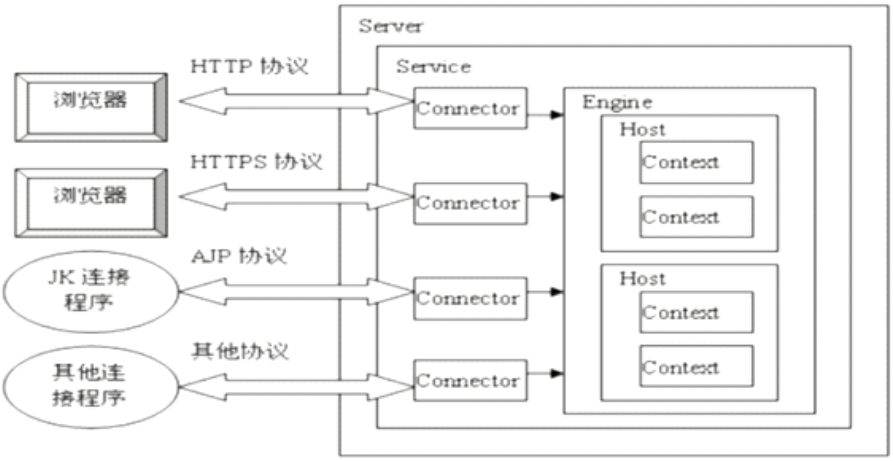
服务器

2018年5月30日　　22:23

服务器的作用：客户端向服务器发出请求，服务器响应请求并将数据返回给客户端。
常用的服务器是 Tomcat,它是一个 Web 容器，后端项目都要部署到 Web 容器才能运行，它其实是一个遵循 Http，通过 Socket 通信与客户端进行交互的服务端程序。
一、启动 Tomcat
1.Tomcat 启动方法：安装目录→bin文件夹→startup，浏览器内输入 localhost:8080 ，如果能访问到 Tomcat 页面，就说明启动成功。
2.Tomcat 安装目录下有一个 config 文件夹，server.xml 文件是 Tomcat 的配置信息，里面有个 Connector 标签，标签的 port 指定了端口，可以自行修改。
3.可以配置 Catilina_home 环境变量，这个环境变量指向的是 Tomcat 的安装目录。如果计算机上安装了多个 Tomcat，那么启动时启动的就是 Catilina_home 环境变量指向的 Tomcat。

二、Tomcat 目录结构
1.bin：启动和关闭 Tomcat 的脚本文件
2.conf：Tomcat 的配置文件
3.lib：支撑 Tomcat 的 jar 包
4.logs：日志
5.temp：运行时产生的临时文件
6.webapps：Web 应用所在的目录，即外界访问的 Web 资源的存放目录。对 Web 开发人员而言最为重要的目录。
7.work：工作目录

三、体系架构



四、登录管理平台
当 Tomcat 上发布了多个 Web 应用时，可以通过 Tomcat 的管理平台管理这些应用。
1.打开 $CATALINA_BASE/conf/tomcat-users.xml 文件，其中有几个 user 标签。可以设置用户的用户名和密码；还有一个 roles 属性，指定这个用户的角色，设置为以下四个值的一个或多个的才是管理员：

- manager-gui：允许访问 HTML 图形界面和服务器状态页面
- manager-script：允许访问文本接口和状态页面
- manager-jmx：允许访问 JMX 代理和状态页面
- manager-status：仅允许访问状态页面

2.其中有个 List Applications 按钮，点击之后就可以看到当前服务器上的 Web 应用列表。
3.有一个 WAR file to deploy 的列表项，使用这个功能可以通过 .war 文件方便地部署一个 Web 应用。
注意，管理员的用户名和密码一定要注意保密。

Web 应用

2018年6月1日　　18:07

一、概述

供浏览器访问的程序，通常称为 Web 应用。一个 Web 应用由多个静态资源和动态资源组成，比如：HTML 文件、CSS 文件、JavaScript 文件、Java 文件、配置文件等等。这些所有的文件都会放在一个目录下，称为 Web 应用所在目录。 把 Web 应用交给服务器管理的过程，称为虚拟目录的映射。

假设应用的所有文件放在 C:/myweb 下，设置虚拟目录映射的两种方法：

1.通过 XML 文件进行配置

在 $CATALINA_BASE/conf/[enginename]/[hostname]/ 目录下新建一个 XML 文件，里面写上：

```
<Context docBase="C:/myweb"/>
```

其中 docBase 属性指向的就是 Web 应用所在目录。然后服务器就会把这个 XML 文件的名称作为所谓的虚拟目录。比如说这个 XML 文件命名为 web.xml ，那么浏览器想要访问 C:/myweb/index.html 的话，输入的 URL 就是：

http://localhost:8080/web/index.html

实际上，通过 XML 文件进行配置的方案还有多种，可以在 Tomcat 文档中找到。

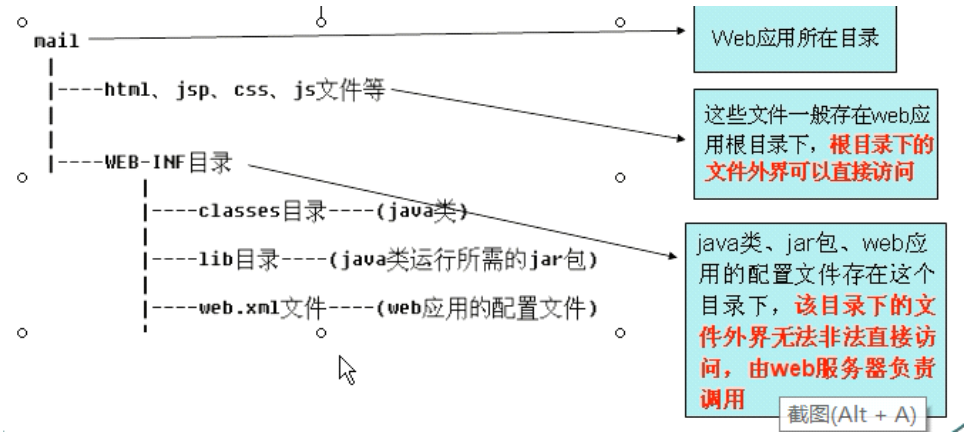
Tip：Context 标签有个 reloadable 属性，如果设置 reloadable="true"，那么 Tomcat 会自动加载更新后的 Web 应用。小型 Web 应用开启这个功能，但是大型应用不建议开启。

2.直接把 Web 应用所在目录放在 $CATALINA_BASE/webapps 目录下。Web 应用所在目录的名称就是虚拟目录。这种方法的局限性在于 Web 应用和服务器必须在同一个根目录下。

虚拟目录的配置是否成功，可以查看日志信息。

二、 Web 应用的组织结构

Web 应用中不同类型的文件有严格的存放规则，如果不按规则进行存放，轻则无法访问 Web 应用，重则服务器报错。



注意，web.xml 文件可以用来进行 Web 应用的所有配置，这是一个非常重要的文件。classes 目下存放的是 .class 文件而不是 .java 文件。

三、web.xml 文件

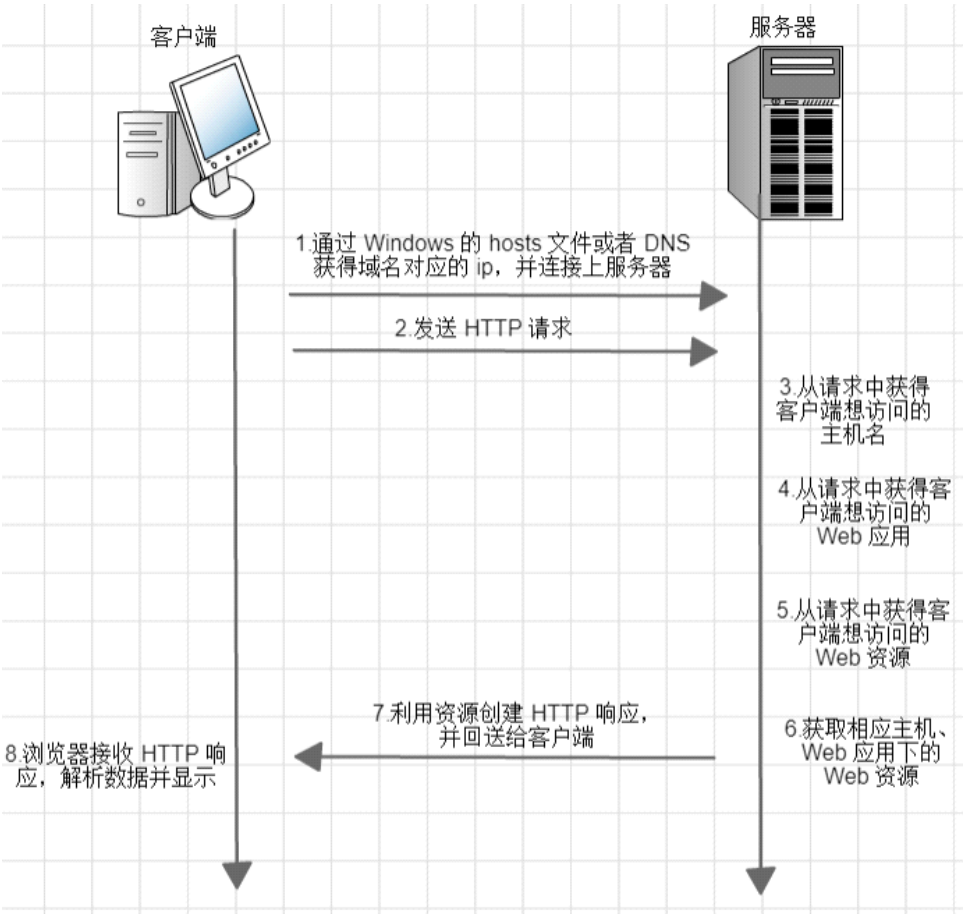
在开发 Web 应用时，通常是利用 web.xml 文件对资源进行配置。例如把某个页面设置为首页，为应用设置监听器、过滤器等等。

四、配置虚拟主机
虚拟主机就是在 Tomcat 服务器中配置一个网站，比如说配置网站 www.google.com。 一个服务器上可以配置多个网站。
配置方法：
1.在 $CATALINA_BASE/conf/server.xml 文件中，添加 <Host> 标签：

```
<Host name="www.myweb.com" appBase="C:/myweb">
      <Context path="/mail" docBase="C:/myweb/mail"/>
</Host>
```

name 属性就是网站名称，appBase 属性指向 Web 应用所在的根目录。子标签 <Context> 则配置了 Web 应用中不同的子应用。
2.如果想让网站被外部访问，还要进行注册。首先需要了解浏览器访问一个网站的基本流程，浏览器得到一个域名之后，首先在 Windows 系统的 hosts 文件查找有没有这个域名对应的 ip，如果有则访问；如果没有，则向 DNS 发出请求，由 DNS 解析域名得到 ip ，然后再访问。因此，网站必须在 DNS 或者 Windows 中注册才能被外部访问。
（1）在 Windows 中注册：打开 C:\Windows\System32\drivers\etc\hosts 文件，按照文件中的示范进行注册。
（2）如果想在 DNS 上注册，就要购买一个域名。

五、 Web 资源的访问过程



六、将 Web 应用打包
1.在命令行下进入 Web 所在目录，输入 jar -cvf name.war name，name 就是 Web 应用的名称。
2.将生成的 .war 文件拷贝到 webapps 目录下，Tomcat 会自动生成一个 Web 应用的目录。可以远程上传这个 .war 文件，由服务器生成 Web 应用目录。

加密

2018年6月2日　　11:59

配置网站的数字证书
1.在命令行输入：keytool -genkey -alias name -keyalg RSA
name 就是这个证书的名字。回车后，密钥库口令必须输入，名字与姓氏输入的是网站的主机 ip 。其他的都可以不写。最后输入 y 表示确定。最后还会提示输入密钥口令。成功后生成 C:\Users\DELL\.keystore 文件，这个密钥库存放的就是数字证书。把这个文件拷贝到 $CATALINA_BASE/conf 目录下。
2.在 $CATALINA_BASE/conf/server.xml 文件中，按照提示添加一个 <Connector> 标签:

```
<Connector port="8443" protocol="org.apache.coyote.http11.Http11AprProtocol"
maxThreads="150" SSLEnabled="true" >
```

并且添加两个属性，keystoreFile 用于指向 .keystore 文件，keystorePass 则是密钥库的口令。
3.重启服务器，配置完成。此时如果要访问加密后的网站，注意端口为 8443，而且采用 HTTPS 协议。
此时虽然配置了一个数字证书，但是这个数字证书并没有得到 CA 的认证，所以依然是不可信任的。

HTTP 基础

2018年6月2日 16:17

一、HTTP 1.0 与 HTTP 1.1 之间区别

一个最明显的区别就是，1.0 每次只允许客户端获取服务器上的一个资源，获取到之后就断开连接；1.1 允许每次获取多个资源。

举个例子：一个网页使用 <img> 引用了三幅图片，当客户端访问这个页面时，客户端总共发送了四次 HTTP 请求。第一次请求，客户端得到这个页面的 HTML 代码。在解析页面的过程中，浏览器发现页面引用了图片，于是就依次请求图片资源，所以在这里又发送了三次请求。可以看到，如果数据量很大，客户端向服务器发送的请求次数就会很多，所以需要进行优化（这主要是前端的工作），否则服务器的负荷就会很大。

二、 HTTP 请求

1.GET和POST

客户端发送的请求报文第一行为请求行，包含了方法字段。GET和POST是最重要的两种HTTP方法，都用于请求资源。这两个方法之间存在以下区别：

（1）参数

- 这两个方法都可以有额外的参数，这些参数其实是要向服务器传输的数据，比如说用户登录时输入的用户名、密码等等。GET 的参数是以查询字符串出现在 URL 中，而 POST 的参数存储在报文主体中。
- GET 的传参方式相比于 POST 安全性较差，因为 GET 传的参数在 URL 中是可见的，可能会泄露私密信息。
- GET 只支持 ASCII 字符，如果参数为中文则可能会出现乱码，而 POST 支持标准字符集。
- 向服务器传输数据，GET 的数据容量通常不能超过1k，POST 传输的数据量无限制。

