Day10

监听器

概述

监听器就是一个实现特定接口的普通java程序，这个程序专门用于监听另一个java对象的方法调用或属性改变，当被监听对象发生上述事件后，监听器某个方法将立即被执行。

监听原理

1、存在事件源

2、提供监听器

3、为事件源注册监听器

4、操作事件源，产生事件对象，将事件对象传递给监听器，并且执行监听器相应监听方法

Servlet 监听器

在Servlet规范中定义了多种类型的监听器，它们用于监听的事件源分别为

ServletContext

HttpSession

ServletRequest 这三个域对象。

Servlet 的监听器不需要配置，但是需要进行注册。

Servlet 监听器分类

Servlet 三大作用域

1. ServletContext （上下文对象）
    a. 生命周期
        i. 创建：服务器启动时，为每个web项目创建一个上下文对象。
        ii. 销毁：服务器关闭时，或者项目移除时。
    b. 作用范围：
        i. 项目内共享，当前项目下所有程序都可以共享。
2. Request （请求对象）
    a. 生命周期：
        i. 创建：请求开始的时候创建，每个请求都会对应自己的request对象。
        ii. 销毁：请求结束，响应开始的时候。
    b. 作用范围：
        i. 在一次请求中共享，只在当前请求中有效。
3. Session（会话对象）

a. 生命周期：
    i. 创建：在第一次调用request.getSession()方法时，web容器会检查是否已经有对应的session对象存在，如果没有就创建一个session对象。
    ii. 销毁：
        1. 当一段时间内session没有被使用（默认30分钟）时销毁；
        2. 服务器非正常关闭（强行关闭）时销毁；
        3. 调用session.invalidate()手动销毁。
b. 作用范围：
    i. 在一次会话中（多次请求）共享数据。

注意：服务器正常关闭，再启动，Session对象会进行 钝化 和 活化 操作。
同时如果服务器钝化的时间在 Session 默认销毁时间之内，则活化后 Session 还是会存在的，否则 Session 不存在。
如果JavaBean 数据在 Session 钝化时，没有实现 Serializable ，则当 Session 活化时，会消失。

范围大小

ServletContext > Session > Request

Servlet中的8大监听器

8 种 Listener 如下表所示：

Listener接口	Event类
ServletContextListener	ServletContextEvent
ServletContextAttributeListener	ServletContextAttributeEvent
HttpSessionListener	HttpSessionEvent
HttpSessionActivationListener	HttpSessionEvent
HttpSessionAttributeListener	HttpSessionBindingEvent
HttpSessionBindingListener	HttpSessionBindingEvent
ServletRequestListener	ServletRequestEvent
ServletRequestAttributeListener	ServletRequestAttributeEvent

八种 Linstener 可以分为三类：

- 监听Context、Request、Session对象的创建和销毁，需要在 web.xml 中注册
  - ServletContextListener
  - ServletRequestListener
  - HttpSessionListener
- 监听Context、Request、Session对象属性的变化，需要在 web.xml 中注册
  - ServletContextAttributeListener
  - ServletRequestAttributeListener
  - HttpSessionAttributeListener
- 监听 Session 内部的对象，不需要再 web.xml 中注册
  - HttpSessionActivationListener
  - HttpSessionBindingListener

使用方式

- 创建一个类，实现 Listener 接口
- 在 web.xml 文件中配置
  - 使用 <listener> 标签和 <listener-class> 标签
  - <listener> 一般配置在 <servlet> 标签的前面
- 或者使用注解

Java 复制代码

```java
package com.rushuni.listener;

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.annotation.WebListener;

/**
 * @author rushuni
 * @date 2021年07月22日 5:23 下午
 */
@WebListener
public class DemoListener implements ServletContextListener {
    @Override
    public void contextInitialized(ServletContextEvent sce) {
        System.out.println("web容器启动, context对象被创建...");
        System.out.println(sce.getServletContext().toString());
    }

    @Override
    public void contextDestroyed(ServletContextEvent sce) {
        System.out.println("web容器关闭, context对象被销毁...");
    }
}
```

xml则如下：

XML 复制代码

```xml
<listener>
  <listener-class>com.rushuni.listener.DemoListener</listener-class>
</listener>
```

