SpringMVC

[了解]MVC 模式

M 即 Model 数据模型，V 即 View 视图，C 即 Controller 控制器。

Model 的广义和狭义之说

广义：Model = Service 层 +Dao 层，由这两层来产生数据模型

狭义：Model 就是数据模型本身，就是 Controller 和 View 之间传递的数据，在这种说法下，MVC 又可归为 Web 层。

经典三层：web 表现层（view+controller）、service 业务层、dao 持久层

[了解]SpringMVC 概述

SpringMVC 是什么

- SpringMVC 是什么

SpringMVC 是一个**轻量级的 Web 表现层框架**，作用：用来写 Controller 接收请求跳转页面的，它是 Spring 框架的一部分。

说白了，用来替代 servlet 的。SpringMVC 是对 Servlet 的封装和增强，简化了 servlet 的操作。它已经超越了 Struts，成为目前最优秀的表现层框架。

原始 Servlet 中

第一步：取参

request.getParameter("param")

第二步：数据类型转换

Integer.valueOf()

http 协议：超文本传输协议

在 SpringMVC 中，因为它是对 servlet 的封装，所以可以很方便的拿到整型数据

- 什么时候使用 SpringMVC？
  你觉得 servlet 繁琐的时候
- 在哪儿使用 SpringMVC？
  在经典三层的 web 表现层使用 SpringMVC 框架
- SpringMVC 的优势
  操作特简单，性能特别高，灵活性特别强
- 与 Struts 框架相比
  解决了 struts 的安全性低,操作难,执行性能低的问题

拓展认识

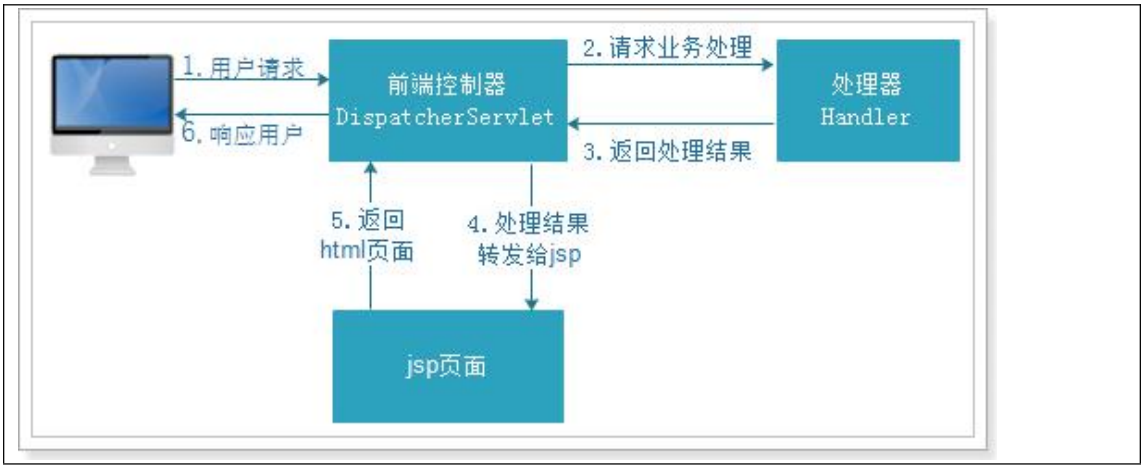
Spring + Spring jdbctemplate + SpringMVC --> Spring 全家桶 1.0 主流

Spring cloud + Spring Boot + Spring Data JPA + SpringMVC --> Spring 全家桶 2.0 潮流
Spring Cloud（微服务框架，单体工程拆分成很多个小（微）工程，订单服务、用户服务等）
dubbo（阿里巴巴微服务框架、hsf 框架）

[理解]SpringMVC 的处理流程

原始 Servlet 方式
一个项目中有很多个 servlet，每个 servlet 负责不同的业务或者功能，前台请求各个 servlet。好比自己去找水电工、木工等。

SpringMVC 全局只有一个 servlet，这个 servlet 就好比装修队的头儿



SpringMVC 的入门案例

入门案例需求

浏览器输入 url 发起请求，该 url 请求被 SpringMVC 框架拦截处理，把后台服务器的当前时间输出到 jsp 页面显示

入门案例实现分析

- jsp 页面（显示当前系统时间）
- 前端控制器 dispatcherServlet 在 web.xml 中配置
- 写 Handler 获取当前系统时间（这里的 Handler 就是 Controller）
- @Controller 标识处理类，并配置扫描

