的结果和附加信息一起以明文文本通过网络发送到服务器。即使这些信息被网络上的窃听者获取，也几乎不可能将密码从散列结果中破译出来。

本节将编写一个 Servlet 程序来分析 BASIC 验证的全部过程，其中涉及 BASE64 编码技术。由于 BASE64 的应用非常普遍，在本书中的其他多处也要使用到 BASE64 编码技术，读者在以后学习其他技术时也经常会遇到 BASE64 编码的问题，所以下面先详细介绍一下 BASE64 编码。

在某些系统中，只允许使用英文的 ASCII 码字符。例如，由于每个 ASCII 码字符只占用一个字节（8 个 bit 位），且最高比特位总为 0，即 ASCII 码字符中的有真正意义的信息只是后面的 7 个低比特位，所以，一些系统只取出每个字节中的 7 个低比特位进行处理。还有，由于所有 ASCII 码字符的 Unicode 码仅仅是在其 ASCII 码的基础上增加一个所有比特位都为 0 的字节，所以，将 Java 中的 Unicode 编码的字符转换成 ASCII 编码时仅仅需要高位字节去掉即可。显然，如果在这些只能处理英文 ASCII 码字符的系统中出现了非 ASCII 码字符的数据，例如中文字符或二进制数据，就会出现严重的问题。如果要在只允许使用英文 ASCII 码字符的系统中处理非 ASCII 字符的数据，必须先采用某种编码方式将它们“编码”成可打印的 ASCII 字符后再进行处理。注意，不要混淆这里的“编码”与字符集编码，这里的“编码”是指采用某种转换规则对数据进行转换，BASE64 就是将二进制数据转换成可打印的 ASCII 字符的一种最常见的编码方式。

BASE64 的编码规则很简单，它将一组连续的字节数据按 6 个 bit 位进行分组，然后对每组数据用一个 ASCII 字符来表示。6 个 bit 位最多能表示 $2^6=64$ 个数值，因此可以使用 64 个 ASCII 字符来对应这 64 个数值，这 64 个 ASCII 字符为：

```
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
```

其中每个字符表示的数值就是该字符在上面的排列中的索引号，索引号从 0 开始编号。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

假设在内存中有如下三个连续的字节数据：

```
[0110,0001] [0110,0010] [0110,0011]
```

将它们按 6 个 bit 位进行分组后的形式如下：

```
[0110,00] [01,0110,] [0010,01] [10,0011]
```

分组后得到了四组数据，每组数据对应的十进制数值分别为 24，22，9，35，它们分别对应 Y，W，J，j 这四个字符，所以，对[0110,0001] [0110,0010] [0110,0011]这三个字节的数据进行 BASE64 编码后的结果是“YWJj”。

BASE64 编码要求把 3 个 8 位字节（即 24 个 bit）的数据转化为 4 个 6 位字节（也是 24 个 bit）的数据，如果原来的 8 位字节数据的字节个数不能被 3 整除，其余数只能是 1 或 2，那么如何对余下的 1 个或 2 个 8 位字节数据进行处理呢？对于这种情况，仍然按 6 个 bit 位对剩余的字节进行分组，在最后不够 6 个 bit 位的内容后面添加几个为 0 的 bit 位来凑成 6 个 bit 位，例如，如果最后剩下的一个 8 位字节的内容如下：

```
[0110,0001]
```

对它进行分组后的结果如下：

[0110,00] [01,***0000***]

其中用黑斜体标识的 0 为填充的 bit 位，所以，最后剩下的这个字节的 BASE64 编码结果为“YQ”。BASE64 编码还有规定，如果编码后的整个结果文本的字符个数不是 4 的整数倍，那么需要在最后填充“=”字符来凑成 4 的倍数，所以，在最后这个字节编码的结果后面还要添加两个“=”字符，即“YQ==”。显然，如果最后剩下两个 8 位字节的内容，它可以被编码成三个字符，最后还需要添加一个“=”字符。经过 BASE64 编码后的结果所占用的字节个数大约是原始内容的 4/3 倍，计算公式为：编码后的大小 = ((原始大小+2)/3)*4，这里的除号表示整除。

一切二进制数据类型的对象在程序的底层都是一片连续的字节，所以，byte 类型的数组自然就成了存储一片连续二进制数据的最理想的结构，BASE64 编码前的数据类型当然也不会例外，它们在程序中也是用一个 byte 类型的数组来表示的。SUN 公司在 JDK 的核心包中提供两个分别用于进行 BASE64 编码和解码的类：sun.misc.BASE64Encoder 和 sun.misc.BASE64Decoder，其中 BASE64Encoder.encode 方法用于将传入的 byte[]型数据编码成 String 类型后返回，BASE64Decoder.decodeBuffer 方法则用于将传入的 String 类型的数据(BASE64 编码的结果)解码成 byte[]型数据返回。用 WinRAR 程序打开 JDK 中的 rt.Jar 包，进入与包名 sun.misc 对应的目录中，就可以看到 BASE64Encoder.class 和 BASE64 Decoder.class 这两个类文件。因为人们只要掌握一般的编程技巧，就可以对 BASE64 编码的数据进行解码，所以，“采用 BASE64 编码对数据进行加密”的这种常见说法显然是错误的，“BASE64 编码的数据是一种未加密的明文数据”才是正确的说法。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

BASIC 验证的交互过程

01 按例程 6-9 编写一个名为 AuthenticateServlet 的 Servlet 程序，在其 service 方法中检查请求消息中是否包含 Authorization 头字段，如果没有的话，服务器发送 401 响应状态码和 BASIC 认证的 WWW-Authenticate 响应头给客户端，如果包含 Authorization 头字段，则从中解码出用户名和密码进行验证，并根据验证结果给客户端返回不同的信息。为了分析 BASIC 验证的全部过程，程序还在命令行窗口中打印出了请求消息中的所有头字段。

例程 6-9 AuthenticateServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Enumeration;
import sun.misc.BASE64Decoder;

public class AuthenticateServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out = response.getWriter();

        //打印出所有头字段
        Enumeration headerNames = request.getHeaderNames();
        System.out.println("------begin------");
        while(headerNames.hasMoreElements())
        {
            String headerName = (String)headerNames.nextElement();
            System.out.println(headerName +": " + request.getHeader(header
            Name));
        }
        System.out.println("------end------");

        String encodedAuth = request.getHeader("Authorization");
        //要求客户端发送身份认证信息,并且只能是 BASIC 认证方式中
        if (encodedAuth == null || !encodedAuth.toUpperCase().startsWith
        ("BASIC"))
        {
            response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
            response.setHeader("WWW-Authenticate","BASIC realm=\"it315\
            "");
            //当用户单击登录框中的“取消”按钮时，将输出下面的内容
            out.println("没有传递用户身份!");
            return ;
        }

        BASE64Decoder decoder = new BASE64Decoder();
        byte [] decodedBytes = decoder.decodeBuffer(encodedAuth.substring
        (6));
        String decodedInfo = new String(decodedBytes);
        int idx = decodedInfo.indexOf(":");
        if (idx < 0)
        {
            out.println("信息格式不完整");
            return ;
        }
        String user = decodedInfo.substring(0, idx);
        String password = decodedInfo.substring(idx + 1);
        //简单地校验用户名和密码，真实应用中可能涉及数据库查询
        if ("zxx".equals(user) && "123456".equals(password))
        {
            out.println("这就是你要看的信息！");
        }
        else
        {
            out.println("你无权访问此信息！");
        }
    }
}
```

编译 AuthenticateServlet. java 文件，确保编译后生成的 class 文件存放在了<tomcat 的安装目录>\webapps\it315\Web-INF\classes 目录中。

02 在<tomcat 的安装目录>\webapps\it315\Web-INF\web.xml 文件中注册该 Servlet，并设置其映射 URL。在 web.xml 文件中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>AuthenticateServlet</servlet-name>
    <servlet-class>AuthenticateServlet</servlet-class>
</servlet>
……
<servlet-mapping>
    <servlet-name>AuthenticateServlet</servlet-name>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
    <url-pattern>/servlet/AuthenticateServlet</url-pattern>
</servlet-mapping>
```

保存 web.xml 文件后，重新启动 Tomcat。

03 在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/servlet/AuthenticateServlet
```

浏览器中将弹出如图 6.8 所示的对话框，要求用户输入用户名和密码。

查看 Tomcat 命令行窗口，可以看到命令行窗口中打印出的请求头列表中没有 Authorization 头字段，Serlvet 程序因此发出了 401 响应状态码和 WWW-Authenticate 响应头，浏览器收到 401 响应状态码和 WWW-Authenticate 响应头后就弹出如图 6.8 所示的对话框。

04 在图 6.8 所示的对话框中输入用户名“zxx”和密码“123456”后，单击“确定”按钮，浏览器中显示的结果如图 6.9 所示。

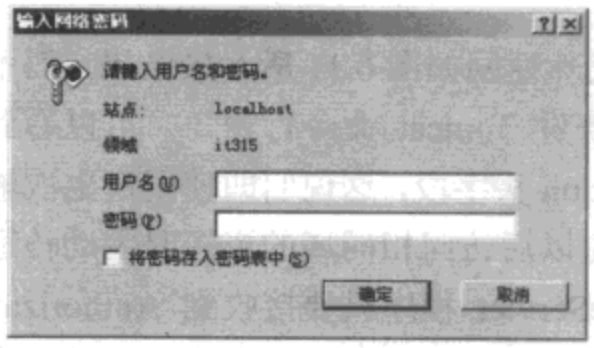


图 6.6

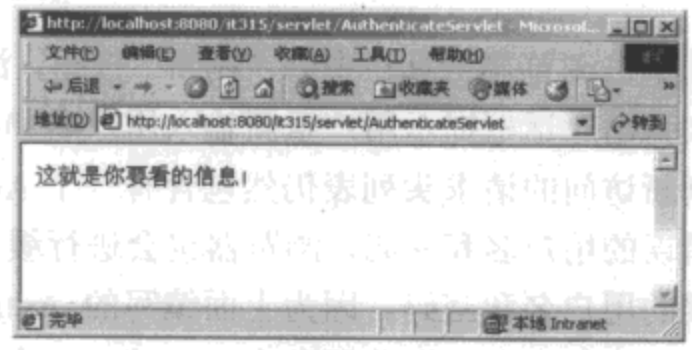


图 6.6

查看 Tomcat 命令行窗口，可以看到浏览器又向 Servlet 程序发出了一次访问请求，并且这次打印出的请求头列表中包含有一个 Authorization 头字段，其值为浏览器对用户名和密码进行 BASE64 编码后的结果，如图 6.10 所示。

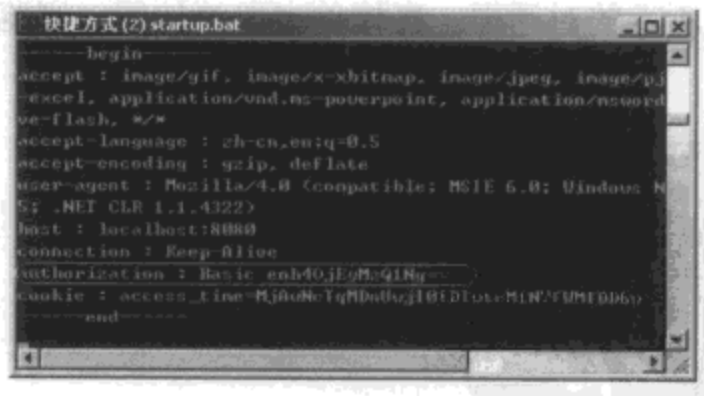


图 6.10

05 单击浏览器工具栏中的“刷新”按钮，查看 Tomcat 命令行窗口，可以看到浏览器又向 Servlet 程序发出了一次访问请求，这次打印出的请求头列表中仍然包含有一个 Authorization 头字段，这是因为浏览器在内存中保存了用户上次输入的用户名和密码，并且在以后访问相同域的资源时都会主动使用 Authorization 头字段将用户名和密码传递给服务器，这样服务器端程序就可以直接从中获取用户名和密码进行校验，而不必让浏览器再

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

次提示用户输入用户名和密码。显然，如果用户想在以后访问相同域的其他资源时改用其他的账户，那么就必须关闭当前的浏览器程序，重新启动一个浏览器程序来进行访问。

06 关闭当前浏览器程序，重新启动一个浏览器程序再次访问 AuthenticateServlet，浏览器又弹出了如图 6.8 所示的对话框。输入一个错误的用户名和密码后，单击“确定”按钮，浏览器中显示的结果如图 6.11 所示。

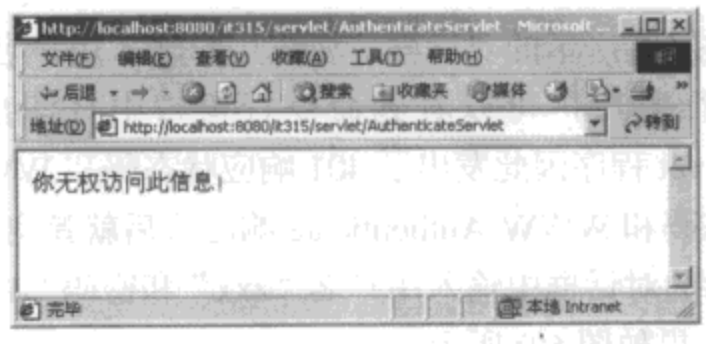


图 6.11

单击浏览器工具栏中的“刷新”按钮，浏览器仍然返回如图 6.11 所示的结果，而不是再次弹出对话框提示用户重新输入用户名和密码。查看 Tomcat 命令行窗口，可以看到这次刷新访问的请求头列表仍然包含有一个 Authorization 头字段，这说明即使用户上次输入了错误的用户名和密码，浏览器也会进行缓存，并在以后访问相同域的资源时主动传递该错误的用户名和密码。因为上面编写的 AuthenticateServlet 程序只要接收到 Authorization 头字段后就不会向浏览器发出 401 响应状态码和 WWW-Authenticate 响应头，而是返回用户身份验证失败的结果信息，所以，浏览器再也不会弹出提示输入用户名和密码的对话框，而是显示用户身份验证失败的信息。针对这种情况，用户只能关闭当前的浏览器程序，并启动一个新的程序来进行后续访问，否则，用户将再也没有输入正确的用户名和密码的机会了。

07 关闭当前浏览器程序，重新启动一个浏览器程序再次访问 AuthenticateServlet，浏览器又弹出了如图 6.8 所示的对话框。直接单击取消按钮，浏览器中将显示如图 6.12 所示的结果。

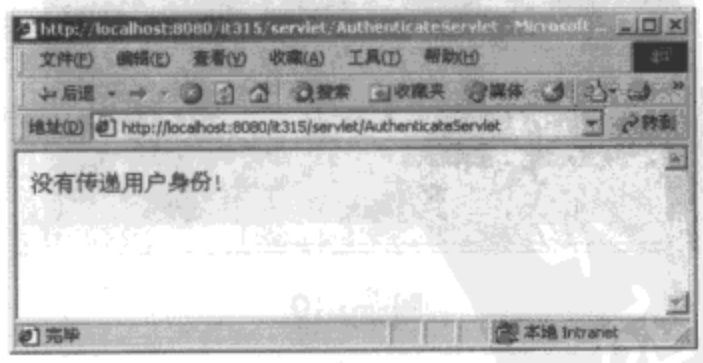


图 6.12

这说明服务器程序在向浏览器发送 401 响应状态码和 WWW-Authenticate 响应头时，也可以发送实体内容，当用户放弃输入用户名和密码时，浏览器将显示出该实体内容。如果在 AuthenticateServlet 程序中向客户端发送 401 响应状态码和 WWW-Authenticate 响应头

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

后，没有向客户端输出任何实体内容，Tomcat 容器将产生一段默认的实体内容传送给浏览器。注释掉程序中 response.setHeader("WWW-Authenticate")语句后的 out.println("没有传递用户身份!")；语句，重新编译 AuthenticateServlet 程序并确保其被 Web 应用程序重新加载后，再次执行本步骤的操作，浏览器中的显示结果如图 6.13 所示。

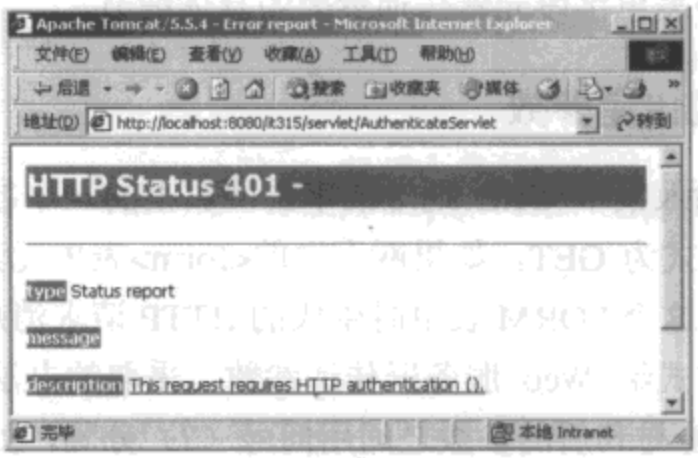


图 6.13

08 如果 AuthenticateServlet 程序在每次接收到 Authorization 头字段并对其进行用户身份验证失败后，在向客户端返回用户身份验证失败的结果信息之前，也向浏览器发出 401 响应状态码和 WWW-Authenticate 响应头，即将程序中的语句：

```
out.println("你无权访问此信息！");
```

修改成如下三句：

```
response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
response.setHeader("WWW-Authenticate","BASIC realm=\"it315\"");
out.println("你无权访问此信息！");
```

那么，在用户输入错误的用户名和密码后，AuthenticateServlet 程序将通知浏览器继续弹出提示输入用户名和密码的对话框，这样，用户就可以在身份验证失败后重新输入用户名和密码。尽管 AuthenticateServlet 程序允许用户无限制次数地传递错误的身份信息，但是浏览器自身最多允许用户输入三次用户名和密码，然后就不再继续弹出提示输入用户名和密码的对话框了，而是直接显示服务器端传送的实体内容（即上面的 out.println 语句输出的“你无权访问此信息!”）。

6.5 浏览器传递参数信息的细节

HttpServletRequest（或 ServletRequest）对象的最基本和最广泛应用就是获取浏览器传递给 Web 服务器的参数信息，这些参数信息既可以是 HTTP 请求消息的请求行中的 URL 地址后的附加信息，又可以是 POST 方式下的“application/x-www-form-urlencoded”编码格式的实体内容。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

在一个 Web 应用中，经常需要通过网页上的各种 FORM 表单字段元素来向服务器程序传递参数信息，不同的表单字段元素传递参数的方式各不一样，同一个表单字段元素在不同状态下的参数传递情况也不一样。对于一个表单字段元素的名称和值，在有些情况下会传递给服务器程序，在有些情况却不会传递给服务器程序。对于要深入掌握 Web 开发的人员来说，必须了解各种表单字段元素传递参数的具体细节。

6.5.1 使用 GET 方式传递参数

在浏览器地址栏中输入某个 URL 地址或单击网页上的一个超链接时，浏览器发出的 HTTP 请求消息的请求方式为 GET。如果网页中的<form>表单元素的 method 属性被设置为了“GET”，浏览器提交这个 FORM 表单时生成的 HTTP 请求消息的请求方式也为 GET。如果要使用 GET 请求方式给 Web 服务器传递参数，采取的办法就是将这些参数附加在 URL 地址后面，URL 地址与其后的参数之间用问号（?）分隔，如下所示：

```
http://www.it315.org/counter.jsp?name=zhangsan&password=123
```

各个参数之间用（&）分隔，每个参数的名称与设置值之间用等号（=）分隔。只有参数名、没有设置值的情况也是允许的，但参数名后面也要跟上等号，如“name=”。对于上面的这个带参数的 URL 地址，浏览器发出的 HTTP 请求消息的请求行如下：

```
GET /counter.jsp?name=zhangsan&password=123  HTTP/1.1
```

由于 URL 的极限长度为 1 024 个字节，所以使用 GET 方式传送的数据量是有限制的，一般限制在 1KB 以下。如果数据超过了 1 024 个字节，那么就不能使用 GET 方式，而应该使用 POST 方式。

对于 URL 中的参数部分的特殊字符必须进行 URL 编码后才能发送给 Web 服务器，关于 URL 编码的细节，请参看本书作者的《JavaScript 网页开发》一书。如果一个 FORM 表单的提交方式为 GET，当提交 FORM 表单时，表单字段元素的名称和值将进行 URL 编码后，再以参数的形式附加到目标 URL 后面。

不要使用 GET 方式提交包含口令的 FORM 表单

务必不要使用 GET 方式来提交询问口令的表单。如果这样做了，则访问者输入的口令将作为 URL 的一部分存储在好几个地方，其中包括 Web 浏览器的历史记录文件和 Web 服务器的日志文件。

6.5.2 使用 POST 方式传递参数

HTTP 协议的 POST 请求方式，主要用于向 Web 服务器端程序提交 FORM 表单中的数

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

据。当使用 POST 方式来提交 FORM 表单中的数据时，浏览器将各个表单字段元素及其数据作为 HTTP 消息的实体内容发送给 Web 服务器，而不是作为 URL 地址的参数传递给 Web 服务器，因此，使用 POST 方式传送的数据量要比使用 GET 方式传送的数据量大得多，并且在浏览器地址栏中也看不到 POST 方式提交的数据。网页中的<form>表单元素的 enctype 属性用于指定浏览器使用哪种编码方法将表单中的数据传送给 Web 服务器，该属性可以有两种取值：

- application/x-www-form-urlencoded
- multipart/form-data

enctype 属性默认的设置值为“application/x-www-form-urlencoded”，即不设置<form>元素的 Enctype 属性与将 Enctype 属性显式设置为“application/x-www-form-urlencoded”的效果一样。在这种情况下，浏览器在提交 FORM 表单数据时，将每个表单字段元素的名称与设置值之间用等号（=）分隔，形成一个参数，并对其中的特殊字符进行 URL 编码处理。各个参数之间用（&）分隔后作为 HTTP 请求消息的实体内容。使用 POST 方式的 HTTP 请求消息具有类似下面的形式：

```
POST /counter.jsp HTTP/1.1
accept: */*
referer: http://localhost:8080/Register.html
accept-language: zh-cn,en-gb;q=0.5
content-type: application/x-www-form-urlencoded
user-agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
host: localhost:8080
content-length: 43
connection: Keep-Alive
cache-control: no-cache

name=zhangsan&password=123
```

注意上面的 Content-Type 和 Content-Length 请求头的设置，Content-Length 的设置值必须等于实体内容的长度。对于采用 POST 请求方式的 HTTP 请求消息，HttpServletRequest 接口中定义的 getParameter 等方法只能够从“application/x-www-form-urlencoded”编码类型的实体内容中将表单字段元素的信息当做参数提取出来，而不能从“multipart/form-data”编码类型的实体内容中提取表单字段元素的信息。只有将<form>元素的 method 属性设置为 POST 方式时，其 enctype 属性才能设置为“multipart/form-data”，这种情况下的数据格式比较复杂，并且只有使用<form>表单向 Web 服务器上传文件时才需要如此设置，在本章稍后部分将对此进行详细讲解。

根据 HTML 标准，如果处理表单的服务器程序不会改变 Web 服务器上存储的数据，则应采用 GET 方式，例如，用来对数据库进行查询的表单通常都使用 GET 方式。反之，

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

如果处理表单的结果会引起服务器上存储的数据发生变化，譬如，将用户的注册信息存储到数据库中，则应采用 POST 方式。

6.5.3 提交、重置、普通按钮的参数传递

了解提交、重置、普通按钮的参数传递细节

01 在<tomcat 的安装目录>\webapps\it315 目录下编写一个名为 TestButton.html 的网页文件，如例程 6-10 所示。

例程 6-10 TestButton.html

```
<form action="" method="GET">
<input type="button" name="button1" value="普通按钮"><br>
<input type="submit" name="submit1" value="提交按钮 1">
<input type="submit" name="submit1" value="提交按钮 2">
<input type="submit" value="匿名提交按钮">
<input type="reset" name="reset1" value="重置按钮">
</form>
```

02 访问 http://localhost:8080/it315/TestButton.html 后，单击页面上的“普通按钮”和“重置按钮”，浏览器地址栏中的路径信息没有发生任何改变，这说明单击“重置按钮”和“普通按钮”都不会提交表单。

03 单击“提交按钮 1”，浏览器地址栏中出现的路径如下：

```
http://localhost:8080/it315/TestButton.html?submit1=%CC%E1%BD%BB%B0%B4%C5%A51
```

注意最后的字符是“1”。

04 单击“提交按钮 2”，浏览器地址栏中出现的路径如下：

```
http://localhost:8080/it315/TestButton.html?submit1=%CC%E1%BD%BB%B0%B4%C5%A52
```

注意最后的字符是“2”。

05 单击“匿名提交按钮”，浏览器地址栏中出现的路径如下：

```
http://localhost:8080/it315/TestButton.html?
```

上面的实验说明：

（1）在一个 FORM 表单中可以有多个提交按钮，单击任何一个提交按钮都可以提交表单，只有被单击的提交按钮的名称和值才被作为参数传递，其他提交按钮的信息不会作为参数传递。如果希望用多种提交方式来提交同一个 FORM 表单，可以为这个 FORM 表单

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

设置多个同名但设置值各不相同的提交按钮，服务器端的处理程序可以根据获取到的提交按钮的值来判断用户单击的是哪个提交按钮，从而采用相应的方式进行处理。

（2）表单中的普通按钮与重置按钮的名称和值不会作为参数传递。

（3）没有设置 name 属性的表单字段元素的信息也不会作为参数传递，尽管单击“匿名提交按钮”也可以提交表单，但由于没有设置它的 name 属性，这个按钮本身的信息不会作为参数传递。

6.5.4 单行与多行文本输入框如何传送参数

了解单行与多行文本输入框的参数传递细节

01 在<tomcat 的安装目录>\webapps\it315 目录下编写一个名为 TestText.html 的网页文件，如例程 6-11 所示。

例程 6-11 TestText.html

```
<form action="" method="GET">
<input type="text" name="text1" value="value1"><br>
<textarea name="textarea1" clos="20" rows="5">
中国
</textarea><br>
<input type="submit">
</form>
```

02 访问 http://localhost:8080/it315/TestText.html 后，单击页面上的提交按钮，浏览器地址栏中出现的路径如下：

```
http://localhost:8080/it315/TestText.html?text1=value1&textarea1=%D6%D0
%B9%FA%0D%0A
```

03 将两个文本输入框中的内容清空后，再次单击提交按钮，浏览器地址栏中出现的路径如下：

```
http://localhost:8080/it315/TestText.html?text1=&textarea1=
```

上面的实验说明：不管单行和多行文本输入框中是否有内容，设置了 name 属性的文本输入框的信息总会作为参数传递。如果文本框中没有输入内容，可以认为其内容为一个空字符串（""），这时紧跟字段名称后面的等号（=）后面仍然有值，只不过此值是一个空字符串，所以最后的参数形式为“text1=”。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

6.5.5 单选按钮与复选框如何传送参数

了解单选按钮与复选框的参数传递细节

01 在<tomcat的安装目录>\webapps\it315目录下编写一个名为TestCheck.html的网页文件，如例程6-12所示。

例程 6-12 TestCheck.html

```
<form action="" method="GET">
<input type="checkbox" name="checkbox1" value="1">checkbox1:1
<input type="checkbox" name="checkbox1" value="2">checkbox1:2
<input type="checkbox" name="checkbox2">checkbox2<br>
<input type="radio" name="radio1" value="1">radio1:1
<input type="radio" name="radio1" value="2">radio1:2
<input type="radio" name="radio2">radio2<br>
<input type="submit">
</form>
```

02 访问 http://localhost:8080/it315/TestCheck.html 后，单击页面上的提交按钮，浏览器地址栏中出现的路径如下：

```
http://localhost:8080/it315/TestCheck.html?
```

03 将上面的每个复选框和单选按钮都单击一次，可以看到页面上所有的复选框都被选中，名称为radio1的两个单选按钮只有第二个被选中，名称为radio2的单选按钮被选中，单击提交按钮，浏览器地址栏中出现的路径如下：

```
http://localhost:8080/it315/TestCheck.html?checkbox1=1&checkbox1=2&checkbox2=on&radio1=1&radio2=on
```

上面的实验说明：

（1）只有被选中的复选框和单选按钮的信息才会作为参数传递，未被选中的复选框和单选按钮的信息不会作为参数传递，就像表单中没有这些字段元素的情况一样。

（2）对于多个名称相同的复选框，它们可以同时被选中；对于多个名称相同的单选按钮，只能同时选中其中的任意一个。

（3）对于被选中的多个同名复选框，它们的信息将以多个名称相同的参数进行传递，即参数列表中会出现多个名称相同的参数。

（4）对于没有设置value属性的单选按钮和复选框，当它们被选中时，它们传递的默认参数值为“on”。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

6.5.6 隐藏表单字段与图像字段如何传送参数

在 FORM 表单中可以使用隐藏表单字段元素向服务器传递参数信息，浏览器在网页上不会显示出隐藏表单字段元素，但当提交表单时，浏览器会将这个隐藏表单字段元素的 name 属性和 value 属性作为参数传递给 Web 服务器。隐藏表单字段元素在实际应用中非常重要，例如，一个网站的用户注册过程由两个步骤完成，每个步骤对应一个表单页面，用户在第一步的表单页面中填写了自己的姓名后，接着进入第二步的页面中填写爱好和特长等信息。在提交第二个页面的表单信息时，需要将第一个页面中收集到的用户名称也传送给 Web 服务器，这就需要在第二个网页的表单中加入一个隐藏表单字段元素，让它的 value 属性值等于第一个网页中收集到的用户名。另外，有一些网页要通过其中的 JavaScript 代码动态产生一些信息后，再把这些信息提交给 Web 服务器，通常的做法就是将这些信息作为一个 FORM 表单的隐藏字段进行预设，然后由 JavaScript 代码对它们进行修改后再提交表单，这些信息就可随着 FORM 表单提交给 Web 服务器。

在 FORM 表单中除了可以使用 submit 按钮提交表单外，还可以使用图像字段元素提交表单。

了解隐藏表单字段元素与图像字段元素的参数传递细节

01 在<tomcat 的安装目录>\webapps\it315 目录下编写一个名为 TestHidden.html 的网页文件，如例程 6-13 所示。

例程 6-13 TestHidden.html

```
<form action="" method="GET">
<input type="hidden" name="hidden1" value="hidden1">
<input type="image" name="image1" src="http://www.it315.org/imgs/logo.jpg">
</form>
```

02 访问 http://localhost:8080/it315/TestHidden.html 后，单击页面上的图像字段元素，浏览器地址栏中出现的路径如下：

```
http://localhost:8080/it315/TestHidden.html?hidden1=hidden1&image1.x=19
&image1.y=19
```

这个实验说明：

（1）隐藏表单字段元素总是被作为参数传递给 Web 服务器。如果要在提交 FORM 表单时将网页中的 JavaScript 程序获得的信息或者上次的 Web 服务器端程序产生的标记信息传送给 Web 服务器，使用隐藏字段就是一个很好的办法。

（2）图像字段元素也可以提交 FORM 表单，用户单击图像字段元素后，表单中的其他

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

信息和图像上单击处的 x、y 坐标一起传送给服务器，x 坐标对应的参数名是图像元素名后加上.x（如 image1.x），y 坐标对应的参数名是图像元素名后加上.y（如 image1.y）。

6.5.7 列表框的参数传递

了解列表框的参数传递细节

01 在<tomcat 的安装目录>\webapps\it315 目录下编写一个名为 TestSelect.html 的网页文件，如例程 6-14 所示。

例程 6-14 TestSelect.html

```
<form action="" method="GET">
<select name="select1" multiple>
    <option value="">--不选--</potion>
    <option >java</option>
    <option value="1">c</option>
    <option value="2">jsp</option>
</select><br>
<input type="submit">
</form>
```

02 访问 http://localhost:8080/it315/TestSelect.html 后，单击页面上的提交按钮，浏览器地址栏中出现的路径如下：

```
http://localhost:8080/it315/TestSelect.html?
```

03 按下 Ctrl 键，选中列表框中的 java、c、jsp 这三个选项后，再单击页面上的提交按钮，浏览器地址栏中出现的路径如下：

```
http://localhost:8080/it315/TestSelect.html?select1=java&select1=1&sele
ct1=2
```

这个实验说明：

（1）没有做出选择的列表框的信息不会作为参数传递，就像表单中没有这个列表框的情况一样。

（2）如果设置了列表框字段元素的 multiple 属性，可以选择列表中的多个选项。对于选中的每个选项，它们都会与列表框的名称分别组合成单独的参数后进行传递，这样，参数列表中会出现多个名称为列表框名的同名参数。

（3）当选中没有设置 value 属性的列表选项时，浏览器使用该选项的标题作为参数值，

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

例如选中上面列表框中的 java 选项时，它传递的参数值为 java。

6.5.8 使用 JavaScript 防止重复提交表单

有时候，由于服务器端负荷过重或网络拥挤而出现响应缓慢的情况，用户提交 FORM 表单后没有立即看到服务器端的响应，他们很可能会怀疑提交没有成功而再次单击“提交”按钮，这将发生同一份表单的重复提交。网站开发人员必须想办法来禁止这种表单重复提交的问题，特别是在金融应用场合，重复提交同一笔交易是绝对不应该出现的。其实，只要利用客户端的 JavaScript 程序，就可以防止用户的这种非故意的表单重复提交问题。

使用 JavaScript 防止重复表单提交

01 按例程 6-15 和例程 6-16 编写一个用于接收 FORM 表单提交信息的 Servlet 程序和一个用于提交 FORM 表单的 HTML 页面，在 Servlet 程序中刻意延时 6 秒钟来模拟服务器端响应缓慢的情况。

例程 6-15 RepeateForm.html

```
<h1>测试表单重复提交的页面</h1>
<form action="servlet/RepeateFormServlet" method="POST">
留言内容: <textarea name="note"></textarea><br>
<input type="submit" value="提交">
</form>
```

例程 6-16 RepeateFormServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class RepeateFormServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        response.setContentType("text/html;charset=GB2312");
        PrintWriter out = response.getWriter();
        try
        {
            Thread.sleep(6000);
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        }catch(Exception e){}
        out.println("已处理你提交的信息!");
        System.out.println("已处理你提交的信息!");
    }
}
```

编译 RepeateFormServlet. java 文件，确保编译后生成的 class 文件存放在了<tomcat 的安装目录>\webapps\it315\Web-INF\classes 目录中。将 UserRegister.html 文件保存在<tomcat 的安装目录>\webapps\it315 目录中。

02 在<tomcat 的安装目录>\webapps\it315\Web-INF\web.xml 文件中注册该 Servlet，并设置其映射 URL。在 web.xml 文件中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>RepeateFormServlet</servlet-name>
    <servlet-class>RepeateFormServlet</servlet-class>
</servlet>
……
<servlet-mapping>
    <servlet-name>RepeateFormServlet</servlet-name>
    <url-pattern>/servlet/RepeateFormServlet</url-pattern>
</servlet-mapping>
```

保存 web.xml 文件后，重新启动 Tomcat。

03 在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/RepeateForm.html
```

单击浏览器显示页面中的“提交”按钮，由于服务器端的响应有一些延时，仍然可以继续单击“提交”按钮几次，最后浏览器窗口中显示出了 RepeateFormServlet 针对最后一次请求的响应结果。进入启动 Tomcat 的命令行窗口，看到其中打印出了多条处理提示信息，这说明尽管浏览器窗口中只显示一个响应结果，但 RepeateFormServlet 实际上对每一次提交请求都进行了处理。每一个浏览器窗口最多只与 Web 服务器保持一个并发的 TCP 连接。如果服务器针对上一次请求的响应还没有结束时，用户再次单击了“提交”按钮，浏览器将不等待服务器的上次响应结果而关闭为上次请求建立的与服务器的连接，然后再与 Web 服务器建立新的连接和提交新的请求。尽管客户端已经中断了上次请求的连接，但 Servlet 程序仍然会忠实地处理完上次的请求，只是响应内容无法返回给客户端。

04 修改 RepeateForm.html 文件，在其中增加防止表单重复提交的 JavaScript 程序代码，如例程 6-17 所示。

例程 6-17 RepeateForm.html

```
<h1>测试表单重复提交的页面</h1>
<script>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
var isCommitted = false;
function checkPost()
{
    if(!isCommitted)
    {
        isCommitted = true;
        return true;
    }
    else
    {
        alert("不能重复提交表单");
        return false;
    }
}
</script>
<form action="servlet/RepeateFormServlet" method="POST"
        onsubmit="return checkPost()">
留言内容：<textarea name="note"></textarea><br>
<input type="submit" value="提交">
</form>
```

05 在浏览器中重新访问 http://localhost:8080/it315/RepeateForm.html 页面，单击其中的“提交”按钮后，在服务器端的响应延时过程中虽然可以继续单击“提交”按钮，但 JavaScript 程序阻止了将后续的请求提交到服务器。

本程序主要是用于防止用户无意之中发生的表单重复提交问题，但是在浏览器窗口中显示出 RepeateFormServlet 的响应结果后，用户单击浏览器工具栏中的“刷新”按钮或“后退”按钮后，仍然可以继续向 RepeateFormServlet 提交与上次相同的请求信息，这些行为显然是用户刻意或希望进行的操作，不属于系统要屏蔽的功能。

6.6 获取请求参数

ServletRequest 和 HttpServletRequest 接口中定义了 getParameter 等方法，这些方法除了能读取 HTTP 请求消息的请求行中的 URL 地址后的附加信息外，还能读取 POST 方式下的“application/x-www-form-urlencoded”编码格式的实体内容。getParameter 等方法内部能够自动完成参数的分解和提取，并对提取的参数值进行 URL 解码后返回。

6.6.1 getParameter 方法

getParameter 方法用于返回某个指定名称的参数的值，其完整的语法定义如下：

```
public java.lang.String getParameter(java.lang.String name)
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

如果请求消息中没有包含指定名称的参数，getParameter 方法返回 null；如果指定名称的参数存在但没有设置值，则返回一个空串；如果请求消息中包含有多个该指定名称的参数，getParameter 方法返回其中的第一个出现的参数值，所以，getParameter 方法通常只用于获取在 HTTP 请求消息中仅出现一次的参数。由于参数名是大小写敏感的，因此，request.getParameter（"Param1"）和 request.getParameter ("param1"）分别用于返回两个不同的参数，不能认为两者等同。

6.6.2 getParameterValues 方法

HTTP 请求消息中可以有多个相同名称的参数（通常由一个包含了多个同名的字段元素的 FORM 表单生成），如果要获得 HTTP 请求消息中的同一个参数名所对应的所有参数值，那么就应该使用 getParameterValues 方法。getParameterValues 方法用于获取某个指定名称的所有参数的值，并以一个 String 数组的形式返回这些参数值，其完整的语法定义如下：

```
public java.lang.String[] getParameterValues(java.lang.String name)
```

如果指定的参数名在 HTTP 请求消息中只出现了一次，返回的数组的长度为 1，其中包含的惟一一个元素值与 getParameter 方法返回的值一样；如果指定名称的参数不存在，则返回值为 null。

6.6.3 getParameterNames 方法

getParameterNames 方法用于返回一个包含请求消息中的所有参数名的 Enumeration 对象，在此基础上，可以对请求消息中的所有参数进行遍历处理。getParameterNames 方法的完整语法定义如下：

```
public java.util.Enumeration getParameterNames()
```

6.6.4 getParameterMap 方法

getParameterMap 方法用于将请求消息中的所有参数名和值装入进一个 Map 对象中返回，其语法定义如下：

```
public java.util.Map getParameterMap()
```

6.6.5 获取请求参数的编程实例

获取请求消息中的所有参数

01 编写一个名为 RequestParamsServlet 的 Servlet 程序，在其 doGet 和 doPost 方法中打印出请求消息中的所有参数信息，如例程 6-18 所示。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

例程 6-18 RequestParamsServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Enumeration;

public class RequestParamsServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        response.setContentType("text/html ;charset=gb2312");
        PrintWriter out = response.getWriter();
        Enumeration paramNames = request.getParameterNames();
        while(paramNames.hasMoreElements())
        {
            String paramName = (String)paramNames.nextElement();
            out.print(paramName + " : " + request.getParameter(paramName)
            + "<br>");
            /*如果要考虑同一个请求头名可能出现多次,
            那么应该用下面的代码段代替上面一行程序代码*/
            /*String [] paramValues = request.getParameterValues
            (paramName);
            //良好的编程习惯，在使用对象和数组之前先判断其是否为null
            if(paramValues != null)
            {
                for(int i=0;i<paramValues.length;i++)
                {
                    out.print(paramName + " : " + (String)paramValues[i]
                    + "<br>");
                }
            }*/
        }
    }

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

编译 RequestParamsServlet. java 文件，确保编译后生成的 class 文件存放在了<tomcat

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

的安装目录>\webapps\it315\Web-INF\classes 目录中。

02 在<tomcat 的安装目录>\webapps\it315\Web-INF\web.xml 文件中注册该 Servlet，并设置其映射 URL。在 web.xml 文件中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>RequestParamsServlet</servlet-name>
    <servlet-class>RequestParamsServlet</servlet-class>
</servlet>
……
<servlet-mapping>
    <servlet-name>RequestParamsServlet</servlet-name>
    <url-pattern>/servlet/RequestParamsServlet</url-pattern>
</servlet-mapping>
```

保存 web.xml 文件后，重新启动 Tomcat。

03 在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/servlet/RequestParamsServlet?p1=a+b&p2=b&p1=x
```

浏览器中显示的结果如图 6.14 所示。

图 6.14

其中，返回的“p1”参数值是字符串“a+b”被 URL 解码后的结果，即“a b”。第二个名称为“p1”的参数没有显示出来，这是因为 getParameter 方法只返回同名参数中的第一个的参数值。如果改用 RequestParamsServlet.java 文件中被注释掉的那段程序代码，就可以获得所有的同名参数。

04 在<tomcat 的安装目录>\webapps\it315 目录中编写一个名称为 ParamsForm.html 的网页文件，内容如下：

```
<h1>使用表单提交请求参数的页面</h1>
<form action="servlet/RequestParamsServlet" method="POST">
第一个 p1 参数：<input type="text" name="p1" value="a b"><br>
惟一的 p2 参数：<input type="text" name="p2" value="b"><br>
第二个 p1 参数：<input type="text" name="p1" value="x"><br>
<input type="submit" value="提交">
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
</form>
```

在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/ParamsForm.html
```

单击返回页面中的提交按钮，浏览器中显示的结果与步骤 **03** 中的完全一样。可见，不管客户端使用 GET 方式，还是 POST 方式向服务器端传递参数，在 Servlet 程序中获取参数值的方式完全一样。

6.6.6 一次性处理多个列表项的实用案例

在许多基于 Web 的条目列表管理程序中，例如，邮件列表、留言列表、用户名单列表，产品列表等等，都需要对多个列表项进行一次性操作的批量处理，操作界面如图 6.15 所示。

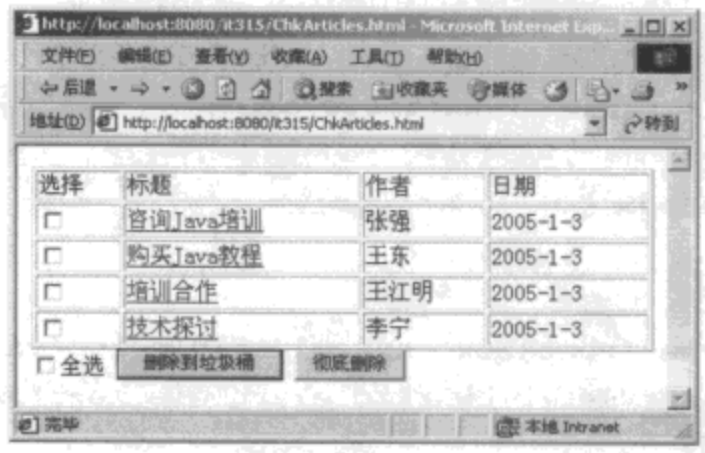


图 6.15

在图 6.15 中，每条列表项左边都有一个复选框，用户通过复选框来选择所要删除的列表项。如果同时选中了多个列表项后单击“删除”提交按钮，Web 服务器端程序将删除所有被选中的列表项。如果为图 6.15 中的每个列表项的复选框都设置各自不同的名称，服务器端程序要逐一检索每个复选框的选择情况，就会比较复杂和烦琐。另外，图 6.15 所示的页面中显示的列表项数目及相关内容通常都是由服务器程序从数据库中动态查询出来并生成的，由于数据库中的记录会有增减，所以列表项的数目是不确定的。如果为每个列表项关联的复选框都设置不同的名称，这也会给服务器端程序的编写带来很大难度。如果为每个列表项关联的复选框都设置一个完全相同的名称，那么在服务器端的 Servlet 程序中，只要使用 ServletRequest.getParameterValues 方法就可以返回所有被选中的列表项，这显然是一种非常可取的方式。

一次性处理多个列表项

01 在<tomcat 的安装目录>\webapps\it315 目录中编写一个用于产生图 6.15 所示页

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

面的 HTML 文件，如例程 6-19 所示。

例程 6-19 ChkArticles.html

```
<form action="servlet/DelArticleServlet" method="POST">
<table border=1 width="100%">
<tr>
    <td>选择</td><td>标题</td><td>作者</td><td>日期</td>
</tr>
<tr>
    <td><input type="checkbox" name="article_id" value="1"></td>
    <td><a href="servlet/viewArticle?article_id=1">咨询 Java 培训</a></td>
    <td>张强</td>
    <td>2005-1-3</td>
</tr>
<tr>
    <td><input type="checkbox" name="article_id" value="2"></td>
    <td><a href="servlet/viewArticle?article_id=2">购买 Java 教程</a></td>
    <td>王东</td>
    <td>2005-1-3</td>
</tr>
<tr>
    <td><input type="checkbox" name="article_id" value="3"></td>
    <td><a href="servlet/viewArticle?article_id=3">培训合作</a></td>
    <td>王江明</td>
    <td>2005-1-3</td>
</tr>
<tr>
    <td><input type="checkbox" name="article_id" value="4"></td>
    <td><a href="servlet/viewArticle?article_id=4">技术探讨</a></td>
    <td>李宁</td>
    <td>2005-1-3</td>
</tr>
</table>
<input type="checkbox" name="selAll">全选
<input type="submit" name="del" value="删除到垃圾桶">
<input type="submit" name="del" value="彻底删除">
</form>
```

02 编写一个名为 DelArticleServlet 的 Servlet 程序，负责处理 ChkArticles.html 页面提交的请求，如例程 6-20 所示。

例程 6-20 DelArticleServlet.java

```
import java.io.*;
import javax.servlet.*;
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


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

public class DelArticleServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        response.setContentType("text/html;charset=gb2312");
        request.setCharacterEncoding("gb2312");
        PrintWriter out = response.getWriter();
        String [] paramValues = request.getParameterValues("article_id");
        if(paramValues != null)
        {
            //将多个参数用逗号连接起来
            StringBuffer params = new StringBuffer(paramValues[0]);
            for(int i=1;i<paramValues.length;i++)
            {
                params.append(",");
                params.append(paramValues[i]);
            }
            String sql = "";
            if("彻底删除".equals(request.getParameter("del")))
            {
                sql = "delete from lyb_Articles where ArticleId in (" +
                    params + ")";
            }
            else if("删除到垃圾桶".equals(request.getParameter("del")))
            {
                sql = "update lyb_Articles set IsDel=1 where ArticleId in
                (" +
                    params + ")";
            }
            out.println("向数据库系统发送如下 sql 命令: <br>");
            out.println(sql);
        }
        else
        {
            out.println("没有选择要删除的帖子! ");
        }
    }

    public void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
            doGet(request,response);
        }
    }
```

编译 DelArticleServlet. java 文件，确保编译后生成的 class 文件存放在了<tomcat 的安装目录>\webapps\it315\Web-INF\classes 目录中。

03 在<tomcat 的安装目录>\webapps\it315\Web-INF\web.xml 文件中注册该 Servlet，并设置其映射 URL。在 web.xml 文件中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>DelArticleServlet</servlet-name>
    <servlet-class>DelArticleServlet</servlet-class>
</servlet>
……
<servlet-mapping>
    <servlet-name>DelArticleServlet</servlet-name>
    <url-pattern>/servlet/DelArticleServlet</url-pattern>
</servlet-mapping>
```

保存 web.xml 文件后，重新启动 Tomcat。

04 在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/ChkArticles.html
```

在浏览器显示的图 6.15 中选择第 2 项和第 4 项，接着执行“永久删除”操作后，浏览器返回的结果如图 6.16 所示。

图 6.16

05 在图 6.15 中不选择任何列表项，执行“永久删除”和“删除到垃圾桶”操作，查看运行结果。选择若干列表项后，再分别执行“永久删除”和“删除到垃圾桶”操作，查看其运行结果。在 ChkArticles.html 文件中并没有实现“全选”复选框的功能，它可以通过一段 JavaScript 脚本程序实现，感兴趣的读者可以参看本书作者的《JavaScript 网页开发》一书。本例子程序只是介绍如何实现一次性删除多个列表项的情况，而没有实现查看各个列表项的功能（即 ChkArticles.html 文件中指向的 ViewArticle 这个 Servlet 程序）。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

6.6.7 单选列表框的处理经验

在使用<select></select>标签创建的下拉列表框中，用户一旦选择了列表框中的某一选项后，虽然还可以改选其他选项，但再也不能让列表框回到不选择任何选项的状态了。如果要让用户在已做出过选择后，还可以表达出不想选择列表框中的任何一个选项的意图，则应将一个空值的选项作为列表中的第一个条目，并提示用户诸如“--不选--”这样的文本信息。这样，如果用户从列表中选择了第一个条目，那么在发送给 Web 服务器的表单结果中，列表框的 name/value 信息对中 value 的值将为一个空字符串，服务器程序可以据此认为用户不想进行任何选择。应用这种设计思想的 HTML 语句类似如下：

```
<select name="subject">
   <option value="">--不选--</potion>
   <option value="0">java</option>
   <option value="1">c++</option>
   <option value="2">jsp</option>
</select>
```

在对上面的列表框进行处理的 Servlet 程序中，通常使用类似如下的代码语句：

```
String subject = request.getParameter ("subject");
//如果没有选择或是选择了“—不选—”项
if(subject == null || "".equals(subject))
{
//这里是用户没有选择时的处理语句
}
else
{
//这里是用户进行了选择时的处理语句
}
```

6.7 获取请求消息的实体内容

以 POST 方式提交 HTML 的 FORM 表单时，其中的表单字段元素信息都将作为 HTTP 消息的实体内容（消息正文）发送给 Web 服务器。对于 HTTP 请求消息中的实体内容，ServletRequest 以输入流的方式提供给 Servlet 程序读取，ServletRequest 接口中定义了一个 getInputStream 方法来返回这个字节输入流对象。为了方便读取纯文本形式的实体内容，ServletRequest 接口中也定义了一个 getReader 方法，以返回一个代表实体内容的字符输入流对象。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

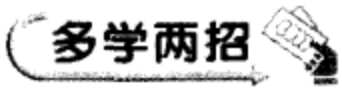
http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

6.7.1 getInputStream 与 getReader 方法

getInputStream 方法用于返回一个代表实体内容的输入流对象，其类型为 javax.servlet.ServletInputStream。使用这个输入流对象可以读取 HTTP 请求消息中的实体内容，如果实体内容中包含二进制数据，那么就只能使用 getInputStream 方法返回的输入流对象来读取。

getReader 方法用于返回一个代表实体内容的 BufferedReader 对象，返回的 BufferedReader 对象将实体内容中的字节数据按照请求消息中指定的字符集编码转换成文本字符串。在调用 getReader 方法之前，可以调用 ServletRequest 的 setCharacterEncoding 方法指定其返回的 BufferedReader 对象所使用的字符集编码。如果请求消息中没有指定实体内容的字符集编码，并且在调用 getReader 方法之前也没有调用 ServletRequest 的 setCharacterEncoding 方法，那么，getReader 方法返回的 BufferedReader 对象将使用 ISO8859-1 作为其默认的字符集编码。

调用了 ServletRequest 对象的 getReader 或 getInputStream 中的任何一个方法后，就不能再调用另一方法，这两个方法是相互排斥的。虽然 Web 容器能够在 Servlet 的 service 方法返回后自动关闭 getReader 或 getInputStream 返回的流对象，但是，最好是在 Servlet 程序读取完实体内容后，接着在其内部调用流对象的 close 方法，以便服务器尽快释放相关的资源。



使用 ServletInputStream 对象读取文本格式的实体内容

虽然使用 ServletInputStream 对象也能读取请求消息中的文本格式的实体内容，但是，如果在 Servlet 程序中要以文本形式来处理这些实体内容，则需要将 ServletInputStream 对象读取到的字节数组数据转换成字符文本。所以，当请求消息中的实体内容全部是文本数据，并且在 Servlet 程序中也需要以文本方式来处理这些内容时，那么最好是调用 getReader 方法返回 BufferedReader 对象来简化程序的编写，而不要调用 getInputStream 方法返回 ServletInputStream 对象。如果请求消息中的实体内容全部是文本数据，但是 Servlet 程序仅仅是需要将它们读取出来，然后再原封不动地存储到文件系统或数据库中，那么就应该使用 ServletInputStream 对象直接进行读取。

6.7.2 分析文件上传的请求消息结构

分析文件上传的请求消息结构

01 按例程 6-21 编写一个名为 ReadBodyServlet 的 Servlet 程序，在其 doPost 方法

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

调用 HttpServletRequest 的 getInputStream 方法返回的 ServletInputStream 对象读取 HTTP 请求消息中的实体内容，然后将读取到的实体内容保存到 Web 应用程序根目录下的 body.out 文件中。

例程 6-21 ReadBodyServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ReadBodyServlet extends HttpServlet
{
    public void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        ServletInputStream sis = request.getInputStream();
        String filePath = getServletContext().getRealPath("/body.out");
        FileOutputStream fos = new FileOutputStream(filePath);
        byte[] buf = new byte[1024];
        int len = sis.read(buf,0,1024);
        while(len != -1)
        {
            fos.write(buf,0,len);
            len = sis.read(buf,0,1024);
        }
        fos.close();
        sis.close();
    }
}
```

在上面的程序代码中，由于没有获取实体内容的长度，只好先定义一个 1 024 个字节的数组去读取实体内容。InputStream 的 read 方法的返回值表示实际读取到的内容的字节数，当返回值等于–1 时，则表示已读取到了输入流中的结尾处。当返回值不等于–1 时，只能说明已从输入流中读取到了内容，但不能说明是否已读取到了输入流中的所有内容，还需要继续调用 InputStream 的 read 方法，周而复始，直到返回值等于–1 时为止。

编译 ReadBodyServlet.java 文件，确保编译后生成的 class 文件存放在了<tomcat 的安装目录>\webapps\it315\Web-INF\classes 目录中。

02 在<tomcat 的安装目录>\webapps\it315\Web-INF\web.xml 文件中注册该 Servlet，并设置其映射 URL。在 web.xml 文件中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>ReadBodyServlet</servlet-name>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
    <servlet-class>ReadBodyServlet</servlet-class>
</servlet>
……
<servlet-mapping>
    <servlet-name>ReadBodyServlet</servlet-name>
    <url-pattern>/servlet/ReadBodyServlet</url-pattern>
</servlet-mapping>
```

保存 web.xml 文件后，重新启动 Tomcat。

03 在<tomcat 的安装目录>\webapps\it315 目录中编写一个名称为 TestBody.html 的网页文件，其中包含一个用于产生实体内容的<form>表单，表单递交的目的地是 ReadBody Servlet，如例程 6-22 所示。

例程 6-22 TestBody.html

```
<h1>了解表单如何创建请求消息的实体内容的页面</h1>
<form action="servlet/ReadBodyServlet" method="post">
    请填写第一个参数：<input type="text" name="p1"><br>
    请填写第二个参数：<input type="text" name="p2"><br>
    请选择上传的文件：<input type="file" name="f1"><br>
    <input type="submit" value="提交">
</form>
```

其中，<input type="file">这个表单字段元素是为了对表单的功能进行了扩展而在 RFC 1867 规范中增加的，以便 Web 浏览器通过 HTML 的 FORM 表单向 Web 服务器上传文件。使用<input type="file">元素，浏览器会自动生成一个文本输入框和一个“浏览...”按钮，供用户选择上传到服务器的文件。另外，由于没有设置 FORM 表单的 enctype 属性值，浏览器将使用其默认值“application/x-www-form-urlencoded”。

04 在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/TestBody.html
```

填写浏览器返回页面中的表单，如图 6.17 所示。

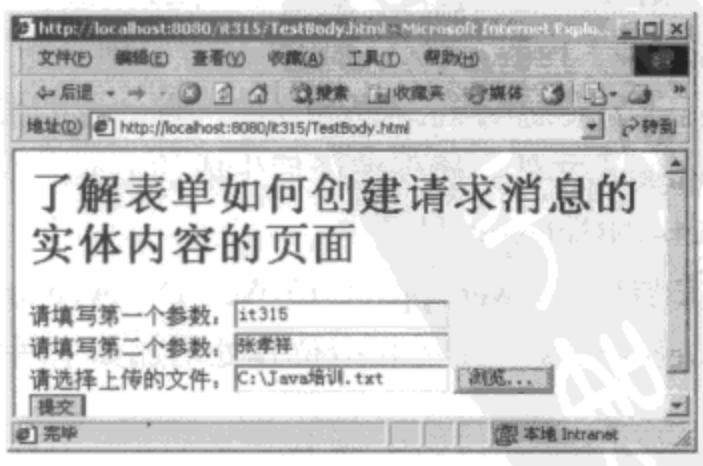


图 6.17

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

单击图 6.17 中的“提交”按钮后，进入<tomcat 的安装目录>\webapps\it315 目录中，可以看到其中新创建了一个名为 body.out 的文件。用记事本或 UltraEdit 程序打开 body.out 文件，结果如图 6.18 所示。

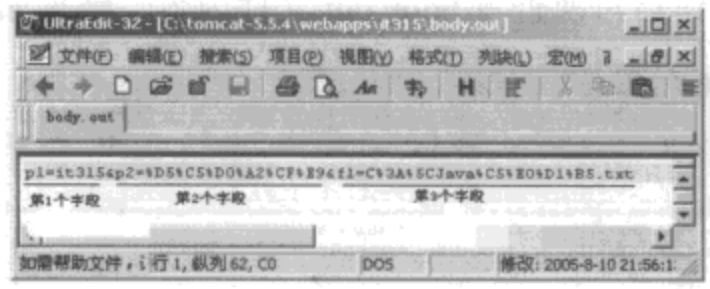


图 6.18

从图 6.18 中可以看到，对于以“application/x-www-form-urlencoded”编码的 FORM 表单，所有字段按 URL 参数的形式组合成了一行，并且对其中的参数值都被进行了 URL 编码处理。要想从图 6.18 的数据流中提取出各个字段的信息，必须对读取到的所有信息进行拆分，然后再进行 URL 解码。由于 HttpServletResponse 接口中定义 getParameter 等方法可以直接读取图 6.18 中的各个表单字段元素的信息，并且得到的参数值为 URL 解码以后的结果，所以，在 Servlet 程序中一般都不用 getInputStream 或 getReader 方法返回输入流对象的方式去读取“application/x-www-form-urlencoded”编码的表单字段信息。

05 从图 6.18 中还可以看到，浏览器只是将<input type="file">元素中填写的文件名传送给了 Web 服务器，而没有将文件内容传送给 Web 服务器。显然，将 FORM 表单的 enctype 属性设置为“application/x-www-form-urlencoded”时，无法将客户端的文件内容上传到 Web 服务器。

如果要上传文件内容，<form>元素的 enctype 属性必须设置为 multipart/form-data，并且 method 属性必须设置为 POST。修改 TestBody.html 文件，将其中的<form>元素的 enctype 属性设置为“multipart/form-data”，使其能够向 Web 服务器上传文件内容，如例程 6-23 所示。

例程 6-23 TestBody.html

```
<h1>测试表单的例子页面</h1>
<form action="servlet/ReadBodyServlet"
        method="post" enctype="multipart/form-data">
    请填写第一个参数：<input type="text" name="p1"><br>
    请填写第二个参数：<input type="text" name="p2"><br>
    请选择上传的文件：<input type="file" name="f1"><br>
    <input type="submit" value="提交">
</form>
```

06 使用浏览器中重新访问 http://localhost:8080/it315/TestBody.html 页面，最好是刷新一下这个页面。为了保险起见，克服人们的一种习惯的假设错觉（大多数人很自然地认

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

为浏览器刷新后，其中显示的内容肯定就是修改过的网页，而忽略网页修改后没存盘的情况），最好还单击 IE 浏览器上的“查看”→“源文件”菜单，查看其中的内容，确保浏览器中显示的网页内容是更新后的内容，以免仍使用旧的网页内容。按图 6.17 所示的内容填写表单，单击“提交”按钮后再次查看 body.out 文件内容，结果如图 6.19 所示。

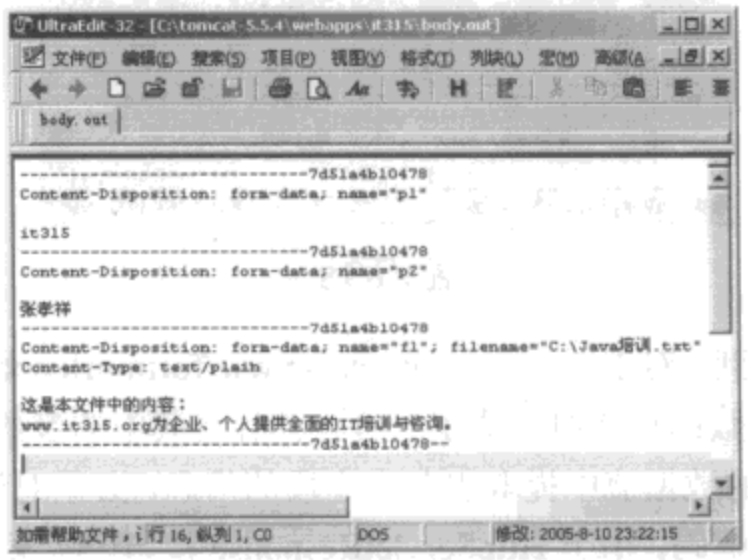


图 6.19

从图 6.19 中可以看到，将 FORM 表单的 enctype 属性设置为“multipart/form-data”后，浏览器将<input type="file">元素中填写的文件名及文件内容都传送给了 Web 服务器。FORM 表单中的各个字段元素之间以一个连续的字符序列作为字段分隔界线，字段分隔界线的字符序列通常都是由一连串的减号（-）字符后跟上一个随机产生的十六进制数字序列组成的，例如，----------1878979834。这种分隔界线就是 MIME 分隔界线，浏览器负责保证实体内容中的其他数据不会形成与分隔界线相同的字符序列。字段元素名与其取值之间用空行分隔，最后一行的内容是在组成字段分隔界线的字符序列的后面再添加两个减号（-）字符。从插入符光标最后的停留位置上可以看出最后一行的内容后面也有回车换行符，当然，更直观的方式是按下 UltraEdit 工具栏上的“H”样的按钮，用十六进制方式查看 body.out 文件中的内容，可以看到最后两个字节的内容为“OD OA”。

在两个分隔界线之间的内容就是一个表单字段元素的信息。除了第一个分隔界线外，所有其他分隔界线前面都有一个回车换行符；除了最后一个分隔界线外，所有其他分隔界线后面都有一个回车换行符。这些回车换行符不属于表单字段元素的信息，在编程提取每个表单字段元素的信息时必须加以考虑。

每个表单字段元素的信息可以分为描述头和主体内容两部分，描述头与主体内容之间有一空行，即描述头后面紧跟两个连续的回车换行符后就是主体内容部分。非文件类型的字段元素的主体内容部分就是表单元素的设置值，文件类型的字段元素的主体内容部分就是文件的内容。非文件类型的字段元素的描述头部分包含有字段元素的名称，格式如下：

```
Content-Disposition: form-data; name="FieldName"
```

其中，FieldName 为表单字段元素的名称，例如：

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
Content-Disposition: form-data; name="p1"
```

文件类型的字段元素的描述头部分除了包含有字段元素的名称外，还包含有上传的文件名称和文件类型，格式如下：

```
Content-Disposition: form-data; name="FieldName"; filename="FileName"
Content-Type: MIME-TYPE
```

其中，FieldName 为表单字段元素的名称，FileName 为上传文件的名称，MIME-TYPE 为上传文件内容的 MIME 类型，filename 与 Content-Type 部分之间用一回车换行符分隔，例如：

```
Content-Disposition: form-data; name="f1"; filename="c:\Java 培训.txt"
Content-Type: text/plain
```

注意 Windows 浏览器会将 FileName 部分设置为上传文件的完整路径名，而 UNIX/Linux 以及 Mac 浏览器只将 FileName 部分设置为上传文件的文件名。

HttpServletRequest 的 getParameter 方法无法读取以“multipart/form-data”编码的 FORM 表单中的各个字段元素的信息，对于“multipart/form-data”编码的 FORM 表单传递的信息，只能调用 HttpServletRequest 的 getInputStream 或 getReader 方法返回的输入流对象进行读取后再提取各个字段元素的信息。

07 用记事本程序将 TestBody.html 另存为 UTF-8 编码格式，然后重复第 **06** 步的操作，用 UltraEdit 查看 body.out 文件内容，结果如图 6.20 所示。

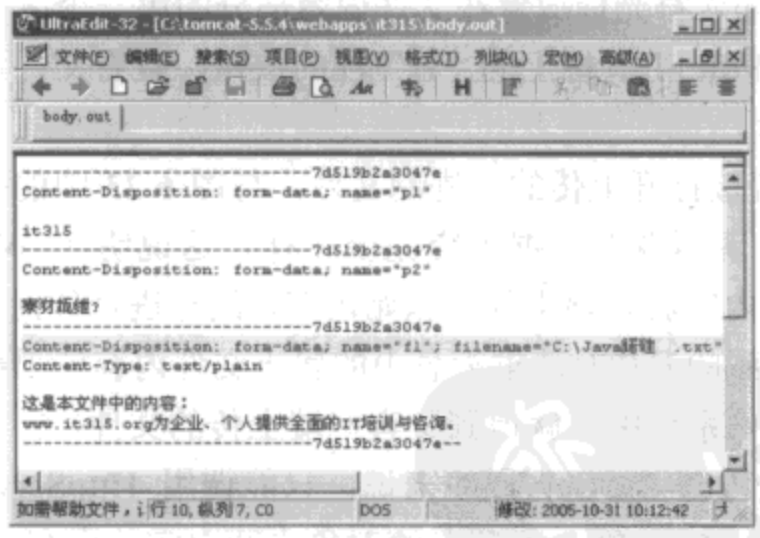


图 6.20

从图 6.20 中可以看到，FORM 表单字段元素中填写的中文字符部分都显示为乱码，这是由于 TestBody.html 页面的当前字符集编码为 UTF-8，浏览器会将其中的 FORM 表单中填写的内容（包括文件字段中填写的文件名）采用 UTF-8 编码后传送给 Web 服务器，而 UltraEdit 无法正常显示 UTF-8 编码的中文字符。对于上传的文件内容部分，则不会受到 TestBody.html 页面所采用的字符集编码的影响，浏览器将其原封不动地上传给 Web 服务

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

器。要想验证图 6.20 中显示的乱码是不是 FORM 表单中填写的中文字符的 UTF-8 编码，可以单击 UltraEdit 的“文件”→“转换”→“UTF-8 转 ASCII”菜单，结果如图 6.21 所示，原来的乱码部分都显示为 FORM 表单中填写的中文字符了，这说明那些乱码确实是这些字符的 UTF-8 编码。

08 上面上传的是一个文本格式的文件，下面接着看看上传一个二进制文件时的情况。单击浏览器工具栏中的“后退”按钮，回到 http://localhost:8080/it315/TestBody.html 页面，在 FORM 表单的文件字段中选择一个较小（几十个字节）的图片文件，单击“提交”按钮后再次查看 body.out 文件内容，结果如图 6.22 所示。

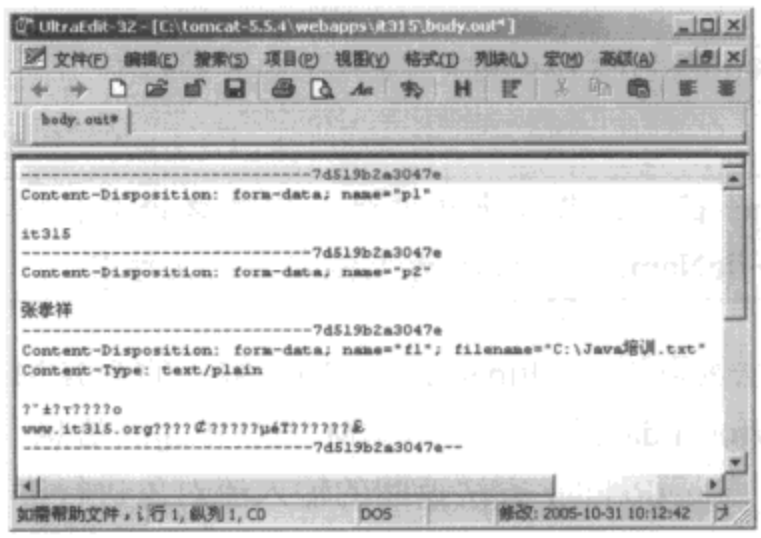


图 6.21

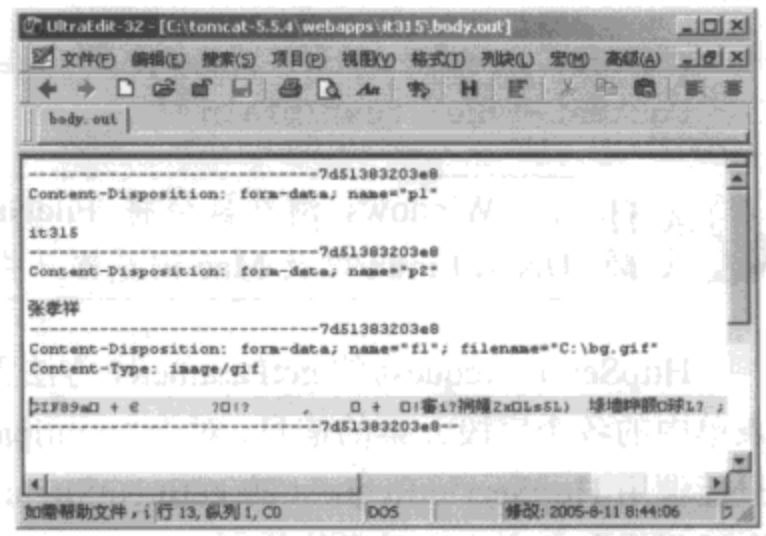


图 6.22

尽管上传文件中的二进制数据显示为乱码，但是，我们还是很容易分辨出哪一部分属于上传文件的内容。将插入符光标放置在文件内容部分的第一个字节前面（如图 6.22 所示的状态），再按下 UltraEdit 工具栏上的“H”样的按钮，查看文件内容对应的十六进制数据部分，如图 6.23 所示。