监听器细节

监听 Context、Request、Session 对象的创建和销毁

监听 Context、Request、Session 对象的创建和销毁，需要配置如下三个监听器：

- ServletContextListener
- ServletRequestListener
- HttpSessionListener

三个监听器的作用和触发时机

ServletContextListener

作用

监听 context 的创建和销毁，context 代表当前的Web应用程序。

该 Listener 可用于启动时获取 web.xml 里面配置的初始化参数。

触发时机

服务器启动或者热部署 war 包时触发 contextInitialized(ServletContextEvent sce) 方法。

服务器关闭时或者只关闭该 web 应用时会触发 contextDestroyed(ServletContextEvent sce) 方法。

ServletRequestListener

作用

监听 request 的创建和销毁。

触发时机

用户每次请求 request 都会触发 requestInitialized(ServletRequestEvent sre) 方法。

request 处理完毕自动销毁前触发 requestDestroyed(ServletRequestEvent sre) 方法。

如果一个 HTML 页面包含多个图片，则请求一次 HTML 页面可能会触发多次 request 事件。

HttpSessionListener

作用

监听 Session 的创建于销毁。

该 Listener 可用于收集在线者信息。

触发时机

创建Session时触发sessionCreated(HttpSessionEvent se)方法。

超时或者执行session.invalidate()方法时执行sessionDestroyed(HttpSessionEvent se)方法。

实例

Java 复制代码

```java
package com.rushuni.listener;

import javax.servlet.*;
import javax.servlet.annotation.WebListener;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;

/**
 * @author rushuni
 * @date 2021年07月22日 5:23 下午
 */
@WebListener
public class DemoListener02 implements ServletContextListener,
ServletRequestListener, HttpSessionListener {

    /**
     * 加载 context
      * @param sce
     */
    @Override
    public void contextInitialized(ServletContextEvent sce) {
        System.out.println("web容器启动, context对象被创建...");
        // 可以获取 context对象
        ServletContext context = sce.getServletContext();
        System.out.println("即将启动 " + context.getContextPath());
    }

    /**
     * 卸载 context
     * @param sce
     */
    @Override
    public void contextDestroyed(ServletContextEvent sce) {
        System.out.println("web容器关闭, context对象被销毁...");
        // 可以获取 context对象
        ServletContext context = sce.getServletContext();
        System.out.println("即将关闭 " + context.getContextPath());
    }

    /**
     * 创建 request
      * @param sre
     */
    @Override
    public void requestInitialized(ServletRequestEvent sre) {
        // 可以获取request对象
```

```
48          HttpServletRequest request = (HttpServletRequest)
    sre.getServletRequest();
49          String uri = request.getRequestURI();
50          uri = request.getQueryString() == null ? uri : (uri + "?" +
    request.getQueryString());
51          System.out.println("请求的uri为: " + uri);
52          // 可以向request对象中放入这次请求中共享的数据
53          request.setAttribute("startTime", System.currentTimeMillis());
54      }
55
56      /**
57       * 销毁 request
58       * @param sre
59       */
60      @Override
61      public void requestDestroyed(ServletRequestEvent sre) {
62          // 可以获取request对象, 获取request对象中的共享数据
63          HttpServletRequest request = (HttpServletRequest)
    sre.getServletRequest();
64          long costTime = System.currentTimeMillis() - (Long)
    request.getAttribute("startTime");
65          System.out.println("本次请求耗时: " + costTime + " ms");
66      }
67
68      /**
69       * 创建 session
70        * @param se
71       */
72      @Override
73      public void sessionCreated(HttpSessionEvent se) {
74          // 可以获取session对象
75          HttpSession session = se.getSession();
76          System.out.println("新创建一个session, id = " + session.getId());
77          // 向session中添加属性
78          session.setAttribute("username", "rushuni");
79          session.setAttribute("password", "123");
80          // 修改session中的属性
81          session.setAttribute("password", "321");
82          // 删除session中的属性
83          session.removeAttribute("username");
84      }
85
86      /**
87       * 销毁 session
88       * @param se
89       */
90      @Override
91      public void sessionDestroyed(HttpSessionEvent se) {
92          // 可以获取session对象
93          HttpSession session = se.getSession();
```

```
94          System.out.println("销毁一个session, id = " + session.getId());
95      }
96
97  }
```

监听Context、Request、Session对象属性的变化

监听 Context、Request、Session 对象属性的变化，需要配置如下监听器：

- ServletContextAttributeListener
- ServletRequestAttributeListener
- HttpSessionAttributeListener

三个监听器的触发时机

当向被监听对象中添加、更新、移除属性时，会分别执行如下方法

- attributeAdded()
- attributeReplaced()
- attributeRemoved()

实例

Java 复制代码

```java
package com.rushuni.listener;

import javax.servlet.*;
import javax.servlet.annotation.WebListener;
import javax.servlet.http.*;

/**
 * @author rushuni
 * @date 2021年07月22日 5:23 下午
 */
@WebListener
public class DemoListener03 implements HttpSessionAttributeListener {

    /**
     * 向session中添加属性时触发
     * @param event
     */
    @Override
    public void attributeAdded(HttpSessionBindingEvent event) {
        HttpSession session = event.getSession();
        String key = event.getName();
        Object value = event.getValue();
        System.out.println("向id = " + session.getId() + " 的session中添加的属性为 <" + key + "," + value + ">");
    }

    /**
     * 从session中移除属性时触发
     * @param event
     */
    @Override
    public void attributeRemoved(HttpSessionBindingEvent event) {
        HttpSession session = event.getSession();
        String key = event.getName();
        Object value = event.getValue();
        System.out.println("从id = " + session.getId() + " 的session中移除的属性为 <" + key + "," + value + ">");
    }

    /**
     * 修改session中属性时触发
      * @param event
     */
    @Override
    public void attributeReplaced(HttpSessionBindingEvent event) {
        HttpSession session = event.getSession();
        String key = event.getName();
        System.out.println("修改id = " + session.getId() + " 的session的属性为
```

```
    key = " + key);
47      }
48
49  }
```

