* 状态管理-Cookie

**1.1. 状态管理**

**1.1.1. 为什么需要状态管理**

Web应用程序使用HTTP协议作为传输数据的标准协议，而HTTP协议是无状态协议，即一次请求对应一次响应，响应结束后连接即断开，状态管理能够做到不同客户端的身份识别。

**1.1.2. 什么是状态管理**

将客户端与服务器之间多次交互当做一个整体来看待，并且将多次交互中涉及的数据保存下来，提供给后续的交互进行数据的管理即状态管理。

这里的状态指的是当前的数据，管理指的是在这个多次交互的过程中对数据的存储、修改、删除。

**1.1.3. 状态管理两种常见模式**

状态管理的过程中重要的是数据的保存，只有存下来的数据才能在多次交互中起到记录的作用，所以可以按照管理的数据的存储方式和位置的不同来区分状态管理的模式。

如果将数据存储在客户端，每次向服务器端发请求时都将存在客户端的数据随着请求发送到服务器端，修改后再发回到客户端保存的这种模式叫做Cookie。

如果将数据存储在服务器端，并且为这组数据标示一个编号，只将编号发回给客户端。当客户端向服务器发送请求时只需要将这个编号发过来，服务器端按照这个编号找到对应的数据进行管理的这种模式叫做Session——会话。

**1.2. Cookie**

**1.2.1. 什么是Cookie**

一小段文本信息随着请求和响应，在客户端和服务器端之间来回传递。根据设定的时间来决定该段文本在客户端保存时长的这种工作模式叫做Cookie。最初服务器将信息发给客户端时是通过响应数据的Set-Cookie头信息来完成的。

**1.2.2. Cookie的原理**

如果客户端向服务器端AddServlet发送请求，遇到创建Cookie的代码时，那么一小段文本信息就会随着response响应中的头信息被传递会客户端。当文本信息到达客户端以后，会被保存在客户端的内存或硬盘上，存在内存中会随着内存的释放而消失，存在硬盘上则会保存更长的时间。

一旦客户端存有服务器发回的文本信息，那么当浏览器再次向服务器发起请求时，如请求FindServlet这个组件，那么存储的文本信息会随着请求数据包的消息头以Cookie：uname=xxx这样的形式将文本信息发送到服务器端。只要Cookie的生命周期没有结束，那么不管是存在内存还是硬盘上的信息都会在客户端向服务器端发出请求时自动的随着消息头发送过去。

**1.2.3. 如何创建Cookie**

Servlet API提供了javax.servlet.http.Cookie这种类型来解释Cookie。其中存储的文本以name-value对的形式进行区分，所以创建Cookie时指定name-value对即可。这个name-value最终是以Set-Cookie这种消息头的形式跟随相应数据包到达客户端，所以要想将数据添加到消息头中需要使用response对象提供的方法。

创建Cookie的代码如下所示：

01.Cookie c = new Cookie(String name,String value);

02.response.addCookie( c );

代码中的第一行实现了这段name-value对的文本的创建。

代码中的第二行执行的效果就是在响应数据包中追加一个Set-Cookie的消息头。如果发送了相同name的Cookie数据，那么之前的数据会被覆盖。能够创建多少个Cookie存放在客户端与当前浏览器的种类相关。

**1.2.4. 如何查询Cookie**

当客户端向服务器发出请求时，服务器端可以尝试着从请求数据包的消息头中获取是否携带了Cookie信息。实现这一功能的代码如下：

01.Cookie[] request.getCookies();

由于客户端是可以存放多个Cookie的，所以request提供的获取Cookie的方法的返回值是Cookie数组，如果想进一步获取某一个Cookie信息可以通过遍历数组，分别获取每一个Cookie的name和value。代码如下：

01.Cookie[] cookies = request.getCookies();

02.if(cookies!=null){

03. for(Cookie c : cookies){

04. String cookieName = c.getName();

05. String cookieValue = c.getValue();

06. }

07.}

**1.2.5. 如何修改Cookie**

所谓Cookie的修改，本质是获取到要变更值的Cookie，通过setValue方法将新的数据存入到cookie中，然后由response响应对象发回到客户端，对原有旧值覆盖后即实现了修改。主要实现代码：

01.Cookie[] cookies = request.getCookies();

02.if(cookies!=null){

03. for(Cookie c : cookies){

04. String cookieName = c.getName();

05. if（name.equals(“uname”)){

06. c.setValue(“Mark”);

07. response.addCookie( c );

08. }

09.}

其中response.addCookie（c）是非常重要的语句，如果没有这一行代码，那么就算是使用setValue方法修改了Cookie的值，但是不发回到客户端的话，也不会实现数值的改变。所以只要见到response.addCookie这行代码，即服务器端发回了带有Set-Cookie消息头的信息。

**1.2.6. Cookie的生存时间**

默认情况下，Cookie会被浏览器保存在内存中，此时Cookie的生命周期由浏览器决定，只要不关闭浏览器Cookie就会一直存在。

如果希望关闭浏览器后Cookie仍存在，则可以通过设置过期时间使得Cookie存在硬盘上得以保存更长的时间。

设置Cookie的过期时间使用如下代码：

01.void setMaxAge(int seconds);

该方法是Cookie提供的实例方法。参数seconds的单位为秒，但精度不是很高。

seconds > 0 ：代表Cookie保存在硬盘上的时长

seconds = 0 ： 代表Cookie的生命时长为现在，而这一刻稍纵即逝，所以马上Cookie就等同于过了生存时间，所以会被立即删除。这也是删除Cookie的实现方式。

seconds < 0 ：缺省值，浏览器会将Cookie保存在内存中。

以下代码实现了Cookie保存在硬盘上40秒：

08.public class AddCookieServlet extends HttpServlet {

10. public void service(HttpServletRequest request,

11.HttpServletResponse response)throws ServletException, IOException {

13. response.setContentType("text/html;charset=utf-8");

