javaWeb-Notes

1. 概念

> Servlet = server + applet
>
> 运行在服务器的小程序

author大白话：Servlet就是一个接口，只能实现了这个接口的java类才能被服务器所识别并运行。将来我们定义的类，要实现Servlet接口，复写它的抽象方法。这样看来，Servlet和JDBC的本质其实是一样的，定义了“规则”。

① 编写第一个Servlet程序

- 实现接口的类

```
package pers.hhj.day01;

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

public class ServletDemo1 implements Servlet {

    @Override
    public void init(ServletConfig servletConfig) throws ServletException {

    }

    @Override
    public ServletConfig getServletConfig() {
        return null;
    }

    @Override
    public void service(ServletRequest servletRequest, ServletResponse servletResponse) throws ServletException, IOException {
        System.out.println("helloservlet");
    }

    @Override
    public String getServletInfo() {
        return null;
    }

    @Override
    public void destroy() {

    }
}
```

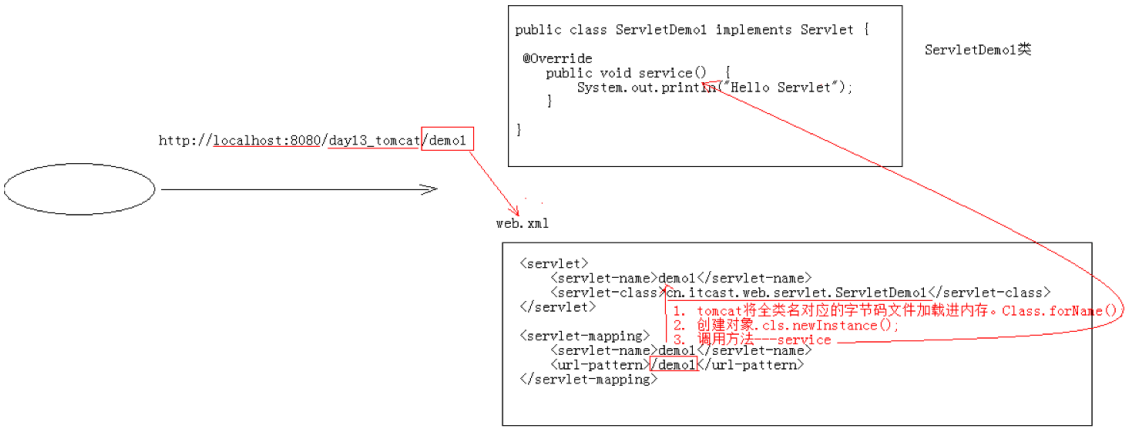
- 配置Servlet

  修改web.xml的内容

```xml
<?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_4_0.xsd"
         version="4.0">

    <!--配置demo1的Servlet-->
    <servlet>
        <servlet-name>demo1</servlet-name>
        <servlet-class>pers.hhj.day01.ServletDemo1</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>demo1</servlet-name>
        <url-pattern>/demo1</url-pattern>
    </servlet-mapping>

</web-app>
```

- 在浏览器上访问http://localhost:8080/web_learning/demo1，控制台会打印helloservlet

1. Servlet执行原理



   - 执行原理：

     1. 当服务器接收到浏览器的请求时，会解析请求URL路径，从而获取要访问的Servlet动态资源路径
     2. 查找web.xml文件，是否有对应的`<url-patter>`标签体内容
     3. 如果有，顺延找到`<servlet-class>`标签体得到全类名
     4. 服务器将该java类的字节码文件加载进内容，并创建对象
     5. 根据对象调用方法

② Servlet的生命周期

先看Servlet实现类的五大基本方法

```
    //五大方法
        public void init(ServletConfig servletConfig) throws ServletException {}

        public ServletConfig getServletConfig() {return null;}

        public void service(ServletRequest servletRequest, ServletResponse servletResponse) throws ServletException, IOException {}

        public String getServletInfo() {return null;}

        public void destroy() {}
```

可以看出Servlet的生命周期分为三个阶段：

- 被创建：执行init方法，执行一次，用于加载资源
  - 默认情况下，第一次访问时创建Servlet对象。
  - 也可以配置servlet对象创建的时机：修改web.xml文件

```
<!--配置demo2的Servlet-->
    <servlet>
        <servlet-name>demo2</servlet-name>
        <servlet-class>pers.hhj.day01.ServletDemo2</servlet-class>
    <!--指定Servlet的创建时间
            1. 第一次被访问时创建，下面值为负数
            2. 在服务器启动时创建，(代价太大，需要时在创建比较好，下面值为0或正数-->
        <load-on-startup>5</load-on-startup>

    </servlet>
```

author大白话

init方法只执行一次，说明servlet在内存中只有一个对象(单例模式)。这就容易产生线程安全问题，比如多个用户同时修改Servlet实现类的成员变量，会产生混乱。

因此每个Servlet的实现类都尽量不要定义成员变量，定义了也不要进行修改

- 提供服务：执行service方法，执行多次

  每个访问Servlet对象时，都会执行service方法

- 被销毁：执行destroy方法，执行一次，用于释放资源

  服务器正常关闭时，会销毁Servlet对象，同时执行destroy方法

2. Servlet3.0新特性：注解

> 我们发现。在我们写Servlet实现类的时候，每一个都要去web.xml配置当前实现类的资源路径patternurl，十分麻烦。
>
> 因此，Servlet3.0就新增了一个注解配置，只要在类上使用@webServlet('资源路径')就可以对Servlet进行配置，不用再使用web.xml进行配置了。

```
    import javax.servlet.*;
    import javax.servlet.annotation.WebServlet;
```

```
import java.io.IOException;

/**
 * 注解特性
 */
@WebServlet ("/demo3")
public class ServletDemo3 implements Servlet {

    @Override
    public void init(ServletConfig servletConfig) throws ServletException
{
     System.out.println("servlet3.0新特性")
    }
```

3. Servlet的体系结构

> 引入：实际开发中，我们需要重写的Servlet抽象方法往往只有service方法，但是我们在创建Servlet的实现类时，也必须实现其他抽象方法，十分的麻烦。
>
> 因此，我们就创建了一个抽象类GenericServlet继承Servlet接口，该类对Servlet接口的其他方法做了默认空实现，只留下了一个service()方法，因此，我们将来定义Servlet的实现类时只需要继承GenericServlet，实现service()方法即可。
>
> 虽然上面这个类也很好，但是实际开发中我们也不用。。。用的是下面这个类
>
> 这个更简单的类是HttpServlet，它是GenericServlet的子类，也是一个抽象类。继承这个类的Servlet实现类之后，你连service方法都不用重写(不得不说，人类在追求'懒'这个方向的创造力是无限的)，因为它把service方法也给实现了，但是需要你重写其他方法doGet()、doPost()...

