1 Servlet简介

什么是Servlet? （服务器上的程序。处理浏览器带来HTTP请求。）

Java Servlet 是运行在 Web 服务器或应用服务器上的程序，

它是作为来自 Web 浏览器或其他 HTTP 客户端的请求和 HTTP 服务器上的数据库或应用程序之间的**中间层。**

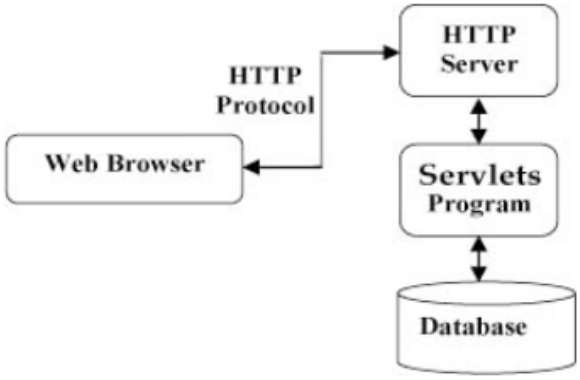
为什么要用到Servlet?

我们编写java程序想要在网上实现 聊天、发帖、这样一些的交互功能，普通的java技术是非常难完成的。sun公司就提供了Servlet这种技术供我们使用。

Servlet的作用

Servlet带给我们最大的作用就是能够处理浏览器带来HTTP请求，并返回一个响应给浏览器，从而**实现浏览器和服务器的交互。**

下图显示了 Servlet 在 Web 应用程序中的位置。



2 HTTP协议

2.1 什么是HTTP协议

超文本传输协议（HTTP，HyperText Transfer Protocol)是互联网上应用最为广泛的一种网络协议。所有的WWW文件都必须遵守这个标准。它是TCP/IP协议的一个**应用层协议**

简单来说，HTTP协议就是客户端和服务器交互**的一种通迅的格式。**

例子:在浏览器点击一个链接，浏览器就为我打开这个链接的网页。

原理：当在浏览器中点击这个链接的时候，浏览器会向服务器发送一段文本，告诉服务器请求打开的是哪一个网页。服务器收到请求后，就返回一段文本给浏览器，浏览器会将该文本解析，然后显示出来。这段文本就是遵循HTTP协议规范的。

2.2 HTTP1.0和HTTP1.1的区别

HTTP1.0协议中，客户端与web服务器建立连接后，只能获得一个web资源【短连接，获取资源后就断开连接】

HTTP1.1协议中，允许客户端与web服务器建立连接后，在一个连接上获取多个web资源【保持连接】

2.3 HTTP请求

浏览器向服务器请求某个web资源时，称之为浏览器向服务器发送了一个http请求。

一个完整http请求应该包含三个部分：

1. 请求行【描述客户端的请求方式、请求的资源名称，以及使用的HTTP协议版本号】
2. 多个消息头【描述客户端请求哪台主机，以及客户端的一些环境信息等】
3. 一个空行

1 请求行

请求行：GET /java.html HTTP/1.1

请求行中的GET称之为请求方式，请求方式有：POST,GET,HEAD,OPTIONS,DELETE,TRACE,PUT。

常用的有：POST,GET

一般来说，当我们点击超链接，通过地址栏访问都是get请求方式。通过表单提交的数据一般是post方式。

可以简单理解GET方式用来查询数据,POST方式用来提交数据，get的提交速度比post快

GET方式：在URL地址后附带的参数是有限制的，其数据容量通常不能超过1K。

POST方式：可以在请求的实体内容中向服务器发送数据，传送的数据量无限制。

2 请求头

- Accept: text/html,image/* 【浏览器告诉服务器，它支持的数据类型】
- Accept-Charset: ISO-8859-1 【浏览器告诉服务器，它支持哪种字符集】
- Accept-Encoding: gzip,compress 【浏览器告诉服务器，它支持的压缩格式】
- Accept-Language: en-us,zh-cn 【浏览器告诉服务器，它的语言环境】
- Host: www.it315.org:80【浏览器告诉服务器，它的想访问哪台主机】
- If-Modified-Since: Tue, 11 Jul 2000 18:23:51 GMT【浏览器告诉服务器，缓存数据的时间】
- Referer: http://www.it315.org/index.jsp【浏览器告诉服务器，客户机是从那个页面来的---反盗链】
- 8.User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)【浏览器告诉服务器，浏览器的内核是什么】
- Cookie【浏览器告诉服务器，带来的Cookie是什么】
- Connection: close/Keep-Alive 【浏览器告诉服务器，请求完后是断开链接还是保持链接】
- Date: Tue, 11 Jul 2000 18:23:51 GMT【浏览器告诉服务器，请求的时间】

2.4 HTTP响应

一个HTTP响应代表着服务器向浏览器回送数据

一个完整的HTTP响应应该包含四个部分:

1. 一个状态行【用于描述服务器对请求的处理结果。】
2. 多个消息头【用于描述服务器的基本信息，以及数据的描述，服务器通过这些数据的描述信息，可以通知客户端如何处理等一会儿它回送的数据】
3. 一个空行
4. 实体内容【服务器向客户端回送的数据】

1 状态行

格式： HTTP版本号　状态码　原因叙述

状态行：HTTP/1.1 200 OK

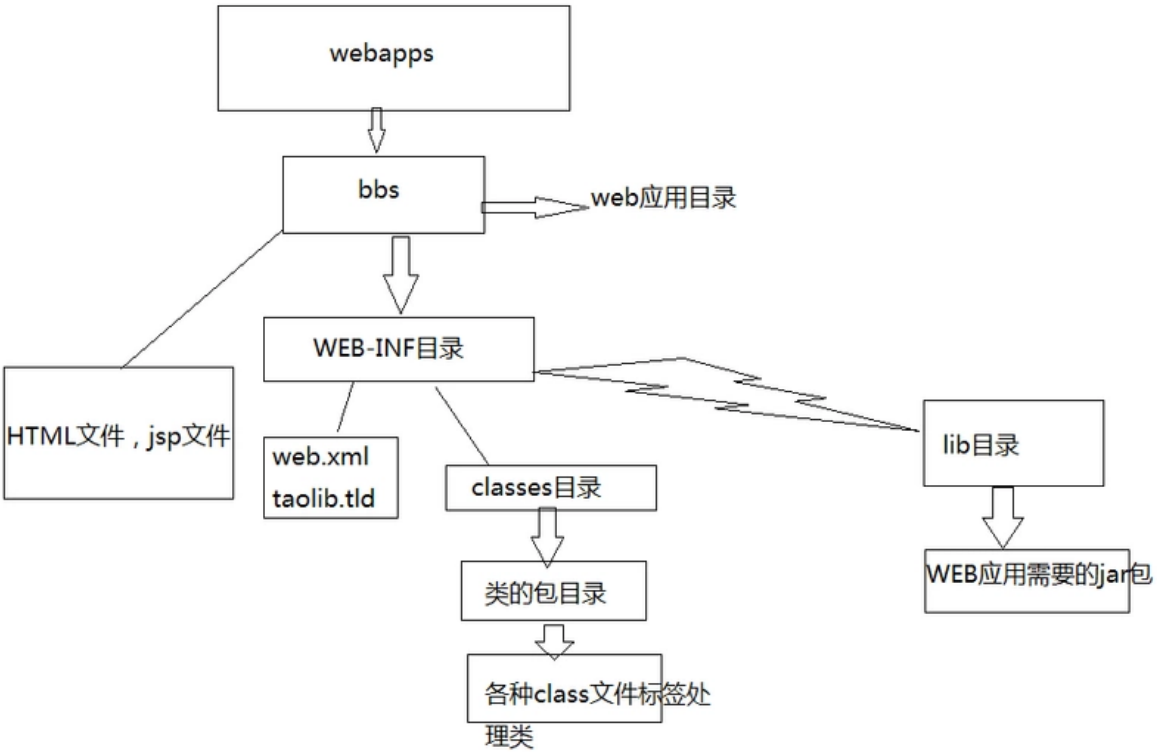
状态码用于表示服务器对请求的处理结果，它是一个三位的十进制数。响应状态码分为5类

状态码	含义
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
200～299	表示成功接收请求并已完成整个处理过程，常用200
300～399	为完成请求，客户需进一步细化请求。例如，请求的资源已经移动一个新地址，常用302、307和304
400～499	客户端的请求有错误，常用404
500～599	服务器端出现错误，常用 500

2 响应头

- Location: http://www.it315.org/index.jsp 【服务器告诉浏览器要跳转到哪个页面】
- Server:apache tomcat【服务器告诉浏览器，服务器的型号是什么】
- Content-Encoding: gzip 【服务器告诉浏览器数据压缩的格式】
- Content-Length: 80 【服务器告诉浏览器回送数据的长度】
- Content-Language: zh-cn 【服务器告诉浏览器，服务器的语言环境】
- Content-Type: text/html; charset=GB2312 【服务器告诉浏览器，回送数据的类型】
- Last-Modified: Tue, 11 Jul 2000 18:23:51 GMT【服务器告诉浏览器该资源上次更新时间】
- Refresh: 1;url=http://www.it315.org【服务器告诉浏览器要定时刷新】
- Content-Disposition: attachment; filename=aaa.zip【服务器告诉浏览器以下载方式打开数据】
- Transfer-Encoding: chunked 【服务器告诉浏览器数据以分块方式回送】
- Set-Cookie:SS=Q0=5Lb_nQ; path=/search【服务器告诉浏览器要保存Cookie】
- Expires: -1【服务器告诉浏览器不要设置缓存】
- Cache-Control: no-cache 【服务器告诉浏览器不要设置缓存】
- Pragma: no-cache 【服务器告诉浏览器不要设置缓存】
- Connection: close/Keep-Alive 【服务器告诉浏览器连接方式】
- Date: Tue, 11 Jul 2000 18:23:51 GMT【服务器告诉浏览器回送数据的时间】