15. PrintWriter out = response.getWriter();

16. //创建cookie

17. Cookie c = new Cookie("username","Lisa");

18. c.setMagAge(40);

19. Cookie c2 = new Cookie("city","NewYork");

20. response.addCookie(c);

21. response.addCookie(c2);

22. out.close();

23. }

25.}

**1.2.7. Cookie编码**

Cookie作为在网络传输的一段字符串文本，只能保存合法的ASCII字符，如果要保存中文需要将中文变成合法的ASCII字符，即编码。使用如下代码可以实现将中文保存到Cookie中。

01.Cookie c = new Cookie("city",URLEncoder.encode("北京","utf-8"));

**1.2.8. Cookie解码**

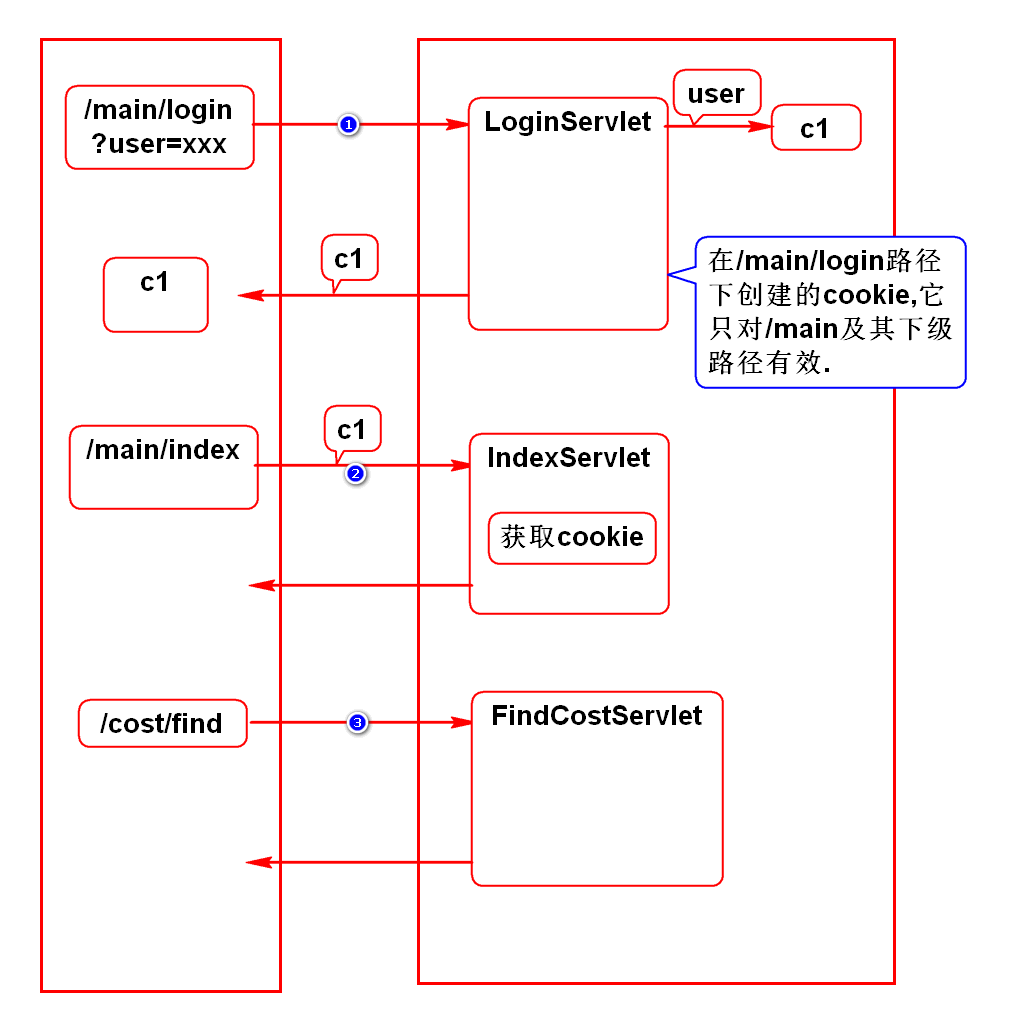
服务器读取客户端经过编码之后的信息时，要想能够正确显示需要将信息解码后才能输出。使用URLDecoder的decode（）方法即可。

**1.3. Cookie的路径问题**

**1.3.1. 什么是Cookie的路径问题**

客户端存储Cookie之后，并不是针对同一个应用访问任何资源时都自动发送Cookie到服务器端，而是会进行路径的判断。只有符合路径规范的请求才会发送Cookie到服务器端。

客户端在接受Cookie时会为该Cookie记录一个默认路径，这个路径记录的是添加这个Cookie的Web组件的路径。如，当客户端向 http://localhost:8080/test/file/addCookie.jsp发送请求时创建了cookie，那么该cookie的路径就是 /test/file.



**1.3.2. 什么时候发送Cookie**

只有当访问的地址是Cookie的路径或者其子路径时，浏览器才发送Cookie到服务器端。

如，Cookie的路径是 /test/file，那么如果访问的是 /test/file/a.jsp 或者 /test/file/b/c.jsp时，都会发送Cookie。

如果访问的是 /test/d.jsp，则浏览器不会发送Cookie。

**1.3.3. 如何设置Cookie的路径**

设置Cookie的路径可以使用Cookie的API方法，setPath(String uri);

如以下代码就实现了设置Cookie的路径为应用的顶级目录，这样所有资源路径要么与此路径相等，要么是子路径，从而实现了客户端发送任何请求时都会发送Cookie。

01.Cookie c = new Cookie(“uname”，“jack”);

02.c.setPath(“/test”);

03.response.addCookie(c);