（2）安全

安全的 HTTP 方法不会改变服务器状态，也就是说它只是可读的。GET 方法是安全的，而 POST 却不是，因为 POST 的目的是传送实体主体内容，这个内容可能是用户上传的表单数据，上传成功之后，服务器可能把这个数据存储到数据库中，因此状态也就发生了改变

（3）幂等性

幂等性就是指同样的请求被执行一次与连续执行多次的效果是一样的，服务器的状态也是一样的。换句话说就是，幂等方法不应该具有副作用（统计用途除外）。所有的安全方法都是幂等的，不安全的方法中除了POST以外也是幂等的。

（4）可缓存

GET可缓存，POST在多数情况下不可缓存。如果要对相应进行缓存，需要满足以下条件：

- 请求报文的 HTTP 方法本身是可缓存的，包括 GET 和 HEAD，但是 PUT 和 DELETE 不可缓存，POST 在多数情况下不可缓存的。
- 响应报文的状态码是可缓存的，包括：200, 203, 204, 206, 300, 301, 404, 405, 410, 414, and 501。
- 响应报文的 Cache-Control 首部字段没有指定不进行缓存。

2.常用请求首部字段

• Accept：客户端支持的数据类型

• Accept-Charset：客户端优先采用的编码

• Accept-Encoding：客户端优先采用的数据压缩格式

• Accept-Language：客户端优先采用的语言

• Host：客户端想访问的主机名

• If-Modified-Since：资源的缓存时间，如果网页的最后更新时间晚于这个值，就说明网页已经更新，需要发送新的页面。这个参数在高并发控制时十分有用

• Referer：客户端从这个参数指定的资源访问服务器，用于防盗链（防盗链就是指禁止某些资源访问服务器）。比如说在页面中通过点击一个超链接来访问另一个页面，那么这个参数的值就是起始页面。

• User-Agent：客户端的程序信息，比如采用了哪款浏览器、是什么操作系统等
• Connection：指示请求结束之后断开连接还是保持连接
• Date：发起请求的时间
• Cookie：客户端通过这个字段向服务器传数据
3.Range 首部字段实现断点下载
Range 字段指示服务器只传输一部分资源。这个字段的格式为：
（1）Range:bytes = 1000-2000　　传输范围是第1000个字节到第2000个字节的数据
（2）Range:bytes=1000-　　传输第1000个字节以后的数据
（3）Range:bytes=1000　　传输最后1000个字节
实例：

```
URL url = new URL("http://localhost:8080/myweb/index.html"); //资源的 URL
HttpURLConnection conn = (HttpURLConnection) url.openConnection();
conn.setRequestProperty("range", "bytes = 5-");  //指定服务器传输的数据范围
InputStream in = conn.getInputStream();
int len = 0;
byte[] buffer = new byte[1024];
FileOutputStream out = new FileOutputStream("file",true); //一定要设置第二个参数为 true
                                                          //否则新的数据就会覆盖原先已经读到的数据
while((len=in.read(buffer)) > 0) {
     out.write(buffer, 0, len);
}
in.close();
out.close();
```

三、HTTP 响应
1.状态行
（1）格式：HTTP版本号 状态码 原因描述<CRLF>
（2）状态码
服务器返回的响应报文中第一行为状态行，包含了状态码以及原因短语，用来告知客户端请求的结果。
1XX 信息
• 100 Continue ：表明到目前为止都很正常，客户端可以继续发送请求或者忽略这个响应。
2XX 成功
• 200 OK
• 204 No Content ：请求已经成功处理，但是返回的响应报文不包含实体的主体部分。一般在只需要从客户端往服务器发送信息，而不需要返回数据时使用。
• 206 Partial Content ：表示客户端进行了范围请求。响应报文包含由 Content-Range 指定范围的实体内容。
3XX 重定向
• 301 Moved Permanently ：永久性重定向
• 302 Found ：临时性重定向
• 303 See Other ：和 302 有着相同的功能，但是 303 明确要求客户端应该采用 GET 方法获取资源。
　　注：虽然 HTTP 协议规定 301、302 状态下重定向时不允许把 POST 方法改成 GET 方法，但是大多数浏览器都会在 301、302 和 303 状态下的重定向把 POST 方法改成 GET 方法。
• 304 Not Modified ：如果请求报文首部包含一些条件，例如：If-Match，If-ModifiedSince，If-None-Match，If-Range，If-Unmodified-Since，如果不满足条件，则服务器会返回 304 状态码。指示客户端获取缓存资源。

• 307 Temporary Redirect ：临时重定向，与 302 的含义类似，但是 307 要求浏览器不会把重定向请求的 POST 方法改成 GET 方法。

4XX 客户端错误

• 400 Bad Request ：请求报文中存在语法错误。

• 401 Unauthorized ：该状态码表示发送的请求需要有认证信息（BASIC 认证、DIGEST 认证）。如果之前已进行过一次请求，则表示用户认证失败。

• 403 Forbidden ：请求被拒绝，服务器端没有必要给出拒绝的详细理由。

• 404 Not Found

5XX 服务器错误

• 500 Internal Server Error ：服务器正在执行请求时发生错误。

• 503 Service Unavilable ：服务器暂时处于超负载或正在进行停机维护，现在无法处理请求。

2.常用响应首部字段

• Location：配合302状态码使用，令客户端重定向至指定 URI
• Server：服务器的安装信息
• Content-Encoding：数据的压缩格式
• Content-Length：数据的大小
• Content-Type：数据的类型
• Last-Modified：资源的最后缓存时间
• Refresh：通知浏览器隔多长时间刷新一次
• Content-Disposition：通知浏览器将数据下载到本地
• Transfer-Encoding：数据的传送格式
• Set-Cookie：由服务器端向客户端发送 cookie
• ETag：与缓存有关，控制浏览器得到的是缓存还是最新的资源，这个字段与 If-Modified-Since 的区别在于它可以实现实时更新
• Expires：通知浏览器把得到的资源缓存多长时间，-1或0就是不缓存
• Cache-Control：控制浏览器是否缓存资源，no-cache 就是不缓存
• Pragma：同上

之所以会出现这么多用来控制浏览器缓存行为的字段，是因为不同的浏览器所支持的字段可能会有所不同。

Tomcat 类加载器

2018年6月10日　　18:52

一、类别
1.服务器类加载器：负责加载 ${CATALINA_HOME}\lib 下的类
2.应用类加载器：负载加载 ${CONTEXT_HOME}\WEB-INF\lib，${CONTEXT_HOME}\WEB-INF\classes 这两个目录下的类

二、类加载顺序
${CONTEXT_HOME}\WEB-INF\classes → ${CONTEXT_HOME}\WEB-INF\lib → ${CATALINA_HOME}\lib

概述

2018年6月3日　　9:28

一、基本概念

1.servlet 是一个在 Web 服务器上运行的 Java 程序，它用来接受和响应客户端的请求，通常遵循的是 HTTP 协议。

2.Jakarta EE 提供的 servlet 接口已经有两个默认的实现类，实际开发中通常是继承这两个类而不是直接实现 Servlet 接口

- javax.servlet.GenericServlet：一般是覆写 service方法
- javax.servlet.http.HttpServlet：这个类专门处理 HTTP 请求，它已经覆写了 servce 方法，会自动判断 HTTP 请求用的是哪个方法，如果是 GET 方法，则调用 doGET 方法；如果是 POST 方法，则调用 doPOST 方法。所以如果继承这个类就要覆写 doGET 和 doPOST 方法。

3.servlet 的生命周期与三个方法有关：

（1）创建 servlet 之后用 init 方法进行初始化。

（2）客户端的请求由 service 方法进行处理。

（3）停止服务后由 destory 方法进行销毁。

这三个方法都由 servlet 的容器调用，也就是服务器。

4.servlet 的生命周期：客户端第一次访问 servlet 时服务器创建 servlet 对象并调用 init 方法进行初始化，随后 servlet 对象就保存在内存中——也就说，**绝大多数情况下 servlet 的实例只会有一个**；服务器通过调用 service 方法对客户端的请求做出响应；当关闭服务器或将 Web 应用从服务器上移除时，服务器调用 destroy 方法销毁 servlet 对象。

二、手动编写 servlet 程序

1.在 webapps 目录下新建一个 Web 应用，新建 WEB-INF/classes 目录。

2.在 classes 目录中编写一个 servlet 程序，注意写包名，并导入相关的包。与 Web 开发相关的 jar 包其实已经包含在 $CATALINA_BASE/lib 目录下。

```
package edu.whu;

import java.io.*;
import javax.servlet.*;

// 访问权限是 public
public class FirstServlet extends GenericServlet {
    @Override
    public void service(ServletRequest req,
    ServletResponse res) throws ServletException,IOException {
        OutputStream out = res.getOutputStream();
        out.write("hello,world".getBytes());
    }
}
```

3.用 javac 编译程序。注意要先设置 classpath： set classpath=%classpath%;$CATALINA_BASE/lib/servlet.jar ，把相关的 jar 包导入；然后才能编译：javac -d . FirstSerlvet.java

4.在 WEB-INF 目录下新建一个 web.xml 文件，配置 servlet 的对外访问路径：

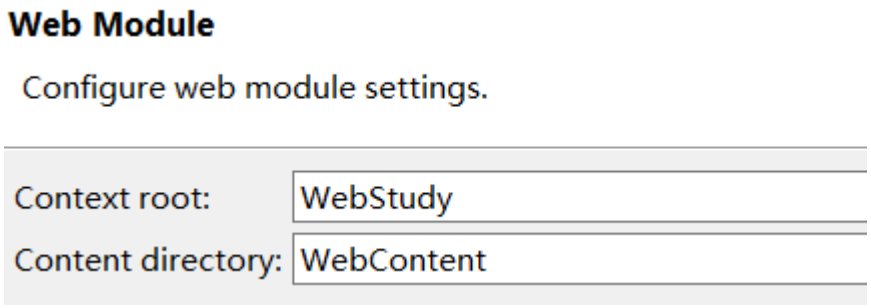
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                            http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
        version="3.1">

            <servlet>
                 <servlet-name>FirstServlet</servlet-name>
                 <servlet-class>edu.whu.FirstServlet</servlet-class>
            </servlet>
            <servlet-mapping>
                <servlet-name>FirstServlet</servlet-name>
                <!-- 注意这里是/名称 -->
                <url-pattern>/FirstServlet</url-pattern>
            </servlet-mapping>

        </web-app>
```

5.启动 Tomcat，访问 servlet。

三、用 Eclipse 开发 servlet

1.new 一个 project，选择 Dynamic Web Project。在配置中，下面的这个步骤中的 Content Root 就是 Web 应用的虚拟目录名称，Content directory 是 Web 应用所在目录。



2.创建完成后，会看到以下的目录结构：

编写所有的 Java 程序放在 Java Resources/src 下，Eclipse 会自动把这里面的程序编译然后发布到 WebContent/WEB-INF/classes 中。

3.点击 window/preferences/Server/Runtime Environments，配置服务器。

4.因为这里用的是 Tomcat，所以要把 Tomcat 提供的依赖 jar 包放入 WEB-INF/lib 中。不同的服务器的依赖 jar 包会有所不同，要注意区分。

5.程序写好之后，点击 Run As/Run On Server，一切正常的话就可以看到结果。

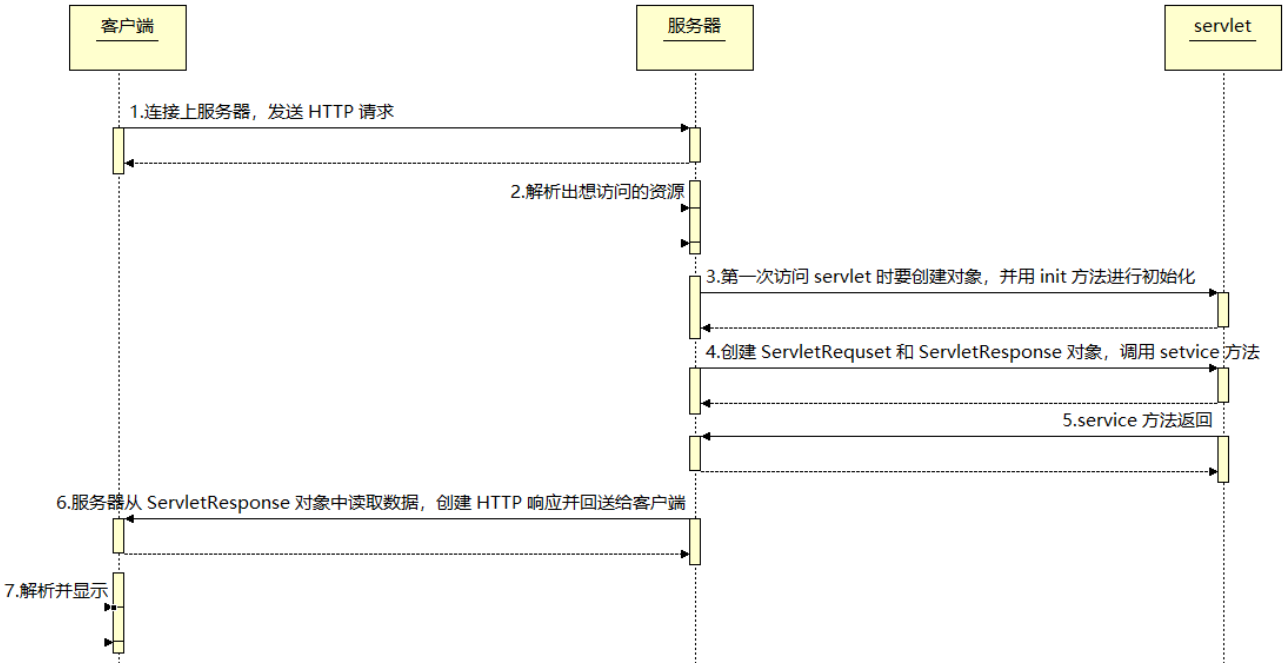
常见的两个错误：

（1）Several ports (8005, 8080, 8009) required by Tomcat Server at localhost are

already in use：已经运行了一个 Tomcat 的实例，要把这个这个实例关掉。运行 $CATALINA_BASE/bin/shutdown.bat。
（2）SEVERE: A child container failed during start：一般是因为依赖 jar 包没有导全。
实际上，Eclipse 提供了开发 Servlet 的模板，可以直接在 src 中 new 一个 Servlet，这样编写好之后 Eclipse 就会自动部署。设置的服务器如果是 Tomcat 7 以上的，那么默认是生成注解而不是 web.xml。

