Servlet

1 Servlet概念

Servlet=Server+applet，是运行在服务器端的 Java 应用程序，用于创建动态 Web 应用。

Servlet 部署、运行在 Web 服务器中，由服务器加载、调用和管理。

Servlet 响应用户请求，为用户提供服务。用户可以在浏览器中像访问 Web 页面一样对 Servlet 进行访问。

Servlet 的功能：读取客户端信息，生成响应结果，发送信息给客户端。

Servlet 的工作过程如图 1.1 所示。

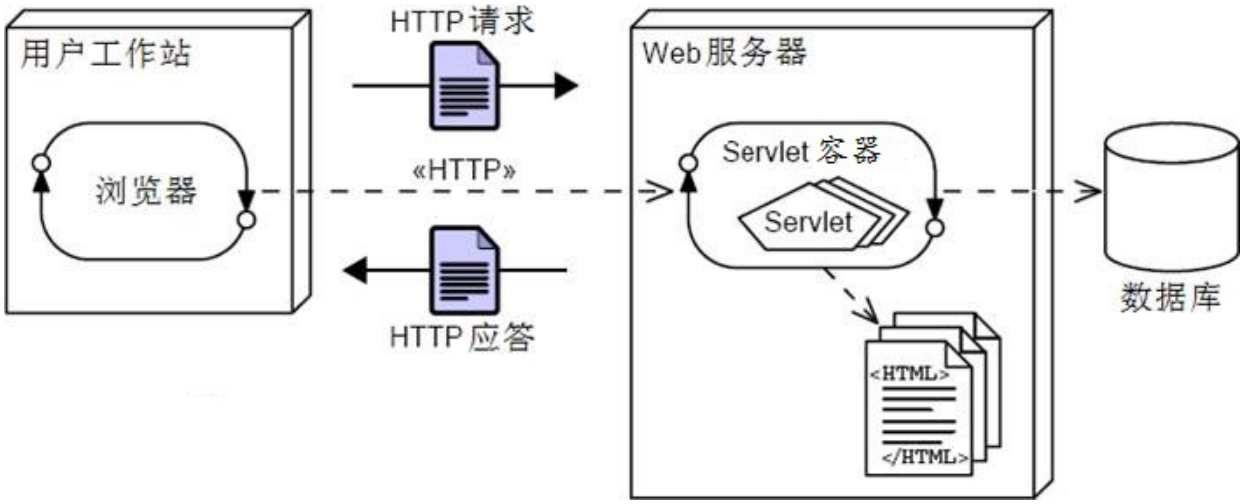

图 1.1　Servlet 的工作过程

Servlet 示例：

```
package javaee.servlet;
import java.io.*;
import javax.servlet.*;
import javax.servlet.annotation.*;
import javax.servlet.http.*;

// 标注，这是一个 Servlet，访问路径为 "/HelloWorld"
@WebServlet("/HelloWorld")
public class HelloWorldServlet extends HttpServlet // 继承 HttpServlet 抽象类
{
    // 处理浏览器发送来的 HTTP GET 类型的请求
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
        throws ServletException, IOException
    {
        response.setContentType("text/html; charset=UTF-8");
        // 获得输出对象
        PrintWriter out = response.getWriter();
        // 向请求端输出信息
        out.println("Hello World. It is HelloWorldServlet.<br>");
    }
}
```