3 JAVAWEB目录结构



以上图说明：

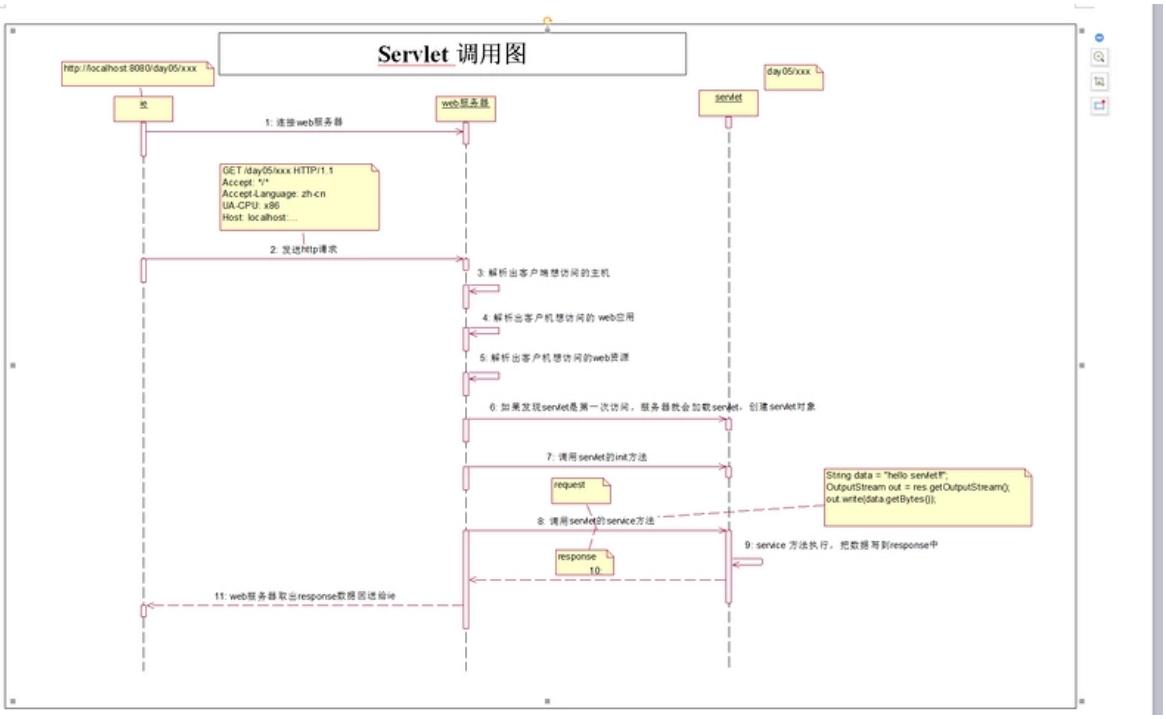
- bbs目录代表一个web应用
- bbs目录下的html,jsp文件可以直接被浏览器访问
- WEB-INF目录下的资源是**不能直接被浏览器访问**的
- web.xml文件是web程序的主要配置文件
- 所有的classes文件都放在classes目录下
- jar文件放在lib目录下

4 Servlet生命周期

1. **加载Servlet。**当Tomcat第一次访问Servlet的时候，Tomcat会负责创建Servlet的实例
2. **初始化。**当Servlet被实例化后，Tomcat会调用init()方法初始化这个对象
3. **处理服务。**当浏览器**访问Servlet**的时候，Servlet 会调用service()方法处理请求
4. **销毁。**当Tomcat关闭时或者检测到Servlet要从Tomcat删除的时候会自动调用destroy()方法，让该实例释放掉所占的资源。一个Servlet如果长时间不被使用的话，也会被Tomcat自动销毁
5. **卸载。**当Servlet调用完destroy()方法后，等待垃圾回收。如果有需要再次使用这个Servlet，会重新调用init()方法进行初始化操作。

简单总结：只要访问Servlet，service()就会被调用。init()只有第一次访问Servlet的时候才会被调用。destroy()只有在Tomcat关闭的时候才会被调用。

我们根据Servlet的生命周期画出Servlet的调用图加深理解。



5 编写Servlet程序

5.1 实现Servlet接口

- 创建一个自定义类，实现Servlet接口
- 重写5个方法，有init【初始化】，destroy【销毁】,service【服务】,ServletConfig【Servlet配置】,getServletInfo【Servlet信息】
- 配置xml文件，让Tomcat知道浏览器怎么访问这个Servlet。
- 访问自己写的Servlet程序

5.2 继承HttpServlet

在上面我们实现Servlet接口，要实现5个方法。这样太麻烦了！而HttpServlet类已经实现了Servlet接口的所有方法，编写Servlet时，只需要继承HttpServlet，重写你需要的方法即可，并且它在原有Servlet接口上添加了一些与HTTP协议处理方法，它比Servlet接口的功能更为强大。

一般我们开发的时候，都是重写doGet()和doPost()方法的。

6 Servlet的细节

6.1 一个已经注册的Servlet可以被多次映射

同一个Servlet可以被映射到多个URL上。

```
<servlet>
    <servlet-name>Demo1</servlet-name>
    <servlet-class>zhongfucheng.web.Demo1</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Demo1</servlet-name>
    <url-pattern>/Demo1</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>Demo1</servlet-name>
    <url-pattern>/ouzicheng</url-pattern>
</servlet-mapping>
```

无论我访问的是http://localhost:8080/Demo1还是http://localhost:8080/ouzicheng。我访问的都是Demo1。

6.2 Servlet映射的URL可以使用通配符

通配符有两种格式：

1. *.扩展名
2. 正斜杠（/）开头并以“/*”结尾。

```
<servlet-mapping>
    <servlet-name>Demo1</servlet-name>
    <url-pattern>/*</url-pattern>
</servlet-mapping>
```

```
<servlet-mapping>
    <servlet-name>Demo1</servlet-name>
    <url-pattern>*.jsp</url-pattern>
</servlet-mapping>
```

如果*.扩展名和正斜杠（/）开头并以“/*”结尾两种通配符同时出现，匹配的是哪一个呢？

1. 看谁的匹配度高，谁就被选择
2. `*.`扩展名的优先级最低

Servlet映射的URL可以使用通配符和Servlet可以被映射到多个URL上的作用：

1. 隐藏网站是用什么编程语言写的【.php,.net,.asp实际上访问的都是同一个资源】
2. 用特定的后缀声明版权【公司缩写】

```
<servlet>
    <servlet-name>Demo1</servlet-name>
    <servlet-class>zhongfucheng.web.Demo1</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Demo1</servlet-name>
    <url-pattern>*.jsp</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>Demo1</servlet-name>
    <url-pattern>*.net</url-pattern>
</servlet-mapping>
```

```
<servlet-mapping>
    <servlet-name>Demo1</servlet-name>
    <url-pattern>*.asp</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>Demo1</servlet-name>
    <url-pattern>*.php</url-pattern>
</servlet-mapping>
```

6.3 Servlet是单例的

1 为什么Servlet是单例的?

浏览器多次对Servlet的请求，一般情况下，服务器只创建一个Servlet对象，也就是说，Servlet对象一旦创建了，就会驻留在内存中，为后续的请求做服务，直到服务器关闭。

2 每次访问请求对象和响应对象都是新的

对于*每次访问*请求，Servlet引擎都会创建一个新的HttpServletRequest请求对象和一个新的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，service方法再根据请求方式分别调用doXXX方法。

3 线程安全问题

当多个用户访问Servlet的时候，服务器会为每个用户创建一个线程。当多个用户并发访问Servlet共享资源的时候就会出现线程安全问题。

原则：

1. 如果一个变量需要多个用户共享，则应当在访问该变量的时候，加同步机制synchronized (对象){}
2. 如果一个变量不需要共享，则直接在 doGet() 或者 doPost()定义.这样不会存在线程安全问题

6.4 load-on-startup

如果在<servlet>元素中配置了一个<load-on-startup>元素，那么WEB应用程序在启动时，就会装载并创建Servlet的实例对象、以及调用Servlet实例对象的init()方法。

```
<servlet>
    <servlet-name>Demo1</servlet-name>
    <servlet-class>zhongfucheng.web.Demo1</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
```

作用：

1. 为web应用写一个InitServlet，这个servlet配置为启动时装载，为整个web应用创建必要的数据库表和数据
2. 完成一些定时的任务【定时写日志，定时备份数据】

6.5 在web访问任何资源都是在访问Servlet

当你启动Tomcat，你在网址上输入http://localhost:8080。为什么会出现Tomcat小猫的页面?

这是由缺省Servlet为你服务的!

我们先看一下web.xml文件中的配置,web.xml文件配置了一个缺省Servlet

```
<servlet>
    <servlet-name>default</servlet-name>
    <servlet-class>org.apache.catalina.servlets.DefaultServlet</servlet-class>
    <init-param>
        <param-name>debug</param-name>
        <param-value>0</param-value>
    </init-param>
    <init-param>
        <param-name>listings</param-name>
        <param-value>false</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>/</url-pattern>     --这里是关键
</servlet-mapping>
```

- 什么叫做缺省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
- 既然我说了在web访问任何资源都是在访问Servlet，那么我访问静态资源【本地图片，本地HTML文件】也是在访问这个缺省Servlet【DefaultServlet】
- 证实一下：当我没有手工配置缺省Servlet的时候，访问本地图片是可以访问得到的

总结：无论在web中访问什么资源【包括JSP】，都是在访问Servlet。没有手工配置缺省Servlet的时候，你访问静态图片，静态网页，缺省Servlet会在你web站点中寻找该图片或网页，如果有就返回给浏览器，没有就报404错误

7 ServletConfig对象

通过此对象可以读取web.xml中**配置的初始化参数。**

现在问题来了，**为什么我们要把参数信息放到web.xml文件中呢**？我们可以直接在程序中都可以定义参数信息，搞到web.xml文件中又有什么好处呢？

好处就是：能够让你的程序更加灵活【更换需求，更改配置文件web.xml即可，程序代码不用改】

- 为Demo1这个Servlet配置一个参数，参数名是name，值是zhongfucheng

```
<servlet>
    <servlet-name>Demo1</servlet-name>
    <servlet-class>zhongfucheng.web.Demo1</servlet-class>
    <init-param>
        <param-name>name</param-name>
        <param-value>zhongfucheng</param-value>
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>Demo1</servlet-name>
    <url-pattern>/Demo1</url-pattern>
</servlet-mapping>
```

- 在Servlet中获取ServletConfig对象，通过ServletConfig对象获取在web.xml文件配置的参数

```
// 获取ServletConfig对象
ServletConfig servletConfig = this.getServletConfig();
// 根据配置的名字获取值
String value = servletConfig.getInitParameter("name");
System.out.println(value);
```

8 ServletContext对象

当Tomcat启动的时候，就会创建一个S**ervletContext对象。它代表着当前web站点**

ServletContext的作用：

1. ServletContext既然代表着当前web站点，那么所有Servlet都共享着一个ServletContext对象，所以Servlet之间可以通过ServletContext实现通讯。
2. ServletConfig获取的是配置的是单个Servlet的参数信息，ServletContext可以获取的是配置整个web站点的参数信息
3. 利用ServletContext读取web站点的资源文件
4. 实现Servlet的转发【用ServletContext转发不多，主要用request转发】