```
/**
 * Servlet implementation class ServletDome2
 */
@WebServlet("/ServletDemo2")
public class ServletDemo2 extends HttpServlet {
     private static final long serialVersionUID = 1L;

    /**
     * @see HttpServlet#HttpServlet()
     */
    public ServletDemo2() {
        super();
    }

     /**
      * @see HttpServlet#doGet(HttpServletRequest request,
     HttpServletResponse response)
      */
     protected void doGet(HttpServletRequest request, HttpServletResponse
     response) throws ServletException, IOException {
          response.getOutputStream().write("hello,world".getBytes());

     }

     /**
      * @see HttpServlet#doPost(HttpServletRequest request,
     HttpServletResponse response)
      */
     protected void doPost(HttpServletRequest request, HttpServletResponse
     response) throws ServletException, IOException {
          doGet(request, response);
     }

}
```

四、servlet 的调用过程


客户端
服务器
servlet
1.连接上服务器，发送 HTTP 请求
2.解析出想访问的资源
3.第一次访问 servlet 时要创建对象，并用 init 方法进行初始化
4.创建 ServletRequset 和 ServletResponse 对象，调用 setvice 方法
5.service 方法返回
6.服务器从 ServletResponse 对象中读取数据，创建 HTTP 响应并回送给客户端
7.解析并显示

详解 servlet

2018年6月3日　　14:37

一、servlet 引擎
servlet 不能独立运行，必须由 servlet 引擎来控制和调度。一般来说，servlet 引擎指的就是服务器。

二、将 servlet 程序映射为 URL
1.如果是通过 web.xml 进行配置，那么要使用以下标签：

```
<servlet-mapping>
  <servlet-name>ServletDemo</servlet-name>
  <url-pattern>/ServletDemo</url-pattern>
</servlet-mapping>
```

一个 <servlet> 标签下可以有多个 <url-pattern> 子标签，指定的就是 URL 的模式。URL 的模式可以是 /名称；也可使用通配符，格式有两种，一种是 *.扩展名，另一种是 /*。
2.如果是通过注解进行配置，那么可以有两种形式。如果只设置一个 URL，就直接设置注解的属性：

```
@WebServlet("/ServletDemo3")
```

如果要设置多个 URL，就用下面这种写法：

```
@WebServlet(urlPatterns = {"/ServletDemo2","/aa","/*"})
```

3.多个 servlet 匹配同一个 URL 请求的情况

对于如下的一些映射关系：

- Servlet1 映射到 /abc/*
- Servlet2 映射到 /*
- Servlet3 映射到 /abc
- Servlet4 映射到 *.do

问题：

- 当请求URL为“/abc/a.html”，“/abc/*”和“/*”都匹配，哪个servlet响应
  Servlet引擎将调用Servlet1。
- 当请求URL为“/abc”时，“/abc/*”和“/abc”都匹配，哪个servlet响应
  Servlet引擎将调用Servlet3。
- 当请求URL为“/abc/a.do”时，“/abc/*”和“*.do”都匹配，哪个servlet响应
  Servlet引擎将调用Servlet1。
- 当请求URL为“/a.do”时，“/*”和“*.do”都匹配，哪个servlet响应
  Servlet引擎将调用Servlet2。
- 当请求URL为“/xxx/yyy/a.do”时，“/*”和“*.do”都匹配，哪个servlet响应
  Servlet引擎将调用Servlet2。

两个原则：（1）形式越接近就越先匹配（2）“*.扩展名”模式的优先级最低 。
4.服务器会自动生成一个 URL 为 / 的缺省 servlet，这个缺省 servlet 可以用来访问静态资源，比如一个 HTML 页面。可以在 conf/web.xml 中看到一个 <servlet-name> defualt</servlet-name>，这个设置的就是缺省 servlet。在实际开发中不要把自己写的 servlet 映射为 /，否则会出现诸如静态资源无法访问等问题。
5.注意：如果某个 servlet 的 url-pattern 是/*，那么访问 localhost:8080/ 会匹配到该 servlet 上，而不是匹配 welcome-file-list；如果 url-pattern 是 /（该 servlet 即为默认 servlet），当其他匹配模式都没有匹配到，则会匹配 welcome-file-list。

三、servlet 的初始化问题
一般而言，servlet 对象是在客户端第一次请求资源的时候创建并初始化。但是，如果在<servlet> 标签下添加一个子标签

```
<load-on-startup>1</load-on-startup>
```

那么这个 servlet 对象就会在 Web 应用程序启动的时候就创建并初始化。标签中的整数代表创建的优先级，数字越小优先级越高。如果有些 servlet 的工作是初始化某些资源，那么就可以设置这个标签。

四、线程安全
servlet 本身是无状态的，一个无状态的 servlet 是绝对线程安全的，无状态对象设计也是解决线程安全问题的一种有效手段。所以，servlet 是否线程安全是由它的实现来决定的，如果它内部的属性或方法会被多个线程改变，它就是线程不安全的，反之，就是线程安全的。比如说，一个 servlet 没有任何的数据会被多个线程共享，那么它就绝对是线程安全的；但是如果这个 servlet 有一个静态域，那它就不是线程安全的。下面这个 servlet 就不是线程安全的：

```
@WebServlet("/ServletDemo3")
public class ServletDemo3 extends HttpServlet {
    private int count = 0;   //每一个线程都会访问到同一个 count

    protected void doGet(HttpServletRequest request, HttpServletResponse
    response)
            throws ServletException, IOException {
        count++;
        // 注意在 servlet 中不能抛异常，一是因为子类不能抛出比父类更多的异
        常，而是抛异常就会影响用户
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(count);
    }


    protected void doPost(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
        doGet(request, response);
    }

}
```

可以有两种方法来实现线程安全：
1.同步对共享数据的操作
使用 synchronized 关键字能保证一次只有一个线程可以访问被保护的区段，可以通过同步块操作来保证 servlet 的线程安全。如果在程序中使用同步来保护要使用的共享的数据，也会使系统的性能大大下降。这是因为同一时间内只能有一个线程访问 servlet，这就会造成阻塞。对于网页而言，阻塞是十分影响用户体验的。

```
@WebServlet("/ServletDemo3")
public class ServletDemo3 extends HttpServlet {
```

```
    private int count = 0; // 每一个线程都会访问到同一个 count

    protected void doGet(HttpServletRequest request, HttpServletResponse
    response)
            throws ServletException, IOException {

        synchronized (this) {
            count++;
            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(count);
        }

    }

    protected void doPost(HttpServletRequest request, HttpServletResponse
    response)
            throws ServletException, IOException {
        doGet(request, response);
    }

}
```

2.避免使用实例变量
线程安全问题很大部分是由实例变量造成的，只要在 servlet 里面的任何方法里面都不使用实例变量，那么该 servlet 就是线程安全的。

```
@WebServlet("/ServletDemo3")
public class ServletDemo3 extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse
    response)
            throws ServletException, IOException {
        int count = 0; // 改为临时变量
        count++;
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(count);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse
    response)
            throws ServletException, IOException {
        doGet(request, response);
    }

}
```

ServletConfig

2018年6月3日　　17:51

ServletConifig 会在 servlet 对象初始化时由服务器传递给 servlet，servlet 可以从中获取自身的一些初始化数据，说白了就是 servlet 的配置信息。比如说有些数据不适合在程序中给定，那么就可通过 ServletConfig 对象传递给 servlet。

一、配置初始化参数
1.在 web.xml 文件中配置
在 <servlet> 标签下添加子标签 <init-param>：

```
<init-param>
    <param-name>data</param-name>
    <param-value>123132</param-value>
</init-param>
```

param-name 是参数名称，param-value 是参数值。一个 servlet 可以设置多个初始化参数。
2.通过注解进行配置
通过注解 @WebInitParam 配置初始化参数，这个注解一般不会单独使用，而是包含在 @WebServlet 或者 @WebFilter 中：

```
@WebServlet(urlPatterns="/ServletDemo4",initParams=
{@WebInitParam(name="data1",value="111"),
        @WebInitParam(name="data2",value="222")})
```

二、获取初始化参数

```
@WebServlet(urlPatterns="/ServletDemo4",initParams=
{@WebInitParam(name="data1",value="111"),
        @WebInitParam(name="data2",value="222")})
public class ServletDemo4 extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
        // 获得指定的初始化参数
        String value1 = this.getServletConfig().getInitParameter("data1");
        response.getOutputStream().write(value1.getBytes());

        // 获得所有的初始化参数
        Enumeration<String> e =
        this.getServletConfig().getInitParameterNames();
        while(e.hasMoreElements()) {
            String name = e.nextElement();
            String value2 =
            this.getServletConfig().getInitParameter(name);
            String str = name + "=" + value2;
            response.getOutputStream().write(str.getBytes());
        }
    }
```

```
        protected void doPost(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {
             doGet(request, response);
        }

    }
```

三、适合使用初始化参数的情形
有几个比较典型的情形：
1.字符的编码方式
2.数据库的连接信息
3.配置文件的路径

ServletContext

2018年6月3日　　19:23

这个对象代表的是当前的 Web 应用。一个 Web 应用就是一些 servlet 和资源的集合，因此所有 servlet 共享同一个 ServletContext 对象。通过这个对象可以使 servlet 与服务器进行通信，也可以进行 servlet 之间的数据传递。
ServletContext 对象通常也叫做 context 域。域表征的是一个范围，ServletContext 对象的作用范围是整个 Web 应用，只要应用程序没有退出，这个对象就会一直存在。

一、获取 ServletContext 对象
ServletContext 对象包含在 ServletConfig 对象中，因此可以通过 ServletConfig 的方法获取，也可以直接 this.getServletContext()。

```
// 获取 ServletContext 对象的方法一
    ServletContext context = this.getServletConfig().getServletContext();

    // 获取 ServletContext 对象的方法二
    context = this.getServletContext();
```

二、应用场景
1.多个 servlet 的数据共享
比如实现一个聊天室，一个用户的浏览器发送信息，另一个用户的浏览器获取这个信息。

```
// 发送信息
@WebServlet("/ServletDemo5")
public class ServletDemo5 extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
        this.getServletContext().setAttribute("message", "hello");
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
        doGet(request, response);
    }

}

//获取数据
@WebServlet("/ServletDemo6")
public class ServletDemo6 extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
        String msg = (String)
        this.getServletContext().getAttribute("message");
        response.getOutputStream().write(msg.getBytes());
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse
```

```
        response) throws ServletException, IOException {
            doGet(request, response);
        }
    }
```

实际上，在进行数据的转发时，不能通过 ServletContext，而要用 ServletRequest 对象。
2.获取整个 Web 应用的初始化参数
可以在 web.xml 文件中获取添加 <context-param> 标签，设置 Web 应用的全局初始化参数。比如说所有的 servlet 都要连接同一个数据库，那么就可把数据库连接信息设置成全局初始化参数。
设置参数：

```
<context-param>
  <param-name>data</param-name>
  <param-value>xxx</param-value>
</context-param>
```

获取参数：

```
this.getServletContext().getInitParameter("data");
```

如果有多个参数，可以先 this.getServletContext().getInitParameterNames() 获取所有参数的名称，再迭代获取每个参数的值。
3.实现 servlet 的转发
类似于 HTTP 中的重定向，客户端请求某个 servlet，服务器进行转发，让客户端得到另一个 servlet 的资源。与重定向不同的是，转发这一行为是服务器完成的，而不是服务器通知客户端请求另一个 servlet。一般而言 servlet 都不用于输出数据，因为这不够美观，应该把需要输出的数据转交给别的程序（比如 JSP）进行美化然后再输出给浏览器。

```
this.getServletContext().getRequestDispatcher("/WebStudy/ServletDemo1").forwa
rd(request,response);
```

4.获取配置文件
配置文件通常有两种类型，propertie 文件和 XML 文件。如果配置文件中的数据是没有关联的，那么就用 properties 文件；否则，用 XML 文件。

```
InutStream in = this.getServletContext().getResourceAsStream("");
Properties prop = new Properties();
prop.load(in);
String prop1 = prop.getProperty("prop1");
```

需要注意的问题：
（1）配置文件的路径。如果配置文件放在 src 目录下，那么实际上程序编译完成、发布到服务器上后这个文件是在 WEB-INF/classes 目录下，此时路径为 "/WEB-INF/classes/prop.properties" ；如果直接放在 Web 应用根目录下，路径为 "/prop.properties"。而且注意最好不要通过传统的读文件的方式读取配置文件，因为如果通过 FileInputStream 读取，此时文件的相对路径是相对于服务器的启动路径。如果非要通过这种方式读取，就要先获得文件的绝对路径：

```
String path = this.getServletContext().getRealPath("/WEB-
INF/classes/prop.properties");
InputStream in = new FileInputStream(path);
```

当然，通过传统方式读也有好处，就是可以得到文件名称。

```
// 找到文件路径中最后一个 / 的位置索引，然后截取其后的子字符串
String fileName = path.subString(path.lastIndexOf("//") + 1);
```

（2）通过 Java SE 程序读取配置文件。比如说配置文件中包含的是数据库连接信息，那么在实际开发中，数据库的操纵是数据访问层的工作，而 Web 应用处于表示层，如果还依赖于 ServletContext 读取文件，那就会导致层与层之间的耦合。

```
@WebServlet("/ServletDemo7")
public class ServletDemo7 extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
        UserDao dao = new UserDao();
        dao.update();
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
        doGet(request, response);
    }

}

// 数据访问层通过类加载器读取
public class UserDao {

    static {
        // 因为配置文件已经放在 src 目录下，而这里是通过类加载器读取的，所
        以可以直接写文件名
        InputStream in =
        UserDao.class.getClassLoader().getResourceAsStream("db.properties")
        ;
        Properties prop = new Properties();
        try {
            prop.load(in);
            // 获取信息
        } catch (IOException e) { // 一定要抛异常
            throw new ExceptionInInitializerError(e);
        }
    }

    public void update() throws IOException {

    }
}
```

注意，通过类加载器读取的文件不能太大，因为这些文件是一开始就载入内存中的。而且通过类加载器读取的文件只在类加载的时候读取一次，所以不管读多少次，读到的数据都是第一次读取到的数据。如果想用类加载器读取到更新后的数据，就先获取文件的路径，然后再用传统方式读文件：