监听Session内部的对象

监听 Session 内部的对象，不需要配置监听器。

保存在 Session 域中的对象可以有多种状态：

- 绑定到 Session 中，session.setAttribute("bean",Object)。
- 从 Session 域中解除绑定，session.removeAttribute("bean")。
- 随 Session 对象持久化到一个存储设备中。
- 随 Session 对象从一个存储设备中恢复。

Servlet 规范中定义了两个特殊的监听器接口 HttpSessionBindingListener 和 HttpSessionActivationListener。

这两个监听器接口帮助 JavaBean 对象了解自己在 Session 域中的状态，实现这两个特殊监听器接口的对象类不需要在 web.xml 文件中进行监听器注册。

HttpSessionBindingListener

- 实现了 HttpSessionBindingListener 接口的 JavaBean对象可以感知自己被绑定到 Session 中和从 Session 中删除的事件
- 当对象被绑定到 HttpSession 对象中时，web服务器调用该对象的 valueBound(HttpSessionBindingEvent event) 方法
- 当对象从 HttpSession 对象中解除绑定时，web服务器调用该对象的 valueUnbound(HttpSessionBindingEvent event) 方法。

HttpSessionActivationListener

- 实现了 HttpSessionActivationListener 接口的 JavaBean对象可以感知自己被活化(反序列化)和钝化(序列化)的事件。
- 当绑定到 HttpSession 对象中的 JavaBean 对象将要随 HttpSession 对象被钝化(序列化)之前，web 服务器调用该 JavaBean 对象的 sessionWillPassivate(HttpSessionEvent event) 方法。这样JavaBean对象就可以知道自己将要和 HttpSession 对象一起被序列化(钝化)到硬盘中。
- 当绑定到 HttpSession 对象中的 JavaBean 对象将要随 HttpSession 对象被活化(反序列化)之后，

web服务器调用该 JavaBean 对象的 sessionDidActive(HttpSessionEvent event) 方法。这样 JavaBean 对象就可以知道自己将要和 HttpSession 对象一起被反序列化(活化)回到内存中。

实例

这类监听器也被称为感知型监听器，使用时无需配置，只需要根据实际需求编写代码。

实现 HttpSessionBindingListener 接口

Java 复制代码

```java
package com.rushuni.listener;

import javax.servlet.http.HttpSessionBindingEvent;
import javax.servlet.http.HttpSessionBindingListener;

/**
 * @author rushuni
 * @date 2021年07月22日 5:23 下午
 */
public class JavaBeanListener01 implements HttpSessionBindingListener {

    private String name;

    public JavaBeanListener01(String name) {
        this.name = name;
    }

    /**
     * JavaBean 被加到 session 中会被触发
      * @param event
     */
    @Override
    public void valueBound(HttpSessionBindingEvent event) {
        System.out.println(name + "被加到session中了...");
    }

    /**
     * 从 session 中移除 JavaBean 时会被触发
      * @param event
     */
    @Override
    public void valueUnbound(HttpSessionBindingEvent event) {
        System.out.println(name + "从session中被移除了...");
    }

}
```

上述的 JavaBeanListener01 这个 JavaBean 实现了 HttpSessionBindingListener 接口，那么这个 JavaBean 对象可以感知自己被绑定到 Session 中和从 Session 中删除的这两个操作。

实现 HttpSessionActivationListener 接口

Java 复制代码

```java
package com.rushuni.listener;

import javax.servlet.http.HttpSessionActivationListener;
import javax.servlet.http.HttpSessionEvent;
import java.io.Serializable;

/**
 * @author rushuni
 * @date 2021年07月22日 5:23 下午
 */
//@WebListener
public class JavaBeanListener02 implements HttpSessionActivationListener, Serializable {

    private static final long serialVersionUID = 1L;

    private String name;

    public JavaBeanListener02(String name) {
        this.name = name;
    }

    /**
     * 钝化前触发
     * @param se
     */
    @Override
    public void sessionWillPassivate(HttpSessionEvent se) {
        System.out.println(name + "将要和session一起被序列化(钝化)到硬盘, session id = "+se.getSession().getId());
    }

    /**
     * 活化后触发
      * @param se
     */
    @Override
    public void sessionDidActivate(HttpSessionEvent se) {
        System.out.println(name + "和session一起从硬盘反序列化(活化)回到内存了, session id = "+se.getSession().getId());
    }

}
```