入门案例步骤

- 使用骨架创建 web 应用，配置 pom.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.springmvc</groupId>
  <artifactId>springmvc01_main</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging>

  <name>springmvc01_main 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>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>

      <!-- 引入 spring 框架-->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>5.0.2.RELEASE</version>
    </dependency>

      <!-- 引入 springmvc 模块-->
    <dependency>
```

```
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.0.2.RELEASE</version>
    </dependency>

        <!-- 引入 web 开发所需要的两个 jar，往往一起引入即可,start-->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jsp-api</artifactId>
        <version>2.0</version>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>servlet-api</artifactId>
        <version>3.0</version>
          <!-- 使用范围 provide，只在开发、编译阶段有效，打包之后是由服务器环境提
供-->
        <scope>provided</scope>
    </dependency>
        <!-- 引入 web 开发所需要的两个 jar，往往一起引入即可,end-->
  </dependencies>

  <build>
    <finalName>springmvc01_main</finalName>
    <pluginManagement><!-- lock down plugins versions to avoid using Maven defaults
(may be moved to parent pom) -->
        <plugins>
          <plugin>
            <artifactId>maven-clean-plugin</artifactId>
            <version>3.0.0</version>
          </plugin>
          <!--                                                          see
http://maven.apache.org/ref/current/maven-core/default-bindings.html#Plugin_bindings_for_
war_packaging -->
          <plugin>
            <artifactId>maven-resources-plugin</artifactId>
            <version>3.0.2</version>
          </plugin>
          <plugin>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.7.0</version>
          </plugin>
```

```
            <plugin>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.20.1</version>
            </plugin>
            <plugin>
                <artifactId>maven-war-plugin</artifactId>
                <version>3.2.0</version>
            </plugin>
            <plugin>
                <artifactId>maven-install-plugin</artifactId>
                <version>2.5.2</version>
            </plugin>
            <plugin>
                <artifactId>maven-deploy-plugin</artifactId>
                <version>2.8.2</version>
            </plugin>
        </plugins>
    </pluginManagement>
  </build>
</project>
```

- result.jsp 页面开发

```
<%@ page isELIgnored="false" contentType="text/html;charset=UTF-8"
language="java" %>
<html>
<head>
    <title>测试结果页</title>
</head>
<body>
当前系统时间：${nowDate}
</body>
</html>
```

- springmvc 配置文件配置

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       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
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd
```

```
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd
">
    <!--扫描 Controller-->
    <context:component-scan base-package="com.springmvc.controller"/>

    <!--自动注册最优最合适的处理器适配器和处理器映射器-->
    <mvc:annotation-driven/>

    <!--配置视图解析器-->
    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <property name="suffix" value=".jsp"/>
    </bean>
</beans>
```

入门案例执行过程

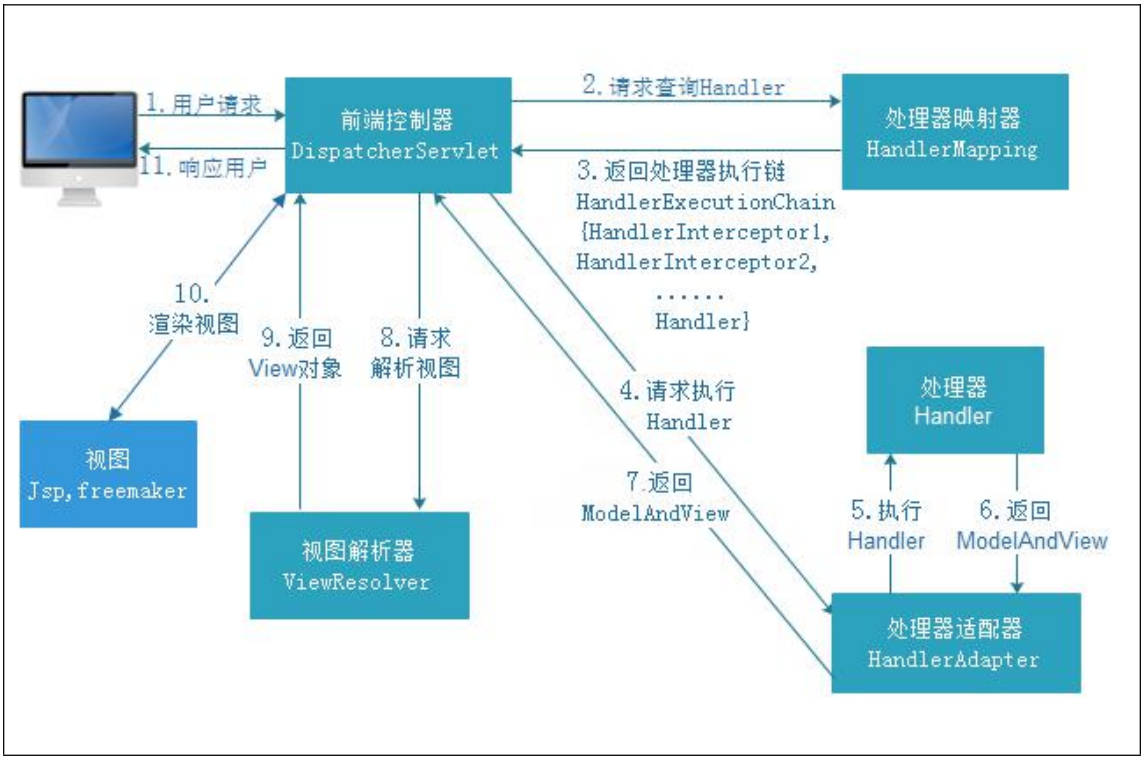
- tomcat 启动加载读取 web.xml
- 进一步读取到 web.xml 中配置的前端控制器 DispatcherServlet
- DispatcherServlet 会读取到 springmvc.xml,进一步去实例化 DefaultController 对象