```
String path =
UserDao.class.getClassLoader().getResource("db.properties").getPath();
InputStream in = new FileInputStream(path);
//...
```

HttpServletResponse

2018年6月4日　　9:06

这个对象封装了服务器的 HTTP 响应，包括向客户端发送数据、发送响应头、发送状态码等方法。

一、向客户端输出数据
实际上 servlet 是先把数据封装到 response 中，然后服务器再用对象创建成 HTTP 响应并发送给客户端。
输出数据的时候，要么用 OutputStream，要么用 PrintWriter，不能同时用这两个。尤其是在用到 servlet 转发的时，多个 servlet 的输出方法要一样。但是：重定向不会有这个问题，因为是多个请求。
通过 getOutputStream 和 getWriter 方法得到的流对象会由服务器管理，也就是说无需手动关闭。
1.字符编码
传输中文数据时，必须指定客户端的字符集，否则编码和解码的字符集不一样就会出现乱码

```
@WebServlet("/ServletDemo8")
public class ServletDemo8 extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse
    response)
            throws ServletException, IOException {

        String data = "啊哈哈哈";
        response.setHeader("Content-type", "text/html;charset=UTF-8"); //
        设置 Content-type 首部字段，注意 text/html 和 charset 之间是以分号
        隔开的！
        response.getOutputStream().write(data.getBytes("UTF-8"));
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
        doGet(request, response);
    }

}
```

指定浏览器字符集的另一种方式是使用 HTML 中的 <meta> 标签：

```
String htmlHead = "<head><meta http-equiv='Content-type'
content='text/html;charset=UTF-8'></head>";
response.getOutputStream().write(htmlHead.getBytes());
response.getOutputStream().write(data.getBytes("UTF-8"));
```

2.通过 PrintWriter 输出数据
如果要输出的数据是字符数据，也可以用 PrintWriter 输出数据。但是也要注意字符编码的问题，既要指定数据的字符集，也要指定浏览器的字符集。

```
String data = "啊哈哈哈";
response.setCharacterEncoding("UTF-8");                // 指定数据的字符集
```

```
response.setContentType("text/html;charset=UTF-8");  // 指定浏览器的字符集
response.getWriter().write(data);
```

方法 setContentType 就相当于第一个参数为"Content-type" 的 setHeader 方法。实际上调用 setContentType 方法时就同时指定了数据和浏览器的字符集，但是有时候为了代码的可读性，还是会同时调用 setCharacterEncoding 和 setContentType。

二、实现文件下载

```
public class ServletDemo9 extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
        // 资源放在 Web 应用目录下，所以用 ServletContext 读取
        String path = this.getServletContext().getRealPath(File.separator +
        "download" + File.separator + "1.png");
        String fileName = path.substring(path.lastIndexOf(File.separator) +
        1);
        InputStream in = new FileInputStream(path);
        int len = 0;
        byte[] b = new byte[1024];

        OutputStream out = response.getOutputStream();
        // 通知浏览器下载该文件
        response.setHeader("content-disposition", "attachment;filename=" +
        fileName);
        while((len = in.read(b)) > 0) {
            out.write(b,0,len);
        }
        out.close();
        in.close();
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
        doGet(request, response);
    }

}
```

如果的文件名是中文，那么就要注意编码问题。

```
response.setHeader("content-disposition", "attachment;filename=" +
URLEncoder.encode(fileName, "UTF-8"));
```

三、生成随机验证码

```
@WebServlet("/ServletDemo10")
public class ServletDemo10 extends HttpServlet {

    private static final int WIDTH = 120;
    private static final int HEIGHT = 30;
```

```
    protected void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
        BufferedImage img = new
        BufferedImage(WIDTH,HEIGHT,BufferedImage.TYPE_INT_RGB);
        Graphics g = img.getGraphics();

        // 设置背景色
        setBackgroundColor(g);

        // 设置边框
        setBorder(g);

        // 画干扰线
        drawRandomLine(g);

        // 画随机数
        drawRandomNumber((Graphics2D)g);

        // 将图形发送给浏览器，并通知浏览器不缓存
        response.setContentType("image/jpeg");
        response.setHeader("Cache-Control", "no-cache");
        response.setHeader("Pragma","no-cache");
        ImageIO.write(img, "jpg", response.getOutputStream());
    }


    private void drawRandomNumber(Graphics2D g) {
        g.setColor(Color.RED);
        g.setFont(new Font("宋体",Font.ITALIC | Font.BOLD,20));

        String source =
        "0123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
        int codeLenght = source.length();
      Random rand = new Random(System.currentTimeMillis());
      int x = 5;
      int y = 20;
      for(int i = 0; i < 4; i++){
           // 从字符源中随机抽取一个字符
          String code = source.charAt(rand.nextInt(codeLenght-1)) + "";
          int rotateDegree  = new Random().nextInt() % 30; // 图片旋转的角
度 -30~30
          double theta = rotateDegree * Math.PI / 180;  // 转成弧度
          g.rotate(theta,x,y);
          g.drawString(code, x, y);
          g.rotate(-theta, x, y);;  // 不影响下一个字的旋转
          x += 30;
      }

    }


    private void drawRandomLine(Graphics g) {
        g.setColor(Color.GREEN);
```

```
            for(int i = 0; i < 5; i++) { // 产生随机线的起止点
                int x1 = new Random().nextInt(WIDTH);
                int y1 = new Random().nextInt(HEIGHT);
                int x2 = new Random().nextInt(WIDTH);
                int y2 = new Random().nextInt(HEIGHT);
                g.drawLine(x1, y1, x2, y2);
            }
        }


        private void setBorder(Graphics g) {
            g.setColor(Color.BLUE);
            g.drawRect(1, 1, WIDTH-2, HEIGHT-2);
        }


        private void setBackgroundColor(Graphics g) {
            g.setColor(Color.WHITE);
            g.fillRect(0, 0, WIDTH, HEIGHT);
        }

        protected void doPost(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {
            doGet(request, response);
        }


    }
```

很多情况会在网站首页用户登录时显示验证码，所以要在 web.xml 文件中配置首页：

```
    <welcome-file-list>
        <welcome-file>register.html</welcome-file>
    </welcome-file-list>
```

很多网站的验证码都有点击就换一张的功能，这个功能用 JavaScript 来实现：

```
    <!DOCTYPE html>
    <html>
    <head>
    <meta charset="UTF-8">
    <title>注册</title>
    <script>
        function refresh(img) {
            // 为地址添加一个随机数，说明是一个新地址；
            //否则浏览器认为这是同一个地址，就从缓存中得到同一张图片
            img.src = img.src + "?" + new Date().getTime();
        }
    </script>
    </head>
    <body>
        <form action="">
            用户名：<input type="text" name="username"></input><br /> 密
```

```
            码：<input
                type="text" name="password"></input><br /> 验证码：<input
                type="text"
                name="verifycode"><img src="/WebStudy/ServletDemo10"
                onclick="refresh(this)"></input><br /> <input type="submit"
                value="注册"></input>
        </form>
    </body>
    </html>
```

四、控制浏览器定时刷新

```
    @WebServlet("/ServletDemo11")
    public class ServletDemo11 extends HttpServlet {

        protected void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {
            // 用户登录成功
            // 提示用户将在3s后跳转到首页，如果没有跳转则点击...
            response.setHeader("refresh", "3;url='/WebStudy/index.html'"); //
            重定向
            response.setContentType("text/html;charset=UTF-8");
            response.getWriter().write("本页面将在3秒后跳转，如果没有跳转请点击
            <a href='http://localhost:8080/WebStudy/index.html'>首页</a>");
        }

        protected void doPost(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {
            doGet(request, response);
        }

    }
```

注意在实际开发中其实并不会这么做，因为数据的输出要由前端完成，这里只是实现了这种刷新页面并跳转的逻辑。

五、控制浏览器缓存
如果数据是不变的，可以控制浏览器进行缓存。

```
    response.setDateHeader("Expires", System.currentTimeMillis() + 1000*3600);
    // 缓存一小时
    // 时间值必须设置为当前时间加上需要缓存的时长
    String data = "aaaa";
    response.getWriter().write(data);
```

六、请求重定向

```
    response.setStatus(302);
    response.setHeader("location", "/WebStudy/index.html");

    // 如果对 HTTP 协议不熟悉，可以向下面这么做
    //response.sendRedirect("/WebStudy/index.html");
```

请求重定向会让服务器响应两次请求，从而加重服务器的负担，所以一般情况下不用请求重定向而用 servlet 转发。但是请求重定向会让浏览器的地址栏发生变化，使用户直到发生了重定向。实际开发中有些场景必须使用请求重定向：登录成功后跳转到某个页面；购物网站上购买某样商品后跳转到购物车页面。

HttpServletRequest

2018年6月4日　　19:49

HTTP 请求首部中的所有信息都封装在这个对象中，可以通过这个方法得到客户端的信息。

一、获取请求资源的 URI/URL
通过 getRequestURI() 和 getRequestURL() 方法可以获得客户端请求的资源的 URI/URL，这可以用在统计资源访问次数、拦截某些请求等方面。

二、获取客户端的 ip 地址、完整主机名、端口

- getRemoteAddr() 得到客户端的 ip 地址
- getRemoteHost() 得到客户端的主机名
- getRemoteHost() 得到客户端的端口，这个端口是指应用程序的端口，比如浏览器的端口

三、获取请求方法
getMethod() 得到 HTTP 请求的方法，一般都是 GET。

四、获取请求首部字段

```
// 获取某个指定的首部字段
String acceptEncoding = request.getHeader("Accept-Encoding");
System.out.println(acceptEncoding);
// 可能存在多个同名字段
Enumeration<String> values = request.getHeaders("Accept-Encoding");
String value;
while(values.hasMoreElements()) {
     value = values.nextElement();
     System.out.println(value);
}

// 获取所有的首部字段
Enumeration<String> names = request.getHeaderNames();
String name;
while(names.hasMoreElements()) {
     name = names.nextElement();
     value = request.getHeader(name);
     System.out.print(name + ": ");
     System.out.println(value);
}
```

五、获取请求中的参数
1.在实际开发中，获取到的数据要先检验才能使用。

```
@WebServlet("/RequestDemo2")
public class RequestDemo2 extends HttpServlet {

     protected void doGet(HttpServletRequest request, HttpServletResponse
     response) throws ServletException, IOException {
          // 获取指定参数
          String value = request.getParameter("username");
```

```
            if(value != null && !value.trim().equals("")) {
                System.out.println(value);
            }


            // 获得同名参数
            String[] values = request.getParameterValues("username");
            for(int i = 0;values != null && i < values.length; i++) {
                System.out.println(values[i]);
            }

            // 获取所有参数
            Enumeration<String> names = request.getParameterNames();
            String name;
            while(names.hasMoreElements()) {
                name = names.nextElement();
                value = request.getParameter(name);
                if(value != null && !value.trim().equals("")) {
                    System.out.print(name + "=");
                    System.out.println(value);
                }
            }

            // 获取名称/值的键值对，便于将数据封装到对象中
            Map<String, String[]> map -= request.getParameterMap();
            User user = new User();
            if (!map.isEmpty()) {
                try {
                    BeanUtils.populate(user, map);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }

            // 很少用的方式，但是如果上传的是文件就要用这个方法
            InputStream in = request.getInputStream();
            int len = 0;
            byte[] b = new byte[1024];
            while((len=in.read(b)) > 0) {
                System.out.println(new String(b,0,len));
            }
        }

        protected void doPost(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {
            doGet(request, response);
        }

    }
```

2.如果参数值包含中文，就要注意处理乱码问题。出现乱码是因为浏览器采用 UTF-8 编码，但是 Java 内部采用 ISO8859-1，所以请求报文中的字符集使用的是 ISO8859-1，程序解码的时候用的也是 ISO8859-1，这就导致了编码和解码的字符集不一致。对于用表单提交的数据，

GET 和 POST 的对应处理方法有所不同。

```
@WebServlet("/RequestDemo4")
public class RequestDemo4 extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
        String username = request.getParameter("userName")
        // 这时程序已经按照 ISO8859-1进行解码，所以要先把数据从字符重新变回
        编码，然后再按照 UTF-8 解码
        username = new String(username.getBytes("iso8859-1"),"UTF-8");
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
        request.setCharacterEncoding("UTF-8");  // 让程序按照 UTF-8 进行解
        码
        String username = request.getParameter("username");
    }

}
```

如果是超链接 URL 中的参数，就用 GET 对应的方法解决，因为超链接的请求方法是 GET。
注意，在上面的 doPOST 方法中，username 就已经是用户提交的中文字符串了，而不再是编码。
3.通过修改服务器配置解决乱码问题（**在实际开发中不要用！**）
在 conf/server.xml 文件中找到标签 <Connector> ，添加属性 URIEncoding="UTF-8"。或者添加属性 useBodyEncodingForURI="true"，但是要用
request.setCharacterEncoding("UTF-8"); 设置网页主体部分的编码。

六、实现请求转发
在 MVC 设计模式中，servlet 作为 controller 接受到客户端的请求后，将需要发送给客户端的数据转发给 viewer，让 viewer 进行输出。在这个过程中，servlet 实际上是把同一个请求连同数据转发给了 viewer，这可以保证多个请求不会互相干扰。

```
String data = "something needs forwarded";
// request 实现转发
request.setAttribute("data", data);   // 把数据封装到 request 中
request.getRequestDispatcher("/message.jsp").forward(request, response); //
把同一个请求转发给 viewer
```

需要注意的细节：
1.如果在转发之前已经把数据发送给浏览器，那么转发的时候就会抛异常。比如说第一个 servlet 在进行转发之后又调用 response.getWriter().write()，然后第二个 servlet 又视图进行转发，那么在第二个 serve了他 进行转发时就会抛出异常。所以，为了避免多次转发，在第一次转发之后就 return。
2.如果在调用 forward 方法之前已经向 response 中写入了内容，比如说调用 response.getWriter().write()，只要流对象没有关闭，转发的时候是不会抛异常的，而且转发时会把原来存在缓冲区中的数据清空。但是已经存在的响应首部字段不会被清空。
3.还需要注意转发的时候浏览器地址栏不会发生，请求重定向则会变化。

Web 开发中各类地址的写法

2018年6月5日　　10:31

有两个基本原则：
（1）地址以路径分隔符开头。
（2）如果这个地址是写给服务器的，那么分隔符代表当前 Web 应用；如果这个地址是写给浏览器的，那么分隔符代表当前网站。一个网站可能有多个 Web 应用。

举例：