监听器实战

需求

实现在线人数的监控。

分析

当有新用户时，会创建新的会话，所以根据需求选择 HttpSessionListener。

步骤

1. 创建一个监听器，实现 HttpSessionListener。
2. 通过 @WebListener 注解配置该监听器。

参考代码

Java 复制代码

```java
package com.rushuni.listener;

import javax.servlet.annotation.WebListener;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;

/**
 * @author rushuni
 * @date 2021年07月22日 7:36 下午
 */
@WebListener
public class OnlineListener implements HttpSessionListener {
    /**
     * 默认在线人数
     */
    private Integer onlineNumber = 0;

    /**
     * 当有新的session对象被创建，则在线人数+1；
     */
    @Override
    public void sessionCreated(HttpSessionEvent se) {
        // 人数+1
        onlineNumber++;
        // 将人数存到 session 作用域中
//          se.getSession().setAttribute("onlineNumber", onlineNumber);
        // 将人数存到 application 作用域中
        se.getSession().getServletContext().setAttribute("onlineNumber",
onlineNumber);
    }

    /**
     * 有session对象被销毁，在线人数-1；
     */
    @Override
    public void sessionDestroyed(HttpSessionEvent se) {
        // 人数-1
        onlineNumber--;
        // 将人数存到session作用域中
         se.getSession().setAttribute("onlineNumber", onlineNumber);
        // 将人数存到application作用域中
//          se.getSession().getServletContext().setAttribute("onlineNumber",
onlineNumber);
    }
}
```

创建 Servlet

Java 复制代码

```java
package com.rushuni.servlet;

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

/**
 * @author rushuni
 * @date 2021年07月22日 8:15 下午
 */
@WebServlet("/onlineServlet")
public class OnlineServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        // 得到参数
        String key = request.getParameter("key");
        // 判断是否为空（不为空，且值为logout则为退出操作）
        if (key != null && "logout".equals(key)) {
            // 传递了参数，表示要做用户退出操作
            request.getSession().invalidate();
            return;
        }
        // 创建 session 对象
        HttpSession session = request.getSession();
        // 获取 session 作用域中的在线人数
        Integer onlineNumber = (Integer)session.getServletContext().getAttribute("onlineNumber");
        // 输出
        response.setContentType("text/html;charset=UTF-8");
        response.getWriter().write("<h2>在线人数：" + onlineNumber + "</h2><h4><a href='/onlineServlet'>登录</a><h4 > ");
    }

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

课后作业

- 熟悉八大监听器，每个案例代码都完成。
- 完成实站案例，统计在线人数。

Spring Web MVC

概述

Spring Web MVC 是最初建立在 Servlet API 之上的 Web 框架，从一开始就包含在 Spring Framework 中。

正式名称“Spring Web MVC”来自其源模块的名称 (spring-webmvc)。

但我们通常称为“Spring MVC”。

Spring MVC 与许多其他 Web 框架一样，是围绕前端控制器模式（front controller pattern）设计的，它的central Servlet 的叫做 DispatcherServlet。

DispatcherServlet 可以看做是一个中央调度器，它负责接收所有的请求，但是实际的工作会交给可配置的代理模块去做。

DispatchServlet 其实就是一个 Servlet，所以它也需要通过 Java 代码或者 web.xml 进行 Servlet 的注册和配置。

接着 DispatchServlet 会通过 Spring 的配置来发现需要处理的请求映射，视图解析，异常处理等等工作。

Spring MVC 是基于 Spring 的一个框架， 作为 Spring 的一个模块， 它主要负责 Controller 这一层，可以理解为 Servlet 的一个升级。

> Java Web 开发底层技术基本都是 Servlet，所以基本上相关的框架都是在 Servlet 的基础上面加入一些功能，让我们的 Web 项目开发起来更加方便而已。

Spring MVC 开发流程分析

Spring MVC 的开发流程可以分为两个部分：

1. 服务器启动部分
    a. 加载 web.xml 注册的 DispatcherServlet。
    b. 读取 spring-mvc.xml 中的配置，加载其中所有被 Spring 容器管理的 Bean。
    c. 读取 Bean 中方法上方标注了 **@RequestMapping** 的内容。
2. 处理请求部分
    a. DispatcherServlet 配置拦截所有请求 /。
    b. 使用请求路径与所有加载的 @RequestMapping 的内容进行比对。
    c. 执行对应的方法。
    d. 根据方法的返回值在 webapp 目录中查找对应的页面并展示。

基础案例

案例需求

1. 用户提交一个请求。
2. 服务端处理器在接收到这个请求后，给出一条欢迎信息，在响应页面中显示该信息。

注意不要使用 Tomcat 10。

步骤

创建 Maven 项目

XML 复制代码

```xml
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.rushuni</groupId>
  <artifactId>springmvc-demo-01</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging>

  <name>springmvc-demo-01 Maven Webapp</name>
  <!-- FIXME change it to the project's website -->
  <url>http://www.example.com</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>5.3.9</version>
    </dependency>
  </dependencies>