接着用 UltraEdit 直接打开上传的二进制文件，与图 6.23 中的属于文件内容的那部分数据进行比较，可以发现两者的数据内容完全一样。这说明在 Web 服务器端接收到的 FORM 表单上传的文件内容就是该文件的原始内容，没有进行任何编码转换处理，服务器端的程序只需要将接收到的 HTTP 请求消息中的属于文件内容的那部分数据原封不动地提取出来并存储到本地文件系统中，就得到了客户端上传的文件。

对于要处理文件上传的 Servlet 程序来说，只能调用 HttpServletRequest 的 getInputStream 方法返回的 ServletInputStream 字节流对象去读取实体内容，而不能调用 getReader 方法返回的 BufferedReader 字符流对象去读取实体内容，因为从 ServletInputStream 流对象中读取的数据是每个字节的原始内容，而从 BufferedReader 流对象中读取的数据是原始的字节内容按某种编码转换成的字符文本，不是原始的数据内容。

09 如果用户没有在 FORM 表单的文件字段中选择任何文件，那么上传后的结果是怎样的呢？单击浏览器工具栏中的“后退”按钮，回到 http://localhost:8080/it315/TestBody.html 页面，在 FORM 表单中不填写任何内容，直接单击“提交”按钮后再次查看

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

body.out 文件内容，结果如图 6.24 所示。

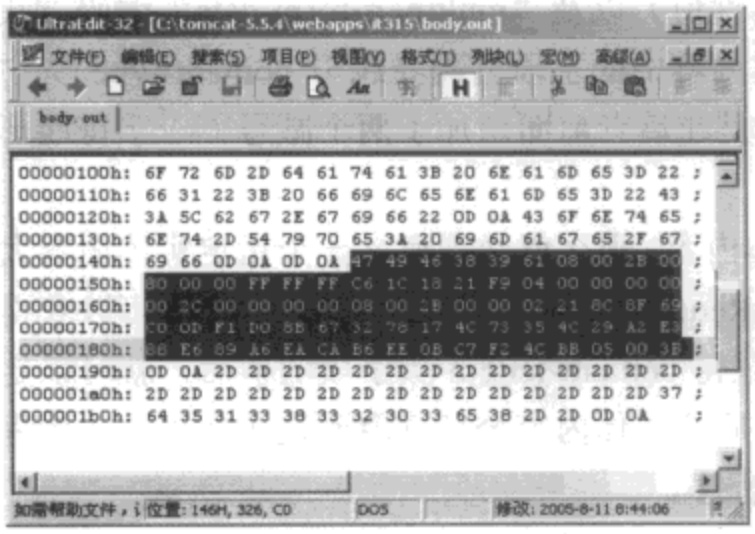


图 6.23

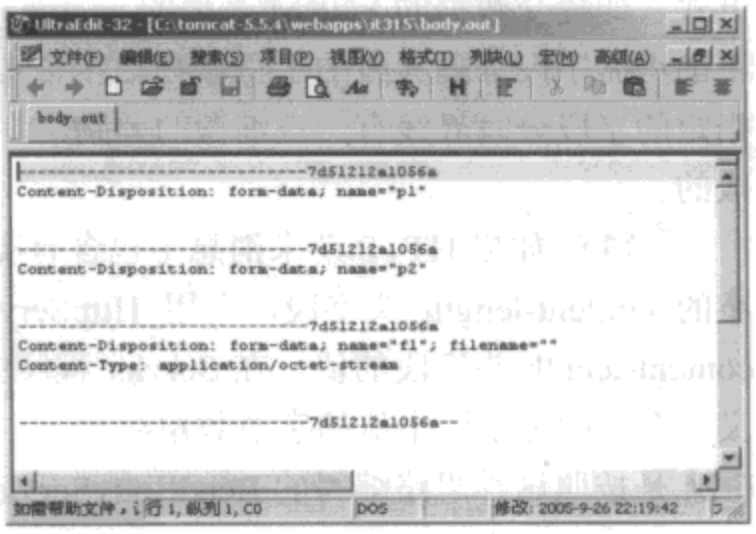


图 6.24

可见，即使用户没有通过表单中的文件字段传递任何文件，但只要设置了其 name 属性，浏览器也会将文件字段的信息传递给服务器，只是文件名和文件内容部分都为空字符串。

10 将 FORM 表单的 enctype 属性设置为“multipart/form-data”后，浏览器发送的 HTTP 请求消息中的请求头部分有什么变化呢？本章前面已经编写了一个可读取 HTTP 请求消息中的所有头字段的 RequestHeadersServlet，只要将 TestBody.html 文件中的 FORM 表单的 action 属性设置为“servlet/RequestHeadersServlet”就可以看到结果。按此说明修改 TestBody.html 文件后，在浏览器中重新访问 http://localhost:8080/it315/TestBody.html 页面，填写并提交表单内容后，浏览器中的显示结果如图 6.25 所示。

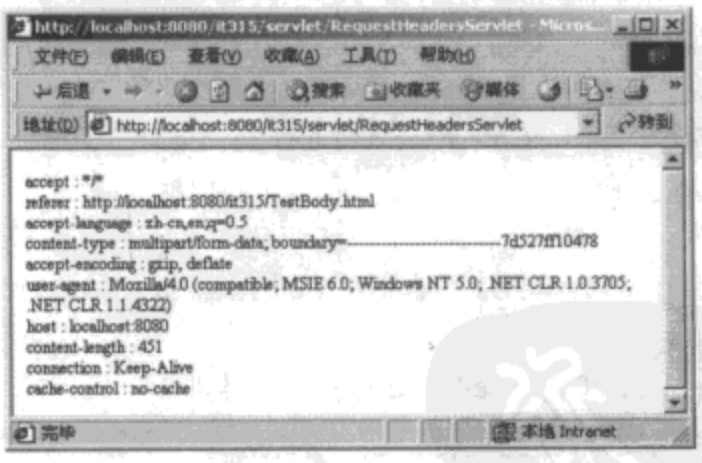


图 6.25

对于提取 FORM 表单中的各个字段信息和上传的文件内容的程序来说，图 6.25 中有两个比较重要的请求头，它们是 content-length 和 content-type。content-length 头字段指定了实体内容的总长度，content-type 头字段中指定了用做字段分隔界线的字符序列，其格式如下：

```
content-type: multipart/form-data; boundary=用做字段分隔界线的字符序列
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

服务器端程序可以根据 content-type 头字段检测出 HTTP 请求消息的实体内容的编码方式（即提交数据的 FORM 表单的 enctype 属性值）以及“multipart/form-data”编码方式下的用做字段分隔界线的字符序列，注意实体内容中的字段分隔界线与 content-type 头中指定的字段分隔界线有一点细微的差别，前者是在后者前面增加了两个减号（-）字符而形成的。

11 如果 HTTP 请求消息中包含有实体内容，那么就必须包含一个指定实体内容大小的 content-length 头字段，调用 HttpServletRequest 的 getContentLength 方法可以获得 content-length 头字段的值。在 Servlet 程序中获得 content-length 头字段的值后，就可以定义一个长度等于这个值的字节数组缓冲区，然后将实体内容全部读取到这个数组中，例程 6-24 是按照这种思路编写的 ReadBodyServlet 程序。

例程 6-24 ReadBodyServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ReadBodyServlet extends HttpServlet
{
    public void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        ServletInputStream sis = request.getInputStream();
        String filePath = getServletContext().getRealPath("/body.out");
        FileOutputStream fos = new FileOutputStream(filePath);
        int total_len = request.getContentLength();
        byte[] buf = new byte[total_len];
        for(int read_len=0,reading_len=0; read_len < total_len; read_len
        += reading_len)
        {
            reading_len = sis.read(buf,read_len,total_len-read_len);
        }
        fos.write(buf,0,total_len);
        fos.close();
        sis.close();
    }
}
```

在例程 6-24 所示的程序代码中，定义的字节数组缓冲区正好可以容纳下所有的实体内容，那么，只要调用一次 InputStream.read(byte[] b)方法，就应该可以将输入流中所有的数据读取到字节数组缓冲区中，那为什么还要循环调用 read 方法去读取 HTTP 消息中的实体内容呢？这是因为即使 read 方法中指定的要读取的字节个数等于 HTTP 消息的实体内容的

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

长度，但这并不能保证在一次 read 方法调用中就能读取到输入流中的所有内容，因为 read 方法在读取到指定个数的字节后或读取完网卡缓冲区中的内容后就会返回。由于网络传送速度通常要比 CPU 读取网卡缓冲区中的数据的速度慢，所以 read 方法很可能在只读取到实体内容的部分数据后就返回了。针对这种情况，上面的程序要循环调用 read 方法，并调整读取到的数据填充到 buf 数组中的位置和随后所要读取的字节个数，直到读取的数据填满整个 buf 数组为止。尽管用类似上面的程序去读取一个文件输入流中的数据时，一般调用一次 read 方法就可以读取到文件中的所有数据，但为了通用和保险起见，也应使用上面程序中的循环读取方式。

重新编译 ReadBodyServlet.java 文件后，重复第（6）步实验，运行结果与修改前的结果完全一样。但是，如果上传的文件很大（例如，100M 以上），修改前的 ReadBodyServlet 没有任何问题，但修改后的 ReadBodyServlet 将发生如下的异常信息：

```
java.lang.OutOfMemoryError: Java heap space
```

因为上传的文件很大时，修改后的 ReadBodyServlet 中定义的字节数组也会随之变得很大，这将导致 Java 虚拟机发生内存不足的严重错误。这个实验说明，尽管对同一问题可以有多种不同的编程解决方式，但应充分比较各种编程解决方式所造成的隐性后果，在实际项目开发中要充分考虑各种可能遇到的情况，并进行充分的测试。

了解了 FORM 表单的 enctype 属性值等于“multipart/form-data”时的 HTTP 请求消息的结构与数据格式后，就可以通过编程的方式来提取 FORM 表单中的各个字段信息和上传的文件名与文件内容了，在本书后面的章节中将分析开源的 apache 文件上传组件的编程实现。

6.8 利用请求域属性传递信息

ServletRequest 接口中定义了一个 setAttribute 方法来将对象存储进 ServletRequest 对象中，还定义了一个 getAttribute 方法来检索存储在 ServletRequest 对象中的对象。ServletRequest 接口的实现类通常都是采用一个 HashMap 对象来存储这些对象，setAttribute 方法就是向这个 HashMap 对象中增加与某个名称绑定的对象，getAttribute 方法则是根据名称从这个 HashMap 对象中检索对象。这种存储在 ServletRequest 对象中的对象称之为请求域属性，属于同一个请求的多个处理模块之间可以通过请求域属性来传递对象数据。

6.8.1 setAttribute 方法

setAttribute 方法用于将一个对象与一个名称关联后存储进 ServletRequest 对象中，其完整语法定义如下：

```
public void setAttribute(java.lang.String name,java.lang.Object o)
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

传递给 setAttribute 方法的属性名应遵循包名的命名习惯，以 java.*、javax.*和 com.sun.*等为前缀的名称保留给 Sun 公司内部使用，在程序中不要以这些前缀定义属性名。如果 ServletRequest 对象中已经存在指定名称的属性，setAttribute 方法先删除原来的属性后再增加新的属性。如果传递给 setAttribute 方法的属性值对象为 null，则删除指定名称的属性，这时的效果等同于 removeAttribute 方法。

6.8.2 getAttribute 方法

getAttribute 方法用于从 ServletRequest 对象中返回指定名称的属性对象，其完整的语法定义如下：

```
public java.lang.String getParameter(java.lang.String name)
```

6.8.3 removeAttribute 方法

removeAttribute 方法用于从 ServletRequest 对象中删除指定名称的属性，其完整的语法定义如下：

```
public void removeAttribute(java.lang.String name)
```

6.8.4 getAttributeNames 方法

getAttributeNames 方法用于返回一个包含 ServletRequest 对象中的所有属性名的 Enumeration 对象，在此基础上，可以对 ServletRequest 对象中的所有属性进行遍历处理。getAttributeNames 方法的完整语法定义如下：

```
public java.util.Enumeration getAttributeNames()
```

6.8.5 请求域属性的编程实例

目前，一种比较先进和流行的 Web 应用开发技术就是采用 MVC 设计模式。它将一次请求的响应过程分成三个功能模块（一般称之为层）来协同完成，这三个模块分别是 Model（模型层）、View（视图层）、Controller（控制层）。对 Web 应用来说，Model 是可作为 JavaBean 使用的业务对象，例如，一个购物车、一个订单等，它包含业务的状态数据和完成业务逻辑的方法；View 是负责创建显示界面的模块，它通常是一个用于创建网页文档内容的 JSP 页面，JSP 页面创建输出内容时可以从 Model 对象中提取状态数据；Controller 通常是一个接收用户请求的 Servlet 程序，它根据请求创建相应的 Model 对象和调用 Model 对象的业务方法，最后再选择一个 View 去创建网页文档内容并回送给客户端。Controller 选择 View 的编程实现方式就是调用 RequestDispatcher.forward 方法将请求转发给作为 View 的 JSP 页面，同时将 Model 对象作为请求域属性传递过去，作为 View 的 JSP 页面再从请求域中检索出 Model 对象。

下面采用 MVC 设计模式实现一个简单示例程序，这个程序包含如下一些组件：

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

User.java、UserRegister.html、ActionServlet.java、JspResultServlet.java。UserRegister.html 是一个包含有让用户填写注册信息的 FORM 表单的 HTML 页面；User.java 是一个代表用户注册信息的普通 Java 类；ActionServlet.java 是一个用于处理 FORM 表单信息的 Servlet 程序，它根据表单提交的信息创建一个 User 类的实例对象，并把这个 User 实例对象存储到请求域中，然后将请求转发给另外一个用于显示用户注册信息的 Servlet 程序；JspResultServlet.java 是一个用于显示用户注册信息的 Servlet 程序（在实际开发中，这里应该用一个 JSP 页面来显示用户注册信息，由于目前还没有介绍 JSP 技术，这里暂时用一个 Servlet 程序来实现），JspResultServlet 从请求域中取出 User 实例对象，并显示出这个 User 实例对象的信息。

请求域属性的应用

01 按上面描述的功能编写如例程 6-25、例程 6-26、例程 6-27 和例程 6-28 所示的一些程序。

例程 6-25 User.java

```
package org.it315;
public class User
{
    private String name = null;
    private String email = null;

    public void setName(String name)
    {
        this.name = name;
    }
    public String getName()
    {
        return this.name;
    }
    public void setEmail(String email)
    {
        this.email = email;
    }
    public String getEmail()
    {
        return this.email;
    }
}
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

例程 6-26 UserRegister.html

```
<h1>请填写您的注册信息</h1>
<form action="servlet/ActionServlet" method="post">
用户名: <input type="text" name="user"><br>
Email: <input type="text" name="email"><br>
<input type="submit" value="提交">
</form>
```

例程 6-27 ActionServlet.java

```
package org.it315;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ActionServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        /*注意下面传递给 getParameter 方法的参数名要与提交请求的
          FORM 表单中的相应字段元素名严格一致。
        */
        String name = request.getParameter("user");
        String email = request.getParameter("email");

        User user = new User();
        user.setName(name);
        user.setEmail(email);
        /*在实际应用中，这里有更多的业务操作，例如校验用户名和 E-mail 的正确性、
          将用户注册信息保存到数据库，或遇到错误时将请求转发到注册页面让
          用户重新注册。
        */
        request.setAttribute("user",user);
        RequestDispatcher rd = request.getRequestDispatcher("JspResult
        Servlet");
        rd.forward(request,response);
    }
}
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

例程 6-28 JspResultServlet.java

```
package org.it315;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/**这是一个模拟 JSP 产生输出的 Servlet 程序，它里面几乎没有任何业务逻辑处理，
   只负责产生显示给用户的网页文档内容。
*/
public class JspResultServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        response.setContentType("text/html;charset=GB2312");
        PrintWriter out = response.getWriter();
        User user = (User)request.getAttribute("user");
        if(user != null)
        {
            out.println("<h1>你的注册信息如下：</h1>");
            out.println("用户名：" + user.getName() + "<br>");
            out.println("Email: " + user.getEmail() + "<br>");
        }
        out.close();
    }
}
```

编译上面的三个 Java 源文件，确保编译后生成的 class 文件存放在了<tomcat 的安装目录>\webapps\it315\Web-INF\classes 目录中，注意要有与 org.it315 包名对应的目录结构。将 UserRegister.html 文件保存在<tomcat 的安装目录>\webapps\it315 目录中。

02 在<tomcat 的安装目录>\webapps\it315\Web-INF\web.xml 文件中注册有关的 Servlet，并设置其映射 URL。在 web.xml 文件中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>ActionServlet</servlet-name>
    <servlet-class>org.it315.ActionServlet</servlet-class>
</servlet>
<servlet>
    <servlet-name>JspResultServlet</servlet-name>
    <servlet-class>org.it315.JspResultServlet</servlet-class>
</servlet>
……
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
<servlet-mapping>
    <servlet-name>ActionServlet</servlet-name>
    <url-pattern>/servlet/ActionServlet</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>JspResultServlet</servlet-name>
    <url-pattern>/servlet/JspResultServlet</url-pattern>
</servlet-mapping>
```

保存 web.xml 文件后，重新启动 Tomcat。

03 在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/UserRegister.html
```

填写浏览器返回页面中的表单，如图 6.26 所示。

单击图 6.26 中的“提交”按钮后，浏览器中返回的结果如图 6.27 所示。

图 6.26

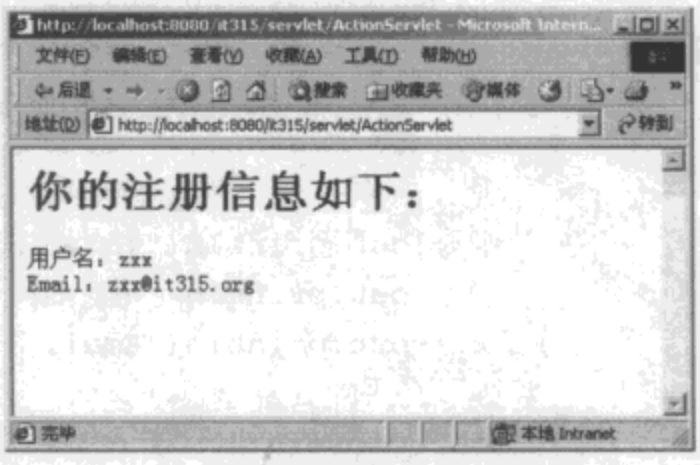


图 6.27

存储在 ServletRequest 对象中的属性只对当前请求有效，而对于其他请求无效，也就是说，请求域属性只能用于在通过 RequestDispatcher 的 forward 和 include 方法贯穿起来的多个 Servlet 或 JSP 页面之间共享数据。

多想一想

请求域属性与 URL 参数的应用区别

尽管通过在 URL 地址后面附加参数的方式也可以在两个 Servlet 之间传递信息，但它只能传递简单的字符文本信息，不能像请求域属性一样传递复杂的对象。

6.9 请求参数的中文读取问题

HTTP 协议规定浏览器向 Web 服务器传递的参数信息中不能出现某些特殊字符，而必

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

须对这些字符进行 URL 编码后再传送。Web 服务器端程序接收到客户端传递的整个参数信息后，首先从中分离出每个参数的名称和值部分，接着对单个的名称和值部分进行 URL 解码，然后将 URL 解码得到的字节数组按照某种字符集编码转换成 Unicode 字符串。

字符在计算机内存中其实是一个数值，URL 编码并不对字符进行编码，而是对代表这个字符的数值进行编码，这个数值具体是多少，就要看字符当前使用的字符集编码了。对于中文来说，目前存在着多种字符集编码，同一个字符在不同的字符集编码下对应不同的数值，譬如，在 GB2312 字符集编码中，“中”对应的是 0xD6D0，占用 2 个字节；而在 UTF-8 字符集编码中，“中”对应的是 0xE4B8AD，占用 3 个字节。

对计算机程序来说，当它看到内存中对应某个字符的数值时，它能否知道这个数值对应的到底是哪个字符呢？如果不告诉计算机程序这是哪种字符集编码的数值，计算机是无法知道数值所表示的字符的。如果本来是某种字符集编码的数值，计算机程序却把它当做了另外一种字符集编码来处理，这样就会产生字符乱码。

6.9.1 了解中文字符的 URL 编码

对 Java 中的某个字符串进行 URL 编码，其实是在对它的某种字符集编码（非 Unicode 码）数据进行 URL 编码，而 Java 中的字符串是以 Unicode 字符集编码存在的，所以，在对 Java 中的字符串进行 URL 编码时，必须指定是对这个字符串的哪种非 Unicode 的字符集编码进行 URL 编码。JDK 的 java.net 包中有两个类：URLEncoder 和 URLDecoder。URLEncoder 类中定义了一个 encode 静态方法来完成 URL 编码；URLDecoder 类中定义了一个 decode 静态方法来完成 URL 解码。它们的完整语法分别定义如下：

```
public static String encode(String s,String enc) throws UnsupportedEncoding
Exception
public static String decode(String s,String enc) throws UnsupportedEncoding
Exception
```

其中第一个参数指定了要进行 URL 编码或解码的字符串，第二个参数指定对字符串以哪种字符集编码进行 URL 编码或解码。encode 方法首先将要 URL 编码的 Unicode 字符串转换成某种字符集编码的字节数组，然后对字节数组中的内容进行 URL 编码；decode 方法首先将要进行 URL 解码的字符串解码成一个字节数组，然后将这个字节数组按照某种字符集编码转换成 Unicode 字符串。

中文字符的 URL 编码和解码

01 按例程 6-29 编写一个普通的 Java 程序，打印出“中国”这两个字符的不同字符集编码的 URL 编码结果，以及查看对某个结果指定不对的字符集编码进行 URL 解码时所造成的问题。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

例程 6-29 TestURLEncode.java

```
import java.net.*;
public class TestURLEncode
{
    public static void main(String [] args) throws Exception
    {
        String utf8_url = URLEncoder.encode("中国","UTF-8");
        String gb2312_url = URLEncoder.encode("中国","gb2312");
        System.out.println("中国的 UTF-8 码的 URL 编码为: " + utf8_url);
        System.out.println("中国的 GB2312 码的 URL 编码为: " + gb2312_url);
        System.out.println(
            "将" + gb2312_url + "按 GB2312 码进行 URL 解码的结果为: " +
            URLDecoder.decode(gb2312_url,"gb2312"));
        System.out.println(
            "将" + gb2312_url + "按 iso8859-1 码进行 URL 解码的结果为: " +
            URLDecoder.decode(gb2312_url,"iso8859-1"));
    }
}
```

02 编译并运行这个程序后，屏幕上打印的结果如下：

```
中国的 UTF-8 码的 URL 编码为: %E4%B8%AD%E5%9B%BD
中国的 GB2312 码的 URL 编码为: %D6%D0%B9%FA
将%D6%D0%B9%FA 按 GB2312 码进行 URL 解码的结果为: 中国
将%D6%D0%B9%FA 按 iso8859-1 码进行 URL 解码的结果为: ???ú
```

从打印结果的最后一行可以看到，对于一个中文字符串的某种字符集编码进行 URL 编码后得到的结果字符串，如果再以另外一个字符集编码进行 URL 解码，得到的结果不是 URL 编码前的原始字符串，而是乱码。

6.9.2 浏览器怎样进行 URL 编码

浏览器对 FORM 表单中输入的中文字符都会先进行 URL 编码，再传送给 Web 服务器，但它是对中文字符的哪种字符集编码进行的 URL 编码呢？在第 5.5.1 节中已经讲过，浏览器在显示网页文档时，需要使用某种字符集编码来显示其中的内容。对于页面中的 FORM 表单中输入的内容，浏览器将按照当前显示页面时所采用的字符集编码来进行 URL 编码，即浏览器将 FORM 表单中输入的内容转换成浏览器当前显示页面时所选择的字符集编码后，再对这些内容进行 URL 编码后传送给服务器。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

浏览器怎样对 FORM 表单内容进行 URL 编码

01 在<tomcat 的安装目录>\webapps\it315 目录中编写一个名为 TestFormCharset.html 的网页文件，这个网页中包含有一个 FORM 表单，通过其中的文本输入框元素填写要传递给 Web 服务器的内容，如例程 6-30 所示。

例程 6-30 TestFormCharset.html

```
<meta http-equiv="Content-Type" content="text/html;charset=gb2312">
<form>
<input type="text" name="text1">
<input type="submit">
</form>
```

这个网页中设置了一个 http-equiv 属性值等于“Content-Type”的<meta>标签，用于在 HTML 文档中模拟 HTTP 消息的 Content-Type 头，告诉浏览器采用 GB2312 字符集编码来显示当前页面中的内容。

02 在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/TestFormCharset.html
```

在返回页面的文本输入框中输入一个“中”字后单击提交按钮，浏览器地址栏中显示的内容如下：

```
http://localhost:8080/it315/TestFormCharset.html?text1=%D6%D0
```

此时，由于浏览器以 GB2312 字符集编码显示当前页面内容，所以，浏览器以 GB2312 字符集编码来对 FORM 表单内填写的“中”进行 URL 编码。

03 单击浏览器的“查看”→“编码”→“Unicode(UTF-8)”菜单，以 UTF-8 字符集编码显示当前页面内容，接着在文本输入框中重新输入一个“中”字后单击提交按钮，浏览器地址栏中显示的内容如下：

```
http://localhost:8080/it315/TestFormCharset.html?text1=%E4%B8%AD
```

此时，由于浏览器以 UTF-8 字符集编码显示当前页面内容，所以，浏览器以 UTF-8 字符集编码来对 FORM 表单内填写的“中”进行 URL 编码。

04 单击浏览器的“查看”→“编码”→“西欧(ISO)”菜单，以 iso8859-1 字符集编码显示当前页面内容，接着在文本输入框中重新输入一个“中”字后单击提交按钮，浏览器地址栏中显示的内容如下：

```
http://localhost:8080/it315/TestFormCharset.html?text1=%26%2320013%3B
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

此时，由于浏览器以 iso8859-1 字符集编码显示当前页面内容，所以，浏览器以 iso8859-1 字符集编码来对 FORM 表单内填写的“中”进行 URL 编码。由于 iso8859-1 字符集中根本就不存在“中”这个字符，所以，浏览器对“中”这个字符进行 URL 编码的结果是一个混乱的数据。如果 TestFormCharset.html 文件中没有第一条设置<meta>标签的语句，这一步的实验结果与此处讲解的结果将有一点出入，浏览器地址栏中显示的内容如下：

```
http://localhost:8080/it315/TestFormCharset.html?text1=%92%86
```

6.9.3 getCharacterEncoding 方法

ServletRequest 接口中定义了一个 getCharacterEncoding 方法，该方法用于返回请求消息中的实体内容的字符集编码名称。如果请求消息中没有指定实体内容的字符集编码名称，则 getCharacterEncoding 方法返回 null。

6.9.4 setCharacterEncoding 方法

ServletRequest 接口中定义了一个 setCharacterEncoding 方法，该方法用于覆盖请求消息中的实体内容的字符集编码名称的设置。getParameter 和 getReader 方法将读取到的实体内容从字节数组形态转换成字符串返回时，都要参照请求消息中的实体内容的字符集编码名称，所以，setCharacterEncoding 方法应早于 getParameter 或 getReader 方法之前进行调用。

6.9.5 getParameter 方法的中文问题

ServletRequest 对象的 getParameter 等方法除了可以读取 HTTP 请求消息的请求行中的 URL 地址后的附加信息外，还可以读取 POST 方式下的“application/x-www-form-urlencoded”编码格式的实体内容。浏览器传递这些供 ServletRequest 对象的 getParameter 等方法读取的参数信息时，必须对其中的包括中文字符在内的特殊字符进行 URL 编码，这些参数是以哪种字符集编码进行的 URL 编码，ServletRequest 对象的 getParameter 等方法就必须以哪种字符集编码进行 URL 解码，这样才能返回正确的中文参数信息。ServletRequest 对象的 getParameter 等方法以哪种字符集编码对参数信息进行 URL 解码，需要记住以下三种情况：

（1）对于 HTTP 请求消息的请求行中的 URL 地址后的参数，getParameter 等方法进行 URL 解码时所采用的字符集编码在 Servlet 规范中没有明确规定，它由各个 Servlet 引擎厂商自行决定。对于这种情况，Tomcat 中的 ServletRequest 对象的 getParameter 等方法默认采用 ISO8859-1 字符集编码进行 URL 解码，因此无法返回正确的中文参数信息。

（2）对于 POST 方式下的“application/x-www-form-urlencoded”编码格式的实体内容，

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

getParameter 等方法以 ServletRequest 对象的 getCharacterEncoding()方法返回的字符集编码对其进行 URL 解码。使用前面编写的获取 HTTP 请求消息的所有请求头的程序，可以看到 IE 浏览器产生的 HTTP 请求消息中没有通过任何方式指定对实体内容进行 URL 编码时所采用的字符集编码，那么，Servlet 引擎将无法知道请求消息中的实体内容的字符集编码，getCharacterEncoding()方法的返回值为 null。对于这种情况，ServletRequest 对象的 getParameter 等方法将使用默认的 ISO8859-1 字符集编码对实体内容中的参数进行 URL 解码，因此也将无法返回正确的中文参数信息。

（3）ServletRequest 接口中定义了一个 setCharacterEncoding 方法来设置请求消息中的实体内容的字符集编码名称，getParameter 方法将以该方法设置的字符集编码对实体内容进行 URL 解码，所以，只要使用 ServletRequest.setCharacterEncoding 方法设置实体内容的字符集编码为其 URL 编码前的字符集编码，那么 getParameter 方法就可以从实体内容中返回正确的中文参数信息。但是，应该注意一点：ServletRequest.setCharacterEncoding 方法设置的是请求消息中的实体内容的字符集编码名称，它只影响 getParameter 方法对 POST 方式下的“application/x-www-form-urlencoded”编码格式的实体内容进行 URL 解码的结果，而不能影响 getParameter 方法对 HTTP 请求消息的请求行中的 URL 地址后的参数进行 URL 解码的结果。

解决使用 getParameter 方法读取请求参数时的中文乱码问题

01 按例程 6-31 编写一个名为 ChineseParamServlet 的 Servlet 程序，在其中获得一个指定名称的请求参数值，然后返回这个参数值给浏览器，并在命令行窗口中打印出该参数值有关的信息。

例程 6-31　ChineseParamServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ChineseParamServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out = response.getWriter();
        String paramValue = request.getParameter("param1");
        if(paramValue == null || paramValue.equals(""))
        {
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
            out.println("请传递一个正确的参数");
            return;
        }

        out.println("刚才输入的参数值为: " + paramValue);

        System.out.println("参数的字符集编码为: " +
                                request.getCharacterEncoding());
        int numsOfChar = paramValue.length();
        System.out.println("参数值为: " + paramValue);
        System.out.println("参数值中的字符个数: " + numsOfChar);
        System.out.print("参数中的每个字符的 Unicode 码值为: ");
        for(int i=0; i<numsOfChar; i++)
        {
            int ch = paramValue.charAt(i);
            System.out.print("[" + Integer.toHexString(ch) + "]");
        }
        System.out.println();
    }

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

编译 ChineseParamServlet.java 文件，确保编译后生成的 class 文件存放在了<tomcat 的安装目录>\webapps\it315\Web-INF\classes 目录中。

02 在<tomcat 的安装目录>\webapps\it315\Web-INF\web.xml 文件中注册该 Servlet，并设置其映射 URL。在 web.xml 文件中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>ChineseParamServlet</servlet-name>
    <servlet-class>ChineseParamServlet</servlet-class>
</servlet>
……
<servlet-mapping>
    <servlet-name>ChineseParamServlet</servlet-name>
    <url-pattern>/servlet/ChineseParamServlet</url-pattern>
</servlet-mapping>
```

保存 web.xml 文件后，重新启动 Tomcat。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

03 在<tomcat 的安装目录>\webapps\it315 目录中编写一个名称为 ChineseParam.html 的网页文件，其中包含一个用于产生参数的<form>表单，表单递交的目标为 ChineseParamServlet，如例程 6-32 所示。

例程 6-32 ChineseParam.html

```
<h1>测试中文参数的页面</h1>
<form action="servlet/ChineseParamServlet" method="POST">
输入测试的中文信息：<input type="text" name="param1" value="中国"><br>
<input type="submit" value="提交">
</form>
```

04 重新启动 Tomcat，并在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/ChineseParam.html
```

浏览器返回的页面如图 6.28 所示。

单击提交按钮后，浏览器中返回的结果如图 6.29 所示。

图 6.28

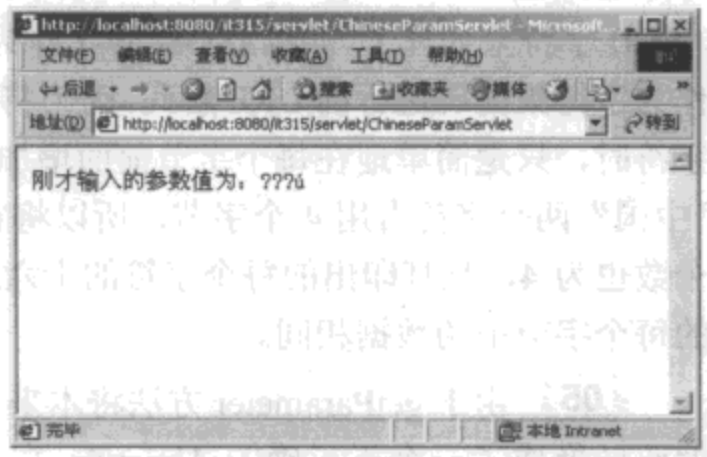


图 6.29

浏览器中返回的中文参数值为乱码。由于这个 Servlet 程序涉及从浏览器中读取输入参数和向浏览器输出字符文本这两个方面，输入时的 ServletRequest.getParameter 方法要将 URL 解码后的字节数组转换成 Unicode 字符串，而输出时的 PrintWriter.println 方法又要将 Unicode 字符串转换某种字符集编码的字节数组，输入或输出时的字符编码转换问题都能造成图 6.29 所示的中文乱码现象。要想找出乱码产生的原因，首先必须确定是在输入时发生了字符编码转换问题，还是在输出时发生了字符编码转换问题。图 6.29 中的浏览器正常显示出了 ChineseParamServlet 程序输出的其他中文字符（即“刚才输入的参数值为”这几个中文字符），据此可以断定 ChineseParamServlet 程序在输出时进行的字符编码转换没有问题。另外，ServletRequest.getParameter 方法返回的字符串既是输入时的字符编码转换结果，又是输出时的字符编码转换源头，只要打印出它里面的每个字符的数值，与输入字符的 Unicode 码进行比较，就可以更加直观地判断中文乱码发生的原因。如果 ServletRequest.getParameter 方法返回的字符串中的每个字符的数值不是输入字符的 Unicode 码，则说明

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

是在 getParameter 方法读取输入参数时发生了字符编码转换问题，否则，说明是在 PrintWriter.println 方法输出时发生了字符编码转换问题。

ChineseParamServlet 程序在命令行窗口中打印出了有关的分析信息，查看启动 Tomcat 的命令行窗口，可以看到其中打印出了如下信息：

```
参数的字符集编码为：null
参数值为：???ú
参数值中的字符个数：4
参数中的每个字符的 Unicode 码值为：[d6][d0][b9][fa]
```

上面的 FORM 表单只传递了“中国”这两个字符给 ChineseParamServlet 程序，字符的个数应该为 2，其正确的 Unicode 码字符串的长度应该为 2，而打印出的参数值的字符个数为 4，这显然是 getParameter 方法读取输入参数时发生了字符编码转换问题。使用 UltraEdit 的十六进制查看方式可以知道“中国”这两个字符的 Unicode 码和 GB2312 码，从打印出的每个字符的 Unicode 码值上可以看出，它们不是“中国”这两个字符的 Unicode 码值，而具有“中国”这两个字符的 GB2312 编码的特征。这是由于 getCharacterEncoding() 方法返回值为 null，getParameter 方法对参数值进行 URL 解码时直接把%XX 当做一个 ASCII 字符来处理了，即以 ISO8859-1 作为字符集编码进行的 URL 解码，而不是以 GB2312 作为字符集编码进行的 URL 解码。因为将字节数据以 iso8859-1 字符集编码转换成 Unicode 字符时，只是简单地在每个字节前面增加一个内容为 0 的字节，原来使用 GB2312 编码的“中国”两个字符占用 4 个字节，所以将它按 iso8859-1 字符集编码转换后的 unicode 字符个数也为 4，且打印出的每个字符的十六进值与采用 GB2312 编码的“中国”这两个字符的每个字节中的数据相同。

05 由于 getParameter 方法将本来是 GB2312 编码的请求参数值按 iso8859-1 字符集编码转换成了一个不正确的 Unicode 字符串，PrintWriter.println 方法再将这个不正确的 Unicode 字符串按 GB2312 字符集编码转换成字节数组输出时，就出现了中文乱码，其具体细节如图 6.30 中的过程①和②。

iso8859-1 字符集编码到 Unicode 编码的转换是一种可逆的运算，没有发生信息损失，只要将转换成的 Unicode 字符串再转换成 iso8859-1 编码，就可以还原出最初的字节数组，其具体细节如图 6.30 中的过程①和③。如果将还原出的初始字节数组按照它本来的字符集编码转换成 Unicode 编码，就可以得到正确的 Unicode 字符串，然后再将正确的 Unicode 字符串按照一种正确的字符集编码进行输出时，浏览器中就可以显示出正确的中文字符了，其具体细节如图 6.30 中的过程④和⑤。针对本例，只要将 ChineseParamServlet.java 中向浏览器输出 paramValue 的语句：

```
out.println("刚才输入的参数值为：" + paramValue);
```

修改成如下形式就可以了：

```
out.println("刚才输入的参数值为：" +
            new String(paramValue.getBytes("iso8859-1"),"gb2312"));
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

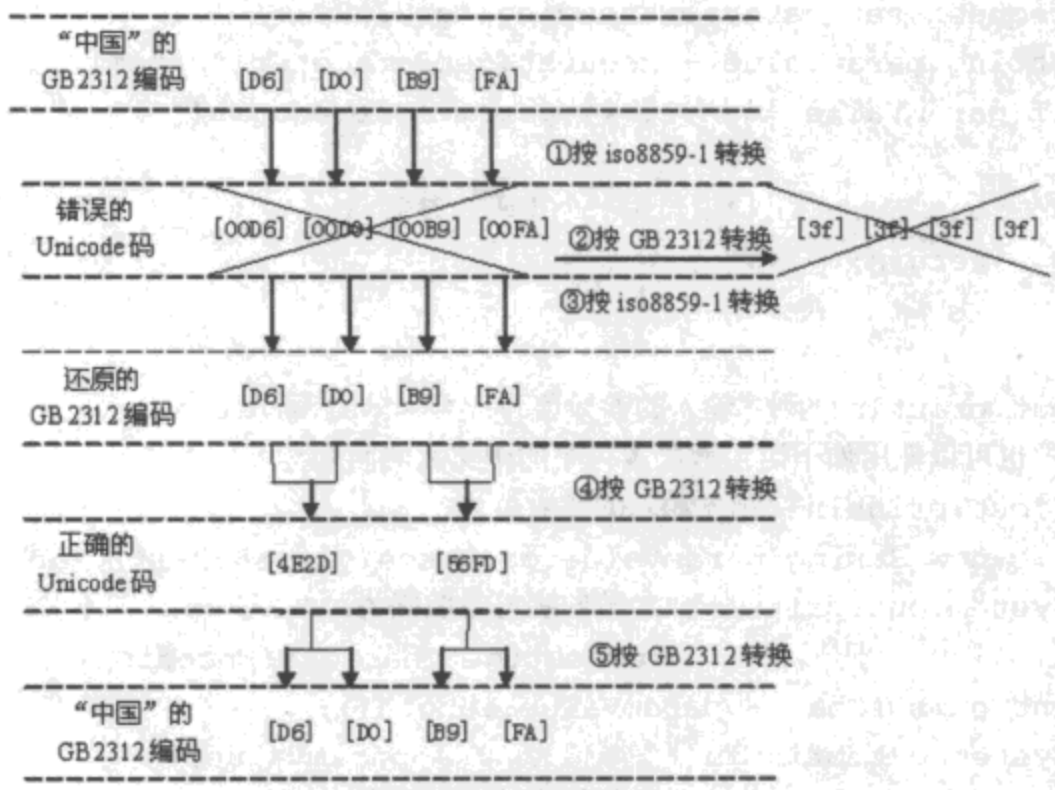


图 6.30

这样，从 getParameter 方法内部处理请求参数开始，GB2312 编码的中文参数值经历了"按 iso8859-1 字符集编码转换成 Unicode 字符串"，再"按 iso8859-1 字符集编码还原成初始的字节数组"，接着是"按 GB2312 字符集编码转换正确的 Unicode 字符串"的过程，最后将正确的 Unicode 字符串转换成 GB2312 编码后输出给浏览器。

上面的方式是一种用于解决中文乱码的常用方法，但是，如果能让 getParameter 方法在处理请求参数时就直接返回正确的中文 Unicode 字符串，岂不更好？对于例程 6-31 所示的 ChineseParamServlet 程序，只要在调用 ServletRequest.getParameter 方法之前，使用 ServletRequest.setCharacterEncoding 方法设置实体内容的字符集编码为 GB2312，getParameter 方法就可以直接返回 GB2312 编码的中文参数值的正确的 Unicode 字符串，如例程 6-33 所示。

例程 6-33 ChineseParamServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ChineseParamServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out = response.getWriter();
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        request.setCharacterEncoding("gb2312");
        String paramValue = request.getParameter("param1");
        if(paramValue == null || paramValue.equals(""))
        {
            out.println("请传递一个正确的参数");
            return;
        }

        out.println("刚才输入的参数值为: " + paramValue);
        //也可以使用如下语句解决本程序的中文乱码问题
        /*out.println("刚才输入的参数值为: " +
            new String(paramValue.getBytes("iso8859-1"),"gb2312"));*/
        System.out.println("参数的字符集编码为: " +
                           request.getCharacterEncoding());
        int numsOfChar = paramValue.length();
        System.out.println("参数值为: " + paramValue);
        System.out.println("参数值中的字符个数: " + numsOfChar);
        System.out.print("参数中的每个字符的 Unicode 码值: ");
        for(int i=0; i<numsOfChar; i++)
        {
            int ch = paramValue.charAt(i);
            System.out.print("[" + Integer.toHexString(ch) + "]");
        }
        System.out.println();
    }

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

重新编译 ChineseParamServlet.java 后，重复第（4）步的操作，浏览器中显示出了正常的中文参数字符，查看启动 Tomcat 的命令行窗口，看到其中打印出了如下信息：

```
参数的字符集编码为: gb2312
参数值为: 中国
参数值中的字符个数: 2
参数中的每个字符的 Unicode 码值: [4e2d][56fd]
```

这时候打印出的参数值的字符个数为 2，打印出的每个字符的 Unicode 码值正好是“中国”这两个字符的 Unicode 码值。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

06 修改例程 6-32 所示的 ChineseParam.html 文件，将 FORM 表单的提交方式由“POST”改为“GET”，如例程 6-34 所示。

例程 6-34 ChineseParam.html

```
<h1>测试中文参数的页面</h1>
<form action="servlet/ChineseParamServlet" method="GET">
输入测试的中文信息：<input type="text" name="param1" value="中国"><br>
<input type="submit" value="提交">
</form>
```

在浏览器中重新访问 http://localhost:8080/it315/ChineseParam.html 页面，最好是刷新后单击 IE 浏览器上的“查看”→“源文件”菜单，确保浏览器中显示的网页内容是更新后的内容。单击提交按钮后，浏览器和启动 Tomcat 的命令行窗口中又出现了与第（4）步完全一样的中文乱码问题。这是由于以 GET 方式提交的表单元素是以请求 URL 后的附加参数的形式，而不是以请求消息的实体内容出现的。ServletRequest.setCharacterEncoding 方法设置的是请求消息中的实体内容的字符集编码名称，它只影响 getParameter 方法对 POST 方式下的“application/x-www-form-urlencoded”编码格式的实体内容进行 URL 解码的结果，而不能影响 getParameter 方法对 HTTP 请求消息的请求行中的 URL 地址后的参数进行 URL 解码的结果。

进入 Tomcat 自带的关于 Tomcat 配置（Tomcat Configuration）的帮助文档，查看 Http 连接器（HTTP Connector）的属性设置说明，可以看到 Http 连接器中提供了 URIEncoding 和 useBodyEncodingForURI 这两个属性来设置对 URL 后的附加参数进行 URL 解码时该如何选择字符集编码，如图 6.31 所示。

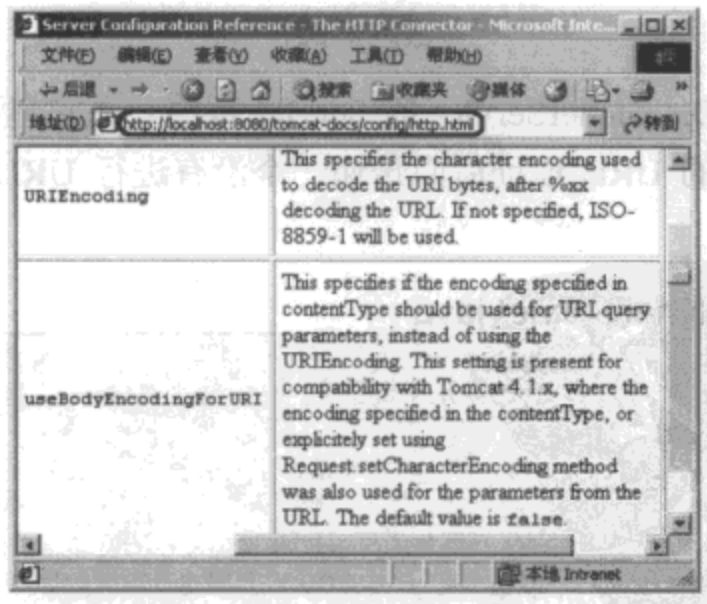


图 6.31

URIEncoding 用于指定 URL 后的附加参数的字符集编码，useBodyEncodingForURI 则说明是否采用实体内容的字符集编码设置来代替 URIEncoding 的设置，也就是说，当

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

useBodyEncodingForURI 属性设置为 true 时，ServletRequest.setCharacterEncoding 方法设置的字符集编码也将影响 getParameter 等方法对 URL 地址后的参数进行 URL 解码的结果。将<tomcat 的安装目录>\conf\server.xml 中的相应一段内容修改成如下形式：

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

重新启动 Tomcat，再次访问 http://localhost:8080/it315/ChineseParam.html 页面的时候，在浏览器和启动 Tomcat 的命令行窗口中都打印出了中文参数的正常信息。

07 HTTP 协议规定浏览器不能向服务器直接传递某些特殊字符，必须是这些字符进行 URL 编码后再传送。但是，有人说："我试过，不进行 URL 编码与进行 URL 编码后的运行效果是一样的，对中文字符没有必要进行 URL 编码！"，例如，在上面实验的基础上，在浏览器地址栏中直接输入如下地址：

```
http://localhost:8080/it315/servlet/ChineseParamServlet?param1=中国
```

浏览器中返回的结果如图 6.32 所示。

图 6.32

接着编写一个名为 DispatcherServlet 的 Servlet 程序，在它里面将请求转发给 ChineseParamServlet，并在转发的 URL 地址后面传递一个没有进行 URL 编码的中文参数，如例程 6-35 所示。

例程 6-35 DispatcherServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DispatcherServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        String china = "中国";
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        RequestDispatcher rd = request.getRequestDispatcher(
                "/servlet/ChineseParamServlet?param1=" + china);
        rd.forward(request,response);
    }
}
```

编译并在 web.xml 文件中部署 DispatcherServlet，使用浏览器访问 DispatcherServlet，浏览器中也能显示出如图 6.32 所示的正常中文字符。

有人说："URL 后面的中文参数不进行 URL 编码，程序运行也不会有问题，所以，对 URL 中的参数进行 URL 编码完全是多此一举"，果真如此吗？其实这个问题与"酒后禁止开车"的规定类似，确实有人酒后开车了也没出现问题，但也确实有人因酒后开车而丢了性命，所以，大家最好还是遵守交通规定，酒后不要开车！不同厂商的 Servlet 引擎内部进行 URL 解码的处理方式不尽相同，对于没有严格遵循 URL 编码规则的字符串进行 URL 解码时，它们返回的结果可能会大不一样。

08 下面看看对 URL 后面的中文参数不进行 URL 编码时会出现问题的一种情况。修改 ChineseParamServlet.java 程序，注释掉其中的如下两条语句：

```
request.setCharacterEncoding("gb2312");
out.println("刚才输入的参数值为：" + paramValue);
```

取消其中的如下语句的注释：

```
/*out.println("刚才输入的参数值为：" +
        new String(paramValue.getBytes("iso8859-1"),"gb2312"));*/
```

重新编译 ChineseParamServlet.java 后，在浏览器地址栏中直接输入如下地址：

```
http://localhost:8080/it315/servlet/ChineseParamServlet?param1=中国
```

浏览器中显示出了如图 6.32 所示的正常中文字符。但是，这次使用浏览器访问 Dispatcher Servlet，浏览器中显示的中文参数值是如图 6.33 所示的中文乱码。

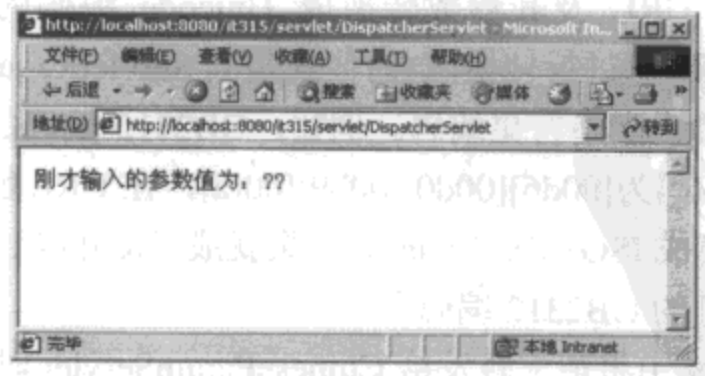


图 6.33

查看启动 Tomcat 的命令行窗口，看到其中打印出了如下信息：

```
参数的字符集编码为：null
参数值为：??
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
参数值中的字符个数：2
参数中的每个字符的 Unicode 码值：[3f][3f]
```

根据命令行窗口中打印的结果，可以判断中文乱码是由于 getParameter 方法读取输入参数时发生了字符编码转换问题所造成的，并且 getParameter 方法返回的字符串中的每个字符的 Unicode 码值为\u003f，这个值无法通过任何编码转换方式转换成一个正确的字符集编码。为什么在浏览器地址栏中直接访问 ChineseParamServlet 时向它传递的中文参数能够正常显示，但是，向它传递的中文参数却不能正常显示呢？下面分析其中的原因。

- URL 的编码和解码不能直接在 Unicode 字符集编码上进行，而是针对字符的 UTF-8 字符集编码或其他本地字符集编码进行的，而 Java 中的 URL 字符串是以 Unicode 字符集编码存在的，所以，在完成 URL 解码的方法内部，它首先要把以 Unicode 字符集编码形式存在的 URL 字符串转换成某种字符集编码的一个字节数组，然后对这个字节数组中的内容进行 URL 解码，解码得到的结果也是一个字节数组，最后需要将这个得到的结果字节数组按某种字符集编码转换成 Unicode 字符集编码的字符串返回。

如果没有指定 URL 解码时所采用的字符集编码，URL 解码方法默认选择 ISO8859-1 字符集编码，它首先将表示 URL 的 Unicode 字符串转换 ISO8859-1 字符集编码的字节数组，在完成 URL 解码后，再将其中的参数值又按 ISO8859-1 字符集编码转换成 Unicode 字符集编码的字符串返回。由于“中国”这两个字符在 ISO8859-1 字符集编码中不存在对应的编码，它们从正确的 Unicode 码转换成 ISO8859-1 码的结果都表示为 0x3f，这个转换过程是有损、不可逆的，内容为 0x3f 的字节再按 ISO8859-1 码转换成 Unicode 码时，结果就成了\u003f，将它以十六进制的整数形式打印出来时的结果就是 3f。

- 在浏览器地址栏中直接访问 ChineseParamServlet 时，URL 及其中的中文参数是以 GB2312 编码传递给 Servlet 引擎的，Servlet 引擎需要将它们转换成 Unicode 编码的 Java 字符串后再封装到 ServletRequest 对象中。由于在正常情况下，URL 都应进行 URL 编码，其中只会出现 ISO8859-1 字符集中的字符，所以 Servlet 引擎把浏览器地址栏中输入的 URL 及其参数转换成 Unicode 编码的 Java 字符串时，将按 ISO8859-1 字符集编码进行转换，这时候封装到 ServletRequest 对象中的中文参数并不是其正确的 Unicode 字符集编码。例如，GB2312 编码的“中国”这两个字符转换成的 Unicode 编码为[00d6][00d0][00b9][00fa]，在 URL 解码过程中，首先将它们从 Unicode 码转换成 ISO8859-1 码时，结果就成了[d6][d0][b9][fa]，又还原成了“中国”这都两个字符的 GB2312 编码。
- 在另外一个 Servlet 中将请求转发给 ChineseParamServlet 时，URL 及其中的参数直接以 Unicode 编码的 Java 字符串传送给了 Servlet 引擎，Servlet 引擎直接将这些参数封装到 ServletRequest 对象中。此时，“中国”这两个字符的 Unicode 编码为[4e2d][56fd]，在 URL 解码过程中，首先将它们从 Unicode 码转换成 ISO8859-1 码时，结果就成了[3f][3f]，初始的信息完全丢失了。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

只要对 URL 中的包括中文字符在内的特殊字符进行了 URL 编码，那么在各个厂商的 Servlet 引擎下都将有完全一致的运行效果和解决方案，而不会出现类似上面的问题。修改 DispatcherServlet.java 程序，对其中转发的 URL 地址后面的中文参数进行 URL 编码，如例程 6-36 所示。

例程 6-36 DispatcherServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DispatcherServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        String china = "中国";
        RequestDispatcher rd = request.getRequestDispatcher(
                "/servlet/ChineseParamServlet?param1=" +
                java.net.URLEncoder.encode(china,"GB2312"));
        rd.forward(request,response);
    }
}
```

重新编译 DispatcherServlet.java 源文件，等待 Tomcat 重新装载 DispatcherServlet 后，在浏览器窗口中刷新对 DispatcherServlet 的访问，这时浏览器中显示出的 getParameter 方法获得的参数值部分变成了正常的中文字符。

09 下面来看看对 URL 后面的中文参数不进行 URL 编码时会出现问题的另外一种情况。修改 DispatcherServlet.java 程序，在它里面将请求重定向到 ChineseParamServlet，并在重定向的 URL 地址后面传递一个没有进行 URL 编码的中文参数，如例程 6-37 所示。

例程 6-37 DispatcherServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DispatcherServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        String china = "中国";
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
            response.sendRedirect(request.getContextPath() +
                "/servlet/ChineseParamServlet?param1=" + china);
        }
    }
```

重新编译 DispatcherServlet.java 源文件，等待 Tomcat 重新装载 DispatcherServlet 后，重新启动一个浏览器窗口访问 DispatcherServlet，显示结果如图 6.34 所示。

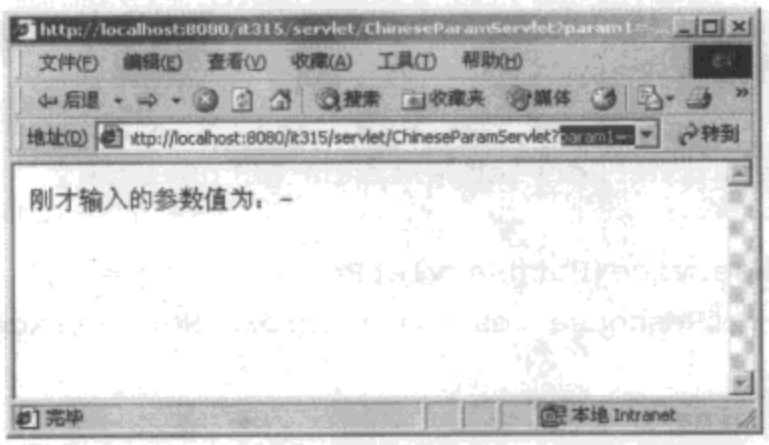


图 6.34

在图 6.34 中的浏览器地址栏中可以看到，浏览器重定向访问 ChineseParamServlet 时，实际传递给 ChineseParamServlet 的参数就已经不是 DispatcherServlet 中设置的参数内容了。这是因为表示 URL 地址的字符串在 DispatcherServlet 程序中是以 Unicode 编码存在的，Servlet 引擎在通知浏览器进行重定向时，需要将该 URL 地址转换成某种本地字符集编码的形式，再传递给浏览器。由于 DispatcherServlet 程序不知道这种情况下该以哪种字符集编码对 URL 地址内容进行转换，所以就采用了默认的 ISO8859-1，这就是对 URL 后面的参数不进行 URL 编码时可能会导致的问题之一。如果不能直接看到图 6.34 的浏览器地址栏中的内容，请单击浏览器工具栏中的“后退”按钮后，再单击“前进按钮”。接着修改 DispatcherServlet.java 程序，对其中的参数进行 URL 编码，如例程 6-38 所示。

例程 6-38　DispatcherServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DispatcherServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        String china = "中国";
        response.sendRedirect(request.getContextPath() +
            "/servlet/ChineseParamServlet?param1=" +
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
            java.net.URLEncoder.encode(china,"GB2312"));
        }
    }
```

重新编译 DispatcherServlet.java，等待 Tomcat 重新装载 DispatcherServlet 后，用浏览器再次访问 DispatcherServlet，这时就可以看到正常的中文显示结果了。

6.9.6 一些参考经验

（1）不考虑人为使用浏览器菜单修改显示页面内容时所采用的字符集编码，浏览器对 FORM 表单中输入的内容进行 URL 编码时所使用的字符集编码与 FORM 表单所在页面的字符集编码是相同的，这意味着 Servlet 程序只要跟踪提交 FORM 表单的网页的字符集编码即可。在通常情况下，提交 FORM 表单的页面与处理表单请求的 Servlet 程序都是由同一个人或同一个项目组开发的，因此，他们是可以知道 FORM 表单内容的字符集编码的，只要在处理表单提交的 Servlet 程序中将传递给 setCharacterEncoding 方法的字符集编码设置为该表单所在页面的字符集编码即可。

（2）如果 Servlet 程序可以接收来自多个其他站点的 FORM 表单提交的数据，而每个其他站点的页面所采用的字符集编码可能各不一样，要想让 Servlet 程序知道 FORM 表单内容的字符集编码，可以在 FORM 表单中增加一个隐藏字段来传递当前 FORM 内容的字符集编码名称。这个隐藏字段的名称和值必须都是英文 ASCII 字符，由于各国的本地字符集都将 ASCII 码字符集纳入进来，作为本地字符集的一个子集使用，当在本地字符集中出现英文 ASCII 码字符时，直接采用该字符的 ASCII 码表示，而不是对它们进行重新编码，所以，英文 ASCII 码字符在各国的本地字符集中的编码都是完全相同的，英文 ASCII 字符被当做任意一个国家的本地字符集来处理都是可以的，一篇文档中的英文字符在采用各个本地字符集的计算机系统中都可以畅通无阻，对英文 ASCII 的解码不会出现问题。Servlet 程序首先检索出这个含有字符集编码名称的参数，然后使用检索到的字符集编码名称来对其他参数进行编码转换。

（3）只要超链接、FORM 表单的 action 属性设置、请求转发和重定向的 URL 等任何一种请求路径中要包含参数，就必须对参数部分进行 URL 编码。对参数进行 URL 编码时所选择的字符集编码应尽量与当前页面的字符集编码保持一致，也可以用一个参数来指定 URL 编码内容的字符集编码名称。

（4）因为所有标准的浏览器和客户机终端都支持 UTF-8 编码，如果 Web 服务器要能兼容处理各个国家和地区版本的浏览器所传递的 FORM 表单信息，网页文档应当使用 UTF-8 编码格式，传递给 setCharacterEncoding 方法的字符集编码也始终设置为 UTF-8。

（5）在编写应用程序时应该注意一些隐性的字符编码错误。例如，有些程序在输出时有编码转换错误，将正常的中文 Unicode 字符串按 ISO8859-1 转换成字节流输出，输出的

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

结果将是乱码。如果该程序在读取输入时返回的字符串也有转换错误，将 GB2312 编码的中文字节流按 ISO8859-1 转换成了一个 Unicode 字符串返回，当该程序输出它所读取到的有问题的字符串时，显示的结果将是正常的中文字符，这是因为对错误读取进的字符串再用错误的方式进行输出，而输入和输出时的错误正好产生了抵消的效果。

6.10 思考与实践

友情提示：请认真地完成以下思考与实践，并将完成的结果以及操作步骤提交给 www.it315.org 站点评审，表现优异者将获得 www.it315.org 站点的奖励和专家的特别指导。

1. 将 6.4.11 节的 HideJSServlet.java 程序修改为不是直接使用 out.println 语句输出 test.js 文件中的内容，而是采用将请求转发给 test.js 文件的方式来输出其中的内容。

注意 为了让浏览器不能直接访问到 test.js 文件，应将 test.js 文件保存在 Web 应用程序的 Web-INF 子目录中。

2. 隐藏表单字段元素在实际应用中非常重要，例如，一个网站的用户注册过程由两个步骤完成，每个步骤对应一个表单页面，用户在第一步的表单页面中填写了自己的姓名后，接着进入第二步的页面中填写爱好和特长等信息。在提交第二个页面的表单信息时，需要将第一个页面中收集到的用户名称也传送给 Web 服务器，这就需要在第二个网页的表单中加入一个隐藏表单字段元素，让它的 value 属性值等于第一个网页中收集到的用户名。另外，有一些网页要通过其中的 JavaScript 代码动态产生一些信息后，再把这些信息提交给 Web 服务器，通常的做法就是将这些信息作为一个 FORM 表单的隐藏字段进行预设，然后由 JavaScript 代码对它们进行修改后再提交表单，这些信息就可随着 FORM 表单提交给 Web 服务器。请对这两种应用分别设计一个案例进行演示。

提示 可以借助搜索引擎搜索“input type hidden”关键字来找到一些可借鉴的应用。

3. 当客户端发出请求后就中断了与 Web 服务器的连接，但 Servlet 程序仍然会忠实地处理完这个请求，只是响应内容无法返回给客户端。请参照本书编写实验步骤的方法设计一个 Telnet 访问 Servlet 程序的实验对此进行验证。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

第 7 章　会话与状态管理

7.1　Web 应用中的会话与会话状态

在 Web 服务器端的程序开发中，会话状态管理是一个重要的问题。要掌握会话状态管理，首先就必须理解会话与会话状态。

7.1.1　会话与会话状态简介

在日常生活中，从拨通电话到挂断电话之间的一连串的你问我答的过程就是一个会话。Web 应用中的会话过程非常类似日常生活中的一次你问我答的电话通话过程，它是指一个客户端浏览器与 Web 服务器之间连续发生的一系列请求和响应过程，例如，一个用户在某网站上的整个购物过程就是一个会话。

在电话通话的过程中，人们说出的每句话是参照前面谈话的信息发出的，同样，在浏览器与 Web 服务器的会话过程中，Web 服务器每次响应的结果都应该参照前面的请求和响应信息而创建。Web 应用的会话状态是指 Web 服务器与浏览器在会话过程中产生的状态信息，借助会话状态，Web 服务器能够把属于同一会话中的一系列的请求和响应过程关联起来，使得它们之间可以互相依赖和传递信息。例如，在购物网站处理产品订购和结算的访问请求时，都必须知道登录请求的处理结果，以便了解是哪个登录账户在订购产品和要进行结算；处理结算请求时还必须知道产品订购请求的处理结果，以便计算出结算总金额；其中，用户的登录账户和订购的产品就是会话的状态信息。

7.1.2　如何实现有状态的会话

在基于 Web 的电子商务和其他一些应用中，浏览器与 Web 服务器的会话过程必须是有状态的，即 Web 服务器程序在处理浏览器的下一个请求时，需要知道对该浏览器前面所发出的一些请求的处理结果。否则，就会出现类似这样的情况：张三从网站的登录页面登入后，再进入购物页面购物时，由于负责处理购物请求的服务器程序不知道对登录页面的处理结果，而只知道发生了购物请求，那将无法知道该购物请求是张三提交的，还是李四提交的。

HTTP 协议是一种无状态的协议，浏览器主动发出一个请求，Web 服务器被动地回应

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

一个结果，只要浏览器发出的请求消息完全一样，不管这个请求消息是在哪个主机上的哪种浏览器上发出的，Web 服务器都用完全一样的方式进行回应。Web 服务器并不关心两次访问请求是否是同一个浏览器发出的、还是两个浏览器分别发出的，所以，Web 服务器本身不能识别出哪些请求是同一个浏览器发出的。即使浏览器发出上一次访问请求后立即发出了下一次访问请求，Web 服务器依然不能知道下一次请求与上一次请求是同一个浏览器发出的，因此，服务器接收到来自某个浏览器的访问请求时，无法确定该浏览器以前的访问请求信息。对于这种无状态的会话过程，浏览器的每一次请求都是完全孤立的，Web 服务器端根本就不认为同一个浏览器发出的请求之间有任何关联，浏览器的下一次访问结果并不依赖于是否有过上一次访问。

由于 HTTP 协议本身不具有会话状态，要想在 Web 应用程序中维持会话状态信息和实现有状态的会话管理，在 Web 服务器端程序的开发方案中需要提供一些专门的技术来进行解决。Web 服务器在一段时间内通常都会同时接收到多个客户端浏览器的访问请求，Web 服务器端程序首先要能从大量的请求消息中区分出哪些请求消息属于同一个会话，即能识别出来自同一个浏览器的访问请求，然后再将来自同一个浏览器的访问请求组织成一个会话过程。Web 服务器怎么能知道哪些请求是来自同一个浏览器的呢？这需要浏览器对其发出的每个请求消息都进行标识，属于同一个会话中的请求消息都附带同样的标识号，而属于不同会话的请求消息总是附带不同的标识号，这个标识号就称之为会话 ID（SessionID）。

会话 ID 可以通过一种称之为 Cookie 的技术在请求消息中进行传递，也可以作为请求 URL 的附加参数进行传递。会话 ID 是 Web 服务器为每个客户端浏览器分配的一个惟一代号，它通常是在 Web 服务器接收到某个浏览器的第一次访问时产生的，并且随同响应消息一道发送给浏览器，浏览器则在以后发出的访问请求消息中都顺带一条信息："我的代号（即 SessionID）是 xxx"。这样，Web 服务器程序就能识别出这次访问请求是来自哪个客户端浏览器了。会话过程由 Web 服务器端的程序开启，一旦开启了一个会话，服务器端程序就要为这个会话创建一个独立的存储结构来保存该会话的状态信息，同一个会话中的访问请求都可以且只能访问属于该会话的存储结构中的状态信息。目前的各种 Web 服务器端程序的开发方案基本上都为此提供了一种称之为会话（session）的技术，这种技术用于在 Web 服务器端保存和跟踪每个客户端浏览器在一次会话过程中的状态信息。

7.2 Cookie

Cookie 不是 HTTP 1.1 的内容，它是 Netscape 的一个扩展，为浏览器和 Web 服务器之间提供了一种有效的状态信息交换方式，在实际中得到了广泛的支持和应用。

7.2.1 什么是 Cookie

Cookie 是一种在客户端保持 HTTP 状态信息的技术，它好比商场发放的优惠卡。顾客在一个商场购物结账离开时，商场可以决定是否赠送给顾客一张优惠卡，不同顾客的优惠

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

卡上记载的信息可以不同，例如，记载该顾客累计购物的金额和有效期限。顾客可以决定是否接受这张优惠卡，一旦顾客接受了这张优惠卡，那么他在以后每次光顾该商场时，都将携带这张优惠卡，商场也将根据这张优惠卡上记载的信息进行一些特殊的事务处理，例如，计算折扣率和累加本次购物金额。

Cookie 是在浏览器访问 Web 服务器的某个资源时，由 Web 服务器在 HTTP 响应消息头中附带传送给浏览器的一片数据，Web 服务器传送给各个客户端浏览器的数据是可以各不相同的。浏览器可以决定是否保存这片数据，一旦 Web 浏览器保存了这片数据，那么它在以后每次访问该 Web 服务器时，都应在 HTTP 请求头中将这片数据回传给 Web 服务器。显然，Cookie 最先是由 Web 服务器发出的，是否发送 Cookie 和发送的 Cookie 的具体内容，完全是由 Web 服务器决定的。