```
// 请求转发
request.getRequestDispatcher(File.separator +
"index.html").forward(request,response);

// 请求重定向
response.sendRedirect(File.separator + "WebStudy" + File.separator +
"index.html");

// 获取文件绝对路径
this.getServletContext().getRealPath(File.separator + "index.html");

// 获取资源
this.getServletContext().getResourceAsStream(File.separator + "index.html");

// HTML 中的路径
/*
    <a href="/WebStudy/index.html">跳转到某个页面</a>
    <form action="/WebStudy/index.html"></form>
 */
```

关于 / 和 \ ：URL 中出现的都是 / ，硬盘上的文件路径用 \。如果区分不了，能用 File.separator 就用 File.separator。

防盗链

2018年6月5日　　10:53

盗链是指服务提供商自己不提供服务的内容，通过技术手段绕过其它有利益的最终用户界面（如广告），直接在自己的网站上向最终用户提供其它服务提供商的服务内容，骗取最终用户的浏览和点击率。

防盗链的简单实现：核心是使用 HTTP 请求首部字段 referer

```
String referer = request.getHeader("referer");
if(referer == null || !referer.startsWith("http://localhost")) {
     response.sendRedirect("/WebStudy/index.html");  // 如果是盗链，就转到网站首页
     return;
}
// 输出正确数据
```

文件上传

2018年6月9日　　10:15

一、实际应用中文件上传的限制
1.对表单的限制
（1）使用 POST 方法
（2）指定属性 enctype="multipart/form-data"
（3）表单中要添加文件表单项 <input type="file" name="file">
2.对 servlet 的限制
不能用 request.getParameter 方法获取从表单上传的文件。只要指定了 enctype="multipart/form-data" 就不能用这个方法。要用 request.getInputStream 获取请求报文的主体。
3.enctype="multipart/form=data" 说明这个表单是多部件表单，即一个表单项是就是一个部件，每个部件都对应一个独立的请求报文。
（1）普通表单项
- 1个首部字段：Content-Disposition: name="xxx" 即表单项的名称
- 报文主体就是表单项的值

（2）文件表单项
- 2个首部字段：Content-Disposition: name="xxx";filename="xxx" 表单项名称和上传的文件的名称。Content-Type: 上传的文件的 MIME 类型。
- 报文主体就是文件的内容。

二、使用 commons-fileupload
这个第三方库可以解析 HTTP 请求中携带的数据，解析后一个表单项数据会被封装到一个 FileItem 对象中。使用时主要有三个类：
- DiskItemFileFactory：工厂类
- ServletFileUpload：解析器
- FileItem：表单项类

注意：这个包依赖于 commons-io
上传文件的样本代码：

```
request.setCharacterEncoding("utf-8");
response.setContentType("text/html");


DiskFileItemFactory factory = new DiskFileItemFactory(); // 首先创建工厂
ServletFileUpload load = new ServletFileUpload(factory);
try {
    List<FileItem> items = load.parseRequest(request);
} catch (FileUploadException e) {
    if (e instanceof FileSizeLimitExceededException) { // 单个文件超限
        request.setAttribute("fileSizeLimitExceeded", "上传文件的大小不能超过1kb");
        System.out.println("上传文件的大小不能超过1kb");
    }
} catch (Exception e) {
    throw new RuntimeException(e);
}
```

三、上传文件时要注意的细节
1.上传的文件保存到 WEB-INF 目录下，目的是不让浏览器直接访问到文件。
2.注意文件名
（1）有些浏览器上传的文件名是绝对路径，那么需要把文件名称提取出来。

```
String fileaName = item.getName();
int index = fileName.lastIndexOf("\\");
if(index != -1) { // 检查文件名是否是绝对路径
    fileName = fileName.substring(index + 1);
}
```

（2）注意设置文件名和普通表单项的编码，可以通过 request.setCharacterEncoding 设置，也可以通过 ServletFileUpload 的 setHeaderEncoding 方法来设置。后一种方法的优先级更高。
（3）实际应用中可能出现同名文件的问题，所以需要在给文件名加上前缀。

```
UUID id = UUID.randomUUID();
fileName = id + "_" + fileName; // 前缀和文件名之间用下划线隔开，需要时可以方便地截取出文件名
```

3.不能在一个目录下存放过多的文件，就是要把目录打散。目录打散的方法有很多中，这里提供一种以哈希值为基础的打散方式。算法为：

- 计算文件名的 hashCode；
- 把 hashCode 转为十六进制，取前两位。比如1B2C3D4，那么前两位就是1B，于是把该文件放在/1/B 目录下。

可以根据需要取 hashCode 的更多位来生成目录，可想而知取的位数越多目录也就越多。这种方法的缺陷在于必须通过文件名的 hashCode 才知道文件在哪个目录下。

```
String root = this.getServletContext().getRealPath("/WEB-INF/files/");
// 文件保存的根目录
String fileName = item2.getName(); // 原始文件名
int index = fileName.lastIndexOf("\\");
if (fileName.lastIndexOf("\\") != -1) { // 检查文件名是否是绝对路径
    fileName = fileName.substring(index + 1);
}

String saveName = UUID.randomUUID() + "_" + fileName; // 给文件名加上前缀，防止同名冲突

int code = fileName.hashCode();
String hex = Integer.toHexString(code); // 将 hashCode 转为十六进制

File dirFile = new File(root,hex.charAt(0) + File.separator +
hex.charAt(1)); // 生成路径
File saveFile = new File(dirFile,saveName);
if(!saveFile.getParentFile().exists()) { // 如果路径原先不存在，则创建
    saveFile.getParentFile().mkdirs();
}
item2.write(saveFile); // 保存
```

4.实际应用中可能需要限制上传文件的大小，既可以是限制单个文件，也可以是限制整个请求的数据大小。

- 限制单个文件大小使用 ServletFileUpload 的 setFileSizeMax，参数的单位是字节。这个方法必须在 parseRequest 方法之前调用。如果上传的文件大小超过限制，就抛异常。可以获取异常信息并使用。
- 限制整个请求的数据大小使用 ServletFileUpload 的 setSizeMax，参数单位是字节。这个方法必须在 parseRequest 方法之前调用。如果超出限制就抛出异常。

```
DiskFileItemFactory factory = new DiskFileItemFactory(); // 首先创建工厂
ServletFileUpload load = new ServletFileUpload(factory);
load.setFileSizeMax(1024);
load.setSizeMax(1024*1024);
try {
    List<FileItem> items = load.parseRequest(request);
} catch (FileUploadException e) {
    if(e instanceof FileSizeLimitExceededException) { // 单个文件超限
        request.setAttribute("fileSizeLimitExceeded", "上传文件的大小不
        能超过1kb");
    }
} catch(Exception e) {
    throw new RuntimeException(e);
}
```

5. 服务器接受上传的文件时会先保存在内存中，如果上传的文件太大，要保存在硬盘的一个临时目录内。当上传完毕时就将文件从临时目录中剪切到指定的保存目录下。

- 缓存大小：文件超出这个值才把文件保存在硬盘中。默认是10kb。
- 临时目录：指定硬盘中的一个目录用来存放缓存文件。

```
DiskFileItemFactory factory = new DiskFileItemFactory(20*1024,new
File("F:" + File.separator + "temp")); // 指定缓存大小和目录
```

文件下载

2018年6月9日　　15:09

一、概述

文件下载的本质就是向客户端发送字节数据。

二、基本步骤

需要在设置两个响应报文首部字段和，并且需要一个流对象。两头一流

1.Content-Type：文件的 MIME 类型，比如 image/pjpeg

2.Content-Disposition：默认值是 inline，表示在浏览器内直接打开文件。如果想打开文件下载窗口，就赋值 attachment;filename=xxx。

3.流对象就是需要下载的文件数据。

```
String fileName = "G:" + File.separator + "test" + File.separator +
"zhaopian.png";
String mime = this.getServletContext().getMimeType(fileName); // 通过文件名获取
MIME 类型
// 设置响应首部字段
response.setHeader("Content-Type", mime);
response.setHeader("Content-Disposition",
"attachment;filename=zhaopian.png");
FileInputStream in = new FileInputStream(fileName); // 读取文件
ServletOutputStream out = response.getOutputStream();
IOUtils.copy(in,out);  // 把输入流的数据输入到输出流中
in.close();
```

上面的代码中使用了 commomns-io 包。

三、解决中文文件名的乱码

不同浏览器的编码方式会有所不同，所以需要设计一种通用的解决方案。

```
//中文文件名乱码的通用解决方案
private String fileNameEncoding(String fileName, HttpServletRequest request)
        throws UnsupportedEncodingException {
    String agent = request.getHeader("User-Agent");
    if (agent.contains("FireFox")) { // FireFox 族浏览器使用 BASE64 编码
        Base64.Encoder encoder = Base64.getEncoder();
        fileName = "?utf-8?B?" +
        encoder.encodeToString(fileName.getBytes("utf-8")) + "?=";
    } else if (agent.contains("MSIE")) { // IE 族浏览器使用  UTF-8 编码
        fileName = URLEncoder.encode(fileName, "utf-8");
    } else { // 其余浏览器也基本上用的是 UTF-8 编码
        fileName = URLEncoder.encode(fileName, "utf-8");
    }
    return fileName;
}
```

servlet 3.0

2018年6月10日 11:02

一、新特性概述
1.注解代替 web.xml 文件
2.异步处理
3.对上传的支持
但是，除了 Tomcat，其他服务器对 servlet 3.0的支持还不是特别完善，

二、注解
注解的好处是配置简单，缺点是注解要写在源码中，不方便修改。
1.@WebServlet
这个注解用在 servlet 上，相当于〈servlet〉标签，注解的属性就相当于〈servlet〉的子标签和〈servlet-mapping〉标签。

```
<servlet>
   <servlet-name>ServeltDemo1</servlet-name>
   <servlet-class>servlet.ServletDemo1</servlet-class>
    <init-param>
     <param-name>value</param-name>
     <param-value>1</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>ServletDemo1</servlet-name>
    <url-pattern>/ServletDemo1</url-pattern>
</servlet-mapping>
```

替换成注解就是：

```
@WebServlet(urlPatterns="/ServletDemo1",initParams=
{@WebInitParam(name="value",value="1")},loadOnStartup=1)
```

2.@WebListener
把这个注解加在监听器类上就说明这个类是一个监听器。
3.@WebFilter
`@WebFilter(urlPatterns="/*")` 用在过滤器类上。

三、异步处理
1.概念
在服务器没有结束响应之前，客户端看不到任何结果。异步处理的作用就是使得客户端可以在服务器完全结束响应之前就看到已经响应的内容。
2.实现异步处理的基本步骤
（1）获取 AsyncContext 对象。
（2）传递一个 Runnable 对象，启动线程。
（3）servlet 和 filter 必须要在注解中设置 asyncSupported=true。

```
@WebServlet(urlPatterns="/ServletDemo2",asyncSupported=true)
public class ServletDemo2 extends HttpServlet {

    protected void doGet(HttpServletRequest request,final
    HttpServletResponse response) throws ServletException, IOException
    {
        response.setContentType("text/html;charset=utf-8"); // 一定要设
```

```
        置 ContentType 首部字段
        AsyncContext ac = request.startAsync(request, response);
        ac.start(() -> {
            print("start<br/>",response);
            sleep(2000);
            for(char c = 'a'; c <= 'z'; c++) {
                print(c + "<br/>",response);
                sleep(250);
            }
            ac.complete();   // 要通知 Tomcat 线程已经结束
        });
    }

    private void sleep(int millis) {
        try {
            Thread.sleep(millis);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    private void print(String text,HttpServletResponse response) {
        try {
            response.getWriter().print(text);
            response.getWriter().flush();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    protected void doPost(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException
    {
        doGet(request, response);
    }

}
```

如果发现 IE 似乎不支持异步，说明响应报文主体的数据大小不足512b，还需要再输出一些别的数据。

四、对上传的支持

1. servlet 3.0 提供了专门的上传文件的接口。
2. 基本步骤

（1）给 servlet 添加注解 @MultipartConfig

（2）使用 request.getPart 获取 Part 对象，这个对象封装的就是上传的文件。

（3）操作 Part 对象。

```
@WebServlet("/ServletDemo4")
@MultipartConfig
public class ServletDemo4 extends HttpServlet {

    protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
            throws ServletException, IOException {
        request.setCharacterEncoding("utf-8");
        Part part = request.getPart("file");
        String mime = part.getContentType(); // 获取文件的 MIME 类型
        long size = part.getSize(); // 获取文件的大小
        String name = part.getName(); // 获取表单中的字段名称
```

```
            String header = part.getHeader("Content-Disposition"); // 文件
            的请求报文首部
            int length = header.length();
            int start = header.lastIndexOf("filename=\"") + 10;
            int end = length - 1;
            String fileName = header.substring(start, end); // 从首部中截取
            文件名
            part.write("G:" + File.separator + "test" + File.separator +
            fileName); // 保存
        }

        protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
                throws ServletException, IOException {
            doGet(request, response);
        }