</project>
```

注册 DispatchServlet

XML 复制代码

```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">

  <listener>
    <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>

  <!-- 定义 springmvc 读取的配置文件的位置 -->
  <context-param>
    <!-- springmvc 的配置文件的位置的属性 -->
    <param-name>contextConfigLocation</param-name>
    <!--指定自定义文件的位置-->
    <param-value>/WEB-INF/app-context.xml</param-value>
  </context-param>

  <servlet>
    <servlet-name>app</servlet-name>
    <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value></param-value>
    </init-param>
    <!-- tomcat启动后就创建 Servlet 对象 -->
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>app</servlet-name>
    <url-pattern>/app/*</url-pattern>
  </servlet-mapping>

</web-app>
```

DispatchServlet 说明

这里通过 web.xml 注册 springmvc 的核心对象 DispatcherServlet，并且需要在 tomcat 服务器启动后就创建DispatcherServlet 对象的实例。

为什么要指明启动立即创建 DispatcherServlet 对象的实例?

因为 DispatcherServlet 在的创建过程中， 会同时创建 springmvc 容器对象，并读取 springmvc 的配置文件，把这个配置文件中的对象都创建好，当用户发起请求时就可以直接使用对象了。

load-on-startup:

- 表示 tomcat 启动后创建对象的顺序。
- 它的值是整数，数值越小，tomcat创建对象的时间越早。
- 数值为大于等于 0 的整数，1 表示tomcat启动后就创建 Servlet 对象

此时启动 tomcat 会报错，因为还没有创建 spring mvc 配置文件。

配置 spring mvc 配置文件

XML 复制代码

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        https://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <mvc:annotation-driven/>

</beans>
```

创建 Controller

Java 复制代码

```java
package com.rushuni.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

/**
 * @author rushuni
 * @date 2021年07月23日 1:59 下午
 */
@Controller
public class Demo01Controller {

    @GetMapping("/hello")
    public String sayHello(Model model) {
        model.addAttribute("message", "Hello World!");
        return "hello.jsp";
    }
    @RequestMapping(value = "/hello2")
    public ModelAndView sayHello(){
        // 开始处理请求
        ModelAndView modelAndView  = new ModelAndView();
        // 添加数据，框架在请求的最后把数据放入到 request 作用域。
        modelAndView.addObject("message","Hello Spring MVC!");
        //指定视图，
        //框架对视图执行了forward操作，
request.getRequestDispather("/hello.jsp).forward(...)
        modelAndView.setViewName("/hello.jsp");
        // 返回结果
        return modelAndView;
    }
}
```

ModelAndView 是 Spring MVC 框架中模型和视图的持有者，方便我们同时设置需要返回的数据和跳转的页面。

课堂作业

- 熟悉 Spring MVC 的概念，完成基础案例需求。

基础配置

静态资源加载

```xml
<!--放行指定类型静态资源配置方式-->
<mvc:resources mapping="/img/**" location="/img/"/>
<mvc:resources mapping="/js/**" location="/js/"/>
<mvc:resources mapping="/css/**" location="/css/"/>

<!--SpringMVC提供的通用资源放行方式-->
<mvc:default-servlet-handler/>
```

中文乱码处理

SpringMVC提供专用的中文字符过滤器，用于处理乱码问题。

配置在 web.xml 里面

```xml
<filter>
  <filter-name>CharacterEncodingFilter</filter-name>
  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>UTF-8</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>CharacterEncodingFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

请求参数

普通类型参数传参

参数名与处理器方法形参名保持一致

```java
@RequestMapping("/requestParam1")
public String requestParam1(String name ,String age){
    System.out.println("name="+name+",age="+age);
    return "page.jsp";
}
// 访问URL： http://localhost/requestParam1?name=rushuni&age=10
```

@RequestParam 的是一个用于形参的注解，可以绑定请求参数与对应处理方法形参间的关系

```java
@RequestMapping("/requestParam2")
public String requestParam2(@RequestParam(
                            name = "userName",
                            required = true,
                            defaultValue = "rushuni") String name){
    System.out.println("name="+name);
    return "page.jsp";
}
```

POJO类型参数传参

当POJO中使用简单类型属性时，参数名称要和 POJO 类属性名保持一致。

```java
@RequestMapping("/requestParam3")
public String requestParam3(User user){
    System.out.println("name="+user.getName());
    return "page.jsp";
}
访问URL： http://localhost/requestParam3?name=rushuni&age=10

public class User {
    private String name;
    private Integer age;
    // ...
}
```

参数冲突

当POJO类型属性与其他形参出现同名问题时，将被同时赋值。

```java
访问URL： http://localhost/requestParam3?name=rushuni&age=10
@RequestMapping("/requestParam4")
public String requestParam4(User user,String age){
    System.out.println("user.age="+user.getAge()+",age="+age);
    return "page.jsp";
}
```

建议使用@RequestParam注解进行区分

参数冲突

□

访问URL： http://localhost/requestParam4?name=itheima&**age**=14