**1.3.4. Cookie的限制**

Cookie由于存放的位置在客户端，所以可以通过修改设置被用户禁止。Cookie本质就是一小段文本，一小段说的是只能保存少量数据，长度是有限制的，一般为4kb左右。文本说的是只能保存字符串，不能保留复杂的对象类型数据。

作为网络中传输的内容，Cookie安全性很低，非常容易通过截取数据包来获取，在没有加密的情况下不要用于存放敏感数据。

就算是能够存放的长度很短，但作为网络中传输的内容也会增加网络的传输量影响带宽。在服务器处理大量请求的时候，Cookie的传递无疑会增加网络的负载量。

* cookie和session的区别(面试题)

•cookie存储在浏览器上,服务器压力小,不安全

•session存储在服务器上,服务器压力大,安全

* 状态管理-Session

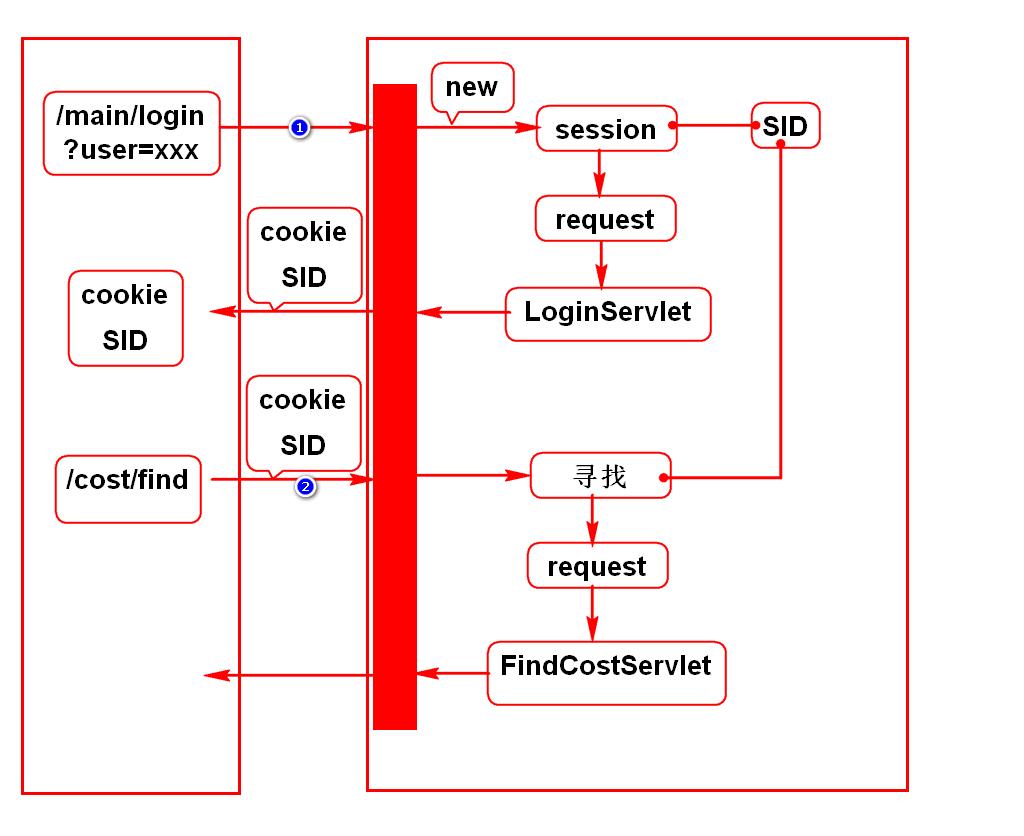
**1.1. Session**

**1.1.1. 什么是Session**

服务器为不同的客户端在内存中创建了用于保存数据的Session对象，并将用于标识该对象的唯一Id发回给与该对象对应的客户端。当浏览器再次发送请求时，SessionId也会被发送过来，服务器凭借这个唯一Id找到与之对应的Session对象。在服务器端维护的这些用于保存与不同客户端交互时的数据的对象叫做Session。

**1.1.2. Session工作原理**

浏览器第一次访问服务器时，服务器会为该客户端分配一块对象空间，并且使用不同的SID来进行标识，该标识SID会随着响应发回到客户端，且被保存在内存中。当同一个客户端再次发送请求时，标识也会被同时发送到服务器端，而服务器判断要使用哪一个Session对象内的数据时，就会根据客户端发来的这个SID来进行查找。



**1.1.3. 如何获得Session**

获得session有两种情况，要么请求中没有SID，则需要创建；要么请求中包含一个SID，根据SID去找对应的对象，但也存在找到找不到的可能。但不管哪种情况都依赖于请求中的这个唯一标识，虽然对于编程人员来讲不需要去查看这个基本不会重复、编号很长的标识，但要想获取到与客户端关联的这个session对象一定要基于请求，所以在Request类型的API中包含获取到session对象的方法，代码如下所示：

01.HttpSession s = request.getSession(boolean flag);

02.HttpSession s = request.getSession( );

使用第一种获取session对象的方法时，

flag = true：先从请求中找找看是否有SID，没有会创建新Session对象，有SID会查找与编号对应的对象，找到匹配的对象则返回，找不到SID对应的对象时则会创建新Session对象。所以，填写true就一定会得到一个Session对象。

flag= false：不存在SID以及按照SID找不到Session对象时都会返回null，只有根据SID找到对应的对象时会返回具体的Session对象。所以，填写false只会返回已经存在并且与SID匹配上了的Session对象。

request.getSession（）方法不填写参数时等同于填写true，提供该方法主要是为了书写代码时更方便，大多数情况下还是希望能够返回一个Session对象的。

**1.1.4. 如何使用Session绑定对象**

Session作为服务器端为各客户端保存交互数据的一种方式，采用name-value对的形式来区分每一组数据。向Session添加数据绑定的代码如下：

01.void session.setAttribute(String name,Object obj);