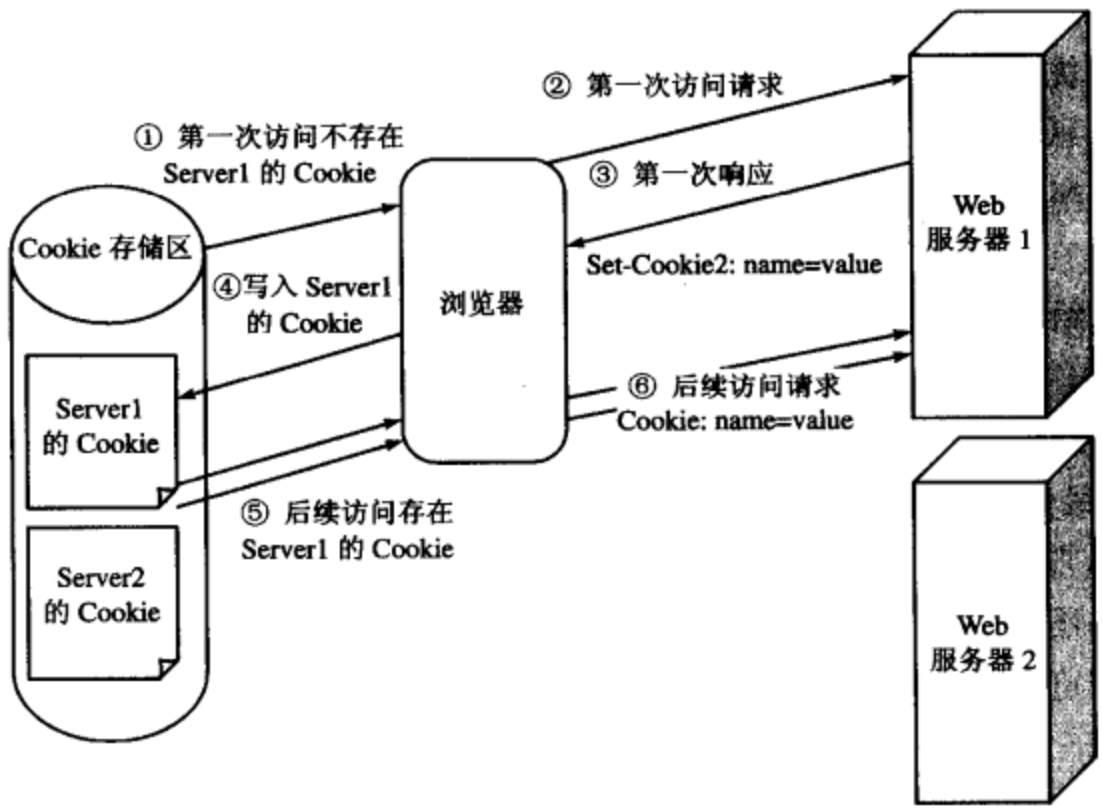
Web 服务器通过在 HTTP 响应消息中增加 Set-Cookie 响应头字段将 Cookie 信息发送给浏览器，浏览器则通过在 HTTP 请求消息中增加 Cookie 请求头字段将 Cookie 回传给 Web 服务器。一个 Cookie 只能标识一种信息，它至少含有一个标识该信息的名称（NAME）和设置值（VALUE）。一个 Web 站点可以给一个 Web 浏览器发送多个 Cookie，这样，在 Web 浏览器和 Web 服务器之间就可以使用多个 Cookie 来传递多种信息，例如，用一个 Cookie 来标识访问者的姓名，用另外一个 Cookie 来标识该用户登录站点的次数。一个 Cookie 除了有名称和设置值外，它还可以有一些其他的附加属性，例如，Cookie 的有效时间。如果设置了 Cookie 的有效时间，接受它的浏览器进程将该 Cookie 保存在计算机硬盘中，只有该 Cookie 超出有效时间后才被删除，这样的 Cookie 将被同一台计算机上启动的多个浏览器进程共享。正如一个顾客可以有多家商场提供的优惠卡一样，一个 Web 浏览器也可以存储多个 Web 站点提供的 Cookie。为了防止 Cookie 塞满客户机的硬盘，浏览器一般只允许存放 300 个 Cookie，每个站点最多存放 20 个 Cookie，每个 Cookie 的大小限制为 4KB。如果没有设置 Cookie 的有效时间，接受它的浏览器进程只将该 Cookie 保存在自己的内存空间中，在该浏览器进程关闭时，它里面保存的所有 Cookie 也将随之消失。

Cookie 在浏览器与 Web 服务器之间传送的过程如图 7.1 所示。

Cookie 实现了一种在浏览器和服务器之间产生有状态会话的方式，它可以把一个浏览器访问的同一个服务器上的所有程序连贯起来，在这些程序之间传递数据。例如，当用户使用浏览器访问某个网站的登录程序进行登录后，无论这个浏览器再访问该网站的哪个程序，其他程序都能知道访问者的身份信息，这是在 Web 站点中非常普遍的一个应用。这种应用通常就是采用 Cookie 技术来实现的，当 Web 服务器程序验证登录请求中的用户名和密码后，产生一个标识该用户身份的标识号，然后在响应消息中将该标识号以 Cookie 的形式传递给浏览器，浏览器在以后每次访问该 Web 服务器时，都自动在请求消息头中将标识号又以 Cookie 的形式返回给 Web 服务器，凭借浏览器返回的标识号，Web 服务器的其他程序就能分辨出当前请求是由哪个用户发出的。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享



但是，有一点要注意，不保存在硬盘中的 Cookie 信息是否可以被同一台计算机上启动的多个浏览器进程共享，不同的浏览器有不同的处理方式。对于 IE 浏览器来说，保存在其中一个浏览器进程的内存空间中的 Cookie 是不能被其他浏览器进程共享的，这就会出现同一台计算机上的每个浏览器进程都会与服务器形成各自独立的会话；而对于 Mozilla Firefox 浏览器来说，所有的进程和标签页都共享 Cookie 信息。另外，在 IE 浏览器中按 Ctrl+N 键（或者单击“文件”→“新建”→“窗口”菜单）打开的窗口或者是用 javascript 的 window.open 语句打开的窗口，都会共享原窗口的 Cookie 信息，因为它们属于同一个浏览器进程内部的多个窗口。

7.2.2 Set-Cookie2 响应头字段

Set-Cookie2 头字段用于指定 Web 服务器向客户端传送的 Cookie 内容，通常是当 Web 服务器想要启动一个有状态的会话时，它就在响应消息头部分附带 Set-Cookie2 头字段来初始化这个会话。按照 Netscape 规范实现 Cookie 功能的 Web 服务器，使用的是 Set-Cookie 头字段，而不是 Set-Cookie2 头字段来完成这个功能。在目前的实际使用中，Set-Cookie 比 Set-Cookie2 更为常见，但两者的语法和作用类似，这里使用 RFC 2965 中定义的 Set-Cookie2 头字段进行讲解。

Set-Cookie2 头字段中设置的 cookie 内容是具有一定格式的字符串，它必须以 Cookie 的名称和设置值开头，格式为“名称=值”，后面可以加上 0 个或多个以分号（;）和空格分隔的其他可选属性，属性格式一般为“属性名=值”，例如，

```
Set-Cookie2: user=it315; Version=1; Path=/
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

除了“名称=值”对必须位于最前面外，其他的可选属性的先后顺序可以任意。Cookie 的名称只能由普通的英文 ASCII 字符组成，浏览器不用关心和理解 Cookie 的值部分的意义和格式，因为它只是存储这部分内容，然后将这部分内容原封不动地返回给 Web 服务器，只要 Web 服务器能理解值部分的意义就行了。大多数现有的 Web 服务器都是采用某种编码方式将值部分的内容编码成可打印的 ASCII 字符，RFC 2965 规范中没有明确限定编码方式。

下面是 Set-Cookie2 头字段中的各个属性的介绍。

1. Comment=value

Web 服务器可以设置 Comment 属性来说明这个 Cookie 的作用和对用户的私人信息可能造成的影响等，浏览器向用户提示这部分信息，让用户决定是否接收这个 Cookie 并继续同服务器会话。value 部分的字符必须使用 UTF-8 [参看 RFC2279]编码。

2. Discard

Discard 属性没有属性值部分，它用于通知浏览器关闭时无条件地删除该 Cookie 信息。如果没有设置这个属性，浏览器删除 Cookie 的行为由 Max-Age 属性决定。

3. Domain=value

Domain 属性用于指定 Cookie 在哪个域中有效，浏览器访问这个域中的所有主机时，都将回传这个 Cookie 信息。例如，下面的 Cookie 信息对 it315.org 域中的所有主机有效：

```
Set-Cookie2: user=it315; Version=1; Path=/; Domain=.it315.org
```

Domain 属性的默认值为当前主机名，浏览器以后只有在访问与当前主机名完全相同的 Web 服务器时，才会将这个 Cookie 回送给该服务器。Domain 属性设置值是不区分大小写的。

4. Max-Age=value

Max-Age 属性用于指定 Cookie 在客户端保持有效的时间，其中的 value 部分是以秒为单位的十进制整数。当 value 部分被设置为 0 时，浏览器将立即删除前面保存的这个 Cookie。如果 value 部分被设置成一个大于 0 的整数，浏览器将在文件系统中保存这个 Cookie 信息，直到没有超过指定的秒数之前，这个 Cookie 都保持有效，并且同一台计算机上运行的所有浏览器进程都可以使用这个 Cookie 信息。如果没有设置 Max-Age 属性，浏览器将这个 Cookie 保存在自身进程的内存中，Cookie 信息随这个浏览器进程的关闭而消失，并且这个 Cookie 信息对同一台计算机上的其他浏览器进程无效。

5. Path=value

Path 属性用于指定 Cookie 对服务器上的哪个 URL 目录和其子目录有效。这个属性的默认值是产生 Set-Cookie2 头字段时的那个请求 URL 地址所在的目录，这样，如果没有设

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

置 Path 属性时，浏览器以后在访问当前请求 URL 地址所在的目录及其子目录下的任何资源时都将回传该 Cookie 信息，而访问服务器上的其他目录下的资源时都不回传该 Cookie 信息。Path 属性设置值是区分大小写的。

6. Port[="portlist"]

Port 属性用于指定浏览器通过哪些端口访问 Web 服务器时，Cookie 才有效。端口号列表中的每个端口号之间用逗号（,）分隔，整个端口号列表必须用双引号（""）引起来。默认值是任意端口。

7. Secure

Secure 属性没有属性值部分，它用于通知浏览器在回传这个 Cookie 信息时，应使用安全的方式访问服务器，以保护 Cookie 中的机密和认证等方面的信息，否则，浏览器不能访问这个被储存的 Cookie 信息。如果没有设置 Secure 属性，浏览器可以用非安全的方式发送这个 Cookie 信息。

8. Version=value

Version 属性用于指定 Cookie 内容所遵循的版本格式，value 部分为一个十进制的整数，目前只有 Version=1 可以使用。

Web 服务器使用 Set-Cookie2 头字段发送一个 Cookie 以后，它在以后的响应消息中可以不再发送针对该 Cookie 的 Set-Cookie2 头字段，也可以发送与以前完全相同的 Set-Cookie2 头字段，还可以发送对以前信息进行了修改的 Set-Cookie2 头字段。如果浏览器接收到的 Set-Cookie2 头字段中设置的 Cookie 与以前存储的 Cookie 的名称相同，并且 Domain 属性值（不区分大小写）和 Path 属性值（区分大小写）都相等，那么就可以认为这两个 Cookie 代表的是同一个信息，新的 Cookie 将替换以前的 Cookie。如果服务器想让浏览器端保存的某个 Cookie 失效，可以重新发送包含该 Cookie 的 Set-Cookie2 头字段，并在设置值部分增加 Max-Age=0 属性。

一个响应头中可以包含多个 Set-Cookie2 头字段来设置多个 Cookie，也可以用一个 Set-Cookie2 头字段来设置多个 Cookie。在响应头中用一个 Set-Cookie2 头字段设置多个 Cookie 是一种非正规的格式，其中的每个 Cookie 之间用逗号（,）分隔，Cookie 内部的每个属性之间用分号（;）和空格分隔。

7.2.3 Cookie 请求头字段

如果浏览器愿意接受 Web 服务器发送过来的 Cookie 信息，它将存储 Cookie 信息，并在以后对该 Web 服务器的每次访问请求中都使用一个 Cookie 请求头字段将 Cookie 信息回送给 Web 服务器。多个 Cookie 信息可以且只能通过一个 Cookie 请求头字段回送给 Web 服务器，即对于 Web 服务器使用多个 Set-Cookie2 响应头字段发送来的多个 Cookie 信息，浏览器应将它们集中到一个 Cookie 请求头字段中回送给 Web 服务器。Cookie 请求头字段

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

中的各个 Cookie 部分的设置，都应与以前从 Set-Cookie2 响应头字段中接收到的内容一致。

浏览器每次向服务器发送访问请求时，它都要根据下面的几个规则，决定是否发送 Cookie 请求头字段，以及在 Cookie 请求头字段中附带哪些 Cookie 信息。

（1）请求的主机名是否与某个存储的 Cookie 的 Domain 属性匹配；

（2）请求的端口号是否在该 Cookie 的 Port 属性列表中；

（3）请求的资源路径是否在该 Cookie 的 Path 属性指定的目录及子目录中；

（4）该 Cookie 的有效期是否已过。

假如有多个存储的 Cookie 满足上面的规则，浏览器将把这些 Cookie 信息都添加到一个 Cookie 请求头字段中，每个 Cookie 之间用逗号（,）或分号（;）分隔。为了兼容以前的情况，目前出现在 Cookie 请求头字段中的每个 Cookie 之间最好都用分号（;）分隔，但在服务器端为了兼容未来的情况，应该也能接受 Cookie 之间以逗号（,）作为分隔符的情况。在 Cookie 请求头字段中，除了必须有“名称=值”的设置外，还可以有 Version、Path、Domain、Port 等几个属性。在 Version、Path、Domain、Port 等属性名之前，都要增加一个“$”字符作为前缀。Version 属性只能出现一次，且要位于 Cookie 请求头字段设置值的最前面，对于使用 Netscape 规范的浏览器，应将 Version 属性设置为 0。如果需要设置某个 Cookie 信息的 Path、Domain、Port 等属性，它们必须位于该 Cookie 信息的“名称=值”设置之后。Path 属性指向子目录的 Cookie 要排在 Path 属性指向父目录的 Cookie 之前。下面是一个 Cookie 头字段的例子：

```
Cookie: $Version=1; Course=Java; $Path=/it315/lesson; Course=vc;
$Path=/it315
```

由于"/it315/lesson"是"/it315"的子目录，所以 Path 属性等于"/it315/lesson"的 Cookie 信息排在了 Path 属性等于"/it315"的 Cookie 信息的前面。

7.3 在 Servlet 程序中使用 Cookie

Servlet API 中提供了一个 javax.servlet.http.Cookie 类来封装 Cookie 信息，在 HttpServletResponse 接口中定义了一个 addCookie 方法来向浏览器发送 Cookie 信息，在 HttpServletRequest 接口中定义了一个 getCookies 方法来读取浏览器回送的 Cookie 信息。

7.3.1 Cookie 类

Cookie 类用于封装 Cookie 信息，它包含有生成 Cookie 信息和提取 Cookie 信息的各个属性的方法。查看 Servlet API 帮助文档，可以看到 Cookie 类的所有方法。

1. 构造方法

Cookie 类只有一个如下的构造方法：

```
public Cookie(java.lang.String name,java.lang.String value)
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

其中的两个参数分别代表 Cookie 的名称和值，名称中不能包含任何空白字符、逗号（,）、分号（;），并且不能以$字符开头。

2. getName 方法

getName 方法用于返回 Cookie 的名称。

3. setValue 与 getValue 方法

setValue 与 getValue 方法分别用于设置和返回 Cookie 的值。

4. setMaxAge 与 getMaxAge 方法

setMaxAge 与 getMaxAge 方法分别用于设置和返回 Cookie 在浏览器客户机上保持有效的秒数。如果设置值为 0，则通知浏览器立即删除这个 Cookie 项。如果设置值为负数，则表示在浏览器关闭时删除这个 Cookie 项，这与没有调用 Cookie 对象的 setMaxAge 方法的效果一样，Cookie 项保留在接收它的浏览器进程的内存中，并且只对该浏览器进程可见，对同一台计算机上的其他浏览器进程不可见，对于这种情况，getMaxAge 方法返回-1。当设置值为正数时，浏览器将该 Cookie 项保留在客户机的硬盘上，这个 Cookie 对该客户机上的所有浏览器进程都有效，即使关闭浏览器和计算机以后，只要设置的时间还没到期，该 Cookie 就一直有效。

5. setPath 与 getPath 方法

setPath 与 getPath 方法分别用于设置和返回该 Cookie 项的有效目录路径。如果创建某个 Cookie 对象时没有设置它的 path 属性，那么该 Cookie 只对当前访问路径所属的目录及其子目录有效，如果想让某个 Cookie 项对站点的所有目录下的访问路径都有效，应调用 Cookie 对象的 setPath 方法将其 path 属性设置为“/”。

6. setDomain 与 getDomain 方法

setDomain 与 getDomain 方法分别用于设置和返回该 Cookie 项的有效域。

7. setVersion 与 getVersion 方法

setVersion 与 getVersion 方法分别用于设置和返回该 Cookie 项采用的协议版本。

8. setComment 与 getComment 方法

setComment 与 getComment 方法分别用于设置和返回该 Cookie 项的注解部分。

9. setSecure 与 getSecure 方法

setSecure 与 getSecure 方法分别用于设置和返回该 Cookie 项是否只能使用安全的协议传送。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

7.3.2 HttpServletResponse.addCookie 方法

HttpServletResponse 接口中定义了一个 addCookie 方法，它用于在发送给浏览器的 HTTP 响应消息中增加一个 Set-Cookie 响应头字段，Set-Cookie 头字段的设置值由传递给该方法的一个 Cookie 对象生成。在 Servlet 程序中可以多次调用 addCookie 方法来设置多个 Set-Cookie 头字段，尽量不要使用 HttpServletResponse.setHeader 方法来直接设置 Set-Cookie 头字段。

7.3.3 HttpServletRequest.getCookies 方法

HttpServletRequest 接口中定义了一个 getCookies 方法，它用于从 HTTP 请求消息的 Cookie 请求头字段中读取所有的 Cookie 项，将每个 Cookie 项封装成各自的 Cookie 对象后，存储在一个数组中返回。在 Servlet 程序中不应使用 HttpServletRequest.getHeader 方法来直接读取 Cookie 请求头字段，而应该使用 HttpServletRequest.getCookies 方法来读取 Cookie 信息。

7.3.4 Cookie 的综合实例

下面编写一个 Cookie 的综合示例程序，这个程序包含两个 Servlet 组件：CookieServlet1.java 和 CookieServlet2.java。CookieServlet1 中产生四个名称分别为 name、nickname、email、phone 的 Cookie 信息。name 和 nickname 这两个 Cookie 的值是通过请求参数来设置的，并且 nickname 这个 Cookie 保持有效的时间为 1 年，email 和 phone 这两个 cookie 的值是在程序中硬编码指定的。在产生 Cookie 信息之后，CookieServlet1 接着从请求消息中查找名称为 nickname 的 Cookie 信息，并根据返回结果打印出相应的问候语，CookieServlet1 还打印出请求消息中的 Cookie 头字段的值。CookieServlet2 首先创建三个名称分别为 email、phone、sign 的 Cookie 信息，并将 phone 这个 Cookie 保持有效的时间设置为 0 秒，然后打印出请求消息中的 Cookie 头字段的值和每个 Cookie 的名称。CookieServlet2 中的 email 和 phone 这两个 Cookie 曾在 CookieServlet1 程序中设置过，在 CookieServlet2 程序再次进行设置，是为了便于查看设置同名 Cookie 的效果和了解如何删除 Cookie。

Cookie 的工作细节

01 按上面描述的功能编写如例程 7-1 和例程 7-2 所示的两个 Servlet 程序。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

例程 7-1 CookieServlet1.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class CookieServlet1 extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out = response.getWriter();

        String name = request.getParameter("name");
        String nickname = request.getParameter("nickname");
        if(name == null || nickname == null)
        {
            out.println("请传递参数name和nickname后继续实验！");
            return;
        }

        /*下面的程序代码最好改成如下结构，以增强代码的阅读性：
            if (…)
            {
                …
                return;
            }
            …
        因为笔者在后来校对书稿的过程中，读到下面的if从句时，脑海里面很自然
        地出现了“if从句执行完后，程序继续干了什么？”的疑问，如果在if从句
        中增加了return语句，显然就不会产生这种疑问。
        */
        if("".equals(name.trim()) || "".equals(nickname.trim()))
        {
            out.println("name和nickname不能为空白串");
        }
        else
        {
            Cookie ckName = new Cookie("name",name);
            Cookie ckNickname = new Cookie("nickname",nickname);
            ckNickname.setMaxAge(365*24*3600);
            Cookie ckEmail = new Cookie("email","test1@it315.org");
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        Cookie ckPhone = new Cookie("phone","1111111");
        response.addCookie(ckName);
        response.addCookie(ckNickname);
        response.addCookie(ckEmail);
        response.addCookie(ckPhone);
    }

    /*应将下面的代码设计成一个专门用于查询某个名称的Cookie的方法，
    然后用这个方法来查询名称为nickname的Cookie值。*/
    String lastNickname = null;
    Cookie [] cks = request.getCookies();
    for(int i=0; cks!=null && i<cks.length; i++)
    {
        if("nickname".equals(cks[i].getName()))
        {
            lastNickname = cks[i].getValue();
            break;
        }
    }
    if(lastNickname != null)
    {
        out.println("欢迎您，<b><i>" + lastNickname + "</i></b>! <br>");
    }
    else
    {
        out.println("欢迎您，新客人! <br>");
    }

    String cookieHeader = request.getHeader("Cookie");
    if(cookieHeader != null)
    {
        out.println("请求头中的Cookie头字段如下:<br>");
        out.println("Cookie: " + cookieHeader + "<br>");
    }
    else
    {
        out.println("请求头中没有Cookie头字段<br>");
    }
  }
}
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

例程 7-2 CookieServlet2.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class CookieServlet2 extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out = response.getWriter();

        Cookie ckEmail = new Cookie("email","test2@it315.org");
        //ckEmail.setPath("/");
        Cookie ckPhone = new Cookie("phone","2222222");
        //ckPhone.setPath("/");
        ckPhone.setMaxAge(0);
        Cookie ckTest = new Cookie("sign","it315");
        response.addCookie(ckEmail);
        response.addCookie(ckPhone);
        response.addCookie(ckTest);

        String cookieHeader = request.getHeader("Cookie");
        if(cookieHeader != null)
        {
            out.println("请求头中的Cookie头字段如下:<br>");
            out.println("Cookie: " + cookieHeader + "<br>");
        }
        else
        {
            out.println("请求头中没有Cookie头字段<br>");
        }

        Cookie [] cks = request.getCookies();
        for(int i=0; cks!=null && i<cks.length; i++)
        {
            out.println(cks[i].getName() + ":" + cks[i].getValue() + "<br>");
        }
    }
}
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

编译上面的两个 Java 源文件，确保编译后生成的 class 文件存放在了<tomcat 安装目录>\Webapps\it315\Web-INF\classes 目录中。

02 在<tomcat 安装目录>\Webapps\it315\Web-INF\Web.xml 文件中注册有关的 Servlet，并设置其映射 URL。在 Web.xml 文件中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>CookieServlet1</servlet-name>
    <servlet-class>CookieServlet1</servlet-class>
</servlet>
<servlet>
    <servlet-name>CookieServlet2</servlet-name>
    <servlet-class>CookieServlet2</servlet-class>
</servlet>
……
<servlet-mapping>
    <servlet-name>CookieServlet1</servlet-name>
    <url-pattern>/CookieServlet1</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>CookieServlet2</servlet-name>
    <url-pattern>/servlet/CookieServlet2</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>CookieServlet2</servlet-name>
    <url-pattern>/CookieServlet2</url-pattern>
</servlet-mapping>
```

注意 为了观察访问路径对 Cookie 的影响，上面的 CookieServlet2 被映射到了两个 URL 上，一个与 CookieServlet1 所映射路径的目录相同，另一个则位于 CookieServlet1 所映射路径的目录的子目录中。保存 Web.xml 文件后，重新启动 Tomcat。

03 在浏览器地址栏中输入如下地址：

http://localhost:8080/it315/CookieServlet1?name=zxx&nickname=javafan

浏览器中显示的结果如图 7.2 所示。

尽管 CookieServlet1 中创建了一些 Cookie 信息，但是 Servlet 程序处理某次访问请求时所创建的 Cookie 信息只对于下一次的访问请求有效，这次访问 CookieServlet1 时看到的 Cookie 信息应是 Web 服务器程序处理以前的访问请求时所创建的 Cookie。由于浏览器以前没有访问过当前 Web 站点下能产生 Cookie 信息的 Servlet 程序，因此浏览器发送的请求消息中没有 Cookie 请求头字段，所以 CookieServlet1 返回了如图 7.2 所示的结果。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

图 7.2

在 Windows 2000 操作系统中，每个用户名都有自己的配置主目录，例如，Administrator 用户的主目录为 C:\Documents and Settings\Administrator。在每个用户的配置主目录中都有一个名为 Cookies 的子目录。当 IE 浏览器将 Cookie 信息保存到文件系统中时，它会在 Cookies 目录中为产生该 Cookie 信息的站点创建一个单独的文本文件，并将 Cookie 信息保存在该站点的文本文件中。进入当前登录用户的 Cookies 目录，也许能够看到许多文本文件，这是 IE 浏览器为保存你平时上网浏览的一些站点的 Cookie 信息而创建的。用 Windows 资源管理器窗口的“详细资料”视图显示出 Cookies 目录中的文件列表，并根据修改时间倒序排列所有文件，找到最新创建的文本文件，如图 7.3 所示。

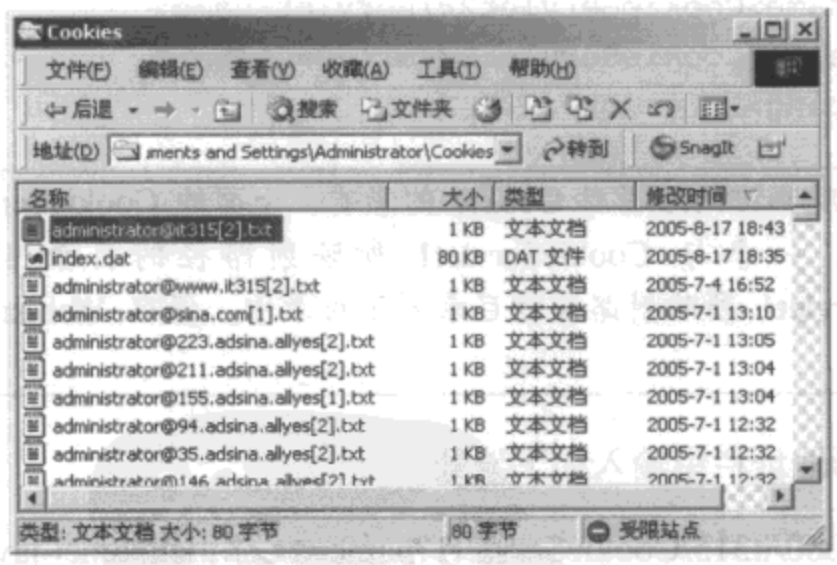


图 7.3

用 UltraEdit 软件打开这个新创建的文本文件，看到内容如下：

```
nickname
javafan
localhost/it315/
1024
176854144
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
29802986
2454270240
29729560
*
```

在浏览器新创建的文本文件中只保存了名为 nickname 的 Cookie 信息，而没有保存其他名称的 Cookie 信息，这是因为在 CookieServlet1 程序中只设置了 nickname 这个 Cookie 保持有效的时间，而没有设置其他 Cookie 保持有效的时间，除了 nickname 以外的其他 Cookie 都将在浏览器关闭时消失。

04 保持打开图 7.2 中的浏览器窗口，并重新启动一个浏览器进程，在新的浏览器进程中输入如下地址：

http://localhost:8080/it315/CookieServlet1?name=flx&nickname=jspfan

浏览器中显示的结果如图 7.4 所示。

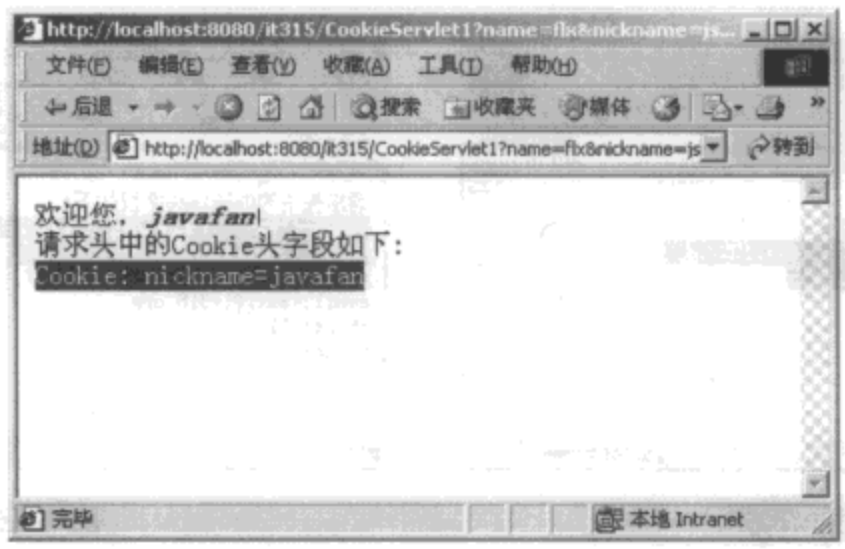


图 7.4

从图 7.4 中可以看到，在新启动的浏览器进程发送的请求消息中包含有 Cookie 请求头字段，这是因为在第 **03** 步中已经访问过了 CookieServlet1，CookieServlet1 运行时产生了 Cookie 信息并发送给了浏览器，浏览器在这次访问时便通过 Cookie 头字段将 Cookie 信息传回给 Web 服务器。但是，请求消息中的 Cookie 头字段只包含了名称为 nickname 的 Cookie 信息，这是因为在 CookieServlet1 程序中设置了 nickname 这个 Cookie 保持有效的时间，nickname 这个 Cookie 对该客户机上的所有浏览器进程都有效，CookieServlet1 程序中没有设置其他 Cookie 保持有效的时间，其他 Cookie 只对接收它的浏览器进程可见，对同一台主机上的其他浏览器进程不可见。即使重新启动计算机后再访问 CookieServlet1，看到的效果也将和图 7.4 一样。CookieServlet1 程序总是能知道该客户机浏览器上次使用的 nickname，所以这被称之为“永久 Cookie 的机制”，这也是 Web 站点上经常用到的一个技术，它能在网页中显示出或在 FORM 表单中自动填写用户上次访问时的信息。

05 刷新图 7.2 中的浏览器的访问，浏览器中显示的结果如图 7.5 所示。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

由于图 7.4 中的浏览器进程访问 CookieServlet1 时，已经将名称为 nickname 的 Cookie 值设置成了“jspfan”，而 nickname 这个 Cookie 又是被同一个客户机上的所有浏览器进程共享的，所以，图 7.2 中的浏览器进程再次访问 CookeServlet1 时，它回送给 Web 服务器的 nickname 这个 Cookie 的值就成了“jspfan”。尽管图 7.4 中的浏览器进程访问 CookeServlet1 时，名称为 name 的 Cookie 被设置成了“flx”，但是，图 7.2 中的浏览器进程再次访问 CookeServlet1 时，它回送给 Web 服务器的 name 这个 Cookie 的值还是“zxx”，即它先前访问 CookeServlet1 时所接收到的值，这是因为 CookeServlet1 没有设置 name 这个 Cookie 的保持有效的时间，名称为 name 的 Cookie 在各个浏览器进程中都是相互独立的。

06 使用图 7.2 中的浏览器访问 CookieServlet2 映射到子目录下的路径，即在浏览器地址栏中输入如下地址：

http://localhost:8080/it315/servlet/CookieServlet2

浏览器中显示的结果如图 7.6 所示。

图 7.5

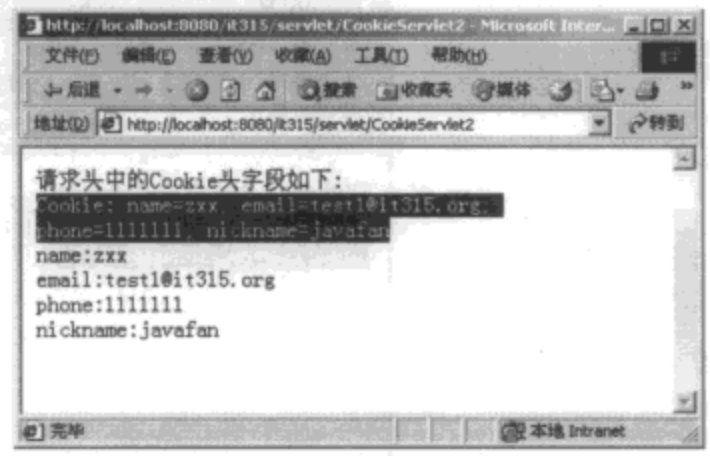


图 7.6

图 7.6 中显示出的请求消息中的 Cookie 头字段，正是在第 **05** 步操作中刷新访问 CookieServlet1 时由 CookieServlet1 所产生的 Cookie 信息，这时候名称为 nickname 的 Cookie 值又被设置成了“javafan”。这说明浏览器在访问子目录下的路径时，将把访问父目录下的路径时所接收到的 Cookie 信息回传给 Web 服务器。

07 刷新图 7.6 中的浏览器的访问，浏览器中显示的结果如图 7.7 所示。

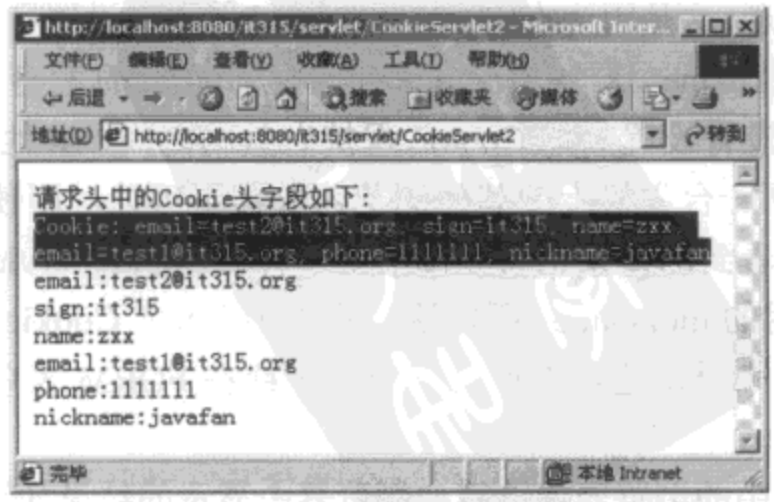


图 7.7

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

图 7.7 中显示出的请求消息中的 Cookie 头字段，比图 7.6 中的多了两个 Cookie 项，这是在第 06 步操作中访问 CookieServlet2 时由 CookieServlet2 所增加的。因为此时的 CookieServlet2 与 CookieServlet1 的访问路径的目录不同，CookieServlet2 中设置的 Cookie 项并不能覆盖 CookieServlet1 中设置的相同名称的 Cookie 项，所以，图 7.7 中显示出的请求消息的 cookie 头字段中出现了两个名称都为 email 的 Cookie 项。尽管 CookieServlet2 中将名称为 phone 的 Cookie 的保持有效的时间设置为了 0 秒，但它并不能删除 CookieServlet1 中创建的名称为 phone 的 Cookie，这也是由于访问路径的目录不同所造成的。另外，从图 7.7 中显示出的请求消息的 cookie 头字段中还可以看到，子目录下产生的 Cookie 项排在了父目录下产生的 Cookie 项之前。

08 使用图 7.7 的浏览器进程接着访问 CookieServlet2 映射到应用程序根目录下的路径，即在浏览器地址栏中输入如下地址：

http://localhost:8080/it315/CookieServlet2

浏览器中显示的结果如图 7.8 所示。

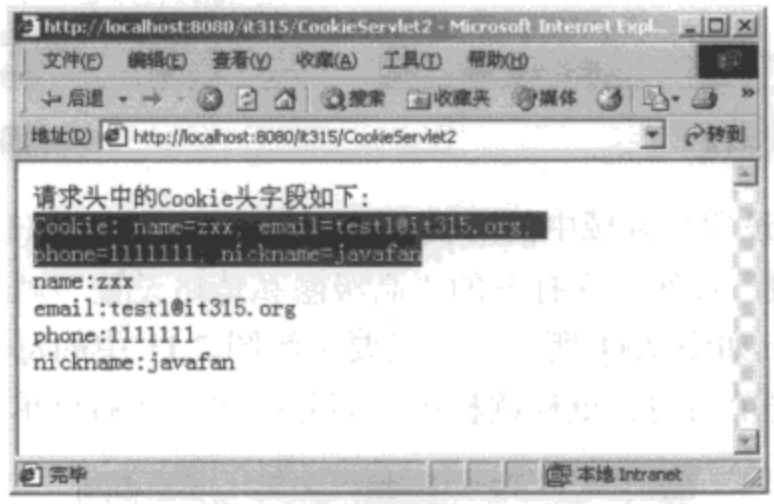


图 7.8

图 7.8 中显示出的请求消息的 cookie 头字段，比图 7.7 中的又少了两个 Cookie 信息，而与图 7.6 中的完全一样，即还是在第 05 步操作中访问 CookieServlet1 时所产生的那些 Cookie 信息，第 07 步操作中访问 CookieServlet2 时所产生的 Cookie 信息并没有出现在这次请求消息中。因为此时访问 CookieServlet2 的路径目录是上次访问 CookieServlet2 的路径目录的父目录，上次访问 CookieServlet2 时产生的 Cookie 信息不会出现在其访问目录的父目录下的资源的访问请求消息中。

09 刷新图 7.8 中的浏览器的访问，浏览器中显示的结果如图 7.9 所示。

从图 7.9 中可以看到，因为第 08 步操作中访问 CookieServlet2 的路径与第 05 步操作中访问 CookieServlet1 的路径具有相同的目录，第 08 步操作中访问 CookieServlet2 时设置的 Cookie 将覆盖第 05 步操作中访问 CookieServlet1 时所设置的相同名称的 Cookie，所以，名称为 email 的 Cookie 的值被 CookieServlet2 设置成了“test2@it315.org”，名称为 phone 的 Cookie 也因 CookieServlet2 将其保持有效的时间设置为 0 秒而被删除了。另外，CookieServlet2 中设置的名称为 sign 的 Cookie 也累加进了请求消息的 cookie 头字

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

段中。

10 IE 浏览器中提供了是否允许使用 Cookie 的设置选项，单击“工具”→“Internet 选项”菜单，单击打开的“Internet 选项”对话框的“隐私”选项卡，如图 7.10 所示。

图 7.9

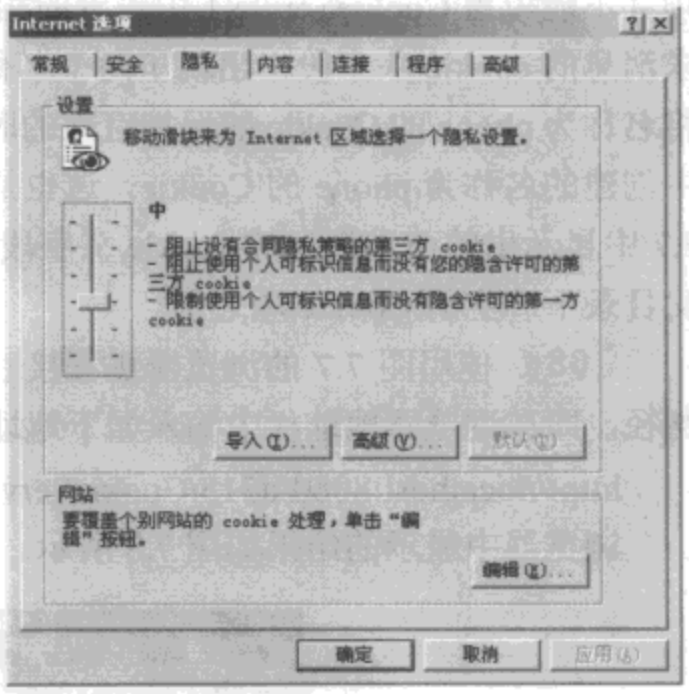


图 7.10

将图 7.10 中的“设置”区域中的滑块移动到最上端，就可以禁止浏览器使用 Cookie 功能。或者单击“高级”按钮，在打开的“高级隐私策略设置”对话框中，选择拒绝第一方和第三方的 Cookie，如图 7.11 所示。如果要了解图 7.11 中的每个选项的作用和其他元素的详细信息，可以先单击对话框标题栏中的问号按钮，然后再单击相应的选项或元素。

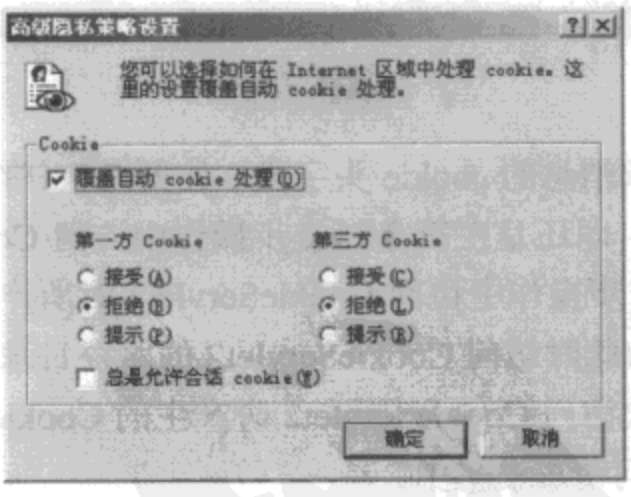


图 7.11

禁止浏览器的 Cookie 功能后，将图 7.9 中的浏览器地址栏中的 URL 地址的主机名部分修改为 Tomcat Web 服务器可对外访问的 IP 地址，例如，假设运行 Tomcat 的计算机的 IP 地址为 192.168.0.181，那么在浏览器地址栏中使用的 URL 地址如下：

http://192.168.0.181:8080/it315/CookieServlet2

按照这个地址发出访问请求，并刷新几次。由于 Web 服务器不可能知道浏览器是否禁

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

用了 Cookie 功能，它依然会向浏览器发送 Cookie 信息，但是，由于浏览器已被设置为禁用 Cookie 功能，它在请求消息中将不再把 Web 服务器发送的 Cookie 信息回传给 Web 服务器，所以，浏览器中显示的结果始终如图 7.12 所示。

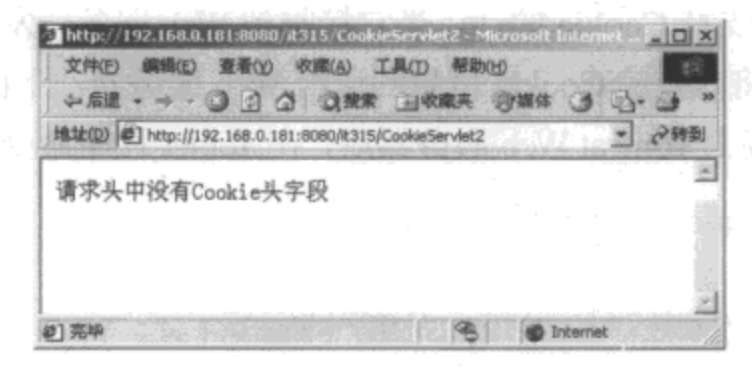


图 7.12

注意 笔者经过实验发现，即使在 IE 浏览器中设置了禁用 Cookie 功能，但如果访问 URL 中的主机名部分为本地回路地址 127.0.0.1 或是被解析成这个地址的主机名（例如，localhost），IE 浏览器仍然能够接受 Web 服务器发送的 Cookie 信息，并向 Web 服务器回传这些 Cookie 信息。

11 使用浏览器访问 CookieServlet1 和 CookieServlet2 时，都无法看到 Cookie 信息在 HTTP 响应消息中是如何传递的，只要使用 Telnet 程序来访问一下 CookieServlet2，就可以直观地看到 Cookie 信息的传递细节了。在本地计算机上使用 telnet 命令接到 Tomcat Web 服务器监听的 8080 端口，在成功连接后启动的 Telnet 程序窗口中，输入如下内容：

```
GET /it315/CookieServlet2 HTTP/1.1
Host:
<空行>
```

Telnet 程序窗口中显示出 Web 服务器返回的响应信息，如图 7.13 所示。

图 7.13

可见，对于每一个 Cookie 信息，在响应消息中都使用一个单独的 Set-Cookie 头字段进行传递。在 CookieServlet2 程序中调用 Cookie 对象的 setMaxAge 方法将 Cookie 项保持有效的时间设置为 0 秒，例如名称为 phone 的 Cookie，在响应消息中也会包含这个 Cookie

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

项，只是它的 Expires 属性被设置成了 1970 年 1 月 1 日的 0 点 0 分。注意：这里使用的不是前面所讲解的 Max-Age 属性，而是一个 Expires 属性，这可以通过查阅有关文档规范去更进一步地了解。

12 如果客户端支持 Cookie 的话，当它接收到 Web 服务器发送来的 Cookie 信息后，在以后的访问请求中应将有效的 Cookie 信息回送给 Web 服务器，下面模拟一下这种的情况。使用 telnet 命令重新连接上 Tomcat Web 服务器后，在 Telnet 程序命令窗口中，输入如下内容：

```
GET /it315/CookieServlet2 HTTP/1.1
Host:
Cookie: email=test2@it315.org; sign=it315
<空行>
```

Telnet 程序窗口中显示出 Web 服务器返回的响应信息，如图 7.14 所示。

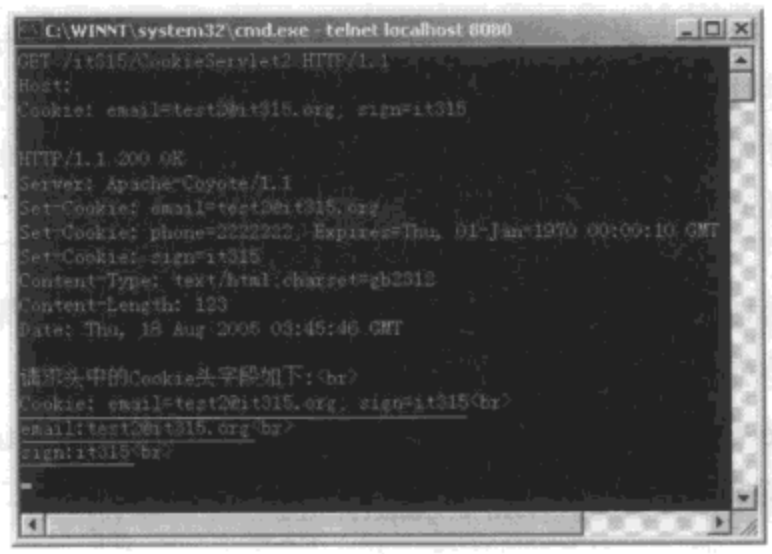


图 7.14

从图 7.14 中可以看到，Web 服务器程序中获得的每个 Cookie 正是从请求消息的 Cookie 头字段的设置值中分解出来的。

7.4 Session

7.4.1 什么是 Session

使用 Cookie 和附加 URL 参数都可以将上一次请求的状态信息传递到下一次请求中，但是如果传递的状态信息较多，将极大降低网络传输效率和增大服务器端程序处理的难度，即使这样，传递的信息量也是非常有限的。为此，各种服务器端的开发方案都提供了一种将会话状态保存在服务器端的技术，即 Session 技术。

如果把 Cookie 比喻成是商场发放给顾客的优惠卡，上面记录着顾客以前的消费金额和有效期限，那么把 Session 就可以比喻成是医院发放给病人的病历卡和医院为每个病人保留的病历档案的结合方式。发放给病人的病历卡上只填写了卡号，而没有记录其他信息，

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

病人每次去该医院就诊时，都需要出示病历卡，医务人员则根据病历卡上的号码找出病人的病历档案袋和在病历档案中增添本次就诊的信息。显然，Cookie 是在客户端保持状态的方案，而 Session 则是在服务器端保持状态的方案。在 Session 技术中对应的病历卡号就是会话标识号，由于客户端需要接收、记忆和回送 Session 的会话标识号，因此，Session 可以且通常是借助 Cookie 来传递会话标识号。

7.4.2 Session 的跟踪机制

Servlet API 规范中定义了一个 HttpSession 接口，HttpSession 接口定义了各种管理和操作会话状态的方法。一个 HttpSession 对象就相当于一个病历档案袋，它是保持会话状态信息的存储结构，一个客户端在 Web 服务器端对应一个各自的 HttpSession 对象。由于创建 HttpSession 对象会消耗内存资源，Web 服务器并不会在客户端开始访问它时就创建 HttpSession 对象。只有客户端访问某个特殊的 Servlet 程序，并且这个 Servlet 程序决定与客户端开启一个会话时，Web 应用程序才会创建一个与该客户端对应的 HttpSession 对象，并为这个 HttpSession 对象分配一个独一无二的会话标识号（SessionID），然后在响应消息中将这个会话标识号传递给客户端。客户端需要记住会话标识号，并在后续的每次访问请求中都把这个会话标识号传送给 Web 服务器，Web 服务器端程序依据回传的会话标识号就知道这次请求是哪个客户端发出的，从而选择与之对应的 HttpSession 对象。

Web 应用程序创建了与某个客户端对应的 HttpSession 对象后，只要没有超出一个限定的空闲时间段，HttpSession 对象就驻留在 Web 服务器内存之中，该客户端此后访问任意的 Servlet 程序时，它们都使用与客户端对应的那个已存在的 HttpSession 对象。HttpSession 接口中专门定义了一个 setAttribute 方法来将对象存储到 HttpSession 对象中，还定义了一个 getAttribute 方法来检索存储在 HttpSession 对象中的对象，由于属于同一个会话的各个请求的处理程序共享同一个 HttpSession 对象，所以，存储进 HttpSession 对象中的对象可以被属于同一个会话的各个请求的处理程序共享。Session 是实现网上商城的购物车的最佳方案，存储在某个客户 Session 中的一个集合对象可以充当该客户的一个购物车。当每个客户登录网上商城时，服务器端的登录处理程序可以为他创建一个集合对象，并把这个集合对象保存到 Session 中，这个集合对象就相当于一个购物车了，客户以后选择的每件商品都加入到他自己的这个集合对象中，在购物结束时再从这个集合对象中取出他选择的所有商品进行结算。

在 ServletRequest、HttpSession 和 ServletContext 等对象中都可以存储对象，读者应注意区分存储在它们里面的对象的作用范围。

7.4.3 Session 的超时管理

在客户端第一次访问某个能开启会话功能的资源时，Web 服务器就会创建一个与该客

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

户端对应的 HttpSession 对象。在 HTTP 协议中，Web 服务器无法判断当前的客户端浏览器是否还会继续访问，也无法检测客户端浏览器是否关闭，所以，即使客户已经离开或关闭了浏览器，Web 服务器也还要保留与之对应的 HttpSession 对象。显然，“只要关闭了浏览器，Session 也就消失了”的这种常见说法是错误的。随着时间的推移而不断增加新的访问客户端，而客户端浏览器的关闭又不会导致服务器端的 HttpSession 对象消失，Web 服务器内存中将会因此积累起大量不再被使用的 HttpSession 对象，并将最终导致服务器内存耗尽。

为此，Web 服务器采用“超时限制”的办法来判断客户端是否还在继续访问，如果某个客户端在一定的时间之内没有发出后续请求，Web 服务器则认为客户端已经停止了活动，结束与该客户端的会话并将与之对应的 HttpSession 对象变成垃圾，等待垃圾收集器将其彻底从内存中清除。如果客户端浏览器超时后再次发出访问请求，Web 服务器则认为这是一个新的会话的开始，将为之创建新的 HttpSession 对象和分配新的会话标识号。

会话的超时间隔可以在 Web.xml 文件中设置，其默认值由 Servlet 容器定义。在<tomcat 安装目录>\conf\Web.xml 文件中，可以找到如下一段配置信息：

```
<session-config>
<session-timeout>30</session-timeout>
</session-config>
```

这里设置的时间值是以分钟为单位的，即 Tomcat 服务器的默认会话超时间隔为 30 分钟。如果将<session-timeout>元素中的时间值设置成 0 或一个负数，则表示会话永不超时。由于<tomcat 安装目录>\config\Web.xml 文件对站点内的所有 Web 应用程序都起作用，如果想单独设置某个 Web 应用程序的会话超时间隔，应该在其自己的 Web.xml 文件中进行设置。

7.4.4 HttpSession 接口中的方法

HttpSession 接口中除了定义了各种管理和操作会话状态的方法，还定义了向当前 HttpSession 对象中存储对象和检索存储在 HttpSession 对象中的对象的方法。查看 Servlet API 文档帮助，可以看到 HttpSession 类的所有方法。

1. getId 方法

getId 方法用于返回与当前 HttpSession 对象关联的会话标识号。

2. getCreationTime 方法

getCreationTime 方法用于返回当前 HttpSession 对象的创建时间，返回值是一个自 1970 年 1 月 1 日的 0 点 0 分 0 秒开始计算的毫秒数。

3. getLastAccessedTime 方法

getLastAccessedTime 方法用于返回当前 HttpSession 对象的上一次被访问的时间，返

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

回值是一个自 1970 年 1 月 1 日的 0 点 0 分 0 秒开始计算的毫秒数。

4. setMaxInactiveInterval 方法

setMaxInactiveInterval 方法用于设置当前 HttpSession 对象可空闲的以秒为单位的最长时间，也就是修改当前会话的默认超时间隔。当某个 HttpSession 对象在超过 setMaxInactiveInterval 设置的时间值后还没有接收到后续的访问请求，该 HttpSession 对象将失效，会话过程彻底结束。如果将会话超时间隔设置成一个负数，则表示会话永不超时。

5. getMaxInactiveInterval 方法

getMaxInactiveInterval 方法用于返回当前 HttpSession 对象可空闲的以秒为单位的最长时间，也就是当前会话的超时间隔。

6. isNew 方法

isNew 方法返回当前 HttpSession 对象是否是新创建的，如果是则返回 true，否则返回 false。如果客户端请求消息中返回了一个与 Servlet 程序当前获得的 HttpSession 对象的会话标识号相同的会话标识号，则认为这个 HttpSession 对象不是新建的，isNew 方法返回 false。在下列情况下，HttpSession 对象都需要新建，isNew 方法返回 true：

（1）请求消息中没有通过任何方式返回会话标识号，这种情况发生在某个客户端浏览器首次访问某个能开启会话功能的 Servlet 程序时。

（2）请求消息中通过某种方式返回了会话标识号，但是返回的会话标识号与当前 HttpSession 对象中保存的会话标识号不匹配，这种情况发生在客户端浏览器超时后再次访问某个能开启会话功能的 Servlet 程序时。

7. invalidate 方法

invalidate 方法用于强制当前 HttpSession 对象无效，Web 服务器则可立即释放该 HttpSession 对象，而不用等到超时后才释放该 HttpSession 对象。尽管一个 HttpSession 对象在会话空闲超时后将自动释放，但是，为了减少服务器端的 HttpSession 对象的数量，节省服务器端的资源开销，最好在不需要某个 HttpSession 对象时显式地调用 invalidate 方法来将它释放掉。在一些 Web 应用中通常都会提供一个注销操作，它的内部实现就是调用 HttpSession 对象的 invalidate 方法来释放当前 HttpSession 对象，从而结束与客户端的会话过程。

8. getServletContext 方法

getServletContext 方法用于返回当前 HttpSession 对象所属于的 Web 应用程序对象，即代表当前 Web 应用程序的 ServletContext 对象。

9. setAttribute 方法

setAttribute 方法用于将一个对象与一个名称关联后存储进当前的 HttpSession 对象中，

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

这种存储在 HttpSession 对象中的对象称之为 Session 域范围的属性。如果 HttpSession 对象中已经存在指定名称的属性，setAttribute 方法先删除原来的属性后再增加新的属性。如果传递给 setAttribute 方法的属性值对象为 null，则删除指定名称的属性，这时的效果等同于 removeAttribute 方法。

10. getAttribute 方法

getAttribute 方法用于从当前 HttpSession 对象中返回指定名称的属性对象。与 ServletRequest 接口的实现类一样，HttpSession 接口的实现类通常都是在其内部定义一个 HashMap 类型的成员变量，setAttribute 方法和 getAttribute 方法则是向这个 HashMap 对象中增加成对的名称/值对象和根据名称检索值对象。

11. removeAttribute 方法

removeAttribute 方法用于从当前 HttpSession 对象中删除指定名称的属性。

12. getAttributeNames 方法

getAttributeNames 方法用于返回一个包含当前 HttpSession 对象中的所有属性名的 Enumeration 对象，在此基础上，可以对当前 HttpSession 对象中的所有属性进行遍历处理。

7.4.5 HttpServletRequest 接口中的 Session 方法

Session 是与每个请求消息紧密相关的，HttpServletRequest 定义了一些与 Session 相关的方法。

1. getSession 方法

getSession 方法用于返回与当前请求相关的 HttpSession 对象，该方法有两种重载形式：

- public HttpSession getSession(boolean create)
- public HttpSession getSession()

当调用这两个方法时，如果客户端的请求消息中包含有 SessionID，服务器检索并返回与这个 SessionID 对应的 HttpSession 对象。如果请求消息中没有 SessionID（通常发生在会话的第一次请求的情况）或服务器端不存在与 SessionID 对应的 HttpSession 对象（通常发生在以前的 HttpSession 对象已超时的情况），第一个方法根据传递的参数来决定是否创建新的 HttpSession 对象，如果参数为 true，则在相关的 HttpSession 对象不存在时创建并返回新的 HttpSession 对象，否则不创建新的 HttpSession 对象，而是返回 null。第二个方法相当于第一个方法的参数为 true 时的情况，在相关的 HttpSession 对象不存在时总是创建新的 HttpSession 对象。注意，由于 getSession 方法可能会产生发送会话标识号的 Cookie 头字段，因此必须在发送任何响应内容之前调用 getSession 方法。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

多想一想

什么情况下该调用 request.getSession(false)方法

很多 Web 开发人员经常困惑这样的一个问题，即不知道在什么情况下该调用 request.getSession(false)方法？有时候，为了节省 Web 服务器端的内存资源，不要创建一些无价值的 HttpSession 对象，系统需要限定一个会话只能从某个 Servlet 程序开始，而不能从其他 Servlet 程序开始，例如，一些 Web 应用程序要求只有用户成功登录后才真正开启与客户端的会话过程。为了简化系统的复杂性和易维护性，系统可以限定只有某一个 Servlet 程序可以调用 request.getSession()方法创建 HttpSession 对象，而其他的 Servlet 程序则只能调用 request.getSession(false)方法来获得 HttpSession 对象，如果其他 Servlet 程序调用 request.getSession(false)方法未获得 HttpSession 对象则跳转到登录页面。这样，可以确保其他 Servlet 程序不会创建 HttpSession 对象，只有用户成功登录后，那个限定的 Servlet 程序才会创建 HttpSession 对象。如果其他 Servlet 程序中调用的是 request.getSession()方法，那么只要用户访问了其他 Servlet 程序，即使用户没有成功登录，但与该客户端对应的 HttpSession 对象却被创建了，这就造成了一些不必要的内存开销。

2. isRequestedSessionIdValid 方法

isRequestedSessionIdValid 方法返回一个指示请求消息中的 SessionID 是否有效的 boolean 值，譬如，当后续请求超出了 HttpSession 对象的最大空闲时间间隔，请求消息中的 SessionID 将被视为无效，isRequestedSessionIdValid 方法返回 false。

3. isRequestedSessionIdFromCookie 方法

isRequestedSessionIdFromCookie 方法用于判断 SessionID 是否是通过请求消息中的 Cookie 传递过来的。

4. isRequestedSessionIdFromURL 方法

isRequestedSessionIdFromURL 方法用于判断 SessionID 是否是通过请求消息的 URL 参数传递过来的。

7.4.6 application 域与 session 域属性的比较

在 ServletContext 对象和 HttpSession 对象中设置的属性都可以被 Web 应用程序中的多个 Servlet 程序访问，从而可实现在多个 Servlet 程序之间传递和共享信息的功能，在 ServletContext 对象和 HttpSession 对象中设置的属性有什么具体的区别呢？

同一个 Web 应用程序中的所有 Servlet 程序得到的是同一个 ServletContext 对象，所以存储在 ServletContext 对象中的属性可以被所有的 Servlet 程序访问和共享，而不管访问来

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

自哪个客户端，如图 7.15 所示，其中 application 为 ServletContext 的实例对象名。

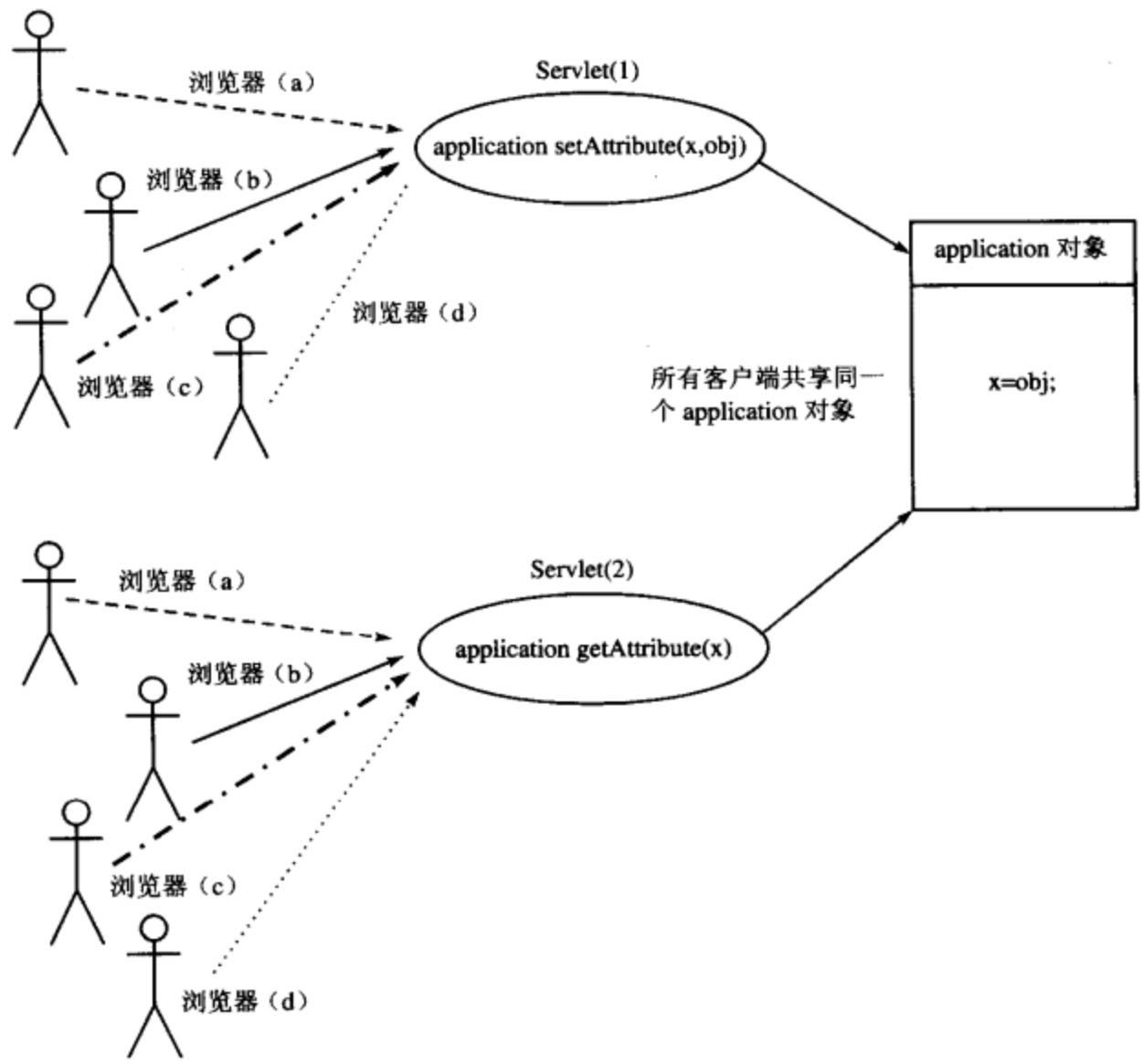


图 7.15

存储在 HttpSession 对象中的属性也可以被所有的 Servlet 程序访问，但仅仅是来自同一个客户端的一组访问才能共享同一个 HttpSession 对象，来自不同客户端的各组访问共享的 HttpSession 对象是不同的，如图 7.16 所示。

从图 7.16 中可以看到，各个 Servlet 程序在处理属于同一个会话的各个访问请求时，它们得到的是同一个 HttpSession 对象，在某个 Servlet 程序中调用 setAttribute 方法存储进 HttpSession 对象中的属性，可以被处理同一个会话中的其他访问请求的 Servlet 程序调用 getAttribute 方法获取，这样就可以实现在同一个会话的各个请求之间传递信息的功能。但是，即使是同一个 Servlet 程序中的同一段调用 HttpSession 对象的程序代码，对于不同客户端的访问请求，它们所调用的 HttpSession 对象是不同的，它们调用的是各个客户端各自对应的 HttpSession 对象。例如，同一个 Servlet 程序中的 getSession 语句，对于不同客户的访问请求，它返回的是各个客户端各自对应的 HttpSession 对象，因此从 HttpSession 对象中存储和检索的对象代表的是各个客户端各自的会话状态。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

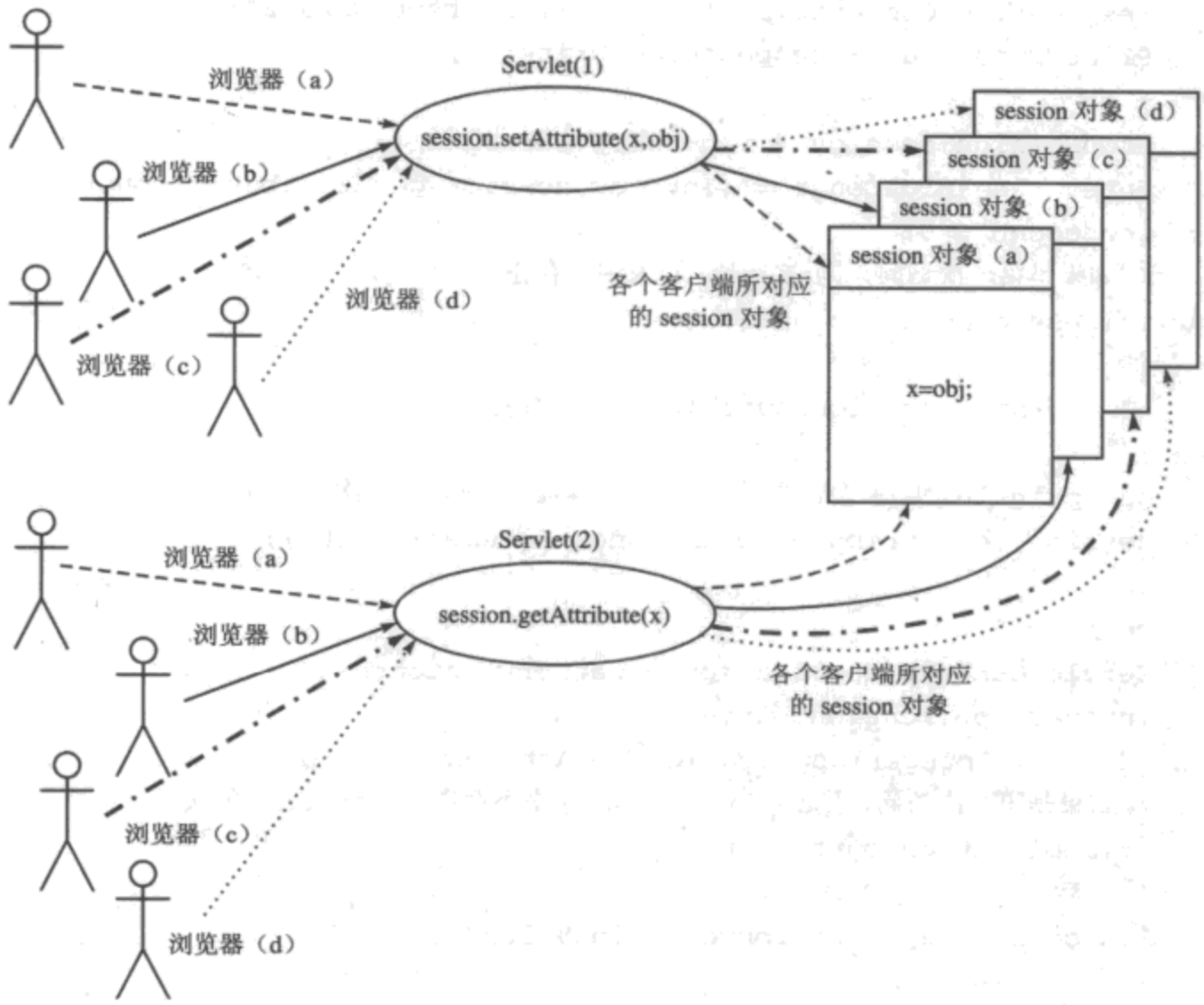


图 7.16

application 域和 session 域范围的属性的差异

01 按例程 7-3 和例程 7-4 编写两个 Servlet 程序，在它们每次被访问时分别将存储在 ServletContext 对象和 HttpSession 对象中的两个计数变量加 1。两个 Servlet 的程序代码基本上一样，只是最后的相互链接的代码不一样。

例程 7-3 SessionServlet1.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SessionServlet1 extends HttpServlet
{
public void service(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException
{
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
    response.setContentType("text/html;charset=gb2312");
    PrintWriter out = response.getWriter();

    HttpSession session = request.getSession();
    Integer sessionCount = (Integer)session.getAttribute("count");
    int count = 0;
    //如果是第一次访问，则session对象中不存在名为"count"的属性。
    if(sessionCount != null)
    {
        count = sessionCount.intValue();
    }
    out.println("当前会话中发生了" + (++count) + "次访问<br>");
    session.setAttribute("count",new Integer(count));

    count = 0;
    ServletContext application = getServletContext();
    Integer applicationCount =
            (Integer)application.getAttribute("count");
    //如果是第一次访问，则application对象中不存在名为"count"的属性。
    if(applicationCount != null)
    {
        count = applicationCount.intValue();
    }
    out.println("Web应用程序中发生了" + (++count) + "次访问<br>");
    application.setAttribute("count",new Integer(count));

    out.println("<a href='SessionServlet2'>访问SessionServlet2</a>");
}
}
```