8.1 Servlet之间实现通讯

ServletContext对象可以被称之为**域对象**

到这里可能有一个疑问，域对象是什么呢？其实域对象可以简单理解成一个容器【类似于Map集合】

实现Servlet之间通讯就要用到ServletContext的setAttribute(String name,Object obj)方法， 第一个参数是关键字，第二个参数是你要存储的对象。

- 这是Demo2的代码

```
//获取到ServletContext对象
ServletContext servletContext = this.getServletContext();
String value = "zhongfucheng";
//MyName作为关键字，value作为值存进    域对象【类型于Map集合】
servletContext.setAttribute("MyName", value);
```

- 这是Demo3的代码

```
//获取ServletContext对象
ServletContext servletContext = this.getServletContext();
//通过关键字获取存储在域对象的值
String value = (String) servletContext.getAttribute("MyName");
System.out.println(value);
```

- 访问Demo3可以获取Demo2存储的信息，从而实现多个Servlet之间通讯

8.2 获取web站点配置的信息

如果我想要让**所有的Servlet都能够获取到连接数据库的信息**，不可能在web.xml文件中每个Servlet中都配置一下，这样代码量太大了！并且会显得非常啰嗦冗余。

- web.xml文件支持对整个站点进行配置参数信息【所有Servlet都可以取到该参数信息】

```
<context-param>
    <param-name>name</param-name>
    <param-value>zhongfucheng</param-value>
</context-param>
```

- Demo4代码

```
//获取到ServletContext对象
ServletContext servletContext = this.getServletContext();
//通过名称获取值
String value = servletContext.getInitParameter("name");
System.out.println(value);
```

8.3 读取资源文件

第一种方式：

- 现在我要通过Servlet111读取1.png图片，按我们以前的方式，代码应该是这样的。

```
FileInputStream fileInputStream = new FileInputStream("1.png");
System.out.println(fileInputStream);
```

- 当我们访问的时候，却出错了！说找不到1.png文件。
- 这是为什么呢？我们以前读取文件的时候，如果程序和文件在同一包名，可以直接通过文件名称获取得到的！，原因很简单，以前我们写的程序都是通过JVM来运行的，而现在，我们是通过Tomcat来运行的
- 根据web的目录规范，Servlet编译后的class文件是存放在WEB-INFclasses文件夹中的
- 看到这里，我们知道了要进入classes目录中读取文件，所以我们将代码改成以下方式

```
FileInputStream fileInputStream = new
FileInputStream("D:\\zhongfucheng\\web\\WEB-
INF\\classes\\zhongfucheng\\web\\1.png");
System.out.println(fileInputStream);
```

- 再去读取时，就发现可以获取到文件了。
- 但是现在问题又来了，我读取文件的时候都要写上绝对路径，这样太不灵活了。试想一下，如果我将该读取文件的模块移到其他的web站点上，我的代码就又要修改了【因为web站点的名字不一样】。
- 我们通过ServletContext读取就可以避免修改代码的情况，因为ServletContext对象是根据当前web站点而生成的

```
//获取到ServletContext对象
ServletContext servletContext = this.getServletContext();
//调用ServletContext方法获取到读取文件的流
InputStream inputStream = servletContext.getResourceAsStream("/WEB-
INF/classes/zhongfucheng/web/1.png");
```

第二种方式：

- 如果我的文件放在web目录下，那么就简单得多了！,直接通过文件名称就能获取

```
//获取到ServletContext对象
ServletContext servletContext = this.getServletContext();
//调用ServletContext方法获取到读取文件的流
InputStream inputStream = servletContext.getResourceAsStream("2.png");
```

第三种方式：

通过类装载器读取资源文件。

- 图片文件放在了src目录下【也叫做类目录】

```
//获取到类装载器
ClassLoader classLoader = Servlet111.class.getClassLoader();
//通过类装载器获取到读取文件流
InputStream inputStream = classLoader.getResourceAsStream("3.png");
```

原则：**如果文件太大，就不能用类装载器的方式去读取，会导致内存溢出**

9 response、request对象

Tomcat收到客户端的http请求，会针对**每一次请求**，分别创建一个代表请求的request对象、和代表响应的response对象。

既然request对象代表http请求，那么我们获取浏览器提交过来的数据，找request对象即可。
response对象代表http响应，那么我们向浏览器输出数据，找response对象即可。

10 HttpServletResponse

什么是HttpServletResponse对象?

http响应由状态行、实体内容、消息头、一个空行组成。HttpServletResponse对象就封装了http响应的信息。

10.1 调用getOutputStream()方法向浏览器输出数据

调用getOutputStream()方法向浏览器输出数据，getOutputStream()方法可以使用print()也可以使用write()，它们有什么区别呢？我们试验一下。代码如下：

```
//获取到OutputStream流
ServletOutputStream servletOutputStream = response.getOutputStream();
//向浏览器输出数据
servletOutputStream.print("aaaa");  //正常显示
servletOutputStream.print("中国！");   //出现异常
```

为什么会出现异常呢？在io中我们学过，outputStream是输出二进制数据的，print()方法接收了一个字符串，print()方法要把“中国”改成二进制数据，**Tomcat使用IOS 8859-1编码对其进行转换**,“中国”根本对ISO 8859-1编码不支持。所以出现了异常。

```
response.getOutputStream().write("aaa".getBytes());//正常显示
response.getOutputStream().write("你好呀我是中国".getBytes());//正常显示
```

为什么使用write()方法能够正常向浏览器输出中文呢？"你好呀我是中国".getBytes()，这句代码在转成byte[]数组的时候默认查的是gb2312编码，而"你好呀我是中国"支持gb2312编码，所以可以正常显示出来。

但是，程序要实现通用性，应该使用的是UTF-8编码，我们在字符串转换成字节数组时指定UTF-8编码，看看会怎么样。

```
response.getOutputStream().write("你好呀我是中国".getBytes("UTF-8"));  //乱码
```

为什么它变成了乱码呢？原因是这样的：我在向服务器输出的中文是UTF-8编码的，而浏览器采用的是GBK，GBK想显示UTF-8的中文数据，不乱码才怪呢！

将浏览器的编码改成UTF-8可以解决乱码问题。可是，每次编写UTF-8程序时都要去网页上改编码格式吗？这样明显不可能的。

既然HTTP响应有对浏览器说明回送数据是什么类型的消息头，那么HttpServletResponse对象就应该有相对应的方法告诉浏览器回送的数据编码格式是什么。

```
//设置头信息，告诉浏览器我回送的数据编码是utf-8的
response.setHeader("Content-Type", "text/html;charset=UTF-8");
response.getOutputStream().write("你好呀我是中国".getBytes("UTF-8"));
```

浏览器在显示数据时，自动把页面的编码格式置换成UTF-8，乱码问题也解决了。

除了使用HttpServletResponse对象设置消息头的方法，我可以使用html的<meta>标签模拟一个http消息头

```
//获取到servletOutputStream对象
ServletOutputStream servletOutputStream = response.getOutputStream();
//使用meta标签模拟http消息头，告诉浏览器回送数据的编码和格式
servletOutputStream.write("<meta http-equiv='content-type'
content='text/html;charset=UTF-8'>".getBytes());
servletOutputStream.write("我是中国".getBytes("UTF-8"));
```

10.2 调用getWriter()方法向浏览器输出数据

对于getWriter()方法而言，是Writer的子类，那么只能向浏览器输出字符数据，不能输出二进制数据。

```
//获取到printWriter对象
PrintWriter printWriter = response.getWriter();
printWriter.write("看完博客点赞！");    //出现了乱码
```

为什么出现乱码了呢？由于Tomcat默认的编码是ISO 8859-1，当我们输出中文数据的时候，Tomcat会依据ISO 8859-1码表给我们的数据编码，中文不支持这个码表，所以出现了乱码。

```
//原本是ISO 8859-1的编码，我设置成UTF-8
response.setCharacterEncoding("UTF-8"); //依然乱码
```

为什么乱码问题还没有解决？上述方法只是在中文转换的时候把码表设置成UTF-8，但是浏览器未必是使用UTF-8码表来显示数据的（浏览器使用GB2312显示UTF-8的数据）。

```
//设置浏览器用UTF-8编码显示数据
response.setContentType("text/html;charset=UTF-8"); //正常显示
```

下面这个方法是最简便的，它不仅设置浏览器**用UTF-8显示数据，内部还把中文转码的码表设置成UTF-8了**，也就是说，也就是说，`response.setContentType("text/html;charset=UTF-8");`把`response.setCharacterEncoding("UTF-8")`的事情也干了！

```
//设置浏览器用UTF-8编码显示数据
response.setContentType("text/html;charset=UTF-8"); //最常用的设置方法！！！
//获取到printWriter对象
PrintWriter printWriter = response.getWriter();
printWriter.write("看完博客点赞！");
```

10.3 实现文件下载

要能够给别人下载，服务器就应该有这个资源。

既然浏览器发送所有的请求都是去找Servlet的话，那么我就写一个Servlet，当别人访问我这个Servlet的时候，它们就可以下载我这个图片了！

java的文件上传下载都是通过io流来完成的。

```
//获取到资源的路径
String path = this.getServletContext().getRealPath("/download/1.png");
//读取资源
FileInputStream fileInputStream = new FileInputStream(path);
//获取到文件名,路径在电脑上保存是\\形式的。
String fileName = path.substring(path.lastIndexOf("\\") + 1);

//设置消息头，告诉浏览器，我要下载1.png这个图片
response.setHeader("Content-Disposition", "attachment; filename="+fileName);
response.setHeader("Content-Disposition", "attachment; filename=" +
URLEncoder.encode(fileName, "UTF-8"));  //如果文件名是中文需要设置编码


//把读取到的资源写给浏览器
int len = 0;
byte[] bytes = new byte[1024];
ServletOutputStream servletOutputStream = response.getOutputStream();

while ((len = fileInputStream.read(bytes)) > 0) {
    servletOutputStream.write(bytes, 0, len);
}
//关闭资源
servletOutputStream.close();
fileInputStream.close();
```

10.4 实现自动刷新

以规定的时间让页面刷新，更新资源。

```
// 每3秒自动刷新网页一次
response.setHeader("Refresh", "3");
response.getWriter().write("time is :" + System.currentTimeMillis());
```