获取绑定数据或移除绑定数据的代码如下：

01.void session.getAttribute(String name);

02.void session.removeAttribute(String name);

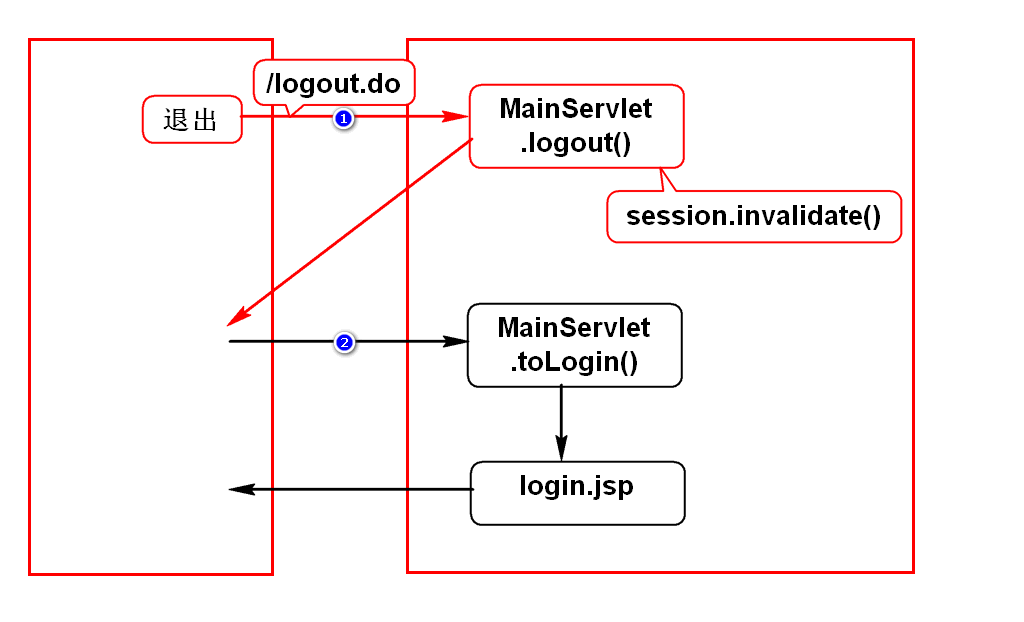
Session对象可以保存更复杂的对象类型数据了，不像Cookie只能保存字符串。

**1.1.5. 如何删除Session对象**

如果客户端想删除SID对应的Session对象时，可以使用Session对象的如下方法：

01.void invalidate()

该方法会使得服务器端与该客户端对应的Session对象不再被Session容器管理，进入到垃圾回收的状态。对于这种立即删除Session对象的操作主要应用于不再需要身份识别的情况下，如登出操作。



**1.2. Session超时**

**1.2.1. 什么是Session超时**

Session会以对象的形式占用服务器端的内存，过多的以及长期的消耗内存会降低服务器端的运行效率，所以Session对象存在于内存中时会有默认的时间限制，一旦Session对象存在的时间超过了这个缺省的时间限制则认为是Session超时，Session会失效，不能再继续访问。

Web服务器缺省的超时时间设置一般是30分钟。

**1.2.2. 如何修改Session的缺省时间限制**

有两种方式可以修改Session的缺省时间限制，编程式和声明式。

编程式：

01.void setMaxInactiveInterval（int seconds）

声明式：

01.<session-config>

02. <session-timeout>30</session-timeout>

03.</session-config>

使用声明式来修改缺省时间，那么该应用创建的所有Session对象的生命周期都会应用这个规定的时间，单位为分钟。

使用编程式来修改缺省时间只会针对调用该方法的Session对象应用这一原则，不会影响到其他对象，所以更灵活。通常在需要特殊设置时使用这种方式。时间单位是秒，与声明式的时间单位不同。

**1.2.3. Session验证**

Session既然区分不同的客户端，所以可以利用Session来实现对访问资源的保护。如，可以将资源划分为登录后才能访问。Session多用于记录身份信息，在保护资源被访问前可以通过判断Session内的信息来决定是否允许。这是依靠Session实现的验证。

使用Session实现验证的步骤如下：

步骤一、为Session对象绑定数据，代码如下：

01.HttpSession s = request.getSession();

02.s.setAttribute(“uname”，“Rose”)；

步骤二、读取Session对象中的绑定值，读取成功代表验证成功，读取失败则跳转回登录页面。

01.HttpSession s = request.getSession();

02.if(s.getAttribute(“uname”)==null){

03. response.sendRedirect(“logIn.jsp”);

04.}else{

05. //… …

06.}

**1.2.4. Session优缺点**

Session对象的数据由于保存在服务器端，并不在网络中进行传输，所以安全一些，并且能够保存的数据类型更丰富，同时Session也能够保存更多的数据，Cookie只能保存大约4kb的字符串。

Session的安全性是以牺牲服务器资源为代价的，如果用户量过大，会严重影响服务器的性能。

**1.2.5. 浏览器禁用Cookie的后果**

Session对象的查找依靠的是SID，而这个ID保存在客户端时是以Cookie的形式保存的。一旦浏览器禁用Cookie，那么SID无法保存，Session对象将不再能使用。

为了在禁用Cookie后依然能使用Session，那么将使用其他的存储方法来完成SID的保存。URL地址在网络传输过程中不仅仅能够起到标示地址的作用，还可以在其后携带一些较短的数据，SID就可以通过URL来实现保存，及URL重写。

**1.2.6. 什么是URL重写**

浏览器在访问服务器的某个地址时，会使用一个改写过的地址，即在原有地址后追加SessionID，这种重新定义URL内容的方式叫做URL重写。

如：原有地址的写法为http://localhost:8080/test/some

重写后的地址写法为http://localhost:8080/test/some;jsessionid=4E113CB3

**1.2.7. 如何实现URL重写**

生成链接地址和表单提交时，使用如下代码：

01.<a href=”<%=response.encodeURL(String url)>”>链接地址</a>

如果是重定向，使用如下代码代替response.sendRedirect()