例程 7-4　SessionServlet2.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SessionServlet2 extends HttpServlet
{
public void service(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException
{
    response.setContentType("text/html;charset=gb2312");
    PrintWriter out = response.getWriter();
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
    HttpSession session = request.getSession();
    Integer sessionCount = (Integer)session.getAttribute("count");
    int count = 0;
    //如果是第一次访问，则session对象中不存在名为"count"的属性。
    if(sessionCount != null)
    {
        count = sessionCount.intValue();
    }
    out.println("当前会话中发生了" + (++count) + "次访问<br>");
    session.setAttribute("count",new Integer(count));

    count = 0;
    ServletContext application = getServletContext();
    Integer applicationCount =
            (Integer)application.getAttribute("count");
    //如果是第一次访问，则application对象中不存在名为"count"的属性。
    if(applicationCount != null)
    {
        count = applicationCount.intValue();
    }
    out.println("Web应用程序中发生了" + (++count) + "次访问<br>");
    application.setAttribute("count",new Integer(count));

    out.println("<a href='SessionServlet1'>访问SessionServlet1</a>");
}
}
```

编译上面的两个 Java 源文件，确保编译后生成的 class 文件存放在了<tomcat 安装目录>\Webapps\it315\Web-INF\classes 目录中。

02 在<tomcat 安装目录>\Webapps\it315\Web-INF\Web.xml 文件中注册有关的 Servlet，并设置其映射 URL。在 Web.xml 文件中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>SessionServlet1</servlet-name>
    <servlet-class>SessionServlet1</servlet-class>
</servlet>
<servlet>
    <servlet-name>SessionServlet2</servlet-name>
    <servlet-class>SessionServlet2</servlet-class>
</servlet>
……
<servlet-mapping>
    <servlet-name>SessionServlet1</servlet-name>
    <url-pattern>/servlet/SessionServlet1</url-pattern>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
</servlet-mapping>
<servlet-mapping>
    <servlet-name>SessionServlet2</servlet-name>
    <url-pattern>/servlet/SessionServlet2</url-pattern>
</servlet-mapping>
```

保存 Web.xml 文件后，重新启动 Tomcat。

03 在浏览器地址栏中输入如下地址：

http://localhost:8080/it315/servlet/SessionServlet1

在返回的页面中单击访问 SessionServlet2 的超链接，然后再单击 SessionServlet2 页面中的超链接访问 SessionServlet1，如此重复多次，期间还可刷新 SessionServlet1 和 SessionServlet2 页面的访问，结果如图 7.17 所示。

从图 7.17 中可以看到，浏览器窗口中显示出的“当前会话中发生的访问次数”与“Web 应用程序中发生的访问次数”总是一样，每操作一次，它们的计数值就都加 1。如果当前会话发生的访问次数总是 1，请将浏览器设置为允许 Cookie 功能。

04 重新启动一个浏览器进程，执行与第 **03** 步类似的操作，浏览器窗口中显示出的“当前会话中发生的访问次数”与“Web 应用程序中发生的访问次数”不再相同，Web 应用程序中发生的访问次数”为两个浏览器程序中的“当前会话中发生的访问次数”的总和。再回到第一个浏览器程序中执行 3 次访问，可以看到如图 7.18 所示的结果。

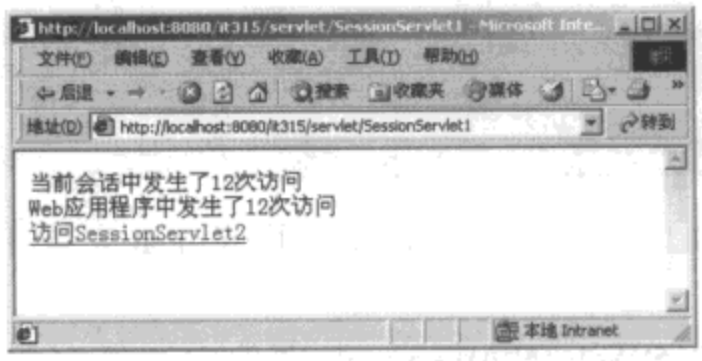


图 7.17

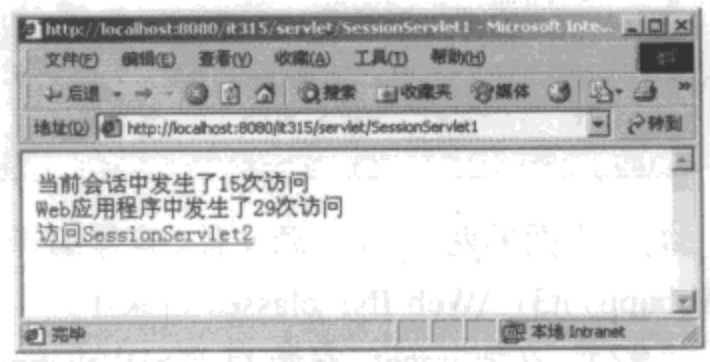


图 7.18

7.4.7 利用 Cookie 实现 Session 跟踪

如果 Web 服务器处理某个访问请求时创建了新的 HttpSession 对象，它将把会话标识号作为一个 Cookie 项加入到响应消息中，通常情况下，浏览器在随后发出的访问请求中又将会话标识号以 Cookie 的形式回传给 Web 服务器。Web 服务器端程序依据回传的会话标识号就知道以前已经为该客户端创建了 HttpSession 对象，不必再为该客户端创建新的 HttpSession 对象，而是直接使用与该会话标识号匹配的 HttpSession 对象，通过这种方式就实现了对同一个客户端的会话状态的跟踪。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

如何利用 Cookie 实现 Session 跟踪

01 接着 7.4.6 节编写和部署的 Servlet 例子程序的基础上，在本地计算机上使用 telnet 命令接到 Tomcat Web 服务器监听的 8080 端口，在成功连接后启动的 Telnet 程序窗口中，输入如下内容：

```
GET /it315/servlet/SessionServlet1 HTTP/1.1
Host:
<空行>
```

Telnet 程序窗口中显示出 Web 服务器返回的响应信息，如图 7.19 所示。

图 7.19

从图 7.19 中可以看到，Tomcat 服务器发送给客户端的会话标识号的 Cookie 名称为 JSESSIONID。JSESSIONID 这个 Cookie 项的 path 属性为当前 Web 应用程序的根目录，这说明浏览器在访问其他 Web 应用程序时将不回送该 Cookie 信息，因此，Tomcat 的两个 Web 应用程序不能共享会话状态，它们与同一个客户端浏览器保持各自独立的会话。

02 还是在上面的 Telnet 程序窗口中，或者在一个新连接上的 Telnet 程序窗口中输入如下内容：

```
GET /it315/servlet/SessionServlet1 HTTP/1.1
Host:
Cookie: JSESSIONID=A2B6D0F56F2022AD4828E30619148ED9
<空行>
```

注意，这里输入的 JSESSIONID 的值要与图 7.19 中的完全一样，为了避免出错或操作太慢导致 Web 服务器因等待超时而断开连接，最好是用“复制”和“粘贴”的方式输入这段内容。不管这次请求和上次请求是否是在同一个 telnet 窗口中发出的，准确的说法是，不管这次请求与上次请求是否是在同一个 TCP 连接上发出的，Web 服务器返回的响应信息都是如图 7.20 所示的效果。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

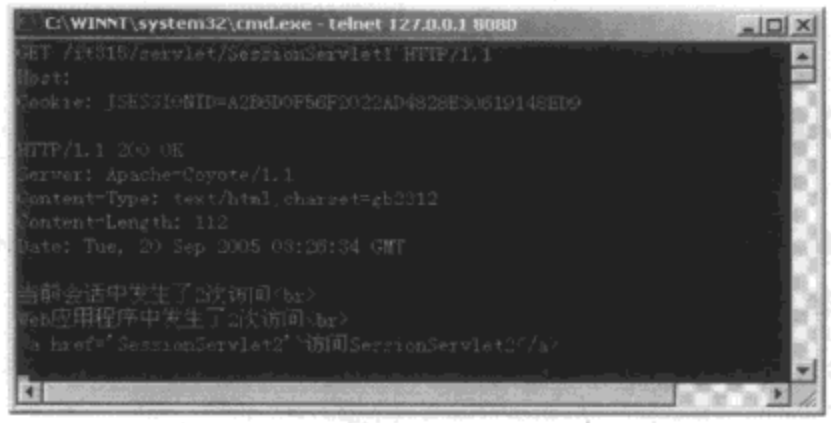


图 7.20

在图 7.20 中，Web 服务器没有再向客户端发送名称为 JSESSIONID 的 Cookie 信息，因为它认为这次请求与上次请求来自同一个客户端，从而直接使用了处理上次请求时所创建的 HttpSession 对象。

03 在 Telnet 程序窗口中再次访问 SessionServlet1，但是这次发送给 Web 服务器的请求消息中不包含 Cookie 头字段，或者是 JSESSIONID 这个 Cookie 项的值与 Web 服务器上次返回的值不一样，那么 Web 服务器将认为这次请求与上次请求不属于同一个客户端，它返回的响应信息如图 7.21 所示。

图 7.21

图 7.21 中的响应消息中又包含了一个名称为 JSESSIONID 的 Cookie 项，但是 JSESSIONID 的值与上次不一样，这是因为 Web 服务器在处理这次请求时又创建了一个新的 HttpSession 对象。从图 7.21 中还可以看到，当前会话的访问次数又从 1 开始重新计数，这也说明 Web 服务器将这次访问当做了一个新的会话。

04 禁止浏览器的 Cookie 功能后，在浏览器中以 Tomcat Web 服务器可对外访问的 IP 地址来访问 SessionServlet1，例如，假设运行 Tomcat 的计算机的 IP 地址为 192.168.0.181，那么在浏览器地址栏中输入的 URL 地址如下：

http://192.168.0.181:8080/it315/servlet/SessionServlet1

按照这个地址发出访问请求，并刷新几次或通过单击浏览器中显示的超链接交互访问 SessionServlet2 和 SessionServlet1，浏览器中显示的“当前会话中发生的访问次数”总是 1。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

这是因为如果客户端浏览器不支持 Cookie，即使它在响应消息中接收到了 Web 服务器以 Cookie 形式传递过来的会话标识号，但它在后续的访问请求中不会把这个会话标识号回传给 Web 服务器，Web 服务器将认为每个后续的访问请求都来自一个新的客户端，它在每次处理该浏览器发出的访问请求时都将创建一个新的 HttpSession 对象。尽管每次处理浏览器发出的访问请求时所创建的 HttpSession 对象都驻留在内存中，但是它们之间没有任何关联，无法真正保持该浏览器的会话信息和跟踪它的会话状态。

注意 做完禁止浏览器的 Cookie 功能的实验后，请重新打开浏览器的 Cookie 功能，否则，你以前经常访问的一些网站的某些功能可能再也无法正常使用，例如，笔者曾经就因为这个事情而无法登录 Sina 的邮箱系统。

7.4.8 利用 URL 重写实现 Session 跟踪

考虑到浏览器有可能不支持 Cookie 的情况，Servlet 规范中还引入了一种补充的会话管理机制，它允许不支持 Cookie 的浏览器也可以与 Web 服务器保持连续的会话。这种补充机制要求在响应消息的实体内容中必须包含下一次请求的超链接，并将会话标识号作为超链接的 URL 地址的一个特殊参数。

当用户单击响应消息中的超链接来发出下一次请求时，如果请求消息中没有包含 Cookie 头字段，Servlet 引擎则认为客户端不支持 Cookie，它将依据请求 URL 的参数中的会话标识号来实施会话跟踪。这种方式要求在响应消息中出现的指向当前应用程序中的资源的超链接地址后面都附带会话标识号参数，且用户必须通过超链接来提交后续的访问请求。

按照正常的网站访问规则，用户本来就应该使用网站提供的超链接进行访问，而不应该直接在浏览器地址栏中输入首页以外的地址，所以，使用这种方式来实现有状态的会话管理也是完全可行的。如果遇到用户在浏览器地址栏中直接输入其他页面地址的情况，相应的处理程序应将这种情况视为非法访问，而不要为此创建新的 HttpSession 对象。

将会话标识号以参数形式附加在超链接的 URL 地址后面的技术称为 URL 重写。Tomcat 服务器发送给客户端的会话标识号的 Cookie 名称为 JSESSIONID，因此，Tomcat 服务器中的 URL 重写就是要将 JSESSIONID 关键字作为参数名和将会话标识号作为参数值附加到 URL 后面。如果在浏览器不支持 Cookie 或者关闭了 Cookie 功能的情况下，Web 服务器还要能够与浏览器实现有状态的会话，就必须对所有可能被客户端访问的请求路径（包括超链接、FORM 表单的 action 属性设置和重定向的 URL）进行 URL 重写。HttpServletResponse 接口中定义了两个用于完成 URL 重写方法：

- **encodeURL 方法** 用于对超链接和 form 表单的 action 属性中设置的 URL 进行重写。
- **encodeRedirectURL 方法** 用于对要传递给 HttpServletResponse.sendRedirect 方法的 URL 进行重写。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

HttpServletResponse.encodeURL 方法和 HttpServletResponse.encodeRedirectedURL 方法根据请求消息中是否包含 Cookie 头字段来决定是否进行 URL 重写。

利用 URL 重写实现 Session 跟踪

01 修改 7.4.6 节编写的 SessionServlet1.java 和 SessionServlet2.java 源文件，使用 HttpServletResponse.encodeURL 方法对其中最后一条语句内的 URL 地址进行 URL 重写，两个源文件中的最后一条语句修改后的结果分别如下：

```
out.println("<a href='" + response.encodeURL("SessionServlet2") +
                                    "'>访问SessionServlet2</a>");
```

和

```
out.println("<a href='" + response.encodeURL("SessionServlet1") +
                                    "'>访问SessionServlet1</a>");
```

重新编译 SessionServlet1.java 和 SessionServlet2.java 源文件，确保编译后生成的 class 文件存放在了<tomcat 安装目录>\Webapps\it315\Web-INF\classes 目录中。

02 禁止浏览器的 Cookie 功能后重新启动一个浏览器程序，然后在浏览器的地址栏中输入如下地址：

http://192.168.0.181:8080/it315/servlet/SessionServlet1

单击浏览器的“查看”→“源文件”菜单，看到 Web 服务器返回给浏览器的网页文档源文件中包含的超链接内容如下：

```
<a href='SessionServlet2;jsessionid=9AF3BF82FF7BA21126E733E65A83F2DA'>
    访问SessionServlet2</a>
```

单击该超链接访问 SessionServlet2 后，再次单击浏览器的“查看”→“源文件”菜单，看到 Web 服务器返回给浏览器的网页文档源文件中包含的超链接内容如下：

```
<a href='SessionServlet1;jsessionid=9AF3BF82FF7BA21126E733E65A83F2DA'>
    访问SessionServlet1</a>
```

通过单击浏览器中显示的超链接交互访问 SessionServlet2 和 SessionServlet1，浏览器中都能显示出正确的“当前会话中发生的访问次数”。

03 在浏览器支持 Cookie 的情况下，HttpServletResponse.encodeURL 方法是如何进行 URL 重写的呢？打开浏览器的 Cookie 功能后重新启动一个浏览器程序，然后在浏览器的地址栏中输入如下地址：

http://192.168.0.181:8080/it315/servlet/SessionServlet1

单击浏览器的“查看”→“源文件”菜单，看到 Web 服务器返回给浏览器的网页文档源文件中包含的超链接内容如下：

```
<a href='SessionServlet2;jsessionid=849D1FE297F08597541F490D24EDB0F7'>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
访问SessionServlet2</a>
```

单击该超链接访问 SessionServlet2 后，再次单击浏览器的“查看”→“源文件”菜单，看到 Web 服务器返回给浏览器的网页文档源文件中包含的超链接内容如下：

```
<a href='SessionServlet1'>访问SessionServlet1</a>
```

接着单击该超链接访问 SessionServlet1 后，单击浏览器的“查看”→“源文件”菜单，看到 Web 服务器返回给浏览器的网页文档源文件中包含的超链接内容如下：

```
<a href='SessionServlet2'>访问SessionServlet2</a>
```

可见，即使在浏览器支持 Cookie 的情况下，HttpServletResponse.encodeURL 方法在会话的第一个响应消息中也会将会话标识号以参数形式附加到超链接的 URL 路径后面，但在后续的响应消息中都不再将会话标识号附加到超链接的 URL 路径后面。这是因为 Web 服务器在处理会话的第一个请求消息时，虽然总是将会话标识号以 Cookie 形式发送给浏览器，但是 Web 服务器并不知道浏览器是否接受 Cookie 和在下一次请求中是否能以 Cookie 形式回传会话标识号，所以，HttpServletResponse.encodeURL 方法在会话的第一个响应消息中必须进行 URL 重写。

如果浏览器支持 Cookie 功能，它在后续的请求消息中将使用 Cookie 请求头字段把会话标识号回传给 Web 服务器，Servlet 引擎据此判断出浏览器支持 Cookie 功能，HttpServletResponse.encodeURL 方法将不再对 URL 进行重写。如果浏览器不支持 Cookie 功能，它的后续请求消息中将不包含 Cookie 请求头字段，HttpServletResponse.encodeURL 方法在后续的每个响应消息中都要对 URL 进行重写。

7.5 Session 的典型案例

7.5.1 使用 Session 实现购物车

下面编写一个实现购物车功能的示例程序，这里使用课程来代替商品，选择一门课程就等于购买了一件商品，整个程序包含一个 HTML 文件和两个 Servlet 组件：logon.html，LogonServlet.java 和 CoursesServlet.java。logon.html 提供用户登录的界面；LogonServlet.java 是一个负责处理用户登录的 Servlet 程序；CoursesServlet.java 也是一个 Servlet 程序，为了减少 Servlet 程序的个数，CoursesServlet.java 完成了三项功能：显示所有可选课程的列表；将用户选择的课程放入购物车；显示购物车中的课程。在实际应用中，为了使得程序模块的结构更加清晰，最好是将 CoursesServlet.java 改写成三个 Servlet 程序来分别完成上面的三个功能。

LogonServlet.java 的登录验证方式非常简单，只要登录的用户名不为空白字符串即可；LogonServlet.java 先将成功登录的用户名保存到 Session 中，然后将请求转发给 CoursesServlet.java 显示所有可选课程的列表；如果登录失败，LogonServlet.java 则再次显

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

示登录界面。

CoursesServlet.java 的功能稍微复杂一些，它需要完成如下任务：

- 首先判断访问请求是否来自一个已登录用户，如果不是，则将请求重新定向到 logon.html 页面。
- 因为用户选择课程的访问请求也是由 CoursesServlet.java 处理的，CoursesServlet.java 接着判断当前访问请求是否是用户选择课程时发出的，如果是，则将用户选择的课程加入购物车。
- CoursesServlet.java 最后显示出所有供选择的课程列表和已放入购物车中的课程列表。

使用 Session 实现购物车功能

01 按上面描述的功能编写如例程 7-5、例程 7-6 和例程 7-7 所示的程序。

例程 7-5 logon.html

```
<base href="http://localhost:8080/it315/logon.html">
<form action="servlet/LogonServlet" method="post">
用户名: <input type="text" name="name"><br>
<input type="submit" value="登录">
</form>
```

例程 7-6 LogonServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class LogonServlet extends HttpServlet
{
public void service(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException
{
    response.setContentType("text/html;charset=gb2312");
    PrintWriter out = response.getWriter();

    HttpSession session = request.getSession();
    //设置session超时间隔为120秒，以便测试超时的情况
    session.setMaxInactiveInterval(120);
    String sessionName = (String)session.getAttribute("name");
    //如果是一个已登录用户，则将请求转发给显示课程的Servlet
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
    if(sessionName != null)
    {
        RequestDispatcher rd =
            request.getRequestDispatcher("CoursesServlet");
        rd.forward(request,response);
        return;
    }

    String paramName = request.getParameter("name");
    if(paramName == null || paramName.trim().equals(""))
    {
        out.println("请传递用户名! <br>");
        RequestDispatcher rd =
            request.getRequestDispatcher("../logon.html");
        rd.include(request,response);
    }
    else
    {
        session.setAttribute("name",paramName);
        RequestDispatcher rd =
            request.getRequestDispatcher("CoursesServlet");
        rd.forward(request,response);
    }
}
}
```

例程 7-7 CoursesServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.net.URLEncoder;
import java.util.Vector;
import java.util.Enumeration;

public class CoursesServlet extends HttpServlet
{
public void service(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException
{
    response.setContentType("text/html;charset=gb2312");
    PrintWriter out = response.getWriter();

    HttpSession session = request.getSession();
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
String sessionName = (String)session.getAttribute("name");
/*如果用户没有登录，sessionName为null，或者上次访问后的空闲时间
超过了session的限制，getSession方法返回的是一个新Session对象，
sessionName也为null*/
if(sessionName == null)
{
    /*因为下面是在同一个Web应用程序内部跳转的，最好是采用forward
    方法进行请求转发，这里仅仅是为了复习sendRedirect方法*/
    response.sendRedirect("../logon.html");
    return;
}

String courseSelect = request.getParameter("course");
if(courseSelect != null)
{
    Vector vCourses = (Vector)session.getAttribute("courses");
    if(vCourses == null)
    {
        vCourses = new Vector();
        vCourses.add(courseSelect);
        session.setAttribute("courses",vCourses);
    }
    else
    {
        if(vCourses.contains(courseSelect))
        {
            out.println(sessionName + "，你以前选择过了" +
                        courseSelect + "<hr>");
        }
        else
        {
            /*vCourses是指向Session中的一个属性对象的引用，
            对vCourses的操作直接影响那个属性对象，不用再将
            vCourses重新增加进Session中*/
            vCourses.add(courseSelect);
        }
    }
}

String [] courses = {"c","c++","vc++","java","jsp"};
out.println(sessionName + "，请选择你要选修的课程：<br>");
for(int i=0; i<courses.length; i++)
{
    //对参数中的特殊字符应进行URL编码
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        out.println(courses[i] +
            "    <a href='CoursesServlet?course=" +
            URLEncoder.encode(courses[i],"gb2312") + "'>选修</a><br>");
    }
    out.println("<hr>");

    Vector vCourses = (Vector)session.getAttribute("courses");
    out.println(sessionName + ", 你已经选择了下面的课程: <br>");
    if(vCourses != null)
    {
        for(Enumeration e=vCourses.elements();e.hasMoreElements();)
        {
            out.println((String)e.nextElement() + "<br>");
        }
    }
}
}
```

在 CoursesServlet.java 程序代码中，往 Session 的“Courses”属性对象中增加新的内容后，没有再调用 session.setAttribute("Courses",vCourses)重新设置“Courses”属性，这是因为程序中的 vCourses 变量是指向 Session 中的“Courses”属性对象的一个引用变量，对 vCourses 的操作直接影响 Session 中的那个属性对象。如果再次调用 session.setAttribute ("Courses",vCourses)的话，它将先删除原来的“Courses”属性，再增加新的“Courses”属性，这样降低了程序的运行效率。但是，在多台服务器协同工作的集群环境中，如果修改“Courses”属性对象中内容后，则必须再次调用 session.setAttribute("Courses",vCourses)进行重新设置，这是因为 Web 服务器需要通过这一动作事件来了解 Session 对象中的内容发生了改变，从而才能向其他 Web 服务器进程复制新的 Session 信息。

编译上面的两个 Java 源文件，确保编译后生成的 class 文件存放在了<tomcat 安装目录>\Webapps\it315\Web-INF\classes 目录中。将 logon.html 文件保存在<tomcat 安装目录>\Webapps\it315 目录中。

02 在<tomcat 安装目录>\Webapps\it315\Web-INF\Web.xml 文件中注册有关的 Servlet，并设置其映射 URL。在 Web.xml 文件中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>LogonServlet</servlet-name>
    <servlet-class>LogonServlet</servlet-class>
</servlet>
<servlet>
    <servlet-name>CoursesServlet</servlet-name>
    <servlet-class>CoursesServlet</servlet-class>
</servlet>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
……
<servlet-mapping>
    <servlet-name>LogonServlet</servlet-name>
    <url-pattern>/servlet/CookieServlet1</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>CoursesServlet</servlet-name>
    <url-pattern>/servlet/CoursesServlet</url-pattern>
</servlet-mapping>
```

保存 Web.xml 文件后，重新启动 Tomcat。

03 在浏览器地址栏中输入如下地址：

http://localhost:8080/it315/logon.html

在返回的 logon.html 页面中不输入任何用户名，单击登录按钮，浏览器中返回的结果如图 7.22 所示。

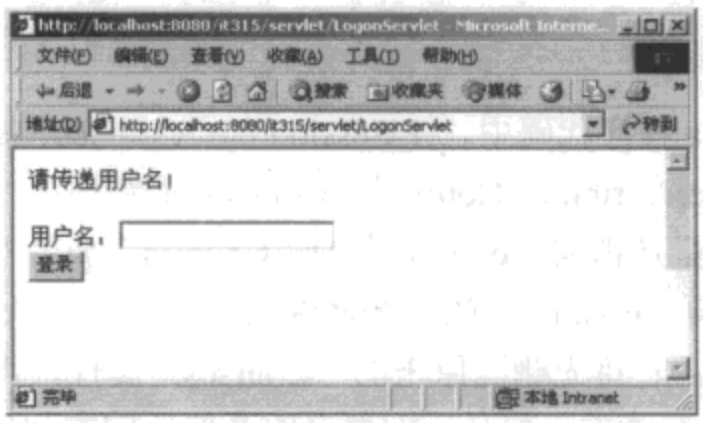


图 7.22

在图 7.22 中输入用户名“zxx”后，单击登录按钮，浏览器中返回的页面如图 7.23 所示。

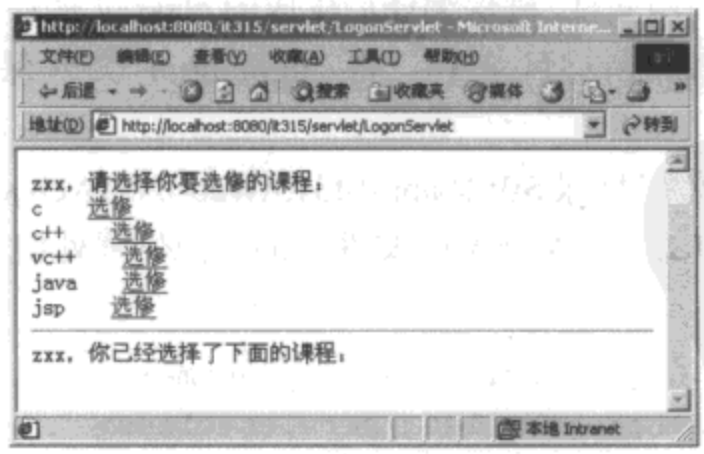


图 7.23

在图 7.23 中选择“c”和“java”后，再次选择“c”，浏览器中显示的结果如图 7.24 所示。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

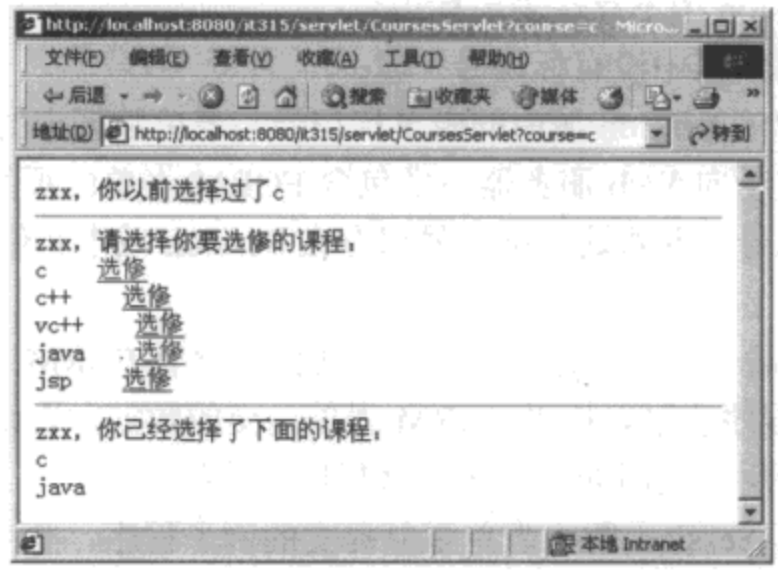


图 7.24

04 由于在 LogonServlet 中已将 Session 的超时间隔为 120s，如果休息两分钟后再接着在图 7.24 中选择课程，浏览器将显示 logon.html 页面要求用户重新登录。

05 再启动一个浏览器进程执行上面的访问过程，同时在原来的浏览器进程中继续执行访问操作，可以看到，两个浏览器进程的执行结果完全独立，互不影响，各有各的状态。可见，使用 Session 技术是处理购物车这类应用的最佳方案，Web 服务器在验证每个用户登录后都来会为他分配一个购物车（Session 对象），用户以后在其他页面选择的商品都会加入到他们各自的购物车中，购物结算页面则是取出当前用户自己的购物车中的所有商品来进行计算的。

7.5.2 利用 Session 防止表单重复提交

6.5.8 节讲解的利用 JavaScript 防止表单重复提交的方案，只能防止用户意外重复提交表单的行为，但不能阻止用户故意重复提交表单的行为，例如，一些捣乱者通过浏览器中的“后退”或“刷新”操作向论坛连续发送大量的重复信息。因为在客户端总有办法绕过浏览器和 Javascript 程序的限制，所以不能依赖于客户端技术来防止用户恶意重复提交表单，如果要阻止用户故意重复提交表单的行为，则必须通过服务器端程序来实现。

通过服务器端程序防止表单重复提交的基本原理如下：

（1）包含有 FORM 表单的页面必须通过一个服务器程序动态产生，服务器程序为每次产生的页面中的 FORM 表单都分配一个惟一的随机标识号，并在 FORM 表单的一个隐藏字段中设置这个标识号，同时在当前用户的 Session 域中保存这个标识号；

（2）当用户提交 FORM 表单时，负责接收这一请求的服务器程序比较 FORM 表单隐藏字段中的标识号与存储在当前用户的 Session 域中的标识号是否相同，如果相同则处理表单数据，处理完后清除当前用户的 Session 域中存储的标识号。经过这样的处理后，当用户重复提交原来得到的 FORM 表单时，当前用户的 Session 域中已不存在相应的表单标识号。在下列情况下，服务器程序将忽略提交的表单请求：

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

- 当前用户的 Session 中不存在表单标识号；
- 用户提交的表单数据中没有标识号字段；
- 存储在当前用户的 Session 域中的表单标识号与表单数据中的标识号不同。

（3）浏览器只有重新向 Web 服务器请求包含 FORM 表单的页面时，服务器程序才会产生另外一个随机标识号，并将这个标识号保存在 Session 域中和作为新返回的 FORM 表单中的隐藏字段值。

可见，经过这样的设计后，只有客户端重新访问包含有 FORM 表单的页面后，服务器端的 FORM 表单处理程序才可以接受 FORM 表单提交的数据，否则，服务器端的 FORM 表单处理程序将不接受用户使用同一个表单重复提交的数据。

下面编写一个利用 Session 防止表单重复提交的例子程序，整个程序包含三个组件：TokenProcessor.java，FormGenerateServlet.java 和 FormDealServlet.java。TokenProcessor.java 是用于管理表单标识号的工具类，它主要用于产生、比较和清除存储在当前用户 Session 中的表单标识号。为了保证表单标识号的惟一性，每次将当前 SessionID 和系统时间的组合值按 MD5 算法计算的结果作为表单标识号，并且将 TokenProcessor 类设计为单件类。FormGenerateServlet.java 是用于产生 FORM 表单的 Servlet 程序，FormDealServlet 是负责处理 FORM 表单请求的 Servlet 程序。

MD5 算法及其应用

MD5 算法可用于对一段信息进行不可逆的变换运算，产生一个 128 比特位的数字摘要。如果对输入信息进行了任何形式的改变，对改变后的信息再次进行 MD5 运算所产生的数字摘要，与改变前的内容所运算出的 MD5 数字摘要都是不同的。在实际项目中，用户注册的密码都不应该直接存储在数据库中，而是应该加密后再存储到数据库中，以防止数据库管理人员查看到用户的密码。在对用户注册的密码进行加密时，不能采用可逆的加密算法，因为人们很容易对可逆算法加密后的结果进行解密。大多数项目都采用 MD5 算法对用户注册的密码进行加密，即使人们看到了加密后的结果信息，但也无法根据这个结果信息推算出原始的信息，这样就不用担心数据库管理人员获取用户的密码信息了。在用户进行登录验证时，系统对用户输入的密码也执行 MD5 运算，然后将运算的结果与数据库中存储的信息进行比较，如果两者相等，则说明用户输入的密码正确，否则，说明用户输入的密码无效。

JDK 的核心包中提供了一个 java.security.MessageDigest 类，应用程序通过这个类可以调用 MD5 和 SHA 等算法来产生数字摘要。应用程序首先调用 MessageDigest 类的 getInstance 静态方法获得某个算法的 MessageDigest 实例对象，然后调用 MessageDigest 实例对象的 update 方法添加要进行数字摘要运算的数据源，数据源可以通过多次调用 update 方法进行累加，最后再调用 MessageDigest 实例对象的 digest 方法产生出数字摘要。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

利用 Session 防止表单重复提交

01 按上面描述的功能编写如例程 7-8、例程 7-9 和例程 7-10 所示的程序。

例程 7-8 TokenProcessor.java

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

public class TokenProcessor
{
    private long previous;  //上次生成表单标识号的时间值
    private static TokenProcessor instance = new TokenProcessor();
    public static String FORM_TOKEN_KEY = "FORM_TOKEN_KEY";

    private TokenProcessor()
    {
    }

    public static TokenProcessor getInstance()
    {
        return instance;
    }

    /**
     *验证请求消息中的标识号是否有效，如果请求消息中的表单标识号与当前
     *用户的Session域中的表单标识号相同，返回结果为true，否则返回false。
     *
     *@param request 封装当前请求消息的HttpServletRequest对象
     */
    public synchronized boolean isTokenValid(HttpServletRequest request)
{
    /*为避免Session对象不存在时创建Session对象，
    下面的语句不用request.getSession()*/
    HttpSession session = request.getSession(false);
    if (session == null)
    {
        return false;
    }
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        String saved = (String) session.getAttribute(FORM_TOKEN_KEY);
        if (saved == null)
        {
            return false;
        }

        String token = request.getParameter(FORM_TOKEN_KEY);
        if (token == null)
        {
            return false;
        }

        return saved.equals(token);
    }

    /**
     * 清除存储在当前用户Session中的表单标识号。
     *
     *@param request 封装当前请求消息的HttpServletRequest对象
     */
    public synchronized void resetToken(HttpServletRequest request)
    {
        HttpSession session = request.getSession(false);
        if (session == null)
        {
            return;
        }

        session.removeAttribute(FORM_TOKEN_KEY);
    }

    /**
     * 产生表单标识号并将之保存在当前用户Session中。
     *
     *@param request 封装当前请求消息的HttpServletRequest对象
     */
    public synchronized void saveToken(HttpServletRequest request)
    {
        HttpSession session = request.getSession();
        try
        {
            byte id[] = session.getId().getBytes();
            long current = System.currentTimeMillis();
            if (current == previous)
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        {
            current++;
        }
        previous = current;
        byte now[] = String.valueOf(current).getBytes();
        MessageDigest md = MessageDigest.getInstance("MD5");
        md.update(id);
        md.update(now);
        String token = toHex(md.digest());
        session.setAttribute(FORM_TOKEN_KEY, token);
    }
    catch (NoSuchAlgorithmException e)
    {
        ;
    }
}

/**
 * 将一个字节数组转换成十六进制的字符串形式返回。
 * @param buffer 要被转换的字节数组
 */
private String toHex(byte buffer[])
{
    StringBuffer sb = new StringBuffer(buffer.length * 2);
    for (int i = 0; i < buffer.length; i++)
    {
            sb.append(Character.forDigit((buffer[i] & 0xf0) >> 4, 16));
        sb.append(Character.forDigit(buffer[i] & 0x0f, 16));
    }
    return sb.toString();
}

}
```

例程 7-9 FormGenerateServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class FormGenerateServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
{
    response.setContentType("text/html;charset=GB2312");
    PrintWriter out = response.getWriter();
    TokenProcessor.getInstance().saveToken(request);
    String token = (String)request.getSession().getAttribute(
                TokenProcessor.FORM_TOKEN_KEY);
    out.println("<form action='FormDealServlet'" + "method='POST'>" +
        "<input type='hidden' name='" + TokenProcessor.FORM_TOKEN_KEY +
            "' value='" + token + "'>" +
        "字段1: <input type='text' name='p1'><br>" +
        "<input type='submit' value='提交'> " +
        "</form>");
}
}
```

例程 7-10　FormDealServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class FormDealServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        response.setContentType("text/html;charset=GB2312");
        PrintWriter out = response.getWriter();
        TokenProcessor tokenProcessor = TokenProcessor.getInstance();

        if(!tokenProcessor.isTokenValid(request))
        {
            out.println("这是重复或非法提交!");
            return;
        }

        String p1 = request.getParameter("p1");
        if(p1==null || p1.trim().equals(""))
        {
            out.println("请输入内容!");
        }
        else
        {
            out.println("提交内容已被处理!");
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
            tokenProcessor.resetToken(request);
        }
    }
}
```

编译上面的三个 Java 源文件，确保编译后生成的 class 文件存放在了<tomcat 安装目录>\Webapps\it315\Web-INF\classes 目录中。

02 在<tomcat 安装目录>\Webapps\it315\Web-INF\Web.xml 文件中注册有关的 Servlet，并设置其映射 URL。在 Web.xml 文件中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>FormGenerateServlet</servlet-name>
    <servlet-class>FormGenerateServlet</servlet-class>
</servlet>
<servlet>
    <servlet-name>FormDealServlet</servlet-name>
    <servlet-class>FormDealServlet</servlet-class>
</servlet>
……
<servlet-mapping>
    <servlet-name>FormGenerateServlet</servlet-name>
    <url-pattern>/servlet/FormGenerateServlet</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>FormDealServlet</servlet-name>
    <url-pattern>/servlet/FormDealServlet</url-pattern>
</servlet-mapping>
```

保存 Web.xml 文件后，重新启动 Tomcat。

03 在浏览器地址栏中输入如下地址：

http://localhost:8080/it315/servlet/FormGenerateServlet

单击浏览器的“查看”→“源文件”菜单，查看服务返回给浏览器的网页文档源文件，可以看到其中设置表单标识号的隐藏字段的源代码如下：

```
<input type='hidden' name='FORM_TOKEN_KEY'
    value='4b15c6b2f573831b4b5107d849fcafb8'>
```

接着在返回页面的 FORM 表单的“字段 1”中填写一些内容后单击“提交”按钮，浏览器中返回“提交内容已被处理!”的提示信息。

04 在单击浏览器工具栏上的“刷新”按钮（这将再次提交上次的表单数据），或者是单击浏览器工具栏上的“后退”按钮回到表单填写页面后单击“提交”按钮，浏览器中都返回“这是重复或非法提交!”的提示信息。可见，本程序避免了同一表单的重复提交，它可以防止用户无意之中重复提交信息的行为，也可以防止用户故意在表单提交后单击浏

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

览器工具栏中的“刷新”或“后退”按钮后继续向服务器重复提交信息的行为。

05 单击浏览器工具栏上的“后退”按钮回到表单填写页面，接着再单击浏览器工具栏上的“刷新”按钮向服务器重新请求表单填写页面，然后在 FORM 表单的“字段 1”中填写一些内容后单击“提交”按钮，浏览器中又返回“提交内容已被处理!”的提示信息。可见，本程序不能有效阻止捣乱者手工或使用工具连续发送大量垃圾信息的行为，因为用户只要再次访问表单填写页面，从 Web 服务器重新获得 FORM 表单后，就可以通过这个新获得的 FORM 表单再次向服务器提交信息。

06 单击浏览器工具栏上的“后退”按钮回到表单填写页面，接着再单击浏览器工具栏上的“刷新”按钮向服务器重新请求表单填写页面，在 FORM 表单的“字段 1”中不填写任何内容就单击“提交”按钮，浏览器中返回“请输入内容!”的提示信息。单击浏览器工具栏上的“后退”按钮回到表单填写页面，然后在 FORM 表单的 “字段 1” 中填写一些内容后单击“提交”按钮，浏览器中又返回“提交内容已被处理!”的提示信息。可见，在表单第一次提交信息不完整而没被接受处理时，本程序允许用户后退到表单填写页面进行补充填写后再提交表单，省得用户重新输入原来填好的字段内容。

07 单击浏览器工具栏上的“后退”按钮回到表单填写页面，单击浏览器工具栏上的“刷新”按钮向服务器重新请求表单填写页面。接着单击浏览器的“文件”→“新建”→“窗口”菜单，在新窗口中也显示出了原窗口中的页面，然后在原窗口的 FORM 表单的“字段 1”中填写一些内容后单击“提交”按钮，浏览器中返回“这是重复或非法提交!”的提示信息。原窗口中的表单填写页面明明是刚才重新访问 Web 服务器得到的，怎么会出现重复提交的问题呢？这是因为 IE 浏览器的新创建窗口中显示出的页面也是重新访问 Web 服务器得到的，查看新窗口与原窗口中的源文件，可以发现两者中的表单标识号是不一样的。

新窗口与原窗口共享 Cookie 信息，Web 服务器将认为它们的请求属于同一会话，服务器程序为新窗口分配表单标识号后存储进当前用户的 Session 域中，这将覆盖 Session 域中存储的原窗口的表单标识号。当原窗口提交表单时，服务器程序发现它所提供的表单标识号与 Session 域中存储的表单标识号不一致，因此认为它是一个非法的表单提交请求。使用 JavaScript 的 window.open 方法打开新窗口时也会出现这种情况，所以，在开发网站程序时，要注意避免主窗口与 window.open 方法打开的窗口中同时出现要提交给 Web 服务器的 FORM 表单的情况。因为各个用户的表单标识号保存在他们各自的 Session 中，所以不同的用户之间不用担心彼此的表单标识号出现相互覆盖的问题，这正是本程序使用 Session 域来保存表单标识号的原因。

同一个用户打开同一个浏览器进程的多个窗口来并发访问同一个 Web 站点的多个 FORM 表单页面的情况并不常见，即便出现，Web 站点也完全可以将这样的操作视为无效，明确告诉用户进行这样的操作将得不到正常的服务功能。如果因为某种特殊情况，Web 站点需要支持在同一会话中并行打开多个 FORM 表单页面的功能，那么可以将 FORM 表单的标识号作为表单隐藏字段的名称，如下所示：

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
<input type='hidden' name='4b15c6b2f573831b4b5107d849fcafb8' value=''>
```

然后将所有的表单标识号存储进一个 Vector 集合对象中，并将 Vector 集合对象存储进 Session 域中。当表单提交时，先从 Session 域中取出 Vector 集合对象，再从 Vector 集合对象中逐一取出每个表单标识号作为参数调用 HttpServletRequest.getParameter 方法，如果其中有一次调用的返回值不为 null，则接受并处理该表单数据，处理完后将该表单标识号从 Vector 集合对象中删除。

注意 虽然上面的方式实现了在同一会话中可并行打开多个 FORM 表单页面的功能，但是如果用户请求了表单填写页面后没有提交该表单，该表单标识号将一直保存在当前用户的 Session 域中。恶意用户可能利用这一问题，不断重复访问表单填写页面，以致 Session 域中保存的表单标识号不断增多，最终严重消耗服务器的内存资源致使服务器无法对外正常提供服务。

7.5.3 利用 Session 实现一次性验证码

一些人为了窃取他人在某个网站上的账号，通常采用的办法就是在网站登录页面中不断尝试各种密码进行登录，如果纯粹采用手工方式，由于劳动强度和效率的问题，他们最终都会由于收获渺茫而放弃。但是，如果这些人借用自动化的密码猜测工具来无限制地尝试各种密码，他们成功窃取别人密码的可能性就非常大了。一次性验证码的主要目的就是为了限制人们利用工具软件来暴力猜测密码，其原理和编程实现与 7.5.3 节的利用 Session 防止表单重复提交的例子程序基本一样，只是将表单标识号变成了验证码的形式，并且要求用户将提示的验证码手工填写进一个表单字段中，而不是通过表单的隐藏字段自动回传给服务器。为了不给正常用户添加太多的输入麻烦，验证码不能太长，通常为 4 个随机的字符。为了增加其他工具程序自动识别出验证码的难度，识别码通常以图片的形式展示给用户，并在图片中随机产生一些杂乱的干扰点，如图 7.25 所示。

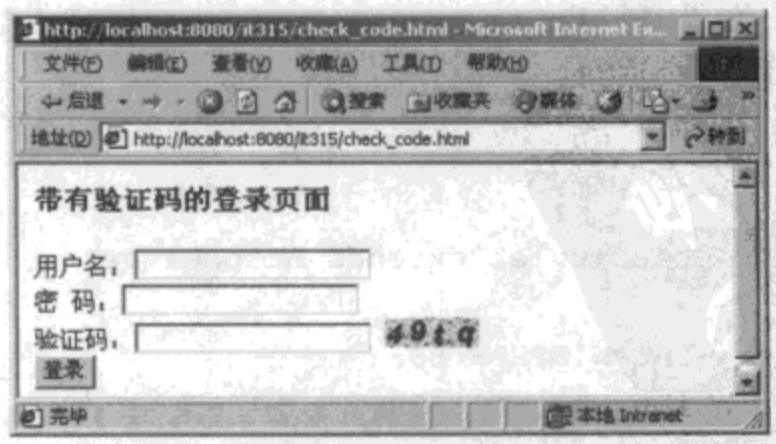


图 7.25

服务器程序接收到表单数据后，首先判断用户是否填写了正确的验证码，只有填写的验证码与服务器端保存的验证码匹配时，服务器程序才开始正常的表单处理流程。验证码

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

使用一次即失效，用户只能重新向服务器发出访问表单填写页面的请求来获得新的验证码，并填写新的验证码后才能再次提交有效的表单请求，这样将大大增加了用户重复操作的难度。密码猜测工具要逐一尝试每个密码的前提条件是先输入正确的验证码，而验证码是一次性有效的，这样基本上就阻断了密码猜测工具的自动处理过程。

下面编写一个利用 Session 实现一次性验证码的例子程序，整个程序包含三个组件：check_code.html，CheckCodeServlet.java 和 LogonFormServlet.java。check_code.html 是引用验证码图片的 FORM 表单页面，CheckCodeServlet.java 是用于产生带有随机验证码图片的 Servlet 程序，LogonFormServlet.java 则是负责处理 FORM 表单请求的 Servlet 程序。

利用 Session 实现一次性验证码

01 按上面描述的功能编写如例程 7-11、例程 7-12 和例程 7-13 所示的程序。

例程 7-11 check_code.html

```
<h3>带有验证码的登录页面</h3>
<form action="servlet/LogonFormServlet" method="post">
用户名：<input type="text" name="name"><br>
密 码：<input type="password" name="pass"><br>
验证码：<input type="text" name="check_code">
<img src="servlet/CheckCodeServlet"><br>
<input type="submit" value="登录">
</form>
```

例程 7-12 CheckCodeServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.awt.*;
import java.awt.image.*;
import javax.imageio.ImageIO;

public class CheckCodeServlet extends HttpServlet
{
    private static int WIDTH = 60;
    private static int HEIGHT = 20;
    public void doGet(HttpServletRequest request,HttpServletResponse response)
            throws ServletException,IOException
    {
        HttpSession session = request.getSession();
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        response.setContentType("image/jpeg");
        ServletOutputStream sos = response.getOutputStream();

        //设置浏览器不要缓存此图片
        response.setHeader("Pragma","No-cache");
        response.setHeader("Cache-Control","no-cache");
        response.setDateHeader("Expires", 0);

        //创建内存图像并获得其图形上下文
        BufferedImage image =
            new BufferedImage(WIDTH, HEIGHT, BufferedImage.TYPE_INT_RGB);
        Graphics g = image.getGraphics();

        //产生随机的验证码
        char [] rands = generateCheckCode();

        //产生图像
        drawBackground(g);
        drawRands(g,rands);

        //结束图像的绘制过程，完成图像
        g.dispose();

        //将图像输出到客户端
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        ImageIO.write(image, "JPEG", bos);
        byte [] buf = bos.toByteArray();
        response.setContentLength(buf.length);
        //下面的语句也可写成: bos.writeTo(sos);
        sos.write(buf);
        bos.close();
        sos.close();

        //将当前验证码存入到Session中
        session.setAttribute("check_code",new String(rands));
        //直接使用下面的代码将有问题，Session对象必须在提交响应前获得
        //request.getSession().setAttribute("check_code",new String(rands));
    }

    private char [] generateCheckCode()
    {
        //定义验证码的字符表
        String chars = "0123456789abcdefghijklmnopqrstuvwxyz";
        char [] rands = new char[4];
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        for(int i=0; i<4; i++)
        {
            int rand = (int)(Math.random() * 36);
            rands[i] = chars.charAt(rand);
        }
        return rands;
    }

    private void drawRands(Graphics g , char [] rands)
    {
        g.setColor(Color.BLACK);
        g.setFont(new Font(null,Font.ITALIC|Font.BOLD,18));
        //在不同的高度上输出验证码的每个字符
        g.drawString("" + rands[0],1,17);
    g.drawString("" + rands[1],16,15);
    g.drawString("" + rands[2],31,18);
    g.drawString("" + rands[3],46,16);
    System.out.println(rands);
    }

    private void drawBackground(Graphics g)
    {
        //画背景
        g.setColor(new Color(0xDCDCDC));
        g.fillRect(0, 0, WIDTH, HEIGHT);
        //随机产生120个干扰点
        for(int i=0; i<120; i++)
        {
            int x = (int)(Math.random() * WIDTH);
            int y = (int)(Math.random() * HEIGHT);
            int red = (int)(Math.random() * 255);
            int green = (int)(Math.random() * 255);
            int blue = (int)(Math.random() * 255);
            g.setColor(new Color(red,green,blue));
            g.drawOval(x,y,1,0);
        }
    }
}
```

例程 7-13　LogonFormServlet.java

```
import java.io.*;
import javax.servlet.*;
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


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

public class LogonFormServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        response.setContentType("text/html;charset=GB2312");
        PrintWriter out = response.getWriter();

        HttpSession session = request.getSession(false);
        if(session == null)
        {
            out.println("验证码处理问题!");
            return;
        }

        String savedCode = (String)session.getAttribute("check_code");
        if(savedCode == null)
        {
            out.println("验证码处理问题!");
            return;
        }

        String checkCode = request.getParameter("check_code");
        if(!savedCode.equals(checkCode))
        {
            /*验证码未通过，不从Session中清除原来的验证码，
            以便用户可以后退回登录页面继续使用原来的验证码进行登录*/
            out.println("验证码无效!");
            return;
        }
        /*验证码检查通过后，从Session中清除原来的验证码，
        以防用户后退回登录页面继续使用原来的验证码进行登录*/
        session.removeAttribute("check_code");
        out.println("验证码通过，服务器正在校验用户名和密码!");
    }
}
```

编译上面的两个 Java 源文件，确保编译后生成的 class 文件存放在了<tomcat 安装目录>\Webapps\it315\Web-INF\classes 目录中。将 check_code.html 文件保存在<tomcat 安装目录>\Webapps\it315 目录中。

02 在<tomcat 安装目录>\Webapps\it315\Web-INF\Web.xml 文件中注册有关的

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

Servlet，并设置其映射 URL。在 Web.xml 文件中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>CheckCodeServlet</servlet-name>
    <servlet-class>CheckCodeServlet</servlet-class>
</servlet>
<servlet>
    <servlet-name>LogonFormServlet</servlet-name>
    <servlet-class>LogonFormServlet</servlet-class>
</servlet>
……
<servlet-mapping>
    <servlet-name>CheckCodeServlet</servlet-name>
    <url-pattern>/servlet/CheckCodeServlet</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>LogonFormServlet</servlet-name>
    <url-pattern>/servlet/LogonFormServlet</url-pattern>
</servlet-mapping>
```

保存 Web.xml 文件后，重新启动 Tomcat。

03 在浏览器地址栏中输入如下地址：

http://localhost:8080/it315/check_code.html

浏览器中显示出如图 7.25 所示的效果，然后就可以对验证码的功能进行测试了。

多学两招

用时间间隔限制用户连续发帖

验证码除了可以用在用户登录页面中以抵御密码猜测工具外，也可以用在一些发表评论的页面中来防止用户故意使用工具连续向服务器重复提交信息的行为。但是验证码方式也不能有效阻止捣乱者采用手工方式连续发送垃圾信息的行为，因为用户只要重新向服务器发出访问表单填写页面的请求而获得新的验证码，就可以再次向服务器提交信息。一些网站通常采用在 Session 中记录用户发帖的时间，然后通过一个时间间隔来限制用户连续发帖的数量。

7.5.4 跟踪用户上次访问站点的时间及 Cookie 中文问题

一些网站能够提示客户端计算机上次访问网站的时间，其实现原理就是将每一个会话作为一次访问过程，将每次会话的开始时间作为每次访问网站的时间，然后将这个时间以 Cookie 的形式存储到客户端的计算机中，客户端进行下次访问时通过该 Cookie 回传上次访问站点的时间值。例程 7-14 是一个跟踪用户上次访问站点的时间的例子程序，其中有两

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

点值得注意的地方：

- 为了让 Cookie 信息在客户端浏览器或计算机关闭后仍然保持存在，Cookie 的保存时间被设置为了一年。
- Cookie 值中包含有中文字符，如果直接将它们传递给浏览器，其中的中文字符将丢失，因此，例程 7-14 采用 BASE64 编码方式将 Cookie 值编码成了可打印的 ASCII 字符后再进行传递。

例程 7-14 AccessTimeServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Date;
import java.text.SimpleDateFormat;
import sun.misc.BASE64Encoder;
import sun.misc.BASE64Decoder;

public class AccessTimeServlet extends HttpServlet
{
public void service(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException
{
    response.setContentType("text/html;charset=gb2312");
    PrintWriter out = response.getWriter();

    //获取记录用户上次访问站点时间的Cookie
    Cookie [] cks = request.getCookies();
    String lastAccessTime = getCookie(cks,"access_time");
    if(lastAccessTime == null)
    {
        out.println("没有您上次的访问记录");
    }
    else
    {
        BASE64Decoder decoder = new BASE64Decoder();
        String decodedAccessTime = new String(
            decoder.decodeBuffer(lastAccessTime),"GB2312");
        out.println("您上次的访问时间为" + decodedAccessTime);
    }
    //将用户开始本次访问的时间存储到客户端Cookie中
    HttpSession session = request.getSession();
    if(session.isNew())
    {
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        long createTime = session.getCreationTime();
        Date dateCreateTime = new Date(createTime);
        SimpleDateFormat sdf =
            new SimpleDateFormat("yyyy年MM月dd HH点mm分ss秒");
        String formatedCreateTime = sdf.format(dateCreateTime);
        BASE64Encoder encoder = new sun.misc.BASE64Encoder();
        String base64CreateTime = encoder.encode(
            formatedCreateTime.getBytes("GB2312"));
        Cookie ckAccessTime = new Cookie("access_time",base64CreateTime);
        ckAccessTime.setMaxAge(365*24*3600);
        out.println(formatedCreateTime);
        response.addCookie(ckAccessTime);
    }
}

private String getCookie(Cookie [] cks,String cookieName)
{
    String cookieValue = null;
    for(int i=0; cks!=null && i<cks.length; i++)
    {
        if(cookieName.equals(cks[i].getName()))
        {
            cookieValue = cks[i].getValue();
            break;
        }
    }
    return cookieValue;
}
}
```

上面的程序首先从请求消息中检索记录用户上次访问站点时间的 Cookie，并将 Cookie 值进行 BASE64 解码后显示出来。如果本次请求是一个会话的开始，则将会话的开始时间格式化为“xxxx 年 xx 月 xx 日 xx 点 xx 分 xx 秒”的形式，接着进行 BASE64 编码后再作为 Cookie 值发送给客户端。

为了向读者讲解如何用 Cookie 传递中文信息，所以上面的程序编写得略微有些复杂，如果仅仅是要将用户上次访问站点时间显示为“xxxx 年 xx 月 xx 日 xx 点 xx 分 xx 秒”这种格式，则可以直接将 session.getCreationTime()方法返回的值作为 Cookie 值传递到客户端，在以后检索到这个 Cookie 值后再将它变成“xxxx 年 xx 月 xx 日 xx 点 xx 分 xx 秒”的形式。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

7.6 Session 的持久化管理

7.6.1 Session 的持久化

只要客户端访问了某个能开启会话功能的资源，Web 服务器就会创建一个与该客户端对应的 HttpSession 对象，每个 HttpSession 对象都要占用一定的内存空间。如果在某一时间段内，访问站点的用户很多，Web 服务器内存中将会因此积累大量的 HttpSession 对象，消耗大量的服务器内存资源。即使用户已经离开或者关闭了浏览器，Web 服务器也并不知道，还要保留与之对应的 HttpSession 对象。尽管最后可以通过超时限制来释放 HttpSession 对象，但是在超时之前，这些 HttpSession 对象仍然占用内存资源。

为了提高服务器内存资源的利用率，Web 服务器通常将那些暂时不活动但未超时的 HttpSession 对象转移到文件系统或数据库中保存，一旦服务器需要使用它们时，再将它们从文件系统或数据库中装载进内存，这种技术称为 Session 的持久化。将 HttpSession 对象保存到文件系统或数据库中时，需要采用序列化的方式将 HttpSession 对象中的每个属性对象保存到文件系统或数据库中；将 HttpSession 对象从文件系统或数据库中装载进内存时，需要采用反序列化的方式恢复 HttpSession 对象中的每个属性对象。所以，在使用 Session 时，有一个非常重要的注意事项：**存储在 HttpSession 对象中的每个属性对象必须是可序列化的，即必须是实现了 Serializable 接口的对象。**

Session 持久化具有如下一些作用：

（1）既可以提高服务器内存资源的利用率，又可以保证那些暂停活动的客户端在会话超时之前继续原来的会话。

（2）在多台 Web 服务器协同对外提供服务的集群系统中，借助 Session 的持久化技术，某台服务器可以将其中发生了改变的 Session 对象复制给其他服务器，否则，集群系统就必须保证来自同一用户的请求总是被定向到处理该用户的第一次访问请求的服务器，并且因为其他服务器中不存在某个服务器的 Session 信息，在某个服务器停止工作后无法启动其他服务器来接替它与客户端的会话。

（3）在一个 Web 应用程序关闭后接着重新启动时，服务器也会持久化该应用程序内存中的所有 HttpSession 对象，即在 Web 应用程序关闭时将所有 HttpSession 对象保存到文件系统或数据库中。如果客户端在 Web 应用程序重新启动后要求继续原来的会话，服务器再将客户端所对应的 HttpSession 对象从文件系统或数据库中装载进内存。通过这种方式，可以保障重启服务器和应用程序时，一些客户端的会话活动仍可继续。

7.6.2 Tomcat 中的 Session 持久化管理

Tomcat 使用 Session Manager 类来管理 Session 的持久化，参看 Tomcat 的配置帮助文档，可以看到 Tomcat 提供了两个 Session Manager 类：

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

- org.apache.catalina.session.StandardManager
- org.apache.catalina.session.PersistentManager

StandardManager 在 Web 应用程序关闭时，对内存中的所有 HttpSession 对象进行持久化，把它们保存到文件系统中，默认的存储文件为：

<tomcat 安装目录>\work\Catalina\<主机名>\<应用程序名>\SESSIONS.ser

PersistentManager 提供了比 StandardManager 更为灵活的 Session 管理功能，只要某个设备提供了实现 org.apache.catalina.Store 接口的驱动类，PersistentManager 就可以将 HttpSession 对象保存到该设备。Tomcat 提供了将 HttpSession 对象保存到文件系统和数据库的 Store 接口实现类。

Tomcat 默认使用 StandardManager，如果要使用其他的 Session Manager，则要在 server.xml 配置文件中的 Context 元素下增加一个名为 Manager 的子元素，如下所示：

```
<Context path= "/it315" docBase="it315">
<Manager className="org.apache.catalina.session.PersistentManager" …>
    <Store className="…"  …>
</Manager>
</Context>
```

详细的配置信息请参看 Tomcat 的配置帮助文档。

下面编写一个测试 Tomcat 的 Session 持久化的例子程序，整个程序包含三个组件：Student1.java，Student2.java 和 PersistentServlet.java。Student1.java 和 Student2.java 是两个普通的 Java 类，Student2 实现了 Serializable 接口，Student1 没有实现 Serializable 接口，除此之外，Student1 和 Student2 中的其他代码完全一样。PersistentServlet.java 是一个 Servlet 程序，它在第一次被访问时将 Student1 和 Student2 的实例对象保存到 Session 域中，在以后被访问时又从 Session 域中读取先前保存的 Student1 和 Student2 的实例对象。

Tomcat 的 Session 持久化

01 按上面描述的功能编写如下程序。

例程 7-15　Student1.java

```
public class Student1
{
private int value = 0;
public void setValue(int value)
{
    this.value = value;
}
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
public int getValue()
{
    return value;
}
}
```

例程 7-16 Student2.java

```
import java.io.Serializable;

public class Student2 implements Serializable
{
private int value = 0;
    public void setValue(int value)
    {
        this.value = value;
    }

    public int getValue()
    {
        return value;
    }
}
```

例程 7-17 PersistentServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class PersistentServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out = response.getWriter();
        HttpSession session = request.getSession();

        Student1 student1 = (Student1)session.getAttribute("student1");
        if(student1 == null)
        {
            student1 = new Student1();
            student1.setValue(35);
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
            session.setAttribute("student1",student1);
            out.println("将student1保存到了Session中! <br>");
        }
        else
        {
            out.println("读取student1:" + student1.getValue() + "<br>");
        }

        Student2 student2 = (Student2)session.getAttribute("student2");
        if(student2 == null)
        {
            student2 = new Student2();
            student2.setValue(33);
            session.setAttribute("student2",student2);
            out.println("将student2保存到了Session中! <br>");
        }
        else
        {
            out.println("读取student2:" + student2.getValue() + "<br>");
        }
    }
}
```

编译上面的三个 Java 源文件，确保编译后生成的 class 文件存放在了<tomcat 安装目录>\Webapps\it315\Web-INF\classes 目录中。

02 在<tomcat 安装目录>\Webapps\it315\Web-INF\Web.xml 文件中注册有关的 Servlet，并设置其映射 URL。在 Web.xml 文件中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>PersistentServlet</servlet-name>
    <servlet-class>PersistentServlet</servlet-class>
</servlet>
……
<servlet-mapping>
    <servlet-name>PersistentServlet</servlet-name>
    <url-pattern>/servlet/PersistentServlet</url-pattern>
</servlet-mapping>
```

保存 Web.xml 文件后，重新启动 Tomcat。

03 在浏览器地址栏中输入如下地址：

http://localhost:8080/it315/servlet/PersistentServlet

浏览器窗口中显示的结果如图 7.26 所示，单击工具栏中的“刷新”按钮再次访问 PersistentServlet，浏览器窗口中显示的结果如图 7.27 所示。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

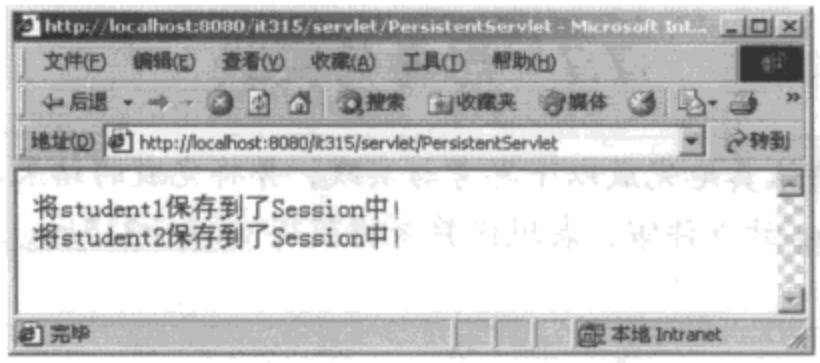


图 7.26

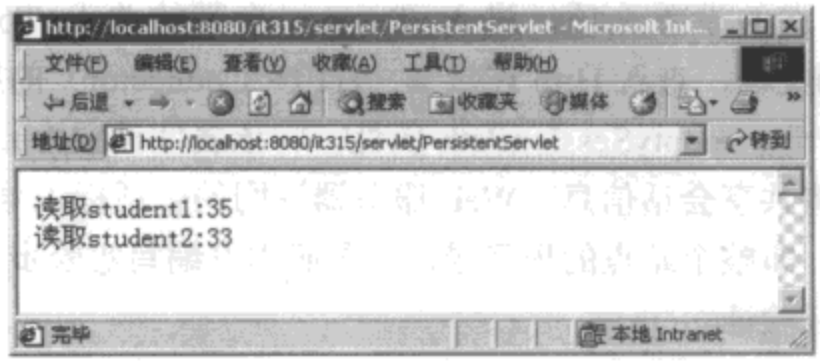


图 7.27

04 关闭 Tomcat 服务程序，然后用 Windows 资源管理器打开“<tomcat 安装目录>\work\Catalina\localhost\it315\”目录，可以看到其中多了一个 SESSIONS.ser 文件，这就是 Tomcat 服务器关闭时，持久化了内存中的 HttpSession 对象所生成的文件。重新启动 Tomcat 服务程序，刚才创建的 SESSIONS.ser 文件又消失了，这说明 Tomcat 在启动时就将持久化了的 HttpSession 对象装入了内存，而不是在需要时才将它们装入内存。

05 单击图 7.27 中的浏览器工具栏中的“刷新”按钮，浏览器中显示的结果如图 7.28 所示。

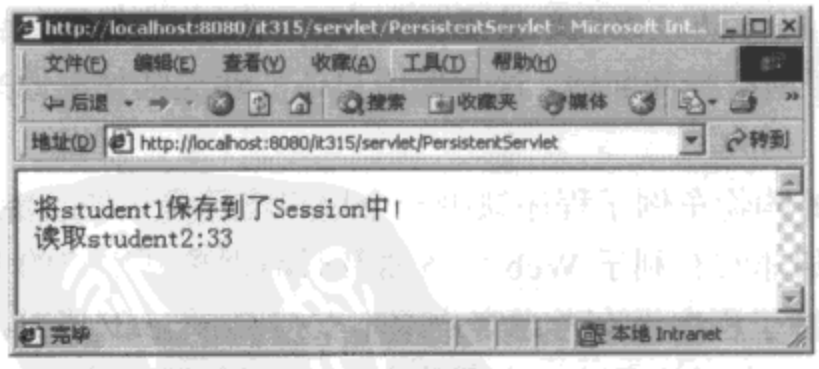


图 7.28

从图 7.28 中可以看到，Tomcat 持久化 HttpSession 对象时，只能将其中的可序列化的属性对象（即实现了 Serializable 接口的类对象）保存到文件系统中，那些不可序列化的属性对象信息将丢失，所以，要存储到 Session 域中的属性对象必须实现 Serializable 接口。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

7.7 思考与实践

友情提示：请认真地完成以下思考与实践，并将完成的结果以及操作步骤提交给 www.it315.org 站点评审，表现优异者将获得 www.it315.org 站点的奖励和专家的特别指导。

1．在默认情况下，同一个浏览器访问 Tomcat 中的两个 Web 应用程序时，两个 Web 应用程序中的 Session 数据是不能共享的。Tomcat 提供了让同一 Web 站点上的所有应用程序之间能共享会话状态的设置选项，进入 Tomcat 自带的关于 Tomcat 配置（Tomcat Configuration）的帮助文档，查看 Http 连接器（HTTP Connector）的属性设置说明，可以看到 Http 连接器中有一个 emptySessionPath 属性，将该属性设置为 true 时，同一 Web 站点上的所有应用程序将共享会话信息，Web 服务器返回的会话标识号的 Cookie 项的 path 属性将被设置为“/”，即整个站点的根目录。请参照本书编写实验步骤的方法设计一个详细的实验步骤对此进行验证。

首先在应用程序 appA 中设置某个 Session 域属性，然后在应用程序 appB 的 Session 域中访问那个属性，如果不能无法访问到该属性，则说明两个 Web 应用程序中的 Session 数据是不能共享的；接着修改 Tomcat 的 Http 连接器的配置，让同一 Web 站点上的所有应用程序之间能共享会话状态，重复上面的操作和查看访问结果。

2．假设浏览器显示出了服务器返回的一个页面后，客户机发生了网络瞬间断线，且重新接通网络后客户端的 IP 地址发生了改变，如果用户继续接着在浏览器原来得到的页面中执行操作，其操作结果有可能受到网络断线的影响吗？例如，服务器端程序按 7.5.2 节的方式使用 Session 技术防止该表单被重复提交，当浏览器得到包含 FORM 表单的页面内容后，客户机发生了网络瞬间断线，这时候用户还能够继续使用原来获得的 FORM 表单提交数据吗？

3．本章 7.5.1 节的购物车例子程序缺少一个用户注销登录的功能，注销登录的程序代码虽然非常简单，但却非常有利于 Web 服务器及时清理离开用户的 HttpSession 对象。另外，7.5.1 节的购物车例子程序没有考虑客户端不支持 Cookie 的情况，即没有增加 URL 重写的方式来完成 Session 会话的跟踪。为了节省 Web 服务器端的内存资源，不要创建一些无价值的 HttpSession 对象，Web 服务器与浏览器的会话过程最好是从用户成功登录后才正式建立，即 Web 服务器只在用户成功登录后才真正创建 HttpSession 对象，而对于其他任何情况都不应创建 HttpSession 对象，7.5.1 节的购物车例子程序没有按照这种思想设计。请在 7.5.1 节的例子程序的基础上增加和完善上述功能。

4．当一个 Servlet 程序使用 RequestDispatcher.forward 方法调用另外一个 Servlet 程序时，调用者程序可以使用 application、session、request 三个域范围中的属性来向被调用者

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

程序传递对象数据，你觉得采用其中的哪个才是最佳方案？请说明不采用其他两个域范围的理由。

5. 请参照 7.5.2 节的例子程序设计如下一个应用程序并写出详细的实验步骤：Web 站点需要利用 Session 防止表单重复提交，但也要支持在同一会话中可并行打开多个 FORM 表单页面的功能，例如，主窗口访问一个 FORM 表单页面后，接着又调用 javascript 的 window.open 方法打开了另外一个 FORM 表单页面，这两个 FORM 表单都可以向 Web 服务器提交一次数据。

6. 一些网站通常采用在 Session 中记录用户发帖的时间，然后通过一个时间间隔来限制用户连续发帖的数量，请编写一个限定用户连续两次提交表单的最小时间间隔的程序来模拟这种应用。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

第 8 章　JSP

8.1　JSP 入门

在很多动态网页中，绝大部分内容都是固定不变的，只有局部内容需要动态产生和改变。譬如，要在一篇新闻报道的网页中显示出该页面被浏览的次数，只有显示的浏览次数需要用程序代码动态产生，而该页面的其他部分都是固定不变的。由于 Servlet 程序返回给客户端的网页文档的全部内容都是通过 Java 程序代码创建的，如果使用 Servlet 程序来输出这种只有局部内容需要动态改变的网页，其中所有的静态内容也需要程序员用 Java 程序代码产生。如果要用 Servlet 程序来输出包含大量静态 HTML 语句和内容的网页文档，整个 Servlet 程序的代码将很臃肿，造成编写和维护也都将非常困难。对大量静态内容的美工设计和相关 HTML 语句的编写，并不是程序员所要做的工作，程序员对此也不一定在行。另外，网页美工设计和制作人员不懂 Java 编程，更是无法来完成这样的工作。

Servlet 将程序的逻辑控制代码与输出网页文档内容的代码混合在一起编写，不利于控制网页文档内容的显示外观和整体布局，给一个网站的实际开发过程带来了很大困难。为了弥补 Servlet 的这些缺陷，Sun 公司又在 Servlet 的基础上推出了 JSP（Java Server Pages）技术作为解决方案。JSP 是简化 Servlet 编写的一种技术，它将 Java 代码和 HTML 语句混合在同一个文件中编写，只对网页中的要动态产生的内容采用 Java 代码来编写，而对固定不变的静态内容采用普通静态 HTML 页面的方式编写。

8.1.1　建立对 JSP 的直观认识

JSP 页面是由 HTML 语句和嵌套在其中的 Java 代码组成的一个普通文本文件。为了与普通 HTML 页面相区别，JSP 页面的文件扩展名必须为.jsp。在 JSP 页面中编写的 Java 代码需要嵌套在<%和%>中，嵌套在<%和%>之间的内容被称之为***脚本片段（Scriptlets）***，没有嵌套在<%和%>之间的内容被称之为 JSP 的***模板元素***。脚本片段被当做 Java 程序代码被解释执行，模板元素则被原封不动地输出给浏览器。脚本片段中的 Java 程序代码可以执行普通 Java 程序一样的功能，例如，进行基本的程序运算，调用其他 Java 类，访问数据库，访问文件系统，等等。JSP 中的 Java 代码可以使用 out.println 语句将其他 Java 程序代码产生的结果字符串输出给客户端，也可以使用 System.out.println 语句将它们打印到命令行窗口。JSP 文件就像普通的 HTML 文件一样，它们可以放置在 Web 应用程序中的除了

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

Web-INF 及其子目录外的其他任何目录中。JSP 页面的访问路径与普通 HTML 页面的访问路径形式也完全一样，即由 Web 应用程序的访问路径加上 JSP 文件在该应用程序中的目录路径组成。

编写最基本的 JSP 程序

01 在 it315 应用程序的根目录（<tomcat 的安装目录>\webapps\it315）中创建一个名称为 jsp 的子目录，然后创建一个内容如例程 8-1 所示的 simple.jsp 文件。

例程 8-1　simple.jsp