```java

```

复杂POJO类型参数

当POJO中出现对象属性时，参数名称与对象层次结构名称保持一致。

```java
@RequestMapping("/requestParam5")
public String requestParam5(User user){
    System.out.println("user.address="+user.getAddress().getProvince());
    return "page.jsp";
}
// 访问URL： http://localhost/requestParam5?address.province=beijing
public class User {
    private String name;
    private Integer age;
    private Address address;
  // ...
}
public class Address {
    private String province;
    private String city;
    private String address;
    // ...
}
```

当POJO中出现List

当POJO中出现List，保存对象数据，参数名称与对象层次结构名称保持一致，使用数组格式描述集合中对象的位置。

```java
@RequestMapping("/requestParam7")
public String requestParam7(User user){
    System.out.println("user.addresses="+user.getAddress());
    return "page.jsp";
}
访问URL： http://localhost/requestParam7?
addresses[0].province=bj&addresses[1].province=tj
public class User {
    private String name;
    private Integer age;
    private List<Address> addresses;
}

public class Address {
    private String province;
    private String city;
    private String address;
}
```

当POJO中出现Map

当POJO中出现Map，保存对象数据，参数名称与对象层次结构名称保持一致，使用映射格式描述集合中对象的位置。

Java 复制代码

```java
//POJO中Map对象保存POJO的对象属性赋值，使用[key]的格式指定为Map中的对象属性赋值
//http://localhost/requestParam8?addressMap['job'].city=beijing&addressMap['home'].province=henan
@RequestMapping("/requestParam8")
public String requestParam8(User user){
    System.out.println(user.getAddressMap());
    return "page.jsp";
}
public class User {
    private String name;
    private Integer age;
    private Map<String,Address> addressMap;
}
public class Address {
    private String province;
    private String city;
    private String address;
}
```

数组与集合类型参数传参

请求参数名与处理器方法形参名保持一致，且请求参数数量＞ 1个

Java 复制代码

```java
//方法传递普通类型的数组参数，URL地址中使用同名变量为数组赋值
//http://localhost/requestParam9?nick=zhangsan&nick=lisi
@RequestMapping("/requestParam9")
public String requestParam9(String[] nick){
    System.out.println(nick[0]+","+nick[1]);
    return "page.jsp";
}
```

集合类型参数

```java
//方法传递保存普通类型的List集合时，无法直接为其赋值，需要使用@RequestParam参数对参数名称进行转换
//http://localhost/requestParam10?nick=Jockme&nick=zahc
@RequestMapping("/requestParam10")
public String requestParam10(@RequestParam("nick") List<String> nick){
    System.out.println(nick);
    return "page.jsp";
}
```

注意： SpringMVC默认将List作为对象处理，赋值前先创建对象，然后将nick作为对象的属性进行处理。

由于 List 是接口，无法创建对象，报无法找到构造方法异常；

修复类型为可创建对象的ArrayList类型后，对象可以创建，但没有nick属性，因此数据为空。此时需要告知SpringMVC的处理器nick是一组数据，而不是一个单一数据。

通过@RequestParam注解，将数量大于1个names参数打包成参数数组后， SpringMVC才能识别该数据格式，并判定形参类型是否为数组或集合，并按数组或集合对象的形式操作数据。

类型转换器

SpringMVC 对接收的数据进行自动类型转换，该工作通过 Converter 接口实现

标量转换器

```
StringToBooleanConverter String→Boolean
ObjectToStringConverter Object→String
StringToNumberConverterFactory String→Number（ Integer、 Long等）
NumberToNumberConverterFactory Number子类型之间(Integer、 Long、 Double等)
StringToCharacterConverter String→java.lang.Character
NumberToCharacterConverter Number子类型(Integer、 Long、 Double等)→java.lang.Character
CharacterToNumberFactory java.lang.Character→Number子类型(Integer、 Long、 Double等)
StringToEnumConverterFactory String→enum类型
EnumToStringConverter enum类型→String
StringToLocaleConverter String→java.util.Local
PropertiesToStringConverter java.util.Properties→String
StringToPropertiesConverter String→java.util.Properties
```

集合、数组相关转换器

```Plain Text
ArrayToCollectionConverter 数组→集合（ List、 Set）
CollectionToArrayConverter 集合（ List、 Set） →数组
ArrayToArrayConverter 数组间
CollectionToCollectionConverter 集合间（ List、 Set）
MapToMapConverter Map间
ArrayToStringConverter 数组→String类型
StringToArrayConverter String→数组， trim后使用“,”split
ArrayToObjectConverter 数组→Object
ObjectToArrayConverter Object→单元素数组
CollectionToStringConverter 集合（ List、 Set） →String
StringToCollectionConverter String→集合（ List、 Set）， trim后使用“,”split
CollectionToObjectConverter 集合→Object
ObjectToCollectionConverter Object→单元素集合
```

默认转换器