自动刷新，能够实现页面的跳转。我们登陆完网站，很多时候都会看见【登陆成功，3秒后自动跳转....】，其实这个就是用Refresh来完成的。

```
response.setContentType("text/html;charset=UTF-8");
response.getWriter().write("3秒后跳转页面.....");

//三秒后跳转到index.jsp页面去，web应用的映射路径我设置成/，url没有写上应用名
response.setHeader("Refresh", "3;url='/index.jsp'");
```

10.5 设置缓存

浏览器本身就存在着缓存机制。

当第一次访问index.jsp时，浏览器向服务器发了两次请求【一个是网页的，一个是图片的】

当第二次访问index.jsp的时候，浏览器将图片缓存起来了！图片不是重新加载的，是从缓存里面取出来的。

像股票类型的网页是不能取缓存的数据的，数据都是要不断更新的。可以禁止缓存的功能。

```
// 禁止缓存的功能。
// 浏览器有三消息头设置缓存，为了兼容性！将三个消息头都设置了
response.setDateHeader("Expires", -1);
response.setHeader("Cache-Control","no-cache");
response.setHeader("Pragma", "no-cache");
```

如果页面有些数据不长期更新，你就将它设置成缓存，这样可以提高服务器的性能。

10.6 实现数据压缩

网页上的信息量是很大的，如果不将数据压缩再回送给浏览器，这样就十分耗费流量。

压缩的原理是什么？我们知道 getOutputStream() 和 getWriter() 都是直接把数据输出给浏览器的。现在我要做的就是让数据不直接输出给浏览器，先让我压缩了，再输出给浏览器。java提供了GZIP压缩类给我们。

```
//创建GZIPOutputStream对象，给予它ByteArrayOutputStream
ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();
GZIPOutputStream gzipOutputStream = new GZIPOutputStream(byteArrayOutputStream);

//GZIP对数据压缩，GZIP写入的数据是保存在byteArrayOutputStream上的
gzipOutputStream.write(ss.getBytes());

//gzipOutputStream有缓冲，把缓冲清了，并顺便关闭流
gzipOutputStream.close();
```

把压缩后的数据取出来，写给浏览器。

```
//将压缩的数据取出来
byte[] bytes = byteArrayOutputStream.toByteArray();
//将压缩的数据写给浏览器
response.getOutputStream().write(bytes);     //显示乱码
```

数据的确是压缩了，然而，为什么又乱码了啊？很简单，既然你压缩了数据，你写给浏览器，浏览器是不知道你这是压缩后的数据，它是以正常的方式打开数据的。这当然造成乱码啦！，现在我要告诉浏览器我这是压缩数据。

```
//告诉浏览器这是gzip压缩的数据
response.setHeader("Content-Encoding","gzip");
//再将压缩的数据写给浏览器
response.getOutputStream().write(bytes);
```

10.7 生出随机图片

生成随机图片这是非常常见的。在我们登陆的时候经常要写验证码，而那些验证码是一张图片，就是通过HttpServletResponse写给浏览器的。

要生成一张图片，java提供了BufferedImage类供我们使用。

```
//在内存中生成一张图片,宽为80,高为20，类型是RGB
BufferedImage bufferedImage = new BufferedImage(80, 20,
BufferedImage.TYPE_INT_RGB);
//获取到这张图片
Graphics graphics = bufferedImage.getGraphics();
//往图片设置颜色和字体
graphics.setColor(Color.BLUE);
graphics.setFont(new Font(null, Font.BOLD, 20));
//往图片上写数据，先写个12345，横坐标是0，纵坐标是20【高度】
graphics.drawString("12345", 0, 20);

//要往浏览器写一张图片，那要告诉浏览器回送的类型是一张图片
response.setHeader("ContentType", "jpeg");
//java提供了图片流给我们使用，这是一个工具类
//把图片传进去，类型是jpg，写给浏览器
ImageIO.write(bufferedImage, "jpg", response.getOutputStream());
```

图片数字不可能是人工写的，数字应该是随机产生的！生成随机数的方法如下：

```
private String makeNum() {
    Random random = new Random();
    //这样就会生成0-7位的随机数，现在问题又来了，如果随机数不够7位呢？如果不够7位，我们加到7位
就行了
    int anInt = random.nextInt(9999999);
    //将数字转成是字符串
    String num = String.valueOf(anInt);
    //判断位数有多少个，不够就加
    StringBuffer stringBuffer = new StringBuffer();
    for (int i = 0; i < 7 - num.length(); i++) {
        stringBuffer.append("0");
    }
    return stringBuffer.append(num).toString();
}
```

10.8 重定向跳转

什么是重定向跳转呢？点击一个超链接，通知浏览器跳转到另外的一个页面就叫重定向跳转。是通知浏览器去跳转，这很重要。

页面之间的跳转有两种方式：重定向和转发，至于什么时候用重定向，什么用转发，我在讲完HttpServletRequest对象的时候会详细说明。

```
// 重定向到index.jsp页面
response.sendRedirect("/zhongfucheng/index.jsp");
```

302状态码在http协议中代表的是临时重定向。举个例子：我找纪律委员说：给我一份请假表，我要回家。纪律委员告诉我：我这里没有请假表，你去找辅导员吧。再看回我访问Sevlet222时：我找Servlet222，Servlet222告诉浏览器：我没有你想要的资源，你要的资源在index.jsp页面中，你自己去找吧。

很容易看出重定向是通过302状态码和跳转地址实现的。于是乎，我们设置http消息头就可以实现重定向跳转。

```
//设置状态码是302
response.setStatus(302);
//HttpServletResponse把常用的状态码封装成静态常量了，所以我们可以使用
SC_MOVED_TEMPORARILY代表着302
response.setStatus(HttpServletResponse.SC_MOVED_TEMPORARILY);
//跳转的地址是index.jsp页面
response.setHeader("Location", "/zhongfucheng/index.jsp");
```

其实sendRedirect()方法就是对setStatus()和setHeader()进行封装，原理就是setStatus()和setHeader()。

10.9 getWriter和getOutputStream细节

1. getWriter() 和 getOutputStream() 两个方法**不能同时调用。**如果同时调用就会出现异常。
2. Servlet程序向ServletOutputStream或PrintWriter对象中写入的数据将被Servlet引擎从response里面获取，Servlet引擎将这些数据当作响应消息的正文，然后再与响应状态行和各响应头组合后输出到客户端。
3. Servlet的service()方法结束后【也就是doPost()或者doGet()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。

11 HttpServletRequest

HttpServletRequest对象代表客户端的请求，**当客户端通过HTTP协议访问服务器时，HTTP请求头中的所有信息都封装在这个对象中**，开发人员通过这个对象的方法，可以获得客户这些信息。

简单来说，要得到浏览器信息，就找HttpServletRequest对象。

11.1 HttpServletRequest常用方法

（浏览器信息：ip、主机名、端口等 请求头、请求参数）

1 获得客户机【浏览器】信息

- getRequestURL方法返回客户端发出请求时的完整URL。
- getRequestURI方法返回请求行中的资源名部分。
- getQueryString 方法返回请求行中的参数部分。
- 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。
- getRemoteAddr方法返回发出请求的客户机的IP地址
- getRemoteHost方法返回发出请求的客户机的完整主机名
- getRemotePort方法返回客户机所使用的网络端口号

- getLocalAddr方法返回WEB服务器的IP地址。
- getLocalName方法返回WEB服务器的主机名

2 获得客户机请求头

- getHeader方法
- getHeaders方法
- getHeaderNames方法

3 获得客户机请求参数(客户端提交的数据)

- getParameter方法
- getParameterValues（String name）方法
- getParameterNames方法
- getParameterMap方法

11.2 防盗链

什么是防盗链呢？比如：我现在有海贼王最新的资源，想要看海贼王的要在我的网页上看。现在别的网站的人看到我有海贼王的资源，想要把我的资源粘贴在他自己的网站上。这样我独家的资源就被一个CTRL+C和CTRL+V抢走了？而防盗链就是不能被他们CRTL+C和CRTL+V。

想要看我的资源，就必须经过我的首页点进去看。

想要实现这样的效果，就要获取Referer这个消息头，判断Referer是不是从我的首页来的。如果不是从我的首页来的，跳转回我的首页。

```
//获取到网页是从哪里来的
String referer = request.getHeader("Referer");
//如果不是从我的首页来或者从地址栏直接访问的，
if ( referer == null ||
!referer.contains("localhost:8080/zhongfucheng/index.jsp") ) {
    //回到首页去
    response.sendRedirect("/zhongfucheng/index.jsp");
    return;
}
//能执行下面的语句，说明是从我的首页点击进来的，那没问题，照常显示
response.setContentType("text/html;charset=UTF-8");
response.getWriter().write("路飞做了XXXXxxxxxxxxxxxxxxxx");
```

11.3 表单提交数据【通过post方式提交数据】

```
<form action="/zhongfucheng/Servlet111" method="post">
    <table>
        <tr>
            <td>用户名</td>
            <td><input type="text" name="username"></td>
        </tr>
        <tr>
            <td>密码</td>
            <td><input type="password" name="password"></td>
        </tr>
        <tr>
            <td>性别</td>
            <td>
                <input type="radio" name="gender" value="男">男
```

```
                <input type="radio" name="gender" value="女">女
            </td>
        </tr>
        <tr>
            <td>爱好</td>
            <td>
                <input type="checkbox" name="hobbies" value="游泳">游泳
                <input type="checkbox" name="hobbies" value="跑步">跑步
                <input type="checkbox" name="hobbies" value="飞翔">飞翔
            </td>
        </tr>
        <input type="hidden" name="aaa" value="my name is zhongfucheng">
        <tr>
            <td>你来自于哪里</td>
            <td>
                <select name="address">
                    <option value="广州">广州</option>
                    <option value="深圳">深圳</option>
                    <option value="北京">北京</option>
                </select>
            </td>
        </tr>
        <tr>
            <td>详细说明:</td>
            <td>
                <textarea cols="30" rows="2" name="textarea"></textarea>
            </td>
        </tr>
        <tr>
            <td><input type="submit" value="提交"></td>
            <td><input type="reset" value="重置"></td>
        </tr>
    </table>
```

在Servlet111中获取到提交的数据，代码如下

```
//设置request字符编码的格式
request.setCharacterEncoding("UTF-8");
//通过html的name属性，获取到值
String username = request.getParameter("username");
String password = request.getParameter("password");
String gender = request.getParameter("gender");
//复选框和下拉框有多个值，获取到多个值
String[] hobbies = request.getParameterValues("hobbies");
String[] address = request.getParameterValues("address");
//获取到文本域的值
String description = request.getParameter("textarea");
//得到隐藏域的值
String hiddenValue = request.getParameter("aaa");
....各种System.out.println().......
```

11.4 超链接方式提交数据

常见的get方式提交数据有：使用超链接/sendRedirect()。

```
sendRedirect("servlet的地址?参数名="+参数值 &"参数名="+参数值);
```

通过超链接将数据带给浏览器：

```
<a href="/zhongfucheng/Servlet111?username=xxx">使用超链接将数据带给浏览器</a>
```

在Servlet111接收数据：

```
// 接收以username为参数名带过来的值
String username = request.getParameter("username");
System.out.println(username);
```

服务器成功接收到浏览器发送过来的数据，并且，传输数据明文的出现在浏览器的地址栏上。

sendRedirect()和超链接类似，在这里就不赘述了。

11.5 解决中文乱码问题

在获取表单数据的时候，有这句代码`request.setCharacterEncoding("UTF-8");`.