01.response.encodeRedirectURL(String url);

**1.3. 验证码**

**1.3.1. 验证码的作用**

验证码技术可以防止对于应用恶意发送数据，因其不规律且不能由机器代劳，所以一定程度上避免了恶意程序对网站的攻击。

验证码本质上是一张图片，图片内容的准确解析不容易用程序来实现，所以能避免内容被快速读取。并且，图片的内容是使用程序随机生成后绘制得到。

注册、登录这样的功能一般都会配备验证码，一定程度上避免恶意代码的攻击。

**1.3.2. 验证码的绘制**

绘制验证码图片不仅仅需要随机生成要绘制的内容，同时要配合Java中与绘图有关的一套API来完成。绘制API将画板、画笔、颜料、字体等都解释为对象，绘制的过程就是这些对象互相配合完成的。主要涉及Graphics、Font等类型。

1.3.3. 验证码图片的绘制步骤

绘制图片的基本步骤如下：

1.常见内存画板对象

2.创建基于该画板的画笔

3.设定画笔的颜色

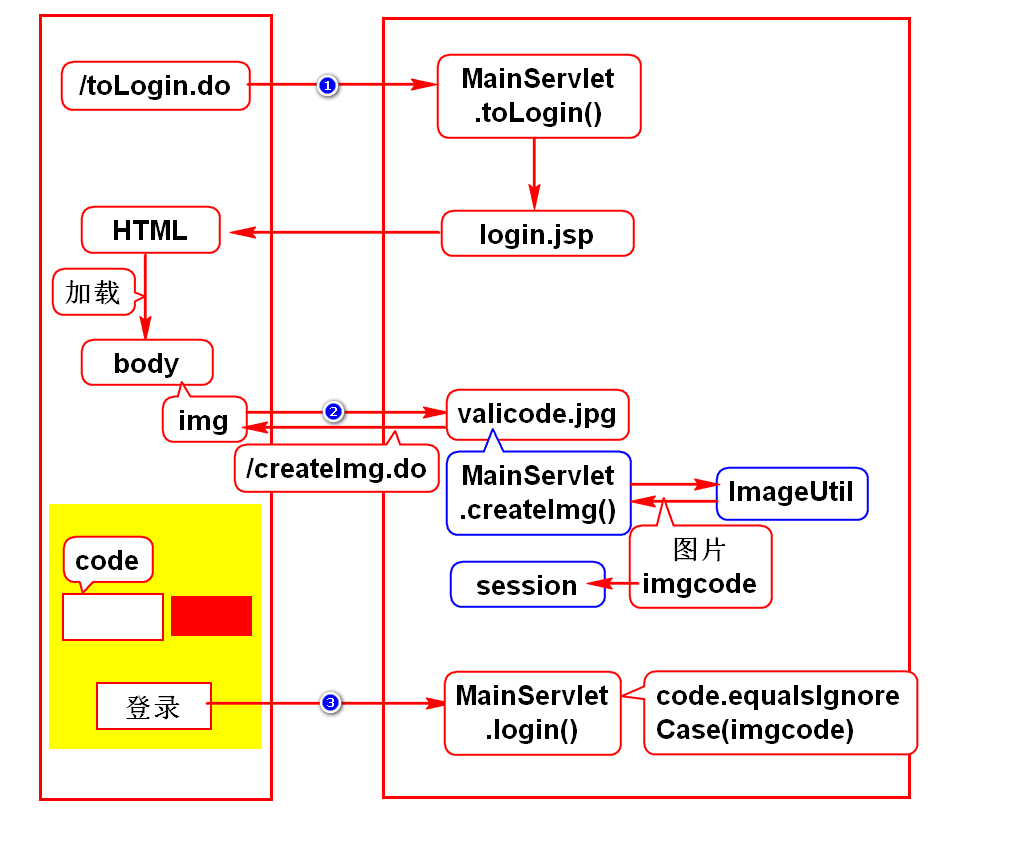
4.设定画板背景的颜色

5.使用画笔的绘制方法绘制随机内容

6.更改画笔颜色

7.绘制随机的两点一线的干扰线

8.绘制完成后将图片压缩并输出到客户端



以上步骤对应的实现代码如下所示：

代码

11.public class CheckcodeServlet extends HttpServlet {

12. private int width = 80; //图片的宽度

13. private int height = 30;//图片的高度

14. public void service(HttpServletRequest request,

15.HttpServletResponse response)throws ServletException, IOException {

18. \* 绘图

20. //step1,创建一个内存映像对象(画板)

21. BufferedImage image = new BufferedImage(width,height,BufferedImage.TYPE\_INT\_RGB);

23. //step2,获得画笔

24. Graphics g = image.getGraphics();

25. //step3,给笔上色

26. Random r = new Random();

27. g.setColor(new Color(r.nextInt(255), r.nextInt(255),r.nextInt(255)));

28. //step4,给画板设置背景颜色

29. g.fillRect(0, 0, width, height);

30. //step5,绘制一个随机的字符串

31. String number = r.nextInt(99999) + "";

32. g.setColor(new Color(0,0,0));

33. //new Font(字体,风格,大小)

34. g.setFont(new Font(null,Font.ITALIC,24));

35. g.drawString(number, 5, 25);

36. //step6,加一些干扰线

37. for(int i=0;i < 8;i++){

38. g.setColor(new Color(r.nextInt(255),

39. r.nextInt(255),r.nextInt(255)));

40. g.drawLine(r.nextInt(width),

41. r.nextInt(height), r.nextInt(width),

42. r.nextInt(height));

43. }

44. /\*

45. \* 压缩图片并输出到客户端(浏览器)

46. \*/

47. response.setContentType("image/jpeg");