图 1.2 展示了 Servlet 的访问路径。

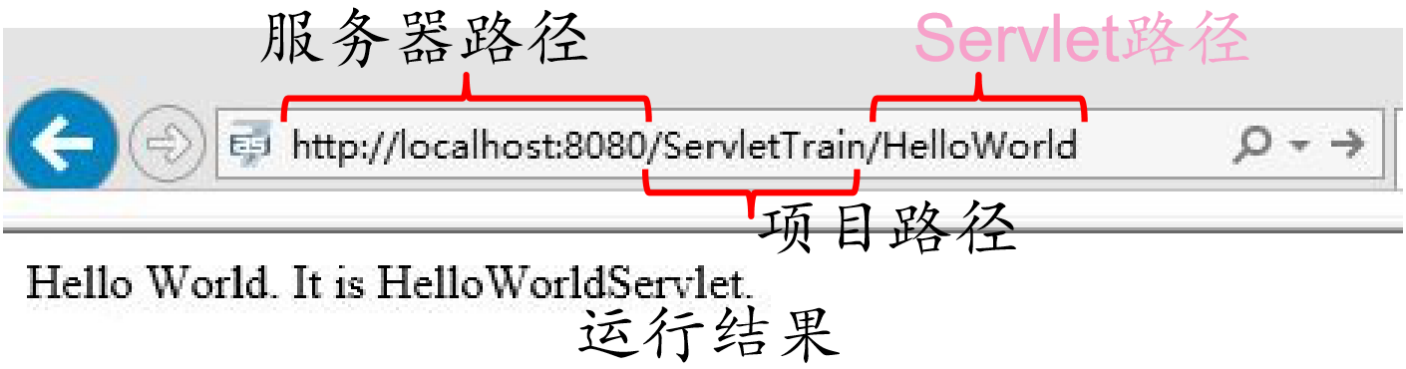

图 1.2　Servlet 的访问路径

Servlet 的优点：可以使用 Java 语言的所有特性，灵活、方便。

Servlet 的缺点：使用 Java 代码打印出 Web 形式的输出结果繁杂，阅读困难。

2 Servlet的工作原理

Servlet 的工作原理如图 2.1 所示。

图 2.1　Servlet 的工作原理

Servlet 的调用过程如图 2.2 所示。

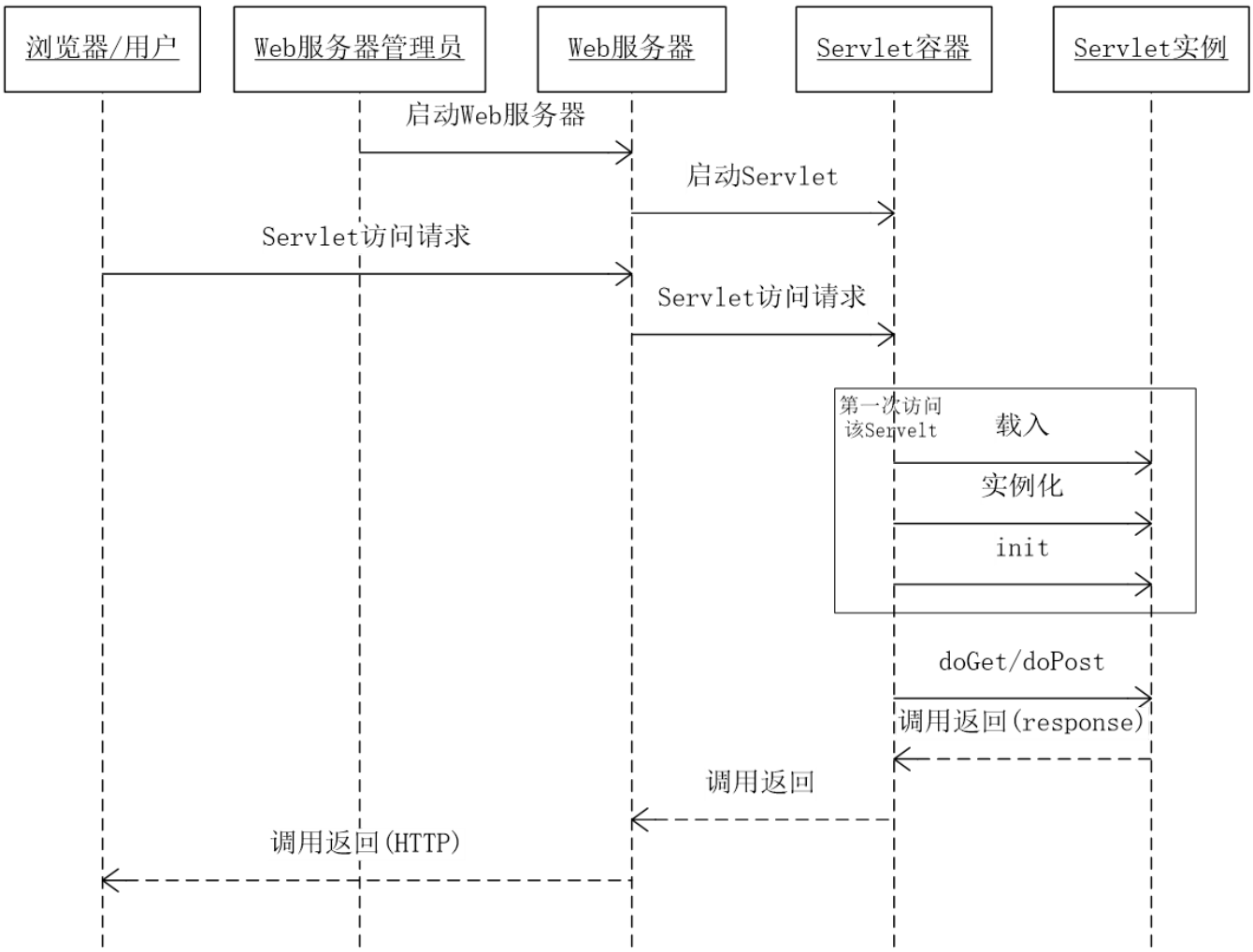

图 2.2　Servlet 的调用过程

Servlet 的运行状态完全由 Servlet 容器维护和管理，一个 Servlet 的生命周期一般有初始化、提供服务和销毁三个阶段。Servlet 的生命周期如图 2.3 所示。

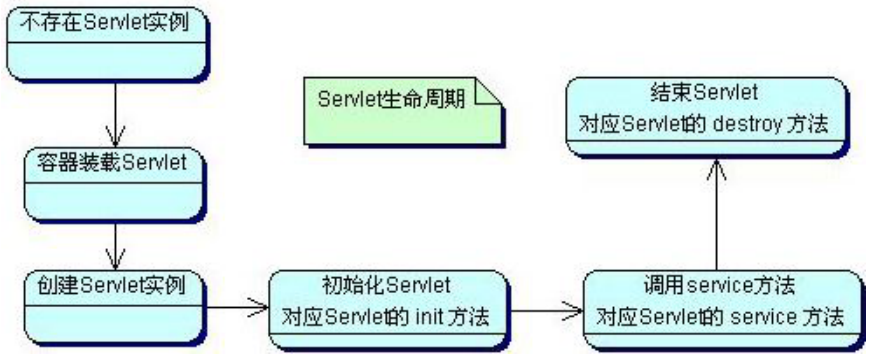

图 2.3　Servlet 的生命周期

Servlet 实质上是以单例的形式被实现的，它在被初始化之后将一直在内存中活动，后续的客户请求 Servlet 服务时将不再创建该 Servlet 的新的对象，只是新建一个线程调用 `service()` 方法提供服务。

3 Servlet开发过程

Servlet 的开发过程包括以下五个步骤：

1. 创建工程。
2. 创建 Servlet 类。
3. 配置 Servlet 类。
4. 发布 Servlet 类。
5. 调用 Servlet 类。

3.1 创建Servlet类

在 Web 应用中如果没有特殊需要，一般开发者定义的 Servlet 类都扩展`HttpServlet`类。

`HttpServlet`类中最为常用的是`doGet()`和`doPost()`方法，在其中加入对客户端请求的相应处理逻辑。

`HttpServletRequest`类的对象用来封装客户端提交的请求，是输入参数。`HttpServletResponse`类的对象将处理结果进行封装，用来向客户端返回，是输出参数。

3.2 配置Servlet类

配置 Servlet 的方法有两种：一种是使用标注的方式进行配置，一种是使用 XML 文件进行配置。

3.2.1 标注配置

Servlet 3.0 中加入了新增的标注支持，用于简化 Servlet 的配置。通过在 Servlet 类文件中类定义之前的位置添加标注信息来实现，类的其它部分不变。

例如，在上面的例子中使用了如下的标注：

```
@WebServlet("/HelloWorld")
```

`@WebServlet`是这个标注的名称，括号中的是这个标注的属性说明。这行标注的作用有两个，第一说明这个类是一个 Servlet，第二说明了这个 Servlet 的访问路径。