这样说也不是太清晰，我们来看看HttpServlet的service源码吧

```
 protected void service(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        String method = req.getMethod();
        long lastModified;
        if (method.equals("GET")) {
            lastModified = this.getLastModified(req);
            if (lastModified == -1L) {
                this.doGet(req, resp);
            } else {
                long ifModifiedSince;
                try {
                    ifModifiedSince = req.getDateHeader("If-Modified-
Since");
                } catch (IllegalArgumentException var9) {
                    ifModifiedSince = -1L;
                }

                if (ifModifiedSince < lastModified / 1000L * 1000L) {
                    this.maybeSetLastModified(resp, lastModified);
                    this.doGet(req, resp);
                } else {
                    resp.setStatus(304);
                }
            }
        } else if (method.equals("HEAD")) {
            lastModified = this.getLastModified(req);
```

```
            this.maybeSetLastModified(resp, lastModified);
            this.doHead(req, resp);
        } else if (method.equals("POST")) {
            this.doPost(req, resp);
        } else if (method.equals("PUT")) {
            this.doPut(req, resp);
        } else if (method.equals("DELETE")) {
            this.doDelete(req, resp);
        } else if (method.equals("OPTIONS")) {
            this.doOptions(req, resp);
        } else if (method.equals("TRACE")) {
            this.doTrace(req, resp);
        } else {
            String errMsg =
lStrings.getString("http.method_not_implemented");
            Object[] errArgs = new Object[]{method};
            errMsg = MessageFormat.format(errMsg, errArgs);
            resp.sendError(501, errMsg);
        }

    }
```

author大白话：也就是说，HttpServlet的service方法是通过浏览器的请求方式来决定重写哪个do方法的。常用的请求有Post请求和Get请求。当浏览器以哪种方式的请求服务器，服务器就自动执行那个方式的do方法。这个do方法是我们需要我们自己重写的。

一个小案例：

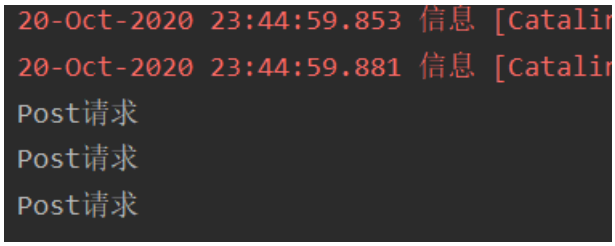
- 当点击提交时会访问demo4这个Servlet实现类，自动调用service方法，然后根据请求方式post，调用执行dopost方法

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>login</title>
</head>
<body>
    <form action="/web_learning/demo4" method="post">
        <input type="email">
        <input type="submit" value="提交">
    </form>
</body>
</html>
```

```
package pers.hhj.day01;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet ("/demo4")
public class ServletDemo4 extends HttpServlet {
    @Override
```

```
13      protected void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
14          System.out.println("Post请求");
15      }
16
17      @Override
18      protected void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
19          System.out.println("GET请求");
20      }
21  }
```

控制台打印：



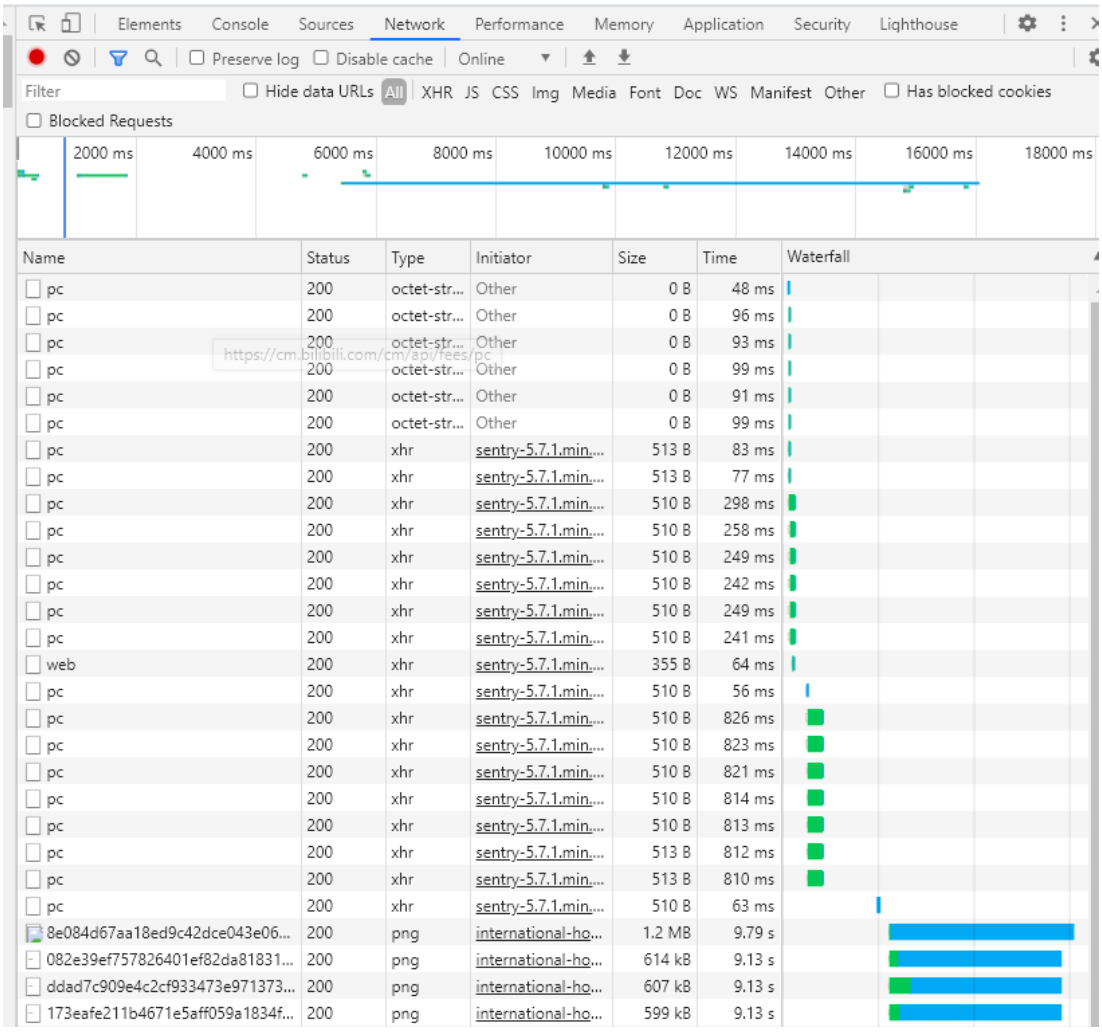
4. Servlet资源路径配置的多种方式

- 一个Servlet可以有多个访问路径：@WebServlet ({"/d1","/dd1","/ddd1"})
- 路径定义有多种规则：
  - /xxx
  - /xxx/xxx /xxx/*
  - *.do

目前学习只用这个方式：/xxx，一个Servlet一个资源路径

5. HTTP协议

> HTTP协议翻译过来是超文本传输协议，是基于TCP/IP协议的**应用层协议**。它不涉及数据包的传输，主要**规定**了客户端和服务器之间发送**数据的格式**，默认端口号是：80。
>
> HTTP协议是基于请求/响应模型的，一次请求对应一次想要，且每次请求直接相互独立。
>
> **author大白话**
>
> 学过计网的应该都懂，协议就是一个规定，信息传输方式的规定。
>
> 比如当前有个客户端有一个登录界面，在你填写完表单点击提交时，客户端就会想浏览器发送一个请求，让服务器通过查询数据库来判断当前登录名和登录密码是否正确。但是你这个请求信息不能随便写，要有一个规定的格式，不然服务器是无法识别的，这个格式就是我们所说的HTTP协议。当前服务器的响应信息的格式也要遵循HTTP协议。这样一来，就可以实现客户端和服务器的通信。
>
> 随便登录一个网站，F12--打开Network抓包工具，可以看到当前客户端向服务器请求的资源



HTTP的历史：1.0---------->1.1

1.0版本每一次请求响应都会建立新的连接，1.1可以复用连接

① 请求信息的数据格式

1. 请求行

   就是：请求方式+请求url+请求协议/版本

   这里是浏览器向我的tomcat服务器请求demo4servlet的请求行：

   POST /web_learning/demo4 HTTP/1.1

   拓展：客户端的七大请求方式✓

   - GET

     请求参数(表单登录名这些)在url后，请求行中

     请求的url长度是有限制的

     不太安全(相对POST)

   - POST

     请求参数在请求体中，url看不见请求参数

     请求的url长度没有限制

     相对安全

2. 请求头

   格式：key : value

   请求demo4的请求头