```Java
ObjectToObjectConverter Object间
IdToEntityConverter Id→Entity
FallbackObjectToStringConverter Object→String
```

SpringMVC对接收的数据进行自动类型转换，该工作通过Converter接口实现。

日期类型格式转换

声明自定义的转换格式并覆盖系统转换格式

```xml
<!--5.启用自定义Converter-->
<mvc:annotation-driven conversion-service="conversionService"/>
<!--1.设定格式类型Converter，注册为Bean，受SpringMVC管理-->
<bean id="conversionService"
class="org.springframework.format.support.FormattingConversionServiceFactoryBean">
  <!--2.自定义Converter格式类型设定，该设定使用的是同类型覆盖的思想-->
  <property name="formatters">
    <!--3.使用set保障相同类型的转换器仅保留一个，避免冲突-->
    <set>
      <!--4.设置具体的格式类型-->
      <bean class="org.springframework.format.datetime.DateFormatter">
        <!--5.类型规则-->
        <property name="pattern" value="yyyy-MM-dd"/>
      </bean>
    </set>
  </property>
</bean>
```

日期类型格式转换（简化版）

```java
public String requestParam12(@DateTimeFormat(pattern = "yyyy-MM-dd") Date date){
    System.out.println("date="+date);
    return "page.jsp";
}

@DateTimeFormat(pattern = "yyyy-MM-dd")
private Date birthday;
```

注意：需要依赖注解驱动支持

```xml
<mvc:annotation-driven />
```

自定义类型转换器

自定义类型转换器，实现Converter接口，并制定转换前与转换后的类型

XML 复制代码

```xml
<!--1.将自定义Converter注册为Bean，受SpringMVC管理-->
<bean id="myDateConverter" class="com.rushuni.converter.MyDateConverter"/>
<!--2.设定自定义Converter服务bean-->
<bean id="conversionService"

class="org.springframework.context.support.ConversionServiceFactoryBean">
  <!--3.注入所有的自定义Converter，该设定使用的是同类型覆盖的思想-->
  <property name="converters">
    <!--4.set保障同类型转换器仅保留一个，去重规则以Converter<S,T>的泛型为准-->
    <set>
      <!--5.具体的类型转换器-->
      <ref bean="myDateConverter"/>
    </set>
  </property>
</bean>
```

Java 复制代码

```java
// 自定义类型转换器，实现Converter接口，接口中指定的泛型即为最终作用的条件
// 本例中的泛型填写的是String，Date，最终出现字符串转日期时，该类型转换器生效
public class MyDateConverter implements Converter<String, Date> {
    // 重写接口的抽象方法，参数由泛型决定
    public Date convert(String source) {
        DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
        Date date = null;
        // 类型转换器无法预计使用过程中出现的异常
        // 因此必须在类型转换器内部捕获，不允许抛出，框架无法预计此类异常如何处理
        try {
            date = df.parse(source);
        } catch (ParseException e) {
            e.printStackTrace();
        }
        return date;
    }
}
```

通过注册自定义转换器，将该功能加入到SpringMVC的转换服务ConverterService中

XML 复制代码

```xml
  <!--开启注解驱动，加载自定义格式化转换器对应的类型转换服务-->
  <mvc:annotation-driven conversion-service="conversionService"/>
```

请求映射 @RequestMapping

处理器类定义上方，为当前处理器中所有方法设定公共的访问路径前缀。
处理器类中的方法定义上方，绑定请求地址与对应处理方法间的关系。
/user/requestURL2

Java 复制代码

```java
@Controller
@RequestMapping("/user")
public class UserController {
    @RequestMapping("/requestURL2")
    public String requestURL2() {
        return "page.jsp";
    }
}
```

常用属性

Java 复制代码

```java
@RequestMapping(
    value="/requestURL3", //设定请求路径，与path属性、 value属性相同
    method = RequestMethod.GET, //设定请求方式
    params = "name", //设定请求参数条件
    headers = "content-type=text/*", //设定请求消息头条件
    consumes = "text/*", //用于指定可以接收的请求正文类型（MIME类型）
    produces = "text/*" //用于指定可以生成的响应正文类型（MIME类型）
)
public String requestURL3() {
    return "/page.jsp";
}
```

响应

页面跳转

转发（默认）

```java
@RequestMapping("/showPage1")
public String showPage1() {
    System.out.println("user mvc controller is running ...");
    return "forward:page.jsp";
}
```

重定向

地址会改变

```java
@RequestMapping("/showPage2")
public String showPage2() {
  System.out.println("user mvc controller is running ...");
  return "redirect:page.jsp";
}
```

页面访问快捷设定 (InternalResourceViewResolver)

展示页面的保存位置通常固定，且结构相似，可以设定通用的访问路径，简化页面配置格式

```xml
<bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/pages/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

```java
public String showPage3() {
    return "page";
}
// 返回 /WEB-INF/pages/page.jsp
```