3.2.2 XML文件配置

传统配置方法是使用 Web 应用中 WEB-INF 文件夹下的 web.xml 部署描述文件来进行。需要在 web.xml 文件的`<webapp></webapp>`元素之间添加如下代码：

```
<webapp>
    <!-- 说明 Servlet 类 -->
    <servlet>
        <!-- Servlet 类的别名 -->
        <servlet-name>helloWorldServlet</servlet-name>
        <!-- Servlet 类的全名 -->
        <servlet-class>javaee.servlet.HelloWorldServlet</servlet-class>
    </servlet>
    <!-- 说明 Servlet 类与访问地址的映射 -->
    <servlet-mapping>
        <!-- Servlet 类的别名，与 servlet 元素中定义的别名保持一致 -->
        <servlet-name>helloWorldServlet</servlet-name>
        <!-- 访问 Servlet 类的地址 -->
        <url-pattern>/HelloWorld</url-pattern>
    </servlet-mapping>
</webapp>
```

3.3 调用Servlet类

调用 Servlet 类的基本方法有两种：

1. 直接使用 Servlet 的 URL 对 Servlet 进行访问，这种情况包括使用 HTML 链接、JSP 跳转，或者浏览器地址栏中输入地址等方式。这种方式 Web 容器将调用 Servlet 的`doGet()`方法为请求提供服务。
2. 在表单中设置提交目标为 Servlet 的 URL，这种方式 Web 容器将调用 Servlet 的`doPost()`方法为请求提供服务。

4 Servlet主要接口和类

Java Servlet 包括两个基本的包，`javax.servlet`和`javax.servlet.http`，主要的接口和类都放在这两个包中。主要接口和类的关系如图 4.1 所示。

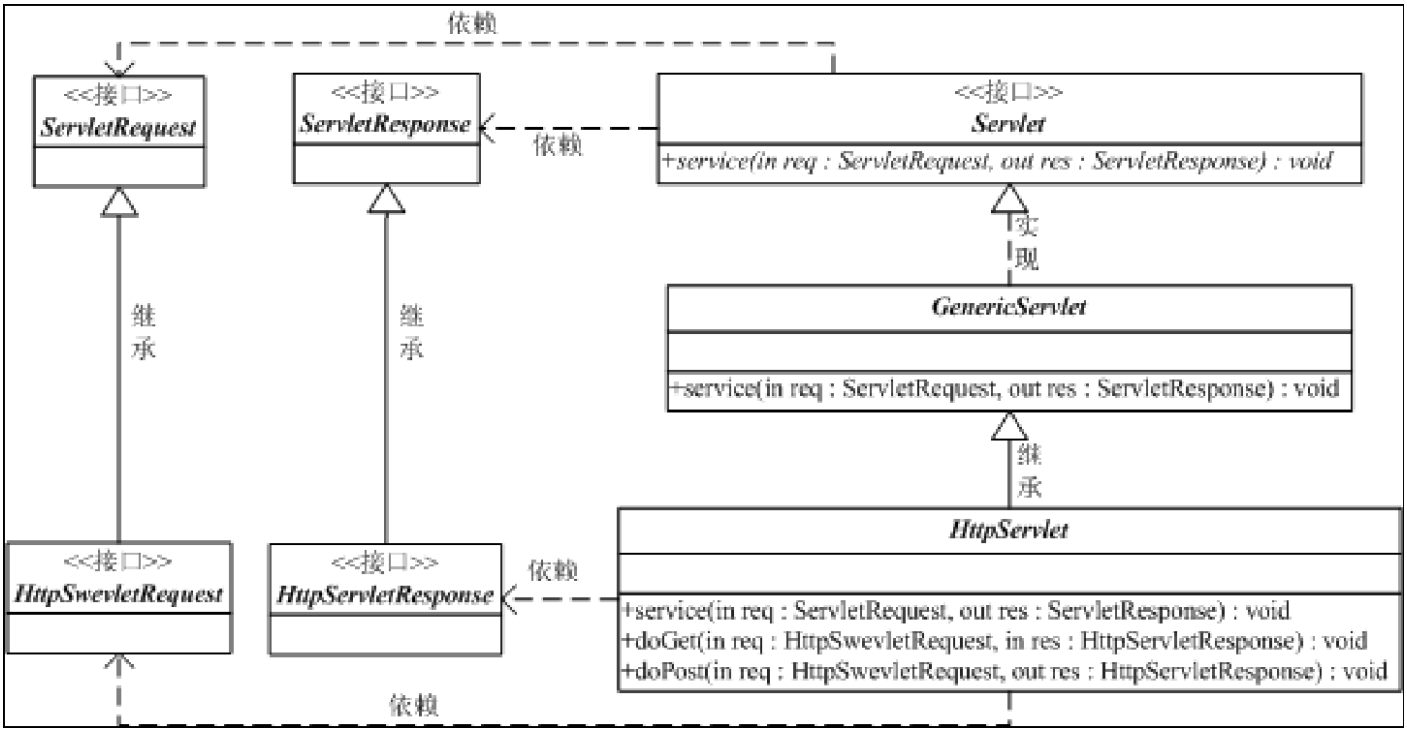

图 4.1　Servlet 主要接口和类

4.1 `Servlet` 接口

`Servlet` 接口定义在 `javax.servlet` 包中，所有的 Servlet 类必须实现 Servlet 接口。该接口中定义了 Servlet 生命周期的相关方法。

`Servlet` 接口中定义的主要方法如下表所示。

方法签名	方法说明
`void init(ServletConfig config)`	对 Servlet 进行初始化，在 Servlet 的生命周期中只执行一次。Servlet 引擎会在 Servlet 实例化之后、置入服务之前，调用 `init()` 方法。
`void service(ServletRequest, ServletResponse)`	`service()` 方法是 Servlet 的核心。 当客户端发送请求时，Servlet 容器将会调用此方法，并传递一个请求（`ServletRequest`）对象和一个响应（`ServletResponse`）对象作为参数。 在调用 `service()` 方法之前，`init()` 方法必须成功退出。
`void destroy()`	用于让 Servlet 释放它正在使用的所有资源、保存所有需要持久化的状态。在 Servlet 的生命周期中只执行一次。当 Servlet 容器判断某个 Servlet 不再提供服务时，将调用其 `destroy()` 方法。 需要确定在调用 `destroy()` 方法时，Servlet 的所有线程已终止或完成。
`ServletConfig getServletConfig()`	返回一个 `ServletConfig` 对象，使用该对象用来获得 Servlet 初始化参数和上下文环境信息（`ServletContext` 对象）。
`String getServletInfo()`	可选方法。它提供一个有关 Servlet 的信息字符串，如作者、版本、版权等。