当 url 请求到来时，首先被 DispatcherServlet 前端控制器拦截，然后前端控制器去找能够处理当前 url 的处理器 Handler（根据@RequestMapping 的配置）

- jsp 页面从 request 域当中获取到变量，进行显示

[理解]SpringMVC 架构

框架结构



架构流程

第一步：用户发送请求至前端控制器DispatcherServlet

第二步：DispatcherServlet收到请求调用HandlerMapping处理器映射器

第三步：处理器映射器根据请求Url找到具体的Handler（后端控制器），生成处理器对象及处理器拦截器(如果有则生成)一并返回DispatcherServlet

第四步：DispatcherServlet调用HandlerAdapter处理器适配器去调用Handler

第五步：处理器适配器执行Handler

第六步：Handler执行完成给处理器适配器返回ModelAndView

第七步：处理器适配器向前端控制器返回 ModelAndView，ModelAndView 是 SpringMVC 框架的一个底层对象，包括 Model 和 View

第八步：前端控制器请求视图解析器去进行视图解析
根据逻辑视图名来解析真正的视图。

第九步：视图解析器向前端控制器返回View

第十步：前端控制器进行视图渲染
就是将模型数据（在 ModelAndView 对象中）填充到 request 域

第十一步：前端控制器向用户响应结果

组件说明

- DispatcherServlet：前端控制器

接收用户请求，响应结果，相当于中央处理器，DispatcherServlet是整个流程控制的中心，由它调用其它组件完成用户请求的处理。DispatcherServlet的存在降低了组件之间的耦合性。

- HandlerMapping：处理器映射器

理解为一个Map<url,Hanlder>

HandlerMapping负责根据用户请求的Url找到Handler即处理器，SpringMVC提供了不同的映射器来实现不同的映射方式，例如：实现接口方式，注解方式等。

- Handler：处理器

在SpringMVC当中有两层意思：Controller或者Controller当中的方法
Handler相对于前端控制器DispatcherServlet来说是**后端控制器**，执行具体业务处理的，它在DispatcherServlet的控制下处理用户的具体请求。

- HandlAdapter：处理器适配器

不同的接口类型转换成usb，体现了万物归一的思想

通过HandlerAdapter对Handler处理器进行执行，这是适配器模式的应用。

- ViewResolver：视图解析器

ViewResolver进行视图解析，首先根据逻辑视图名解析成物理视图名即具体的页面地址，再生成View视图对象。

```
<!--视图解析器，配置视图的前缀和后缀
    InternalResourceViewResolver
-->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <!--配置视图页面的前缀-->
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <!--配置视图页面的后缀-->
    <property name="suffix" value=".jsp"/>
</bean>
```