```
POST /web_learning/demo4 HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Content-Length: 0
Cache-Control: max-age=0
sec-ch-ua: "\\Not\"A;Brand";v="99", "Chromium";v="84", "Google Chrome";v="84"
sec-ch-ua-mobile: ?0
Upgrade-Insecure-Requests: 1
Origin: http://localhost:8080
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/84.0.4147.38 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://localhost:8080/web_learning/login.html
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8
Cookie: JSESSIONID=58FB624F0E86AD497F1D9B14CAF391AD; Idea-abce074e=ccc90c3d-4446-4c5d-8b1b-eb09b704173a; username-localhost-8888="2|1:0|10:1603089754|23:username-localhost-8888|44:NjllODljYTg2Zjg1NGFmN2IzZTMyM2NjYjhmMmJiYjI=|a49b0746c27e20cff150c40a47847a11422878bf7053c533edb26b736b435c86"; _xsrf=2|8f223098|7154414db8860f2a40a6b5aa99e6d23d|1603089754
```

   对服务器来说重要的请求头信息：

   - User-Agent：告诉服务器当前浏览器的版本信息，用被服务器用于解决浏览器的兼容问题
   - Referer：告诉服务器当前请求的url，可用于防盗链、统计

3. 请求空行

   > 空行，分割POST请求的请求头和请求体

4. 请求体

   > 封装POST请求消息的请求参数的
   >
   > GET不用，没有请求体

② 响应信息的数据格式

1. 响应行：协议/版本 响应状态码

   > **响应状态码**：服务器告诉客户端浏览器本次请求和相应的一个状态，响应状态码都是3位数字
   >
   > - 1xx：服务器接收客户端消息但没有接收完成，等待一段时间后发送1xx状态码，询问客户端还有没有数据
   > - 2xx：成功。代表：200

> - 3xx：重定向。代表302，让该请求去找它给的资源路径实现，自己实现不了。还有一个重要的状态码304，重定向到浏览器本地缓存
> - 4xx：客户端错误。代表：404(请求路径没有对应的资源)；405(请求方式没有对应的doXXX方法)
> - 5xx：服务器端错误。代表：500(服务器内部异常)

2. 响应头：头名称：头的值

> 常见的响应头：
>
> - Content-Type：服务器告诉客户端本次响应体数据格式和编码格式（浏览器会用这个格式去解码数据）
> - Content-disposition：服务器告诉客户端以什么格式打开响应体数据
>
> in-line：默认值，当前页面打开
>
> attachment;filename=xxx：以附件形式打开响应体，比如文件下载

3. 响应空行
4. 响应体：真实的传输的数据

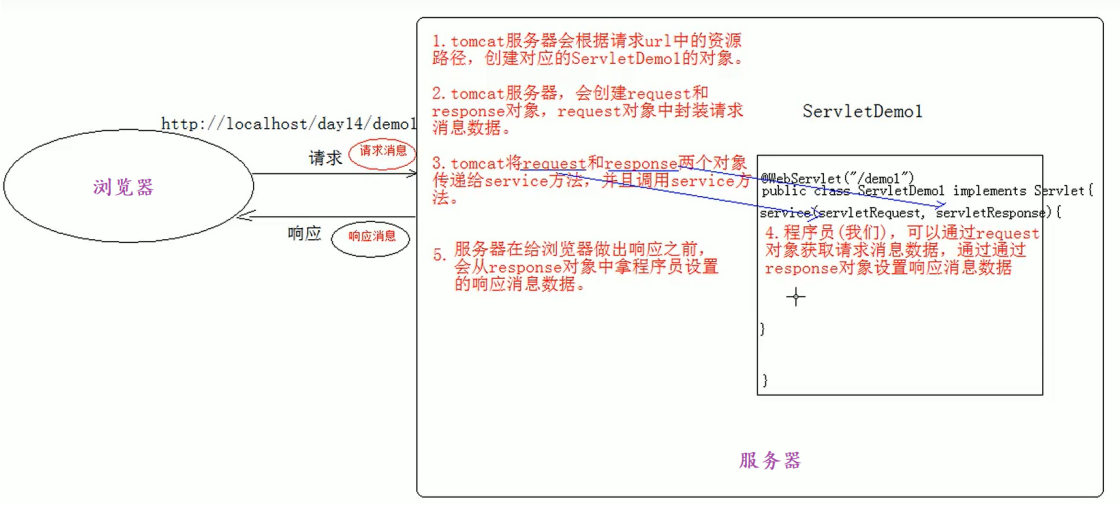
直接看一个案例吧，客户端访问login.html的响应信息