4.2 `ServletRequest` 接口

`ServletRequest` 接口定义在 `javax.servlet` 包中，用来获取客户端发来的请求数据。

`ServletRequest` 接口中的主要方法如下表所示。

方法签名	方法说明
`Object getAttribute(String name)`	返回请求中指定属性的值

方法签名	方法说明
`Enumeration getAttributeNames()`	返回包含在这个请求中的所有属性的属性名
`String getCharacterEncoding()`	返回请求中输入内容的字符编码类型
`String getContentType()`	返回请求的 MIME 类型
`int getContentLength()`	返回请求内容的长度
`ServletInputStream getInputStream()`	返回一个用来从请求中读取二进制数据的输入流
`String getParameter(String name)`	返回指定参数的参数值
`Enumeration getParameterNames()`	返回所有输入参数的参数名
`String[] getParameterValues(String name)`	返回指定参数的所有参数值
`BufferReader getReader()`	返回一个`BufferReader`用来读取请求的实体， 其编码方式依照请求数据的编码方式
`String getProtocol()`	返回这个请求所采用的协议
`String getRemoteAddr()`	返回请求发送端的 IP 地址
`String getRemoteHost()`	返回请求发送端的主机名称
`String getServerName()`	返回接收请求的服务器主机名
`int getServerPort()`	返回接收请求的端口号
`void setAttribute(String name, Object object)`	在请求中添加一个属性
`void removeAttribute(String name)`	在请求中删除指定的属性
`String getScheme()`	返回请求所使用的 URL 模式

4.3 `ServletResponse`接口

`ServletResponse`接口定义在`javax.servlet`包中，用来响应客户端请求。

`ServletResponse`接口常用的方法如下表所示。

方法签名	方法说明
`String getCharacterEncoding()`	返回响应实体的字符编码
`ServletOutputStream getOutputStream()`	返回一个二进制的响应数据的输出流
`PrintWriter getWriter()`	返回一个`PrintWriter`对象，用来记录格式化的响应实体
`void setContentLength(int length)`	设置响应的内容长度
`void setContentType(String type)`	设定响应的内容类型

方法签名	方法说明
`void setCharacterEncoding(String charset)`	设置响应实体的字符编码

4.4 `GenericServlet` 抽象类

`GenericServlet` 抽象类实现了 `Servlet` 接口，能够处理一般的 Servlet 请求，是一种通用的 Servlet 基类，与应用层协议无关。开发者通过继承该类，可以实现支持 FTP、HTTP 等多种协议的 Servlet 类。

4.5 `HttpServlet` 抽象类

`HttpServlet` 抽象类是 `GenericServlet` 的子类，主要应用于 HTTP 协议的请求和响应。在基于 HTTP 的 Web 应用中所使用的 Servlet 绝大多数都是由该类派生的。

`HttpServlet` 抽象类中常用的方法如下表所示。

方法签名	方法说明
`void doDelete(HttpServleteRequest, HttpServletResponse)`	被 `service()` 方法调用，用来处理 HTTP DELETE 操作
`void doGet(HttpServleteRequest, HttpServletResponse)`	被 `service()` 方法调用，用来处理 HTTP GET 操作
`void doPost(HttpServleteRequest, HttpServletResponse)`	被 `service()` 方法调用，用来处理 HTTP POST 操作
`void doPut(HttpServleteRequest, HttpServletResponse)`	被 `service()` 方法调用，用来处理 HTTP PUT 操作
`void doHead(HttpServleteRequest, HttpServletResponse)`	被 `service()` 方法调用，用来处理 HTTP HEAD 操作
`void doOptions(HttpServleteRequest, HttpServletResponse)`	被 `service()` 方法调用，用来处理 HTTP OPTION 操作
`void doTrace(HttpServleteRequest, HttpServletResponse)`	被 `service()` 方法调用，用来处理 HTTP TRACE 操作

4.6 `HttpSerlvetRequest` 和 `HttpServletResponse` 接口

`HttpSerlvetRequest` 和 `HttpServletResponse` 接口分别继承自 `ServletRequest` 和 `ServletResponse` 接口，并基于 HTTP 协议“升级改造”而来。与 `ServletRequest` 和 `ServletResponse` 接口相比，这两个接口多出的方法主要和 HTTP 协议相关。

`HttpSerlvetRequest` 接口定义的主要方法如下表所示。

方法签名	方法说明
`Cookie[] getCookies()`	返回一个数组，该数组包含这个请求中当前的所有 Cookie
`String getHeader(String name)`	返回指定名字的请求报头的值
`String getMethod()`	返回此次请求所使用的 HTTP 方法的名字，如 GET、POST 等
`String getQueryString()`	返回这个请求 URL 所包含的查询字符串（由`?`引出的部分），如果没有查询字符串则返回空值
`String getRequestURI()`	返回 URL 中被请求的资源
`HttpSession getSession(boolean create)`	返回与这个请求关联的、当前有效的 session。如果没有 session 与这个请求关联，根据参数不同采取不同措施： （1）如果没有参数，则新建一个 session； （2）如果参数为`true`，则新建一个 session； （3）如果参数为`false`，则不新建 session。

`HttpServletResponse`接口的主要方法如下表所示。

方法签名	方法说明
`void addCookie(Cookie cookie)`	在响应中添加一个 Cookie。 该方法应该在响应被提交之前调用。
`void sendRedirect(String location)`	使用给定的路径，给客户端发出一个临时转向的响应（`SC_MOVED_TEMPORARILY`）。给定的路径必须是绝对 URL。 该方法应该在响应被提交之前调用。调用这个方法后，响应立即被提交，Servlet 不会再有更多输出。
`void setHeader(String name, String value)`	使用一个给定的名称和域设置响应头。 如果响应头已经被设置，则新的值将覆盖当前值。
`void setStatus(int statusCode)`	设置响应的状态码。如果状态码已经被设置，则新的值将覆盖当前值。
`String encodeRedirectURL(String url)`	对`sendRedirect()`方法使用的 URL 进行编码。
`String encodeURL(String url)`	对包含 session ID 的 URL 进行编码

4.7 `ServletContext`接口

`ServletContext`是 Servlet 与 Servlet 容器之间直接进行通信的接口。Servlet 容器在创建 Web 应用时，会为它创建一个实现了`ServletContext`接口的对象，每个 Web 应用有唯一的`ServletContext`对象。