    }
```

会话

2018年6月5日　　12:10

一、概念

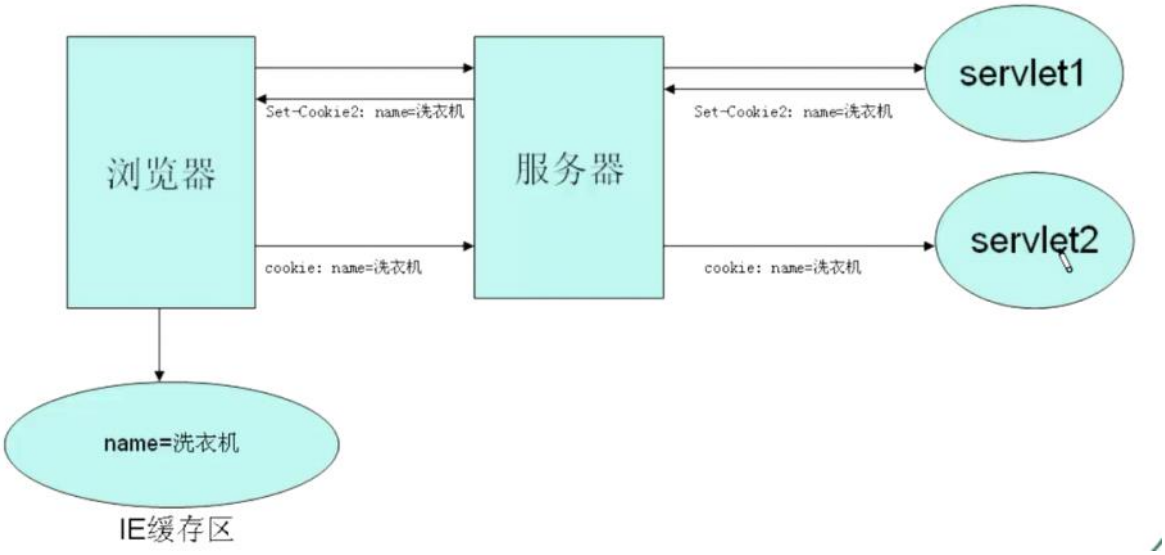
打开浏览器→访问 Web 资源→关闭浏览器，整个过程就叫做一个会话。

二、需要解决的问题

每个用户在与服务器进行交互的过程中都会产生各自的数据，程序要把有用的数据保存下来。比如说，用户将一件商品加入购物车，程序应该保存这件商品，便于用户结账。

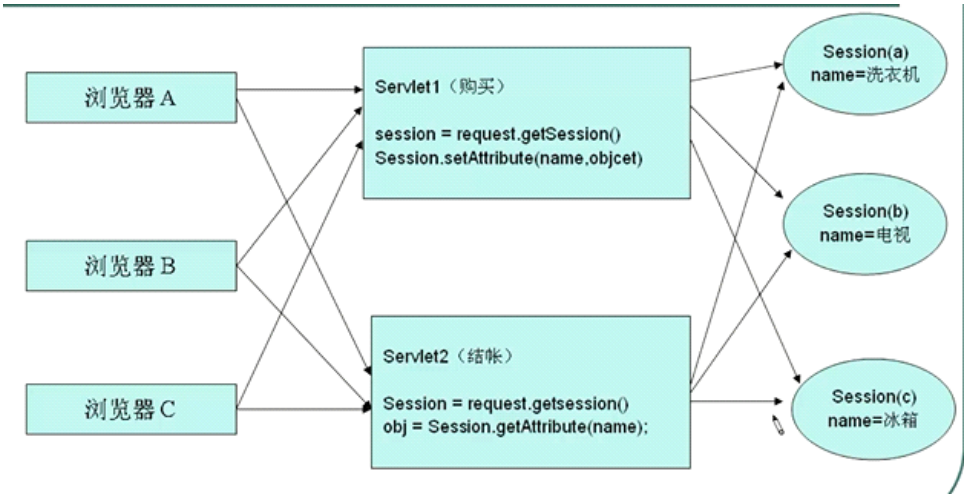
三、cookie

cookie 用于客户端。程序把会话中用户的数据以 cookie 的形式保存在浏览器中，当用户使用浏览器再次访问同一 Web 资源时，程序就可以得到这些数据。在这个技术中，数据保存在客户端。



四、session

session 用于服务器。服务器为每个用户的浏览器创建一个其独享的 session，当用户访问 Web 资源时可以把数据放在各自的 session 中，如果用户再访问别的资源，那么别的资源就可以从 session 中得到数据。在这个技术中，数据保存在服务器。



Cookie

2018年6月5日　　17:47

一、Cookie 类用来创建一个 cookie 对象，构造器是

Cookie(String name, String value)

这说明每个 cookie 都有一个名称来标识，cookie 的 value 用来保存数据。

二、方法
1.setValue 与 getValue
2.setMaxAge 与 getMaxAge ：maxAge 是 cookie 的有效期，即 cookie 在客户端保存多长时间。如果不设置这个值，cookie 的有效期就是浏览器进程时间，也就是说浏览器关闭之后 cookie 就被销毁了。
3.setPath 与 getPath：path 是 cookie 的有效目录，有效目录是指客户端访问这一目录下的 Web 资源时才会把 cookie 回送给服务器。默认的有效目录是发送 cookie 的 servlet 所在目录。
4.setDomain 与 getDomain：设置 cookie 的有效域，就是访问指定域名时才会回送 cookie。其实这个技术并不可行，因为恶意攻击者可以将这个有效域设定为想要攻击的网站，从而加大网站服务器的负担。所以现在浏览器一般都会拒收这种 cookie。
5.getName

三、request 和 response 中与 cookie 相关的方法
1.response 有一个 addCookie 方法，用来在响应报文首部中添加一个相应的 Set-Cookie 字段。
2.request 有一个 getCookies 方法，用于获取客户端回送的 cookie。

四、细节
1.每个 cookie 都只能标识一种信息，一个 cookie 至少含有一个标识该信息的 name 和 value。
2.一个服务器可以给一个浏览器发送多个 cookie，一个浏览器可以存储多个服务器发送的 cookie。
3.浏览器允许存放的 cookie 数量、站点允许存放的 cookie 数量都是有限的。每个 cookie 的内存大小也有限制，为4kb。
4.如果 setMaxAge(0)，则是通知浏览器删除该 cookie。删除 cookie 的时候，path 要一致。

五、应用示例
1.当用户访问网站时，显示用户的上一次访问时间。 有一个 cookie 的 名称是 lastAccessTime，记录了用户的上一次访问时间。

```
@WebServlet("/CookieDemo1")
public class CookieDemo1 extends HttpServlet {

    protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException
    {
        response.setCharacterEncoding("utf-8");
        response.setContentType("text/html;charset=utf-8");

        PrintWriter out = response.getWriter();
        out.print("您上次访问时间是：");
```

```
            // 获得用户的 cookie
            List<Cookie> cookieList =
            Arrays.asList(request.getCookies());
            List<Cookie> timeList = cookieList.stream().filter(c->
            c.getName()
                    .equals("lastAccessTime")).collect(Collectors.toList
                    ()); // 流操作
            if(!timeList.isEmpty()) {
                long milliseconds =
                Long.parseLong(timeList.get(0).getValue());
                LocalDateTime time = Instant.ofEpochMilli(milliseconds)
                        .atZone(ZoneId.systemDefault()).toLocalDateTime(
                        ); // 将 long 转换为 LocalDateTime
                out.print(time.format(DateTimeFormatter
                        .ofPattern("uuuu-MM-dd HH:mm:ss"))); //格式化
            }

            // 发送新的 cookie，注意设置有效期
            Cookie cookie = new
            Cookie("lastAccessTime",System.currentTimeMillis()+"");
            cookie.setMaxAge(1*30*24*3600);
            cookie.setPath("/WebStudy");
            response.addCookie(cookie);
        }

        protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
        {
            doGet(request, response);
        }

    }
```

2. 显示用户上次浏览过的商品

```
    // 显示首页信息
    @WebServlet("/CookieDemo3")
    public class CookieDemo3 extends HttpServlet {

        protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
        {

            response.setContentType("text/html;charset=utf-8");    // 要通
            知浏览器这是一个 HTML 文件，否则浏览器会将其视为普通文本
            PrintWriter out = response.getWriter();
            // 输出所有商品
            out.print("We have commodities as following:<br/>");
            Map<String,Book> map = DB.getAll();
            Book book;
            for(Map.Entry<String, Book> e : map.entrySet()) {
                book = e.getValue();
                out.print("<a href=\"/WebStudy/CookieDemo2?id=" +
                book.getId() + "\">"    // 超链接的 URL 之后要跟上 bookId 作
                为参数
                + book.getName() + "</a><br/>");
                // 这里只是模拟这种逻辑，实际开发中向页面输出内容不会这么干
            }
            // 显示用户曾经看过的商品
```

```
            out.print("You once watched these commodities:</br>");
            List<Cookie> cookieList =
            Arrays.asList(request.getCookies());
            List<Cookie> bookHistory = cookieList.stream().filter(c ->
            c.getName().equals("bookHistory"))
                    .collect(Collectors.toList());
            if (!bookHistory.isEmpty()) {
                String[] ids =
                bookHistory.get(0).getValue().split("\\#"); // 这个
                cookie 值的格式是1,2,3
                for (String id : ids) {  // 获取每个 id 对应的 book
                    String name = DB.getAll().get(id).getName();
                    out.print(name + "<br/>");
                }
            }
        }

        protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
        {
            doGet(request, response);
        }

}

// 显示商品详细信息
@WebServlet("/CookieDemo2")
public class CookieDemo2 extends HttpServlet {

        protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
        {

            response.setContentType("text/html;charset=utf-8");    // 要通
            知浏览器这是一个 HTML 文件，否则浏览器会将其视为普通文本
            PrintWriter out = response.getWriter();
            // 根据 cookie 显示相应商品
            String id = request.getParameter("id");
            String name = DB.getAll().get(id).getName();
            out.print(id + "<br/>");
            out.print(name + "<br/>");

            // 回送 cookie
            Cookie cookie = generateCookie(id,request);
            response.addCookie(cookie);
        }

        private Cookie generateCookie(String id,HttpServletRequest
        request) {

            String bookHistory = null;
            List<Cookie> cookieList =
            Arrays.asList(request.getCookies());
            List<Cookie> history = cookieList.stream().filter(c ->
            c.getName().equals("bookHistory"))
                    .collect(Collectors.toList());
            if (!history.isEmpty()) {
                bookHistory = history.get(0).getValue();
            }

            if (bookHistory == null) { // 用户第一次访问网站
```

```
                bookHistory = id;
            } else {
                LinkedList<String> histories = new LinkedList<String>(
                        Arrays.asList(bookHistory.split("\\#")));
                if (histories.contains(id)) { // 用户曾经浏览过这个商品
                    histories.remove(id); // 先删除曾经的记录
                } else { // 用户第一次浏览这个商品
                    if (histories.size() > 3) { // 浏览记录最多显示三件商品
                        histories.removeLast();
                    }
                }
                histories.addFirst(id);
                String[] arr = histories.toArray(new String[0]);
                StringBuffer bf = new StringBuffer();
                for(String s : arr) {
                    bf.append(s + "#");
                }
                bookHistory = bf.deleteCharAt(bf.length() -
                1).toString(); // 最后会多出一个#，所以要删除
            }

            Cookie cookie = new Cookie("bookHistory",bookHistory);
            cookie.setMaxAge(30 * 24 * 3600);
            cookie.setPath("/WebStudy");
            return cookie;
        }

        protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
        {
            doGet(request, response);
        }

    }


    class DB { // 模拟数据库
        private static Map<String, Book> map = new LinkedHashMap<>(); //
        为了使元素有序所以使用 LinkedHashMap
        static {
            map.put("1", new Book("1", "book1"));
            map.put("2", new Book("2", "book2"));
            map.put("3", new Book("3", "book3"));
        }

        public static Map<String, Book> getAll() {
            return map;
        }
    }

    class Book{
        private String id;
        private String name;

        public Book() {}

        public Book(String id, String name) {
            super();
            this.id = id;
            this.name = name;
        }
        public String getId() {
```

```
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

}
```

HttpSession

2018年6月5日　　19:09

一、每个 session 都唯一标识了一个浏览器的一次会话的数据。

```
@WebServlet("/SessionDemo1")
public class SessionDemo1 extends HttpServlet {

    protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException
    {
        HttpSession session = request.getSession();  // 得到 session
        session.setAttribute("product", "washing machine"); // 存放数据
    }

    protected void doPost(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException
    {
        doGet(request, response);
    }

}

@WebServlet("/SessionDemo2")
public class SessionDemo2 extends HttpServlet {
    protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException
    {
        HttpSession session = request.getSession(false);
        String product = (String) session.getAttribute("product");
        response.getWriter().write(product);

    }

    protected void doPost(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException
    {
        doGet(request, response);
    }

}
```

二、session 生命周期

1.第一次创建会话时，服务器才会创建出一个 session。注意，重载的 getSession(false) 方法的功能是只获取不创建 session。比如说有一个 servlet 的功能是将商品加到购物车内，那么这个 servlet 需要创建新的 session；负责结账的 servlet 则不需要创建 session，只需获取已存在的 session。

2.默认情况下30分钟以内没有程序使用 session 的话服务器就会销毁 session，不管浏览器有没有关闭。

在 web.xml 添加 <session-config> 标签可以设置 session 的生存时间：

```
<session-config>
    <session-timeout>10</session-timeout>
</session-config>
```

这里的数值单位是分钟。也可以通过调用 session 对象的 invalidate 方法来手动销毁 session 对象。

3.经测试，目前 Chrome 创建新的 session 的时机：第一次访问资源时创建一个新的 session；在同一个浏览器实例中用一个新的标签页访问同一资源，不会创建新的 session；在一个新的浏览器实例中访问同一资源，也不会创建新的 session。三十分钟之后再次访问时才会创建新的 session。

三、session 工作原理

每个 session 都有一个唯一标识：session ID。当服务器创建了一个 session 时，给客户端发送的响应报文就包含了 Set-Cookie 字段，其中有一个名为 JSESSIONID 的键值对，这个键值对就是 session ID。客户端收到后就把 cookie 保存在浏览器中，并且之后发送的请求报文都包含 session ID，这样服务器就能区分出每一个 session。

但是，服务器在创建 cookie 时并没有设置 cookie 的有效期，一旦浏览器被关闭，数据就会丢失。解决这一问题的一个办法是，创建 session 时就获取这个 session 的 ID，然后创建一个 cookie，把 session 存入这个 cookie，并且设置 cookie 的有效期，最后再把 cookie 发送给浏览器。

```
HttpSession session = request.getSession();  // 得到 session
session.setAttribute("product", "washing machine"); // 存放数据
String sid = session.getId();   //得到 session 的 id
Cookie cookie = new Cookie("JSESSIONID",sid);
cookie.setMaxAge(30*60);       // 设置有效期
cookie.setPath("/WebStudy");   // 设置有效路径
response.addCookie(cookie);
```

但是，有可能用户设置浏览器禁止接受 cookie，那么服务器就无法从请求报文首部获取 session ID 从而区分不同的 session 了。不过除了 cookie 可以携带 session ID 以外，URL 也可携带 session ID，所以可以先获取 session ID，并把 session ID 附加在网页的 URL 之后，这样服务器也能区分每一个 session。response 的 encodeURL 就可以实现在 URL 之后加上 session ID 的效果。不过由于 URL 最终是写在 HTML 页面中的，所以这个工作最好由前端来完成。注意只有当浏览器禁用 cookie 时，服务器才会在 URL 之后加上 session ID。

但是关闭浏览器之后用户数据丢失的问题就无法解决了，因为每次关闭浏览器之后再打开浏览访问页面，这时的请求报文中没有包含 session ID，服务器就会创建一个新的 session。

四、应用实例

1.用户登录