- View：视图

SpringMVC 框架提供了很多的 View 视图类型的支持，包括：jsp、freemarkerView 等。我们最常用的视图就是 jsp

注意：在 SpringMVC 的各个组件中，处理器映射器、处理器适配器、视图解析器称为 SpringMVC 的三大组件。

需要我们开发的组件有 handler（后台的处理逻辑）、view（前端的显示页面）

配置<mvc:annotation-driven>

SpringMVC 对于三大件默认在底层已经进行了注册和使用（三大件已经存在于 SpringMVC 的运行机制中），默认使用的三大件已经过时了（虽然过时，但是依然能够保证程序运行的），建议我们使用更好更合适的三大件，只需要在 SpringMVC 的配置文件中添加一个标签<mvc:annotation-driven>，配置完成后会自动注册最合适的三大件

[掌握]RequestMapping 使用

- 多个 URL 路径映射到同一个 Handler（同一个方法）

```
/*
* 用例：多个url映射到同一个处理器（方法当中）
* */
@RequestMapping(value = {"gotoResultMultiUrl1","gotoResultMultiUrl2"})
public ModelAndView gotoResultMultiUrl(ModelAndView modelAndView) {
    Date date = new Date();
    modelAndView.addObject( attributeName: "nowDate", attributeValue: date + "--->>>gotoResultMultiUrl");
    modelAndView.setViewName("result");
    return modelAndView;
}
```

- 添加到类上面（分类管理，限定类中方法访问的前缀），一看到 url，就知道这个 url 所处理的业务领域

```
@Controller
/*分类管理，限定handler方法url请求的前缀*/
@RequestMapping("default")
public class DefaultController {
```

```
<fieldset>
    <p>测试用例USE_CASE01:SpringMVC入门案例</p>
    <a href="${pageContext.request.contextPath}/default/gotoResult.action">点击测试</a>
</fieldset>
```

- <method>属性限定请求方法，请求的 url 相同，请求方式（get/post）不同进入不同方法处理

  访问的 url 是一样的（Handler 名字是一样的），但是我根据你不同的请求方式（get/post）进入不同的方法处理

  请求的 url 一样，但是请求方式不一样（get/post）

```
/*
 * 用例：url 相同，根据请求方式（get/post）不同进入不同方法处理
 * */
@RequestMapping(value = "gotoResultSameUrl",method = {RequestMethod.GET})
public ModelAndView gotoResultSameUrlGet(ModelAndView modelAndView) {
    Date date = new Date();
    modelAndView.addObject("nowDate",date + "--->>>gotoResultSameUrlGet");
    modelAndView.setViewName("result");
    return modelAndView;
}
```

```
/*
 * 用例：url 相同，根据请求方式（get/post）不同进入不同方法处理
 * */
@RequestMapping(value = "gotoResultSameUrl",method = {RequestMethod.POST})
public ModelAndView gotoResultSameUrlPost(ModelAndView modelAndView) {
    Date date = new Date();
    modelAndView.addObject("nowDate",date + "--->>>gotoResultSameUrlPost");
    modelAndView.setViewName("result");
    return modelAndView;
}
```

- <params>属性限定请求参数，支持简单的表达式语法
  url 一样，根据携带参数的不同进入不同的方法处理
  url 相同，请求方式相同，请求参数不同进入不同 hanlder 方法处理

param1：表示请求必须包含名为 param1 的请求参数 !param1：表示请求不能包含名为 param1 的请求参数 param1 != value1：表示请求包含名为 param1 的请求参数，但其值不能为 value1 {“param1=value1”, “param2”}：请求必须包含名为 param1 和 param2 的两个请求参数，且 param1 参数的值必须为 value1

[掌握]请求参数绑定（接收）

默认支持 Servlet API 作为方法参数

```
/**
 * 测试用例：springmvc对原生servlet api的使用支持
 * 在SpringMVC当中可以在方法中声明HttpServletRequest\HttpServletResponse\HttpSession使用即可
 * @param modelAndView
 * @return
 */
@RequestMapping("sendSimpleParams")
public ModelAndView sendSimpleParams(HttpServletRequest request, HttpServletResponse response,
    String id = request.getParameter( name: "id");
    Date date = new Date();
    modelAndView.addObject( attributeName: "nowDate", attributeValue: date + "--->>>id:" + id);
    modelAndView.setViewName("result");
    return modelAndView;
}
```

绑定简单类型参数

参数类型推荐使用包装数据类型，因为基础数据类型不可以为 null
整型：Integer、int

字符串：String
单精度：Float、float
双精度：Double、double
布尔型：Boolean、boolean
说明：对于布尔类型的参数，**请求的参数值为 true 或 false。或者 1 或 0**