`ServletContext`接口提供的方法分为以下几种类型：

1. 用于在 Web 应用范围内存取共享数据的方法：`setAttribute()`、`getAttribute()`、`removeAttribute()`

2. 访问当前 Web 应用的资源：`getInitParameter()`、`getRequestDispatcher()`
3. 访问 Servlet 容器的相关信息：`getContext()`
4. 访问 Web 容器的相关信息：`getServerInfo()`
5. 访问服务器端的文件系统资源：`getRealPath()`、`getResource()`
6. 输出日志：`log()`

4.8 `ServletConfig` 接口

`ServletConfig` 接口用来获取 Servlet 的配置信息。每一个 `ServletConfig` 对象对应着唯一的一个 Servlet。Servlet 容器在调用 `init()` 方法时，把 `ServletConfig` 对象当做参数传递给 Servlet 对象。

`getInitParameter()` 和 `getServletContext()` 方法用于获取配置信息。

4.9 `HttpSession` 接口

通过 `HttpSession` 接口，Servlet 引擎可以有效地跟踪用户的会话。同时可以通过绑定对象到 session 中实现数据共享。

`HttpSession` 接口的主要方法有：`getId()`、`getValue()`、`putValue()`、`removeValue()`、`setMaxInactiveInterval()`、`getMaxInactiveInterval()`。

4.10 `Cookie` 类

Cookie 是一个由服务器端生成，发送给客户端保存和使用的 name-value 对，用于服务器对客户端的识别。

4.11 `RequestDispatcher` 接口

`RequestDispatcher` 接口用于将请求发送到服务器上的任何资源（比如 Servlet、HTML 文件或 JSP 文件）的对象。

定义了两种方法用于请求转发：

1. forward：转发，将当前 request 和 response 对象保存，交给指定的 url 处理。
2. include：包含，用当前 Servlet 的 request 和 respose 来执行 url 中的内容处理业务，并将执行结果包含进当前的 Servlet 当中来。

4.12 关于路径

Servlet 中用到的路径有以下四种：

1. 链接地址：`<a href="url"></a>`
2. 表单提交：`<form action="url">`
3. 重定向：`response.sendRedirect("url");`
4. 转发：`getRequestDispatcher("url");`

路径的两种表示方法：

1. 不以“/”开头的路径，如`<a href="del.do"></a>`，叫做**相对路径**，以当前路径为参照，进行跳转。
2. 以“/”开头的路径，如`<a href="/appname/del.do"></a>`，叫做**绝对路径**，以应用的根目录为参照，进行跳转。在链接地址、表单提交、重定向中，绝对路径从应用名开始写，而转发要从应用名之后开始写。

4.13 Servlet共享变量

在 Servlet 中进行变量的共享可以通过 Servlet 容器中存在的`ServletContext`、`HttpSession`和`HttpServletRequest`的实例来实现。这几种方式共享变量和获得变量的方法都是一致的，只是在变量的作用域，也就是共享的范围上有所不同，如图 4.2 所示。

图 4.2　共享变量的作用域

`ServletContext`范围最大，应用程序级别，整个应用程序都能访问；`HttpSession`次之，会话级别，在当前的浏览器中都能访问；`HttpServletRequest`范围最小，请求级别，变量的作用域随着请求的结束而结束。

这三种方式共享变量的方法是使用 Context、Session 或 Request 类型的实例调用`setAttribute(varName, obj)`方法将需要共享的变量存储到对象当中，然后在需要使用该共享变量的地方通过实例的`getAttribute(varName)`方法来获得变量。

在同一个页面中使用共享变量，存储共享变量时使用的`ServletContext`、`HttpSession`和`HttpServletRequest`实例与读取共享变量时使用的`ServletContext`、`HttpSession`和`HttpServletRequest`实例是同一个，所以能够获得存储的共享变量的值。例如：

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    // 1、使用 ServletContext 共享变量
    ServletContext sc = this.getServletContext();
    sc.setAttribute("sharingvar_sc", "context");

    // 2、使用 HttpSession 共享变量
    HttpSession session = request.getSession();
    session.setAttribute("sharingvar_se", "session");

    // 3、使用 HttpServletRequest 共享变量
    request.setAttribute("sharingvar_req", "request");

    // 4、在同一个页面中读取共享变量
    String sc_value = (String) sc.getAttribute("sharingvar_sc");
    String session_value = (String) session.getAttribute("sharingvar_se");
    String request_value = (String) request.getAttribute("sharingvar_req");

    // 5、在对客户端的响应中显示读取的共享变量值
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    out.println("<!DOCTYPE html>");
    out.println("<html>");
    out.println("<head>");
    out.println("    <meta charset=\"UTF-8\">");
    out.println("    <title>A Servlet</title>");
    out.println("</head>");
    out.println("<body>");
    out.println(    "在同一个页面中读取共享变量<br>");
    out.println(    "Context:" + sc_value  + "<br>"); // Context:context
    out.println(    "Session:" + session_value + "<br>"); // Session:session
    out.println(    "Request:" + request_value + "<br>"); // Request:request
    out.println("</body>");
    out.println("</html>");
    out.flush();
    out.close();
}
```

如果通过重定向函数、超链接或者在浏览器中输入页面地址的方式跳转到新的页面，可以读取到`ServletContext`和`HttpSession`中的共享变量的值，而`HttpServletRequest`不能。

为了在其他页面读取到`HttpServletRequest`中的共享变量，需要将这个`HttpServletRequest`对象转发到相应的页面。例如：

```java
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    // 1、使用 ServletContext 共享变量
    ServletContext sc = this.getServletContext();
    sc.setAttribute("sharingvar_sc", "context");

    // 2、使用 HttpSession 共享变量
    HttpSession session = request.getSession();
    session.setAttribute("sharingvar_se", "session");

    // 3、使用 HttpServletRequest 共享变量
    request.setAttribute("sharingvar_req", "request");

    // 4、转发
    request.getRequestDispatcher("/GetSharingVars").forward(request, response);
}
```

重启浏览器后跳转到新页面，`HttpSession`中的共享变量也无法读取。因为重新启动了浏览器，将会启用一个新的会话，`HttpSession`实例随之发生了改变。

5 Servlet文件操作

Servlet 类实质上就是一个继承了`HttpServlet`的 Java 类。该类由 Web 容器进行实例化，运行在服务器上。因此在 Servlet 类中可以使用 Java 提供的输入/输出流来完成对服务器上文件的读取和写入。

用来读写文件的 Servlet 类和其它普通 Servlet 类没有区别，客户端的请求会被`doPost()`或者`doGet()`方法响应，在请求中一般会包含读取/写入文件的文件路径。Servlet 类在`doPost()`或者`doGet()`中获得客户端请求的信息，然后根据这些信息完成对文件的读写操作。

5.1 读文件

在 Servlet 中读取文件的操作和步骤与在普通 Java 程序中读取文件相同，一般利用`File`、`FileReader`和`BufferedReader`类的组合来完成。例如，下面的代码将服务器上当前运行项目中的 /WEB-INF/web.xml 文件读取出来，并输出到页面中进行显示。