Tomcat服务器默认编码是ISO 8859-1，而浏览器使用的是UTF-8编码。浏览器的中文数据提交给服务器，Tomcat以ISO 8859-1编码对中文编码，当在Servlet读取数据的时候，拿到的当然是乱码。而设置request的编码为UTF-8，乱码就解决了。

接下来使用get方式传递中文数据，把表单的方式改成get。（之前都是post方式）

即使把request对象设置编码为UTF-8结果还是乱码。

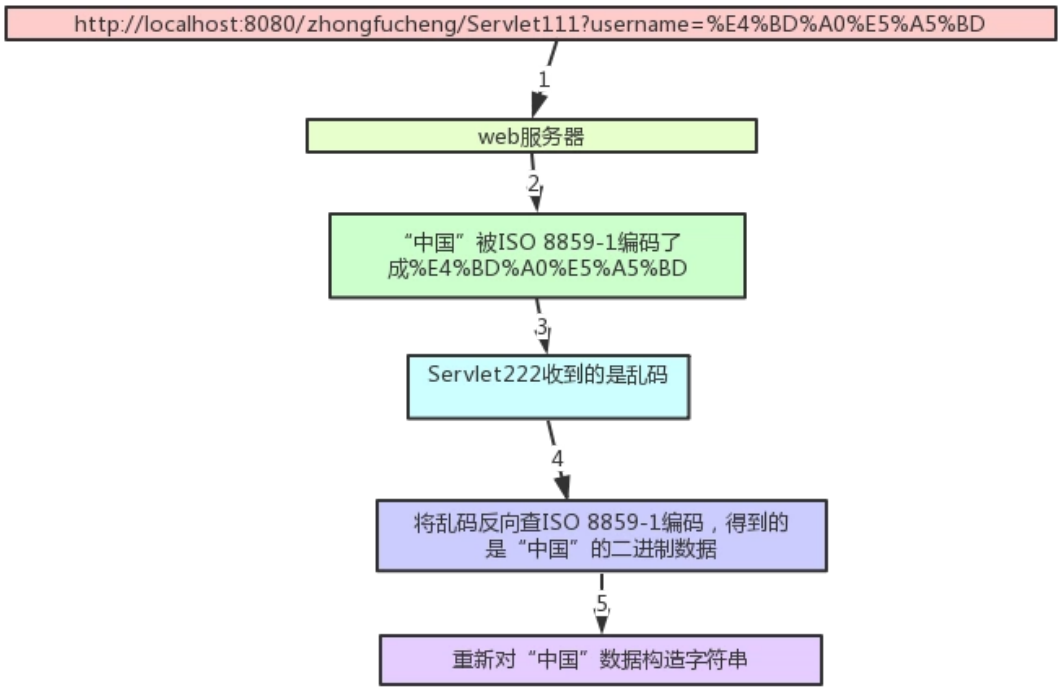
为什么post方式设置了request编码就可以解决乱码问题，而get方式不能呢?

首先我们来看一下post方法是怎么进行参数传递的。当我们点击提交按钮的时候，**数据封装进了Form Data中**，**http请求中把实体主体带过去了【传输的数据称之为实体主体】，既然**request对象封装了http请求**，所以request对象可以解析到发送过来的数据，于是只要把编码设置成UTF-8就可以解决乱码问题了。

而get方式不同，它的数据是从消息行带过去的，没有封装到request对象里面，所以使用request设置编码是无效的。

要解决get方式乱码问题也不难，我们既然知道Tomcat默认的编码是ISO 8859-1，那么get方式由消息体带过去给浏览器的时候肯定是用ISO 8859-1编码了。

```
//此时得到的数据已经是被ISO 8859-1编码后的字符串了，这个是乱码
String name = request.getParameter("username");
//乱码通过反向查ISO 8859-1得到原始的数据
byte[] bytes = name.getBytes("ISO8859-1");
//通过原始的数据，设置正确的码表，构建字符串
String value = new String(bytes, "UTF-8");
```



除了手工转换，get方式还可以改Tomcat服务器的配置来解决乱码，但是不推荐使用，这样不灵活。

我们都知道Tomcat默认的编码是ISO 8859-1,如果在Tomcat服务器的配置下改成是UTF-8的编码，那么就解决服务器在解析数据的时候造成乱码问题了。在8080端口的Connector上加入`URIEncoding="utf-8"`，设置Tomcat的访问该端口时的编码为utf-8，从而解决乱码，这种改法是固定使用UTF-8编码的。

还有有另一种改服务器编码的方式。设置Tomcat的访问该端口时的编码为页面的编码，这种改法是随着页面的编码而变。

- post方式直接改request对象的编码
- get方式需要手工转换编码
- get方式也可以修改Tomcat服务器的编码，不推荐，因为会太依赖服务器了！
- 提交数据能用post就用post

11.6 实现转发

之前讲过使用response的sendRedirect()可以实现重定向，做到的功能是页面跳转，使用request的getRequestDispatcher.forward(request,response) 实现转发，做到的功能也是页面跳转，他们有什么区别呢?

```
//获取到requestDispatcher对象，跳转到index.jsp
RequestDispatcher requestDispatcher =
request.getRequestDispatcher("/index.jsp");
//调用requestDispatcher对象的forward()实现转发,传入request和response方法
requestDispatcher.forward(request, response);
```

通过sendRedirect()重定向可以在资源尾部添加参数提交数据给服务器。那么转发能不能提交数据给服务器呢?

答案明显是可以的，并且使用这种方法非常频繁。

在讲ServletContext的时候，曾经说过Servlet之间可以通过ServletContext实现通讯，ServletContext也能称之为域对象。而request也可以称之为域对象，只不过ServletContext的域是整个web应用，而request的域仅仅代表一次http请求。

使用request实现Servlet之间的通讯：

Servlet111:

```
//以username为关键字存zhongfucheng值
request.setAttribute("username", "zhongfucheng");
//获取到requestDispatcher对象
RequestDispatcher requestDispatcher =
request.getRequestDispatcher("/Servlet222");
//调用requestDispatcher对象的forward()实现转发,传入request和response方法
requestDispatcher.forward(request, response);
```

Servlet222:

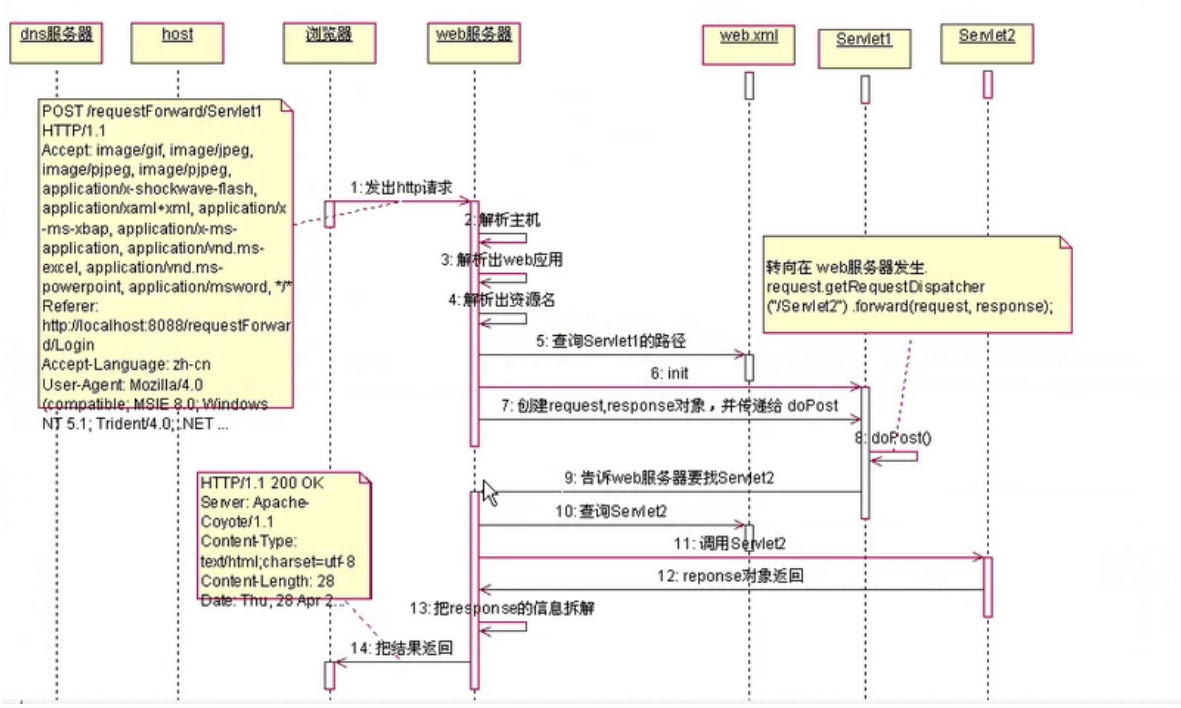
```
//获取到存进request对象的值
String userName = (String) request.getAttribute("username");
//在浏览器输出该值
response.getWriter().write("i am :"+userName);
```

Servlet222成功拿到了request对象在Servlet111存进的数据。

使用ServletContext和request实现Servlet之间的通讯，那么我们用哪一种呢?