```
HTTP/1.1 200    <!--响应行，http协议等-->
Date: Sat, 24 Oct 2020 08:33:21 GMT
Accept-Ranges: bytes
ETag: W/"418-1603514135335"          <!--中间的是响应头，返回服务器信息，编码啥的-->
Last-Modified: Sat, 24 Oct 2020 04:35:35 GMT
Content-Type: text/html
Content-Length: 418

<!--响应体，就是html文档内容-->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>login界面</title>

</head>
<body>
    <form action="/web_learning/loginServlet" method="POST">
        <input type="text" name="username" placeholder="请输入用户名"><br>
        <input type="text" name="password" placeholder="请输入密码"><br>
        <input type="submit" value="登录">
    </form>

</body>
</html>
```

6. Request和Response对象的原理

> service方法的原型：
>
> ```
> public void service(ServletRequest servletRequest, ServletResponse servletResponse) throws ServletException, IOException {
>
>     }
> ```

再来看看这张图：



authour大白话：

实际开发中，我们不需要创建request和response对象，服务器已经创建好了，我们是去使用他们的。request对象用于获取请求信息，respose对象用来返回响应信息，respose对象是需要我们去配置的，因为响应什么需要我们自己决定。

① request对象详解

- 体系结构

  ServletRequest　　-- 父接口

  　　| 继承

  HttpServletRequest　　--子接口

  　　| 实现

  org.apache.catalina.connector.RequestFacade　　--实现类(tomcat实现)

- 方法

  1. 获取请求消息数据

     - 获取请求行

       login.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>login</title>
</head>
<body>
    <form action="/web_learning/resdemo1" method="post">
        <input name="username">
        <input type="submit" value="提交">
    </form>
</body>
</html>
```

```
// 重温格式: POST /web_learning/demo4 HTTP/1.1
package pers.hhj.day02_Request;

```

```java
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet ("/resdemo1")
public class RequestDemo1 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req,
HttpServletResponse resp) throws ServletException, IOException
{
        // 我们使用的是post请求方式，因此重载下面那个
    }

    @Override
    protected void doPost(HttpServletRequest req,
HttpServletResponse resp) throws ServletException, IOException
{
        // 1.获取请求方式
        System.out.println(req.getMethod());
        // 2.获取虚拟目录---重点
        System.out.println(req.getContextPath());
        // 3.获取Servlet路径
        System.out.println(req.getServletPath());
        // 4.获取post方式请求参数
        System.out.println(req.getQueryString());
        // 5.获取URI(短，没有http://虚拟目录) 以及URL---重点
        System.out.println(req.getRequestURI());
        System.out.println(req.getRequestURL());
        // 6.获取协议和版本
        System.out.println(req.getProtocol());
        // 7.获取IP地址
        System.out.println(req.getRemoteAddr());
    }
}
```

执行结果：

```
POST
/web_learning
/resdemo1
null
/web_learning/resdemo1
http://localhost:8080/web_learning/resdemo1
HTTP/1.1
0:0:0:0:0:0:0:1
```

- 获取请求头

  两个方法

```java
package pers.hhj.day02_Request;

import javax.servlet.ServletException;
```

```java
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.Enumeration;

@WebServlet ("/resdemo2")
public class RequestDemo2 extends HttpServlet {

    @Override
    protected void doPost(HttpServletRequest req,
HttpServletResponse resp) throws ServletException, IOException
{

        // 1.获取请求头数据
        // 获取所有的请求请求头名称，返回Enumeration<String>
        // Enumeration<String>当作一个迭代器即可
        Enumeration<String> headerNames = req.getHeaderNames();
        while(headerNames.hasMoreElements()){
            String name = headerNames.nextElement();
            // 通过请求头的名称key获取它的value
            String value = req.getHeader(name);
            System.out.println(name+"--------"+value);
        }

    }
}
```

运行结果：

```
host--------localhost:8080
connection--------keep-alive
content-length--------21
cache-control--------max-age=0
sec-ch-ua--------"\\Not\"A;Brand";v="99", "Chromium";v="8
sec-ch-ua-mobile--------?0
upgrade-insecure-requests--------1
user-agent--------Mozilla/5.0 (Windows NT 10.0; Win64; x6
origin--------http://localhost:8080
content-type--------application/x-www-form-urlencoded
accept--------text/html,application/xhtml+xml,application
sec-fetch-site--------same-origin
sec-fetch-mode--------navigate
sec-fetch-user--------?1
sec-fetch-dest--------document
referer--------http://localhost:8080/web_learning/login.h
accept-encoding--------gzip, deflate, br
accept-language--------zh-CN,zh;q=0.9,en;q=0.8
cookie--------JSESSIONID=35C0B49E0358B224D684B1773A4E7079
```

- 获取请求体

> 注意：只有POST请求方式才有请求体，请求体中封装了POST请求的请求参数，格式是字符串(map)。如:username:zhangsan

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>注册·</title>
</head>
<body>
    <form action="/web_learning/requestDemo3" method="POST">
        <input name="username" type="text" placeholder="请输入用户名">
        <input name="password" type="text" placeholder="请输入密码">
        <input type="submit" value="提交">
    </form>
</body>
</html>
```

获取步骤

```
package pers.hhj.day02_Request;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.BufferedReader;
import java.io.IOException;

@WebServlet("/requestDemo3")
public class RequestDemo3 extends HttpServlet {
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        // 1.获取流对象
        BufferedReader reader = request.getReader();
        // 上面那个是字符流对象，还有一个获取字节流对象的方法getInputStream
        // 这个方法一般用于获取文件参数
        // 2.读取信息
        String s = null;
        while((s=reader.readLine())!=null){
            System.out.println(s);
        }
    }

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

    }
}

```

运行结果：

```
username=zhangsan&password=111
```

2. 其他方法

- 获取请求参数的通用方式

> 有了这个方法，就不用通过获取请求体的流对象来获取参数了
>
> 无论是post方式还是get方式都一样，也就是说post和get基本没区别了，代码可以一样

1. **String getParameter(String name)**:根据参数名称获取参数值
   username=zs&password=123
2. **String[] getParameterValues(String name)**:根据参数名称获取参数值的数组
   hobby=xx&hobby=game
3. **Enumeration getParameterNames()**:获取所有请求的参数名称
4. **Map<String,String[]> getParameterMap()**:获取所有参数的map集合

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>注册·</title>
</head>
<body>
    <form action="/web_learning/requestDemo4" method="POST">
        <input name="username" type="text" placeholder="请输入用户名">
        <input name="password" type="text" placeholder="请输入密码">
        <input name="sex" type="checkbox" value="男">男
        <input name="sex" type="checkbox" value="女">女
        <input type="submit" value="提交">
    </form>
</body>
</html>
```

```java
package pers.hhj.day02_Request;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.Enumeration;
import java.util.Map;
import java.util.Set;

@WebServlet ("/requestDemo4")
public class RequestDemo4 extends HttpServlet {
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        System.out.println(req.getParameter("username"));