48. OutputStream ops =response.getOutputStream();

49. javax.imageio.ImageIO.write(image, "jpeg", ops);

50. ops.close();

51. }

52.}

配置web.xml文件代码如下：

07.<servlet>

08.<servlet-name>CheckcodeServlet</servlet-name>

09.<servlet-class>web.CheckcodeServlet</servlet-class>

10.</servlet>

12.<servlet-mapping>

13.<servlet-name>CheckcodeServlet</servlet-name>

14.<url-pattern>/checkcode</url-pattern>

15.</servlet-mapping>

HTML页面中添加如下代码加入该图片：

01.<img src="checkcode"/>

**1.4. 密码加密**

**1.4.1. 摘要加密**

摘要加密的特点：不适用密钥，使用摘要加密算法对明文进行加密之后会得到密文，无法反推出明文。唯一性：不同的明文有不同的密文。不可逆性：即使知道了摘要加密算法，也无法反推出明文。

**1.4.2. 如何实现摘要加密**

* 过滤器

**1.1. 过滤器**

**1.1.1. 什么是过滤器**

过滤器是Servlet2.3规范之中一种特殊的Web组件，可以作为Servlet的辅助性插件存在。例如，对信息的筛选是很多Servlet里面的一个必须的前提，但是相同的功能在每个Servlet中都编写不仅仅不利于以后修改过滤逻辑，也不利于功能的重用，这时可以将这一部分非决定性的功能放在一个过滤器中，通过配置由容器控制所有请求在到达真正的处理逻辑Servlet之前先通过过滤器的检查。如果过滤逻辑需要修改，那么只需要修改这一个组件即可。所以这种可随时添加、修改，作为Servlet补充功能的Web组件是非常必要的。

**1.1.2. 如何编写过滤器**

编写过滤器遵循下列步骤：

1.编写一个实现了Filter接口的类

2.实现Filter接口的三个方法，过滤逻辑在doFilter方法中实现

3.在Web程序中注册过滤器

4.把过滤器和Web应用一起打包部署

步骤一、实现Filter接口

Filter是过滤器API中最核心的接口，定义一个Java类实现该接口以提供过滤逻辑。代码如下：

05.public class CommentFilter implements Filter{

06. //… …

07.}

步骤二、实现doFilter方法

Filter接口中共定义了三个方法，分别是init，doFilter，destroy方法。init方法在创建Filter时会被调用，且只调用一次，一般在该方法中做一些数据的准备工作，可以通过传入的FilterConfig参数获取在web.xml文件中配置提供给过滤器的初始参数。destroy方法只有在销毁过滤器对象时被调用一次，用于释放一些资源的操作。doFilter方法内编写过滤器的具体处理逻辑，会被多次执行。该方法共有三个参数，请求和响应用于获取数据以及追加数据，FilterChain是过滤器链，负责多过滤器的传递。

实现代码如下：

07.public class CommentFilter implements Filter{

08. private FilterConfig config;

10. public void destroy() {

11. System.out.println("CommentFilter1's destroy...");

12. }

15. \* 容器会调用doFilter方法来处理请求(

16. \* 相当于servlet的service方法)。

17. \* 容器会将request对象(arg0)和response对象

18. \* (arg1)作为参数传给doFilter方法。

19. \* 如果调用了FilterChain(arg2)的doFilter(request,response)方法，

20.\* 则容器会调用后续的过滤器或者servlet，否则请求处理完毕。

22. public void doFilter(ServletRequest arg0,

23. ServletResponse arg1, FilterChain arg2)

24. throws IOException, ServletException {

25. HttpServletRequest request = (HttpServletRequest)arg0;

26. HttpServletResponse response =(HttpServletResponse)arg1;

27. request.setCharacterEncoding("utf-8");

28. response.setContentType("text/html;charset=utf-8");

29. PrintWriter out = response.getWriter();

30. String content = request.getParameter("content");

31. String illegalStr = config.getInitParameter("illegalStr");

32. if(content.indexOf(illegalStr) != -1){

33. //有敏感字

34. out.println("<h1>评论内容包含了敏感字</h1>");

35. }else{

36. //没有敏感字

37. // 执行FilterChain的doFilter会调用后续的过滤器或者servlet。

38. arg2.doFilter(arg0, arg1);

39. }

40. System.out.println("Filter1's doFilter end.");

41. }

42.

43. /\*

44. \* FilterConfig对象可以用来访问过滤器的初始化参数。

45. \* init方法只会执行一次。

46. \*/

47. public void init(FilterConfig arg0) throws ServletException {

48. System.out.println("CommentFilter1's init...");

49. config = arg0;

50. }

51.}

步骤三、注册过滤器

在web.xml文件中注册过滤器，代码如下：

07. <!-- 过滤器 -->

08. <filter>

09. <filter-name>filter1</filter-name>

10. <filter-class>web.CommentFilter</filter-class>

11. <!-- 初始化参数，由FilterConfig对象读取-->

12. <init-param>

13. <param-name>illegalStr</param-name>

14. <param-value>胡萝卜</param-value>

15. </init-param>

16. </filter>

17. <filter-mapping>

18. <filter-name>filter1</filter-name>

19. <url-pattern>/comment</url-pattern>

20. </filter-mapping>

21.</web-app>

步骤四、部署过滤器

将编译后的过滤器和其他Web组件类合在一起，连同web.xml文件放进应用程序结构中即可。

**1.1.3. 过滤器的执行流程**

客户端发来请求后，不会直接将请求送达Servlet，而是先走过滤器1的doFilter方法中的code1，当遇到chain.doFilter（）方法时，控制权交到service（）方法，执行业务逻辑，但执行结束后并不会立即将响应返回给客户端，而是回到过滤器1的doFilter（）方法中code2部分，如果该部分有代码就会执行，执行结束后才会将response对象返回给客户端。