```java
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    // 设置页面的文档类型和字符集，页面中的字符所采用的字符集
    response.setContentType("text/html;charset=UTF-8");
    // 设置页面的编码方式，即以什么样的编码方式来保存页面文件
    response.setCharacterEncoding("UTF-8");
    // 从 response 中获得 PrintWriter 类的对象，用于向页面输出信息
    PrintWriter out = response.getWriter();
    // 向页面输出信息
    out.println("HTML文件头......");

    // 文件相对路径
    String fileName = "/WEB-INF/web.xml";
    // 文件绝对路径
    String filePath = this.getServletContext().getRealPath(fileName);
    // 使用文件的绝对路经打开文件
    File file = new File(filePath);
    if (file.exists())
    {
        // 使用打开的文件对象，创建 FileReader 类的实例
        FileReader reader = new FileReader(file);
        // 使用打开文件对应的 reader 对象，创建 BufferedReader 类的实例
        BufferedReader bufferReader = new BufferedReader(reader);
        String line = null;
        // 逐行读取文件，并输出到页面上
        while((line = bufferReader.readLine())!=null)
        {
            out.println(line);
        }
        bufferReader.close();
    }
    else
    {
        out.println("未找到文件! ");
    }
    out.println("HTML文件尾......");
    out.flush();
    out.close();
}
```

5.2 写文件

在 Servlet 中写文件的方法和步骤也是和普通的 Java 程序一致的，通常使用`File`、`FileWriter`和`BufferedWriter`的组合来完成。例如，下面的代码在服务器上运行的当前项目的根目录下创建一个名为 temp.txt 的文件，并向其中写入信息。

```java
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    response.setContentType("text/html;charset=UTF-8");
    response.setCharacterEncoding("UTF-8");
    // 从 response 中获得 PrintWriter 类的对象，以用于向页面输出信息
    PrintWriter out = response.getWriter();

    // 文件相对路径
    String fileName = "temp.txt";
    // 文件绝对路径
    String filePath = this.getServletContext().getRealPath(fileName);
    // 使用文件的绝对路经打开文件，如果文件不存在则创建文件
    File file = new File(filePath);
    // 使用打开的文件对象，创建 FileWriter 类的实例
    FileWriter writer = new FileWriter(file);
    // 使用打开文件对应的 writer 对象，创建 BufferedWriter 类的实例
    BufferedWriter bufferWriter = new BufferedWriter(writer);
    // 通过 BufferedWriter 类的实例，向文件中写入信息
    bufferWriter.write("Servlet写文件");
    // 刷新缓存，将缓存中的内容写入到文件中
    bufferWriter.flush();
    bufferWriter.close();
    writer.close();

    out.print("文件写入完毕，路径:" + file.getAbsolutePath());
}
```

5.3 上传文件

在 Servlet 3.0 之前，Servlet 本身没有提供对文件上传的直接支持，需要使用第三方框架来实现，而且使用起来也不够简单。

Servlet 3.0 提供了对文件上传的支持，通过 `@MultipartConfig` 标注和 `HttpServletRequest` 接口的两个新方法 `getPart()` 和 `getParts()`，开发者能够很容易地实现文件上传操作。

`@MultipartConfig` 标注写在 Servlet 类的声明之前，以表示该 Servlet 希望处理的请求是 `multipart/form-data` 类型的。另外，该标注还提供了若干属性用于简化对上传文件的处理，如下表所示。

属性名	类型	可选	描述
`fileSizeThreshold`	`int`	是	当数据量大于该值时，内容将被写入文件
`location`	`String`	是	存放生成的文件地址
`maxFileSize`	`long`	是	允许上传的文件的最大长度。 默认值为 -1，表示没有限制。
`maxRequestSize`	`long`	是	请求的最大数量。 默认值为 -1，表示没有限制。

`HttpServletRequest` 接口提供的两个新方法如下所示：

1. `Part getPart(String name)`：用于获取给定名字的文件。

2. `Collection<Part> getParts()`：用于获取所有的文件。

注意：如果请求的 MIME 类型不是`multipart/form-data`，则不能使用上面的两个方法，否则将抛出异常。

每一个文件用一个`javax.servlet.http.Part`对象来表示。该接口提供了处理文件的简易方法，比如`write()`、`delete()`等。

Servlet 上传文件的示例代码如下所示：

```
<!DOCTYPE html>
<html>
    <head>
        <meta content="text/html; charset=UTF-8">
        <title>Title</title>
    </head>
    <body>
        <form action="/ServletTrain/Upload" method="post"
              enctype="multipart/form-data" target="workspace">
            <input type="file" name="file">
            <input type="submit" name="upload" value="文件上传">
        </form>
    </body>
</html>
```

```
@WebServlet("/Upload")
@MultipartConfig
public class Upload entends HttpServlet
{
    // 构造器和 doGet 方法省略