```

```
        String[] sexes = req.getParameterValues("sex");
        for(String s:sexes){
            System.out.println(s);
        }

        Enumeration<String> parameterNames = req.getParameterNames();
        while (parameterNames.hasMoreElements()){

 System.out.println(req.getParameter(parameterNames.nextElement()));
        }

        Map<String, String[]> parameterMap = req.getParameterMap();
        Set<String> strings = parameterMap.keySet();
        for(String s:strings) {
            String[] values = parameterMap.get(s);
            System.out.println(s);
            for(String s1:values){
                System.out.println(s1);
            }
            System.out.println("---------");
        }
    }

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        doPost(req,resp);
    }
}

```

- 请求转发

> 请求转发是一种服务器内部的资源跳转方式，现在还不知道有什么用，大概就是从一个Servlet实现类转到另外一个，利用另外一个的servlet方法解决剩下的问题。

具体实现:

通过request对象的geRequestDispatcher方法t获取请求转发器对象RequestDispatcher，使用这个对象的forward方法进行转发

```
package pers.hhj.day02_Request;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet("/requestDemo5")
public class RequestDemo5 extends HttpServlet {
    @Override

```

```
    protected void doGet(HttpServletRequest req,
HttpServletResponse resp) throws ServletException, IOException
{
        System.out.println("requestDemo5在此");
        // 请求并转发

 req.getRequestDispatcher("/requestDemo6").forward(req,resp);

    }
}
```

运行结果：

```
requestDemo5在此
这里是requestDemo6
```

ps：requestDemo6就不贴了，很简单

请求转发的特点：

1. 浏览器的地址栏路径不会变化，也就是不会显示demo6的资源路径
2. 只能转发到当前服务器的内部资源中，不能搞个baidu.com
3. 转发是**一次请求**，使用多个servlet对象来共同解决问题

author大白话：

> 怎么说呢，请求转法就是把浏览器请求服务器完成的业务逻辑分给多个servlet对象合作完成，使得程序更加高效

- 共享数据

> 接上面的请求转发，多个servlet资源想要共同完成同一个业务逻辑，那就必须实现它们之间的通信，因此共享数据就是用于请求转发的多个资源中共享数据。
>
> 介绍几个概念：
>
> 1. 域对象：一个有作用范围的对象，可以在范围内共享数据
> 2. request域：代表一次请求的域范围
>
> 需要共享的数据是存储在request对象里面的，我们要做的就是往这个对象里面添加数据 or 查找数据 or 删除数据

方法：

```
/**
 *  - void setAttribute(String name,Object obj):键值对方式存储数据
 *  - Object getAttribute(String name):通过key得到value
 *  - void removeAttribute(String name):移除域中键值对数据
 */
```

```
package pers.hhj.day02_Request;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet("/requestDemo5")
```

```java
public class RequestDemo5 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req,
HttpServletResponse resp) throws ServletException, IOException
{
        System.out.println("requestDemo5在此");
        // 存储数据到request域中
        req.setAttribute("name","zhangsan");

        // 请求并转发

 req.getRequestDispatcher("/requestDemo6").forward(req,resp);

    }
}

```

```java
package pers.hhj.day02_Request;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet ("/requestDemo6")
public class RequestDemo6 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req,
HttpServletResponse resp) throws ServletException, IOException
{
        System.out.println("这里是requestDemo6");

        //到request域中取数据
        Object value = req.getAttribute("name");
        System.out.println(value);
    }
}


```

- 获取ServletContext

```java
req.getServletContext()
// 先介绍如何用req获取这个对象
// 重点是这个对象，这里先不说，后面会详细展开这个对象的功能
```

② Response对象详解

> 服务器与客户端的交互就是接收请求消息数据，设置响应消息数据。Request对象用于接收请求消息，这个上面我们已经介绍过了。而Response对象就是用来设置响应消息的。

- 设置响应头：HTTP/1.1 200(就是设置这玩意)

```
setStatus(int sc)
```

- 设置响应行

```
setHeader(String name,String value)
```

- 设置响应体

> 获取字符输出流PrintWriter或者字节输出流ServletOutputStream，通过流的形式设置响应体，传输真正要传输的数据

几个重要案例

1. 重定向(可实现自动跳转)

> 重定向的特点：(区分请求转发)
>
> 地址栏发生变化
>
> 重定向也可访问其他服务器的资源
>
> 重定向是两次请求，不能使用request对象来共享数据

```
package pers.hhj.day03_Response;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet("/responseDemo1")
public class ResponseDemo1 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        System.out.println("1111");
        // 重定向，自动跳转到responseDemo2
        // 通过设置响应行状态码为302，以为响应头的location
        resp.setStatus(302);
        resp.setHeader("location","/web_learning/responseDemo2");

        // 一种简单的重定向方法，原理就是上面的，只不过封装了起来
        resp.sendRedirect("/web_learning/responseDemo2");

    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        doGet(req,resp);
    }
}

```

2. 服务器输出字符数据到浏览器(修改响应体数据)

> 获取字符输出流，输出数据
>
> **注意：字符输出流输出数据到浏览器要解决中文乱码的问题！！**
>
> 我们知道，中文乱码的原因是编码和解码的字符集不同，因此解决乱码的问题只需要设置response的编码和浏览器解析的编码相同即可

```java
package pers.hhj.day03_Response;

import *

@WebServlet("/responseDemo3")
public class ResponseDemo3 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        //获取流对象之前，设置编码
        //resp.setCharacterEncoding("UTF-8");
        //让浏览器以utf-8的编码表解码
        //但是实际上，下面这个方式不仅可以告诉浏览器的解码方式，话可以设置resp对象的编码。因此上面那行可以注释掉了
        //resp.setHeader("content-type","text/html;charset=UTF-8");

        //这样还不是最简单的.....有一个专门的方法用于设置编解码方式。所以上面那个也可以注释了，我们用这个
        resp.setContentType("text/html;charset=UTF-8");

        //获取字符输出流
        PrintWriter pw = resp.getWriter();
        pw.write("你好hello");//这里也可以写入html文档，浏览器会自动解析
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        doGet(req,resp);
    }
}

```

3. 将服务器数据以字节流输出到浏览器

> 步骤和上面差不多，获取流对象。
>
> **同样需要设置编解码相同！！！**

```java
package pers.hhj.day03_Response;

import *

@WebServlet("/responseDemo4")
public class ResponseDemo4 extends HttpServlet {
 @Override
 protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
     resp.setContentType("text/html;charset=UTF-8");
     //获取字杰输出流
     ServletOutputStream sos = resp.getOutputStream();
```

```
        sos.write("你好".getBytes());
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse
resp) throws ServletException, IOException {
        doGet(req,resp);
    }
}