从流程中可以看到，过滤器不仅仅对Servlet的执行前起到过滤作用，对于执行后同样有过滤效果。所以，过滤器是对request和response的检查。

**1.1.4. 过滤器的优先级**

在一个Web应用中，可以有多个过滤器，它们的优先级由位于web.xml文件中的声明顺序决定，具体是按照<filter-mapping>的顺序来决定的。如下代码所示，filter1和filter2都已经注册，执行顺序是filter2 ( filter1 。

07. <!-- 过滤器 -->

08. <filter>

09. <filter-name>filter1</filter-name>

10. <filter-class></filter-class>

11. </filter>

12. <filter>

13. <filter-name>filter2</filter-name>

14. <filter-class></filter-class>

15. </filter>

16. <filter-mapping>

17. <filter-name>filter2</filter-name>

18. <url-pattern>/comment2</url-pattern>

19. </filter-mapping>

20. <filter-mapping>

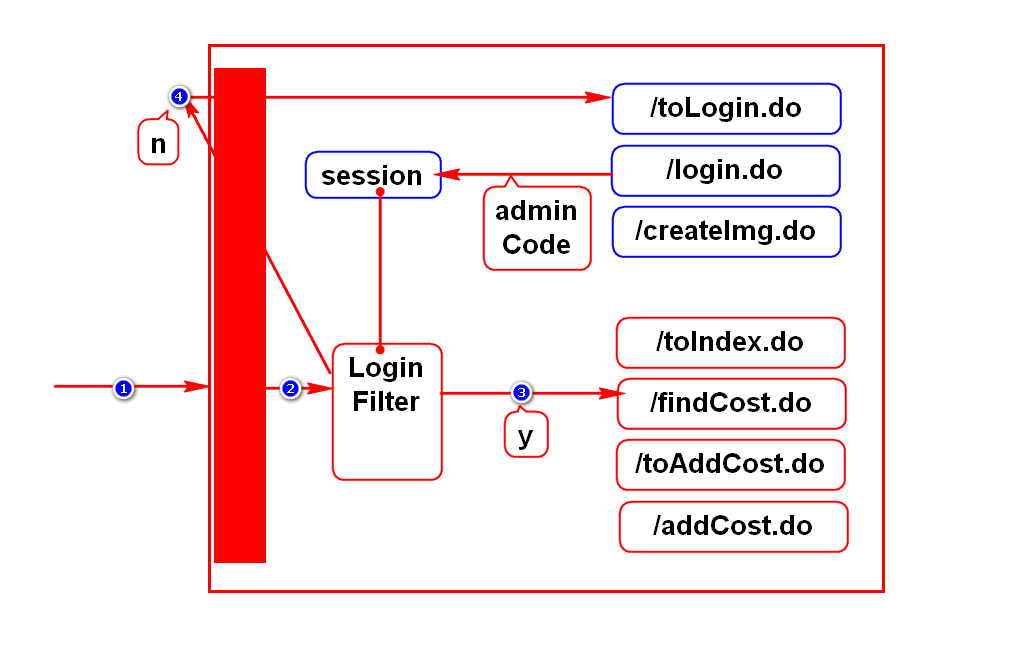
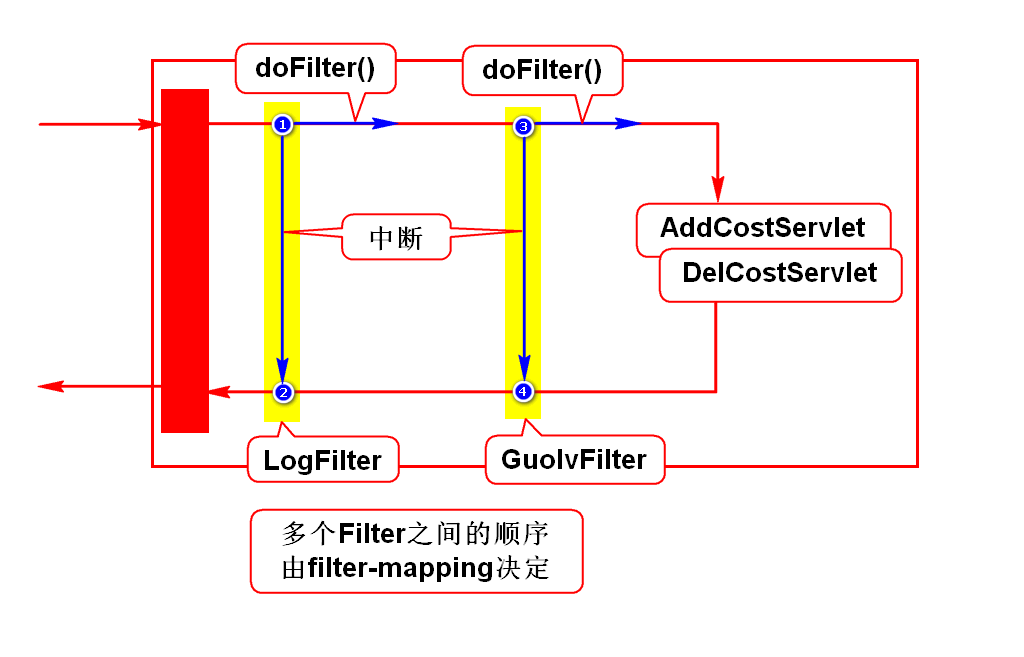
21. <filter-name>filter1</filter-name>