一般的原则：可以使用request就尽可能使用request。因为ServletContext代表着整个web应用，使用ServletContext会消耗大量的资源，而request对象会随着请求的结束而结束，资源会被回收。使用request域进行Servlet之间的通讯在开发中是非常频繁的。

11.7 转发的时序图



11.8 请求转发的细节

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。 也就是说：不要在转发之前写数据给浏览器。

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，但是，已写入到HttpServletResponse对象中的响应头字段信息保持有效。

11.9 转发和重定向的区别

1 实际发生位置不同，地址栏不同

- **转发是发生在服务器的**

  转发是由服务器进行跳转的，细心的朋友会发现，在转发的时候，浏览器的地址栏是没有发生变化的，在我访问Servlet111的时候，即使跳转到了Servlet222的页面，浏览器的地址还是Servlet111的。也就是说浏览器是不知道该跳转的动作，转发是对浏览器透明的。通过上面的转发时序图我们也可以发现，实现转发只是一次的http请求，一次转发中request和response对象都是同一个。这也解释了，为什么可以使用request作为域对象进行Servlet之间的通讯。

- **重定向是发生在浏览器的**

  重定向是由浏览器进行跳转的，进行重定向跳转的时候，浏览器的地址会发生变化的。曾经介绍过：实现重定向的原理是由response的状态码和Location头组合而实现的。这是由浏览器进行的页面跳转实现重定向会发出两个http请求，request域对象是无效的，因为它不是同一个request对象。

2 用法不同

很多人都搞不清楚转发和重定向的时候，资源地址究竟怎么写。有的时候要把应用名写上，有的时候不用把应用名写上。很容易把人搞晕。记住一个原则：**给服务器用的直接从资源名开始写，给浏览器用的要把应用名写上**

- request.getRequestDispatcher("/资源名 URI").forward(request,response)

  转发时"/"代表的是本应用程序的根目录【zhongfucheng】

- response.send("/web应用/资源名 URI");

  重定向时"/"代表的是webapps目录

3 能够去往的URL的范围不一样

- 转发是服务器跳转只能去往当前web应用的资源
- 重定向是服务器跳转，可以去往任何的资源

4 传递数据的类型不同

- 转发的request对象可以传递各种类型的数据，包括对象
- 重定向只能传递字符串

5 跳转的时间不同

- 转发时：执行到跳转语句时就会立刻跳转
- 重定向：整个页面执行完之后才执行跳转

根据上面说明了转发和重定向的区别也可以很容易概括出来。转发是带着转发前的请求的参数的。重定向是新的请求。

典型的应用场景：

1. 转发: 访问 Servlet 处理业务逻辑，然后 forward 到 jsp 显示处理结果，浏览器里 URL 不变
2. 重定向: 提交表单，处理成功后 redirect 到另一个 jsp，防止表单重复提交，浏览器里 URL 变了

11.10 RequestDispatcher再说明

RequestDispatcher对象调用forward()可以实现转发上面已经说过了。RequestDispatcher还有另外一个方法include()，该方法可以实现包含，有什么用呢?

我们在写网页的时候，一般网页的头部和尾部是不需要改变的。如果我们多个地方使用Servlet输出网头和网尾的话，需要把代码重新写一遍。而使用RequestDispatcher的include()方法就可以实现包含网头和网尾的效果了。

```
request.getRequestDispatcher("/Head").include(request, response);
response.getWriter().write("-------------------------------------------");
request.getRequestDispatcher("/Foot").include(request, response);
```

访问一下Servlet111,成功把网头和网尾包含了。

12 Cookie

会话技术

基本概念: 指用户开一个浏览器，访问一个网站，只要不关闭该浏览器，不管该用户点击多少个超链接，访问多少资源，直到用户关闭浏览器，整个这个过程我们称为一次会话。

会话技术的用途

- 在论坛登陆的时候，很多时候会有一个小框框问你是否要自动登陆，当你下次登陆的时候就不用输入密码了
- 根据我以前浏览过的商品，猜我喜欢什么商品

12.1 什么是Cookie

会话跟踪技术有Cookie和Session，Cookie技术是先出现的。

网页之间的交互是通过HTTP协议传输数据的，而Http协议是无状态的协议。无状态的协议是什么意思呢？一旦数据提交完后，浏览器和服务器的连接就会关闭，再次交互的时候需要重新建立新的连接。

服务器无法确认用户的信息，于是W3C就提出了：给每一个用户都发一个通行证，无论谁访问的时候都需要携带通行证，这样服务器就可以从通行证上确认用户的信息。通行证就是Cookie。

Cookie的流程：浏览器访问服务器，如果服务器需要记录该用户的状态，就使用response向浏览器发送一个Cookie，浏览器会把Cookie保存起来。当浏览器再次访问服务器的时候，浏览器会把请求的网址连同Cookie一同交给服务器。

12.2 Cookie API

- Cookie类用于创建一个Cookie对象
- response接口中定义了一个addCookie方法，它用于在其响应头中增加一个相应的Set-Cookie头字段
- request接口中定义了一个getCookies方法，它用于获取客户端提交的Cookie

常用的Cookie方法:

- public Cookie(String name,String value)

- setValue与getValue方法
- setMaxAge与getMaxAge方法
- setPath与getPath方法
- setDomain与getDomain方法
- getName方法

12.3 简单使用Cookie

创建Cookie对象，发送Cookie给浏览器。

```
//设置response的编码
response.setContentType("text/html;charset=UTF-8");
//创建Cookie对象，指定名称和值
Cookie cookie = new Cookie("username", "zhongfucheng");
//向浏览器给一个Cookie
response.addCookie(cookie);
response.getWriter().write("我已经向浏览器发送了一个Cookie");
```

浏览器本身没有任何Cookie。

访问Servlet1，再回到文件夹中，还是没有发现Cookie，这是为什么呢？我明明向浏览器发送了一个Cookie的。

原因是发送Cookie给浏览器是需要设置Cookie的时间的。在给浏览器之前，设置一下Cookie的时间。

```
//设置Cookie的时间
cookie.setMaxAge(1000);
```

12.4 Cookie细节

1 Cookie不可跨域名性

在访问Servlet的时候浏览器是不是把所有的Cookie都带过去给服务器，会不会修改了别的网站的Cookie?

答案是否定的。Cookie具有不可跨域名性。浏览器判断一个网站是否能操作另一个网站的Cookie的依据是域名。所以一般来说，当我访问baidu的时候，浏览器只会把baidu颁发的Cookie带过去，而不会带上google的Cookie。

2 Cookie保存中文

```
response.setContentType("text/html;charset=UTF-8");
PrintWriter printWriter = response.getWriter();

String name = "中国";
Cookie cookie = new Cookie("country", name);
cookie.setMaxAge(2000);
response.addCookie(cookie);

printWriter.write("我颁发了一个Cookie，值保存的是中文数据");     //访问出现了异常
```

中文属于Unicode字符，英文数据ASCII字符，中文占4个字符或者3个字符，英文占2个字符，所以会出现异常。

解决方法：Cookie使用Unicode字符时需要对Unicode字符进行编码。

```
//对Unicode字符进行编码
Cookie cookie = new Cookie("country", URLEncoder.encode(name, "UTF-8"));
```

我们发现Cookie保存在硬盘的中文数据是经过编码的，那么我们在取出Cookie的时候要对中文数据进行解码。

```
Cookie[] cookies = request.getCookies();
for (int i = 0; cookies != null && i < cookies.length; i++) {
    String name = cookies[i].getName();

    //经过URLEncoding就要URLDecoding
    String value = URLDecoder.decode(cookies[i].getValue(), "UTF-8");

    printWriter.write(name + "------" + value);
}
```

3 Cookie的有效期

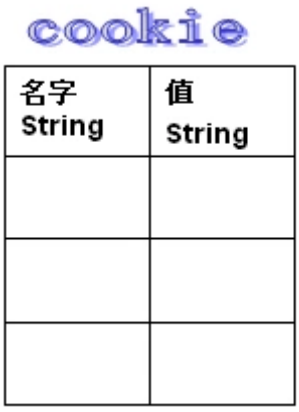
Cookie的有效期是通过setMaxAge()来设置的。

- 如果MaxAge为正数，浏览器会把Cookie写到硬盘中，只要还在MaxAge秒之前，登陆网站时该Cookie就有效【不论关闭了浏览器还是电脑】
- 如果MaxAge为负数，Cookie是临时性的，仅在本浏览器内有效，关闭浏览器Cookie就失效了，Cookie不会写到硬盘中。Cookie默认值就是-1。这也就为什么在我第一个例子中，如果我没设置Cookie的有效期，在硬盘中就找不到对应的文件。
- 如果MaxAge为0，则表示删除该Cookie。Cookie机制没有提供删除Cookie对应的方法，把MaxAge设置为0等同于删除Cookie

4 Cookie的修改和删除

上面我们已经知道了Cookie机制没有提供删除Cookie的方法。其实细心点我们可以发现，Cookie机制也没有提供修改Cookie的方法。那么我们怎么修改Cookie的值呢?

Cookie存储的方式类似于Map集合，如下图所示：



修改：Cookie的名称相同，通过response添加到浏览器中，会覆盖原来的Cookie。

删除：要删除该Cookie，把MaxAge设置为0，并添加到浏览器中即可。

注意：删除，修改Cookie时，新建的Cookie除了value、maxAge之外的所有属性都要与原Cookie相同。否则浏览器将视为不同的Cookie，不予覆盖，导致删除修改失败!