```
<%
    String currentTime = new java.util.Date().toString();
    out.println("current time is <i><u>" + currentTime + "</u></i>");
%>
```

02 启动 Tomcat 后，在浏览器地址栏中输入如下地址：

http://localhost:8080/it315/jsp/simple.jsp

浏览器中显示的返回结果如图 8.1 所示。

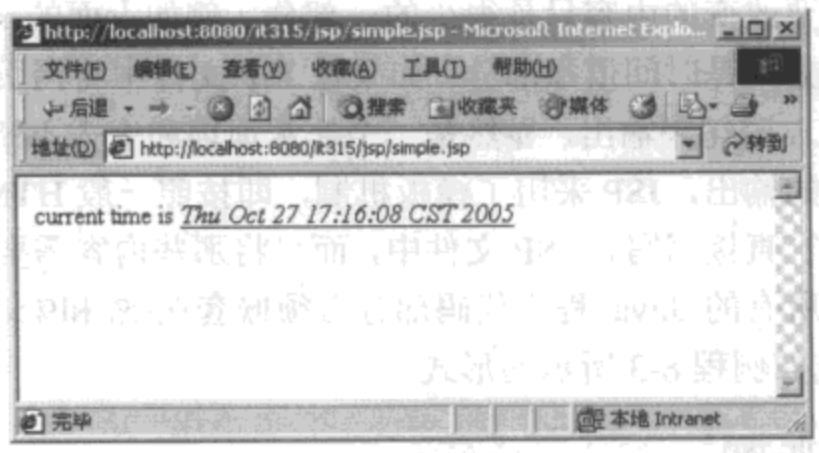


图 8.1

单击 IE 浏览器上的“查看”→“源文件”菜单，可以看到源文件的内容正是 simple.jsp 文件中的 out.println()语句输出的结果。可见，JSP 是由 Web 服务器解释执行的程序代码，主要用于产生返回给浏览器的网页文档内容。

03 普通的 Java 程序可以通过 System.out.println 语句向命令行窗口打印内容，可以用多条 System.out.println 语句在命令行窗口中打印出一段 HTML 代码。JSP 程序则可以通过 out.println 语句向浏览器打印内容，使用多条 out.println 语句，JSP 程序就可以向浏览器输出一个网页文档的所有 HTML 代码。如果要让 simple.jsp 程序输出一段完整的 HTML 页面代码，将例程 8-1 的 simple.jsp 文件修改成如例程 8-2 所示的形式。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

例程 8-2 simple.jsp

```
<%
    out.println("<html>");
    out.println("<head>");
    out.println("<title>a simple jsp</title>");
    out.println("</head>");
    out.println("<body>");
    out.print("current time is <i><u>");
    String currentTime = new java.util.Date().toString();
    out.println(currentTime);
    out.println("</u></i></body>");
    out.println("</html>")
%>
```

在浏览器地址栏中再次输入 simple.jsp 页面的访问地址或者刷新图 8.1 中的浏览器窗口，单击 IE 浏览器上的“查看”→“源文件”菜单，可以看到源文件的内容为 simple.jsp 文件中的所有 out.println()语句输出的结果。

04 在例程 8-2 的 simple.jsp 文件中，只有 out.println(currentTime)语句输出的内容在每次访问时有可能变化，而其他多条 out.println 语句输出的内容在每次访问时都固定不变。在动态网页中，这种现象是非常常见的，即网页文档中的绝大部分内容是固定不变的，而需要改变的内容只是很少的一部分，例如上面的 simple.jsp 程序输出的网页中需要改变的地方只是时间值那部分。如果在 JSP 文件中将网页文档的固定内容部分都采用 out.println 语句来输出，显然是一个非常烦琐和令人不情愿的事情。为了简化那些固定不变的内容的输出，JSP 采用了模板机制，即按照一般 HTML 文件的编写方式，将那些固定不变的内容直接书写在 JSP 文件中，而只将那些内容需要改变的部分使用 Java 程序代码产生，并且所有的 Java 程序代码部分必须嵌套在<%和%>之间。将例程 8-2 的 simple.jsp 文件修改成如例程 8-3 所示的形式。

例程 8-3 simple.jsp

```
<html>
<head>
<title>a simple jsp</title>
</head>
<body>
current time is <i><u>
<%
    String currentTime = new java.util.Date().toString();
    out.println(currentTime);
%>
</u></i></body>
</html>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

当浏览器访问这个 simple.jsp 页面时，Web 容器会把那些没有嵌套在<%和%>之间的模板元素一字不变地输出给浏览器，嵌套在<%和%>之间的脚本片断中的 out.println 语句输出的内容将被顺序插进该<%%>在 JSP 模板元素中的出现位置处，然后与模板元素一同输出到浏览器。在 JSP 模板元素中的任意位置处都可以穿插多个脚本片断（即多个<%%>片断），脚本片断中的 Java 代码必须严格遵循 Java 语法，例如，每条命令执行语句后面必须用分号（;）结束。

05 在 JSP 页面中也可以使用一种称之为 **JSP 表达式**的元素，JSP 表达式简化了将一个 Java 变量或表达式的计算结果输出到客户端的方式，只需将要输出的变量或表达式直接封装在<%= 和 %>之中，就可以向客户端输出这个变量或表达式的运算结果。JSP 表达式中的变量或表达式的计算结果将被转换成一个字符串，然后被插入进整个 JSP 页面输出结果的相应位置处。**注意，在 JSP 表达式中嵌套的变量或表达式后面不能有分号。**

如果采用 JSP 表达式来输出 Java 表达式，例程 8-3 的 simple.jsp 文件可以修改成如例程 8-4 所示的形式。

例程 8-4　simple.jsp

```
<html>
<head>
<title>a simple jsp</title>
</head>
<body>
current time is <i><u><%= new java.util.Date().toString() %></u></i>
</body>
</html>
```

如果采用 JSP 表达式来输出 Java 变量值，例程 8-4 的 simple.jsp 文件可以修改成如例程 8-5 所示的形式。

例程 8-5　simple.jsp

```
<%
    String currentTime = new java.util.Date().toString();
%>
<html>
<head>
<title>a simple jsp</title>
</head>
<body>
current time is <i><u><%= currentTime %></u></i>
</body>
</html>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

使用浏览器访问上面的任意一种形式的 simple.jsp 文件，它们返回的运行效果与修改前的运行效果完全一样。

通过上面的实验可以看到，JSP 页面可以按照编写 HTML 页面的方式来编写，对于其中需要动态变化的内容，则可以通过在 JSP 页面中插入脚本片断和 JSP 表达式来创建。JSP 比较好地解决了网页美工设计、制作人员与编程人员分工的问题，将表现层分派给 HTML 开发人员，将业务逻辑工作分派给 Java 开发人员。

首先，由网页美工设计和制作人员制作出网页文档的 HTML 内容，对于需要动态改变的部分先用静态的内容替代，譬如，网页中的页面浏览次数用一个固定的数值来表示。然后，由程序员直接在网页美工设计和制作人员创建的 HTML 网页文档中，将需要动态改变的内容替换成产生该动态内容的 Java 程序代码，而保持网页静态部分的内容不变。

8.1.2 JSP 的运行原理

Web 容器（Servlet 引擎）接收到以.jsp 为扩展名的 URL 的访问请求时，它将把该访问请求交给 JSP 引擎去处理，JSP 引擎负责解释和执行 JSP 页面。前面的章节曾经讲解过，当 Web 容器接收到客户端的访问请求时，它将根据请求的 URL 在 web.xml 配置文件中查找匹配的<servlet-mapping>元素，然后将请求交给<servlet-mapping>元素中指定的 Servlet 程序去处理，所以，我们很容易想到要去全局的 web.xml 文件中看看是否有与 JSP 文件相关的<servlet-mapping>元素。用 UltraEdit 或记事本程序打开<TOMCAT_HOME>\conf\web.xml 文件，可以找到如下两段内容：

```
<servlet>
    <servlet-name>jsp</servlet-name>
    <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
    ……
</servlet>
……
<servlet-mapping>
    <servlet-name>jsp</servlet-name>
    <url-pattern>*.jsp</url-pattern>
</servlet-mapping>
```

上面的内容说明：以.jsp 为扩展名的 URL 的访问请求由 org.apache.jasper.servlet.JspServlet 负责处理，所以，Tomcat 中的 JSP 引擎就是这个 Servlet 程序。

每个 JSP 页面在第一次被访问时，JSP 引擎将它翻译成一个 Servlet 源程序，接着再把这个 Servlet 源程序编译成 Servlet 的 class 类文件，然后再由 Web 容器（Servlet 引擎）像调用普通 Servlet 程序一样的方式来装载和解释执行这个由 JSP 页面翻译成的 Servlet 程序。如果在翻译 JSP 页面的过程中发现了任何 JSP 语法错误，或者是在编译由 JSP 页面所翻译成的 Servlet 源程序的过程中发现了任何 Java 语法错误，JSP 引擎将中断翻译或编译过程，并将错误的详细信息发送给客户端。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

Tomcat 5.x把为JSP页面创建的Servlet源文件和class类文件放置在“<TOMCAT_HOME>\work\Catalina\<主机名>\<应用程序名>\”目录中，例如，localhost主机的根目录中的JSP页面所生成的Servlet源文件和class类文件的存放目录为<TOMCAT_HOME> \work\Catalina\localhost_\，其中的“_”代表某个Web站点下的根目录，it315应用程序中的JSP页面所生成的Servlet源文件和class类文件的存放目录为<TOMCAT_HOME> \work\Catalina\localhost\it315\。Tomcat将JSP页面翻译成的Servlet的包名为org.apache.jsp.<JSP页面在Web应用程序内的目录名>，所以，Tomcat还要生成与包名对应的目录结构，上面的simple.jsp生成的Servlet源文件与class类文件的目录结构如图8.2所示。

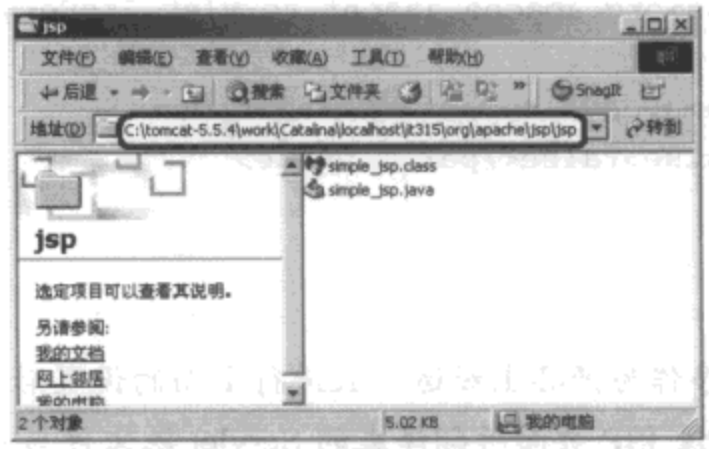


图 8.2

将JSP页面翻译成Servlet程序后再运行的机制有一个明显的好处，那就是JSP引擎仅需要将JSP页面翻译为Servlet程序，而与客户端的交互过程仍然由Servlet引擎处理，重复利用了已有的Servlet引擎技术。JSP规范中并没有规定如何将JSP页面转换为Servlet程序，不同的JSP引擎转换出的Servlet程序可能不一样，但其运行结果必须一样。JSP规范也没有明确要求JSP中的脚本程序代码必须采用Java语言，JSP中的脚本程序代码可以采用Java语言之外的其他脚本语言来编写，但是，JSP页面最终必须转换成Java Servlet程序，采用其他脚本语言编写JSP页面需要有将其转换成Java Servlet的JSP引擎的支持。

一个JSP页面只在第一次被访问时才需要被翻译成Servlet程序，对于该JSP页面的后续访问，Web容器将直接调用其翻译成的Servlet程序。在JSP页面每次被访问时，JSP引擎默认都会检测该JSP文件和编译成的Servlet类的最后更改时间，如果JSP文件自上次编译以后又发生了修改，JSP引擎将重新编译该JSP文件。因为在开发阶段经常要反复地调试和修改程序，所以在开发阶段让JSP引擎自动检测和重新编译修改过的JSP文件是非常有用的。

多学两招

让JSP引擎不检测JSP页面是否发生修改

让JSP引擎对JSP页面的每次访问都检测它是否发生了修改，需要额外的运行开销，降低了系统的运行效率，所以，在Web应用程序作为产品真正上线运行时，不要让JSP

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

引擎去执行 JSP 页面是否发生修改的检测工作。在<TOMCAT_HOME>\conf\web.xml 文件中，在配置 org.apache.jasper.servlet.JspServlet 的<servlet>元素前面有一段针对 JspServlet 的注释说明，其中描述了 JspServlet 可以有一个名称为 development 的初始化参数，其默认值为 true，只要将这个参数设置为 false，即可让 JspServlet 不再以开发模式运行，从而不再检测 JSP 页面是否发生了修改。在<TOMCAT_HOME>\conf\web.xml 文件中，将 JspServlet 的 development 初始化参数设置为 false 的代码片断如下：

```
<servlet>
    <servlet-name>jsp</servlet-name>
    <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
    <init-param>
        <param-name>development</param-name>
        <param-value>false</param-value>
    </init-param>
    ……
</servlet>
```

如果 Web 应用程序作为产品上线运行且进行上面的设置后，偶尔对某个 JSP 页面进行了修改，只要删除该 JSP 页面所翻译成的 Servlet 源文件与 class 文件，然后再访问一下该 JSP 页面，就可以让 JSP 引擎重新编译该 JSP 页面。

由于 JSP 页面在第一次被访问时需要翻译和编译成 Servlet，所以对 JSP 页面第一次访问的响应速度有比较明显的缓慢。如果能够在 Web 应用程序正式发布之前，将其中的所有 JSP 页面预先编译成 Servlet，那么就可以有效地解决这个问题。在 Tomcat 4.x 的 bin 子目录中包含一个 JSP 编译器（jspc.bat），它可以对某个 Web 应用程序的目录下的所有 JSP 页面提前进行编译，也可以只对某个特定的 JSP 文件进行编译。在 Tomcat 5.x 中没有提供 jspc.bat 预编译工具，但是在其自带的帮助文档中介绍了如何使用 ant 工具来预编译 Web 应用程序中的 JSP 页面。进入 Tomcat 帮助文档的首页，如图 8.3 所示，单击其中的“JSPs”超链接即可进入包含有关信息的页面。

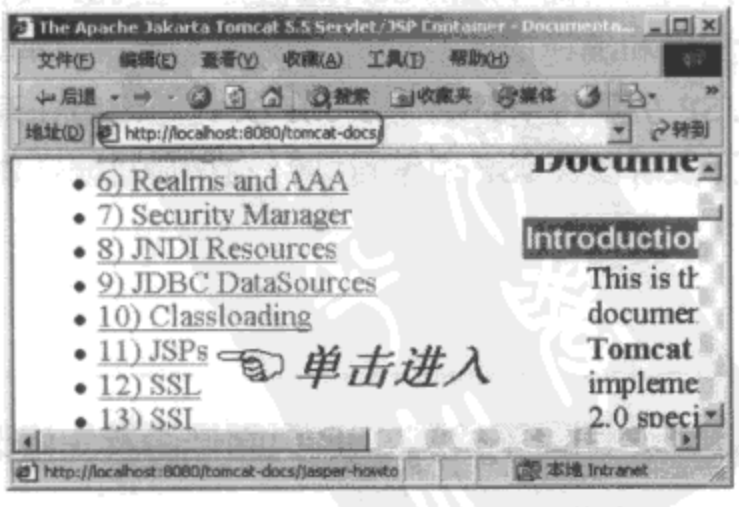


图 8.3

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

8.1.3 分析 JSP 所生成的 Servlet 代码

假设 simple.jsp 文件的内容如下：

```
<%
    String currentTime = new java.util.Date().toString();
%>
<html>
<head>
<title>a simple jsp</title>
</head>
<body>
current time is <i><u><%= currentTime %></u></i>
</body>
</html>
```

访问这个 simple.jsp 文件后，进入如图 8.2 所示的目录中，用 UtralEdit 或记事本程序打开 simple.jsp 文件所生成的 simple_jsp.java 文件，可以看到其中的内容如下：

```
package org.apache.jsp.jsp;

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

public final class simple_jsp extends org.apache.jasper.runtime.HttpJspBase
        implements org.apache.jasper.runtime.JspSourceDependent {

        private static java.util.Vector _jspx_dependants;

        public java.util.List getDependants()   {
                return _jspx_dependants;
    }

    public void _jspService(HttpServletRequest request,
                            HttpServletResponse response)
                    throws java.io.IOException, ServletException {

            JspFactory _jspxFactory = null;
            PageContext pageContext = null;
            HttpSession session = null;
            ServletContext application = null;
            ServletConfig config = null;
            JspWriter out = null;
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
Object page = this;
JspWriter _jspx_out = null;
PageContext _jspx_page_context = null;


try{
        _jspxFactory = JspFactory.getDefaultFactory();
        response.setContentType("text/html");
        pageContext=_jspxFactory.getPageContext(this,request,
        response,null, true,8192, true);
        _jspx_page_context = pageContext;
        application = pageContext.getServletContext();
        config = pageContext.getServletConfig();
        session = pageContext.getSession();
        out = pageContext.getOut();
        _jspx_out = out;

String currentTime = new java.util.Date().toString();

out.write("\r\n");
out.write("<html>\r\n");
out.write("<head>\r\n");
out.write("<title>a simple jsp</title>\r\n");
out.write("</head>\r\n");
out.write("<body>\r\n");
out.write("current time is <i><u>");
out.print( currentTime );
out.write("</u></i>\r\n");
out.write("</body>\r\n");
out.write("</html>\r\n");
out.write("\r\n");
}
catch(Throwable t){
        if (!(t instanceof SkipPageException)){
                out = _jspx_out;
                if (out != null && out.getBufferSize() != 0)
                        out.clearBuffer();
                if (_jspx_page_context != null)
                _jspx_page_context.handlePageException(t);
        }
}
finally {
        if (_jspxFactory != null)
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
                    _jspxFactory.releasePageContext(_jspx_page_context);
            }
        }
    }
```

对于上面的 simple_jsp.java 文件中的程序代码，读者暂时只需关注笔者用黑斜体进行了标记的那些部分，各部分的解释如下：

（1）由 simple.jsp 生成的 Servlet 类名为 simple_jsp，simple_jsp 类继承了 org.apache.jasper.runtime.HttpJspBase 类。HttpJspBase 类是 HttpServlet 的一个子类，因此 simple.jsp 生成的 simple_jsp 类是 HttpServlet 的一个孙子类，显然，simple_jsp 类就是一个 Servlet。笔者下载整个 Tomcat 的源文件进行研究后，找到了 org.apache.jasper.runtime.HttpJspBase 类的定义如下：

```
public abstract class HttpJspBase extends HttpServlet implements HttpJspPage
{
    ……
    public final void init(ServletConfig config)
            throws ServletException
    {
            super.init(config);
            jspInit();
            _jspInit();
    }
    public final void destroy() {
            jspDestroy();
            _jspDestroy();
    }
    ……
    public final void service(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException
    {
            _jspService(request, response);
    }
            ……
    public void jspInit() {
    }

    public void _jspInit() {
    }

    public void jspDestroy() {
    }
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
    protected void _jspDestroy() {
    }
}
```

可见，HttpJspBase 类还实现了 javax.servlet.jsp.HttpJspPage 接口，但只实现了其中的部分方法（jspInit 和 jspDestroy），因此，HttpJspBase 类是抽象的。simple_jsp 类则是在 HttpJspBase 类的基础上，实现了 javax.servlet.jsp.HttpJspPage 接口中的其他方法（_jspService）。

（2）Sun 公司为 JSP 的 Web 容器开发商和 JSP 页面开发人员提供了一套专门应用于开发 JSP 程序的 Java 类，这套 Java 类被称为 JSP API。各种 Web 服务器中附带的 JSP API 的 Jar 包的文件名称可能各不一样，例如，在 Tomcat 4.x 中，JSP API 与 Servlet API 一同放置在<tomcat 的安装目录>\common\lib\servlet.Jar 包内，但在 Tomcat 5.x 中，JSP API 单独放置在了<tomcat 的安装目录>\common\lib\jsp-api.Jar 包内。Tomcat 中也附带了 sun 公司提供的 JSP API 的帮助文档，与学习和使用其他 API 的方式一样，读者有时候也需要来查阅 JSP API 的帮助文档。Tomcat 4.x 中的 JSP API 帮助文档是与 Servlet API 帮助文档集成在一起的，Tomcat 5.x 中的 JSP API 帮助文档是与 Servlet API 帮助文档分别单独提供的。按照进入 Servlet API 帮助文档的方式，进入 Tomcat 5.x 中的 JSP API 帮助文档，可以查看到 javax.servlet.jsp.HttpJspPage 接口的帮助页面，如图 8.4 所示。

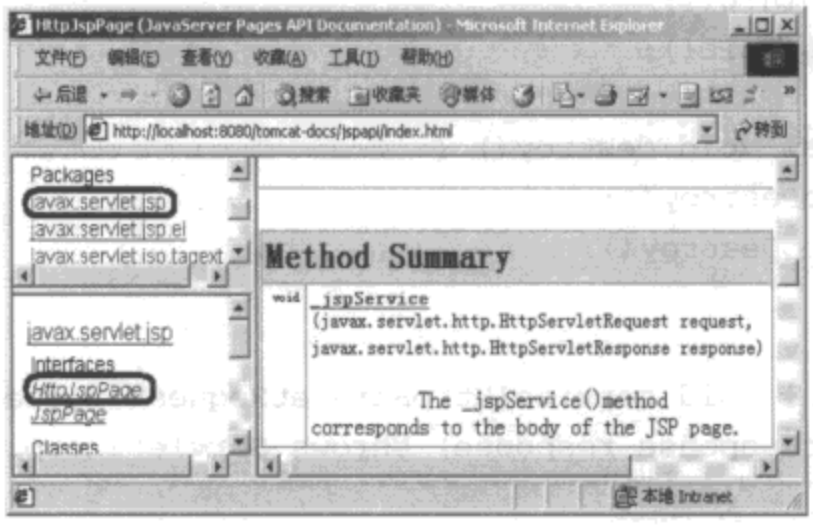


图 8.4

在 HttpJspPage 接口中只定义了一个 ***_jspService*** 方法，但它继承了 JspPage 接口，JspPage 接口中定义了两个方法：jspInit()和 jspDestroy()。从 simple.jsp 生成的 simple_jsp.java 和 HttpJspBase.java 源文件可以看到，***simple_jsp*** 类实现了 ***_jspService*** 方法，***HttpJspBase*** 类实现了 jspInit()和 jspDestroy 这两个方法。

（3）从 org.apache.jasper.runtime.HttpJspBase 的源代码中可以看到，其 init 方法内部调用了 jspInit 和_jspInit 这两个方法，其 destroy 方法内部调用了 jspDestroy 和_jspDestroy 这两个方法。在 HttpJspBase 中实现的 init 和 destroy 方法都使用 final 修饰符进行了声明，所以，在 JSP 页面所生成的 Servlet 中不能覆盖这两个方法。如果要在 JSP 页面中完成 Servlet 的 init 方法的功能，只能覆盖 jspInit 和_jspInit 这两个方法中的任何一个，_jspInit 方法是

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

保留给 JSP 引擎使用的（例如，初始化自定义标签处理类的实例对象），JSP 设计人员通常应使用 jspInit 方法来完成自己的初始化工作；同样，如果要在 JSP 页面中完成 Servlet 的 destroy 方法的功能，只能覆盖 jspDestroy 和_jspDestroy 两个方法中的任何一个，_jspDestroy 方法是保留给 JSP 引擎使用的，JSP 设计人员通常应使用 jspDestroy 方法来完成自己的清尾工作。

（4）从 org.apache.jasper.runtime.HttpJspBase 的源代码中可以看到，其 service 方法内部仅仅是调用了***_jspService***方法，所以可以认为***_jspService***方法的作用就相当于 HttpServlet 的 service 方法的作用，即对于 simple.jsp 页面的每次访问请求，Servlet 引擎都会调用***simple_jsp*** 类的***_jspService*** 方法进行响应。仔细分析***_jspService*** 方法内部被笔者以黑斜体进行标记了的那部分代码，可以发现在 simple.jsp 中的位于<% %>外面的每行和紧邻<%%>两边的每段文本都被转换成了以这些文本作为参数的一条条 out.write 语句，JSP 脚本片断（位于<%%>内的一段 java 代码）中的 java 代码被原封不动地搬移进了***_jspService*** 方法中的相应位置处，JSP 表达式（位于<%=和%>之中的内容）则被转换成以其中的变量或表达式作为参数的 out.print 语句。

指点迷津

out.print 语句与 out.write 语句的区别

在 simple_jsp 类的_jspService 方法中，既用到了 out.write 语句，又用到了 out.print 语句进行输出，它们有什么区别呢？在_jspService 方法中查看 out 对象的定义，可以看到 out 对象的类型为 JspWriter。在 JSP API 帮助文档的 javax.servlet.jsp 包中，可以找到 JspWriter 的帮助信息，JspWriter 继承了 java.io.Writer。print 方法是在子类 JspWriter 中定义的，而 write 方法是在父类 Writer 中定义的，重载的 print 系列方法可将各种类型的数据转换成字符串的形式输出，而重载的 write 系列方法只能输出字符、字符数组和字符串等与字符相关的数据。JspWriter 类型的 out 对象使用 print 方法和 write 方法都可以输出字符串，但是，如果字符串对象的值为 null，print 方法将输出内容为“null”的字符串，而 write 方法则是抛出 NullPointerException 异常。编写一个内容如下的 test.jsp 文件，分别注释掉 out.print 和 out.write 两条语句中的一条，查看各自的运行结果，就可以看到这种区别了。

文件清单：test.jsp

```
<%String strTest = null;
out.print(strTest);
//out.write(strTest);
%>
```

8.1.4 JSP 隐式对象

为了简化 JSP 表达式和脚本片断代码的编写，JSP 一共提供了 9 个预先定义的变量，

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

这些变量也被称之为隐式对象，一些书籍也将它们称之为 JSP 内部对象。这 9 个隐式对象分别是 request，response，out，session，application，config，pageContext，page 和 exception，其中，exception 对象只能在处理错误的 JSP 页面（参看 8.3.2 节的 isErrorPage 属性的讲解）中使用。查看各个 JSP 页面翻译成的 Servlet 源文件，几乎所有的_jspService 方法都具有类似如下的程序代码结构：

```
public void _jspService(HttpServletRequest request,
                        HttpServletResponse response)
            throws java.io.IOException, ServletException
{
    JspFactory _jspxFactory = null;
    PageContext pageContext = null;
    HttpSession session = null;
    ServletContext application = null;
    ServletConfig config = null;
    JspWriter out = null;
    Object page = this;
    JspWriter _jspx_out = null;
    PageContext _jspx_page_context = null;

    try {
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/html;charset=gb2312");
            pageContext = _jspxFactory.getPageContext(this,
                    request, response,null, true, 8192, true);
            _jspx_page_context = pageContext;
            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
            out = pageContext.getOut();
            _jspx_out = out;

            ……
            这里是根据 JSP 页面内容转换成的 java 代码
            ……

    } catch (Throwable t) {
            if (!(t instanceof SkipPageException)){
                    out = _jspx_out;
            if (out != null && out.getBufferSize() != 0)
                    out.clearBuffer();
            if (_jspx_page_context != null)
                    _jspx_page_context.handlePageException(t);
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
            }
        } finally {
            if (_jspxFactory != null)
                _jspxFactory.releasePageContext(_jspx_page_context);
        }
    }
```

对于专用于错误处理的 JSP 页面，在_jspService 方法内的开始部分还有如下一段定义 exception 对象的代码：

```
Throwable exception =
    org.apache.jasper.runtime.JspRuntimeLibrary.getThrowable(request);
if (exception != null) {
    response.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
}
```

从上面的_jspService 方法的源代码中，可以清楚地看到 JSP 中的 9 个预定义变量的类型，以及它们是如何产生的，其中的 request 和 response 对象是作为_jspService 方法的参数传递进来的，而其他隐式对象都是在_jspService 方法的开始部分定义的。JSP 脚本片断和 JSP 表达式所生成的 Java 代码都被插入进_jspService 方法内的位于这些变量定义和初始化之后的位置，所以在 JSP 脚本片断和 JSP 表达式中可以直接使用这些变量。由于 JSP 脚本片断和 JSP 表达式所生成的 Java 代码都被插入进了_jspService 方法内的一个 try…catch 处理语句中，所以，即使 JSP 脚本片断和 JSP 表达式中调用了某些声明可能要抛出异常的方法，但没有为此编写专门的异常捕获处理语句，编译也不会发生“未捕获的异常”问题。

除了 page、pageContext 和 out 三个隐式对象外，JSP 中内置的其他隐式对象其实都已经在前面的章节中讲解过了，其中，application 为 ServletContext 类型的实例对象，session 为 HTTPSession 类型的实例对象，config 为 ServletConfig 类型的实例对象，request 为 HttpServletRequest 类型的实例对象，response 为 HttpServletResponse 类型的实例对象。page 对象的类型为 Object，它被赋值为 this，即代表 JSP 页面所翻译成的 Servlet 对象自身，page 对象当前的用处不大，它是为了 Java 不再是惟一的 JSP 编程语言而准备的占位符。学习到目前为止，对 JSP 的 9 个隐式对象来说，读者应该只对其中的 out 和 pageContext 两个对象不熟悉，笔者将在本章后面部分对这两个对象进行详细讲解。

使用 JSP 页面学习 Servlet API

在前面的章节中，如果想通过一个实验来了解 Servlet API 中的 ServletContext，ServletConfig 等接口中定义的某个方法的具体情况，例如，想了解 ServletContext.getRealPath 的用途时，必须编写一个 Servlet 程序来调用这个方法，接着还要对 Servlet 程序进行编译和在部署描述符文件中进行配置，并且每次出错后都还要重新进行手工编译。每次开

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

始这一系列的操作过程时，都会令人感到麻烦和枯燥。有了 JSP 技术以后，则只需在 JSP 文件的脚本片断中直接编写要实验的方法的相关语句，然后用浏览器直接访问这个 JSP 页面即查看到实验结果。可见，JSP 确实比 Servlet 方便了不少。

8.1.5 注册与配置 JSP 页面的访问路径

JSP 页面也可以像 Servlet 程序一样在 web.xml 文件中进行注册，包括宣称它可以处理的请求 URL 和设置供 ServletConfig.getInitParameter 方法调用的初始化参数。在 web.xml 文件中注册 JSP 页面的方式与 Servlet 类似，只需将<servlet-class>元素修改为<jsp-file>元素，例如，在<tomcat 的安装目录>\webapps\it315\Web-INF\web.xml 文件中的相应位置处增加如下两段内容：

```
<servlet>
        <servlet-name>SimpleJspServlet</servlet-name>
        <jsp-file>/jsp/simple.jsp</jsp-file>
        <load-on-startup>1</load-on-startup >
</servlet>
……
<servlet-mapping>
        <servlet-name>SimpleJspServlet</servlet-name>
        <url-pattern>/xxx/yyy.html</url-pattern>
</servlet-mapping>
```

其中，JSP 文件的路径必须以“/”开头，这个“/”表示 JSP 文件所在的 Web 应用程序的根目录。上面也设置了<load-on-startup>子元素，以便 simple.jsp 在 Web 容器启动时就被编译和装载。重新启动 Tomcat，在浏览器地址栏中输入 http://localhost:8080/it315/xxx/yyy.html，返回结果与直接在浏览器地址栏中输入 http://localhost:8080/it315/jsp/simple.jsp 的运行效果一样。

8.1.6 JSP 与 JavaScript 的综合应用

JavaScript 是由浏览器解释执行的程序代码，它是网页内容的一部分，主要用于在网页上产生动态的显示效果和实现与用户的交互功能。JSP 程序则是由 Web 服务器解释执行的程序代码，它用于产生供浏览器显示的网页内容。由于 JavaScript 代码属于网页的内容部分，所以，JavaScript 代码也可以通过 JSP 页面输出。JavaScript 和 JSP 的关系犹如 HTML 代码和 JSP 的关系，即 JavaScript 代码是 JSP 程序输出的一段可以被浏览器解释执行的特殊文本内容。只要先写好一个包含有 JavaScript 代码的网页文档，然后将这个网页文档的内容改为用 JSP 程序输出，这样就可以一下子明白 JSP 与 JavaScript 的区别和应用关系了。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

JSP 与 JavaScript 的综合应用

01 在一个 Web 邮件系统中，当用户打开邮箱的收件夹时，其中的所有邮件标题将在一个网页中列出，如图 8.5 所示。

图 8.5

用户可以选中某些邮件，然后对其执行删除、移动等操作。要标识邮件的选中状态，通常都是在每个邮件旁边放置一个复选框，用户选中某个复选框，即表示用户要选中它旁边的邮件。另外，为了方便用户通过一次操作就选中所有的邮件，网页中通常都要放置一个全选复选框，当选中这个全选复选框时，所有的邮件复选框也随之被自动选中，当去掉全选复选框的选中状态时，所有的邮件复选框的选中状态也随之被去掉。要想通过单击图 8.5 中的“全选”复选框来一次操作就选中所有的邮件，显然需要用一段 JavaScript 代码来实现。由于邮箱中的邮件数目是不确定的，因此图 8.5 中的代表每封邮件的复选框的个数也是不确定的，它们通常是由 ASP、JSP 等服务器端程序动态产生的。在这种情况下，如果为每个复选框都设置一个不同的名称，每个复选框在程序中的引用名称也就各不一样，这样，要通过 JavaScript 将每个复选框设置为选中状态时，不可能用一段循环代码来统一处理所有的复选框，而要为每个复选框的处理都编写单独的程序代码。较好的做法是为每个复选框都设置一个完全相同的名称，但它们各自的设置值不同，分别代表每封邮件的编号，这样，在 JavaScript 中就可以用一个数组来访问这些复选框，从而可以使用一段循环语句来对每个复选框进行处理。另外，将每个复选框的名称都设置成一样，也有利于服务器端程序通过简单的代码就可获知客户端所选中的多封邮件。例程 8-6 是一个模拟图 8.5 中的“全选”复选框功能的 HTML 文件，其中的 JavaScript 代码细节请参看笔者的《JavaScript 网页开发》一书。

例程 8-6 maillist.html

```
<script language="javascript">
    function checkAll()
    {
        var count = document.mailForm.mail.length;
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        for(var i=0; i<count; i++)
        {
                document.mailForm.mail[i].checked =
                        document.mailForm.ifAll.checked;
        }
    }
</script>
<form name="mailForm">
    <input type="checkbox" name="mail" value="1">邮件 1<br>
    <input type="checkbox" name="mail" value="2">邮件 2<br>
    <input type="checkbox" name="mail" value="3">邮件 3<br>
    <input type="checkbox" name="ifAll" onclick="checkAll()">全选
</form>
```

用浏览器直接打开例程 8-6 所示的 maillist.html 文件，单击“全选”复选框，可以看到所有其他的复选框也随之被选中。

02 在实际应用中，某个邮箱的收件夹中可能有多封邮件，也可能只有一封邮件，甚至是一封邮件都没有，而上面的 maillist.html 文件中的 JavaScript 代码不能正确处理只有一封邮件和一封邮件都没有的情况。修改 maillist.html 文件，将其中的与邮件 2 和邮件 3 相关的语句注释掉，如下所示：

```
…
<input type="checkbox" name="mail" value="1">邮件 1<br>
<!--<input type="checkbox" name="mail" value="2">邮件 2<br>
<input type="checkbox" name="mail" value="3">邮件 3<br>-->
<input type="checkbox" name="ifAll" onclick="checkAll()">全选
…
```

刷新浏览器对 maillist.html 文件的访问，单击“全选”复选框，可以看到代表邮件 1 的复选框并没有随之被选中。修改 maillist.html 文件，将其中的与邮件 1、邮件 2 和邮件 3 相关的语句全部注释掉，刷新浏览器对 maillist.html 文件的访问，单击“全选”复选框，浏览器底部的状态栏中将出现“网页上有错误”的提示信息。显然，上面编写的 checkAll 函数只考虑到了收件夹中有多封邮件的情况，而没有考虑到只有一封邮件和一封邮件都没有的情况，在实际应用中必须对这些细微的情况进行考虑。如果收件夹中连一封邮件都没有，那么将不会出现名称为“mail”的复选框，对这种情况可以用下面的任何一条语句进行判断：

- `if(document.mailForm.mail == null)`
- `if(!document.mailForm.mail)`
- `if(typeof(document.mailForm.mail) == "undefined")`

如果收件夹中只有一封邮件，那么将只会出现一个名称为“mail”的复选框，这时用在 JavaScript 代码中的 document.mailForm.mail 表达式将不是一个数组，即 document.mailForm.mail 对象中不会 length 属性，这可以在排除连一封邮件都没有的程序代码的基础

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

上，使用下面的任何一条语句进行判断：

- if(document.mailForm.mail.length == null)
- if(!document.mailForm.mail.length)
- if(typeof(document.mailForm.mail.length) == "undefined")

经过上面的分析，例程 8-6 中的 checkAll 函数可以改写成如下形式：

```
<script language="javascript">
    function checkAll()
    {
            //如果邮箱中存在邮件
            if(document.mailForm.mail != null)
            {
                    //如果邮箱中有多封邮件
                    if(document.mailForm.mail.length != null)
                    {
                            var count = document.mailForm.mail.length;
                            for(var i=0; i<count; i++)
                            {
                                    document.mailForm.mail[i].checked =
                                        document.mailForm.ifAll.checked;
                            }
                    }
                    else    //如果邮箱中只有一封邮件
                    {
                            document.mailForm.mail.checked =
                                    document.mailForm.ifAll.checked;
                    }
            }
    }
</script>
```

用上面的 JavaScript 代码分别测试三种情况，即 maillist.html 文件中连一封邮件都没有、只有一封邮件和有多封邮件，可以看到每次单击“全选”复选框都能得到理想的结果。

多学两招

降低代码的嵌套层次，提高程序的易读性

上面编写的 checkAll 函数中的程序代码的嵌套层次较多，会降低程序的易读性和易理解性，可以通过减少其嵌套层次来增强程序的易读和易理解性。修改后的 checkAll 函数如下：

```
<script language="javascript">
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
function checkAll()
{
        //如果邮箱中不存在邮件，就直接返回。
        if(document.mailForm.mail == null)
        {
                return;
        }

        /*如果邮箱中存在一封或多封邮件，程序的执行流程才会到达下面的部分。下面
        的 if 条件判断邮箱中只有一封邮件的情况。*/
        if(document.mailForm.mail.length == null)
        {
                document.mailForm.mail.checked =
                        document.mailForm.ifAll.checked;
                return;
        }

        //下面的代码用于处理邮箱中有多封邮件的情况
        var count = document.mailForm.mail.length;
        for(var i=0; i<count; i++)
        {
                document.mailForm.mail[i].checked =
                        document.mailForm.ifAll.checked;
        }
    }
</script>
```

不可否认，这里编写的 checkAll 函数中的程序代码确实要比原来的程序代码优雅一些。如果一个函数内部的总体结构就是一个 if...else...语句，并且 if...else...语句的从句中的程序代码很多，如下所示：

```
fuction x()
{
        if(...)
        {
                111...
        }
        else
        {
                222...
        }
}
```

按上面的方式读/写程序时，总是要把 if 和 else 部分的从句结合在一起来考虑和处

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

理，会让人感觉到一下子就面临了较大的任务，无形中增加了大脑的压力和难度。最好的办法就是把大事分解成一件件小事来一一处理，如果将上面的代码结构改写成如下形式：

```
fuction x()
{
        if(...)
        {
            111...
            return;
    }

    //在这里可以先歇一会了，然后再继续考虑下面的代码
    222...
}
```

那么程序员编写和阅读程序代码时，在编写和阅读完 if 语句部分的内容后，至少就会感觉到已经完成了一个小的阶段任务，可以先歇下来喝杯咖啡后再继续编写和阅读后面的代码。如果一个函数内部的总体结构就是在一个单独的 if 条件语句中要嵌套很多代码，特别是这些代码中又出现了其他的结构化语句，如下所示：

```
fuction x()
{
        if(...)
        {
            if(...)
            {
                111...
            }
            else
            {
                222...
            }
        }
}
```

上面程序代码中的嵌套层次较多，会降低程序的易读性和易理解性，则应改写成如下形式来减少嵌套层次：

```
fuction x()
{
        if(!...)
        {
            return;
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        }

        //原来位于里层的 if 语句可以直接在下面编写了。
        if(...)
        {
            111...
        }
        else
        {
            222...
        }
    }
```

对于不是直接嵌套在一个函数中，而是嵌套在 while 和 for 循环中的类似情况，只是将上面的 return 语句改为 continue 语句就可以了。

03 将 maillist.html 文件改名为 maillist.jsp，并保存到<tomcat 的安装目录>\webapps\it315\jsp\目录中，启动 Tomcat 服务器后在浏览器中输入如下地址：

```
http://localhost:8080/it315/jsp/maillist.jsp
```

浏览器窗口中显示出的中文字符为乱码。修改 maillist.jsp 文件，在其最开始处增加如下一条语句：

```
<%@page contentType="text/html;charset=gb2312"%>
```

刷新浏览器对 maillist.jsp 页面的访问，这时候浏览器中显示出了与直接打开 maillist.html 文件的相同效果。单击浏览器的“查看”→“源文件”菜单，可以看到浏览器接收到的内容与 maillist.html 文件的内容完全相同。进入<TOMCAT_HOME>\work\Catalina\localhost\it315\org\apache\jsp\jsp 目录，用记事本程序查看 maillist.jsp 所翻译成的 maillist_jsp.java 源文件，可以看到浏览器中接收到的内容正是通过 maillist_jsp.java 程序中的一行行 out.write 语句输出的结果。

04 如果仅仅只是将上面的 maillist.html 文件的内容改为 maillist.jsp 文件来输出，其意义显然不大，还不如直接使用 maillist.html 文件本身。但是，如果要让浏览器窗口中显示的邮件列表条目能够随着邮件服务器上的邮件信息而改变，那么就必须将 maillist.html 页面中的内容改为使用 JSP 页面来输出了。下面使用一个文本文件来模拟邮件服务器上存储的邮件列表信息，该文本文件中的每一行内容代表一封邮件标题，例程 8-7 的 maillist.jsp 程序根据该文本文件中的内容来产生输出给客户端显示的邮件列表信息，即浏览器访问到的邮件列表信息随着该文本文件中的内容的改变而改变。例程 8-7 的 maillist.jsp 文件中的开始处的前几行 page 指令的语法细节请参看 8.3.2 节和 8.2.4 节的讲解。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

例程 8-7 maillist.jsp

```
<%@page contentType="text/html;charset=gb2312"%>
<%@page import="java.io.*"%>
<%@page import="java.util.*"%>

<%!
    //定义一个方法读取并返回文本文件中的每行数据
    Vector getMails(ServletContext application) throws Exception
    {
        //要注意 mails.txt 文件的位置
        InputStream ips = application.getResourceAsStream("/mails.txt");
        BufferedReader br =
            new BufferedReader(new InputStreamReader(ips));
        Vector vMails = new Vector();
        String mail = br.readLine();

        while(mail != null)
        {
            vMails.add(mail);
            mail = br.readLine();
        }

        br.close();
        return vMails;
    }
%>

<script language="javascript">
    function checkAll()
    {
        //如果邮箱中不存在邮件，就直接返回。
        if(document.mailForm.mail == null)
        {
            return;
        }

        /*如果邮箱中存在一封或多封邮件，程序的执行流程才会到达下面的部分。下面的
        if 条件判断邮箱中只有一封邮件的情况。*/
        if(document.mailForm.mail.length == null)
        {
            document.mailForm.mail.checked =
                    document.mailForm.ifAll.checked;
            return;
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        }

        //下面的代码用于处理邮箱中有多封邮件的情况
        var count = document.mailForm.mail.length;
        for(var i=0; i<count; i++)
        {
            document.mailForm.mail[i].checked =
                    document.mailForm.ifAll.checked;
        }
    }
</script>

<form name="mailForm">
    <%
    //调用 getMails()方法，读取文本文件中的数据并输出给客户端
    Vector v = getMails(application);
    for(Enumeration e = v.elements();e.hasMoreElements();)
    {
        String mail = (String)e.nextElement();
        out.println("<input type ='checkbox' name='mail'>" +
                    mail + "<br>");
    }
    %>
    <input type="checkbox" name="ifAll" onclick="checkAll()">全选
</form>
```

在<tomcat 的安装目录>\webapps\it315 目录中编写一个名为 mails.txt 文件，内容如下：

邮件 1
邮件 2
邮件 3

接着访问 maillist.jsp 页面，看看输出效果并进行全选操作。再接着修改 mails.txt 文件，只留下“邮件 1”这一行内容，刷新浏览器对 maillist.jsp 页面的访问，看看输出效果并进行全选操作。最后删除 mails.txt 文件中的所有内容，刷新浏览器对 maillist.jsp 页面的访问，看看输出效果并进行全选操作。

8.1.7 JSP 与 Servlet 的应用比较

由于 JSP 引擎可以自动将 JSP 页面翻译和编译成 Servlet 程序运行，JSP 几乎提供了 Servlet 的所有好处，同时，JSP 相对 Servlet 还有如下诸多优点：

（1）JSP 是一种以产生网页显示内容为中心的 Web 开发技术，它可以直接使用模板元

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

素来产生网页文档的内容，对于要输出的大量 HTML 文本内容，开发者可以按照编写 HTML 页面的方式来编写 JSP 页面。如果要使用 Servlet 来输出大量的网页文档内容，则必须使用大量的 out.println 语句。

（2）JSP 页面的源文件要比 Servlet 源文件简单，并且 JSP 页面的开发过程要比 Servlet 的开发过程简单得多。JSP 页面不用手工进行编译和在 Web 部署描述符文件中进行设置就可以执行，而 Servlet 源程序需要手工编译和在 Web 部署描述符文件中进行设置后才能执行。

（3）JSP 引擎可以对 JSP 页面的修改进行检测，并自动重新翻译和编译修改过的 JSP 文件，而 Servlet 源文件修改后必须重新进行手工编译。

（4）在 JSP 中使用 JavaBean 和自定义标签，可以分离用于表现外观的 HTML 文本内容和用于处理业务逻辑的 Java 代码。

尽管 JSP 技术有许多优点，但是它的主要优势在于输出网页文档内容，虽然 JSP 也能完成大量的程序流程控制和业务逻辑处理，但就这方面而言，JSP 并不比 Servlet 具有更大的优势。另外，JSP 技术是建立在 Servlet 技术基础之上的，所有的 JSP 页面最终都要被转换成 Servlet 来运行，要想深入掌握 JSP，必须先对 Servlet 有比较充分的理解。在大型 Web 应用程序的开发中，Servlet 与 JSP 经常要混合使用，各司其职，Servlet 通常用做控制组件，并处理一些复杂的后台业务，JSP 则作为显示组件。

对于一个具体的请求处理，是选择 Servlet，还是选择 JSP 页面，可以参考如下几条原则：

（1）如果对于用户的一次请求的响应过程需要用 Java 代码完成大量的流程控制和业务逻辑处理，那么就应该使用 Servlet；

（2）如果对于用户的一次请求的响应过程仅仅是输出丰富的静态网页文档内容和少量的动态内容，那么就应该使用 JSP；

（3）如果对于用户的一次请求的响应过程需要完成大量的流程控制和业务逻辑处理，同时还要输出丰富的网页文档内容，对于这种情况，单独使用 Servlet 或 JSP 都不是好的方案，它们都将混合用于网页显示（即表现外观）的 HTML 文本内容和用于处理业务逻辑的 Java 代码，很难区分开团队人员的专业工作。最好的办法是结合使用 Servlet 和 JSP 来共同完成这种请求的响应过程，将响应过程划分成几个功能模块来协同完成。首先由 Servlet 完成流程控制和业务处理，并将结果数据存储在 request 或 session 域中，然后将请求转发（forward）到 JSP 页面，再由 JSP 页面从 request 或 session 域中取出结果数据并完成响应内容的输出。通过这种方式实现业务逻辑与显示内容的分离，从而将应用程序开发者和网页作者的工作分离。Java 开发人员将不再扮演 HTML 页面设计的角色，而是专注于业务逻辑的 Java 程序开发，同时，不懂 Java 代码的普通 HTML 设计人员也可以编写和维护 jsp 页面，并且能专注于表现层的处理工作。

8.2 JSP 基本语法

JSP 页面中可以包含模板元素、脚本元素、指令、EL（Expression Language）表达式、

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

行为元素和注释等内容。有三种类型的脚本元素：JSP 脚本片断、JSP 表达式和 JSP 声明，它们内部使用的都是 Java 脚本代码。

8.2.1 JSP 模板元素

JSP 页面可以按照编写 HTML 页面的方式来编写，其中可以包含 HTML 文件的所有静态内容，在静态的 HTML 内容之中可以嵌套 JSP 的其他各种元素来产生动态内容和执行业务逻辑，JSP 页面中的静态 HTML 内容称之为 JSP 模板元素。JSP 模板元素定义了网页的基本骨架，即定义了页面的结构和外观。

8.2.2 JSP 表达式

JSP 表达式（expression）提供了将一个 Java 变量或表达式的计算结果输出到客户端的简化方式，它将要输出的变量或表达式直接封装在<%= 和 %>之中。JSP 表达式中的变量或表达式的计算结果将被转换成一个字符串，然后被插入进整个 JSP 页面输出结果的相应位置处。例如，下面的代码将显示 JSP 页面被访问时的日期和时间：

```
Current time: <%= new java.util.Date() %>
```

JSP 表达式中的变量或表达式后面不能有分号（;），JSP 表达式被翻译成 Servlet 程序中的一条 out.print(...)语句，上面的 JSP 表达式在被翻译成的 Servlet 源程序中对应的语句为：out.print(new java.util.Date());。

8.2.3 JSP 脚本片断

JSP 脚本片断（scriptlet）是指嵌套在<% 和 %>之中的一条或多条 Java 程序代码。在 JSP 脚本片断中，可以实现定义变量、执行基本的程序运算、调用其他 Java 类、访问数据库、访问文件系统等普通 Java 程序所能实现的功能。在 JSP 脚本片断中可以直接使用 JSP 提供的隐式对象来完成 Web 应用程序特有的功能，例如，使用 out.println 语句将数据输出到浏览器。由于 JSP 脚本片断中的 Java 代码将被原封不动地搬移进由 JSP 页面所翻译成的 Servlet 的 ***_jspService*** 方法中，所以，JSP 脚本片断之中只能是符合 Java 语法要求的程序代码，除此之外的任何文本、HTML 标记、其他 JSP 元素都必须在脚本片断之外编写。JSP 脚本片断中的 Java 代码必须严格遵循 Java 语法，例如，每条命令执行语句后面必须用分号（;）结束。

在一个 JSP 页面中可以有多个脚本片断(每个脚本片断代码嵌套在各自独立的一对<% 和 %>之间)，在两个或多个脚本片断之间可以嵌入文本、HTML 标记和其他 JSP 元素，如下所示：

```
<%
    int x = 3;
%>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
<p>这是一个 HTML 段落</p>
<%
    out.println(x);
%>
```

多个脚本片断中的代码可以相互访问，犹如将所有的代码放在一对<% %>之中的情况，例如，上面的 JSP 内容与下面的 JSP 内容具有同样的运行效果：

```
<p>这是一个 HTML 段落</p>
<%
    int x = 3;
    out.println(x);
%>
```

单个脚本片断中的 Java 语句可以是不完整的，但是，多个脚本片断组合后的结果必须是完整的 Java 语句，例如，涉及条件和循环处理时，多个脚本片断及其他元素组合的结果必须能形成完整的条件和循环控制语句。由于脚本片断中的 Java 代码将被原封不动地搬移进由 JSP 页面所翻译成的 Servlet 的_jspService 方法中，脚本片断之外的任何文本、HTML 标记和其他 JSP 元素也都会被转换成相应的 Java 程序代码插入进_jspService 方法中，且脚本片断和其他 JSP 元素的插入位置与它们在 JSP 页面中的原始位置相对应，因此，在脚本片断中可以使用条件、循环、选择等流程控制语句来创建其周围的其他元素的执行逻辑。在编写 JSP 页面时应考虑各个元素之间的先后顺序和相互关系，特别是将循环、条件判断等语句分布在若干个脚本片断中编写时对其邻近的其他 JSP 元素产生的影响，例如下面的 JSP 程序组合使用脚本片断、表达式和模板元素：

```
<%for (int i=1; i<5; i++) {%>
    <H<%=i%>>www.it315.org</H<%=i%>>
<%}%>
```

转换成 Servlet 后对应的 Java 代码如下：

```
for (int i=1; i<5; i++) {
    out.write("\r\n");
    out.write("\t<H");
    out.print(i);
    out.write(">www.it315.org</H");
    out.print(i);
    out.write(">\r\n");
}
```

程序执行结果如图 8.6 所示。

下面是使用 if...else 语句进行条件控制的情况：

```
<%if (java 条件表达式) {%>
    其他元素
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
<%}else{%>
    其他元素
<%}%>
```

图 8.6

8.2.4 JSP 声明

JSP 页面最终要被转换成 Servlet 程序，JSP 页面中的脚本片断、表达式、模板元素等都将转换成 Servlet 的_jspService 方法中的程序代码，显然，在脚本片断中声明的变量将成为_jspService 方法中的局部变量。另外，在脚本片断中声明的方法也将被插入到_jspService 方法中，这就会出现在方法中定义方法的语法错误。如果希望在 JSP 页面中为其转换成的 Servlet 定义成员变量和方法，那该怎么做呢？JSP 声明（declaration）专门用来解决这个问题。JSP 声明将 Java 代码封装在<%!和%>之中，它里面的代码将被插进 Servlet 的_jspService 方法的外面，所以，JSP 声明可用于定义 JSP 页面转换成的 Servlet 程序的静态代码块、成员变量和方法，并且 JSP 声明中的 Java 代码只能是静态代码块、变量和方法的定义，除此之外的情况都将视为非法。多个静态代码块、变量和函数可以定义在一个 JSP 声明中，也可以分别单独定义在多个 JSP 声明中。由于 JSP 隐式对象的作用范围仅限于 Servlet 的_jspService 方法，所以在 JSP 声明中不能使用这些隐式对象。例程 8-8 是一个使用 JSP 声明的例子程序。

例程 8-8 declare.jsp

```
<%!
static { System.out.println("loading Servlet!"); }
private int globalVar = 0;
public void jspInit()
{
    System.out.println("initializing jsp!");
}
%>
<%!
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
public void jspDestroy()
{
    System.out.println("destroying jsp!");
}
%>
<%
    int localVar = 0;
%>
globalVar:<%= ++globalVar %><br>
localVar:<%= ++localVar %>
```

用浏览器访问 declare.jsp 页面后，JSP 引擎把它翻译成了如例程 8-9 所示的 Servlet 源程序。

例程 8-9　declare_jsp.java

```
......
public final class declare_jsp extends org.apache.jasper.runtime.
HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent
{

    static { System.out.println("loading Servlet!"); }
    private int globalVar = 0;
    public void jspInit()
    {
        System.out.println("initializing jsp!");
    }

    public void jspDestroy()
    {
        System.out.println("destroying jsp!");
    }
    ......

    public void _jspService(HttpServletRequest request,
                            HttpServletResponse response)
            throws java.io.IOException, ServletException
    {
            ......
            int localVar = 0;
            out.write("globalVar:");
            out.print( ++globalVar );
            out.write("<br> \r\n");
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
            out.write("localVar:");
            out.print( ++localVar   );
            ......
        }
    }
```

首次访问 declare.jsp 页面时，浏览器窗口中显示的 globalVar 和 localVar 变量值都为 1，在启动 Tomcat 的命令行窗口中打印出了如下两行信息：

```
loading Servlet!
initializing jsp!
```

刷新浏览器对 declare.jsp 页面的访问，可以看到 globalVar 变量的值随之增长，而 localVar 变量的值始终为 1。在 declare.jsp 页面中的任意处增加一空格后又删除该空格，然后用浏览器重新访问 declare.jsp 页面，JSP 引擎将以为 declare.jsp 页面发生了修改而重新编译和加载 declare.jsp 页面，这时在启动 Tomcat 的命令行窗口中又打出了如下三行信息：

```
destroying jsp!
loading Servlet!
initializing jsp!
```

下面是对 declare.jsp 页面的运行结果的解释：

（1）Java 虚拟机加载完一个类的字节码后，将立即调用其中的静态代码块，静态代码块的调用与是否创建实例对象无关，并且发生在创建第一个实例对象之前，所以在启动 Tomcat 的命令行窗口中首先打印出了“loading Servlet!”。

（2）Servlet 引擎在加载并创建某个 Servlet 实例对象后将调用其 init 方法，以便执行 Servlet 的一些初始化工作；Servlet 引擎在卸载某个 Servlet 实例对象之前调用其 destroy 方法，以便执行 Servlet 的一些清尾工作。JSP 页面所生成的 Servlet 的 init 方法和 destroy 方法内部分别调用了 jspInit 方法和 jspDestroy 方法，所以在 JSP 页面中应该使用 jspInit 方法和 jspDestroy 方法进行初始化和清理工作。在创建 JSP 页面所生成的 Servlet 的实例对象后，Servlet 引擎调用其 jspInit 方法，所以在启动 Tomcat 的命令行窗口中接着打印出了“initializing jsp!”。修改 JSP 页面后，旧的 JSP 页面生成的 Servlet 被卸载，Servlet 引擎调用其 jspDestroy 方法，所以在启动 Tomcat 的命令行窗口中接着打印出了“destroying jsp!”，Servlet 引擎接着加载新生成的 Servlet 并调用其 jspInit 方法，所以在命令行窗口中又打印出了“loading Servlet!”和“initializing jsp!”。

（3）在 JSP 声明中定义的 globalVar 被翻译成了 Servlet 的成员变量，而在脚本片断中定义的 localVar 被翻译成了_jspService 方法中的一个局部变量。在 JSP 页面没有发生修改的情况下，对于 JSP 页面的每次访问请求，Servlet 引擎都会调用其所生成的 Servlet 的_jspService 方法进行响应。每次调用_jspService 方法时所访问的 globalVar 都是同一个变量，所以其值不断增长，而每次调用_jspService 方法时所访问的 localVar 是各自独立的变量，所以这些 localVar 变量的值总是 1。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

8.2.5 EL 表达式

EL（Expression Language）表达式语言是 JSP 2.0 中新增加的一种可简化 JSP 开发的技术，它可以非常方便地访问 JSP 页面相关的数据和支持基本的算术、关系和逻辑运算。EL 表达式的基本语法格式为“${表达式}”，它可以出现在 JSP 自定义标签和标准标签的属性值中，其计算结果将作为标签的属性值或属性值的一部分；EL 表达式也可以出现在模板元素中，其计算结果将插入进当前的输出流中。EL 表达式相比 JSP 脚本元素而言，具有以下一些优点：

（1）使用 EL 表达式可以非常方便地访问 JSP 页面相关的数据，例如，要访问名称为“user”的请求参数，其 JSP 表达式为：

```
<%= reqeust.getParameter("user") %>
```

而相应的 EL 表达式为：

```
${param.user}
```

要访问 JavaBean 中嵌套的属性，其 JSP 表达式通常为：

```
<%CustomerBean custormBean =
            (CustomerBean)pageContext.findAttribute("customerBean");%>
<%= customerBean.getAddress().getCountry()%>
```

其相应的 EL 表达式为：

```
${customerBean.address.country}
```

在 EL 表达式中还可以直接使用${cookie.user}来访问名称为 user 的 Cookie 信息，在 JSP 脚本表达式中却无法实现这样的功能，虽然使用 JSP 脚本片断也可以实现这个功能，但是其程序代码却比较复杂。

（2）如果表达式的计算结果为 null，JSP 脚本表达式将输出内容为“null”的字符文本，而 EL 表达式会将它强制转换成空字符串（""）后再输出，而不是显示为“null”，这样的处理结果非常有用。

（3）在 EL 表达式中还可以使用 empty 操作符来判断一个参数是否为 null，某个集合或字符串中的内容是否为空。

（4）EL 表达式还具有 JavaScript 等脚本语言的优点，能够对字符串、数字等各种类型的数据进行自动转换，摆脱了 Java 语言严格的数据类型限制，使得应用起来更加简单。

尽管 EL 表达式具有如此多的优点，但它仅仅是专门为网页作者量身定做的一种数据访问语言，可以方便地访问和处理应用程序数据，但是它并不是一种程序设计语言，不能替代 Java 语言的强大功能。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

EL 表达式将 null 输出为空字符串（""）的好处

01 当用户在一个网站上填写注册信息时，如果其填写的信息有误，网站通常都会重新返回刚才的注册页面给用户，以便用户重新输入注册信息。在返回的让用户重新输入注册信息的页面中，除了显示出错误的提示信息外，通常还要将用户上次输入的注册信息作为相应的 FORM 表单字段元素的初始值。例程 8-10 的 register.jsp 页面是一个简单的模拟程序，它将表单的提交请求交给 register.jsp 页面自己处理，即用户提交表单后直接又回到了 register.jsp 页面，要求返回的页面中保留有用户上次填写的信息。

例程 8-10 register.jsp

```
<form method="post">
user:<input type="text" name="user"><br>
<input type="submit" value="register">
</form>
```

用浏览器访问 register.jsp 页面后，在“user”文本框中填写一个任意名称，接着单击“register”按钮，在重新返回的 register.jsp 页面的“user”文本框中没有显示出上次填写的内容。

02 修改例程 8-10 的 register.jsp 文件，使用 JSP 脚本表达式设置“user”文本框的初始值，如例程 8-11 所示。

例程 8-11 register.jsp

```
<form method="post">
user:<input type="text" name="user" value="<%= request.getParameter
("user") %>">
<br>
<input type="submit" value="register">
</form>
```

重新启动一个浏览器窗口，重复上面的操作过程，尽管通过单击“register”按钮后返回的 register.jsp 页面的“user”文本框中都能显示出上次填写的内容，但是在首次访问到的 register.jsp 页面的“user”文本框中也显示出了一个内容为“null”的字符文本，这是因为第一次访问 register.jsp 页面的请求消息中没有名称为“user”的参数，request.getParameter ("user")返回的结果为 null，这显然是一个需要解决的问题。

03 再次修改 register.jsp 文件，这次改为使用 EL 表达式设置“user”文本框的初始值，如例程 8-12 所示。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

例程 8-12 register.jsp

```
<form method="post">
user:<input type="text" name="user" value="${param.user}"><br>
<input type="submit" value="register">
</form>
```

重新启动一个浏览器窗口，重复上面的操作过程，这次在首次访问到的 register.jsp 页面的“user”文本框中没有显示出内容为“null”的字符文本，通过单击“register”按钮后返回的 register.jsp 页面的“user”文本框中也都能显示出上次填写的内容。

8.2.6 JSP 注释

与其他各种编程语言一样，JSP 也有自己的注释方式，其格式如下：

```
<%-- 注释信息 --%>
```

JSP 引擎在将 JSP 页面翻译成 Servlet 程序时，忽略 JSP 页面中被注释的内容，即不考虑被注释的内容。

JSP 注释与 JSP 页面中的 Java 注释和 HTML 注释的区别

01 编写一个内容如例程 8-13 所示的 JSP 页面。

例程 8-13 comment.jsp

```
<%-- this is JSP comment --%>
<%
    /* this is java comment */
    //another java comment
%>
<!-- this is HTML comment -->
```

02 使用 IE 浏览器访问 comment.jsp 页面后，上面的三段内容都没有在浏览器窗口中显示出来，单击“查看”→“源文件”菜单，可以看到 JSP 页面向浏览器输出了最后的那段 HTML 注释文本，只是浏览器自己没有显示这段文本。

03 查看由 comment.jsp 文件所翻译成的 Servlet 源文件，其中与 comment.jsp 页面内容相关的主要代码如下：

```
public void _jspService(HttpServletRequest request,
                        HttpServletResponse response)
    throws java.io.IOException, ServletException {
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        ......
        /* this is java comment */
        //another java comment
        out.write("\r\n");
        out.write("<!-- this is HTML comment -->\r\n");
        ......
    }
```

可见，JSP 引擎在翻译 JSP 页面时，将忽略 JSP 注释中的所有内容。在 JSP 引擎看来，出现在 JSP 脚本元素中的 Java 注释属于 Java 代码的一部分，它们将被原封不动地插入进 Servlet 源文件中，但在 Servlet 源文件被编译成 class 文件时由 Java 编译程序忽略。在 JSP 引擎看来，JSP 页面中的 HTML 注释就是一段文本，其处理方式与普通的文本内容完全一样，都被转换成 out.write 语句输出到客户端。

注意 出现在 JSP 脚本元素之外的/*、*/和//不再属于 Java 注释，而属于普通的文本字符，将被转换成 out.write 语句输出到客户端。由于 HTML 注释相当于一段普通文本，其中当然可以嵌入 JSP 的其他各种元素，JSP 引擎依然会对这些嵌套的 JSP 元素进行解释处理，例如，对于下面的语句：

```
<!--This page was loaded on <%=(new java.util.Date()).toLocaleString()
%>-->
```

输出到客户端的文本内容如下所示：

```
<!--This page was loaded on 2005-7-28 14:36:01-->
```

8.2.7 特殊字符序列的转义处理

由于三种 JSP 脚本元素的分界符都使用到了“<%”和“%>”，如果要在 JSP 模板元素和 JSP 脚本元素内部表示字面意义的“<%”和“%>”，需要进行转义处理。由于 EL 表达式使用“${”和“}”作为分界符，在一些应用中，也需要对它们进行转义处理。JSP 页面中的具体的转义处理规则如下。

1. EL 表达式内的转义

EL 表达式内的任何字符序列都不需要进行转义处理，其中的“${”和“}”与普通字符序列的处理方式完全相同，例如，${'${'}和${"${"}都可直接输出 ${，${'}'}和${"}"}都可直接输出}，这就好像${'xyz'}和${"xyz"}的输出结果都是 xyz 一样的道理，“${”、“}”和“xyz”都是 EL 表达式中的字符序列。

2. JSP 脚本元素内的转义

因为脚本元素开始处已经使用了字符序列“<%”来表示脚本元素的开始，JSP 引擎不

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

会再把脚本元素内的字符序列“<%”当作脚本元素的开始分界符处理，所以，脚本元素内部可以直接使用字符序列“<%”来表达其字面意义。但是，为了使 JSP 引擎不要把脚本元素内的字符序列“%>”当作脚本元素的结束分界符处理，要在脚本元素内表达字面意义上的“%>”，必须用“%\>”进行转义。JSP 引擎在生成 Servlet 源文件时，将把脚本元素内部的“%\>”转换成“%>”。因为“%\>”在脚本元素内要被转义处理，如果要表达字面意义上的“%\>”，应该使用“%\\>”。下面是一个在脚本元素内表达字面意义的“<%”和“%>”的例子语句：

```
<% out.println("use <% and %\>"); %>
```

在浏览器中使用查看源文件的方式可以看到上面的语句执行后输出到客户端的结果为：

```
use <% and %>
```

3. JSP 模板元素内的转义

为了让 JSP 引擎不要把 JSP 模板元素内的字符序列“<%”当作脚本元素的开始，要在 JSP 模板元素内表达字面意义上的“<%”，必须用“<\%”进行转义。因为在模板元素中，JSP 引擎还没有遇到脚本元素的开始分界符，它不会把“%>”当作脚本元素的结束分界符处理，所以，JSP 模板元素内可以直接使用字符序列“%>”来表达其字面意义。因为“<\%”在 JSP 模板元素内要被转义处理，如果要表达字面意义上的“<\%”，应该使用“<\\%”。下面是一个在 JSP 模板元素内表达字面意义的“<%”和“%>”的例子语句：