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        request.setCharacterEncoding("utf-8");
        // Servlet 3.0 中新引入的方法，用来处理 multipart/form-data 类型的表单
        Part part = request.getPart("file");
        // 获取 HTTP 头信息
        String headerInfo = part.getHeader("content-disposition");
        // 从 HTTP 头信息中获取文件名
        String fileName = headerInfo.substring(headerInfo.lastIndexOf("\\") + 1,
                                               headerInfo.length() - 1);
        // 获得存储上传文件的完整路径（文件夹路径+文件名）
        // 文件夹位置固定，文件名采用与上传文件的原始名字相同
        String fileSavingFolder = this.getServletContext().getRealPath("/UpLoad_3.0");
        String fileSavingPath = fileSavingFolder + File.separator + fileName;
        // 如果存储上传文件的文件夹不存在，则创建文件夹
        File f = new File(fileSavingFolder + File.separator);
        if (!f.exists()) f.mkdirs();
        // 将上传的文件内容写入服务器文件中
        part.write(fileSavingPath);
        // 输出上传成功信息
        OutPut.outputToClient("文件已上传到：" + fileSavingPath, response);
    }
}
```

5.4 下载文件

Servlet 实现文件下载是通过对响应对象 `response` 的操作来完成的。首先需要在 `response` 中对需要下载的文件类型、头信息、文件长度等信息进行设置，然后读取服务器上文件的内容并写入到 `response` 的输出流中。例如：

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    try
    {
        // 服务器上文件的相对路径
        String filepath = "/WEB-INF/web.xml";
        // 服务器上文件的绝对路径
        String fullFilePath = getServletContext().getRealPath(filepath);
        // 打开文件，创建文件对象
        File file = new File(fullFilePath);
        if (file.exists()) // 如果文件存在
        {
            // 获得文件名，并采用 UTF-8 编码方式进行编码，以解决中文问题
            String filename = URLEncoder.encode(file.getName(), "UTF-8");
            // 重置 response 对象
            response.reset();
            // 设置文件的类型，xml 文件采用 text/xml 类型，详见 MIME 类型说明
            response.setContentType("text/xml");
            // 设置 HTTP 头信息中内容
            response.addHeader("Content-Disposition",
                               "attachment; filename=\"" + filename + "\"");
            // 设置文件长度
            int fileLength = (int) file.length();
            response.setContentLength(fileLength);
            if (fileLength > 0) // 如果文件长度大于 0
            {
                // 创建输入流
                InputStream inStream = new FileInputStream(file);
                byte[] buf = new byte[4096];
                // 创建输出流
                ServletOutputStream servletOS = response.getOutputStream();
                int readLength;
                // 读取文件内容并写入到 response 的输出流当中
                while ((readLength = inStream.read(buf)) != -1)
                {
                    servletOS.write(buf, 0, readLength);
                }
                inStream.close(); // 关闭输入流
                servletOS.flush(); // 刷新输出缓冲
                servletOS.close(); // 关闭输出流
            }
        }
        else
        {
            System.out.println("文件不存在");
            PrintWriter out = response.getWriter();
            out.println("文件 \"" + fullFilePath + "\" 不存在");
        }
    }
    catch (Exception e)
    {
        System.out.println(e);
    }
}
```

6 过滤器

6.1 过滤器的基本概念

过滤器是在 Servlet 2.3 之后增加的新功能。当需要限制用户访问某些资源或者在处理请求时提前处理某些资源的时候，就可以使用过滤器。

过滤器是以一种组件的形式绑定到 Web 应用程序中的。与其他的 Web 应用程序组件不同的是，过滤器采用了“链”的方式进行处理，如图 6.1 所示。

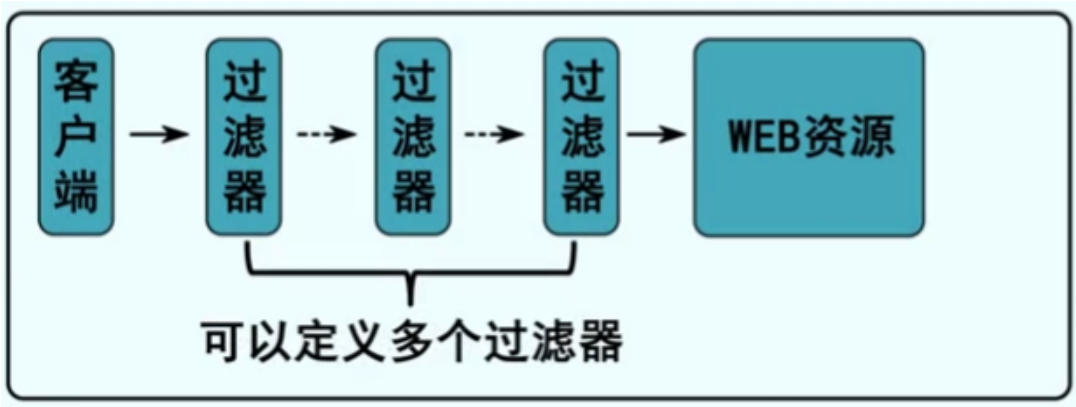

图 6.1　过滤器链

6.2 过滤器的执行流程

过滤器的执行流程如图 6.2 所示。

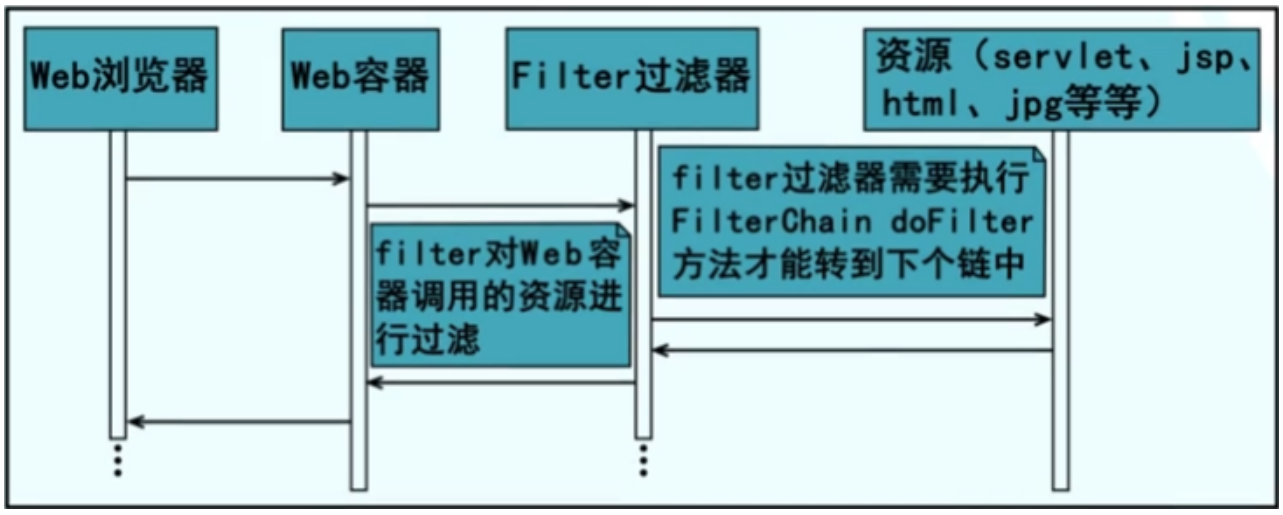

图 6.2　过滤器的执行流程

在 `HttpServletRequest` 到达 Servlet 之前，过滤器拦截客户的 `HttpServletRequest`，根据需要检查 `HttpServletRequest`，也可以修改 `HttpServletRequest` 头和数据。在 `HttpServletResponse` 到达客户端之前，过滤器拦截 `HttpServletResponse`，根据需要检查 `HttpServletResponse`，也可以修改 `HttpServletResponse` 头和数据。