22. <url-pattern>/comment1</url-pattern>

23. </filter-mapping>

24.</web-app>

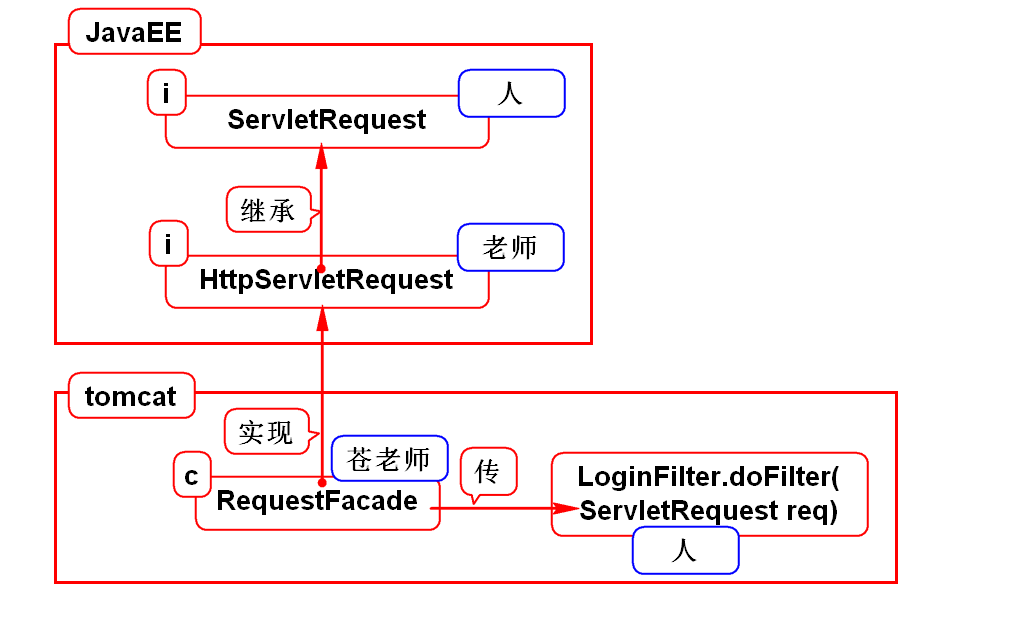
**1.1.5. 多个过滤器的执行流程**



**1.1.6. 过滤器的初始化参数**

容器启动之后，会创建过滤器实例。通过init方法来完成过滤器的初始化。初始化时可以添加一些配置，提升动态性。而这些参数通过在web.xml文件中的<init-param>以name-value对的形式存在。

读取这些name-value对需要使用FilterConfig对象，从web.xml文件到FilterConfig对象的过程由容器完成，并通过参数传入到init方法之中，只需要设定一些成员对象保存数值就可以在doFilter方法中使用。



**1.1.7. 初始化参数的配置**

在web.xml文件中添加过滤器初始化参数的配置代码如下：

07. <!-- 过滤器 -->

08. <filter>

09. <filter-name>filter1</filter-name>

10. <filter-class>web.CommentFilter1</filter-class>

11. <!-- 初始化参数 -->

12. <init-param>

13. <param-name>illegalStr</param-name>

14. <param-value>胡萝卜</param-value>

15. </init-param>

16. </filter>

17. <filter-mapping>

18. <filter-name>filter1</filter-name>

19. <url-pattern>/comment</url-pattern>

20. </filter-mapping>

21.</web-app>

**1.1.8. 读取初始化参数**

读取初始化参数使用如下代码：

01.public class CommentFilter implements Filter{

02. private FilterConfig config;

03. public void init(FilterConfig arg0) throws ServletException {

04. config = arg0;

05. }

06. public void doFilter(ServletRequest arg0,

07. ServletResponse arg1, FilterChain arg2)

08. throws IOException, ServletException {

09. String illegalStr = config.getInitParameter("illegalStr");

10. // … …

11. }

12.public void destroy() {

13. // … …

14. }

15.}

**1.1.9. 过滤器的特点**

方便增加或减少某个功能模块，需要添加过滤就多部署一个class修改一下web.xml文件，需要减少某个功能只要删除web.xml中对应的声明即可。

方便修改处理逻辑。当把一些功能独立到某个模块时，如果逻辑变了，只修改这一个文件并更新就可以实现功能的改变，而不用修改每一个使用这个插件的组件。

* 监听器

**2.1. 监听器**

**2.1.1. 什么是监听器**

servlet规范当中定义的一种特殊的组件，用来监听servlet容器产生的事件并进行响应的处理。

**2.1.2. 生命周期相关的事件**

容器创建或者销毁request，session，ServletContext（上下文/环境）时产生的事件（统计在线人数)。

**2.1.3. 绑定数据相关的事件**

调用了以上三个对象（request，response，ServletContext）的setAttribute，removeAttribute方法时产生的事件。

**2.1.4. 如何编写监听器**

step1，写一个java类，实现相应的监听器接口（共有8个接口）。要依据监听的事件类型来选择相应的监听器接口，比如要监听session对象的创建和销毁，要实现HttpSessionListener。

step2,在监听器接口方法中，实现相应的监听处理逻辑。比如，session对象被删除了，将人数减1。

step3，注册（在web.xml文件中配置即可）。

**2.1.5. 编写Java类**

01.public class CountListener implements HttpSessionListener{

03.private int count = 0;

05.public void sessionCreated(HttpSessionEvent arg0){

06.System.out.println("sessionCreated…");

07.count ++;

08.}

10.public sessionDestroyed(HttpSessionEvent arg0){

11.System.out.println("session destroyed…");

12.count--;

13.}

14.}

**2.1.6. 实现处理逻辑**

01.public void sessionCreated(HttpSessionEvent arg0){

02.System.out.print("sessionCreated…");

03.HttpSession session = args.getSession();

04.ServletContext ctx = session.getServletContext();

05.ctx.setAttribute("count",count);

06.}

**2.1.7. 注册监听器**

01.<listener>

02. <listener-class>web.CountListener</listener-class>

03.</listener>

**2.1.8. 监听器的应用场景**

系统框架级别的代码经常需要检测容器中数据或对象的变化,以一个不受人为控制因素的触发为执行时机,所以对于需要根据数据变化来做出自动反馈的功能都可以使用到监听器.