```
use <\% and %>
```

在浏览器中使用查看源文件的方式可以看到上面的语句执行后输出到客户端的结果为：

```
use <% and %>
```

为了让 JSP 引擎不要把 JSP 模板元素内的字符序列“${”当作 EL 表达式的开始，在支持 EL 表达式的 JSP 页面中，“$”可以使用“\$”来表示，但这不是必需的。如果要表达字面意义上的“${”，这时候就必须使用“\${”。如果表达字面意义上的“\$”，必须使用“\\$”。

4. JSP 标签的属性内的转义

JSP 标签是指 JSP 规范定义的一些标准标签或 JSP 开发人员自己开发的一些标签，它们不同于 HTML 标签，详细信息请参看 8.6 节的讲解。JSP 标签中出现的一些特殊字符需要按如下规则进行转义处理：

- 如果使用单引号（'）将属性值部分引起来，那么属性值中的单引号要用“\'”表示。
- 如果使用双引号（"）将属性值部分引起来，那么属性值中的双引号要用“\"”表示。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

- 反斜杠（\）要用“\\”表示。
- 在支持 EL 表达式的 JSP 页面中，“$”可以使用“\$”来表示。
- “<%”要用“<\%”表示。
- “%>”要用“%\>”表示。

注意 这里要注意区分 JSP 页面内的 HTML 标签与 JSP 标签，出现在 JSP 页面中的 HTML 标签属于模板元素的一部分，它们属性的转义处理方式与普通模板元素内容的处理方式完全一样，而不能采用 JSP 标签的属性内的转义方式。

8.2.8 如何查找 JSP 页面中的错误

在编写 JSP 页面的过程中，难免要遇到形形色色的错误，而排除这些错误是令初学者感到非常棘手的事情。另外，当用浏览器成功访问一个 JSP 页面后，接着修改 JSP 页面，如果修改过的 JSP 页面有错误，再刷新浏览器对这个 JSP 页面的访问，浏览器将显示出修改后的错误信息，但是，如果连续多次刷新几次，浏览器都会显示出 JSP 页面修改前的返回结果，这也给许多初学者带来了极大的困惑，搞不清 JSP 页面的真实返回结果是哪个？JSP 页面到底有没有错误？到底是怎样的错误？

查找 JSP 页面中的错误

01 在<tomcat 的安装目录>\webapps\it315\jsp\目录中编写一个名为 debug.jsp 的 jsp 文件，如例程 8-14 所示。

例程 8-14 debug.jsp

```
<%= "it315" %>
```

用浏览器访问 debug.jsp 页面，浏览器中显示的结果如图 8.7 所示。

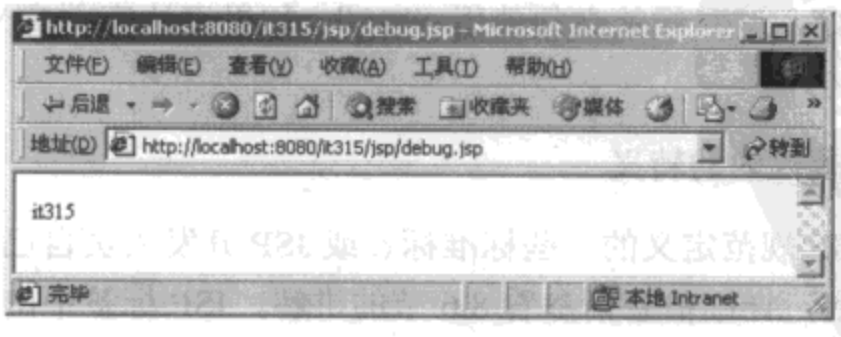


图 8.7

02 修改 debug.jsp 文件，去掉"it315"两边的双引号，如例程 8-15 所示。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

例程 8-15 debug.jsp

```
<%= it315 %>
```

刷新浏览器对 debug.jsp 页面的访问，浏览器显示出如图 8.8 所示的错误信息。

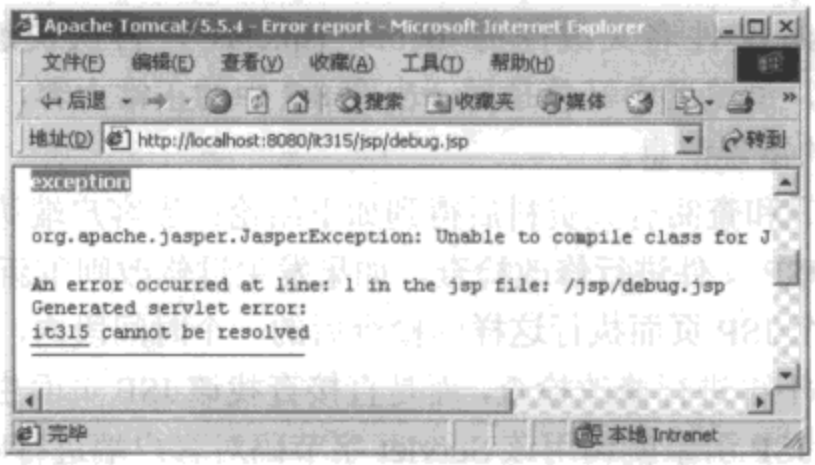


图 8.8

快速连续刷新浏览器对 debug.jsp 页面的访问，浏览器中又总是显示出图 8.7 中的正常内容，这个过程中会间歇显示一下如图 8.8 所示的错误信息。

03 修改 debug.jsp 文件，去掉后面的一个“%”，如例程 8-16 所示。

例程 8-16 debug.jsp

```
<%= it315 >
```

刷新浏览器对 debug.jsp 页面的访问，浏览器显示出如图 8.9 所示的错误信息。

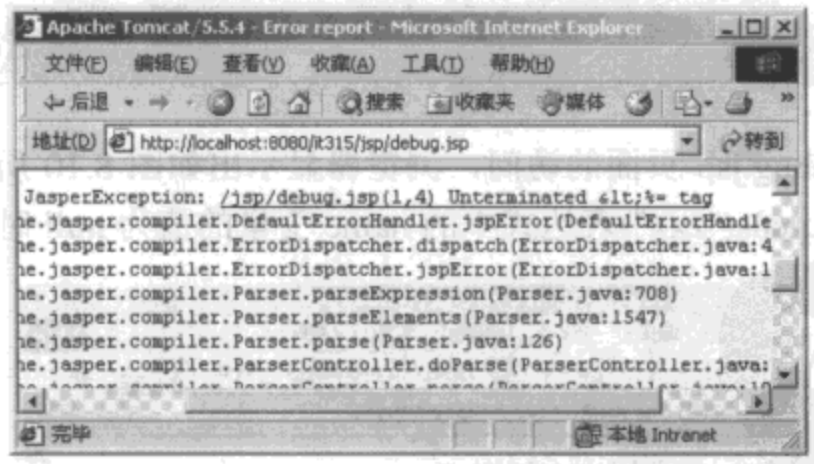


图 8.9

快速连续刷新浏览器对 debug.jsp 页面的访问，浏览器中又总是显示出图 8.7 中的正常内容，这个过程中会间歇显示一下如图 8.9 所示的错误信息。

04 进入<TOMCAT_HOME>\work\Catalina\localhost\it315\org\apache\jsp\jsp 目录，可以看到一个名为 debug_jsp.class 的文件，这说明只要一个 JSP 页面被成功地翻译和编译成 Servlet 字节码后，如果以后修改了 JSP 页面，并且其中的错误导致 JSP 引擎无法重新翻译和

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

编译成 Servlet 字节码，JSP 引擎并不会删除上次编译成功的那个旧的 Servlet 字节码。在该目录中不存在名为 debug_jsp.java 的文件，这是由于 JSP 引擎在翻译 JSP 文件时将首先删除原来生成的 Java 源文件，而 debug.jsp 文件中的错误导致 JSP 引擎无法将它翻译成新的 Java 源文件。

修改 debug_jsp.class 文件的名称为 debug_jsp_bak.class，重新启动 Tomcat，快速连续刷新浏览器对 debug.jsp 页面的访问，这时浏览器总是固定出现如图 8.9 所示的错误信息。将 debug_jsp_bak.class 文件名又改回为 debug_jsp.class，再次快速连续刷新浏览器对 debug.jsp 页面的访问，浏览器中又总是显示出图 8.7 中的正常内容，这个过程中会间歇显示一下如图 8.9 所示的错误信息。

笔者经过仔细研究和查阅有关资料后得到如下结论：当客户端访问一个 JSP 页面时，JSP 引擎将对相应的 JSP 文件进行修改检查，如果发生过修改则重新翻译和编译该 JSP 文件。在 JSP 引擎对一个 JSP 页面执行这样的检查后的一个时间段内，如果客户端又访问该 JSP 页面，JSP 引擎将不再进行修改检查，而是直接查找该 JSP 页面生成的 Servlet 字节码，无论新旧，只要找到，JSP 引擎就调用该 Servlet 字节码对客户端进行响应。当修改一个 JSP 页面出现错误后，不要快速连续刷新浏览器对它的访问，而是每次等待一会后再进行刷新，就能查看到其真实的错误信息。

05 修改 debug.jsp 文件，以便演示一个运行时的 NullPointerException 异常，如例程 8-17 所示。

例程 8-17 debug.jsp

```
<%String name = request.getParameter("name");
    if(name.equals("zxx"))
    {
        out.println("ok!");
    }
%>
```

刷新浏览器对 debug.jsp 页面的访问，浏览器显示出如图 8.10 所示的错误信息，这是因为程序中的 name.equals("zxx")表达式中的 name 变量的值为 null 时不能调用它的方法。

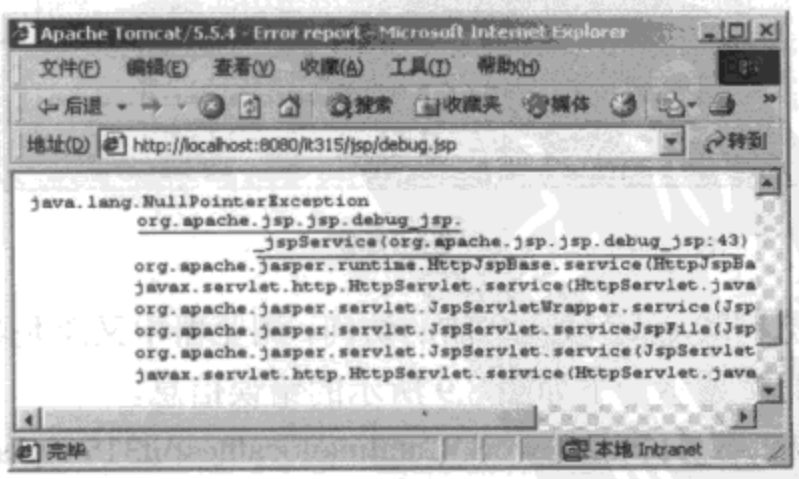


图 8.10

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

通过上面的实验，可以将 JSP 页面中经常遇到的错误归纳为以下三类：

（1）JSP 页面中的 JSP 语法格式有问题，导致其不能被翻译成 Servlet 源文件，JSP 引擎将提示这类错误发生在 JSP 页面中的位置（行和列）及相关信息，如图 8.9 所示。

（2）JSP 页面中的 JSP 语法格式没有问题，但被翻译成的 Servlet 源文件中出现了 Java 语法问题，导致 JSP 页面翻译成的 Servlet 源文件不能通过编译，JSP 引擎也将提示这类错误发生在 JSP 页面中的位置（行和列）及相关信息，如图 8.8 所示。

（3）JSP 页面翻译成的 Servlet 程序在运行时出现异常，这与普通 Java 程序的运行时错误完全一样，Java 虚拟机将提示错误发生在 Servlet 源文件中的位置（行和列）以及相关信息，如图 8.10 所示。

总之，只要注意服务器返回到浏览器中的提示信息，排除 JSP 页面中所发生的错误根本就不是一件困难的事情。

8.3 JSP 指令

8.3.1 JSP 指令简介

JSP 中还定义了一种称之为 JSP 指令（directive）的元素，JSP 指令是为 JSP 引擎而设计的，它们并不直接产生任何可见输出，而只是告诉引擎如何处理 JSP 页面中的其余部分。

JSP 指令必须嵌套在<%@ 和%>之中，其基本语法格式如下：

```
<%@ 指令 属性名="值" %>
```

其中，属性名是大小写敏感的。下面是一条设置 page 指令的 contentType 属性的语句：

```
<%@ page contentType="text/html;charset=gb2312"%>
```

在目前的 JSP 2.0 中，定义了 page，include 和 taglib 三种指令，每种指令中又都定义了一些各自的属性。这里只介绍 page 和 include 指令，对于 taglib 指令，则放在讲解自定义标签时专门介绍。

如果要在一个 JSP 页面中设置同一条指令的多个属性，可以使用多条指令语句单独设置每个属性，也可以使用同一条指令语句设置该指令的多个属性，例如下面的两种指令设置方式具有相同的效果。

第一种方式：

```
<%@ page contentType="text/html;charset=gb2312"%>
<%@ page import="java.util.Date"%>
```

第二种方式：

```
<%@ page contentType="text/html;charset=gb2312" import="java.util.Date"%>
```

8.3.2 page 指令

page 指令用于定义 JSP 页面的各种属性，几乎所有的 JSP 页面中都会出现 page 指令。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

无论 page 指令出现在 JSP 页面中的什么地方，它作用的都是整个 JSP 页面，为了保持程序的可读性和遵循良好的编程习惯，page 指令最好是放在整个 JSP 页面的起始位置。JSP 2.0 规范中定义的 page 指令的完整语法如下：

```
<%@ page
    [ language="java" ]
    [ extends="package.class" ]
    [ import="{package.class | package.*}, ..." ]
    [ session="true | false" ]
    [ buffer="none | 8kb | sizekb" ]
    [ autoFlush="true | false" ]
    [ isThreadSafe="true | false" ]
    [ info="text" ]
    [ errorPage="relative_url" ]
    [ isErrorPage="true | false" ]
    [ contentType="mimeType [ ;charset=characterSet ]" | "text/html ;
charset=ISO-8859-1" ]
    [ pageEncoding="characterSet | ISO-8859-1" ]
    [ isELIgnored="true | false" ]
%>
```

其中的每对方括号（[ ]）分别描述 page 指令的一个属性，属性值中的竖杠（|）用于分隔该属性的多个可选值，黑体标记的部分为该属性的默认值。import 属性是 page 指令的惟一一个允许在同一个 JSP 页面中设置多次的属性，且其每次的设置值都不相同，除此之外，一个 JSP 页面中不允许出现多个设置值不同的同一属性，但允许有多个设置值完全相同的同一属性，这种重复设置实际上也只相当于对该属性设置了一次。下面是对各个属性的详细说明。

1. language 属性

language 属性用来说明 JSP 页面所使用的脚本语言，其默认值为 java。Java 是当前惟一可用的语言，因此该属性可以不用设置。

2. extends 属性

extends 属性用来指定 JSP 页面翻译成的 Servlet 所继承的父类，一般不要设置这个属性，而是让 JSP 引擎自行处理就足够了。如果在某些特殊情况下要使用这个属性，应该注意它对 JSP 页面的其他部分所造成的影响。

3. import 属性

import 属性指定在 JSP 页面翻译成的 Servlet 源文件中导入的包或类。在 Java 源程序中可以省略其引用的类的包名部分，但此时必须在源文件的开始处使用 import 语句来指明这个类的包名或者是这个类的完整名称。同样，如果在一个 JSP 页面的脚本元素中省略了

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

某个类的包名部分，那么也必须使用 page 指令的 import 属性来指明这个类的包名或者完整名称，在 JSP 页面翻译成的 Servlet 源文件中，JSP 引擎将把 page 指令的 import 属性翻译成相应的 import 语句。JSP 引擎自动导入下面的包：

- java.lang.*
- javax.servlet.*
- javax.servlet.jsp.*
- javax.servlet.http.*

因此，在 JSP 页面中不必再显式地导入这些包。如果在一个 JSP 页面中要使用到上述包之外的多个类或包，可以在一条 page 指令的 import 属性中设置所有要引入的类或包，其中的每个包或类之间使用逗号分隔，如下所示：

```
<%@ page import="java.util.Date,java.sql.*,java.io.*"%>
```

上面的语句也可以改写为使用多条 page 指令的 import 属性来分别引入各个包或类，如下所示：

```
<%@ page import="java.util.Date"%>
<%@ page import="java.sql.*"%>
<%@ page import="java.io.*"%>
```

4. session 属性

session 属性用于指定在 JSP 页面翻译成的 Servlet 中是否预先创建好 session 隐式对象，也就是指定在 JSP 脚本元素中是否可以使用 session 隐式对象。如果将 session 属性设置为 false，则表明在 JSP 页面翻译成的 Servlet 中将不自动预先创建好 session 隐式对象，即在 JSP 页面翻译成的 Servlet 中不包含如下两条语句：

```
    HttpSession session = null;
……
session = pageContext.getSession();
```

这样，在 JSP 脚本元素中就不能使用 session 隐式对象了，对 session 变量的访问将会导致编译错误。page 指令的 session 属性的默认值为 true，即默认创建 session 隐式对象。如果 Web 容器中已经存在当前客户端的会话对象，则让 session 变量绑定到该会话对象上；如果 Web 容器中不存在当前客户的会话对象，则新建一个会话对象并将 session 变量绑定到该对象上。

有一个爱思考的学员曾经问过我一个问题："Session 对象被创建后将在浏览器的会话期间一直存在，怎么会出现 page 指令的 session 属性设置为 false 时就变得不可访问了的情况呢？例如，浏览器首先访问一个 JSP 页面，该 JSP 页面创建了一个 Session 对象后，浏览器接着访问第二个 JSP 页面，即使第二个 JSP 页面的 page 指令的 session 属性设置为 false，在第一个 JSP 页面中创建的那个 Session 对象也应该存在和保持有效啊？怎么第二个 JSP

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

页面就无法访问那个 Session 对象了，我实在搞不明白？”

其实，在第一个 JSP 页面中创建的 Session 对象对第二个 JSP 页面仍然是有效的，只是第二个 JSP 页面所翻译成的 Servlet 中没有定义一个 session 变量和没有让 session 变量去引用那个 Session 对象罢了，所以第二个 JSP 页面不能使用 session 这个隐式变量，但是，在第二个 JSP 页面的脚本代码中仍然可以使用如下语句来获得那个 Session 对象：

```
HttpSession session1 = pageContext.getSession();
```

注意 如果某个 JSP 页面中没有使用 Session 隐式对象，那么最好是将该页面的 page 指令的 session 属性设置为 false。例如，有的客户端只是不经意地访问了 Web 站点上的某个 JSP 页面，而没有访问其他任何页面就离开了，如果 Web 站点不需要对这样的访客进行会话跟踪，Web 容器就不必为这样的客户端创建对应的 Session 对象。如果没有将这个 JSP 页面的 page 指令的 session 属性设置为 false，Web 容器将为这样的客户端创建 Session 对象，这就造成了内存资源浪费。

5. buffer 属性

buffer 属性用于指定 out 隐式对象（详细介绍参见 8.4 节）的缓冲区大小，其默认值因服务器而定，一般为 8 KB。如果将 buffer 属性设置为 none，则 out 对象不使用缓冲区。

6. autoFlush 属性

autoFlush 属性用于设置当 out 隐式对象的缓冲区已满时，是将其中的内容刷新到客户端，还是抛出缓冲区溢出的异常。其默认值为 true，表明缓冲区已满时刷新其中的内容。如果将 buffer 属性设置为了 none，那么就不能把 autoFlush 属性设置为 false，因为将 buffer 属性设置为 none，这就相当于缓冲区总是满的情况，而将 autoFlush 属性设置为 false，则表明缓冲区已满时抛出异常，这就产生了冲突。

7. isThreadSafe 属性

isThreadSafe 属性用于设置 JSP 页面是否是线程安全的，是否还需要让 JSP 页面翻译成的 Servlet 实现 SingleThreadModel 接口。其默认值 true，表明 JSP 页面开发者已经考虑了线程安全的问题，即已经对共享资源的访问进行了同步处理，不需要 JSP 引擎对当前 JSP 页面翻译成的 Servlet 做线程安全的处理。如果将该属性设置为 false，则表明当前 JSP 页面翻译成的 Servlet 应该实现 SingleThreadModel 接口。关于 SingleThreadModel 接口的作用，请参看 4.3.3 节的讲解。

8. info 属性

info 属性用于将一个文本字符串定义为 JSP 页面翻译成的 Servlet 的描述信息，Web 容器可以通过 Servlet.getServletInfo 方法获得该文本字符串。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

9. errorPage 属性

errorPage 属性用于设置另外一个 JSP 页面来处理当前 JSP 页面发生的异常。如果当前 JSP 页面内产生了未被捕获的异常，则跳转到另外那个 JSP 页面去处理。errorPage 属性的设置值必须使用相对路径，如果以“/”开头，表示相对于当前 Web 应用程序的根目录（注意不是站点根目录），否则，表示相对于当前页面。

可以在 web.xml 文件中使用<error-page>元素为整个 Web 应用程序设置错误处理页面，其中的<exception-type>子元素指定异常类的完全限定名（即含有包名的类名），<location>元素指定以“/”开头的错误处理页面的路径，这里的“/”表示当前应用程序的根目录。如果设置了某个 JSP 页面的 errorPage 属性，那么在 web.xml 文件中设置的异常错误处理将不对该页面起作用。

10. isErrorPage 属性

isErrorPage 属性用于设置当前页面是否是专门用于处理其他页面的未捕获异常的 JSP 页面，其默认值为 false。只有明确地将 page 指令的 isErrorPage 属性设置为 true，当前页面才能被用做异常处理的 JSP 页面，脚本元素中才可以使用 exception 隐式对象。

11. contentType 属性

contentType 属性用于设置响应正文的 MIME 类型（即指定 Content-Type 响应头字段的值）和说明 JSP 文件中的文本内容的字符集编码。contentType 属性的默认 MIME 类型为 text/html，默认字符集为 ISO-8859-1。

对于一个要处理非英文内容的 JSP 页面来说，contentType 是一个非常重要的属性。在 JSP 页面翻译成的 Servlet 源文件中，JSP 引擎会根据 page 指令的 contentType 属性生成相应的调用 ServletResponse.setContentType 方法的语句，所以，contentType 属性具有指定 Servlet 运行时输出给客户端的响应正文的字符集编码的作用。

另外，page 指令的 contentType 属性还具有说明 JSP 源文件的字符编码的作用，但这是在页面中没有通过其他方式指定 JSP 源文件的字符集编码时才起作用的一个次要功能，具体应用请参看 8.7 节的 JSP 的中文乱码问题的讲解。

12. pageEncoding 属性

pageEncoding 属性用于指定 JSP 源文件中的字符所使用的字符集编码。如果设置了 page 指令的 pageEncoding 属性，page 指令的 contentType 属性就不再具有说明 JSP 源文件的字符集编码的作用了。如果在 JSP 页面中没有设置 page 指令的 contentType 属性，那么 page 指令的 pageEncoding 属性还具有指定 Servlet 运行时输出给客户端的响应正文的字符集编码的作用，即让 JSP 引擎在 JSP 页面所翻译成的 Servlet 源文件中生成相应的 response.setContentType 语句。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

13. isELIgnored 属性

isELIgnored 属性用于指定当前 JSP 页面中是否支持 EL（Expression Language）表达式。如果 Web 应用程序使用的是遵循 Servlet 2.3 规范及更低版本的 web.xml 文件，那么该 Web 应用程序内的 JSP 页面的 page 指令的 isELIgnored 属性的默认值为 true，表明 JSP 页面默认不支持 EL 表达式。如果 Web 应用程序使用的是遵循 Servlet 2.4 规范及更高版本的 web.xml 文件，那么该 Web 应用程序内的 JSP 页面的 page 指令的 isELIgnored 属性的默认值为 false，表明 JSP 页面默认支持 EL 表达式。

JSP 页面中的异常处理

01 在 it315 应用程序的 jsp 目录（<tomcat 的安装目录>\webapps\it315\jsp）中按例程 8-18 和例程 8-19 编写一个使用 page 指令的 errorPage 属性和 isErrorPage 属性的例子程序，其中 testError.jsp 文件为发生异常的页面，dealError.jsp 文件则是处理异常的页面。

例程 8-18 testError.jsp

```
<%@ page errorPage="dealError.jsp"%>
<%
    out.println("before exception!");
    int x = 1/0;
    out.println("after exception!");
%>
```

例程 8-19 dealError.jsp

```
<%@ page isErrorPage="true" %>
<%@ page import="java.io.PrintWriter" %>
<%
    out.println("output of dealError!<br>");
    exception.printStackTrace(new PrintWriter(out));
%>
```

02 使用浏览器访问 testError.jsp 页面后，浏览器中显示的结果如图 8.11 所示。

浏览器窗口中显示出的内容为 dealError.jsp 页面的运行结果，testError.jsp 页面向客户端写入的内容没有被传送给浏览器。这是因为 testError.jsp 发生未捕获的异常时，只要它向输出缓冲区中写入的内容还没有被真正传送到客户端，就可以跳转到异常处理页面，并且将输出缓冲区中原来写入的内容清空。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

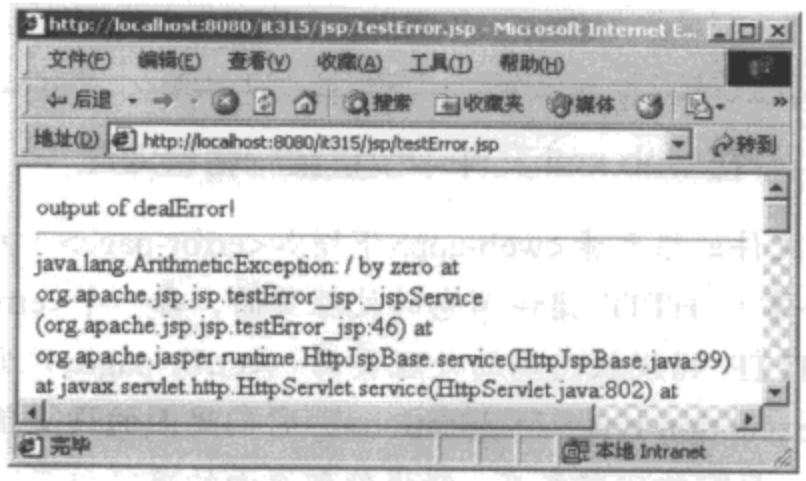


图 8.11

03 修改 testError.jsp 页面，将其中的第一行（即定义错误处理页面的 page 指令）采用 JSP 注释的语法注释掉，然后在浏览器窗口中刷新对 testError.jsp 页面的访问，浏览器中显示出没有进行异常处理时的错误情况。

04 将 dealError.jsp 页面复制成一个名为 dealError2.jsp 的文件，并将其中要输出的内容"output of dealError!
"修改为"output of dealError2!
"，如例程 8-20 所示。

例程 8-20 dealError2.jsp

```
<%@ page isErrorPage="true" %>
<%@ page import="java.io.PrintWriter" %>
<%
    out.println("output of dealError2!<br>");
    exception.printStackTrace(new PrintWriter(out));
%>
```

05 在<tomcat 的安装目录>\webapps\it315\Web-INF\web.xml 文件中的根元素<web-app>中增加一个<error-page>子元素，如下所示：

```
<error-page>
    <exception-type>java.lang.ArithmeticException</exception-type>
    <location>/jsp/dealError2.jsp</location>
</error-page>
```

重新启动 Tomcat，然后在浏览器窗口中刷新对 testError.jsp 页面的访问，浏览器中显示出 dealError2.jsp 页面的运行结果。

06 取消 testError.jsp 页面中的注释，将其恢复成例程 8-18 所示的情况，然后在浏览器窗口中刷新对 testError.jsp 页面的访问，浏览器这时又显示出了 dealError.jsp 页面的运行结果。这说明如果某个 JSP 页面设置了 page 指令的 errorPage 属性，那么在 web.xml 文件中设置的异常错误处理将不对该页面起作用。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

多学两招

在 web.xml 文件中设置全局错误处理

通过在 web.xml 文件的根元素<web-app>下增加<error-page>子元素，就可以为整个 Web 应用程序设置异常和 HTTP 错误消息的处理页面，每一个<error-page>元素用于设置一种异常或一个 HTTP 错误状态码的处理页面。<error-page>元素中定义了三个子元素：<error-code>、<exception-type>和<location>，它们里面的设置值分别是 HTTP 响应消息中的错误状态码、异常的完整类名、错误处理页面的路径位置。下面是<error-page>元素的几种应用举例。

第一种：

```
<error-page>
   <exception-type>java.lang.ArithmeticException</exception-type>
   <location>/jsp/dealError2.jsp</location>
</error-page>
```

第二种：

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

其中，<exception-type>与<error-code>元素是二选一的关系，不能在同一个<error-page>元素中同时出现，<location>元素中设置的错误处理页面的路径必须以“/”开头，表示相对当前 Web 应用程序的根目录。当 Web 容器捕获到<exception-type>元素中设置的异常类型或<error-code>元素中设置的 HTTP 消息响应状态码时，它将把请求转交给<location>元素中设置的错误处理页面。

使用<error-page>元素配置 HTTP 错误消息的处理页面非常有用，例如，某些站点经常要进行内容升级和结构调整，原有的一些资源可能被删除或移动到别的位置，从而导致搜索引擎或其他地方收藏的该站点的一些页面的访问链接变得无效，当用户访问这些无效的页面路径时，Web 容器将发送 404 的 HTTP 错误状态码和找不到页面的错误提示信息；如果站点希望用户不要看到原始的错误状态码信息，而是返回一个友好的相关页面，这时就可以使用<error-page>元素来设置 404 状态码的处理页面。另外，当 Servlet 程序内部发生错误时，Web 容器将返回一个 500 状态码给客户端，用户对这个错误信息根本就不感兴趣，相反，站点管理人员需要及时获得这个错误信息，这时，可以使用<error-page>元素来设置 500 状态码的处理页面，在该处理页面中给用户返回友好的提示信息，并通过某种方式将错误信息通知给管理人员。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

由于 Servlet 的 service 方法只声明抛出 ServletException 和 IOException 异常类型，另外，由于在方法中抛出运行时异常时不必方法定义中显示声明，所以，在 Servlet 的 service 方法中可以抛出如下异常：

- RuntimeException 或 Error 及其子类；
- ServletException 及其子类；
- IOException 及其子类。

如果 Servlet 的 service 方法中发生了除上面这些类型之外的其他异常，service 方法内部必须自行进行捕获处理，否则，Servlet 源程序将无法被成功编译！由于在 Servlet 程序中永远不可能抛出除上面这些类型之外的其他异常，Web 容器也就不可能捕获到其他类型的异常，所以，在<exception-type>元素中只有设置上面这些异常类型才有意义。

如何识别 web.xml 文件所遵循的 Servlet 规范版本

只要打开某个 web.xml 文件，查看其开始部分的模式声明语句，就可以知道该 web.xml 文件所遵循的 Servlet 规范的版本号。

采用 Servlet 2.3 规范的 web.xml 文件的开始部分的内容如下：

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

采用 Servlet 2.4 规范的 web.xml 文件的开始部分的内容如下：

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

如果在 web.xml 文件中没有显示指明其所遵循的 Servlet 规范的版本号，Tomcat 5.x 将默认其遵循 Servlet 2.3 规范。只要在某个 Web 应用程序中编写一个 JSP 页面，且 JSP 页面中仅包含如下一条语句：

```
${"www.it315.org"}
```

然后使用浏览器访问该 JSP 页面。如果浏览器中显示的结果为 www.it315.org，则说明该 Web 应用程序中的 JSP 页面默认支持 EL 表达式，该 Web 应用程序遵循 Servlet 2.4 规范；如果浏览器中显示的结果为${"www.it315.org"}，则说明该 Web 应用程序中的 JSP 页面默认不支持 EL 表达式，该 Web 应用程序遵循 Servlet 2.3 规范。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

8.3.3 include 指令

include 指令用于通知 JSP 引擎在翻译当前 JSP 页面时将其他文件中的内容合并进当前 JSP 页面转换成的 Servlet 源文件中，这种在源文件级别进行引入的方式称之为静态引入。当前 JSP 页面与静态引入的页面紧密结合为一个 Servlet。使用 include 指令的语法如下：

```
<%@ include file="relativeURL"%>
```

其中的 file 属性用于指定被引入文件的相对路径。关于 include 指令的具体应用，需要注意下面的一些问题：

（1）被引入的文件必须遵循 JSP 语法，可以包含静态 HTML、JSP 脚本元素、JSP 指令和 JSP 行为元素等普通 JSP 页面所具有的一切内容。

（2）被引入的文件可以使用任意的扩展名，即使其扩展名是.html，JSP 引擎也会按照处理 jsp 页面的方式处理它里面的内容，为了见名知义，JSP 规范建议使用.jspf（JSP fragments）作为静态引入文件的扩展名。

（3）在将 JSP 文件翻译成 Servlet 源文件时，JSP 引擎将合并被引入的文件与当前 JSP 页面中的指令元素（设置 pageEncoding 属性的 page 指令除外），所以，除了 import 和 pageEncoding 属性之外，page 指令的其他属性不能在这两个页面中有不同的设置值。

（4）除了指令元素之外，被引入的文件中的其他元素都被转换成相应的 Java 源代码，然后插进当前 JSP 页面所翻译成的 Servlet 源文件中，插入位置与 include 指令在当前 JSP 页面中的位置保持一致。

（5）引入文件与被引入文件是在被 JSP 引擎翻译成 Servlet 的过程中进行合并的，而不是先合并源文件后再对合并的结果进行翻译。当前 JSP 页面的源文件与被引入文件的源文件可以采用不同的字符集编码，即使在一个页面中使用 page 指令的 pageEncoding 或 contentType 属性指定了其源文件的字符集为 gb2312，但是，JSP 引擎并不会认为另外一个页面的源文件的字符集也是 gb2312，在另外一个页面中还需要用 page 指令的 pageEncoding 或 contentType 属性指定其源文件所使用的字符集。当这两个页面的源文件采用不同的字符集编码时，由于不能有两个不同的 contentType 属性设置，这时候必须在两个或其中一个页面中使用 pageEncoding 属性来指定其字符集编码，这也是允许在两个页面中有不同的 pageEncoding 属性设置值的原因。

（6）Web 容器在访问 JSP 页面时，可以检测它所引入的其他文件是否发生了修改，如果发生了修改，则重新编译当前 JSP 页面。Tomcat 5.x 中实现了对被引入文件进行检测的功能，具体做法就是在 JSP 页面所翻译成的 Servlet 源文件中增加了如下一条语句：

```
_jspx_dependants.add("/included.jspf");
```

这条语句将被引入文件作为当前 JSP 页面的依赖文件，每次访问 JSP 页面时，JSP 引擎除了检查 JSP 页面本身是否发生了修改，还检查它所依赖的文件是否发生了修改。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

（7）file 属性的设置值必须使用相对路径，如果以“/”开头，则表示相对于当前 Web 应用程序的根目录（注意不是站点根目录），否则，表示相对于当前文件。

注意 这里的 file 属性指定的相对路径是相对于文件（file），而不是相对于页面（page），这也是这个属性被命名为 file 的原因。

假设 myweb 应用程序的根目录下有一个 a.jsp 文件，其一般的访问路径形式为 http://localhost:8080/myweb/a.jsp，在 a.jsp 页面中使用了如下语句引入 b.jspf 文件：

```
<%@ include file="b.jspf"%>
```

如果将 a.jsp 页面映射为 http://localhost:8080/myweb/dir1/a.html 这个路径来访问，由于 include 指令的 file 属性设置值是相对于 a.jsp 文件本身的路径，而不是相对于 http://localhost:8080/myweb/dir1/a.html 这个访问页面的路径，JSP 引擎将调用 a.jsp 文件所在目录（myweb 应用程序的根目录）下的 b.jspf 文件，即 http://localhost:8080/myweb/b.jspf，而不调用 http://localhost:8080/myweb/dir1/b.jspf 这个页面所对应的文件。

include 指令的特性

01 在<tomcat 的安装目录>\webapps\it315\jsp 目录中编写如例程 8-21 和例程 8-22 所示的两个文件，其中 included.jspf 用做被引入的文件，testInclude.jsp 则是引入 included.jspf 的页面。

例程 8-21 included.jspf

```
<%@ page contentType="text/html;charset=GB2312"%>
<%@ page import="java.util.Date" %>
included.jspf 内的中文<br>
```

例程 8-22 testInclude.jsp

```
<%@ page contentType="text/html;charset=gb2312"%>
<%@ include file="included.jspf"%>
testInclude.jsp 内的中文<br>
<%=new Date()%>
```

用浏览器访问 testInclude.jsp 页面，在浏览器中返回一段如下的错误信息：

```
org.apache.jasper.JasperException: /included.jspf(1,1) Page directive: illegal to have multiple occurrences of contentType with different values (old: text/html;charset=gb2312, new: text/html;charset=GB2312)
……
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

其大概意思是 testInclude.jsp 和 included.jspf 两个文件中都设置了 page 指令的 contentType 属性，但两者的设置值不一样而产生了冲突。注意，其中一个是小写的“gb2312”，而另外一个是大写的“GB2312”。

02 将被引入文件 included.jspf 的第一条语句中的大写“GB2312”改为小写“gb2312”，用浏览器重新访问 testInclude.jsp 页面，浏览器中显示的结果如图 8.12 所示。

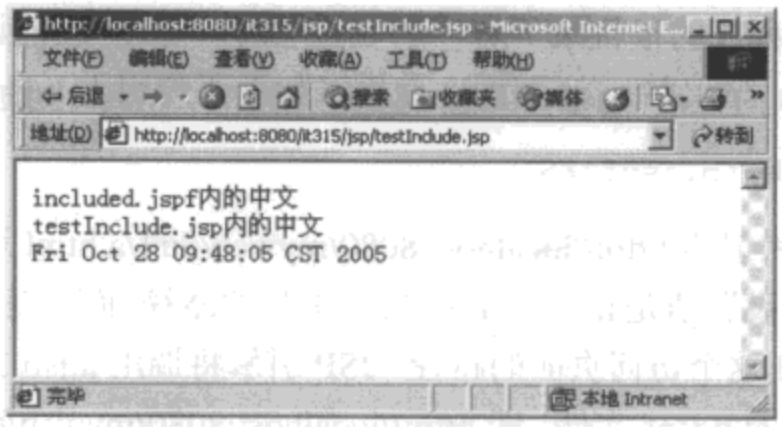


图 8.12

由于被引入文件 included.jspf 中设置的 page 指令的 import 属性被合并到了引入文件 testInclude.jsp 中，所以，即使在 testInclude.jsp 文件中没有编写如下 import 语句：

```
import java.util.Date;
```

但是，testInclude.jsp 文件中的<%=new Date()%>语句却能够顺利通过编译和运行。查看 testInclude.jsp 所翻译成的 Servlet 源文件，可以看到 JSP 引擎处理 include 指令的结果。

03 在<tomcat 的安装目录>\webapps\it315\Web-INF\web.xml 文件中的相应位置处增加如下两段内容：

```
<servlet>
        <servlet-name>TestIncludeJspServlet</servlet-name>
        <jsp-file>/jsp/testInclude.jsp</jsp-file>
</servlet>
……
<servlet-mapping>
        <servlet-name>TestIncludeJspServlet</servlet-name>
        <url-pattern>/xxx/include.html</url-pattern>
</servlet-mapping>
```

重新启动 Tomcat，在浏览器地址栏中输入如下地址：

http://localhost:8080/it315/xxx/include.html

浏览器中返回的结果与直接在浏览器地址栏中输入 http://localhost:8080/it315/jsp/testInclude.jsp 的运行效果一样，这说明 jsp 引擎查找 include 指令引入文件的路径不受访问路径的影响。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

8.4 out 对象

在 JSP 页面中应使用 out 隐式对象来向客户端发送文本形式的实体内容。out 对象是通过调用 pageContext 对象的 getOut 方法返回的，其作用及用法与 ServletResponse.getWriter 方法返回的 PrintWriter 对象非常相似，但两者是两种不同类型的对象。JSP 页面中的 out 隐式对象的类型为 JspWriter，JspWriter 相当于一种带缓存功能的 PrintWriter，设置 JSP 页面的 page 指令的 buffer 属性可以调整它的缓存大小，甚至关闭它的缓存。JSP 页面中的 out 隐式对象相当于插入到 ServletResponse.getWriter 方法返回的 PrintWriter 对象前面的缓冲包装类对象，只有向 out 对象中写入了内容，且满足如下任何一个条件时，out 对象才去调用 ServletResponse.getWriter 方法，并通过该方法返回的 PrintWriter 对象将 out 对象的缓冲区中的内容真正写入到 Servlet 引擎提供的缓冲区中：

- 设置 page 指令的 buffer 属性关闭了 out 对象的缓存功能
- 写入到 out 对象中的内容充满了 out 对象的缓冲区
- 整个 JSP 页面结束

JSP 页面的 out 隐式对象中的缓冲区与 Servlet 引擎提供的缓冲区之间的工作关系如图 8.13 所示。

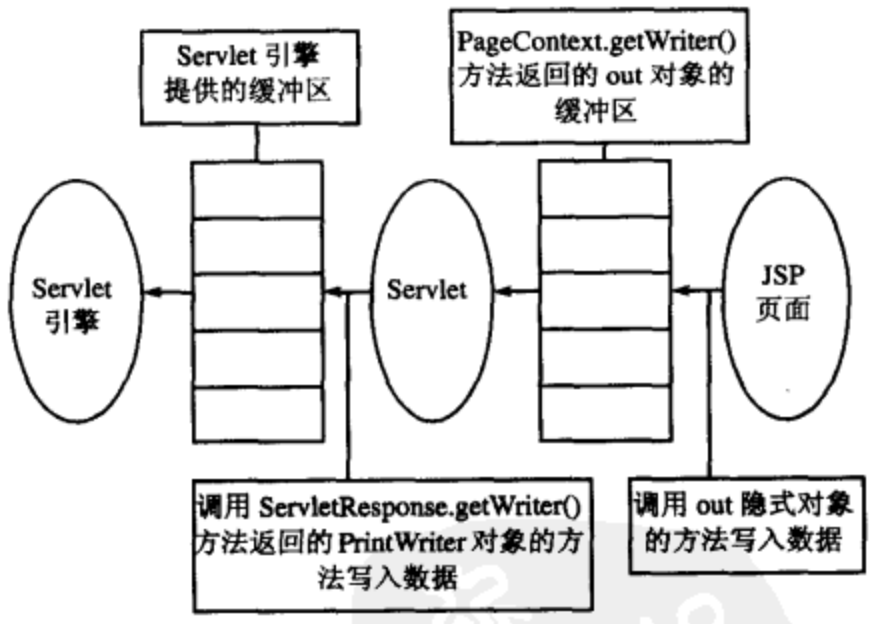


图 8.13

out 对象的应用特性

01 在<tomcat 的安装目录>\webapps\it315\jsp\目录中编写一个名为 jspout.jsp 的 jsp 文件，如例程 8-23 所示。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

例程 8-23 jspout.jsp

```
<%
    out.println("first line<br>");
    response.getWriter().println("second line<br>");
%>
```

用浏览器访问 jspout.jsp 页面，浏览器中显示的结果如图 8.14 所示。

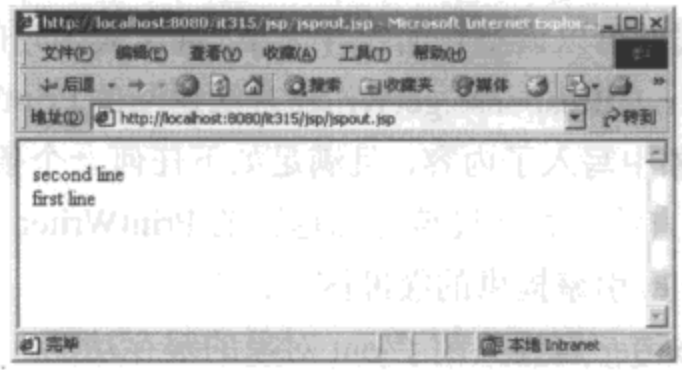


图 8.14

从图 8.14 中可以看出，尽管 out.println 语句位于 response.getWriter().println 语句之前，但它输出的内容却位于后者输出的内容之后。这是因为 out.println 语句只是把内容写入到了 out 对象的缓冲区中，直到整个 JSP 页面结束时，out 对象才把它的缓冲区里面的内容真正写入到 Servlet 引擎提供的缓冲区中，而 response.getWriter().println 语句则是直接把内容写入到了 Servlet 引擎提供的缓冲区中。

02 将 jspout.jsp 文件中的内容修改成如例程 8-24 所示的形式。

例程 8-24 jspout.jsp

```
<%
    ServletOutputStream sos = response.getOutputStream();
    sos.println("www.it315.org");
%>
```

如果在例程 8-24 的 jspout.jsp 文件中的最开始的“<%”前面或最后的“%>”后面输入了任意字符，例如，在最后的“%>”后面输入了一个回车键，用浏览器访问 jspout.jsp 页面，浏览器中显示的结果如图 8.15 所示。

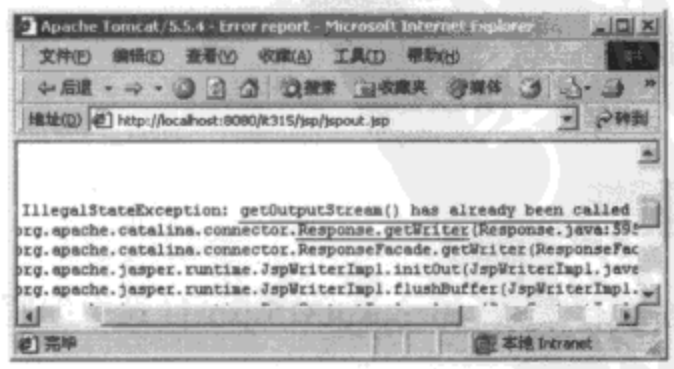


图 8.15

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

位于 JSP 脚本元素之外的任何字符文本，在 JSP 页面所翻译成的 Servlet 源文件中都要被转换成以这些字符文本作为参数的 out.write 语句输出到客户端，即使在 JSP 脚本元素之外输入的是一个回车符，它也将被转换成一条 out.write("\r\n");语句。由于只要向 out 对象中写入了内容，它就会在 JSP 页面结束时去调用 ServletResponse.getWriter 方法，这就与程序中调用的 response.getOutputStream 方法发生了冲突，所以出现了图 8.15 中的异常信息。如果在上面的 jspout.jsp 文件中的最开始的“<%”前面和最后的“%>”后面都没有输入任何字符（包括回车和其他空格字符），用浏览器访问 jspout.jsp 页面可以显示出正常的输出内容，这是因为没有向 out 对象中写入任何内容时，在 JSP 页面结束时将不会调用 ServletResponse.getWriter 方法。可见，只要在脚本片断和指令元素之外没有任何空格和换行字符，那么，在 JSP 页面中还是可以调用 ServletResponse.getOutputStream 方法的。

03 将 jspout.jsp 文件中的内容修改成如例程 8-25 所示的形式。

例程 8-25 jspout.jsp

```
<%
    RequestDispatcher rd = application.getRequestDispatcher("/test.html");
    rd.forward(request,response);
%>
```

其中，test.html 是一个已经存在于<tomcat 的安装目录>\webapps\it315\目录中的静态 HTML 页面。如果在 jspout.jsp 文件中的最开始的“<%”前面或最后的“%>”后面输入了任意字符，例如，还是在最后的“%>”后面输入一个回车键，用浏览器访问 jspout.jsp 页面，浏览器中也将显示出如图 8.15 所示的异常信息。因为客户端对静态 HTML 文件和图片的访问请求都是由 Tomcat 的缺省 Servlet 来处理的，缺省 Servlet 首先检查是否已经调用过当前 HttpServletResponse 对象的 getWriter 方法返回了 PrintWriter 对象，如果已经调用，则使用 getWriter 方法返回的 PrintWriter 对象输出静态 HTML 文件中的内容，否则，缺省 Servlet 调用 getOutputStream 方法返回的 ServletOutputStream 对象来将静态 HTML 文件中的内容按字节流的形式原封不动地输出到客户端。

上面的 jspout.jsp 文件将请求转发给 test.html 页面时，还没有真正调用过 ServletResponse.getWriter 方法，缺省 Servlet 将调用 ServletResponse.getOutputStream 方法，缺省 Servlet 处理完后，程序又回到 jspout.jsp 页面中继续执行，因为 JSP 页面随后又向 out 对象中写入了内容（回车符造成的“\r\n”），所以，在 JSP 页面结束时又将调用 ServletResponse.getWriter 方法，这就造成了冲突。如果在 jspout.jsp 的 JSP 脚本片断中的第一条语句前面增加如下一条语句：

```
response.getWriter();
```

然后使用浏览器重新访问 jspout.jsp 页面，就可以显示出 test.html 页面中的内容，这是因为缺省 Servlet 此时也将调用 ServletResponse.getWriter 方法返回的 PrintWriter 对象来输出 test.html 文件的内容。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

JSP 规范禁止在 JSP 页面中直接调用 ServletResponse.getWriter 返回 PrintWriter 对象和调用 ServletResponse.getOutputStream 返回 ServletOutputStream 对象，就是为了避免发生类似上面的实验中所出现的各种问题。

指点迷津

将请求转发给静态 HTML 页面的实验现象为何不稳定

针对上面的步骤 03 中的实验，只要浏览器在访问 jspout.jsp 页面时曾经成功地获得过 test.html 页面的内容，那么接着再重做步骤 03 的实验时，步骤 03 中所描述的错误现象就不会出现，这是怎么回事呢？这是因为缺省 Servlet 在输出 test.html 页面内容时，也输出了一个 Last-Modified 头字段来说明 test.html 文件最近的修改时间，浏览器缓存了 test.html 页面内容，当它再次访问 jspout.jsp 页面时，将发送 If-Modified-Since 请求头询问服务器是否可以使用缓存，尽管 jspout.jsp 因发生修改会重新编译和运行，但是 jspout.jsp 运行时又把请求转发给了缺省 Servlet，由于 test.html 文件未发生改变，缺省 Servlet 将回复浏览器一个 304（Not Modified）状态码表示可以使用缓存，而不向浏览器回送任何实体内容，也就不会调用 ServletResponse.getOutputStream 方法了。这时，可以采取下面任何一种方式来重现上面的步骤 03 所描述的错误现象：

- 删除 Internet 临时文件（在浏览器的“Internet 选项”对话框的“常规”选项卡中，单击“Internet 临时文件”栏目中的“删除文件”按钮，再单击打开的“删除文件”对话框中的“确定”按钮）；
- 在 test.html 文件中增加一个空格后又删除这个空格，以造成 test.html 文件被修改的假象。

8.5 pageContext 对象

8.5.1 pageContext 对象简介

pageContext 对象是 javax.servlet.jsp.PageContext 类的实例对象，javax.servlet.jsp.PageContext 类是 javax.servlet.jsp.JspContext 的子类。pageContext 对象封装了当前 JSP 页面的运行信息，它提供了返回 JSP 页面的其他隐式对象的方法。pageContext 对象是通过调用 JspFactory.getPageContext 方法返回的，getPageContext 方法的完整语法如下：

```
public PageContext getPageContext(
    javax.servlet.Servlet servlet,
    javax.servlet.ServletRequest request,
    javax.servlet.ServletResponse response,
    java.lang.String errorPageURL, boolean needsSession,
    int buffer, boolean autoflush)
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

其中第一个参数代表当前 JSP 页面所翻译成的 Servlet；第二个参数和第三个参数分别是当前的请求对象和响应对象；第四个参数为错误处理页；第五个参数表示是否创建 Session 对象；第六个参数为 out 对象的缓冲区大小；第七个参数表示缓冲区已满时是否自动刷新。由于当前 JSP 页面所翻译成的 Servlet 对象、当前的请求对象和响应对象都传递给了 getPageContext 方法，所以，getPageContext 返回的 pageContext 对象中封装了 JSP 页面的所有其他的隐式对象。传递给 getPageContext 方法的后四个参数是 JSP 引擎根据 JSP 页面中的 page 指令设置的属性来生成的，因此，pageContext 对象也封装了部分 page 指令的属性设置。

如果一个 JSP 页面调用了一个普通 Java 类，而这个普通 Java 类要访问 JSP 页面中的多个隐式对象，由于 pageContext 对象中封装了其他的隐式对象并提供了返回这些隐式对象的方法，所以在 JSP 页面中只需将 pageContext 对象传递给那个普通 Java 类，在那个普通 Java 类中就可以访问和调用其他隐式对象的功能了。PageContext 类中也提供了一些方法来调用其他隐式对象的方法，以便开发人员可以直接调用 pageContext 对象的这些方法来完成一些常见和重要的功能，而不用先获得隐式对象后再调用隐式对象的相应方法，例如，调用 pageContext.forward()方法即可将请求转发到另外一个资源。

PageContext 类中定义了一个 setAttribute 方法来将对象存储进 pageContext 对象内部的一个 HashMap 对象中，同时也定义了一个 getAttribute 方法来检索存储在该 HashMap 对象中的对象，以便在 JSP 页面与它调用的普通 Java 类之间传递对象信息。PageContext 类除了可以存储和检索自身中的属性对象外，还定义了可以存储和检索其他域范围内的属性对象的方法。

8.5.2 获得其他隐式对象

PageContext 类中定义了如下一些方法来分别返回 JSP 页面中的其他隐式对象：

- getException 方法返回 exception 隐式对象；
- getPage 方法返回 page 隐式对象；
- getRequest 方法返回 request 隐式对象；
- getResponse 方法返回 response 隐式对象；
- getServletConfig 方法返回 config 隐式对象；
- getServletContext 方法返回 application 隐式对象；
- getSession 方法返回 session 隐式对象；
- getOut 方法返回 out 隐式对象。

8.5.3 引入和跳转到其他资源

PageContext 类中定义了一个 forward 方法和两个 include 方法来分别简化和替代 RequestDispatcher.forward 方法和 RequestDispatcher.include 方法的调用，它们的完整定义语法如下：

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

- public void forward(java.lang.String relativeUrlPath)
  throws javax.servlet.ServletException,java.io.IOException
- public void include(java.lang.String relativeUrlPath)
  throws javax.servlet.ServletException,java.io.IOException
- public void include(java.lang.String relativeUrlPath,boolean flush)
  throws javax.servlet.ServletException,java.io.IOException

传递给这些方法的资源路径都只能是相对路径，如果路径以“/”开头，表示相对于当前 Web 应用程序的根目录，否则，表示相对于当前 JSP 所映射到的访问路径。

注意 同一个 JSP 文件可以被映射成各种不同的访问路径，这里的相对路径是相对于当前的访问路径，而不是相对于 JSP 文件的目录路径。

第一个 inlcude 方法在引入资源前先将 out 对象中的内容刷新到 Servlet 引擎提供的缓冲区中；第二个 inlcude 方法则通过一个参数来决定在引入资源前是否刷新 out 对象中的内容。

forward 方法在将请求转发给其他资源之前，先清空 out 对象中已写入的内容，然后调用 RequestDispatcher.forward 方法执行请求转发。

PageContext.forward 方法的应用细节

01 在<tomcat 的安装目录>\webapps\it315\jsp\目录中编写一个名为 pageforward.jsp 的 jsp 文件，如例程 8-26 所示。

例程 8-26 pageforward.jsp

```
<%
    pageContext.forward("/test.html");
%>
```

其中，test.html 为<tomcat 的安装目录>\webapps\it315\目录中的一个已经存在的静态 HTML 页面，确保上面的 pageforward.jsp 文件中的最开始的“<%”前面和最后的“%>”后面都没有输入任何字符（包括回车和其他空格字符），用浏览器访问 pageforward.jsp 页面可以正常显示出 test.html 页面的内容。

02 修改 pageforward.jsp 文件内容，仅仅是在最后的“%>”后面输入一个回车键，再用浏览器访问 pageforward.jsp 页面，由于缓存的原因，浏览器仍然正常显示出了 test.html 页面的内容。删除 Internet 临时文件（在浏览器的“Internet 选项”对话框的“常规”选项卡中，单击“Internet 临时文件”栏目中的“删除文件”按钮，再单击打开的“删除文件”对话框中的“确定”按钮）后，刷新浏览器的访问，浏览器中将显示出如图 8.15 所示的异常信息。关于出现这个错误的原因和为什么要删除 Internet 临时文件后才能显示此错误的

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

原因都已在 8.4 节的“指点迷津”中分析过，这里不再重复讲解。

03 修改 pageforward.jsp 文件内容，删除最后的“%>”后面的回车符，在最开始的“<%”前面输入一个回车键，刷新浏览器的访问，浏览器这次又正常显示出了 test.html 页面的内容。这说明当 JSP 脚本片断的前面有模板内容时，JSP 脚本片断中调用的 PageContext.forward 方法不会出现异常，而在 JSP 脚本片断的后面有模板内容时，JSP 脚本片断中调用的 PageContext.forward 方法会出现如图 8.15 所示的异常，这是为什么呢？因为只要 out 对象的缓冲区中有了内容，它在 JSP 页面结束时就会去调用 ServletResponse.获至宝 getWriter 方法，这就与处理 test.html 页面的缺省 Servlet 中调用的 response.getOutputStream 方法发生了冲突，就会出现图 8.15 中的异常信息。PageContext.forward 方法内部在执行请求转发前，它先清空 out 对象的缓冲区中已写入的内容。JSP 脚本片断前面出现的模板内容在调用 PageContext.forward 方法前已被写入到了 out 对象的缓冲区中，在调用 PageContext.forward 方法时将被作废，从而不会触发 JSP 页面在结束时去调用 ServletResponse.getWriter 方法。但是 JSP 脚本片断后面出现的模板内容等到 PageContext.forward 方法返回后才被写入到 out 对象的缓冲区中，它们将触发 JSP 页面在结束时去调用 ServletResponse.getWriter 方法。如果使用 RequestDispatcher.forward 方法执行请求转发，由于该方法不会去清空 out 对象中的内容，所以，在 JSP 脚本片断的前面或后面有模板内容时都会出现如图 8.15 所示的异常信息。注意，RequestDispatcher. forward 方法执行请求转发时会清空 Servlet 引擎提供的缓冲区中的内容，但这里的问题是由于 JSP 页面提供的 out 对象的缓冲区中有内容造成的，它是在将 out 对象的缓冲区中的内容写入到 Servlet 引擎提供的缓冲区前发生的问题，即问题在 Servlet 引擎提供的缓冲区中还没有内容时就发生了。

8.5.4 访问各个域范围中的属性

在 application，session，request，pageContext 对象中都可以调用 setAttribute 方法和 getAttribute 方法来设置和检索属于各自域范围内的属性，在这 4 个对象中设置的属性对象的作用域是不同的。存储在 application 对象中的属性可以被同一个 Web 应用程序中的所有 Servlet 和 JSP 页面访问；存储在 session 对象中的属性可以被属于同一个会话的所有 Servlet 和 JSP 页面访问；存储在 request 对象中的属性可以被属于同一个请求的所有 Servlet 和 JSP 页面访问，例如，使用 PageContext.forward 和 PageContext.include 方法连接起来的多个 Servlet 和 JSP 页面；存储在 pageContext 对象中的属性仅可以被当前 JSP 页面的当前响应过程中调用的各个组件访问，例如，正在响应当前请求的 JSP 页面和它调用的各个自定义标签类。application、session、request、pageContext 对象中都提供了管理各自域范围内的属性的方法，PageContext 类中还提供了对各个域范围的属性进行统一管理的方法，以简化对各个域范围内属性的访问。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

1. setAttribute 方法

setAttribute 方法有两种重载形式，它们的语法定义如下：

- publicvoid setAttribute(java.lang.String name,java.lang.Object.value)
- public void setAttribute(java.lang.String name,
  java.lang.Object value,int scope)

第一个 setAttribute 方法用于将一个对象与一个名称关联后存储进当前的 PageContext 对象中，这种存储在 PageContext 对象中的对象称为 page 域范围的属性。如果 PageContext 对象中已经存在指定名称的属性，setAttribute 方法先删除原来的属性后再增加新的属性。如果传递给 setAttribute 方法的属性值对象为 null，则删除指定名称的属性，这时的效果等同于 removeAttribute 方法。

第二个 setAttribute 方法用于将一个对象与一个名称关联后存储进 scope 参数指定的域范围中，域范围用 PageContext 类中定义的 4 个整数常量来表示，它们分别是：.APPLICATION_SCOPE，SESSION_SCOPE，REQUEST_SCOPE，PAGE_SCOPE。

2. getAttribute 方法

getAttribute 方法有两种重载形式，它们的语法定义如下：

- public java.lang.Object getAttribute(java.lang.String name)
- public java.lang.Object getAttribute(java.lang.String name,int scope)

第一个 getAttribute 方法用于从当前 PageContext 对象中返回指定名称的属性对象；第二个 getAttribute 方法用于从 scope 参数指定的域范围中返回指定名称的属性对象。

3. removeAttribute 方法

removeAttribute 方法有两种重载形式，它们的语法定义如下：

- public void removeAttribute(java.lang.String name)
- public void removeAttribute(java.lang.String name,int scope)

第一个 removeAttribute 方法用于从当前 PageContext 对象中删除指定名称的属性；第二个 removeAttribute 方法用于从 scope 参数指定的域范围中删除指定名称的属性。

4. getAttributeNamesInScope 方法

getAttributeNamesInScope 方法用于返回某个域范围中的所有属性名，这些属性名将被包含在一个 Enumeration 对象中返回。

5. findAttribute 方法

findAttribute 方法用于依次从 page、request、session、application4 个域范围中查找某个指定名称的属性，找到之后，返回找到的属性对象。如果从这 4 个域范围中都没有找到

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

指定名称的属性，则返回 null。

存储在 pageContext 对象中的属性的作用域

01 在<tomcat 的安装目录>\webapps\it315\jsp\目录中编写一个名为 pagecontext.jsp 的 jsp 文件来验证这个情况，如例程 8-27 所示。

例程 8-27 pagecontext.jsp

```
<%@ page contentType="text/html;charset=GB2312"%>
<%!
    PageContext lastPageContext = null;
    int count = 0;
%>
<%
    if(count != 0)
    {
        if(lastPageContext == pageContext)
        {
            out.println("本次与上次的 pageContext 相同<br>");
        }
        else
        {
            out.println("本次与上次的 pageContext 不相同<br>");
        }

        String x = (String)pageContext.getAttribute("x");
        if("abc".equals(x))
        {
            out.println("本次已获得上次存储在 pageContext 中的属性<br>");
        }
        else
        {
            out.println("本次未获得上次存储在 pageContext 中的属性<br>");
        }

    }
    lastPageContext = pageContext;
    pageContext.setAttribute("x",new String("abc"));
    count++;
%>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

上面的程序定义了两个成员变量：lastPageContext 和 count。lastPageContext 用于记住上次访问 JSP 页面时的 pageContext 对象；count 用于记住 JSP 页面的访问次数。在第一次访问 JSP 页面时将 pageContext 赋值给 lastPageContext，并在 pageContext 中存储一个属性，在第二次访问 JSP 页面时比较 lastPageContext 与当前 pageContext 是否相同，以及能否从当前 pageContext 中取出上次存储的属性。

02 用浏览器访问 pagecontext.jsp 页面，然后刷新浏览器对 pagecontext.jsp 页面进行第二次访问，浏览器中显示的结果如图 8.16 所示。

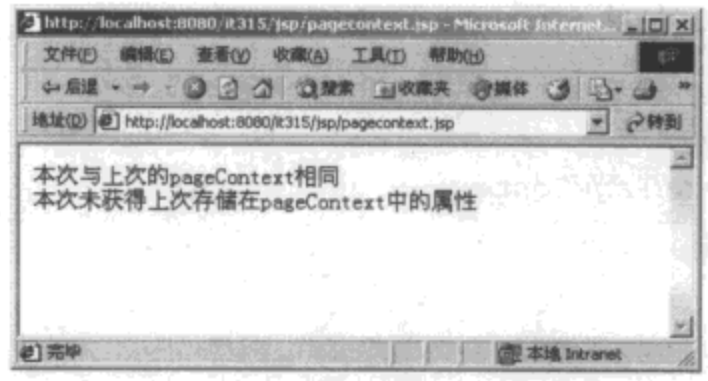


图 8.16

图 8.16 的结果说明同一个 JSP 页面的每次访问请求的响应过程共享同一个 pageContext 对象，但是每次响应过程结束时都将重新初始化 pageContext 对象的内部状态，并释放相关的引用，所以，pageContext 对象中存储的属性对该页面以后请求的响应过程不可见，即同一个 JSP 页面的每次访问请求的响应过程不能共享存储进 pageContext 对象中的属性。

8.5.5 pushBody 方法与 popBody 方法

PageContext 类中有一对方法，即 pushBody 方法和 popBody 方法来管理嵌套的 JspWriter 流，以便支持扩展标签的功能。PageContext 对象内部使用一个“out”属性来指向一个 JspWriter 对象，PageContext.getOut()方法的返回值就是“out”属性所指向的 JspWriter 对象，在第一次调用 pushBody 方法前，“out”属性指向 JSP 页面的隐式 out 对象。pushBody 方法用于返回一个新产生的 BodyContent 对象，并让 PageContext 对象中的“out”属性指向这个新产生的 BodyContent 对象和将“out”属性原来指向的 JspWriter 对象保存起来，此后的 PageContext.getOut()方法返回的结果将是 pushBody 方法内产生的那个 BodyContent 对象。BodyContent 是 JspWriter 的一个子类，它提供了比 JspWriter 更多的功能，例如，将写入到它里面的内容转换成一个字符串、清空写入到它里面的内容，以及将它里面的内容写入到另外一个 JspWriter 对象中。对于每一个 pushBody 方法调用，最后都要调用一个对应的 popBody 方法来恢复 pushBody 方法调用前的状态，popBody 方法将 PageContext 对象中的“out”属性重新指向上次的 pushBody 方法保存起来的 JspWriter 对象，并返回这个 JspWriter 对象。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

JSP 开发人员通常不必调用 pushBody 与 popBody 方法，而是由 JSP 引擎将 JSP 页面翻译成 Servlet 源程序时生成调用这两个方法的代码，但是，了解这两个方法后将有助于理解带标签体的自定义标签的工作方式。下面在<tomcat 的安装目录>\webapps\it315\jsp\目录中编写一个名为 pushBody.jsp 的 JSP 文件，来帮助读者更好地了解 pushBody 与 popBody 方法的作用。

例程 8-28　pushBody.jsp

```
<%@ page contentType="text/html;charset=gb2312"
    import="javax.servlet.jsp.tagext.BodyContent"%>
<%
    JspWriter anotherOut = pageContext.getOut();
    if(anotherOut == out)
    {
        out.println("1. 初始的 pageContext.getOut 方法的返回值" +
            "就是隐式 out 对象。<br>");
    }

    BodyContent out1 = pageContext.pushBody();
    JspWriter out2 = pageContext.getOut();
    if(out1 == out2)
    {
        out.println("2. pageContext.pushBody 方法与它后面调用的" +
            "pageContext.getOut 方法的返回值相同，");
    }
    if(out2 != out)
    {
        out.println("这个返回值不等于隐式 out 对象，");
        out2.println("此时写入到 pageContext.getOut 方法返回的" +
                "JspWriter 对象中的内容不会发送给客户端，但是以后可以" +
                "将这些内容直接写入到另外一个字符输出流对象中，或者" +
                "将这些内容以字符串返回后进行其他方式的处理！<br>");

        /*BodyContent.getEnclosingWriter 方法的返回值就是调用 pushBody
        方法前的 PageContext 对象中的“out”属性指向的 JspWriter 对象，
        此时 out2.getEnclosingWriter 方法的返回值就是隐式 out 对象。
        下面的代码并没有按常理来编写，就是为了让读者更好地理解 out1、
        out2、out2.getEnclosingWriter 方法的返回值、out 三者之间的关系。
        */
        out1.writeOut(((BodyContent)out2).getEnclosingWriter());
        out.println("将上面内容以字符串返回后再输出一遍：" +
                out1.getString());
    }
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
    JspWriter out3 = pageContext.popBody();
    JspWriter out4 = pageContext.getOut();
    if(out3 == out4)
    {
            out.println("3. pageContext.popBody 方法与它后面调用的" +
                "pageContext.getOut 方法的返回值相同，");
    }
    if(out4 == out)
    {
            out.println("这个返回值又等于隐式 out 对象了。<br>");
    }
%>
```

用浏览器访问 pushbody.jsp 页面，浏览器中显示的结果如图 8.17 所示。

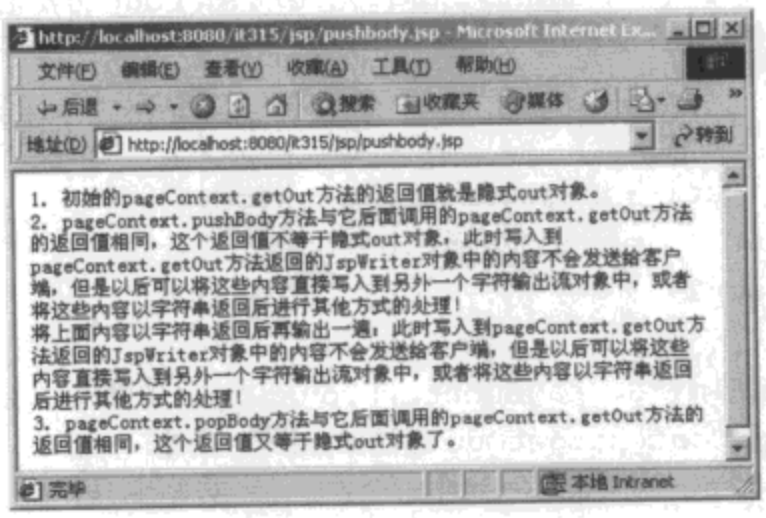


图 8.17

8.6 JSP 标签

JSP 还提供了一种称之为 Action 的元素，在 JSP 页面中使用 Action 元素可以完成各种通用的 JSP 页面功能，也可以实现一些处理复杂业务逻辑的专用功能。Action 元素采用 XML 元素的语法格式，即每个 Action 元素在 JSP 页面中都以 XML 标签的形式出现，所以，有人也干脆将 Action 元素称之为 JSP 标签，笔者本人也习惯使用这一称呼。JSP 规范中定义了一些标准的 Action 元素，这些元素的标签名都以 jsp 作为前缀，并且全部采用小写，例如，<jsp:include>、<jsp:forward>等。此外，JSP 规范也允许编程人员开发自己的 Action 元素，也就是后面章节中要讲解的自定义标签。