```

4. 验证码案例(不贴了)

> 验证码的本质其实就是一张图片。
>
> 大致思路就是通过java的api动态的创建一张验证码图片，并使用response对象把验证码图片流响应给浏览器。
>
> 即：img.src="servlet资源路径"

5. 文件下载案例

7. ServletContext对象

> 代表整个web项目，可以和程序的服务器进行通信

获取这个对象的方式有两种：

- 通过request对象：req.getServletContext()
- 通过HttpServlet对象：this.getServletContext()

方法：

- 获取文件的MIME类型

> MIME：在互联网通信过程种定义的一种文件数据类型。格式是：大类型/小类型
>
> 比如：
>
> html文件的MIME类型是：text/html
>
> jpg文件的MIME类型是：image/jpeg

```
String getMimeType(String file);
sc.getMimeType("a.jpg")
```

- 域对象，共享数据

> 和request对象请求转发的域对象有点类似，只不过这个的范围更大，里面储存了所有用户所有请求的数据。
>
> 也就是说，一个web项目的ServletContext只有一个，所有用户都访问的是同一个
>
> 而request对象的域是每次请求产生一个，每个都不同

```
setAttribute(String name,Object value)
getAttribute(String name)
removeAttribute(String name)
```

- 获取文件的真实路径，即服务器路径

```
String getRealPath(String path)
path路径怎么写：
    - web目录下的文件路径：/b.txt
    - web-inf目录下的文件路径：/WEB-INF/c.txt
    - src目录下的文件路径：/WEB-INF/classes/a.txt
```

8. Cookie和Session

① Cookie

> 客户端的会话技术，将数据保存到客户端
>
> **author大白话**：Cookie的本质就是存储在浏览器上的一个小的文本文件，用于实现各个请求之间的**通信**。
>
> 那为什么不用request对象的域呢？因为每次请求都会生成一个request对象，每个对象的域都是独立的，不能用于请求的互相通信。
>
> 那为什么不用ServletContext对象的域呢？因为所有的请求都使用同一个Context对象，这样一来服务器就无法区分不同的用户了。

- 快速入门

```
package pers.hhj.day04_Cookie;

@WebServlet("/cookieDemo1")
public class CookieDemo1 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse
resp) throws ServletException, IOException {
        // 新建一个cookie对象
        Cookie cookie = new Cookie("name","zhangsan");
        //发送给浏览器
        resp.addCookie(cookie);

    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse
resp) throws ServletException, IOException {
        doGet(req,resp);
    }
}

```

```
package pers.hhj.day04_Cookie;

@WebServlet("/cookieDemo2")
public class CookieDemo2 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse
resp) throws ServletException, IOException {
        //通过req接受浏览器的cookie
        Cookie[] cookies = req.getCookies();
        for(Cookie a:cookies){
            System.out.println(a.getName()+":"+a.getValue());

```

```
        }

    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse
resp) throws ServletException, IOException {
        doGet(req,resp);
    }
}

```

```
JSESSIONID:CC6A0B1B1493A672374A9228CCEABD53
name:zhangsan
Idea-abce074e:ccc90c3d-4446-4c5d-8b1b-eb09b704173a
username-localhost-8888:2|1:0|10:1603089754|23:username-localhost-8888|44:NjllODljYTg2Zjg1NGFmN2IzZTMyM2NjYjhmM
_xsrf:2|8f223098|7154414db8860f2a40a6b5aa99e6d23d|1603089754
Idea-8f715f3d:d1f63f0c-1da0-44b0-90f1-d17640805c3f
```

- Cookie的实现原理

  > 基于响应头set-cookie和请求头cookie实现
  >
  > 当服务器的响应消息的响应头中包含set-cookie时，浏览器就会自动保存这些信息到本地。当客户端再次请求服务器时，会自动设置请求头cookie，并把本地保存的cookie信息添加到请求头中。这样就实现了请求和请求之间的通信。

- Cookie的细节问题

  1. 能否一次发送多个cookie?

     > 可以，创建多个cookie对象，调用多次addCookie方法即可

  2. cookie在浏览器中保存多长时间?

     > 默认情况下：浏览器关闭时，cookie信息销毁
     >
     > 设置cookie的生米周期，使得可以持久化存储

     ```
     cookie.setMaxAge(int seconds)
     // 传参正数：将cookie数据写道硬盘的文件中。数值代表存活时间，时间到了自动删除
     // 传参负数：就是我们的默认值
     // 传0：删除cookie信息
     ```

  3. cookie的共享问题

     > - 同一个服务器下的不同项目
     >
     >   默认情况下多个项目间不能共享
     >
     >   可通过方法cookie.setPath()实现项目间共享
     >
     >   ```
     >   cookie.setPath("/")
     >   //默认情况下setPath的参数为当前项目的虚拟目录
     >   ```
     >
     > - 不同服务器的项目的cookie共享问题
     >
     >   使用域名共享
     >
     >   ```
     >   cookie.setDomain(String path)
     >   // 设置一级域名相同，多个服务器之间cookie可以共享
     >   ```

4. cookie的用处和特点

- cookie数据存储在浏览器，所以相对来说不安全
- 浏览器对于单个cookie的大小有限制(4kb)，同一个域名下的总cookie也是有限制的(20)

作用：

- 用于存储少量的不太重要的数据
- 在不登陆的情况下完成对客户端信息的识别

5. cookie小案例

> 记住上一次登陆的时间

```
package pers.hhj.day04_Cookie;

import *;
@WebServlet("/cookieDemo3")
/**
 * 记住上次登陆时间的案例
 */
public class CookieDemo3 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req,
HttpServletResponse resp) throws ServletException, IOException {
        resp.setContentType("text/html;charset=utf-8");

        Cookie[] cookies = req.getCookies();
        boolean flag=true;
        if(cookies!=null&& cookies.length!=0){
            for(Cookie cookie:cookies){
                String name = cookie.getName();
                if(name.equals("last_time")){
                    String value = cookie.getValue();
                    resp.getWriter().write("欢迎你再次访问本站，您上次的
访问时间是"+value);
                    Date date=new Date();
                    SimpleDateFormat sdf=new SimpleDateFormat("yyyy
年MM月dd日HH:mm:ss");
                    value=sdf.format(date);
                    cookie.setValue(value);
                    //保存一周
                    cookie.setMaxAge(60*60*24*7);
                    resp.addCookie(cookie);

                    flag=false;
                    break;
                }
            }
        }

        if(cookies==null||cookies.length==0||flag==true){
            Date date=new Date();
            SimpleDateFormat sdf=new SimpleDateFormat("yyyy年MM月dd
日HH:mm:ss");
            String value=sdf.format(date);
            Cookie cookie = new Cookie("last_time",value);
            resp.addCookie(cookie);
            resp.getWriter().write("欢迎首次访问本站");