如果未设定了返回值，使用void类型，则默认使用访问路径作页面地址的前缀后缀。

```java
//最简页面配置方式，使用访问路径作为页面名称，省略返回值
@RequestMapping("/showPage5")
public void showPage5() {
    System.out.println("user mvc controller is running ...");
}
```

带数据页面跳转

使用 HttpServletRequest 类型形参进行数据传递

```java
@RequestMapping("/showPageAndData1")
public String showPageAndData1(HttpServletRequest request) {
    request.setAttribute("name","rushuni");
    return "page";
}
```

使用 Model 类型形参进行数据传递

```java
@RequestMapping("/showPageAndData2")
public String showPageAndData2(Model model) {
    model.addAttribute("name","rushuni");
    Book book = new Book();
    book.setName("SpringMVC");
    book.setPrice(66.6d);
    model.addAttribute("book",book);
    return "page";
}
```

使用ModelAndView类型形参进行数据传递，将该对象作为返回值传递给调用者

```java
//使用ModelAndView形参传递参数，该对象还封装了页面信息
@RequestMapping("/showPageAndData3")
public ModelAndView showPageAndData3(ModelAndView modelAndView) {
    //ModelAndView mav = new ModelAndView();     替换形参中的参数
    Book book  = new Book();
    book.setName("SpringMVC入门案例");
    book.setPrice(66.66d);
    //添加数据的方式，key对value
    modelAndView.addObject("book",book);
    //添加数据的方式，key对value
    modelAndView.addObject("name","Jockme");
    //设置页面的方式，该方法最后一次执行的结果生效
    modelAndView.setViewName("page");
    //返回值设定成ModelAndView对象
    return modelAndView;
}
```

返回json数据

基于 response 返回数据的简化格式，返回JSON数据

```java
//使用jackson进行json数据格式转化
@RequestMapping("/showData3")
@ResponseBody
public String showData3() throws JsonProcessingException {
    Book book  = new Book();
    book.setName("SpringMVC入门");
    book.setPrice(66.66d);
    ObjectMapper om = new ObjectMapper();
    return om.writeValueAsString(book);
}
```

使用SpringMVC提供的消息类型转换器将对象与集合数据自动转换为JSON数据

最常用

```java
//使用 SpringMVC 注解驱动，对标注@ResponseBody注解的控制器方法进行结果转换，由于返回值为引用类型，自动调用jackson提供的类型转换器进行格式转换
@RequestMapping("/showData4")
@ResponseBody
public Book showData4() {
    Book book  = new Book();
    book.setName("SpringMVC入门");
    book.setPrice(66.66d);
    return book;
}
```

如果没有开启 springmvc 注解驱动，则需要手工添加信息类型转换器

```xml
<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">
  <property name="messageConverters">
    <list>
      <bean class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter"/>
    </list>
  </property>
</bean>
```

使用SpringMVC注解驱动简化配置

```xml
  <!--开启springmvc注解驱动，对@ResponseBody的注解进行格式增强，追加其类型转换的功能，具体实现由MappingJackson2HttpMessageConverter进行-->
  <mvc:annotation-driven/>
```

Servlet相关接口-Servlet相关接口替换方案

HttpServletRequest / HttpServletResponse / HttpSession

SpringMVC提供访问原始Servlet接口API的功能，通过形参声明即可

```java
@RequestMapping("/servletApi")
public String servletApi(HttpServletRequest request,
                         HttpServletResponse response, HttpSession session){
    System.out.println(request);
    System.out.println(response);
    System.out.println(session);
    request.setAttribute("name","rushuni");
    System.out.println(request.getAttribute("name"));
    return "page.jsp";
}
```

Head数据获取

```java
// 绑定请求头数据与对应处理方法形参间的关系
@RequestMapping("/headApi")
public String headApi(@RequestHeader("Accept-Language") String head){
    System.out.println(head);
    return "page.jsp";
}
```

Cookie 数据获取

```java
// 绑定请求Cookie数据与对应处理方法形参间的关系
@RequestMapping("/cookieApi")
public String cookieApi(@CookieValue("JSESSIONID") String jsessionid){
    System.out.println(jsessionid);
    return "page.jsp";
}
```

Session数据获取

```java
// 绑定请求Session数据与对应处理方法形参间的关系
@RequestMapping("/sessionApi")
public String sessionApi(@SessionAttribute("name") String name){
    System.out.println(name);
    return "page.jsp";
}
```

Session数据设置（了解）

Java 复制代码

```java
// 声明放入session范围的变量名称，适用于Model类型数据传参
@Controller
@SessionAttributes(names={"name"})
public class ServletController {
    @RequestMapping("/setSessionData2")
    public String setSessionDate2(Model model) {
        model.addAttribute("name", "Jock2");
        return "page.jsp";
    }
}
```

课后作业

- 熟悉 Spring MVC 的基础使用
  - 基础配置，请求，类型转换器的日期，请求映射，页面跳转
  - 返回 Json 数据
  - spring MVC 获取 Servelt 相关接口实例。