```
@WebServlet("/SessionDemo3")
public class SessionDemo3 extends HttpServlet {

    protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
            throws ServletException, IOException {
        String username = request.getParameter("username");
        String password = request.getParameter("password");

        List<User> users = DB.getAll();
        List<User> user = users.stream()
                .filter(u -> u.getUserName().equals(username) &&
                u.getPassWord().equals(password))
                .collect(Collectors.toList());
        if (!user.isEmpty()) {
            request.getSession().setAttribute("user", user.get(0));
            // 登录成功则向 session 中存入一条记录
            response.sendRedirect("/WebStudy/index.html");  // 跳转到
            首页
```

```
                return;
            }
            response.getWriter().write("Username or password is
            incorrect!");
        }

        protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
                throws ServletException, IOException {
            doGet(request, response);
        }

    }

    class DB {
        private static List<User> list = new ArrayList<>();

        static {
            list.add(new User("123","aaa"));
            list.add(new User("123","bbb"));
            list.add(new User("122","aaa"));
        }

        public static List<User> getAll(){
            return list;
        }
    }

    @WebServlet("/SessionDemo4")
    public class SessionDemo4 extends HttpServlet {

        protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
                throws ServletException, IOException {
            HttpSession session = request.getSession(false);
            if (session == null) {
                response.sendRedirect("/WebStudy/index.html");
            } else {
                session.removeAttribute("user"); //退出登录就是把 session
                中的相关属性删除
                response.sendRedirect("/WebStudy/login.html");
            }
        }

        protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
        {
            doGet(request, response);
        }

    }
```

在一次会话中有多个资源需要得知用户的登录情况，所以要用 session。

2. 校验验证码

登录或者注册的时候一般要求用户填写验证码，那么程序就要检验用户是否填写了验证码；如果写了，是否填写正确。

首先要在生成验证码时把验证码存入 session 中，因为生成验证码的 servlet 和处理用户登录的 servlet 不是同一个。因此，在生成验证码的 servlet 中，生成验证码之后把验证码存入 session。

然后，处理用户登录的请求：

```
if(verifycode == null) { //用户没写验证码
    response.getWriter().write("please fill in the verifycode");
    return;
}
if(!
request.getSession(false).getAttribute("verifycode").equals(verifycode
)) { //验证码错误
    response.getWriter().write("verifycode is incorrect");
    return;
}
```

3.放置表单重复提交

比如说注册表单，同一个用户名和密码不能重复注册，所以在处理用户提交的表单时要注意检验。为了能有效地防止出现这种问题，需要在前端和后端都进行检验。

（1）前端代码

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>login</title>
<script>
    function dosubmit() {
        var input = document.getElementById("submit");
        input.disabled = "disabled"; // 点击提交按钮后按钮不可用
        return true;
    }

    function refresh(img) {
        // 为地址添加一个随机数，说明是一个新地址；
        //否则浏览器认为这是同一个地址，就从缓存中得到同一张图片
        img.src = img.src + "?" + new Date().getTime();
    }
</script>
</head>
<body>
    <form action="/WebStudy/SessionDemo3" method="post"
    onsubmit="return dosubmit()">
        username: <input type="text" name="username"> <br />
         password: <input type="text" name="password"> <br />
         verifycode: <input type="text" name="verifycode"><img
        src="/WebStudy/ServletDemo10" onclick="refresh(this)"><br />
       <input type="submit" value="login">
    </form>
</body>
</html>
```

（2）后端代码

用户提交表单时，程序会生成一个 token，这个 token 可以作为表单的隐藏域一起提交；同时程序把 token 存入 session 中。处理表单的 servlet 进行检验，只有当客户端提交的 token 和 session 中的 token 一致时，表单才能成功提交。

生成 token 的代码：

```
class TokenGenerator { // 采用单例模式，用来为每一个表单产生一个唯一标识符

    private TokenGenerator() { }

    private static final TokenGenerator instance = new
    TokenGenerator();
```

```
    public static TokenGenerator getInstace() {
        return instance;
    }

    public String generate() { // 产生随机数作为标识符
        String token = System.currentTimeMillis() + new
        Random().nextInt() + "";
        try {
            // 生成的随机数长度不一，不适合用来做标识符；所以改用随机数的数据摘
            要作为标识符
            MessageDigest md = MessageDigest.getInstance("md5");
            byte[] digest = md.digest(token.getBytes());
            // base64 编码
            return Base64.getEncoder().encodeToString(digest);
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
    }
}
```

处理表单提交申请的代码：

```
public class Session5 extends HttpServlet {

    protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException
    {
        if(isTokenValid(request)) {// 检验 token
            request.getSession().removeAttribute("token"); // 如果有
            效，则将 token 从 session 中删除
            //提交成功
        } else {
            //提交失败
        }
    }

    private boolean isTokenValid(HttpServletRequest request) {

        String clientToken = request.getParameter("token");
        if(clientToken == null) { // 客户端提交的表单中没有 token
            return false;
        }

        String serverToken = (String)
        request.getSession().getAttribute("token");
        if(serverToken == null) { // session 中没有 token，说明之前已经成
        功提交过一次了
            return false;
        }

        if(!clientToken.equals(serverToken)) { // 两个 token 不一致
            return false;
        }

        return true;
    }
```

```
        protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
        {
             doGet(request, response);
        }

    }
```

其实，这种防止表单重复提交的方法已经在一些框架中实现了。

域对象间的数据共享/会话中数据的保存

2018年6月7日　　13:14

1. ServletContext 通常会保存一些全局性的数据，比如整个 Web 应用下的所有资源都可能会用到的数据。
2.ServletRequest 通过转发请求的方式使多个 servlet 共享数据。
3.session 不需要转发，只要 session 没有被销毁，多个 servlet 都可以获取 session 中的数据。有些数据不适合保存在 ServletContext 中，则保存在 session 中。

概述

2018年6月7日　　10:07

一、Java Server Pages，简称 JSP ，是一种特殊的 servlet。JSP 可以在页面中编写 Java 代码，并且允许在页面中动态地获取 request、response 等对象，实现与浏览器的交互。

二、调用和运行原理
1.当客户端访问 JSP 时，实际上访问的是一个 servlet。这个 servlet 会把 JSP 中的 Java 代码的执行结果和 JSP 中的 HTML 代码一起进行排版，然后输出给浏览器。
2.服务器在调用 JSP 时，会给 JSP 提供一些已经提前创建好的对象，比如 response、request、context、config 等等。

三、最佳实践
JSP 负责数据的输出，servlet 只负责响应请求产生数据并把数据转发给 JSP。所以说在一次会话中 servlet 和 JSP 协同工作处理同一个请求。注意这里要用转发而不是重定向。

语法

2018年6月7日　　11:27

一、模板元素
JSP 中的 HTML 称为模板元素，因为这些代码用来控制输出的样式。

二、脚本表达式
用于向浏览器输出数据，比如 <%=time%> 输出了 time 这个变量的值。 servlet 会将这个表达式处理为 out.print(time);

三、脚本片段
脚本片段就是在页面中嵌入的多行 Java 代码。servlet 将这些代码原封不动地写到 service 方法中。在一个 JSP 页面中可以有多个脚本片段，脚本片段之间可以嵌入文本、HTML、其他 JSP 元素。多个脚本片段的代码可以相互访问。

```
<%
    // Java代码
%>
```

四、声明
一般情况下 JSP 中的所有代码会被写到 servlet 的 service 方法中，但是声明中的代码会被放在 service 方法的外面。所以声明可以用来定义 service 方法需要用到的静态成员等等。

```
    <%!
        // Java代码
    %>
```

声明不能引用 JSP 中的隐式对象。

五、注释
JSP 引擎在将 JSP 页面解析成 servlet 程序时将忽略注释中的内容。
<%-- 注释内容 --%>

国际化

2018年6月7日　　19:18

```
//实现国际化的最基本要求就是能根据用户的语言环境切换页面的语言文字
//通常会将不同的语言文字保存在配置文件中，然后根据情况进行配置
@WebServlet("/InternationalDemo")
public class InternationalDemo extends HttpServlet {

protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException
{
// 获取用户的语言环境
Locale locale = request.getLocale();
// 使用 ResourceBundle 处理配置文件，配置文件的命名要按照一定的规范：基本名称
+Locale
// getBundle 的第一个参数则是文件的基本名称，第二个参数就是 Locale
ResourceBundle rb = ResourceBundle.getBundle("res",locale);
String username = rb.getString("username");
String password = rb.getString("password");
}


protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException
{
doGet(request, response);
}

}
```

把其中涉及到语言文字处理的代码放到前端页面中，就可以实现最基本的国际化了。

概述

2018年6月7日　　14:50

一、观察者模式
观察者模式就是把监听器注册在事件源上，当特定事件发生时监听器就会调用特定的方法。

二、JavaWeb 中的监听器
JavaWeb 中的事件源就是 ServletContext,HttpSession,ServletRequest 这三个域对象。每个域对象都有两种基本监听器，一种是生存周期监听器，监听域对象的创建和销毁；一种是属性监听器，监听域对象属性的添加、替换和删除。所以总共有六个基本监听器类。
三个生存周期监听器：
ServletContextListener、HttpSessionListener、ServletRequestListener
三个属性监听器：
ServletContextAttributeListener、HttpSessionAttributeListener、ServletRequestAttributeListener
-> 三种域对象的创建时机总结：
（1）ServletContext：它表征的是整个 Web 应用，因此只要 Web 应用开始运行，它就会被创建出来，而且一个 Web 应用只有一个 ServletContext 对象。
（2）HttpSession：整个 Web 应用第一次调用 request.getSession() 方法时被创建。
（3）ServletRequest：客户端每请求一次动态资源，服务器就会创建一个 ServletRequest 对象。
->三种域对象的销毁时机总结：
（1）ServletContext：Web 应用被移除或者服务器关闭。
（2）HttpSession：超出 session 的生存时间。
（3）ServletRequest：

三、实现监听器的基本步骤
1.写一个监听器类，实现了某个监听器接口。
以 ServletContext 的生存周期监听器为例：

```
@WebListener
public class ListenerDemo1 implements {

    // 在域对象将要被销毁之前调用
    public void contextDestroyed(ServletContextEvent sce)  {
            System.out.println("going to be destroyed");
    }

    // 在域对象创建之后调用
    // 通常用这个方法执行一些在服务器启动时就要完成的任务
    public void contextInitialized(ServletContextEvent sce)  {
            System.out.println("has been initialized");
    }

}
```

2.在 web.xml 中注册，或者用 @WebListener 注解。如果是在 web.xml 中进行配置，添加 <listener> 标签：

```
<listener>
   <listener-class>edu.whu.listener.ListenerDemo1</listener-class>
</listener>
```

四、事件对象

事件对象就是事件源。

1.三个生存周期监听器的事件对象

（1）ServletContextEvent：getServletContext()

（2）HttpSessionEvent：getSession

（3）ServletRequest：getServletRequest

2.三个属性监听器的事件对象

（1）ServletContextAttributeEvent

（2）HttpSessionBindingEvent

（3）ServletRequestAttributeEvent

感知监听

2018年6月7日　　16:16

HttpSession 还有两个相关的监听器，HttpSessionBindingListener 和 HttpSessionActivationListener。
特点：这两个监听器用来添加到 JavaBean 上而不是域对象上；这两个监听器都无需在 web.xml 中注册。

一、HttpSessionBindingListener
实现了这个接口的 JavaBean 可以监听自身是否被绑定为 session 的属性。

二、HttpSessionActivationListener
1.session 的序列化
客户端在访问 Web 资源时服务器创建了 session 来保存数据，然后用户关闭浏览器，那么这些 session 就会被序列化并保存在服务器上，等用户再次访问时服务器就可以获取 session 中的数据。$CATALINA_BASE\work 中 Web 应用的目录下的 SESSIONS.ser 文件就是序列化的 session。
在 context.xml 文件的 <Context> 标签下添加一个子标签 <Manager pathname=""/> 就可以关闭 session 序列化的功能。
2.session 的钝化和活化
（1）一般情况下 session 保存在服务器的内存中，但是太多的 session 会加大服务器的负担，于是服务器就会把长时间没有被使用的 session 通过序列化保存在硬盘上。这个过程就叫做钝化。session 钝化时保存在 session 中的对象也会被钝化。
（2）当 session 需要被再次使用时，服务器就会对其进行反序列化，重新加载到内存中，这个过程叫做活化。session 活化时保存在 session 中的对象也会被活化。
可以通过 XML 文件对钝化和活化进行配置。在 $CATALINA_BASE\conf\catalina\localhost 目录下添加一个 XML 文件，文件名是 Web 应用名称。文件内容为：

```
    <Context>
    <Manager className="org.apache.catalina.session.PersistentManager"
maxIdleSwap="1">
        <Stroe className="org.apache.catalina.session.FileStore"
directory="mysession">
    </Manager>
    </Context>
```

maxIdleSwap 设定了 session 的最大闲置时间，超过这个时间 session 就会被钝化。这个属性的最小值是1分钟。directory 设定了钝化后的文件的存储目录，假设 Web 应用名称为 web，则文件的存储目录为 $CATALINA_BASE\conf\catalina\localhost\web\mysession。钝化后的 session 以 .SESSION 文件保存在硬盘上，文件名是 SESSION ID。
也可以对 context.xml 进行配置，这样配置的话则是对所有的 Web 应用都有效。
3.HttpSessionActivationListener 就是钝化和活化的监听器。实现了这个接口的 JavaBean 可以监听自身是否随着 session 被钝化和活化。注意，由于钝化和活化用到的是序列化技术，所以 JavaBean 还要实现 Serializable 接口。

概述

2018年6月7日　　19:24

一、功能
1.过滤器有拦截请求的功能，过滤器可以允许某些请求得到资源，也可以阻止某些请求得到资源。
2.过滤器通常是针对一组资源进行过滤请求的操作，而 servlet 只能处理对自身的访问请求。
3.过滤器是单例的。