```

```
        }
    }

    @Override
    protected void doPost(HttpServletRequest req,
HttpServletResponse resp) throws ServletException, IOException {
        doGet(req,resp);
    }
}

```

② Session

> 服务器端的会话技术，在一次会话的多次请求间共享数据，将数据保存在服务器端的对象

- 快速入门

```
package pers.hhj.day05_Session;

import *;

@WebServlet("/sessionDemo1")
public class SessionDemo1 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        // 获取session对象
        HttpSession session = req.getSession();

        //添加信息
        session.setAttribute("name","zhangsan");
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        doGet(req,resp);
    }
}
```

```
package pers.hhj.day05_Session;

import *;

@WebServlet("/sessionDemo2")
public class SessionDemo2 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        //通过req对象获取session对象
        HttpSession session = req.getSession();
        //获取值
        Object value = session.getAttribute("name");
        System.out.println(value);

//      session.removeAttribute("name");
    }
```

```
17
18      @Override
19      protected void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
20          doGet(req,resp);
21      }
22  }
23
```

- 实现原理

> Session的实现是依赖于Cookie的。
>
> 会话开始即浏览器第一次访问服务器资源时，服务器会自动创建一个Session对象，每个Session对象都有一个ID，这个ID就是这个Session区别于其他对象的标志。
>
> 然后响应头set-Cookie添加一条记录为：JSESSIONID= ID，这条信息会被保存到浏览器本地
>
> 当客户端再次请求服务器时，会自动设置请求头cookie，并把本地保存的cookie信息添加到请求头中，其中就包含了JESSIONID的信息，这样服务器就能选择使用内存中的哪个Session对象，实现同一个会话的多个请求间的通信。

- Session的细节问题

1. 服务器不关闭，客户端关闭，获取的session是都一个吗?

> 默认情况下不是，因为不是同一个会话
>
> 可以通过手动配置cookie，设置cookie的生命周期(因为默认配置的cookie的浏览器关闭后销毁)，使得即使关闭浏览器后，再次访问服务器也能访问到同一个session

```
1   // 设置session的生命周期
2       // 获取session对象
3       HttpSession session = req.getSession();
4
5       Cookie cookie = new Cookie("JSESSIONID",
    session.getId());
6       cookie.setMaxAge(60*60);//一小时
7       resp.addCookie(cookie);
8
9       //添加信息
10      session.setAttribute("name","zhangsan");
```

2. 客户端不关闭，服务器关闭后，两次获取的session是同一个吗

> 默认情况下对象不是，因为服务器中的session对象已经被销毁了
>
> 但是可以确保Session对象的数据不丢失
>
> - Session的钝化：在服务器正常关闭之前，将Session对象序列化到硬盘上
> - Session的活化：在服务器启动后，将Session文件转化内存中的Session对象

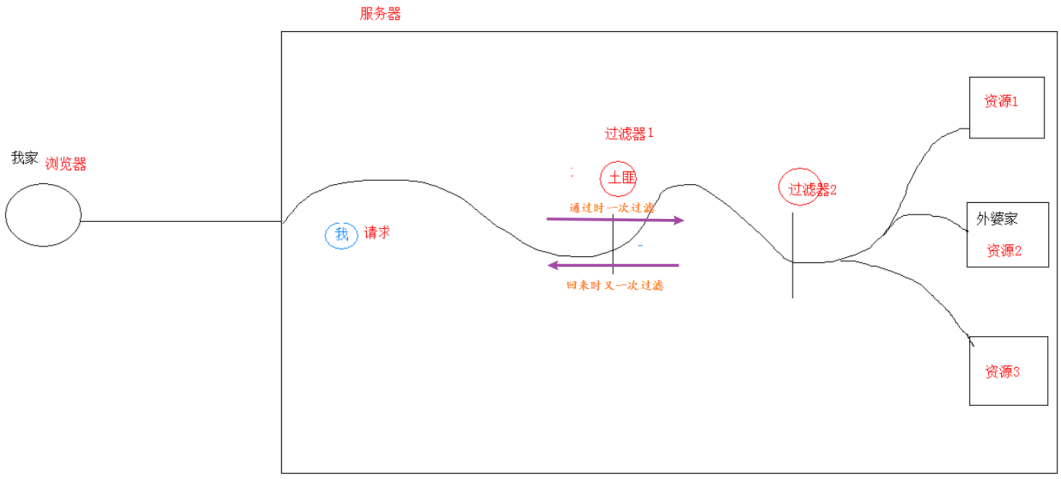
3. Session对象失效时间

> 默认30分钟，可以通过web-xml修改
>
> ...

4. Session的特点

> - session用于存储一次会话的多次请求的数据
> - session可以存储任意类型、任意大小的数据
> - session的数据存储在服务器端，比较安全

9. Filter过滤器

> 先看这幅图：
>
> 
>
>
> 过滤器的理解：
>
> 可以把过滤器想象成一个好的土匪，当浏览器访问服务器资源时，会把请求拦截下来，完成一些通用的操作(如登录验证、同一编码处理，敏感字符过滤)，起到增强resquest的作用。当服务器完成请求将信息响应回服务器时，也要经过过滤，起到增强response的作用。

- 快速入门

```
package pers.hhj.day06_Filter;

import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import java.io.IOException;

//通过WebFilter配置拦截路径,也就是要拦截哪些Servlet
@WebFilter("/*")
public class FilterDemo1 implements Filter {
    public void destroy() {
    }

    public void doFilter(ServletRequest req, ServletResponse resp,
FilterChain chain) throws ServletException, IOException {
        System.out.println("访问服务器资源区执行的代码");
        //放行，让该请求去执行它的Servlet
        chain.doFilter(req, resp);
        System.out.println("响应服务器前执行的代码");

    }

    public void init(FilterConfig config) throws ServletException {

    }
}
```