5 Cookie的域名

Cookie的domain属性决定运行访问Cookie的域名。domain的值规定为“.域名”

Cookie的隐私安全机制决定Cookie是不可跨域名的。也就是说www.baidu.com和www.google.com之间的Cookie是互不交接的。即使是同一级域名，不同二级域名也不能交接，也就是说：www.goole.com和www.image.goole.com的Cookie也不能访问。

如果希望一级域名相同的网页Cookie之间可以相互访问。也就是说www.image.zhongfucheng.com可以获取到www.zhongfucheng.com的Cookie就需要使用到domain方法。

```
Cookie cookie = new Cookie("name", "ouzicheng");
cookie.setMaxAge(1000);
cookie.setDomain(".zhongfucheng.com");
response.addCookie(cookie);
printWriter.write("使用www.zhongfucheng.com域名添加了一个Cookie,只要一级是
zhongfucheng.com即可访问");
```

6 Cookie的路径

Cookie的path属性决定允许访问Cookie的路径。

一般地，Cookie发布出来，整个网页的资源都可以使用。现在我只想Servlet1可以获取到Cookie，其他的资源不能获取。

使用Servlet2颁发一个Cookie给浏览器,设置路径为"/Servlet1"。

```
Cookie cookie = new Cookie("username", "java");
cookie.setPath("/Servlet1");
cookie.setMaxAge(1000);
response.addCookie(cookie);
printWriter.write("该Cookie只有Servlet1获取得到");
```

7 Cookie的安全属性

HTTP协议不仅仅是无状态的，而且是不安全的！如果不希望Cookie在非安全协议中传输，可以设置Cookie的secure属性为true，浏览器只会在HTTPS和SSL等安全协议中传输该Cookie。

当然了，设置secure属性不会将Cookie的内容加密。如果想要保证安全，最好使用md5算法加密【后面有】。

12.5 Cookie的应用

1 显示用户上次访问的时间

其实就是每次登陆的时候，取到Cookie保存的值，再更新下Cookie的值。

按照正常的逻辑来写，程序流程应该是这样子的。先创建Cookie对象，回送Cookie给浏览器。再遍历Cookie，更新Cookie的值。

但是，按照上面的逻辑是做不到的！因为每次访问Servlet的时候都会覆盖原来的Cookie，取到Cookie的值永远都是当前时间，而不是上次保存的时间。

我们换一个逻辑写：先检查（遍历）所有Cookie有没有我要的，如果得不到我想要的Cookie，Cookie的值是null，那么就是第一次登陆，于是就有了下面的代码了。

```
SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
response.setContentType("text/html;charset=UTF-8");
PrintWriter printWriter = response.getWriter();
//获取网页上所有的Cookie
Cookie[] cookies = request.getCookies();
//判断Cookie的值是否为空
String cookieValue = null;
for (int i = 0; cookies != null && i < cookies.length; i++) {

    //获取到以time为名的Cookie
    if (cookies[i].getName().equals("time")) {
        printWriter.write("您上次登陆的时间是：");
        cookieValue = cookies[i].getValue();
        printWriter.write(cookieValue);

        cookies[i].setValue(simpleDateFormat.format(new Date()));
        response.addCookie(cookies[i]);

        //既然已经找到了就可以break循环了
        break;
    }
}
//如果Cookie的值是空的，那么就是第一次访问
if (cookieValue == null) {
    //创建一个Cookie对象，日期为当前时间
    Cookie cookie = new Cookie("time", simpleDateFormat.format(new Date()));
    //设置Cookie的生命期
    cookie.setMaxAge(20000);
    //response对象回送Cookie给浏览器
    response.addCookie(cookie);
    printWriter.write("您是第一次登陆啊！");
}
```

2 显示上次浏览过商品

以浏览书籍信息为例。

1. 首先定义Book类表示书籍，其中包含id、name、author等信息。
2. 设计一个简单的数据库存储书籍信息，使用LinkedHashMap集合表示。
3. 编写Servlet，将存储的所有书籍信息显示到网页上，同时给显示的书籍挂上超链接使其可点击，点击后跳转到书籍的详细信息页面。超链接应该把书的id传递过去，不然处理页面是不知道用户想看的是哪一本书的！
4. 在书籍详细页面接收id，找到用户想看的书，输出该书的详细信息。

既然用户点击了书籍，那么服务器就应该颁发Cookie给浏览器，记住用户点击了该书籍。

Cookie的值应该是什么呢?

- 待会还要把浏览过的书籍显示出来，所以用书籍的id是最好不过的。
- 浏览了非常多的书籍，只显示3本最近浏览过的书籍。
- 书籍的id都是数字，要把存储到Cookie的书籍id分割起来。所以定义"_"作为分隔符。

总体的逻辑应该是：先遍历下Cookie，看下有没有我们想要的Cookie。如果找到想要的Cookie，那就取出Cookie的值。

```
String bookHistory = null;
Cookie[] cookies = request.getCookies();
for (int i = 0; cookies != null && i < cookies.length; i++) {
    if (cookies[i].getName().equals("bookHistory")) {
        bookHistory = cookies[i].getValue();
    }
}
```

取出了Cookie的值也分几种情况

1. Cookie的值为null【直接把传入进来的id当做是Cookie的值】
2. Cookie的值长度有3个了【把排在最后的id去掉，把传进来的id排在最前边】
3. Cookie的值已经包含有传递进来的id了【把已经包含的id先去掉，再把id排在最前面】
4. Cookie的值就只有1个或2个，直接把id排在最前边

```
// cookie的值为空
if (bookHistory == null) {
    return id;
}
//如果Cookie的值不是null的，那么就分解Cookie的得到之前的id。
String[] strings = bookHistory.split("\\_");
//为了增删容易并且还要判断id是否存在于该字符串内-----我们使用LinkedList集合装载分解出来的id
List list = Arrays.asList(strings);
LinkedList<String> linkedList = new LinkedList<>();
linkedList.addAll(list);

if (linkedList.contains(id)) {
    linkedList.remove(id);
    linkedList.addFirst(id);
}else {
    if (linkedList.size() >= 3) {
        linkedList.removeLast();
        linkedList.addFirst(id);
    } else {
        linkedList.addFirst(id);
```

```
    }
}
```

Cookie值就在LinkedList集合里边了。接下来，要做的就是把集合中的值取出来，拼接成一个字符串。

```
StringBuffer stringBuffer = new StringBuffer();
//遍历LinkedList集合，添加个下划线“_”
for (String s : linkedList) {
    stringBuffer.append(s + "_");
}
//最后一个元素后面就不需要下划线了
return stringBuffer.deleteCharAt(stringBuffer.length() - 1).toString();
```

接下来设置Cookie的生命周期，回送给浏览器即可。

```
String bookHistory = makeHistory(request, id);
Cookie cookie = new Cookie("bookHistory", bookHistory);
cookie.setMaxAge(30000);
response.addCookie(cookie);
```

接下来在首页上获取Cookie的值，显示用户浏览过什么商品。

```
printWriter.write("您曾经浏览过的商品：");
printWriter.write("<br/>");
//显示用户浏览过的商品
Cookie[] cookies = request.getCookies();
for (int i = 0; cookies != null && i < cookies.length; i++) {
    if (cookies[i].getName().equals("bookHistory")) {
        //获取到的bookHistory是2_3_1之类的
        String bookHistory = cookies[i].getValue();
        //拆解成每一个id值
        String[] ids = bookHistory.split("\\_");
        //得到每一个id值
        for (String id : ids) {
            //通过id找到每一本书
            Book book = linkedHashMap.get(id);
            printWriter.write(book.getName());
            printWriter.write("<br/>");
        }
        break;
    }
}
```

13 Session

13.1 什么是Session

Session 是另一种记录浏览器状态的机制。不同的是Cookie保存在浏览器中，Session保存在服务器中。用户使用浏览器访问服务器的时候，服务器把用户的信息以某种的形式记录在服务器，这就是Session。

如果说Cookie是检查用户身上的”通行证“来确认用户的身份，那么Session就是通过检查服务器上的”客户明细表“来确认用户的身份的。Session相当于在服务器中建立了一份“客户明细表”。

Session比Cookie使用方便，Session可以解决Cookie解决不了的事情【Session可以存储对象，Cookie只能存储字符串】

13.2 Session API

- long getCreationTime();【获取Session被创建时间】
- String getId();【获取Session的id】
- long getLastAccessedTime();【返回Session最后活跃的时间】
- ServletContext getServletContext();【获取ServletContext对象】
- void setMaxInactiveInterval(int var1);【设置Session超时时间】
- int getMaxInactiveInterval();【获取Session超时时间】
- Object getAttribute(String var1);【获取Session属性】
- Enumeration<String> getAttributeNames();【获取Session所有的属性名】
- void setAttribute(String var1, Object var2);【设置Session属性】
- void removeAttribute(String var1);【移除Session属性】
- void invalidate();【销毁该Session】
- boolean isNew();【该Session是否为新的】

13.3 Session作为域对象

从上面的API看出，Session有着request和ServletContext类似的方法。其实Session也是一个域对象。Session作为一种记录浏览器状态的机制，只要Session对象没有被销毁，Servlet之间就可以通过Session对象实现通讯。

一般来讲，当我们要存进的是用户级别的数据就用Session，那什么是用户级别呢？只要浏览器不关闭，希望数据还在，就使用Session来保存。

13.4 Session的生命周期和有效期

Session在用户第一次访问服务器Servlet，jsp等动态资源就会被自动创建，Session对象保存在内存里，这也就为什么上面的例子可以直接使用request对象获取得到Session对象。如果访问HTML,IMAGE等静态资源Session不会被创建。

Session生成后，只要用户继续访问，服务器就会更新Session的最后访问时间，无论是否对Session进行读写，服务器都会认为Session活跃了一次。

由于会有越来越多的用户访问服务器，因此Session也会越来越多。为了防止内存溢出，服务器会把长时间没有活跃的Session从内存中删除，这个时间也就是Session的超时时间。

Session的超时时间默认是30分钟，有三种方式可以对Session的超时时间进行修改。

第一种方式：在tomcat/conf/web.xml文件中设置，时间值为20分钟，所有的WEB应用都有效。

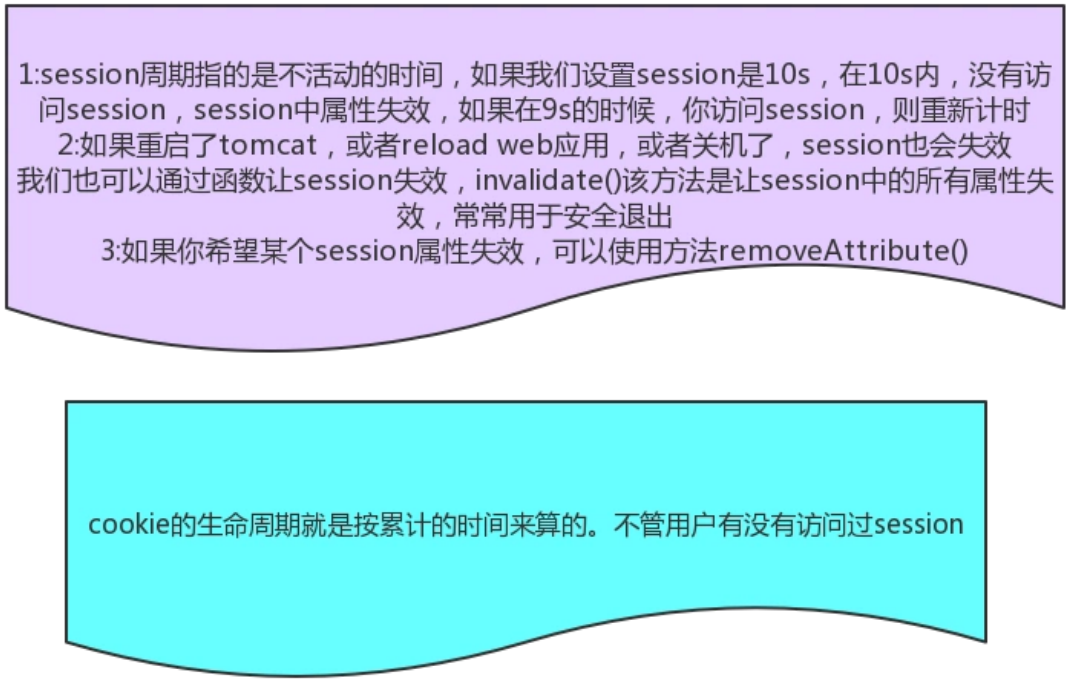
```
<session-config>
    <session-timeout>20</session-timeout>
</session-config>
```

第二种方式：在单个的web.xml文件中设置，对单个web应用有效，如果有冲突，以自己的web应用为准。

第三种方式：通过setMaxInactiveInterval()方法设置。

```
//设置Session最长超时时间为60秒，这里的单位是秒
httpSession.setMaxInactiveInterval(60);
System.out.println(httpSession.getMaxInactiveInterval());
```

Session的有效期与Cookie的是不同的。



13.5 Session的实现原理

问题：服务器是如何实现一个session为一个用户浏览器服务的？换个说法：为什么服务器能够为不同的用户浏览器提供不同session?