下面先介绍一部分标准的 JSP 标签，而其他一些标签将在相应的章节中进行讲解，例如，对于<jsp:useBean>、<jsp:setProperty>、<jsp:getProperty>等用于 JavaBean 的标签，将在第 9 章中结合 JavaBean 实例进行讲解。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

8.6.1 <jsp:include>标签

<jsp:include>标签用于把另外一个资源的输出内容插入进当前 JSP 页面的输出内容之中，这种在 JSP 页面执行时的引入方式称之为动态引入，其语法如下：

```
<jsp:include page="relativeURL | <%=expression%>" flush="true|false" />
```

其中，page 属性用于指定被引入资源的相对路径，它也可以通过执行一个表达式来获得。flush 属性指定在插入其他资源的输出内容时，是否先将当前 JSP 页面的已输出的内容刷新到客户端，其默认值是“false”，表示先不刷新已输出的内容。下面是<jsp:include>标签与 include 指令的比较。

（1）<jsp:include>标签是在当前 JSP 页面的执行期间插入被引入资源的输出内容，当前 JSP 页面与被动态引入的资源是两个彼此独立的执行实体，被动态引入的资源必须是一个能独立被 Web 容器调用和执行的资源。而前面讲解的 include 指令只能引入遵循 JSP 格式的文件，被引入文件与当前 JSP 文件共同合并翻译成一个 Servlet 的源文件。

（2）<jsp:include>标签的作用与运行原理类似 RequestDispatcher.include 方法，被引入页面不能改变响应状态码，也不能设置响应头，与此相关的语句的执行结果将被忽略。include 指令没有这方面的限制。

（3）<jsp:include>标签的执行效率要比 include 指令稍微差一点，但它的灵活性却要好得多。

（4）网页顶部的导航菜单和底部的网站版权信息要重复出现在整个网站的多个页面上，如果在每个网页中直接编写这些相同的 HTML 代码，一旦要修改这些内容，则需要对所有网页都进行修改。使用<jsp:include>标签和 include 指令都可以把一个页面的内容分成多个组件来生成，开发者不必再把页眉和页脚部分的相同 HTML 代码复制到每个 JSP 文件中，从而可以更轻松地完成维护工作。例如，使用下面三个<jsp:include>标签来将导航菜单页、版权信息页和验证用户的 JSP 程序组合成一个完整的用户登录页面：

```
<jsp:include page="top.htm" />
<jsp:include page="admin/login.jsp" />
<jsp:include page="copyright.htm" />
```

无论是使用<jsp:include>标签的动态引入，还是使用 include 指令的静态引入，都应注意最终的输出结果内容应遵循 HTML 语法结构，例如，如果当前页面产生了<html>、</html>、<body>、</body>等标记，那么在被引入文件中就不能再输出<html>、</html>、<body>、</body>等标记。

（5）<jsp:include>标签对 JSP 引擎翻译 JSP 页面的过程没有影响，它是在 JSP 页面的执行期间才被调用，因此不会影响两个页面的编译，例如，一个页面的 page 指令不影响另外一个页面。由于 include 指令是在 JSP 引擎翻译 JSP 页面的过程中被解释处理的，所以它对 JSP 引擎翻译 JSP 页面的过程起作用，如果多个 JSP 页面中都要用到一些相同的声

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

明，那么就可以把这些声明语句放在一个单独的文件中编写，然后在每个 JSP 页面中使用 include 指令将那个文件包含进来。

（6）因为 include 指令引入的文件被合并翻译到当前 JSP 页面中，向被引入的文件传递参数信息的做法根本就没有任何意义，它们将被 JSP 引擎忽略。而<jsp:include>标签可以传递参数信息和接受通过表达式动态产生的被引入资源的名称，所以，如果要用参数来控制引入的结果或动态生成引入的资源名，只能使用<jsp:include>标签。

（7）<jsp:include>标签的 page 属性的设置值必须使用相对路径，如果以“/”开头，表示相对于当前 Web 应用程序的根目录（注意不是站点根目录），否则，表示相对于当前页面。

注意 这里的 page 属性指定的相对路径是相对于页面（page），而不是相对于文件（file），这也是这个属性被命名为 page 的原因。

假设 myweb 应用程序的根目录下有一个 a.jsp 文件，其一般的访问路径形式为 http://localhost:8080/myweb/a.jsp，在 a.jsp 页面中使用了如下语句引入 b.jsp 文件：

```
<jsp:include page="b.jsp" flush="true" />
```

如果将 a.jsp 页面映射为 http://localhost:8080/myweb/dir1/a.html 这个路径来访问，由于<jsp:include>标签的 page 属性设置值是相对当前的访问路径，而不是相对于 a.jsp 文件本身的路径，所以，JSP 引擎将调用 http://localhost:8080/myweb/dir1/b.jsp 页面所对应的资源，而不调用 http://localhost:8080/myweb/b.jsp 这个页面。

<jsp:include>标签的特性

01 在<tomcat 的安装目录>\webapps\it315\jsp 目录中编写如例程 8-29 和例程 8-30 所示的两个文件：included.jsp 和 dynamicInclude.jsp。included.jsp 用做被引入的文件，它暂停 5 秒钟后才输出内容，以便测试<jsp:include>标签的 flush 属性，dynamicInclude.jsp 则是引入 included.jsp 的页面。

例程 8-29 included.jsp

```
<%@ page contentType="text/html;charset=GB2312"%>
<%Thread.sleep(5000);%>
included.jsp 内的中文<br>
```

例程 8-30 dynamicInclude.jsp

```
<%@ page contentType="text/html;charset=gb2312"%>
dynamicInclude.jsp 内的中文<br>
<jsp:include page="included.jsp" flush="true" />
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

用浏览器访问 dynamicInclude.jsp 页面，浏览器首先显示出 dynamicInclude.jsp 页面输出的内容，等待 5 秒钟以后，才又显示出了 included.jsp 页面的输出内容，这说明 Web 容器将当前 JSP 页面的已输出的内容刷新到客户端后才调用被引入的资源。查看 dynamicInclude.jsp 所翻译成的 Servlet 源文件，可以看到<jsp:include>标签被翻译成了一个方法调用。

02 修改 dynamicInclude.jsp 文件，去掉<jsp:include>标签的 flush 属性设置，如例程 8-31 所示。

例程 8-31 dynamicInclude.jsp

```
<%@ page contentType="text/html;charset=gb2312"%>
dynamicInclude.jsp 内的中文<br>
<jsp:include page="included.jsp" />
```

用浏览器重新访问 dynamicInclude.jsp 页面，浏览器等待 5 秒钟以后同时显示出了 dynamicInclude.jsp 和 included.jsp 页面的内容，这说明 Web 容器调用被引入的资源时没有将当前 JSP 页面的已输出的内容刷新到客户端。

03 在<tomcat 的安装目录>\webapps\it315\Web-INF\web.xml 文件中的相应位置处增加如下两段内容：

```
    <servlet>
        <servlet-name>DynamicIncludeJspServlet</servlet-name>
        <jsp-file>/jsp/dynamicInclude.jsp</jsp-file>
    </servlet>
    ……
    <servlet-mapping>
        <servlet-name>DynamicIncludeJspServlet</servlet-name>
        <url-pattern>/xxx/dynamicInclude.html</url-pattern>
    </servlet-mapping>
```

重新启动 Tomcat，在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/xxx/dynamicInclude.html
```

浏览器中返回的结果如图 8.18 所示。

图 8.18

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

图 8.18 中仅显示出了 dynamicInclude.jsp 页面输出的内容，而它调用的 included.jsp 页面输出的内容没有显示出来，这是因为 dynamicInclude.jsp 此时要调用的 included.jsp 页面的完整路径为 http://localhost:8080/it315/xxx/included.jsp，而 Web 容器根本就没找到该路径所对应的资源。

8.6.2 <jsp:forward>标签

<jsp:forward>标签用于把请求转发给另外一个资源，其语法如下：

```
<jsp:forward page="relativeURL | <%=expression%>" />
```

其中，page 属性用于指定请求转发到的资源的相对路径，它也可以通过执行一个表达式来获得。

<jsp:forward>标签的特性

01 在<tomcat 的安装目录>\webapps\it315\jsp 目录中编写一个如例程 8-32 所示的 jspforward.jsp 的文件，它使用<jsp:forward>标签将请求转发给本章刚开始编写的 simple.jsp 页面。

例程 8-32 jspforward.jsp

```
xxx
<jsp:forward page="simple.jsp" />
yyy
```

02 用浏览器访问 jspforward.jsp 页面，浏览器仅显示出了 simple.jsp 页面输出的内容。查看 jspforward.jsp 页面所翻译成的 Servlet 源文件，其中的关键代码如下所示：

```
...
out.write("xxx\r\n");
if (true) {
    _jspx_page_context.forward("simple.jsp");
    return;
}
out.write("\r\n");
out.write("yyy");
...
```

可见，<jsp:forward>标签实际上被翻译成了调用 PageContext.forward 方法的语句，并在调用 PageContext.forward 方法后使用 return 语句结束了 service 方法的执行流程，从而不再执行<jsp:forward>标签后面的语句。如果直接在 JSP 脚本片断中调用 PageContext.forward

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

方法，当脚本片断后面还有模板内容时，将发生如图 8.15 所示的异常。如果在 JSP 页面中使用<jsp:forward>标签，即使<jsp:forward>标签后面还有模板内容，但是由于<jsp:forward>标签可以让 JSP 页面翻译成的 Servlet 程序终止执行其后面的内容，这时就不会出现如图 8.15 所示的异常。

8.6.3 <jsp:param>标签

当使用<jsp:include>和<jsp:forward>标签引入或将请求转发给的资源是一个能动态执行的程序时，例如 Servlet 和 JSP 页面，那么，还可以使用<jsp:param>标签向这个程序传递参数信息。使用<jsp:param>标签传递参数的<jsp:include>标签的语法如下：

```
<jsp:include page="relativeURL | <%=expression%>">
    <jsp:param name="parameterName" value="parameterValue | <%= expression
%>" />
</jsp:include>
```

使用<jsp:param>标签传递参数的<jsp:forward>标签的语法如下：

```
<jsp:forward page="relativeURL | <%=expression%>">
    <jsp:param name="parameterName" value="parameterValue | <%= expression
%>" />
</jsp:include>
```

<jsp:param>标签的 name 属性用于指定参数名，value 属性用于指定参数值。在<jsp:include>和<jsp:forward>标签中可以使用多个<jsp:param>标签来传递多个参数。关于<jsp:param>标签的特性和应用举例，请参见 8.7.3 节的讲解。

8.6.4 <jsp:plugin>标签

<jsp:plugin>标签用来根据浏览器的类型，在当前 JSP 页面产生的网页文档内容中插入调用 Java Applet 的语句，即生成引用 Java Applet 的<OBJECT>或<EMBED>元素内容。其语法如下：

```
<jsp:plugin type="bean | applet" code="classFileName" codebase="classFile
DirectoryName"
        [ name="instanceName" ]
        [ archive="URIToArchive, ..." ]
        [ align="bottom | top | middle | left | right" ]
        [ height="displayPixels" ]
        [ width="displayPixels" ]
        [ hspace="leftRightPixels" ]
        [ vspace="topBottomPixels" ]
        [ jreversion="JREVersionNumber" ]
        [ nspluginurl="URLToPlugin" ]
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        [ iepluginurl="URLToPlugin" ] >
        [ <jsp:params>
        <jsp:param name="parameterName" value="parameterValue | <%= expression
%>" />
        …
        </jsp:params> ]
        [ <jsp:fallback> text message for user </jsp:fallback> ]
    </jsp:plugin>
```

JSP 引擎在翻译 JSP 文件时，根据浏览器的版本和类型将<jsp:plugin>标签转换成 HTML 文档中的引用 Java Applet 的<OBJECT>或<EMBED>元素。当浏览器为支持 HTML4.0 的 IE 浏览器时，<jsp:plugin>标签被转换成<OBJECT>元素；当浏览器为 Netscape Navigator 或仅支持 HTML3.2 的浏览器时，<jsp:plugin>标签被转换成<EMBED>元素。只要掌握了在 HTML 文档中如何使用<applet>标签或<OBJECT>标签来引用 Java Applet 的语法，就会比较容易理解<jsp:plugin>标签的语法。<jsp:plugin>标签中的各个属性的作用如下：

（1）type="bean | applet"

type 属性用于指定插件对象的类型，其设置值只能是 bean 或 applet，这个属性没有默认值，必须进行显示设置。

（2）code="classFileName"

code 属性用于指定浏览器要调用的 Applet 或 Bean 的 Java 类名，必须以.class 结尾。

（3）codebase="classFileDirectoryName"

codebase 属性用于指定 Applet 或 Bean 的 Java Class 文件所在的目录。

（4）jreversion="JREVersionNumber"

jreversion 用于指定运行 Applet 或 Bean 的插件的 Java Runtime Environment（JRE）的版本，默认值是 1.2。

（5）nspluginurl="URLToPlugin"

nspluginurl 用于指定 Netscape Navigator 浏览器下载 JRE 插件的地址，一般情况下不用对此进行设置，它的默认值已经被设置好。

（6）iepluginurl="URLToPlugin"

iepluginurl 用于指定 IE 浏览器下载 JRE 插件的地址，一般情况下不用对此进行设置，它的默认值已经被设置好。

（7）<jsp:params>标签

嵌套在<jsp:plugin>标签中的<jsp:params>和<jsp:param>标签用于向 Applet 或 Bean 传递参数，<jsp:params>标签中可以嵌套多个<jsp:param>子标签来传递多个参数。

（8）<jsp:fallback>标签

<jsp:fallback>标签用于设置浏览器不能正常显示 Applet 或 Bean 时，显示一段替代的文本给用户。

<jsp:plugin>标签的其他一些属性都属于 HTML 规范的<OBJECT>和<EMBED>标记的

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

属性，不属于本书的讲解范围，建议不熟悉 Applet 的读者参看笔者的《Java 就业培训教程》一书。

<Tomcat 5.x 的安装目录>\webapps\jsp-examples\plugin\plugin.jsp 文件就是一个使用<jsp:plugin>标签的例子程序，建议读者查看该 jsp 文件返回到浏览器中的 HTML 内容和该 jsp 文件所生成的 Servlet 源文件，这样有助于更好地理解<jsp:plugin>标签的作用。在 Tomcat 5.x 提供的 jsp-examples 应用程序中，将所有的例子 JSP 页面的访问路径都映射到了各自的一个 Servlet 类上，所以，访问这些 JSP 页面时不用再对 jsp 源文件进行翻译和编译。为了能够看到 plugin.jsp 文件所翻译成的 Servlet 源文件，请在 jsp-examples 应用程序的 web.xml 文件中找到映射/plugin/plugin.jsp 这个 URL 的<servlet-mapping>元素，将该元素注释掉后重新启动 Tomcat 和访问 plugin.jsp。

8.6.5 脚本元素标签

JSP 2.0 规范中定义了一些标签来替代 JSP 页面中的脚本片断、JSP 声明、JSP 表达式，以便这些脚本元素都可以采用 XML 的语法编写，具体的替代方式如下：

- <% code %>可替代为<jsp:scriptlet> code </jsp:scriptlet>
- <%! code %>可替代为<jsp:declaration> code </jsp:declaration>
- <%= expression %>可替代为<jsp:expression> expression </jsp:expression>

8.6.6 指令标签

JSP 2.0 规范中定义了<jsp:directive.directiveType>标签来替代<%@ directive … %>语句，以便采用 XML 的语法格式来定义 JSP 指令。<jsp:directive.directiveType>标签的语法如下：

```
<jsp:directive.directiveType attribute="value" />
```

下面是一些采用 XML 语法定义 JSP 指令的例子：

- <%@ page import="java.util.*"%>可替换为<jsp:directive.page import="java.util.*" />
- <%@ include file="b.jspf" %>可替换为<jsp:directive.include file="b.jspf" />

8.6.7 <jsp:text>标签

JSP 2.0 规范中定义了一个<jsp:text>标签，可以将 JSP 页面中的模板内容封装在这个标签当中，嵌套在<jsp:text>标签中的内容将被作为模板内容输出给客户端。<jsp:text>标签中不能嵌套任何形式的子标签，如果输出给客户端的内容中包含 HTML 标签或 XML 语法规定的特殊字符，应将这些内容封装在 CDATA 区中，或用预定义实体替代特殊字符。

8.6.8 完全使用标签编写 JSP 页面

在 JSP 1.2 规范中就已经定义了<jsp:scriptlet>、<jsp:declaration>、<jsp:expression>、

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

<jsp:directive.directiveType>、<jsp:text>等标签，但该规范只允许在 JSP 文档使用这些标签，而不能在普通的 JSP 页面中使用。为了在普通的 JSP 页面中逐步引入 XML 的语法格式，JSP 2.0 规范允许在普通的 JSP 页面中使用上面这些 JSP 标签，例程 8-33 是一个综合使用了上述标签的普通 JSP 文件。

例程 8-33 jspaction.jsp

```
<jsp:directive.page import="java.util.Date" />
<jsp:declaration>
    int count = 1;
</jsp:declaration>
<jsp:scriptlet>
    String currentTime = new Date().toString();
</jsp:scriptlet>
<jsp:text>
<![CDATA[
<html>
<head>
<title>a simple jsp</title>
</head>
<body>
current time is <i><u>
]]>
</jsp:text>
<jsp:expression>currentTime</jsp:expression>
<jsp:text><![CDATA[
</u></i>
</body>
</html>
]]>
</jsp:text>
```

由于 Tomcat 到了 Tomcat 5.x 这一版本才开始支持 JSP 2.0，所以 jspaction.jsp 文件只能放在 Tomcat 5.x 中运行。

8.7 JSP 的中文乱码问题

8.7.1 JSP 中文乱码的根源

Java 虚拟机中的字符文本的编码采用 Unicode 编码，而输入/输出设备通常都采用其他的字符集编码（例如，中文系统的键盘输入和显示屏输出都采用 GB2312 编码），因此 Java 程序在从外设的字节输入流中读取文本和将文本写入到外设的字节输出流时都需要进行

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

字符编码的转换。另外，Java 程序中的一些字符文本也可能是从 Java 源文件中编译过来的，而 Java 源文件可以采用多种字符集编码来编写，所以，将 Java 源文件编译成 Java 程序时也要进行字符编码转换。由于 JSP 页面最终也是被翻译成 Servlet 程序去运行，而 Servlet 程序就是一个 Java 类，所以，JSP 程序存在着与 Servlet 程序和普通 Java 类完全相同的中文乱码问题：

（1）输出响应正文时出现的中文乱码问题

（2）读取浏览器传递的参数信息时出现的中文乱码问题

（3）读写其他输入/输出设备时出现的中文乱码问题

对于 JSP 来说，还涉及将 JSP 源文件翻译成 Servlet 源文件的过程，而 JSP 源文件与其翻译成的 Servlet 源文件的字符集编码也可能不同，直接在 JSP 源文件中编写的字符串在翻译成 Servlet 源文件时也要发生字符编码转换，所以，JSP 引擎将 JSP 页面翻译成 Servlet 源文件时也可能导致中文乱码问题。

一个 JSP 程序所涉及的字符编码转换过程如图 8.19 所示。

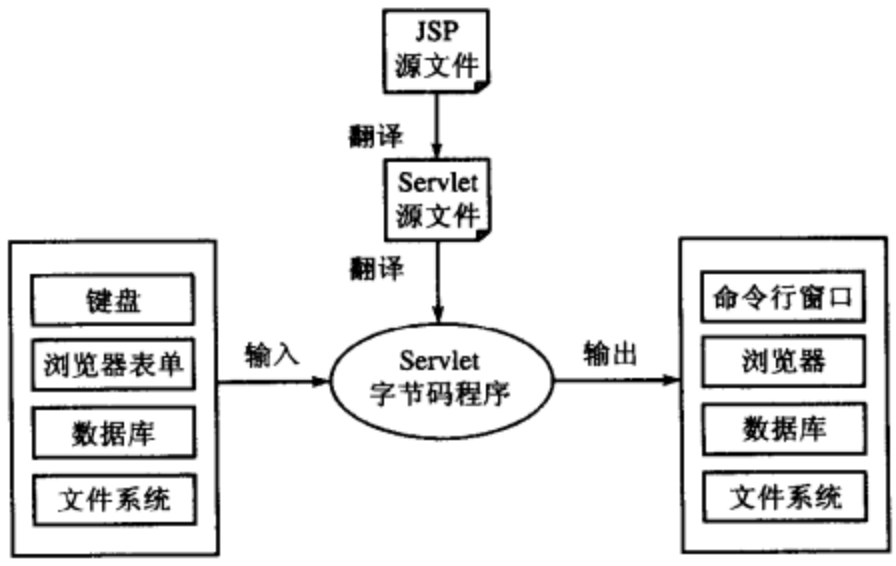


图 8.19

根据图 8.19，可以分析出 JSP 程序中出现中文乱码的各种可能性：

（1）Servlet 程序从请求消息中获取请求参数和从数据库、文件、键盘等外设中读取一个字符串时都要将底层的字节流转换成字符串，但转换过程中指定的字符集编码与外设所输入内容的字符集编码不一致。这里要注意，应用程序从外设中读取字符串的过程通常都是调用第三方提供的 API 来完成的，例如，调用数据库驱动程序 API 来获得一个字符串类型的字段值，但是，这些第三方 API 有可能没有考虑到将底层的字节流转换成字符串时有编码问题，而是统一按操作系统的默认字符集或 ISO8859-1 字符集来进行转换。假设某个第三方 API 将底层设备中的字节流数据总是按 ISO8859-1 转换成字符串返回的，那么，对于底层设备中的 GB2312 编码的中文字符来说，返回的将不是其正确的 Unicode 码，而应用程序编写者又不可能去修改这些底层的 API，这时解决字符乱码的方法就是将底层 API 返回的有问题的字符串按 ISO8859-1 还原成底层设备中的原始字节数据，再将这些字节数据按 GB2312 编码转换成其 Unicode 码的字符串形式，即采用如下语句：

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
strNew = new String(strOld.getBytes("ISO8859-1"),"GB2312");
```

（2）Servlet 程序将字符串输出到浏览器、屏幕、文件和数据库时都要将字符串转换成底层的字节流，但转换过程中指定的字符集编码与外设所能显示的字符集编码不一致。

（3）JSP 引擎将 JSP 源文件翻译成 Servlet 源文件时，其选择的字符集编码与 JSP 源文件实际使用的字符集编码不一致。

（4）JSP 引擎编译由 JSP 源文件翻译成的 Servlet 源文件时，其选择的字符集编码与 Servlet 源文件的字符集编码不一致。不过，JSP 源文件翻译成的 Servlet 源文件的字符集编码是由 JSP 引擎决定的，默认为 UTF-8，而翻译成的 Servlet 源文件也是由 JSP 引擎负责编译的，JSP 引擎显然知道 Servlet 源文件的字符集编码，所以，JSP 引擎在编译 Servlet 源文件时通常不会出现字符编码转换的异常问题。

在通常情况下，当在一个 JSP 页面中出现中文乱码后，可以使用 System.out.println 语句打印出现乱码的字符串。如果在命令行窗口中能看到正常的中文字符，则说明 Java 虚拟机中的字符串本身没有问题，而是在输出给其他设备（例如，浏览器、数据库等）时的字符编码转换过程有问题。否则，就是输入或编译所得到的字符串有问题，应该对输入或编译时的字符编码转换进行分析。如果情况比较复杂，例如在输入和输出时都有问题，则应该跟踪某个中文字符在 JSP 页面运行过程中的每个阶段的编码值，并与它们的正常编码值进行对比，这样就很容易找出问题发生的阶段。

8.7.2 JSP 引擎翻译 JSP 页面时的中文乱码问题

JSP 引擎将 JSP 源文件翻译成的 Servlet 源文件默认采用 UTF-8 编码，而 JSP 开发人员可以采用各种字符集编码来编写 JSP 源文件，因此，JSP 引擎将 JSP 源文件翻译成 Servlet 源文件时，需要进行字符编码转换。JSP 引擎在进行这种转换时，必须要知道 JSP 源文件所使用的字符集编码，才能将源文件中的字符文本转换成其正确的 UTF-8 字符集编码。如果 JSP 文件中没有说明它采用的字符集编码，JSP 引擎将把它当做默认的 ISO8859-1 字符集编码处理，当 JSP 源文件中有中文字符时，这些中文字符按 ISO8859-1 字符集编码转换成 UTF-8 字符集编码，结果就肯定不正确了。

page 指令的 contentType 属性除了具有指定响应正文的字符集编码的作用外，它还具有说明 JSP 源文件的字符集编码的作用。JSP 规范建议尽量不要使用 page 指令的 contentType 属性来说明 JSP 源文件的字符集编码，而是使用 page 指令的 pageEncoding 属性或在部署描述符中说明 JSP 源文件的字符集编码。如果设置了 page 指令的 pageEncoding 属性，page 指令的 contentType 属性就不再具有说明 JSP 源文件的字符集编码的作用了。如果一个 Web 应用程序或其某个目录中的所有 JSP 源文件使用的字符集编码完全相同，且不是 ISO8859-1，那么在每个 JSP 源文件中都使用 page 指令的 pageEncoding 属性来说明它的字符集编码是非常麻烦的事情。JSP2.0 规范专门为此提供了一个解决方案，即在部署描述符文件中说明一组 JSP 文件的字符集编码。假设/it315 这个 Web 应用程序的 jsp 目录中的所有 JSP 文件的字符集编码都为 GB2312，那么在/it315 这个 Web 应用程序的部署描述

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

符文件中就可以使用如下一段语句进行设置：

```
<jsp-config>
    <jsp-property-group>
            <url-pattern>/jsp/*</url-pattern>
            <page-encoding>GB2312</page-encoding>
    </jsp-property-group>
</jsp-config>
```

当 JSP 引擎要确定某个 JSP 文件的字符集编码时，首先在 Web 应用程序的部署描述符中查找是否有与该 JSP 文件匹配的<page-encoding>元素设置，如果没有找到则在 JSP 源文件中查找 page 指令的 pageEncoding 属性。如果没有找到<page-encoding>元素设置和 page 指令的 pageEncoding 属性，JSP 引擎接着在 page 指令的 contentType 属性中查找字符集编码，如果还没找到则使用默认的 ISO8859-1 作为 JSP 源文件的字符集编码。可以同时设置<page-encoding>元素和 pageEncoding 属性，但两者的设置值必须相同。

尽管 page 指令的 contentType 属性和 pageEncoding 属性都可以说明 JSP 源文件的字符集编码，但是，JSP 引擎必须只有读取到这样的 page 指令后才能知道 JSP 源文件的字符集编码，在读取到这些 page 指令之前，JSP 引擎是无法知道 JSP 源文件的字符集编码的。由于英文 ASCII 码字符在 UTF-8 编码和各个国家的本地字符集编码中的值完全一样，所以 JSP 引擎总是先按 UTF-8 编码来读取 JSP 源文件中的内容，这就要求设置 page 指令的 contentType 属性和 pageEncoding 属性的语句之前不能有非 UTF-8 编码的字符，最好是在 JSP 源文件的开始处设置 page 指令的 contentType 属性和 pageEncoding 属性。由于英文 ASCII 码字符的 UTF-16 编码和 EBCDIC 编码与它们的 UTF-8 编码并不相同（不只占用一个字节），所以，JSP 源文件也不能使用 UTF-16 编码和 EBCDIC 编码。

JSP 引擎翻译 JSP 页面时的中文乱码问题

01 在<tomcat 的安装目录>\webapps\it315\jsp\目录中编写一个名为 chinese.jsp 的 JSP 文件，如例程 8-34 所示。

例程 8-34　chinese.jsp

```
<%
    out.println("hello 中国");
%>
```

02 使用浏览器访问 chinese.jsp 页面，浏览器中显示出如图 8.20 所示的中文乱码。

单击浏览器的“查看”→“编码”→“简体中文（GB2312）”菜单，浏览器中显示出了正常的中文字符。这说明 Web 服务器端输出给浏览器的实体内容确实是中文字符的

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

GB2312 编码，但是浏览器却以 ISO8859-1 字符集编码显示它们，所以导致了如图 8.20 所示的乱码现象。

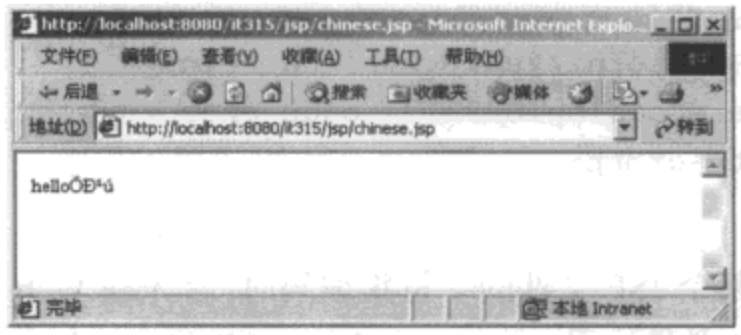


图 8.20

03 修改 chinese.jsp 文件，使用<meta>标签让浏览器采用 GB2312 字符集编码来显示其接收到的实体内容，如例程 8-35 所示。

例程 8-35 chinese.jsp

```
<meta http-equiv="Content-Type" content="text/html;charset=GB2312" />
<%
    out.println("hello 中国");
%>
```

刷新浏览器对 chinese.jsp 页面的访问，浏览器中显示的效果与第 **02** 步中看到的效果完全一样。查看网页文档的源文件，可以看到浏览器确实接收到了新增的<meta>标签语句，奇怪的是，这条<meta>标签语句怎么没起作用呢？

04 在运行 Tomcat 的本地计算机上的命令行窗口中执行 telnet 127.0.0.1 8080 命令，在成功连接后启动的 Telnet 程序窗口中，输入如下两行内容：

```
GET /it315/jsp/chinese.jsp HTTP/1.1
Host:
```

接着按回车键输入一个空行后，Telnet 程序窗口中显示出 Web 服务器返回的响应信息，如图 8.21 所示。

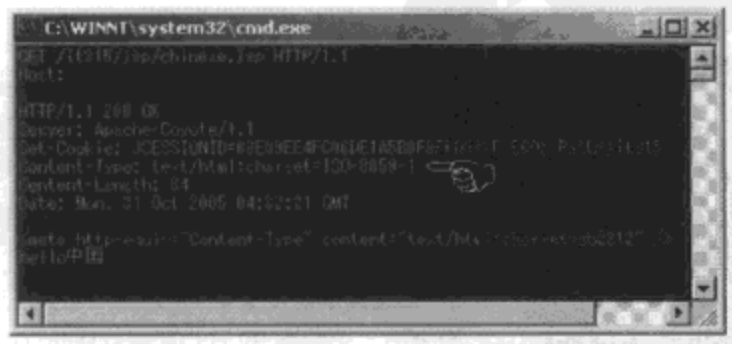


图 8.21

从图 8.21 中可以看到，Web 服务器发送的响应消息中设置了 Content-Type 头字段的字

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

符集编码为“ISO8859-1”，实体内容的开始部分也使用了<meta>标签指定当前页面的字符集编码为“GB2312”。根据第 **03** 步的实验现象可以推测出 Web 服务器实际发送的响应头字段比实体内容中用<meta>标签模拟的响应头字段的优先权高，浏览器优先采用 Web 服务器实际发送的响应头字段，所以浏览器采用 ISO8859-1 字符集编码来显示它所接收到的实体内容。

05 使用 UltraEdit 程序打开 chinese.jsp 所翻译成的 Servlet 源文件，然后切换到十六进制下查看，找到与 JSP 源文件中的“中国”这两个字符所对应内容，如图 8.22 所示。

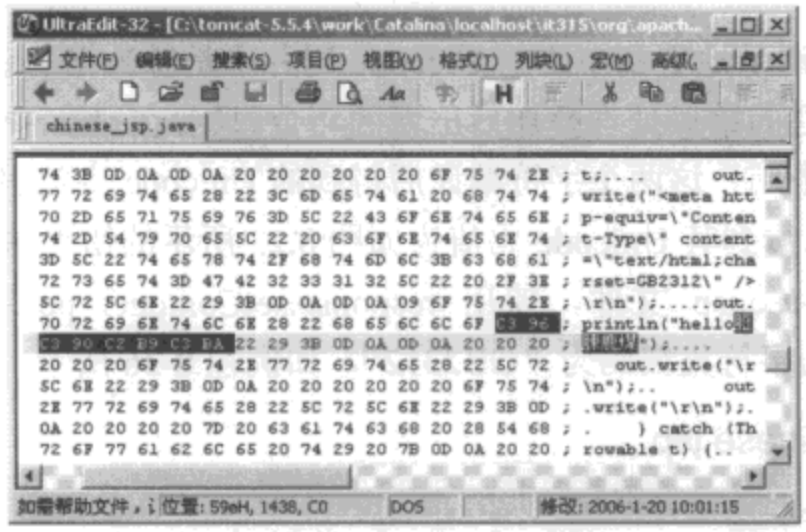


图 8.22

从图 8.22 中可以看到，JSP 源文件中的“中国”这两个字符在翻译成的 Servlet 源文件中占了 8 个字节，而一个中文字符的 UTF-8 编码应该是三个字节，“中国”这两个字符的正确 UTF-8 编码应该是 6 个字节，这里显然出现了问题。下面分析一下其中的原因：上面的 chinese.jsp 文件实际上采用的是 GB2312 编码，但是它没有说明自己所使用的字符集编码，JSP 引擎把它当做了默认的 ISO8859-1 字符集编码处理；“中国”这两个字符的 GB2312 码为[D6][D0][B9][FA]，当把它们按 ISO8859-1 字符集编码转换成的 Unicode 编码时，得到的结果为[00D6][00D0][00B9][00FA]，这 4 个 Unicode 码值都位于 0080 至 07ff 之间，它们对应的 UTF-8 编码都是两个字节，所以最终得到的 UTF-8 编码就占了 8 个字节。

既然 JSP 源文件中的中文字符在翻译成 Servlet 源文件时已经出现了问题，那为什么在第 **04** 步中看到服务器发送给浏览器的实体内容又是正常的中文字符呢？这是由于 JSP 引擎将中文字符按 ISO8859-1 字符集编码转换成的 Unicode 编码（Unicode 编码与 UTF-8 编码是一一映射的关系，这里可以认为两者等同）时，虽然会出现问题，但其转换过程是一种可逆的运算，没有发生信息损失。由于 chinese.jsp 文件中没有指定以何种字符集编码输出响应正文，Servlet 引擎将采用默认的 ISO8859-1 字符集编码进行输出，所以在 chinese.jsp 文件翻译成的 Servlet 程序运行时又将这些不正确的 Unicode 编码转换成 iso8859-1 字符集编码输出给浏览器，这就等于将中文字符又恢复成了它们在 chinese.jsp 源文件中的 GB2312 编码状态，这个转换过程如图 8.23 所示。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

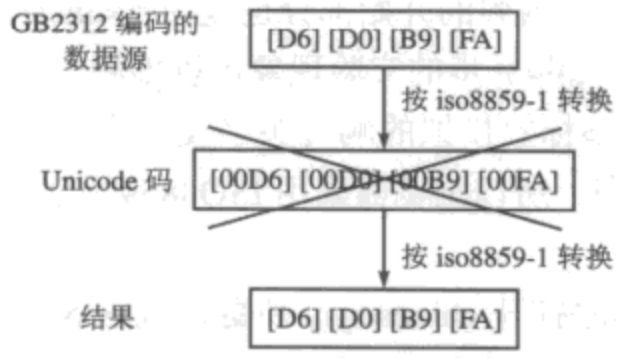


图 8.23

从图 8.23 中可以清楚地看到中间的 Unicode 码并不是源数据的正确 Unicode 码，最后的输出结果是两次相逆的错误处理叠加后出现的巧合。只要在 chinese.jsp 中增加一些代码，打印出源文件中的“中国”这两个字符编译后得到的 Unicode 字符串的长度和每个字符的 Unicode 码值，就可以看到其 Unicode 码值为如图 8.23 所示的错误情况。

06 修改 chinese.jsp 文件，使用 response.setContentType 方法指定响应消息的 Content-Type 头字段的字符集编码为“GB2312”，如例程 8-36 所示。

例程 8-36 chinese.jsp

```
<%
    response.setContentType("text/html;charset=GB2312");
    out.println("hello 中国");
%>
```

刷新浏览器对 chinese.jsp 页面的访问，浏览器中显示的结果如图 8.24 所示。

图 8.24 中的中文字符仍然是乱码，单击浏览器的“查看”→“编码”菜单，可以看到浏览器当前查看网页文档的字符集编码确实已经被设置成了“简体中文（GB2312）”。重复第 **04** 步操作，Telnet 程序窗口中显示的响应信息如图 8.25 所示。

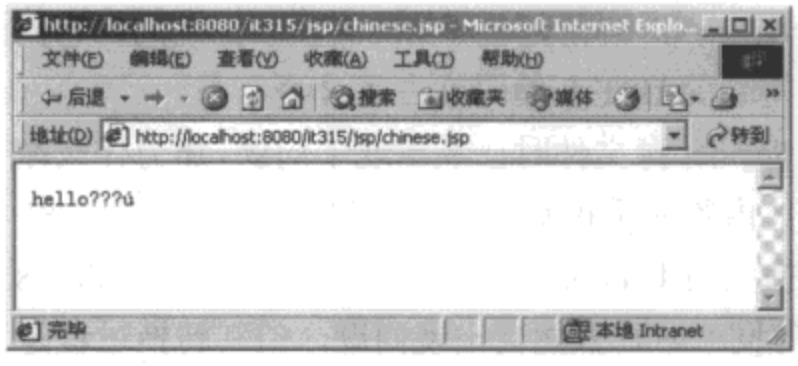


图 8.24

图 8.25

Web 服务器发送的响应消息中已将 Content-Type 头字段的字符集编码设置为了“GB2312”，但 Web 服务器输出的实体内容中的中文字符却不是正确的 GB2312 编码。这是由于 JSP 源文件中的中文字符在翻译成的 Servlet 源文件中已经不是其正确的 Unicode 编码了，当 chinese.jsp 文件中指定以 GB2312 字符集编码输出响应正文时，不正确的 Unicode

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

编码按 GB2312 字符集编码进行转换的结果当然不会是正确的 GB2312 编码。

07 修改 chinese.jsp 文件，设置 page 指令的 Content-Type 属性中的字符集编码为“GB2312”，如例程 8-37 所示。

例程 8-37 chinese.jsp

```
<%@ page contentType="text/html;charset=gb2312"%>
<%
    out.println("hello 中国");
%>
```

刷新浏览器对 chinese.jsp 页面的访问，这时浏览器中显示出了正常的中文字符，可见，经常听到的“在 JSP 页面中设置 page 指令的 contentType 属性与直接调用 response.setContentType 方法的作用相同”的说法是错误的。page 指令的 contentType 属性除了具有 ServletResponse.setContentType 方法的作用外，它还具有说明 JSP 源文件的字符集编码的作用。使用 UltraEdit 程序重新打开 chinese.jsp 所翻译成的 Servlet 源文件，然后切换到十六进制下查看，找到与 JSP 源文件中的“中国”这两个字符所对应的内容，如图 8.26 所示，JSP 源文件中的“中国”这两个字符在翻译成的 Servlet 源文件中占了 6 个字节，并且这 6 个字节的内容确实是“中国”这两个字符的 UTF-8 编码。

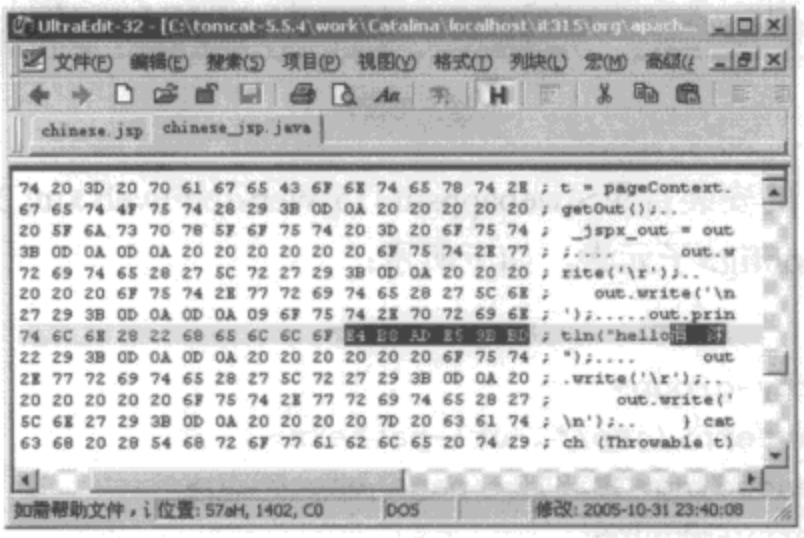


图 8.26

08 如果采用 UTF-8 字符编码保存 chinese.jsp 文件内容，但 chinese.jsp 文件输出到客户端的实体内容要求采用 GB2312 编码，那该怎么做呢？显然，这时候无法采用 page 指令的 contentType 属性来同时指定输出的实体内容的字符集编码，以及说明 JSP 源文件的字符集编码，而是必须使用 page 指令的 pageEncoding 属性来说明 JSP 源文件的字符集编码，如例程 8-38 所示。

例程 8-38 chinese.jsp

```
<%@ page pageEncoding="utf-8"%>
<%@ page contentType="text/html;charset=gb2312"%>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
<%
    out.println("hello 中国");
%>
```

先让 chinese.jsp 文件的字符集编码仍然保持为 GB2312，刷新浏览器对 chinese.jsp 页面的访问，这时浏览器显示的中文字符为乱码。用记事本程序将 chinese.jsp 文件另存为 UTF-8 编码格式，然后再刷新浏览器对 chinese.jsp 页面的访问，这时浏览器显示出了正常的中文字符。

09 将 chinese.jsp 文件改回为 GB2312 编码保存，然后在 chinese.jsp 文件的开头处增加一个 GB2312 编码的中文字符，刷新浏览器对 chinese.jsp 页面的访问，这时候 JSP 引擎报告出如图 8.27 所示的错误。

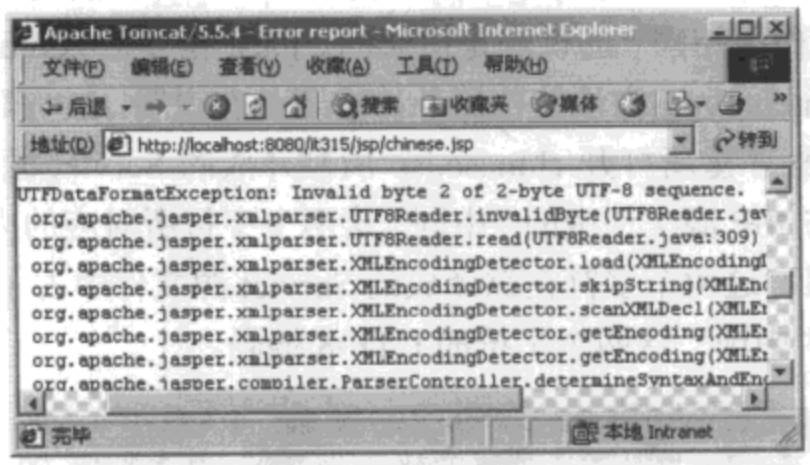


图 8.27

10 在<tomcat 的安装目录>\webapps\it315\Web-INF\web.xml 文件中的根元素<web-app>中增加一个<jsp-config>子元素，如下所示：

```
<jsp-config>
    <jsp-property-group>
        <url-pattern>/jsp/*</url-pattern>
        <page-encoding>GB2312</page-encoding>
    </jsp-property-group>
</jsp-config>
```

由于<tomcat 的安装目录>\webapps\it315\Web-INF\web.xml 文件中没有声明它所遵循的 Servlet 规范版本，Tomcat 默认它遵循 Servlet 2.3 规范，而 jsp-config、jsp-property-group 和 page-encoding 等元素是在 Servlet 2.4 规范的部署描述符文件中新定义的元素，所以，上面增加的<jsp-config>元素及其中的内容不会生效。将根元素<web-app>的开始标签修改成如下形式，以声明 web.xml 文件遵循 Servlet 2.4 规范：

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
                    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
```

虽然上面这段内容比较复杂，但是只需要找到一个现成的遵循 Servlet 2.4 规范的 web.xml 文件，将它复制进来就可以了，没必要死记硬背和参照书本输入，作者编写上面这段内容时，就是从<Tomcat 5.x 的安装目录>\webapps\jsp-examples\Web-INF\web.xml 文件中复制过来的。参看 web.xml 文件的 Schema 文档，即查看文件 http://java.sun.com/ xml/ns/j2ee/web-app_2_4.xsd，可以看到<web-abb>元素中的各个子元素并没有严格的先后顺序，所以，上面编写的<jsp-config>元素只要作为<web-app>元素的子元素出现即可，其位置没有严格规定。

修改<tomcat 的安装目录>\webapps\it315\Web-INF\web.xml 文件后，重新启动 Tomcat。接着将 chinese.jsp 文件修改成如例程 8-39 所示的形式。

例程 8-39　chinese.jsp

```
<%@ page contentType="text/html;charset= utf-8"%>
<%
    out.println("hello 中国");
%>
```

先用字符集编码 GB2312 保存 chinese.jsp 文件，刷新浏览器对 chinese.jsp 页面的访问，这时浏览器中显示出了正常的中文字符。用记事本程序将 chinese.jsp 文件另存为 UTF-8 编码，然后再刷新浏览器对 chinese.jsp 页面的访问，这时浏览器中显示出的中文字符为乱码。这个结果说明部署描述符文件中的<page-encoding>元素确实起到了说明 chinese.jsp 文件的字符集编码的作用，并且其优先权比 page 指令的 contentType 属性要高。

8.7.3　静态引入时的中文乱码问题

静态引入时的中文乱码问题

01 为了不受 8.7.1 节的实验环境的影响，将 8.3.3 节的 testInclude.jsp 文件和 included.jspf 文件复制到本机上的 Tomcat 站点的根目录中，然后将 included.jspf 文件中的第一条语句删除，即删除设置 page 指令的 contentType 属性的那条语句，如例程 8-40 和例程 8-41 所示。

例程 8-40　included.jspf

```
<%@ page import="java.util.Date" %>
included.jspf 内的中文<br>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

例程 8-41 testInclude.jsp

```
<%@ page contentType="text/html;charset=gb2312"%>
<%@ include file="included.jspf"%>
testInclude.jsp 内的中文<br>
<%=new Date()%>
```

用浏览器访问 testInclude.jsp 页面，显示结果如图 8.28 所示，被引入文件 include.jspf 输出的中文部分为乱码。

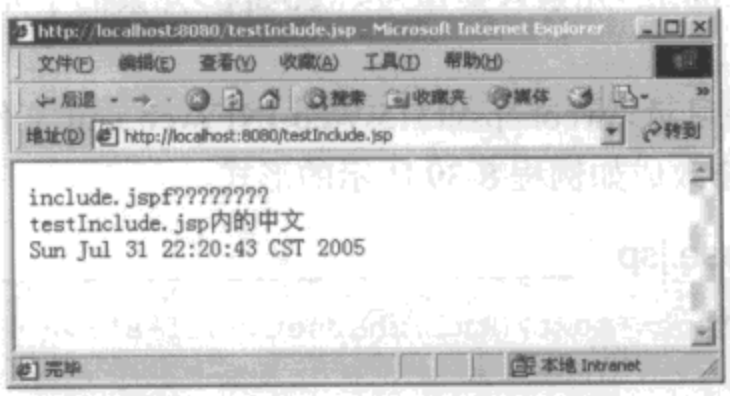


图 8.28

02 JSP 引擎将 JSP 源文件翻译成的 Servlet 源文件默认采用 UTF-8 编码，由于 UltraEdit 不能正常显示出 UTF-8 编码的中文内容，所以用记事本程序打开 testInclude.jsp 文件所翻译成的 Servlet 源文件，看到其中的关键代码如下：

```
……
out.write("\r\n");
out.write("include.jspfÃÚµÄÕÐÎÄ<br>\r\n");
out.write("\r\n");
out.write("testInclude.jsp 内的中文<br>\r\n");
out.print(new Date());
out.write("\r\n");
……
```

在上面的程序片断中，可以清楚地看到 included.jspf 文件内的中文没有被正确地转换成其 UTF-8 编码。这是由于在 included.jspf 文件中没有采用任何方式指定源文件的字符集，而 testInclude.jsp 文件中的 page 指令的 contentType 属性也不能说明 included.jspf 文件的字符集编码，JSP 引擎把 included.jspf 源文件当做默认的 ISO-8859-1 字符集编码进行翻译处理，其中的 GB2312 编码的中文字符就被转换成了错误的 UTF-8 编码。

03 在被引入文件 included.jspf 的开始处增加如下两条 page 指令语句中的任何一条：

- `<%@ page contentType="text/html;charset=gb2312"%>`
- `<%@ page pageEncoding="gb2312"%>`

都可以解决上面的中文乱码问题。如果 testInclude.jsp 和 included.jspf 文件采用的字符

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

集编码不同，由于两个源文件中设置的 page 指令的 contentType 属性必须相同，这时候就不能在两个源文件中都使用 page 指令的 contentType 属性来说明各自的字符集编码了，必须在其中一个或两个文件中使用 page 指令的 pageEncoding 属性来指定源文件的字符集编码。

8.7.4 <jsp:param>标签传递中文参数时的问题

<jsp:param>标签传递中文参数时的问题

01 在<tomcat 的安装目录>\webapps\it315\jsp 目录中编写如例程 8-42 和例程 8-43 所示的两个文件：jspparam.jsp 和 getparam.jsp。jspparam.jsp 文件内使用<jsp:forward>标签将请求转发给 getparam.jsp 页面，在转发的 url 后面直接增加一个参数，同时也使用<jsp:param>标签传递一个参数，getparam.jsp 则获取并显示传递的参数。

例程 8-42 jspparam.jsp

```
<%@ page pageEncoding="GB2312"%>
<jsp:forward page="getparam.jsp?country1=中国">
    <jsp:param name="country2" value="中国" />
</jsp:forward>
```

例程 8-43 getparam.jsp

```
<%@ page contentType="text/html;charset=gb2312"%>
<%
    String country1 = request.getParameter("country1");
    String country2 = request.getParameter("country2");
%>
country1:<%= country1 %><br>
country2:<%= country2 %><br>
```

因为 jspparam.jsp 源文件中直接编写有中文字符，为确保 JSP 引擎将它翻译成 Servlet 源文件时不会发生编码转换错误，在 jspparam.jsp 源文件中要设置 page 指令的 pageEncoding 属性为“GB2312”。其实，在 8.7.1 节的实验中已经在部署描述符文件中设置了<page-encoding>元素值为“GB2312”，即使这里不设置 pageEncoding 属性也不会有什么问题。因为 getparam.jsp 页面要输出中文内容（查询参数信息时获得），所以要设置 page 指令的 contentType 属性的字符集编码为“GB2312”或“UTF-8”，这里选择了“GB2312”。虽然程序中考虑了上面这些可能造成中文乱码的因素，但是用浏览器访问 jspparam.jsp 页面，浏览器中仍然显示出如图 8.29 所示的乱码现象。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

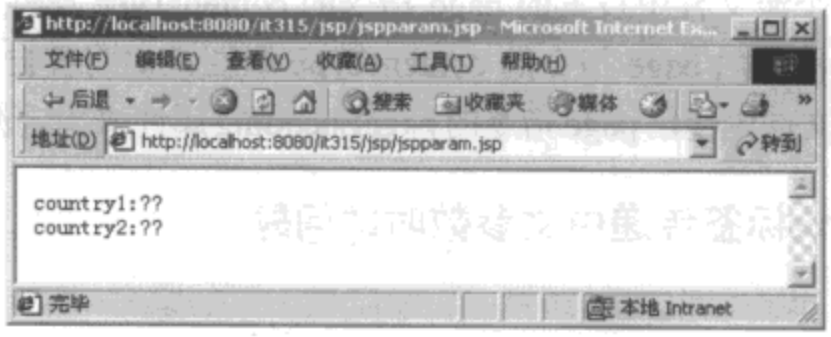


图 8.29

02 用记事本程序打开 jspparam.jsp 文件所翻译成的 Servlet 源文件，看到其中的关键代码如下：

```
...
_jspx_page_context.forward("getparam.jsp?country1=中国" +
    (("getparam.jsp?country1=中国").indexOf('?')>0? '&': '?') +
org.apache.jasper.runtime.JspRuntimeLibrary.URLEncode("country2",
    request.getCharacterEncoding())+ "=" +
    org.apache.jasper.runtime.JspRuntimeLibrary.URLEncode("中国",
                                    request.getCharacterEncoding()));
...
```

从上面的代码中可以看到，直接增加在<jsp:forward>标签转发的 URL 地址后面的参数在被翻译成的 Serlvet 源文件中保持不变，而使用<jsp:param>标签增加的参数被进行了 URL 编码处理，显然，当要向<jsp:include>和<jsp:forward>标签引入或将请求转发给的资源传递参数时，最好是使用<jsp:param>标签，而不要直接将参数增加到资源路径的后面。从上面的代码中还可以看到，在对<jsp:param>标签传递的参数进行 URL 编码时，JSP 引擎选择的字符集编码为 request.getCharacterEncoding()方法的返回值，但该返回值为 null（只要在 japparam.jsp 中增加一条打印出该值的语句就可以验证），所以，必须先解决这个问题才能继续分析图 8.29 中显示的乱码原因。

03 修改 jspparam.jsp 文件，设置请求消息对象的字符集编码为“GB2312”，如例程 8-44 所示。

例程 8-44 jspparam.jsp

```
<%@ page pageEncoding="GB2312"%>
<%request.setCharacterEncoding("GB2312");%>
<jsp:forward page="getparam.jsp?country1=中国">
    <jsp:param name="country2" value="中国" />
</jsp:forward>
```

刷新浏览器对 jspparam.jsp 页面的访问，如果读者已经按照 6.9.5 节的第 **06** 步操作在 server.xml 文件中设置了<Connector>元素的 useBodyEncodingForURI 属性为 true，那么

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

浏览器中将显示出正常的中文字符。只要明白了 JSP 引擎翻译<jsp:param>标签的细节后，对于其中的中文乱码问题，则可以按照 6.9.5 节的讲解进行仔细分析和解决。

8.8 思考与实践

友情提示：请认真地完成以下思考与实践，并将完成的结果以及操作步骤提交给 www.it315.org 站点评审，表现优异者将获得 www.it315.org 站点的奖励和专家的特别指导。

1．请问：JSP 引擎会处理下面代码中的 include 指令吗？included.jspf 文件中的指令元素会影响当前 JSP 页面的编译吗？included.jspf 文件中的内容会输出到客户端吗？

```
<%if(false){%>
<%@ include file="included.jspf"%>
<%}%>
```

2．在 JSP 脚本代码中应使用下面的哪条语句来提前结束 JSP 执行？请编写一个例子程序并解释为什么？

A．System.exit(0);

B．return;

C．两个都不是

3．JSP 规范的设计者为什么不允许 JSP 开发人员直接在 JSP 页面中重新定义 HttpServlet 的 init 方法和 destroy 方法，而是定义 jspInit 方法和 jspDestroy 方法来替代这两个方法的功能呢？

4．在某个 JSP 页面中使用 JSP 声明定义的一个成员变量 globalVar，在 JSP 页面的 jspInit 方法被调用时，将数据库中存储的该页面的以前访问次数读取出来作为 globalVar 变量的初始值，然后在该页面每次被访问时都将 globalVar 变量加 1，当该页面自服务器此次启动以来的访问次数（用另外一个变量来记忆）达到一定的数目后或 JSP 页面的_jspService 方法被调用时，将 globalVar 变量值再保存到数据库，如果要统计访问次数的 JSP 页面比较多，采用这种方式可以大大减少读写数据库的次数。请读者自己编写一个这样的模拟程序，并用一个文件来代替数据库存储页面的访问次数。

5．假设在两个 JSP 页面的 JSP 脚本片断中都定义了一个名称相同的变量，如果其中一个 JSP 页面要引入另外一个 JSP 页面，请问<jsp:include>标签和<%@ include %>指令这两种引入方式中的哪一种会发生错误？为什么？

6．在 JSP 页面中使用如下的指令元素：

```
<%@ page contentType="text/html; charset=GB2312"%>
```

与直接 JSP 脚本片断的开始处使用如下程序代码：

```
<% response.setContentType=" text/html; charset=GB2312"%>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

的运行效果一样吗？

请通过比较下面两个 JSP 文件的运行效果来描述它们的区别。

文件清单 chinese1.jsp：

```
<%
    response.setContentType("text/html;charset=GB2312");
    out.println("hello 中国");
%>
```

文件清单 chinese2.jsp：

```
<%@ page contentType="text/html;charset=gb2312"%>
<%
    out.println("hello 中国");
%>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

第 9 章　JavaBean 在 JSP 中的应用

9.1　JavaBean 简介

JavaBean 是一种特殊的 Java 类，它遵从一定的设计模式，开发工具和其他组件可以根据这种模式来调用 JavaBean。Sun 最初引入 JavaBean 是为了扩展 GUI 程序的开发，事实上，JavaBean 目前已经应用到了 Java 开发的各个方面，其中包括 JSP 开发。JavaBean 可以设计得像 Swing 组件一样复杂，提供方法、属性、事件、甚至可以供开发工具将它实例化到一个 GUI 容器中，并对其属性和行为进行定制。JavaBean 也可以设计得很简单，仅仅提供一些方法和属性供外部环境和其他组件调用。JSP 中使用的 JavaBean 非常简单，本章只介绍这种简单 JavaBean 的设计，它具有如下两个特点：

（1）用做 JavaBean 的类必须具有一个公共的、无参数的构造方法，这个方法可以是通过编译器自动产生的那个默认构造方法。

（2）JavaBean 的属性通过遵循某种规范的公共方法暴露给外部，外部的其他程序可以通过 Java 的反射 API 来查找 JavaBean 中遵循这一规范的方法，从而发现 JavaBean 中的属性。

9.2　JavaBean 的属性

9.1.1　JavaBean 的属性简介

JavaBean 的属性与普通 Java 类的属性的概念不一样，JavaBean 的属性是以方法定义的形式出现的。用于对属性赋值的方法称为属性修改器或 setter 方法；用于读取属性值的方法称为属性访问器或 getter 方法。

属性修改器必须以小写的 set 前缀开始，后跟属性名，且属性名的第一个字母要改为大写，例如，nickName 属性的修改器名称为 setNickName，password 属性的修改器名称为 setPassword。属性访问器通常以小写的 get 前缀开始，后跟属性名，且属性名的第一个字母要改为大写，例如，nickName 属性的访问器名称为 getNickName，password 属性的访问器名称为 getPassword。

JavaBean 的属性名是根据 setter 方法与 getter 方法的名称来生成的，setter 方法或 getter 方法中除去前缀“set”和“get”后的部分即为属性名，但属性名的首字母必须小写，例如

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

nickName、password 等。如果一个属性只有访问器，则该属性为只读属性，例如，列表框中的列表项的个数是不能随意修改的，只能根据列表项的实际个数来返回，所以列表框的列表项数目为只读属性；如果一个属性只有修改器，则该属性为只写属性，只写属性比较少见；如果一个属性既有修改器，又有访问器，则该属性为读写属性，读写属性最为常见。根据属性的复杂性，可以将 JavaBean 的属性分为简单属性和 Indexed 属性。

9.1.2 简单属性

简单属性就是非数组类型的属性。例程 9-1 是一个代表用户注册信息的 JavaBean，该 JavaBean 的名称为 RegisterBean，当前只为它设计了两个简单的属性：nickName（昵称）和 male（性别），其中 male 为布尔类型的属性。

例程 9-1 RegisterBean.jsp

```
package org.it315;
public class RegisterBean
{
    private String nickName = null;
    private boolean male = false;

    public void setNickName(String nickName)
    {
        this.nickName = nickName;
    }
    public String getNickName()
    {
        return nickName;
    }

    public void setMale(boolean male)
    {
        this.male = male;
    }
    public boolean isMale()
    {
        return male;
    }
}
```

由于属性修改器仅仅是用于设置属性的值，而不用返回任何结果，所以其返回值类型是 void；简单属性的修改器必须接受一个参数，参数类型由属性的特性决定，可以是任意类型的。简单属性的访问器仅仅是用于返回属性的值，它不接受任何参数，但它要返回一

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

个值，返回值的类型必须与修改器所接受的参数类型一致。如果属性的类型是一个布尔型的逻辑值，不管访问器返回的类型是 boolean（基本数据类型）还是 Boolean（包装数据类型），访问器名称都可以不使用 get 为前缀，而是以 is 代替。如果 JavaBean 中同时存在 getXxx 和 isXxx 方法，那么 xxx 属性的访问器为 isXxx 方法。

注意 RegisterBean 这个 JavaBean 的属性是根据 setNickName/getNickName 和 setMale/isMale 等方法来生成的，而与 RegisterBean.java 中定义的成员变量 nickName 和 male 的名称无关。RegisterBean.java 中定义的成员变量 nickName 和 male 的名称完全可以修改成其他形式，但将它们命名为与 JavaBean 属性名相同的名称是一个良好习惯。

9.1.3 Indexed 属性

Indexed 属性就是数组类型的属性。Indexed 属性的修改器有两种重载形式：一个是对整个数组进行赋值；另外一个则是对数组中的每个元素进行赋值。Indexed 属性的访问器也有两种重载形式：一个用于返回整个数组；另外一个则是用于返回数组中的单个元素。例程 9-2 在例程 9-1 中的 RegisterBean 的基础上增加了一个表示特长的属性 specialities，由于一个人可以有多个特长，所以 specialities 属性应该是数组类型。

例程 9-2 RegisterBean.jsp

```
package org.it315;
public class RegisterBean
{
    private String nickName = null;
    private boolean male = false;
    private String [] specialities =
        new String[]{"none","none","none","none","none"};
    public void setNickName(String nickName)
    {
        this.nickName = nickName;
    }
    public String getNickName()
    {
        return nickName;
    }

    public void setMale(boolean male)
    {
        this.male = male;
    }
    public boolean isMale()
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
    {
        return male;
    }

    public void setSpecialities(String [] sp)
    {
        this.specialities = sp;
    }
    public void setSpecialities(int index,String sp)
    {
        specialities[index] = sp;
    }
    public String [] getSpecialities()
    {
        return specialities;
    }
    public String getSpecialities(int index)
    {
        return specialities[index];
    }
}
```

对于应用在 JSP 中的 JavaBean 的数组属性，一般不会涉及对其中的单个元素进行赋值和读取的情况，因此，在这种 JavaBean 中可以没有对数组属性的单个元素进行赋值和读取的方法。

9.3 JavaBean 在 JSP 中的应用

9.3.1 在 JSP 中如何使用 JavaBean

尽管 JSP 将网页中要动态产生的内容和固定不变的静态内容采用了不同方式来分别处理，但是，HTML 代码与 Java 代码仍然并存在同一个文件之中，相互嵌套在一起。JSP 自身没有实现 HTML 代码与 Java 代码的完全分离，网页设计人员和 Java 编程人员需要操作同一个 JSP 文件，不易维护和管理。如果把 JSP 页面中的业务逻辑代码封装到一个 JavaBean 中，JSP 页面再调用 JavaBean，则可提供一种分离显示内容和业务逻辑的简单方式，并且有利于软件的组件化、模块化。

为了能在 JSP 页面中集成 JavaBean 组件和支持 JavaBean 的功能，JSP 规范专门定义了三个 JSP 标签：<jsp:useBean>、<jsp:setProperty>和<jsp:getPropperty>，它们分别用于创建和查找 JavaBean 的实例对象，设置 JavaBean 对象的属性，读取 JavaBean 对象的属性。对于 JSP 页面来说，只要一个类具有一个公共的、无参数的构造方法，就可以把这个类当

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

做 JavaBean 来使用，如果类中有不接受任何参数的 getter 方法或只接受一个参数的 setter 方法，就可以把前缀“get”或“set”后面的部分当着一个属性名来引用。例如，JDK 自带的 java.util.Date 类具有一个公共的、无参数的构造方法，其中还包含一个如下定义的方法：

```
public void setTime(long time)
```

因此，在 JSP 页面可以把 java.util.Date 类当做一个 JavaBean 来看待，且认为该 Javaean 包含一个名为 time 的属性。例程 9-3 演示了在 JSP 页面按照调用 JavaBean 的方式对 java.util.Date 类进行调用的情况。

例程 9-3 testBean.jsp

```
<jsp:useBean id="currentDate" class="java.util.Date"/>
<jsp:setProperty name="currentDate" property="time" value="1234567"/>
<jsp:getProperty name="currentDate" property="time"/>
```

JSP 页面可以像调用一个普通 Java 类的方式去调用 JavaBean，即先使用 Java 代码创建 JavaBean 的实例对象，然后直接调用 JavaBean 对象的 getter 方法和 setter 方法。如果按照这种方式来调用 JavaBean，JSP 页面中仍然要出现不少 Java 代码。例程 9-4 演示了在 JSP 页面按照调用普通 Java 类的方式对 java.util.Date 类进行调用的情况。

例程 9-4 testBean.jsp

```
<%
    java.util.Date currentDate = new java.util.Date();
    currentDate.setTime(1234567);
    out.println(currentDate.getTime());
%>
```

使用浏览器分别访问如例程 9-3 和例程 9-4 所示的 testBean.jsp 页面，返回的结果完全相同。如例程 9-3 所示的 testBean.jsp 页面中没有出现一句 Java 代码，这种在 JSP 页面中使用 JavaBean 的方式能有效地解决网页美工制作人员与 Java 编程人员的分工问题，将表现层分派给 HTML 开发人员，将业务逻辑层分派给 Java 开发人员。如果按照例程 9-4 的方式调用一个 JavaBean，根本就没有体现出 JavaBean 相比普通 Java 类的任何优势。要想尽最大限度地减少 JSP 页面中的 Java 代码，JSP 页面就应充分利用 JavaBean 的特点，而不能按照调用普通 Java 类的方式来调用 JavaBean 组件。

9.3.2 <jsp:useBean>标签

<jsp:useBean>标签用于在某个指定的域范围（application、session、request、pageContext 等）中查找一个指定名称的 JavaBean 对象，如果存在则直接返回该 JavaBean 对象的引用，

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

如果不存在则实例化一个新的 JavaBean 对象，并将它按指定的名称存储在指定的域范围中。<jsp:useBean>标签的最常见语法如下：

```
<jsp:useBean id="beanInstanceName " class="package.class"
                scope="page|request|session|application"/>
```

其中，class 属性用于指定 JavaBean 的完整类名（即必须带有包名），JSP 引擎将使用这个类名来创建 JavaBean 的实例对象或作为查找到的 JavaBean 对象的类型；id 属性用于指定 JavaBean 实例对象的引用名称和其存储在域范围中的名称；scope 属性用于指定 JavaBean 实例对象所存储的域范围，其取值只能是 page、request、session 和 application 四个值中的一个，其默认值是 page。scope 属性的各个设置值的意义和作用如下：

- page 表示将 JavaBean 实例对象存储在 PageContext 对象中，存储在 pageContext 对象中的 JavaBean 对象仅可以被当前 JSP 页面的当前请求的响应过程中调用的各个组件访问。
- request 表示将 JavaBean 实例对象存储在 ServletRequest 对象中，存储在 request 对象中的 JavaBean 对象可以被属于同一个请求的所有 Servlet 和 JSP 页面访问。
- session 表示将 JavaBean 实例对象存储在 HttpSession 对象中，存储在 HttpSession 对象中的 JavaBean 对象可以被属于同一个会话的所有 Servlet 和 JSP 页面访问，此设置要求当前 JSP 页面支持 Session，即没有将 page 指令的 session 属性设置为 false。
- application 表示将 JavaBean 实例对象存储在 ServletContext 对象中，存储在 ServletContext 对象中的 JavaBean 对象可以被同一个 Web 应用程序中的所有 Servlet 和 JSP 页面访问。

如果某个 JSP 页面中使用了如下语句：

```
<jsp:useBean id="currentDate" class="java.util.Date"/>
```

查看 JSP 文件所生成的 Servlet 源文件，可以看到该语句所翻译成的 Java 代码如下：

```
…
java.util.Date currentDate = null;
synchronized (_jspx_page_context) {
    currentDate = (java.util.Date) _jspx_page_context.getAttribute(
                "currentDate", PageContext.PAGE_SCOPE);
    if (currentDate == null){
        currentDate = new java.util.Date();
        _jspx_page_context.setAttribute("currentDate",
                currentDate, PageContext.PAGE_SCOPE);
    }
}
…
```

从上面的代码中可以看到，JSP 引擎首先在<jsp:useBean>标签的 scope 属性指定的域

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

范围中查找 id 属性指定的 JavaBean 对象，只有该域范围中不存在 id 属性指定的 JavaBean 对象时，JSP 引擎才会根据 class 属性指定的类名去创建新的 JavaBean 实例对象，并将新建的 JavaBean 实例对象以 id 属性指定的名称存储到 scope 属性指定的域范围中。从上面的代码中还可以看到，JSP 引擎所生成的 JavaBean 实例对象的引用名称和其存储在域范围中的名称都是 currentDate，即都是<jsp:useBean>标签的 id 属性的设置值，这也是 JSP 规范的一个明确要求。

除了 id、class 和 scope 属性之外，<jsp:useBean>标签还有其他两个属性：type 和 beanName。

- type 属性

用于指定 JavaBean 实例对象的引用变量的类型，它必须是 JavaBean 对象的类名称、超类名称和所实现的接口名称中的一个。type 属性的默认值为 class 属性的设置值，当 JSP 引擎将<jsp:useBean>标签翻译成 Servlet 程序中的 Java 代码时，它将使用 type 属性值作为 JavaBean 对象的引用变量的类型。

- beanName 属性

用于指定 JavaBean 的名称，这个名称不是 JavaBean 实例对象的引用名称，而是 JavaBeans 规范中定义的一种名称，它将被作为传递给 java.beans.Beans 类的 instantiate 方法的参数，例如，a.b.c。a.b.c 可以代表一个类的完整名称，也可以代表 a/b/c.ser 这样的资源文件，java.beans.Beans 类的 instantiate 方法可以从这个资源文件中产生出 JavaBean 的实例对象。beanName 属性值还可以通过一个表达式来产生。

<jsp:useBean>标签的完整语法如下：