二、实现过滤器的基本步骤
1.写一个实现了 Filter 接口的类。Filter 接口有三个方法：
（1）init(FilterConfig fConfig)：服务器在启动时就创建过滤器，创建完成后调用这个方法进行初始化。
（2）doFilter(ServletRequest request, ServletResponse response, FilterChain chain)：每次过滤时都会调用这个方法。
（3）destory()：销毁过滤器之前执行，用来释放非内存的资源。过滤器会在服务器关闭时被销毁。
这就是过滤器的生存周期。
2.在 web.xml 中配置或者使用 @WebFilter 注解，指定过滤对哪个资源的访问。
（1）在 web.xml 中配置

```
<filter>
   <filter-name>name</filter-name>
   <filter-class>package.name</filter-class>
</filter>
<filter-mapping>
   <filter-name>name</filter-name>
   <url-patter>/*</url-pattern>
</filter-mapping>
```

<url-patter>/*</url-pattern> 则意味着访问这个 Web 应用下的任意资源的请求都要经过这个过滤器的过滤。如果要指定过滤对某个 servlet 的访问，就在
<filter-mapping> 下添加 <servlet-name> 子标签。
（2）使用注解：@WebFilter 的默认属性是 urlPatterns，就是要对访问哪些资源的请求进行过滤。

三、Filter 接口方法详解
1.init
这个方法的参数 FilterConfig 与 ServletConfig 类似，可以通过它获取初始化参数、过滤器的名称等。
2.doFilter
chain.doFilter(request, response); 这一句代码的作用是允许过滤之后的请求访问资源，也就是说调用了客户端请求访问的 servlet 的 service 方法。这一句代码之后代码在执行完 servlet 的 service 方法后会接着执行。

四、多个过滤器
1.在有多个过滤器的情况下，chain.doFilter(request, response); 的作用：（1）如果还有下一个过滤器，则执行下一次过滤（2）如果已经是最后一个过滤器，则执行目标 servlet。
2.多个过滤器的执行顺序就是 <filter-mapping> 在 web.xml 文件中的部署顺序。如果是基于注解的，那么就是通过对 Filter 类的名字进行字典排序来决定执行顺序。

五、拦截方式

客户端可以通过多种方式对资源进行访问，比如直接通过请求，也可以是通过转发。过滤器可以拦截请求、转发、包含（通过 RequestDispatcher#include 或者 <j sp:include>标签访问）和错误（目标资源 在web.xml 中配置为 <error-page>中时，并且真的出现了异常，转发到目标资源）。一个过滤器可以有多种拦截方式，默认的拦截方式是拦截请求。

1.在 web.xml 中进行配置

在 <filter-mapping> 标签下添加子标签 <dispatcher>，用来指定过滤器的拦截方式：

```
<dispatcher>REQUEST</dispatcher>
<dispatcher>FORWARD</dispatcher>
<dispatcher>INCLUDE</dispatcher>
<dispatcher>ERROR</dispatcher>
```

2.使用注解

@WebFilter 有一个属性 dispatcherTypes，用来设置拦截方式：

```
@WebFilter(urlPatterns= {"/ServletFilter"},dispatcherTypes=
{DispatcherType.REQUEST,DispatcherType.FORWARD})
```

六、应用场景

1.执行目标资源之前进行预处理，比如设置字符集。这种过滤器并不是真的要拦截什么，只是为执行目标资源做了一些预处理工作。

2.判断客户端是否满足某些条件从而决定是否拦截，比如检验当前用户有无访问权限等。

3.在目标资源执行之后，做一些后续的特殊处理工作，例如把目标资源输出的数据进行处理，称为回程拦截。

应用实例

2018年6月7日　　20:42

一、统计每个 ip 的访问次数
用一个 map 来保存每个 ip 的访问次数。这些数据应该是全局性的即对整个 Web 应用下的所有资源都有效，所以把这个 map 保存到 ServletContext 中， ServletContextListener 中完成这个 map 的创建。

```
// 在服务器启动时创建 map 并保存到 ServletContext 中
@WebListener
public class ContextListener implements ServletContextListener {

    public void contextDestroyed(ServletContextEvent sce)  {

    }

    public void contextInitialized(ServletContextEvent sce)  {
        Map<String,Integer> statics = new LinkedHashMap<>();
        sce.getServletContext().setAttribute("map", statics); // 将
map 保存到 context 中
    }

}

// 进行统计工作
@WebFilter("/*")
public class CountFilter implements Filter {

    private ServletContext application = null;

    public void destroy() {

    }

    public void doFilter(ServletRequest request, ServletResponse
    response, FilterChain chain) throws IOException, ServletException
    {
        Map<String,Integer> statics = (Map<String, Integer>)
        application.getAttribute("map");
        String ip = request.getRemoteAddr(); // 获取来访 ip


        if(statics.containsKey(ip)) { // 不是第一次访问
            int count = statics.get(ip);
            statics.put(ip, ++count);
        } else { // 第一次访问
            statics.put(ip, 1);
        }


        application.setAttribute("map", statics); // 把 map 放回
        context 中

        chain.doFilter(request, response);
    }

    // 服务器启动时执行这个方法，且只执行一次
```

```
    public void init(FilterConfig fConfig) throws ServletException {
        application = fConfig.getServletContext();
    }

}
```

二、基于角色的权限管理（RBAC，Role-Based Access Control）
当用户试图访问某些资源时，先用过滤器来判断用户是否具有相应权限。

三、解决全站的乱码问题
这要用到装饰器模式：

```
// 装饰器类，用来设置 GET 请求的编码
public class EncodingRequest extends HttpServletRequestWrapper{

    private HttpServletRequest request;     // 真正的 request 对象

    public EncodingRequest(HttpServletRequest request) {
        super(request);
        this.request = request;
    }

    @Override
    public String getParameter(String name) { // 处理编码
        String value = request.getParameter(name);
        try {
            value = new String(value.getBytes("iso-8859-1"),"utf-8");
        } catch (UnsupportedEncodingException e) {
            throw new RuntimeException(e);
        }
        return value;
    }
}

@WebFilter("/*")
public class EncodingFilter implements Filter {

    public void destroy() {

    }

    public void doFilter(ServletRequest request, ServletResponse
    response, FilterChain chain)
            throws IOException, ServletException {
        HttpServletRequest req = (HttpServletRequest) request;
        if (req.getMethod().equals("GET")) { // 处理 GET 请求的编码
            // 装饰器模式，增强 HttpServletRequest 的功能
            EncodingRequest encodingReq = new EncodingRequest(req);
            chain.doFilter(encodingReq, response);
        }
        if (req.getMethod().equals("POST")) { // 处理 POST 请求的编码
            request.setCharacterEncoding("utf-8");
            chain.doFilter(request, response);
        }
    }

    public void init(FilterConfig fConfig) throws ServletException {

    }
```

```
}
```

使用装饰器模式的原因：在处理 GET 请求时，比如客户端的请求中带了一个 username 的参数，那么在使用过滤器之前是这么处理的

```
String username = request.getParameter("username");
username = new String(username.getBytes("iso-8859-1"),"utf-8");
```

但如果想要根本地解决乱码问题，就不能只是针对某个参数来处理。客户端的请求中包含的参数是可变的，所以要写一个 HttpServletRequest 的装饰器类，使经过装饰的 request 不管获取什么参数都能进行编码的处理。所以在这里写了 EncodingRequest 类（注意这个类继承 HttpServletRequestWrapper，它实现了 HttpServletRequest 的其他方法），这个类覆写了 getParameter 方法，获得参数之后就进行编码的处理。然后在调用 doFilter 方法时，参数中的 request 就不是原本的 request，而是经过装饰的 EncodingRequest 的一个实例。

概述

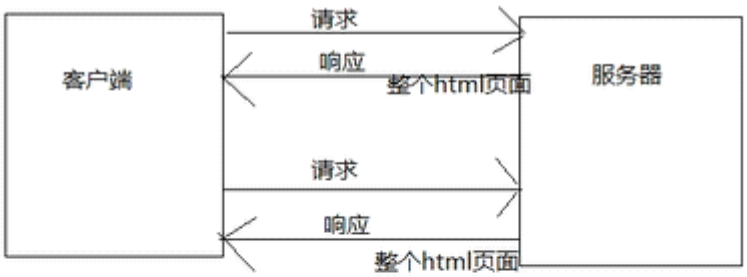
2018年6月21日　　19:37

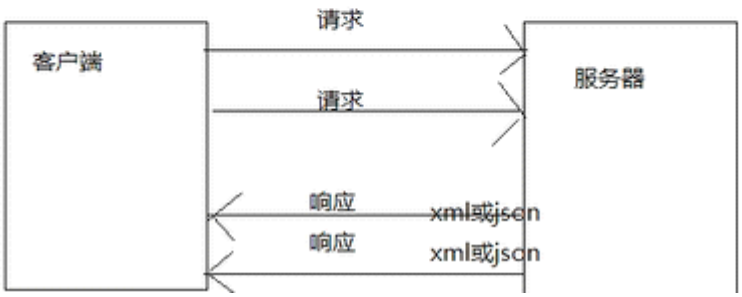
一、异步

发送一个请求之后，无需等待服务器的响应就可以发送第二个请求。可以用 JavaScript 来接受服务器的响应，然后“局部”刷新页面。

AJAX，Asynchronous Javascript And XML，不是新的编程语言，而是一种使用现有标准的新方法。不重新加载整个页面的情况下，可以与服务器交换数据并更新部分网页内容，此时并不需要传输整个页面。

数据的格式：text、XML、JSON





二、AJAX 的常见应用场景

在搜索引擎的输入框内输入关键字，就会出现相关推荐，这个时候页面并没有整个刷新，而是只刷新了出现推荐内容的这一部分。

三、AJAX 的优缺点

1.优点

（1）异步交互，改进改善用户体验。

（2）服务器只需响应页面中刷新的一部分内容，所以减轻了服务器的压力。

2.缺点

（1）AJAX 无法应用在所有的场景。

（2）AJAX 会增加对服务器的访问次数，这就又增加了服务器的压力。

AJAX 基本操作

2018年6月21日　　20:11

对 AJAX 而言，最重要的就是 XMLHttpRequest 对象。
发送异步请求的四步操作：

一、得到 XMLHttpReques 对象

```
function createXMLHttpRequest() {
    try{ // 大部分浏览器
        return new XMLHttpRequest();
    } catch(e){
        try{ // IE 6.0
            return new ActiveXObject("Msxml2.XMLHTTP");
        } catch(e){
            try{ // IE 5.5 及更早版本
                return new ActiveXObject("Microsoft.XMLHTTP");
            } catch(e){
                alert("不支持 ajax");
                throw e;
            }
        }
    }
}
var xmlHttp = createXMLHttpRequest();
```

二、打开与服务器的连接
open 方法打开与服务器的连接，需要三个参数：
（1）请求方式：GET POST
（2）请求的 URL：用来指定所请求的资源
（3）请求是否为异步：true 则为异步
如果是发送 POST 请求，还要设置 Content-Type 字段：
xmlHttp.setRequestHeader("Content-Type","application/x-www-form-urlencoded")

三、发送请求
send 方法用来发送请求，如果不调用这个方法可能会造成部分浏览器无法发送请求。参数就是请求报文主体内容，如果是 GET 方法，则为 null。

四、监听状态
在 XMLHttpRequest 对象的 onreadystatechange 事件上注册监听器，这个事件用来表征对象的状态是否发生改变。XMLHttpRequest 对象有以下5种状态：
（1）0：刚刚创建，还没有调用 open 方法
（2）1：请求开始，即调用了 open 方法但是还没有调用 send 方法
（3）2：发送请求，调用了 send 方法
（4）3：服务器已经开始响应，但不知道是否响应完
（5）4：服务器**已经结束响应**，一般说来只需关注这个状态

```
xmlHttp.onreadystatechange = function () {
    // 得到 XMLHttpRequest 对象的状态
    var state = xmlHttp.readyState;
    // 得到服务器响应的状态码
    var status = xmlHttp.status;
    if(state === 4 && status === 200){ // 判断服务器是否已经成功地完成响应
```

```
            var contentText = xmlHttp.responseText; // 得到服务器的响应内容，
            格式为文本
        }
    }
```

JSON

2018年6月22日　　13:03

JSON 是 JavaScript 提供的一种数据交换格式。

一、语法
{} 包围起来的是对象，{}里面的属性可以是 null、数值、字符串、数组（用 [] 包围）、布尔值。属性必需用双引号包围而不能用单引号。属性的格式是 "name":value

```
var person =  {"name":"ppp","age":50};
```

有时候需要进行这样的转换：

```
var str = "{\"name\":\"aaa\",\"age\":45}";
var person = eval("(" + str + ")");
```

二、与 XML 的比较
可读性：XML 更佳
解析难度： JSON 本身就是 JavaScript 的一个子集，用 JavaScript 来解析自然很简单
流行度： XML 已经在多个领域流行多年，JSON 更适于与 AJAX 配合使用

三、将 Javabean 转换成 JSON——json-lib
1. jar 包

2. 核心类
- JSONObject：继承自 Map
- JSONArray：继承自 List

3. 示例

```
public class JsonTest {

    @Test
    public void test1() {
        JSONObject map = new JSONObject(); // 直接生成单个 JSON 对象
        map.put("name", "aaa");
        map.put("age", 45);

        System.out.println(map.toString());
    }

    @Test
    public void test2() {
        // 由已有的 Javabean 对象生成 JSON 对象
        Book book = new Book("Harry Potter","Rowling");
```

```
        JSONObject json = JSONObject.fromObject(book);
        System.out.println(json.toString());
    }

    @Test
    public void test3() {
        Book book1 = new Book("Harry Potter","Rowling");
        Book book2 = new Book("sadas","ass");

        JSONArray array = new JSONArray();
        array.add(book1);
        array.add(book2);

        System.out.println(array);
    }

    @Test
    public void test4() {
        Book book1 = new Book("Harry Potter","Rowling");
        Book book2 = new Book("sadas","ass");
        List<Book> books = new ArrayList<>();
        books.add(book1);
        books.add(book2);
       // 由已有的 list 生成 JSONArray
        JSONArray jsons = JSONArray.fromObject(books);
        System.out.println(jsons);
    }
}
```