HTTP协议是无状态的，Session不能依据HTTP连接来判断是否为同一个用户。于是服务器向用户浏览器发送了一个名为JESSIONID的Cookie，它的值是Session的id值。其实Session依据Cookie来识别是否是同一个用户。

简单来说：Session 之所以可以识别不同的用户，依靠的就是Cookie。

该Cookie是服务器自动颁发给浏览器的，不用我们手工创建的。该Cookie的maxAge值默认是-1，也就是说仅当前浏览器使用，不将该Cookie存在硬盘中。

流程：

1. 当我们访问Servlet1的时候，服务器就会创建一个Session对象，执行我们的程序代码，并自动颁发个Cookie给用户浏览器。
2. 当我们用同一个浏览器访问Servlet2的时候，浏览器会把Cookie的值通过http协议带过去给服务器，服务器就知道用哪一Session。
3. 而当我们使用新会话的浏览器访问Servlet2的时候，该新浏览器并没有Cookie，服务器无法辨认使用哪一个Session，所以就获取不到值。

13.6 浏览器禁用了Cookie，Session还能用吗?

上面说了Session是依靠Cookie来识别用户浏览器的。如果我的用户浏览器禁用了Cookie了呢？绝大多数的手机浏览器都不支持Cookie，那我的Session怎么办?

用户浏览器访问Servlet1的时候，服务器向用户浏览器颁发了一个Cookie。

但是当用户浏览器访问Servlet2的时候，由于我们禁用了Cookie，所以用户浏览器并没有把Cookie带过去给服务器。

解决方案：**URL地址重写**

原则：把Session的属性带过去【传递给】另外一个Servlet，都要URL地址重写。

HttpServletResponse类提供了两个URL地址重写的方法：

- **encodeURL(String url)**
- **encodeRedirectURL(String url)**

这两个方法会自动判断该浏览器是否支持Cookie，如果支持Cookie，重写后的URL地址就不会带有jsessionid了【即使浏览器支持Cookie，第一次输出URL地址的时候还是会出现jsessionid（因为没有任何Cookie可带）】

```
String url = "/ouzicheng/Servlet2";
response.sendRedirect(response.encodeURL(url));
```

Session的id通过URL地址重写，使用的是同一个Session。

URL地址重写的原理：将Session的id信息重写到URL地址中。服务器解析重写后URL，获取Session的id。这样一来，即使浏览器禁用掉了Cookie，但Session的id通过服务器端传递，还是可以使用Session来记录用户的状态。

13.7 Session禁用Cookie

禁用自己项目的Cookie：在META-INF文件夹下的context.xml文件中修改（没有则创建）

```
<Context path="/ouzicheng" cookies="false">
</Context>
```

禁用全部web应用的Cookie：在conf/context.xml中修改

注意：该配置只是让服务器不能自动维护名为jsessionid的Cookie，并不能阻止Cookie的读写。

13.8 Session案例

1 使用Session完成用户简单登陆

1. 创建代表用户的User类
2. 使用简单的集合List模拟一个数据库
3. 在JSP文件中进行表单提交
4. 在Servlet中获取到表单提交的数据，查看数据库中是否有相对应的用户名和密码，如果没有就提示出错，如果有就跳转到另外一个界面

2 利用Session防止表单重复提交

重复提交的危害：

- 在投票的网页上不停地提交，实现了刷票的效果。
- 注册多个用户，不断发帖子，扰乱正常发帖秩序。

首先我们来看一下常见的重复提交。

- 在处理表单的Servlet中刷新。
- 后退再提交
- 网络延迟，多次点击提交按钮

网络延迟解决方法：

1. 当用户第一次点击提交按钮时，把数据提交给服务器。当用户再次点击提交按钮时，就不把数据提交给服务器了。监听用户提交事件。只能让用户提交一次表单。
2. 当点击过一次提交按钮时，就把提交的按钮隐藏起来。不能让用户点击了！

在处理表单的Servlet中刷新和后退再提交这两种方式不能只靠客户端来限制了。也就是说javaScript代码无法阻止这两种情况的发生。

Session可以用来标识一个用户是否登陆了。

Session的原理说明了不同的用户浏览器会拥有不同的Session。而request和ServletContext为什么就不行呢？request的域对象只能是一次http请求，提交表单数据的时候request域对象的数据取不出来。ServletContext代表整个web应用，如果有几个用户浏览器同时访问，ServletContext域对象的数据会被多次覆盖掉，也就是说域对象的数据就毫无意义了。

在提交数据的时候，存进Session域对象的数据，在处理提交数据的Servlet中判断Session域对象数据?。究竟判断Session什么？判断Session域对象的数据不为null？没用呀，既然已经提交过来了，那肯定不为null。

在表单中还有一个**隐藏域**，可以通过隐藏域把数据交给服务器。

- 判断Session域对象的数据和jsp隐藏域提交的数据是否对应。
- 判断隐藏域的数据是否为空【如果为空，就是直接访问表单处理页面的Servlet】
- 判断Session的数据是否为空【servlet判断完是否重复提交，最好能立马移除Session的数据，不然还没有移除的时候，客户端那边儿的请求又来了，就又能匹配了，产生了重复提交。如果Session域对象数据为空，证明已经提交过数据了！】

我们向Session域对象的存入数据究竟是什么呢？简单的一个数字？好像也行啊。因为只要Session域对象的数据和jsp隐藏域带过去的数据对得上号就行了呀，反正在Servlet上判断完是否重复提交，会立马把Session的数据移除掉的。更专业的做法是：向Session域对象存入的数据是一个随机数【Token--令牌】。

实现原理是非常简单的：

- 在session域中存储一个token
- 然后前台页面的隐藏域获取得到这个token
- 在第一次访问的时候，我们就判断seesion有没有值，如果有就比对。对比正确后我们就处理请求，接着就把session存储的数据给删除了
- 等到再次访问的时候，我们session就没有值了，就不受理前台的请求了！

3 一次性校验码

一次性校验码其实就是为了防止暴力猜测密码。

在讲response对象的时候，我们使用response对象输出过验证码，但是没有去验证！

验证的原理也非常简单：生成验证码后，把验证码的数据存进Session域对象中，判断用户输入验证码是否和Session域对象的数据一致。

对于校验码实现思路是这样子的：

- 使用awt语法来描写一张验证码，生成随机数保存在seesion域中，我们让验证码不能缓存起来【做到验证码都不一样】

- 页面直接访问Servlet来获取我们的验证码，于是我们验证码的值就会改变【同时session的值也会被改变】
- 当用户验证的时候，就是session内的值的验证了。

13.9 Session和Cookie的区别

从存储方式上比较

- Cookie只能存储字符串，如果要存储非ASCII字符串还要对其编码。
- Session可以存储任何类型的数据，可以把Session看成是一个容器

从隐私安全上比较

- Cookie存储在浏览器中，对客户端是可见的。信息容易泄露出去。如果使用Cookie，最好将Cookie加密
- Session存储在服务器上，对客户端是透明的。不存在敏感信息泄露问题。

从有效期上比较

- Cookie保存在硬盘中，只需要设置maxAge属性为比较大的正整数，即使关闭浏览器，Cookie还是存在的
- Session的保存在服务器中，设置maxInactiveInterval属性值来确定Session的有效期。并且Session依赖于名为JSESSIONID的Cookie，该Cookie默认的maxAge属性为-1。如果关闭了浏览器，该Session虽然没有从服务器中消亡，但也就失效了。

从对服务器的负担比较

- Session是保存在服务器的，每个用户都会产生一个Session，如果是并发访问的用户非常多，是不能使用Session的，Session会消耗大量的内存。
- Cookie是保存在客户端的。不占用服务器的资源。像baidu、Sina这样的大型网站，一般都是使用Cookie来进行会话跟踪。

从浏览器的支持上比较

- 如果浏览器禁用了Cookie，那么Cookie是无用的了！
- 如果浏览器禁用了Cookie，Session可以通过URL地址重写来进行会话跟踪。

从跨域名上比较

- Cookie可以设置domain属性来实现跨域名
- Session只在当前的域名内有效，不可跨域名

13.10 Cookie和Session共同使用

如果仅仅使用Cookie或仅仅使用Session可能达不到理想的效果。这时应该尝试一下同时使用Session和Cookie。

问题：我在购物的途中，不小心关闭了浏览器。当我再返回进去浏览器的时候，发现我购买过的商品记录都没了！！为什么会没了呢？原因也非常简单：服务器为Session自动维护的Cookie的maxAge属性默认是-1的，当浏览器关闭掉了，该Cookie就自动消亡了。当用户再次访问的时候，已经不是原来的Cookie了。

想达到的效果：即使我不小心关闭了浏览器了，我重新进去网站，我还能找到我的购买记录。

服务器为Session自动维护的Cookie的maxAge属性是-1，Cookie没有保存在硬盘中。我现在要做的就是：把Cookie保存在硬盘中，即使我关闭了浏览器，浏览器再次访问页面的时候，可以带上Cookie，从而服务器识别出Session。

第一种方式：只需要在处理购买页面上创建Cookie，Cookie的值是Session的id返回给浏览器即可

第二种方式：在server.xml文件中配置，将每个用户的Session在服务器关闭的时候序列化到硬盘或数据库上保存。