```
<jsp:useBean id="beanInstanceName" scope="page | request | session |
application"
        {
            class="package.class" |
            type="package.class" |
            class="package.class" type="package.class" |
            beanName="{package.class | <%= expression %>}" type="package.class"
        } />
```

<jsp:useBean>标签首先在指定的域范围中查找指定名称的 JavaBean 实例对象，如果能够找到，这时候只需要为<jsp:useBean>标签设置 type 属性或 class 属性，以便 JSP 引擎在 JSP 页面所翻译成的 Servlet 源文件中设置该 JavaBean 对象的引用变量的类型。如果在指定的域范围中查找不到指定名称的 JavaBean 实例对象，那么必须设置 class 属性或 beanName 属性来指定产生 JavaBean 实例对象的类名或资源名，class 属性和 beanName 属性不能同时设置。因为 type 属性的默认值为 class 属性的设置值，所以设置 class 属性时可以省略 type 属性，而设置 beanName 属性时必须同时设置 type 属性。

<jsp:useBean>标签除了可以按照空元素的方式进行使用外，它还可以按容器元素的格式进行使用，如下所示：

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
<jsp:useBean ...>
    Body
</jsp:useBean>
```

这种格式被称之为带标签体的<jsp:useBean>标签，Body 部分的内容只在<jsp:useBean>标签创建 JavaBean 的实例对象时才执行。也就是说，如果在<jsp:useBean>标签的 scope 属性指定的域范围中存在 id 属性所指定的 JavaBean 对象，<jsp:useBean>的开始标签和结束标签之间的 Body 内容将被忽略，否则，Body 部分的内容将被执行。

带标签体的<jsp:useBean>标签的执行情况

01 在<tomcat 的安装目录>\webapps\it315\jsp\目录中编写一个名为 beanBody.jsp 的 JSP 文件，如例程 9-5 所示。

例程 9-5 beanBody.jsp

```
<jsp:useBean id="currentDate" class="java.util.Date" scope="page">
这是在创建 JavaBean 对象时才看得到的内容
</jsp:useBean>
```

02 使用浏览器访问 beanBody.jsp 页面，浏览器中显示出了“这是在创建 JavaBean 对象时才看得到的内容”。刷新浏览器的访问，浏览器中再次显示出了上述内容。因为存储在 page 域范围中的 JavaBean 对象仅对当前 JSP 页面的当前请求的响应过程有效，存储在 pageContext 对象中的属性不可以被同一个 JSP 页面的每次执行过程共享，所以在每次执行 beanBody.jsp 页面时都要创建新的 page 域范围中的名为 currentDate 的 JavaBean 对象。

03 修改 testBody.jsp 文件，将<jsp:useBean>标签的 scope 属性设置为"session"，如例程 9-6 所示。

例程 9-6 beanBody.jsp

```
<jsp:useBean id="currentDate" class="java.util.Date" scope="session">
这是在创建 JavaBean 对象时才看得到的内容
</jsp:useBean>
```

04 使用浏览器访问修改过的 beanBody.jsp 页面，浏览器中显示出了“这是在创建 JavaBean 对象时才看得到的内容”。刷新浏览器的访问，浏览器中这次不再显示出上述内容。在第一次访问 beanBody.jsp 页面时，由于在当前 session 域范围中还不存在名为 currentDate 的 JavaBean 对象，beanBody.jsp 将创建该 JavaBean 对象并显示出<jsp:useBean>标签体中的内容。当使用同一个浏览器进程再次访问 beanBody.jsp 页面时，由于存储在 session 域范围中的 JavaBean 对象对属于当前会话的所有 JSP 页面的执行过程都有效，所以，beanBody.jsp

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

的这次执行过程将不再创建名为 currentDate 的 JavaBean 对象，<jsp:useBean>标签体中的内容也就不会被执行。

05 重新启动一个浏览器进程访问修改过的 beanBody.jsp 页面，浏览器中又显示出了“这是在创建 JavaBean 对象时才看得到的内容”。因为新启动的浏览器进程发出的访问与原来的浏览器进程发出的访问不属于同一个会话，新启动的浏览器进程开启了一个新的会话过程，在新启动的浏览器的 session 域范围中显然还不存在名为 currentDate 的 JavaBean 对象，所以 beanBody.jsp 将创建该 JavaBean 对象并显示出<jsp:useBean>标签体中的内容。

9.3.3 <jsp:setProperty>标签

<jsp:setProperty>标签用于设置 JavaBean 对象的属性，也就是调用 JavaBean 对象的 setter 方法，其使用语法如下：

```
<jsp:setProperty name="beanInstanceName"
{
    property="propertyName" value="{string | <%= expression %>}" |
    property="propertyName" [ param="parameterName" ] |
    property= "*"
}/>
```

下面是其中各个属性的意义和作用：

- name 属性是必不可少的，它用于指定 JavaBean 实例对象的名称，其值应与<jsp:useBean>标签的 id 属性值相同。
- property 属性是必不可少的，它用于指定 JavaBean 实例对象的属性名。
- value 属性是可选的，它用于指定 JavaBean 实例对象的某个属性的值。value 属性的设置值可以是一个字符串，也可以是一个表达式。如果 value 属性的设置值是字符串，那么它将被自动转换成所要设置的 JavaBean 属性的类型，例如，通过 Boolean.valueOf 方法将“true”转换成 Boolean 和 boolean 类型，通过 Integer.valueOf 方法将“315”转换成 Integer 和 int 类型。如果 value 属性的设置值是一个表达式，那么该表达式的结果类型必须与所要设置的 JavaBean 属性的类型一致。
- param 属性是可选的，它用于将一个请求参数的值赋值给 JavaBean 实例对象的某个属性，它可以将作为字符串类型返回的请求参数值自动转换成要设置的 JavaBean 属性的类型。如果当前请求消息中没有 param 属性所指定的请求参数，那么<jsp:setProperty>标签什么事情也不做，它不会将 null 值赋给 JavaBean 属性，所设置的 JavaBean 属性仍将等于其原来的初始值，以保证 JavaBean 属性的初始值不被破坏。value 和 param 属性不能同时使用，只能使用其中任意一个。

下面是其中每一种应用形式的意义和作用：

（1）<jsp:setProperty name="beanInstanceName" property="propertyName" value= "{string

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

| <%= expression %>}" />

这种形式通过 value 属性来指定 JavaBean 实例对象的某个属性的值，value 属性的设置值可以是一个字符串，也可以是一个表达式。

（2）<jsp:setProperty name="beanInstanceName" property="propertyName" param="parameterName" />

这种形式通过 param 属性来将某个请求参数值赋给 JavaBean 实例对象的一个属性。

（3）<jsp:setProperty name="beanInstanceName" property="propertyName" />

这种形式将 JavaBean 实例对象的某个属性值设置为与该属性同名的请求参数值，它等效于 param 属性的设置值与 property 属性值相同时的情况。也就是说，如果<jsp:setProperty>标签的 param 属性和 value 属性都没有设置，则表示将当前要设置的 JavaBean 实例对象的属性设置为与该属性同名的请求参数值，且 JavaBean 属性名与请求参数名的大小写应完全一致。

（4）<jsp:setProperty name="beanInstanceName" property="*" />

这种形式用于对 JavaBean 实例对象中的多个属性进行赋值，它表示将请求消息中的参数逐一与 JavaBean 实例对象中的属性进行比较，如果找到同名的属性，则将该参数值赋给该属性。对于这种情况，请求消息中的参数的排列顺序与 JavaBean 中的属性的排列顺序没有严格要求，如果在请求消息中不存在与 JavaBean 中的某个属性同名的参数，那么 JavaBean 的这个属性将不会被赋值。

<jsp:setProperty>标签也可以嵌套在<jsp:useBean>标签中使用，如下所示：

```
<jsp:useBean id="beanInstanceName" …/>
    …
    <jsp:setProperty name="beanInstanceName" property="propertyName" …/>
    …
</jsp:useBean>
```

由于嵌套在<jsp:useBean>标签中的主体内容只有在<jsp:useBean>标签创建 JavaBean 实例对象时才执行，如果<jsp:useBean>标签所引用的 JavaBean 实例对象已经存在，嵌套在其中的主体内容将不被执行，所以，这种情况下的<jsp:setProperty>标签仅在创建 JavaBean 实例对象时才执行，只能用于对 JavaBean 实例对象进行初始化。如果<jsp:setProperty>标签在<jsp:useBean>标签之后独立使用，那么，不管<jsp:useBean>标签是创建新的 JavaBean 实例对象，还是引用已有的 JavaBean 实例对象，<jsp:setProperty>标签都会被执行。

<jsp:setProperty>标签的应用

01 修改例程 9-3 中的 testBean.jsp 文件，使用一个获取请求参数值的表达式来对 java.util.Date 对象的 time 属性进行赋值，如例程 9-7 所示。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

例程 9-7 testBean.jsp

```
<jsp:useBean id="currentDate" class="java.util.Date"/>
<jsp:setProperty name="currentDate" property="time"
        value='<%= request.getParameter("timeValue")%>'/>
<jsp:getProperty name="currentDate" property="time"/>
```

上面的 value 属性值部分使用的是单引号（'），这主要是为了防止与表达式中的字符串的双引号（"）产生冲突而导致有些 JSP 引擎发生解析错误。在浏览器地址栏中输入如下地址访问 testBean.jsp 页面：

http://localhost:8080/it315/jsp/testBean.jsp?timeValue=1234567

浏览器中显示出如图9.1所示的错误。

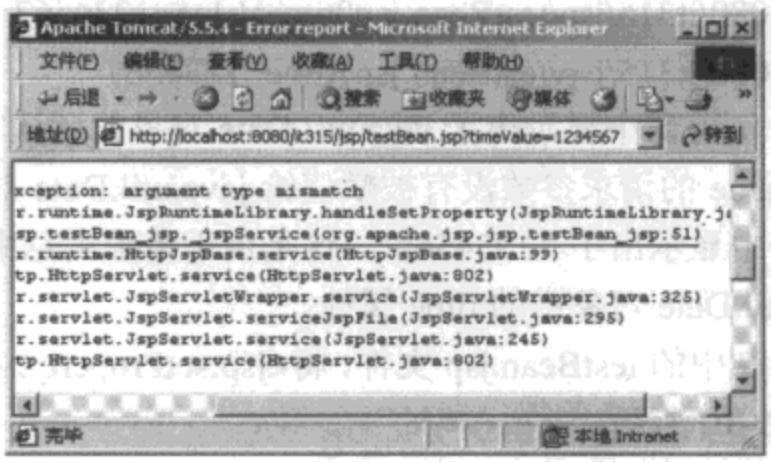


图 9.1

如图 9.1 所示的错误提示信息中可以看到，JSP 页面已经被成功地翻译成了 Servlet，错误是在 Servlet 运行时出现的，并且错误位于翻译成的 Servlet 源文件中的 51 行。这个错误的根本原因就是 value 属性的设置值中的表达式的结果类型与所要设置的 JavaBean 属性的类型不一致，将 testBean.jsp 文件修改成如下形式。

```
<jsp:useBean id="currentDate" class="java.util.Date"/>
<jsp:setProperty name="currentDate" property="time"
        value='<%= Integer.parseInt(request.getParameter("timeValue")) %>'/>
<jsp:getProperty name="currentDate" property="time"/>
```

刷新浏览器的访问，浏览器中这时显示出了正常的效果。

02 修改例程 9-7 中的 testBean.jsp 文件，使用<jsp:setProperty>标签的 param 属性来对 java.util.Date 对象的 time 属性进行赋值，如例程 9-8 所示。

例程 9-8 testBean.jsp

```
<jsp:useBean id="currentDate" class="java.util.Date"/>
<jsp:setProperty name="currentDate" property="time" param="timeValue"/>
<jsp:getProperty name="currentDate" property="time"/>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

在浏览器地址栏中输入如下地址访问 testBean.jsp 页面：

```
http://localhost:8080/it315/jsp/testBean.jsp?timeValue=1234567
```

浏览器中显示出了正常的效果。

03 修改例程 9-8 中的 testBean.jsp 文件，将<jsp:setProperty>标签中的 param 属性部分删除掉，如例程 9-9 所示。

例程 9-8 testBean.jsp

```
<jsp:useBean id="currentDate" class="java.util.Date"/>
<jsp:setProperty name="currentDate" property="time" />
<jsp:getProperty name="currentDate" property="time"/>
```

在浏览器地址栏中分别输入使用如下两个参数名的地址访问 testBean.jsp 页面：

- http://localhost:8080/it315/jsp/testBean.jsp?timeValue=1234567
- http://localhost:8080/it315/jsp/testBean.jsp?time=1234567

对于第一种情况，浏览器中显示出了 java.util.Date 对象的默认时间值，这是由于请求消息中的名称为 timeValue 的请求参数没有被赋值给 java.util.Date 对象中的 time 属性；对于第二种情况，浏览器中显示出了“1234567”，这是由于请求消息中的名称为 time 的请求参数被赋值给了 java.util.Date 对象中的 time 属性。

04 修改例程 9-9 中的 testBean.jsp 文件，将<jsp:setProperty>标签中的 property 属性设置为“*”，如例程 9-10 所示。

例程 9-10 testBean.jsp

```
<jsp:useBean id="currentDate" class="java.util.Date"/>
<jsp:setProperty name="currentDate" property="*" />
<jsp:getProperty name="currentDate" property="time"/>
```

在浏览器地址栏中重复第 **03** 步的访问操作，可以看到其执行效果与第 **03** 步看到的效果完全相同。

05 由于在 JSP 页面所翻译成的 Servlet 源文件中，JSP 引擎为<jsp:useBean>标签定义了一个指向 JavaBean 实例对象的引用变量，该变量的名称为<jsp:useBean>标签的 id 属性值，所以，在<jsp:useBean>标签后面除了可以使用<jsp:setProperty>标签设置 JavaBean 的属性外，还可以在 JSP 脚本片断中调用 id 属性所命名的 JavaBean 实例对象的 setter 方法来设置其属性。修改例程 9-10 中的 testBean.jsp 文件，用 JSP 脚本片断中的代码来替代<jsp:setProperty>标签对 java.util.Date 对象中的 time 属性赋值，如例程 9-11 所示。

例程 9-11 testBean.jsp

```
<jsp:useBean id="currentDate" class="java.util.Date"/>
<% currentDate.setTime(1234567);%>
<jsp:getProperty name="currentDate" property="time"/>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

使用浏览器访问 testBean.jsp 页面，浏览器中显示出了正常的效果。

多学两招

利用 JSP 表达式对 JavaBean 的特殊类型的属性进行赋值

当在 JSP 标签的属性值中使用 JSP 表达式时，JSP 引擎不再是将表达式的结果插入到响应消息中，而是用表达式的计算结果对标签的属性进行赋值。当一个 JavaBean 的某个属性的类型为一个特殊的类时，在<jsp:setProperty>标签中就必须使用 JSP 表达式对该属性进行赋值。例如，假设有一个代表论坛帖子的 JavaBean，该 JavaBean 的类名为 org.it315.Article，且具有一个代表帖子板块的属性，该属性名为 board，属性类型为 org.it315.Board，采用<jsp:setProperty>标签对 org.it315.Atticle 这个 JavaBean 的 board 属性进行赋值的例子程序如下所示：

```
<jsp:useBean id="article" class="org.it315.Article"/>
<% board = new org.it315.Board("java");%>
<jsp:setProperty name="article" property="board" value="<%= board %>"/>
```

9.3.4 <jsp:getProperty>标签

<jsp:getProperty>标签用于读取 JavaBean 对象的属性，也就是调用 JavaBean 对象的 getter 方法，然后将读取的属性值转换成字符串后插入进输出的响应正文中，其使用语法如下：

```
<jsp:getProperty name="beanInstanceName" property="PropertyName" />
```

其中，name 属性用于指定 JavaBean 实例对象的名称，其值应与<jsp:useBean>标签的 id 属性值相同，property 属性用于指定 JavaBean 实例对象的属性名。如果一个 JavaBean 实例对象的某个属性的值为 null，那么，使用<jsp:getProperty>标签输出该属性的结果将是一个内容为“null”的字符串。在前面的例子已经多次使用过<jsp:getProperty>标签，这里不再单独对此进行举例讲解。

与<jsp:setProperty>标签的原理一样，除了可以使用<jsp:getProperty>标签来读取 JavaBean 的属性外，也可以在 JSP 脚本片断中调用 id 属性所命名的 JavaBean 实例对象的 getter 方法来读取其属性。例程 9-11 中的 testBean.jsp 文件也可以修改成如例程 9-12 所示的形式。

例程 9-12 testBean.jsp

```
<jsp:useBean id="currentDate" class="java.util.Date"/>
<% currentDate.setTime(1234567);%>
<%= currentDate.getTime()%>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

9.3.5 使用 JavaBean 的注意事项

在 JSP 页面中使用 JavaBean 时，应注意如下一些事项：

（1）因为 JavaBean 要被 JSP 页面调用，所以，JavaBean 应放置在 JSP 页面的类装载器或其父级类装载器所能装载的目录中，通常放置于 Web 应用程序下的 Web-INF/classes 目录中。

（2）有些版本的 Tomcat 不会自动重新加载修改过的 JavaBean，如果 JSP 页面加载 JavaBean 以后又修改和重新编译了 JavaBean 程序，那么需要修改 JSP 页面或者重新启动 Tomcat，否则，JSP 页面使用的仍然是修改前的 JavaBean 对象。

（3）由于带包名的类无法调用不带包名的类，且 JSP 页面所翻译成的 Servlet 程序都带有包名，所以，JavaBean 必须带有包名，不能用默认包名。

（4）在选择存储 JavaBean 的域范围时，如果使用 request 域能够满足需求的话，则不要使用 Session 域。因为存储在 request 域中的 JavaBean 对象只在同一个请求的页面之间有效，在请求处理完成时即被自动释放，而存储在 Session 域中的 JavaBean 对象则需要等到 Session 失效时才会自动释放，造成了一些不必要的内存消耗。

9.4 JSP 开发模式

9.4.1 MVC 设计模式

在开发一个软件之前，必须先设计好其架构，一个基本的架构思想是将软件分成不同的模块，问题的关键就是如何划分模块。MVC（模型-视图-控制器）是 20 世纪 80 年代为 Smalltalk-80 编程语言发明的一种软件设计模式，它是一种分离业务逻辑与显示界面的设计方法，至今已经被广泛地应用。一个软件系统通常可分为数据系统、表现系统和交互系统，MVC 模式扩大了这种系统划分的概念，它不仅用来划分一个大的软件系统，而且可以应用在一个小的程序设计中。MVC 的模型（Model）部分负责管理程序的业务数据，视图（View）部分负责显示界面，控制器（Controller）部分则负责与用户进行交互（接受请求和选择响应视图），这几个功能模块的关系可以表示为如图 9.2 所示的一个互相连接的三角形。

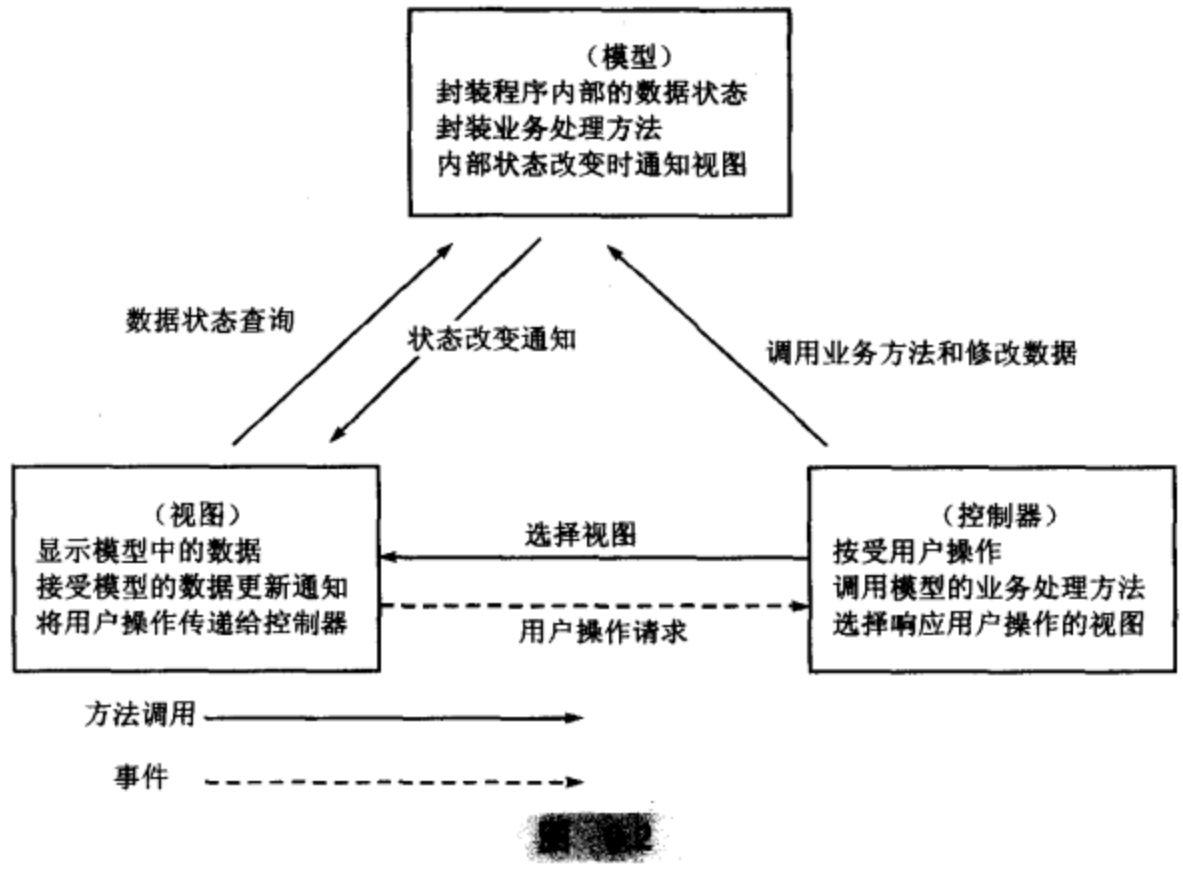
图 9.2 中的视图和模型不止一个，一个视图可以查询多个模型，一个模型发生更改时可以通知多个视图，一个控制器可以操作多个模型和视图。在采用 MVC 模式构建的软件开始工作时，用户根据某个视图的显示信息进行操作，操作请求通过视图组件提交给控制器，控制器再根据请求信息调用模型组件的业务方法和修改模型的数据状态。控制器调用模型组件进行处理后，根据模型的返回结果选择相应的视图组件来显示处理结果和模型中的数据。

另外，当模型组件的数据发生改变后，它也可以通知与之相关的视图组件更新显示，以保持与模型数据的状态同步。模型的“状态改变通知”一般都是使用观察者通知模式来

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

实现的，可以将多个视图注册为一个模型的观察者，模型在数据状态发生改变时，主动给所有注册的视图发送消息。

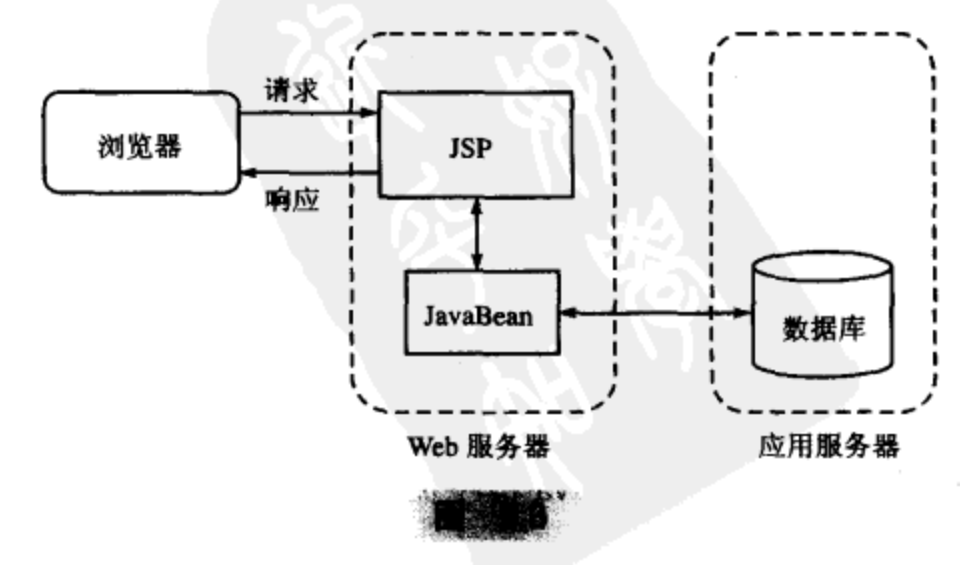


MVC 的数据模块和显示模块实现了良好的松耦合，只要它们相互连接的接口保持不变，对其中一个的修改不会影响另外一个，所以，采用 MVC 模式构建的应用系统具有极高的可维护性、可扩展性、可移植性和组件可复用性。许多现代应用系统和开发环境，例如 Java 中的 Swing 组件，都使用 MVC 作为它们的基础架构。

9.4.2 JSP 开发模式

为了帮助开发人员更好地利用 JSP 来开发 Web 应用程序，Sun 在早期的 JSP 规范中定义了两种使用 JSP 的模式：Model 1（模式一）和 Model 2（模式二）。

模式一主要采用 JSP+JavaBean 技术，JSP 页面负责处理请求和输出响应结果，并将其中发生的一些业务逻辑则交给 JavaBean 处理，如图 9.3 所示。

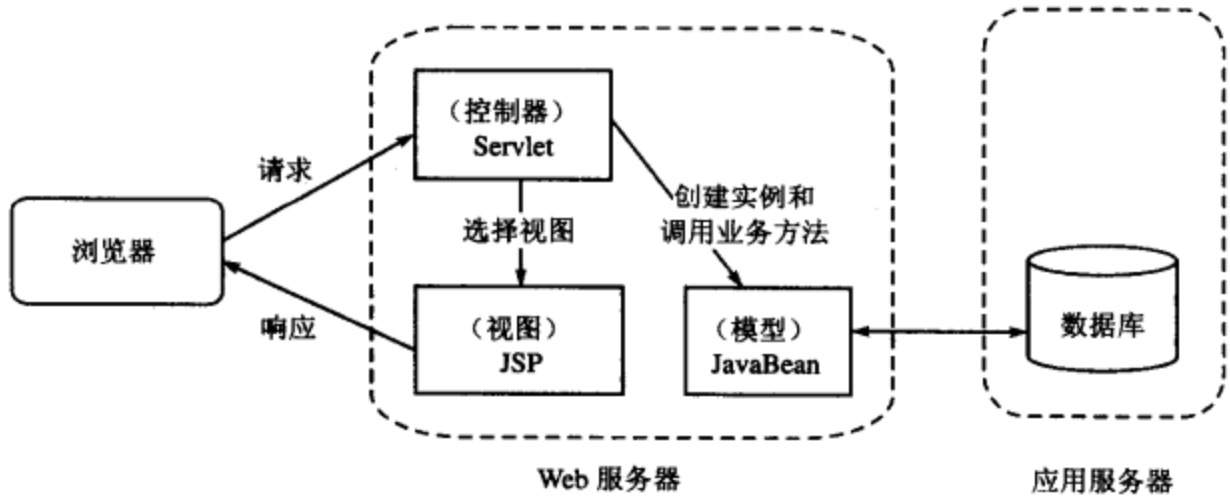


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

模式一最大的特点就是将一次请求的响应过程完全交给一个 JSP 页面负责，其典型的应用就是让创建表单的 JSP 页面同时负责处理该表单提交的请求，也就是 JSP 页面既要输出创建表单的 HTML 代码，又要处理表单提交的数据。

模式二借鉴了 MVC（模型—视图—控制器）设计模式来实现显示内容和业务逻辑的完全分离，它综合采用 Servlet＋JSP＋JavaBean 技术，由 Servlet 处理请求和控制业务流程，由 JSP 输出响应结果，由 JavaBean 负责具体的业务数据和业务逻辑，如图 9.4 所示。



模式二与模式一的最大区别就是将处理请求的功能与产生显示内容的功能分配给两个独立的模块来完成，由 Servlet 负责接受用户请求和处理业务流程，由 JSP 产生显示页面。在模型二中，Servlet 扮演着控制器的角色，当它接收到客户端的请求后，首先创建和调用相应的 JavaBean 来完成具体的业务逻辑，然后调用合适的 JSP 页面来产生显示内容。因为 JSP 页面可以直接使用模板元素来产生网页文档的内容，同时可以获取 JavaBean 中的数据，根据这一特点，JSP 页面在模型二中专门负责视图表示层，即只产生要输出给客户端的显示内容。JavaBean 充当了数据模型的角色。由于模型位于 Web 服务器端，而最终的视图位于用户浏览器端，只有浏览器发出请求后，服务器才会做出响应，没有请求就没有响应，所以，在 Web 应用中难以实现模型的“状态改变通知”事件，视图不能随模型的改变而自动更新。由于在 Web 应用中无法完全实现真正意义上的 MVC 模式，模式二只能算是 MVC 设计模式的一种变体，有人干脆将它称之为 Web MVC。

在模式二中，Servlet 不再担负生成显示内容的任务，只负责控制业务流程的任务，而 JSP 页面中没有任何业务流程和商业逻辑，它只是简单地从 Servlet 先前创建好的 JavaBean 对象中检索数据，再将这些数据动态插入预定义的模板中，不懂得 Java 代码的普通 HTML 设计人员完全可以编写和维护 JSP 页面。显然，模式二将 Java 应用程序开发者和网页制作者的工作进行了有效分离，让 Java 程序员专注于 Java 程序代码的编写，而 HTML 设计人员专注于页面的表现。在模式二中，只要各个组件相互连接的接口保持不变，任意一个组件发生修改，其他组件都不用随之改变，例如，HTML 设计人员可以改变页面显示的外观，根本就不需要告诉负责 Servlet 开发的 Java 程序员。在这种开发模式中，一个 Web 应用程

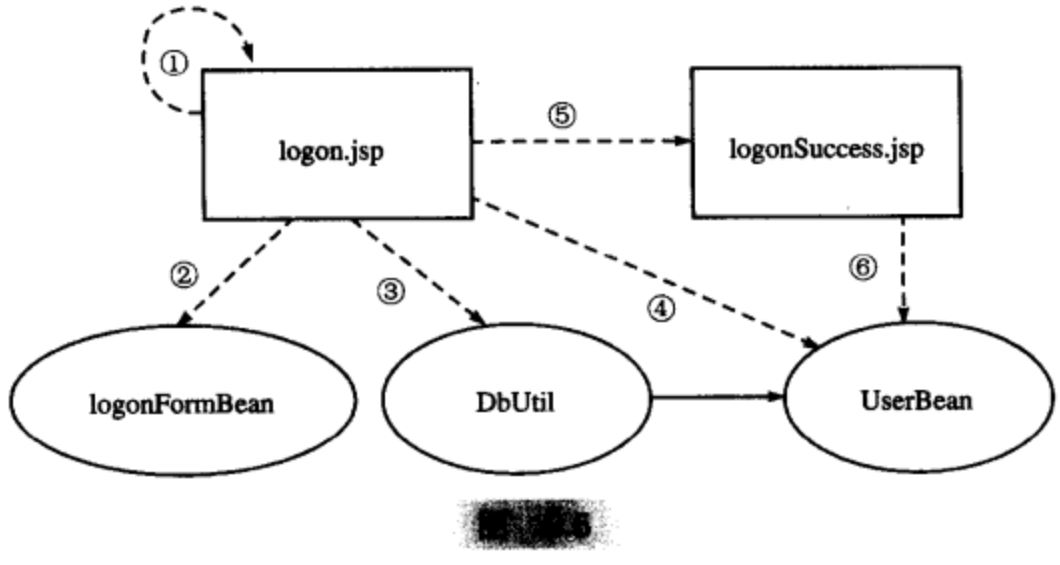
http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

序可以由一个 Servlet 来控制其整个业务流程，也可以由多个 Servlet 来分别控制该应用程序中的各个用例的业务流程，例如，用户注册和用户登录是一个 Web 应用中的两个用例，它们的业务流程可以由一个 Servlet 进行统一控制，也可以由两个 Servlet 来单独控制。

9.4.3 用模式一实现用户登录功能

如图 9.5 所示的是一个采用 JSP 模式一设计的验证用户登录信息的程序架构，它包含两个 JSP 页面、两个 JavaBean 和一个访问数据库的辅助类：logon.jsp、logonSuccess.jsp、LogonFormBean.java、UserBean.java 和 DbUtil.java。



各个程序组件的功能和相互之间的工作关系如下：

（1）logon.jsp 为显示登录表单和处理登录请求的页面，登录成功后跳转到 logon Success.jsp 页面，登录失败时重新显示登录表单和失败的帮助信息，以便用户重新登录；如果已登录用户再次访问 logon.jsp，login.jsp 将请求转发给 logonSuccess.jsp 页面。

（2）一个表单中填写的信息最好是绑定到一个 JavaBean 对象上后再进行处理，即一个表单通常都有一个与之对应的 JavaBean 对象，LogonFormBean.java 就是封装登录表单信息的 JavaBean 程序，它可以对各个属性（也就是登录表单内的各个字段中所填写的数据）进行语法格式方面的校验。

（3）UserBean.java 为代表用户信息的 JavaBean 程序，logon.jsp 页面根据登录信息从 DbUtil 对象中检索出相应的 UserBean 对象，logonSuccess.jsp 页面从 UserBean 对象中提取用户信息进行显示。

（4）DbUtil.java 是用于访问数据库的辅助类，它相当于一个 DAO（数据访问对象），这里用 DbUtil 类中封装的一个 HashTable 对象来代替数据库，HashTable 对象中的每一个元素即为一个 UserBean 对象。

（5）logonSuccess.jsp 是用户登录成功后进入的页面，这里仅仅是显示出登录用户的信息和提供注销用户登录的功能；如果用户没有登录就直接访问 logonSuccess.jsp 页面，logonSuccess.jsp 将请求转发给 logon.jsp 页面以要求用户登录。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

用 JSP 模式一实现用户登录功能

01 按上面描述的功能需求编写出图 9.5 中的各个程序组件的代码。

例程 9-13 UserBean.java

```
package org.it315.bean;
import java.io.Serializable;

/**UserBean 中仅仅是定义了三个属性：name、password 和 email。
注意：要存储在 Session 域中或跨 JVM 传输的 JavaBean 应实现 Serializable 接口
*/
public class UserBean implements Serializable
{
    private String name = "";
    private String password = "";
    private String email = "";

    public void setName(String name)
    {
        this.name = name;
    }
    public String getName()
    {
        return this.name;
    }

    public void setPassword(String password)
    {
        this.password = password;
    }
    public String getPassword()
    {
        return this.password;
    }

    public void setEmail(String email)
    {
        this.email = email;
    }
    public String getEmail()
    {
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        return this.email;
    }

    public boolean validatePassword(String password)
    {
        if(this.password.equals(password))
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

例程 9-14 DbUtil.java

```
package org.it315.util;
import org.it315.bean.UserBean;
import java.util.Hashtable;

/**DbUtil 类是一个单件类，在整个 Web 应用程序中只能创建 DbUtil 类的一个实例对象，
在 DbUtil 类内部预存储了两个用户信息来模拟数据库中的用户记录。*/
public class DbUtil
{
    private static DbUtil instance = new DbUtil();
    private Hashtable users = new Hashtable();

    private DbUtil()
    {
        UserBean user1 = new UserBean();
        user1.setName("zxx");
        user1.setPassword("12345678");
        user1.setEmail("zxx@it315.org");
        users.put("zxx",user1);

        UserBean user2 = new UserBean();
        user2.setName("flx");
        user2.setPassword("abcdefg");
        user2.setEmail("flx@it315.org");
        users.put("flx",user2);
    }
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
    public static DbUtil getInstance()
    {
        return instance;
    }

    public UserBean getUser(String userName)
    {
        UserBean user = (UserBean) users.get(userName);
        return user;
    }
}
```

例程 9-15 LogonFormBean.java

```
package org.it315.bean;
import java.util.*;

/*LogonFormBean 用于封装登录表单的信息，其中定义了两个属性：name 和 password。
LogonFormBean 对这两个属性进行基本的格式验证，如果有误则将相应的错误信息
保存到一个 HashTable 对象中，同时也定义了供 JSP 页面检索错误信息的方法。*/
public class LogonFormBean
{
    private String name = "";
    private String password = "";

    private Hashtable errors = new Hashtable();
    public void setName(String name)
    {
        this.name = name;
    }
    public String getName()
    {
        return this.name;
    }
    public void setPassword(String password)
    {
        this.password = password;
    }
    public String getPassword()
    {
        return this.password;
    }
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
    public boolean validate()
    {
        boolean allOk = true;
        if(name.trim().equals(""))
        {
            errors.put("name","Please input your name.");
            allOk = false;
        }
        if(password.length()>10 || password.length()<6)
        {
            errors.put("password","password must have 6-10 characters.");
            allOk = false;
        }
        return allOk;
    }

    public void setErrorMsg(String err,String errMsg)
    {
        if((err != null) && (errMsg != null))
        {
            errors.put(err,errMsg);
        }
    }

    public String getErrorMsg(String err)
    {
        String err_msg = (String)errors.get(err);
        return (err_msg == null) ? "" : err_msg;
    }
}
```

例程 9-16 logon.jsp

```
<%@ page contentType="text/html;charset=GB2312" %>
<%@ page import="org.it315.util.DbUtil" %>
<%@ page import="org.it315.bean.UserBean" %>

<%
    //如果当前会话的用户已经登录
    if(session.getAttribute("logonUser") != null)
    {
%>
        <jsp:forward page="logonSuccess.jsp" />
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
<%
    }
%>
<jsp:useBean id="logonForm" class="org.it315.bean.LogonFormBean" scope=
"page"/>
<jsp:setProperty name="logonForm" property="*" />
<%
//如果 JSP 页面的当前执行过程是对表单提交的响应
if(request.getParameter("submit") != null)
{
    //如果表单字段中填写的内容格式没有问题
    if(logonForm.validate())
    {
        DbUtil db = DbUtil.getInstance();
        UserBean user = db.getUser(logonForm.getName());
        //如果数据库中不存在登录用户名的记录
        if(user == null)
        {
            logonForm.setErrorMsg("name","no this user!");
        }
        else
        {
            //如果用户密码不正确
            if(user.validatePassword(logonForm.getPassword()))
            {
                /*用户成功登录后的一系列会话活动中都可能要使用
                当前用户的信息，所以，应该将当前登录用户的信息
                保存到 Session 域中*/
                session.setAttribute("logonUser",user);
%>
                <jsp:forward page="logonSuccess.jsp" />
<%
            }
            else
            {
                logonForm.setErrorMsg("password","password error!");
            }
        }
    }
}
%>

<%--
如果下面的 FORM 表单是登录失败后的再次显示，其中的各个字段应自动设置为
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
上次输入的内容，并在发生错误的字段旁用红色字体显示出错误的提示信息。
--%>
<form action="logon.jsp" method="post">
姓名: <input type="text" name="name" value='<%=logonForm.getName()%>'>
    <font color="red"><%=logonForm.getErrorMsg("name")%></font><br>
密码: <input type="password" name="password" value='<%=logonForm.get
Password()%>'>
    <font color="red"><%=logonForm.getErrorMsg("password")%></font><br>
<input type="submit" name="submit" value="登录">
</form>
```

例程 9-17 logonSuccess.jsp

```
<%@ page contentType="text/html;GB2312" %>

<%
    //如果 JSP 页面的当前执行过程是对注销行为的响应
    if("logout".equals(request.getParameter("action")))
    {
        session.invalidate();
%>
        <jsp:forward page="logon.jsp"/>
<%
    }
    //如果当前会话的用户还没有登录
    if(session.getAttribute("logonUser") == null)
    {
%>
        <jsp:forward page="logon.jsp" />
<%
    }
%>
<jsp:useBean id="logonUser" class="org.it315.bean.UserBean" scope=
"session" />

恭喜你，登录成功！<br>
<jsp:getProperty name="logonUser" property="name" /><br>
<jsp:getProperty name="logonUser" property="password" /><br>
<jsp:getProperty name="logonUser" property="email" /><br>
<a href="logonSuccess.jsp?action=logout">注销</a><br>
<a href="logon.jsp">重新登录</a>
```

编译上面的三个 Java 源文件，确保编译后生成的 class 文件及其包名对应的目录存放

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

在了<tomcat 安装目录>\webapps\it315\Web-INF\classes 目录中。将两个 JSP 文件保存在<tomcat 安装目录>\webapps\it315\jsp 目录中。

02 模拟故意跳过登录过程，直接访问 logonSuccess.jsp 页面的情况。在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/jsp/logonSuccess.jsp
```

浏览器中显示出了 logon.jsp 页面中的 FORM 表单要求用户进行登录，这是由于 logonSuccess.jsp 检查到当前会话的用户还没有通过登录，于是将请求转发给 logon.jsp 页面。

03 在浏览器显示的表单的姓名字段中填写“zxx”，在密码字段中填写“123”，单击登录按钮，浏览器中显示出重新登录的界面，并提示出登录失败的信息，这是在语法格式检查阶段就发现的错误，如图 9.6 所示。

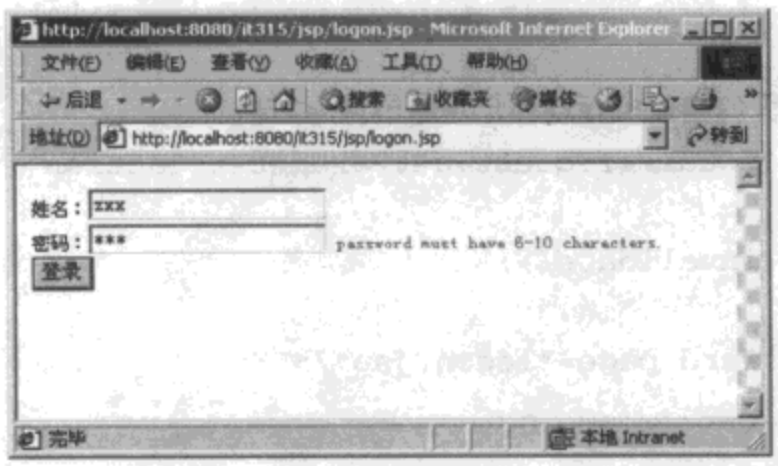


图 9.6

从图 9.6 中可以看到，FORM 表单内的各个字段中自动填写出了上次输入的内容，并在发生错误的字段旁用红色字体显示出了错误的提示信息。

04 在如图 9.6 所示的表单的姓名字段中填写“zxx”，在密码字段中填写“123456”，单击登录按钮，浏览器中又显示出了重新登录的界面和另外一种登录失败的信息，这是在业务逻辑检查阶段发现的错误，如图 9.7 所示。

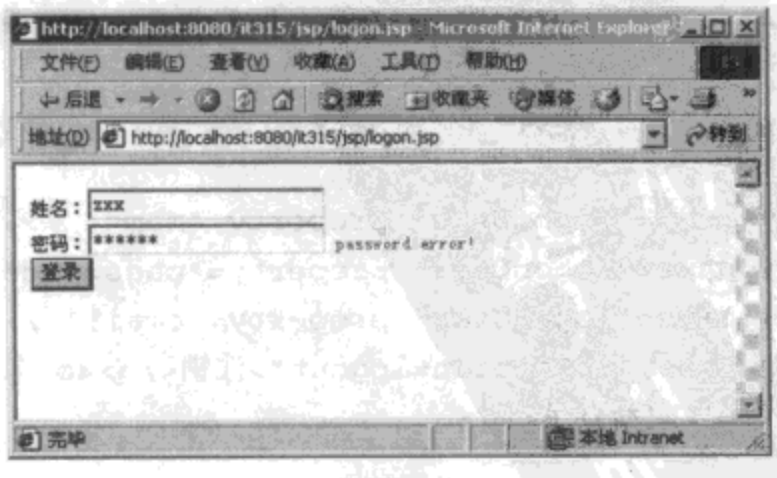


图 9.7

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

05 在如图 9.7 所示的表单的姓名字段中填写“zxx”，在密码字段中填写“12345678”，单击登录按钮，浏览器中显示出了登录成功的信息，如图 9.8 所示。

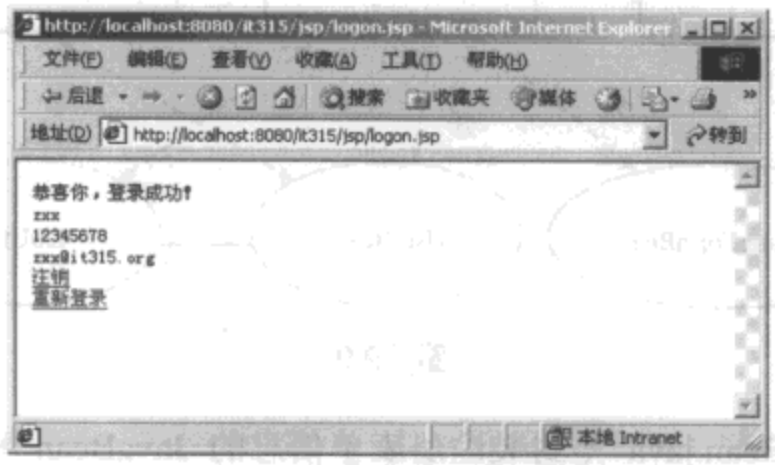


图 9.8

06 单击图 9.8 中的“重新登录”超链接，尽管该超链接的目标是 logon.jsp，但是由于该会话的用户已经登录，logon.jsp 页面将请求转发给了 logonSuccess.jsp 页面，所以浏览器中没有显示出登录表单，而是依旧显示出了 logonSuccess.jsp 页面的输出内容。单击图 9.8 中的“注销”超链接，浏览器中将显示出登录表单。

从上面的例子程序中可以看到，模式一虽然在一定程度上实现了显示内容和业务逻辑的分离，但 JSP 页面除了要负责内容的显示外，它还要负责处理请求和控制业务流程（根据不同的请求信息调用不同的程序代码），没有做到显示和控制的完全分离。当要处理复杂的业务流程时，JSP 页面中内嵌的 Java 代码就会很多，给 JSP 页面的开发和系统维护都将带来极大困难，所以在大型项目中尽量不要使用模式一。模式一的惟一优点就是简单，如果系统的规模比较小，程序处理的业务流程比较简单，且希望加快系统的开发进度，那么就可以采用模式一。

9.4.4 用模式二实现用户注册功能

图 9.9 是一个在 9.4.3 节的例子程序的基础上，采用模式二设计的实现用户注册功能的程序架构，它包含两个 JSP 页面、一个 Servlet、两个 JavaBean 和一个访问数据库的辅助类：register.jsp、logonSuccess.jsp、ControllerServlet.java、RegisterFormBean.java、UserBean.java 和 DbUtil.java。

各个程序组件的功能和相互之间的工作关系如下：

（1）register.jsp 为显示用户注册表单的页面，它的注册请求提交给 ControllerServlet 程序处理。

（2）ControllerServlet 是作为控制器的 Servlet 程序，它负责处理注册用户信息的请求，注册成功后跳转到 logonSuccess.jsp 页面，注册失败时跳回到 register.jsp 页面重新显示注册表单和失败的帮助信息。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

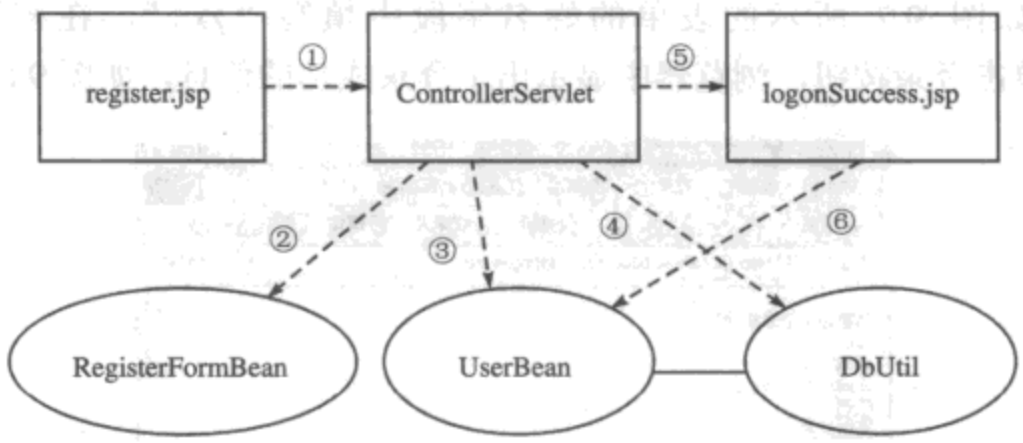


图 9.9

（3）RegisterFormBean.java 是封装注册表单信息的 JavaBean 程序，ControllerServlet 将注册表单信息绑定到 RegisterFormBean 对象上，然后调用 RegisterFormBean 对象的方法对各个属性（也就是注册表单内的各个字段中所填写的数据）进行语法格式方面的校验。

（4）UserBean.java 为代表用户信息的 JavaBean 程序，ControllerServlet 根据用户注册信息创建出一个 UserBean 对象后添加到 DbUtil 对象中，logonSuccess.jsp 页面从 UserBean 对象中提取用户信息进行显示。

（5）DbUtil.java 是用于访问数据库的辅助类，它相当于一个 DAO（数据访问对象），这里用 DbUtil 类中封装的一个 HashTable 对象来代替数据库，HashTable 对象中的每一个元素即为一个 UserBean 对象。

（6）logonSuccess.jsp 是用户登录成功后进入的页面，新注册成功的用户自动完成登录，直接进入 logonSuccess.jsp 页面。

用 JSP 模式二实现用户注册功能

01 按上面描述的功能需求编写出图 9.9 中的各个程序组件的代码。其中，ControllerServlet 程序的映射路径为/servlet/controller，UserBean 类直接采用 9.4.3 节的程序代码，DbUtil 类在 9.4.3 节的程序代码基础上添加一个增加新用户的 insertUser 方法。由于很可能出现一些无法添加新用户的情况，这些情况需要以异常的形式通知给调用程序，调用程序需要获得发生异常的具体原因，并且这些信息最终要能被 JSP 页面以表单字段名作为参数来检索，所以，insertUser 方法抛出的异常应是一个满足此功能的自定义异常类，这个异常类被命名为 DbUtilException。

例程 9-18 DbUtil.java

```
package org.it315.util;
import org.it315.bean.UserBean;
import java.util.Hashtable;
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
/**DbUtil 类是一个单件类，在整个 Web 应用程序中只能创建 DbUtil 类的一个实例对象，
在 DbUtil 类内部预存储了两个用户记录，并提供了增加和检索用户记录的方法。*/
public class DbUtil
{
    private static DbUtil instance = new DbUtil();
    private Hashtable users = new Hashtable();

    private DbUtil()
    {
        UserBean user1 = new UserBean();
        user1.setName("zxx");
        user1.setPassword("12345678");
        user1.setEmail("zxx@it315.org");
        users.put("zxx",user1);

        UserBean user2 = new UserBean();
        user2.setName("flx");
        user2.setPassword("abcdefg");
        user2.setEmail("flx@it315.org");
        users.put("flx",user2);
    }

    public static DbUtil getInstance()
    {
        return instance;
    }

    public UserBean getUser(String userName)
    {
        UserBean user = (UserBean)users.get(userName);
        return user;
    }

    public void insertUser(UserBean user) throws DbUtilException
    {
        if(user == null)
        {
            return;
        }

        String userName = user.getName();
        if(users.get(userName) != null)
        {
            DbUtilException ex = new DbUtilException();
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
            ex.setErrorMsg("name","username have used.");
            throw ex;
        }
        //在实际应用中有更多的异常消息，例如，E-mail 重复、密码无效等
        users.put(userName,user);
    }
}
```

例程 9-19 DbUtilException.java

```
package org.it315.util;
import java.util.Hashtable;

public class DbUtilException extends Exception
{
    private Hashtable errors = new Hashtable();

    public void setErrorMsg(String err,String errMsg)
    {
        if((err != null) && (errMsg != null))
        {
            errors.put(err,errMsg);
        }
    }

    public String getErrorMsg(String err)
    {
        String err_msg = (String)errors.get(err);
        return (err_msg == null) ? "" : err_msg;
    }
}
```

例程 9-20 register.jsp

```
<%@ page contentType="text/html;GB2312" %>
<%-- 注册失败也回到此页面，此时，FORM 表单中的各个字段应自动设置为
上次输入的内容，并在发生错误的字段旁用红色字体显示出错误的提示信息。
 --%>

<jsp:useBean id="registerForm" class="org.it315.bean.RegisterFormBean"
        scope="request"/>
<form action="../servlet/controller" method="post">
姓名: <input type="text" name="name"
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
    value='<jsp:getProperty name="registerForm" property="name" />'>
    <font color="red"><%=registerForm.getErrorMsg("name")%></font><br>
密码: <input type="password" name="password1"
    value='<jsp:getProperty name="registerForm" property="password1" />'>
    <font color="red"><%=registerForm.getErrorMsg("password1")%></font><br>
确认密码: <input type="password" name="password2"
    value='<jsp:getProperty name="registerForm" property="password2" />'>
    <font color="red"><%=registerForm.getErrorMsg("password2")%></font><br>
Email:<input type="text" name="email"
    value='<jsp:getProperty name="registerForm" property="email" />'>
    <font color="red"><%=registerForm.getErrorMsg("email")%></font><br>
<input type="submit" name="submit" value="注册">
</form>
```

例程 9-21 RegisterFormBean.java

```
package org.it315.bean;
import java.util.*;

/*RegisterFormBean 用于封装注册表单的信息，其中定义了四个属性：name、password1、
password2 和 email。RegisterFormBean 还对这四个属性进行基本的格式验证，如果
有误则将相应的错误信息保存到一个 HashTable 对象中，同时也定义了供 JSP 页面
检索错误信息的方法。*/
public class RegisterFormBean
{
    private String name = "";
    private String password1 = "";
    private String password2 = "";
    private String email = "";
    private Hashtable errors = new Hashtable();

    public void setName(String name)
    {
        this.name = name;
    }
    public String getName()
    {
        return this.name;
    }

    public void setPassword1(String password1)
    {
        this.password1 = password1;
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
    }
    public String getPassword1()
    {
        return this.password1;
    }

    public void setPassword2(String password2)
    {
        this.password2 = password2;
    }
    public String getPassword2()
    {
        return this.password2;
    }

    public void setEmail(String email)
    {
        this.email = email;
    }
    public String getEmail()
    {
        return this.email;
    }

    public boolean validate()
    {
        boolean allOk = true;
        if(name.trim().equals(""))
        {
            errors.put("name","Please input your name.");
            allOk = false;
        }
        if(password1.length()>10 || password1.length()<6)
        {
            errors.put("password1","password must have 6-10 characters.");
            allOk = false;
        }
        if(!password2.equals(password1))
        {
            errors.put("password2","passwords do not match.");
            allOk = false;
        }
        /*对 E-mail 格式的校验采用了正则表达式，关于正则表达式的技术细节，
        请参看笔者编写的《JavaScript 网页开发》一书*/
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        if(!email.matches("[a-zA-Z0-9_-]+@[a-zA-Z0-9_-]+(\\.[a-zA-Z0
        -9_-]+)+"))
        {
            errors.put("email","illegal email.");
            allOk = false;
        }
        return allOk;
    }

    public void setErrorMsg(String err,String errMsg)
    {
        if((err != null) && (errMsg != null))
        {
            errors.put(err,errMsg);
        }
    }

    public String getErrorMsg(String err)
    {
        String err_msg = (String)errors.get(err);
        return (err_msg == null) ? "" : err_msg;
    }
}
```

例程 9-22 ControllerServlet.java

```
package org.it315.servlet;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import org.it315.bean.*;
import org.it315.util.*;

public class ControllerServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
    {
        RequestDispatcher rd = null;

        String name = request.getParameter("name");
        String password1 = request.getParameter("password1");
        String password2 = request.getParameter("password2");
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
        String email = request.getParameter("email");

        RegisterFormBean registerForm = new RegisterFormBean();
        registerForm.setName(name);
        registerForm.setPassword1(password1);
        registerForm.setPassword2(password2);
        registerForm.setEmail(email);
        request.setAttribute("registerForm",registerForm);

        //如果表单字段中填写的内容格式有问题，则跳转到注册页面
        if(!registerForm.validate())
        {
            rd = request.getRequestDispatcher("/jsp/register.jsp");
            rd.forward(request,response);
            return;
        }

        UserBean user = new UserBean();
        user.setName(name);
        user.setPassword(password1);
        user.setEmail(email);
        try
        {
            DbUtil db = DbUtil.getInstance();
            db.insertUser(user);
        }
        catch(DbUtilException ex)
        {
            /*如果用户注册不成功，则取出不成功的原因并跳转到注册
            页面。在实际应用中可能有更多的异常消息，本例子只演示
            名称重复的问题。*/
            registerForm.setErrorMsg("name",ex.getErrorMsg("name"));
            rd = request.getRequestDispatcher("/jsp/register.jsp");
            rd.forward(request,response);
            return;
        }

        /*用户成功注册的同时也自动完成登录，直接进入成功登录后的页面，
        所以，应该将当前注册的用户信息要保存到 Session 域中*/
        HttpSession session = request.getSession();
        session.setAttribute("logonUser",user);
        rd = request.getRequestDispatcher("/jsp/logonSuccess.jsp");
        rd.forward(request,response);
    }
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


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

02 编译上面编写的所有 Java 源文件，确保编译后生成的 class 文件及其包名对应的目录存放在了<tomcat 安装目录>\webapps\it315\Web-INF\classes 目录中。将 register.jsp 文件保存在<tomcat 安装目录>\webapps\it315\jsp 目录中，然后修改 9.4.3 节编写的 logon.jsp 页面，在其中的“登录”按钮旁边增加一个“注册新用户”的超链接，超链接的目标指向 register.jsp 页面，如下所示：

```
...
<input type="submit" name="submit" value="登录">
  <a href="register.jsp">注册新用户</a>
</form>
```

03 在<tomcat 安装目录>\webapps\it315\Web-INF\web.xml 文件中注册 Controller Servlet，并设置其映射 URL。在 web.xml 文件中的相应位置处增加如下两段内容：

```
<servlet>
    <servlet-name>ControllerServlet</servlet-name>
    <servlet-class>org.it315.servlet.ControllerServlet</servlet-class>
</servlet>
.....
<servlet-mapping>
    <servlet-name>ControllerServlet</servlet-name>
    <url-pattern>/servlet/controller</url-pattern>
</servlet-mapping>
```

保存 web.xml 文件后，重新启动 Tomcat。

04 在浏览器地址栏中输入如下地址：

```
http://localhost:8080/it315/jsp/logon.jsp
```

单击返回的 logon.jsp 页面中“注册新用户”超链接，进入 register.jsp 页面，然后分别对注册表单中填写的用户名、密码、E-mail 格式不对以及密码与确认密码不同的情况进行测试，验证程序是否满足了设计需求。

05 在注册表单中填写格式完全正确的信息，如图 9.10 所示。

单击“注册”按钮，浏览器中返回的结果如图 9.11 所示。

06 单击图 9.11 中的“注销”超链接，浏览器中显示出如图 9.12 所示的错误信息。仔细查看错误提示信息，可以发现“注销”超链接所指向的目标地址为“/servlet/logon

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

Success.jsp”，而我们期望访问的是“/jsp/logonSuccess.jsp”，这正是在第 5 章中已经讲过的请求转发的相对路径问题所造成的。

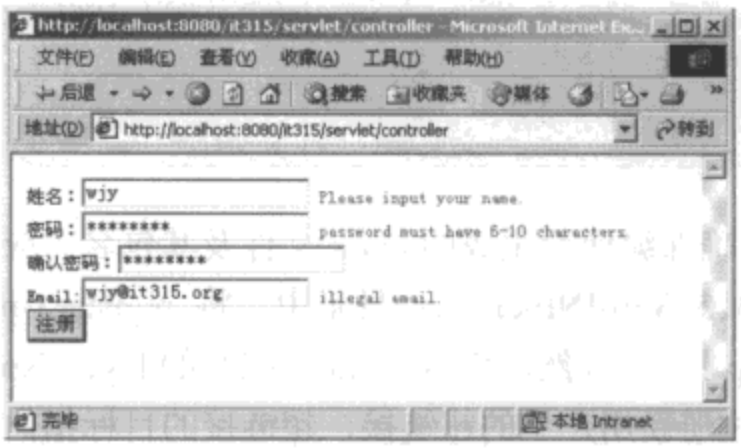


图 9.10

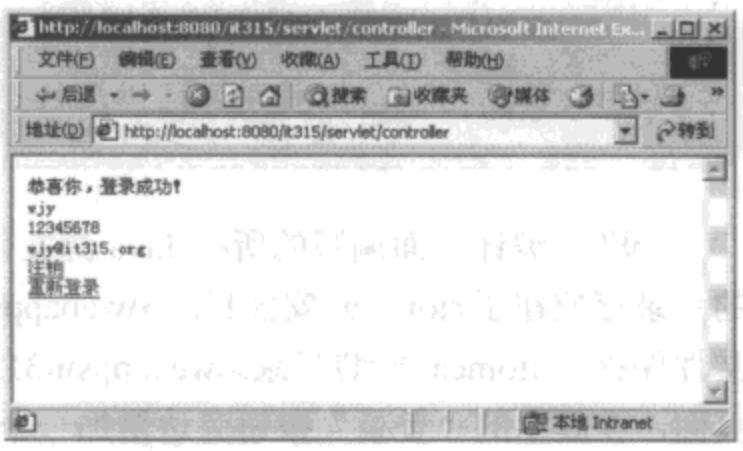


图 9.11

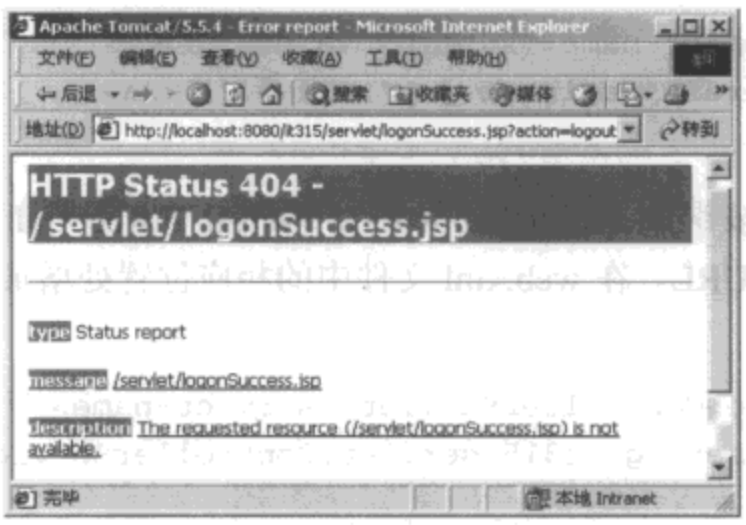


图 9.12

07 修改 logonSuccess.jsp 文件，在其中增加一个 base 标签来解决请求转发时的相对路径问题。

文件清单：logonSuccess.jsp

```
<%@ page contentType="text/html;GB2312" %>

<%- - 用 base 标签解决请求转发时的相对路径问题 - -%>
<%
    String basePath = request.getScheme() + "://"+request.getServerName()
        + ":" + request.getServerPort() + request.getRequestURI();
%>
<base href="<%= basePath%>" />
<%
    //如果 JSP 页面的当前执行过程是对注销行为的响应
    if("logout".equals(request.getParameter("action")))
    {
        session.invalidate();
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
%>
        <jsp:forward page="logon.jsp"/>
<%
    }
    //如果当前会话的用户还没有登录中
    if(session.getAttribute("logonUser") == null)
    {
%>
        <jsp:forward page="logon.jsp" />
<%
    }
%>
<jsp:useBean id="logonUser" class="org.it315.bean.UserBean" scope="session" />
恭喜你，登录成功！<br>
<jsp:getProperty name="logonUser" property="name" /><br>
<jsp:getProperty name="logonUser" property="password" /><br>
<jsp:getProperty name="logonUser" property="email" /><br>
<a href="logonSuccess.jsp?action=logout">注销</a><br>
<a href="logon.jsp">重新登录</a>
```

08 重复第 **05** 步操作，浏览器中显示出如图 9.13 所示的信息。

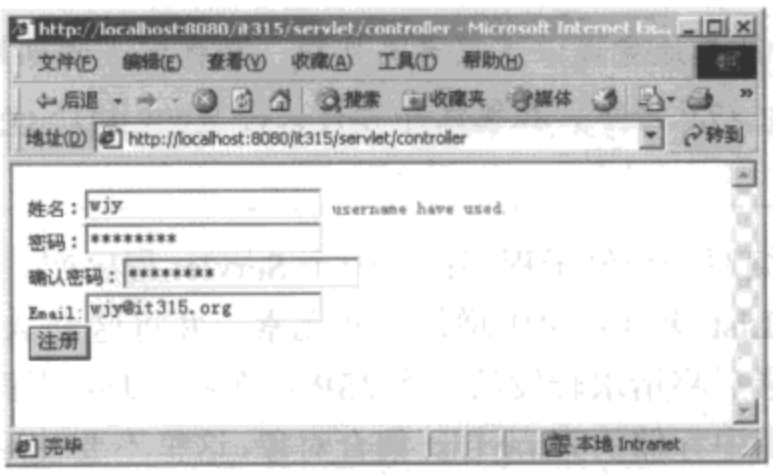


图 9.13

修改注册用户名为“wjy1”，再次单击“注册”按钮，然后在返回的成功登录页面中单击“注销”超链接，这次出现了预期的正常结果。

从上面的例子程序中可以看到，在程序较小的时候，模式二看起来确实要比模式一复杂一些。但是，对于稍微大一点的程序，模式二的优点很快就能显现出来，因为采用模式二除了可以帮助你更清晰地构思整个系统外，更主要的是一旦创建了一个正确的应用后，对其中的某一组件的修改，基本不会影响到其他组件，给系统的维护带来极大便利。特别是如果使用基于 MVC 的 Java Web 开发框架来进行开发时，作为控制器部分的 Servlet 程序的编写也将非常简单，很多工作都可以通过 XML 格式的配置文件来自动完成。如果使

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

用自定义标签和标准标签库以后，JSP 页面的编写也将更加简单，甚至不用包含任何 Java 代码。

在 MVC 模式中应用<jsp:useBean>标签的 type 属性

如果在 JSP 页面中使用<jsp:useBean>标签的 type 属性来指定 JavaBean 所实现的接口，而不使用<jsp:useBean>标签的 class 属性来指定 JavaBean 的具体类名，即使 JavaBean 的类名以后发生了更改，只要它所实现的接口保持不变，JSP 页面就不用随之修改。在 MVC 模式中，JSP 页面引用的 JavaBean 实例对象已经在 Servlet 控制器中生成，在 JSP 页面中不用再创建 JavaBean 实例对象，所以，JSP 页面中应该使用<jsp:useBean>标签的 type 属性来指定 JavaBean 所实现的接口，而不应使用<jsp:useBean>标签的 class 属性来指定 JavaBean 的具体类名，这样更有利于实现 JSP 与 JavaBean 的良好松耦合。当然，要想充分利用<jsp:useBean>标签的 type 属性来实现系统的可扩展性，在设计 JavaBean 时要多做一些工作，即要先定义 JavaBean 的接口，然后让 JavaBean 实现这个接口。

9.5 思考与实践

友情提示：请认真地完成以下思考与实践，并将完成的结果以及操作步骤提交给 www.it315.org 站点评审，表现优异者将获得 www.it315.org 站点的奖励和专家的特别指导。

1. 请按如下要求编写一个例子程序：在一个 Servlet 程序产生一个 ArrayList 集合对象，然后在这个 ArrayList 集合对象中增加一些元素，并将这个 ArrayList 集合对象存储在 request 域范围中，最后将请求转发给一个 JSP 页面；在 JSP 页面中使用<jsp:useBean>标签获取 request 域范围中存储的 ArrayList 集合对象，这里不要使用<jsp:useBean>的 class 属性，而是采用将<jsp:useBean>标签的 type 属性设置为“Collection”的方式，然后将集合对象中的每个元素依次显示出来。

2. 在同一个 jsp 文件中可以有两个 id 属性值相同，但 scope 属性值不同的<jsp:useBean>标签吗？请从<jsp:useBean>标签翻译成 Servlet 源文件中的代码上分析为什么？

3. 使用一段脚本代码产生一个 JavaBean 实例对象并将它存储进一个域范围中，然后使用<jsp:setProperty>和<jsp:getProperty>标签来访问这个 JavaBean 实例对象，如下所示：

```
<%
    java.util.Date d = new java.util.Date();
    pageContext.setAttribute("x",d);
%>
```


http://www.ziwu.org/bbs/?u=39508 更多好书与您分享

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享


```
<jsp:setProperty name="x" property="time" value="1234567"/>
<jsp:getProperty name="x" property="time"/>
```

上面的程序能够正常运行吗？

4. 如果 a.jsp 文件使用 include 指令静态包含了 b.jsp 文件，且 a.jsp 和 b.jsp 使用了相同的<jsp:useBean>标签语句，如下所示：

文件清单：a.jsp

```
<jsp:useBean id="myDate" class="java.util.Date" scope="session"/>
<%@ include file="b.jsp"%>
```

文件清单：b.jsp

```
<jsp:useBean id="myDate" class="java.util.Date" scope="session"/>
```

请问：a.jsp 页面运行时会出现错误吗？为什么？如果 a.jsp 文件改用<jsp:include>标签来动态包含 b.jsp 页面会出现错误吗？为什么？

5. 针对 9.4.3 节的 logonSuccess.jsp 程序，能否使用下面的代码来将未登录用户的请求转发给 logon.jsp 页面？

提示

下面的代码可以在 Session 域范围中不存在名为 logonUser 的 JavaBean 实例对象时执行<jsp:forward>语句。

```
<jsp:useBean id="logonUser"
        class="org.it315.bean.UserBean" scope="session" >
    <jsp:forward page="logon.jsp"/>
</jsp:useBean>
```

6. 请采用 JSP 开发模式二实现 9.4.3 节的用户登录程序，要求在登录表单中加上一次性验证码。

http://www.ziwu.org/bbs/?u=39508 更多好书与您分享