运行结果：

```
访问服务器资源区执行的代码
响应服务器前执行的代码
```

注意：服务器刚启动访问index.jsp时，Filter会收到两个请求，一个是index.jsp请求，一个时服务器默认的图标请求，因此Filter会被执行两次。

通过这段代码可以看到Filter的执行流程：

执行过滤器　to　执行放行后的资源　to　回来执行过滤器放行代码下的代码

- 过滤器配置

> 过滤器的配置和Servlet一样，有两种方式，一是通过web.xml配置，二是通过注解WebFilter配置
>
> - web.xml配置
>
> ```
> <!--    配置FilterDemo1-->
>     <filter>
>         <filter-name>Filterdemo1</filter-name>
>         <filter-
> class>pers.hhj.day06_Filter.FilterDemo1</filter-class>
>     </filter>
>     <filter-mapping>
>         <filter-name>Filterdemo1</filter-name>
> <!--        拦截路径-->
>         <url-pattern>/*</url-pattern>
>     </filter-mapping>
> ```
>
> - WebFilter配置看快速入门的代码即可，和WebServlet一样
>
> **拦截路径配置：**
>
> 1. /index.jsp　具体资源路径。只有访问index.jsp时过滤器才会被执行
> 2. /user/*　具体目录。该目录下的所有资源都会被拦截
> 3. *.jsp　后缀名拦截
> 4. /*　拦截所有
>
> **拦截方式配置：**
>
> 资源被访问的方式有：直接访问、请求转发、包含访问、错误跳转和异步访问。通过配置WebFilter的dispatcherTypes属性或者web.xml下的标签，可以设置过滤器只对哪种或哪几种访问方式生效
>
> dispatcherTypes有五个属性：
>
> - REQUEST：默认值，直接访问资源
> - FORWARD：请求转发访问资源
> - INCLUDE：包含访问资源
> - ERROR：错误跳转资源
> - ASYNC：异步访问资源
>
> ```
> package *;
>
> @WebFilter(urlPatterns = {"/index.jsp"},dispatcherTypes =
> {DispatcherType.REQUEST,DispatcherType.FORWARD})
> public class FilterDemo2 implements Filter {
>     public void destroy() {
>     }
>
>     public void doFilter(ServletRequest req, ServletResponse resp,
> FilterChain chain) throws ServletException, IOException {
>         chain.doFilter(req, resp);
> ```

```
    }

    public void init(FilterConfig config) throws ServletException {

    }
}
```

- 过滤器的生命周期方法
  - init：在服务器启动会，会创建Filter对象(Servlet是在客户端请求它的资源时才创建，不是在服务器启动时创建，当然也可以通过配置web.xml使其在服务器启动时创建，不过一般不那样做，代价太大)，然后调用init方法，用于加载资源，只执行一次
  - doFilter：每一次请求被拦截资源时被执行，执行多次
  - destroy：在服务器关闭后，Filter对象被销毁。如果服务器是正常关闭，会调用destroy进行销毁，如果非正常，通过其他机制销毁。
- 过滤器链

  > 配置多个过滤器，同时生效，拦截请求
  >
  > 这里我们要考虑的问题有两个，一是配置多个过滤器后的执行顺序，二是如何判断哪个过滤器先执行
  >
  > 1. 若当前有过滤器1和过滤器2同时拦截i请求ndex.jsp资源的请求，它们的执行顺序是怎么样的呢?
  >
  >   > 答案是1221，不难理解。如果有两个土匪前后在路上，回来的时候肯定是先碰到后面的那个土匪。
  >   >
  >    > ```
  >    > 访问服务器资源区执行的代码
  >    > Filter2 come
  >    > Filter2 go
  >    > 响应服务器前执行的代码
  >    > ```
  >
  > 2. 那过滤器1和过滤器2哪个先执行呢?
  >
  >   > 主要看过滤器是如何配置的
  >   >
  >   > - 若使用注解配置，则按照类名字符串的字典顺序比较，小的先执行。如FilterDemo1比FilterDemo2先执行
  >   > - 若使用web.xml配置，则标签谁定义在上面谁先执行。

10. Listener监听器

> 和JS中的监听器机制类似：事件、事件源、监听器和注册监听。
>
> 当事件源发生某事件后，就会触发执行监听器上的代码

服务器中的监听器

- ServletContextListener

  > 监听ServletContext对象的创建和销毁

  两个方法：

  - void contextDestroyed(ServletContextEvent sce)

    > ServletContext对象被销毁之后执行该方法

- void contextInitialized(SevletContextEvent sce)

  > ServletContext对象被创建后执行该方法
  >
  > ServletContext是在服务器被启动时创建的，因此这个方法一般用于加载资源文件

快速入门

```
import *

@WebListener()
public class ListenerDemo1 implements ServletContextListener{

    // Public constructor is required by servlet spec
    public ListenerDemo1() {
    }

    // -------------------------------------------------------
    // ServletContextListener implementation
    // -------------------------------------------------------
    public void contextInitialized(ServletContextEvent sce) {
      /* This method is called when the servlet context is
         initialized(when the Web application is deployed).
         You can initialize servlet context related data here.
      */
        System.out.println("ServletContext对象被创建了");
    }

    public void contextDestroyed(ServletContextEvent sce) {
      /* This method is invoked when the Servlet Context
         (the Web application) is undeployed or
         Application Server shuts down.
      */
        System.out.println("ServletContext对象被销毁了");
    }
}

```

> 服务器中的监听器对象有很多，这里我们直接介绍并实行了ServletContextListener。
>
> 大致步骤是：
>
> 创建一个类实现监听器接口，复写抽象方法，在web.xml配置或者WebListener配置。
>
> web.xml中的配置方法
>
> ```
> <!--    配置监听器-->
>  <listener>
>      <listener-
> class>pers.hhj.day07_Listener.ListenerDemo1</listener-class>
>  </listener>
> ```
>
> WebListener的配置比较简单，因为不用配置url路径，直接加@WebListener即可

一个小案例：

```
package cn.itcast.web.listener;

import javax.servlet.ServletContext;
```

```java
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.annotation.WebListener;
import java.io.FileInputStream;


@WebListener
public class ContextLoaderListener implements ServletContextListener {

    /**
     * 监听ServletContext对象创建的。ServletContext对象服务器启动后自动创建。
     *
     * 在服务器启动后自动调用
     * @param servletContextEvent
     */
    @Override
    public void contextInitialized(ServletContextEvent servletContextEvent) {
        //加载资源文件
        //1.获取ServletContext对象
        ServletContext servletContext = servletContextEvent.getServletContext();

        //2.加载资源文件
        String contextConfigLocation = servletContext.getInitParameter("contextConfigLocation");

        //3.获取真实路径
        String realPath = servletContext.getRealPath(contextConfigLocation);

        //4.加载进内存
        try{
            FileInputStream fis = new FileInputStream(realPath);
            System.out.println(fis);
        }catch (Exception e){
            e.printStackTrace();
        }
        System.out.println("ServletContext对象被创建了。。。");
    }

    /**
     * 在服务器关闭后，ServletContext对象被销毁。当服务器正常关闭后该方法被调用
     * @param servletContextEvent
     */
    @Override
    public void contextDestroyed(ServletContextEvent servletContextEvent) {
        System.out.println("ServletContext对象被销毁了。。。");
    }
}

```