6.3 过滤器的实现

过滤器的实现：

1. 定义过滤器，让一个类实现 `javax.servlet.Filter` 接口。
2. 配置过滤器，在 web.xml 文件中添加配置。

`Filter` 接口定义了三个方法：

```
// 初始化
public void init(FilterConfig) throws ServletException

// 执行过滤操作
public void doFilter(ServletRequest, ServletResponse, FilterChain)
    throws IOException, ServletException

// 销毁过滤器
public void destroy()
```

例如，下面的代码实现了字符编码过滤的功能。

```
package jlujee.servlet.filter;

import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

public class EncodingFilter implements Filter
{
    private String charset; // 设置字符编码

    // 初始化
    public void init(FilterConfig config) throws ServletException
    {
        this.charset = config.getInitParameter("charset"); // 取得初始化参数
    }

    // 执行过滤
    public void doFilter(ServletRequest request, ServletResponse response,
                         FilterChain chain)
        throws IOException, ServletException
    {
        request.setCharacterEncoding(this.charset); // 设置统一编码
        chain.doFilter(request, response); // 将请求继续传递
    }

    // 销毁过滤器
    public void destroy()
    {
        System.out.println("过滤器销毁");
    }
}
```

```xml
<!-- 配置 web.xml 文件 -->
<filter>
    <!-- 过滤器名称 -->
    <filter-name>encoding</filter-name>
    <!-- 过滤器类名 -->
    <filter-class>jlujee.servlet.filter.EncodingFilter</filter-class>
    <!-- 初始化参数 -->
    <init-param>
        <param-name>charset</param-name>
        <param-value>UTF-8</param-value>
    </init-param>
</filter>
<filter-mapping>
    <!-- 与上面的 filter-name 保持一致 -->
    <filter-name>encoding</filter-name>
    <!-- 需要使用过滤器的资源，当访问这些资源时，会自动调用过滤器 -->
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

7 监听器

7.1 监听器的基本概念

Servlet 监听器是 Servlet 规范中定义的一种特殊类，用于监听 `ServletContext`、`HttpSession` 和 `ServletRequest` 等域对象的创建与销毁事件，以及监听这些域对象中属性发生修改的事件。

监听对象：

1. `ServletContext`：application，整个应用只存在一个。
2. `HttpSession`：session，针对每一个会话。
3. `ServletRequest`：request，针对每一个客户请求。

监听内容：创建、销毁、属性改变事件。

监听作用：可以在事件发生前、发生后进行一些处理，一般可以用来统计在线人数和在线用户、统计网站访问量、系统启动时初始化信息等。

7.2 监听器的使用方法

监听器的使用需要 2 步：

1. 创建一个实现监听器接口的类。
2. 配置 web.xml 文件，注册监听器。

配置 web.xml 文件的方法如下所示：

```xml
<listener>
    <listener-class>完整类名</listener-class>
</listener>
```

监听器的启动顺序：按照 web.xml 的配置顺序来启动。

加载顺序：监听器 > 过滤器 > Servlet

7.2.1 监听域对象的创建和销毁事件

对于监听域对象自身创建和销毁的事件监听器，根据监听对象不同分别实现`ServletContextListener`、`HttpSessionListener`、`ServletRequestListener`接口。

7.2.1.1 `ServletContextListener`接口

`ServletContextListener`接口用于监听 application 对象的创建和销毁。其方法有：

1. `public void contextInitialized(ServletContextEvent sce)`：`ServletContext`创建时调用。
2. `public void contextDestroyed(ServletContextEvent sce)`：`ServletContext`销毁时调用。

当`ServletContext`对象被创建和销毁时，会产生`ServletContextEvent`事件。

`ServletContextEvent`事件的方法：`public ServletContext getServletContext()`。

主要用途：作为定时器、加载全局属性对象、创建全局数据库连接、加载缓存信息等。

7.2.1.2 `HttpSessionListener`接口

`HttpSessionListener`接口用于监听 session 对象的创建和销毁。其方法有：

1. `public void sessionCreated(HttpSessionEvent se)`：session 创建时调用。
2. `public void sessionDestroyed(HttpSessionEvent se)`：session 销毁时调用。

`HttpSessionEvent`事件的方法：`public HttpSession getSession()`，获得当前正在被创建或者销毁的 session 对象。

主要用途：统计在线人数、记录访问日志等。

7.2.1.3 `ServletRequestListener`接口

`ServletRequestListener`接口用于监听 request 对象的创建和销毁。其方法有：

1. `public void requestInitialized(ServletRequestEvent src)`：request 创建时调用。
2. `public void requestDestroyed(ServletRequestEvent src)`：request 销毁时调用。

`ServletRequestEvent`事件的方法：

1. `public ServletRequest getServletRequest()`：获得当前正在被创建或销毁的 request 对象。
2. `public ServletContext getServletContext()`：获得当前正在被创建或销毁的 request 对象所属的上下文环境。

主要用途：读取 request 参数，记录访问历史。

7.2.2 监听域对象的属性改变事件

对于监听域对象中属性的增加和删除的事件监听器，根据监听对象不同分别实现`ServletContextAttributeListener`、`HttpSessionAttributeListener`、`ServletRequestAttributeListener`接口。

这三个接口需要实现的方法相同，都为`attributeAdded()`、`attributeRemoved()`、`attributeReplaced()`，但是产生的事件各不相同。

application 对象中属性变化产生的事件为`ServletContextAttributeEvent`，事件方法有：

1. `public ServletContext getServletContext()`
2. `public String getName()`：返回被改变的属性的名字。
3. `public Object getValue()`：返回被改变的属性的值。

session 对象中属性变化产生的事件为`HttpSessionBindingEvent`，事件方法有：

1. `public HttpSession getSession()`
2. `public String getName()`：返回被改变的属性的名字。
3. `public Object getValue()`：返回被改变的属性的值。

request 对象中属性变化产生的事件为`ServletRequestAttributeEvent`，事件方法有：

1. `public String getName()`：返回被改变的属性的名字。
2. `public Object getValue()`：返回被改变的属性的值。