- 绑定简单数据类型参数，只需要直接形参声明
  注意：形参的参数名和传递的参数名保持一致
  建议使用包装类型

参数绑定两步骤：1、从 request 中 getParameter；2、根据形参定义的参数类型进行类型转换和赋值

@RequestParam 使用

当形参和传递的参数名称不一致时使用 RequestParam 进行手动映射，类似于 Mybatis 当中国的 resultMap 的作用

```
/**
 * @RequestParam:从请求中把某一个参数取出赋值给当前形参
 * value 属性：指定从请求中获取哪个参数赋值给当前形参
 * defaultValue：默认值，如果获取的参数值为空，则取该默认值（场景：分页的第一页）
 * @param modelAndView
 * @param id
 * @return
 */
@RequestMapping(value = "sendSimpleParams2")
public ModelAndView sendSimpleParams2(ModelAndView modelAndView,@RequestParam(value = "ids",defaultValue = "3") Integer id) {
    Date date = new Date();
    modelAndView.addObject("nowDate", date  + " id==>>" + id);
    modelAndView.setViewName("result");
    return modelAndView;
}
```

绑定 pojo 类型参数

直接形参声明 pojo 即可接收
要求：传递的参数名必须和 pojo 属性名对应

```
/**
 * 绑定 pojo 类型参数，只需要直接在形参中声明 pojo 即可
 * 要求：请求参数名和 pojo 属性名保持一致
 * @param modelAndView
```

```
 * @param user
 * @return
 */
@RequestMapping("sendPojo")
public ModelAndView sendPojo(ModelAndView modelAndView,User user) {
    Date date = new Date();
    modelAndView.addObject("nowDate", date   + " user==>>" + user);
    modelAndView.setViewName("result");
    return modelAndView;
}
```

重点在于传参参数的命名（和 pojo 属性名保持一致）

绑定 pojo 包装对象参数

重点在于传参参数的命名

pojo 包装对象首先就是一个普通的 pojo，就应该按照上面绑定 pojo 的要求来，然后进一步处理

传参时参数名，首先应该定位到包装对象的属性名，如果不能确定数据，通过属性.的方式进一步锁定即可

- 前台传参

```
<fieldset>
    <p>传递（绑定）pojo 包装类型参数</p>
    <a
href="${pageContext.request.contextPath}/user/sendPojoPackage.action?user.id=1&user.usernam
e=lisi">测试</a>
</fieldset>
```

- 后台接收

```
/**
 * 绑定 pojo 包装类型参数，只需要直接在形参中声明 pojo 包装对象即可
 * 注意：绑定 pojo 包装类型参数，重点在于前台参数参数名的取法，首先定位到 pojo 的
属性，然后通过.的方式进一步确定
 * @param modelAndView
 * @return
 */
@RequestMapping("sendPojoPackage")
public ModelAndView sendPojoPackage(ModelAndView modelAndView, QueryVo queryVo) {
    Date date = new Date();
    modelAndView.addObject("nowDate", date   + " queryVo==>>" + queryVo);
    modelAndView.setViewName("result");
    return modelAndView;
}
```

绑定 pojo 包含集合类型参数

传递绑定集合类型(List/Map)，作为 POJO 的一个属性来传递

- 前端页面

```
<fieldset>
    <p>测试用例 USE_CASE10:SpringMVC 接收集合（List）类型参数</p>
    <form                                                    method="post"
action="${pageContext.request.contextPath}/user/sendList.action">
        <table>
            <tr>
                <td>主键</td>
                <td>用户名</td>
                <td>性别</td>
            </tr>
            <tr>
                <td><input   type="text" name="userList[0].id"/></td>
                <td><input   type="text" name="userList[0].username"/></td>
                <td><input   type="text" name="userList[0].sex"/></td>
            </tr>
            <tr>
                <td><input   type="text" name="userList[1.id"/></td>
                <td><input   type="text" name="userList[1].username"/></td>
                <td><input   type="text" name="userList[1].sex"/></td>
            </tr>
            <tr>
                <td><input   type="text" name="userList[2].id"/></td>
                <td><input   type="text" name="userList[2].username"/></td>
                <td><input   type="text" name="userList[2].sex"/></td>
            </tr>

        </table>
        <input type="submit" value="批量修改提交">
    </form>
</fieldset>

<fieldset>
    <p>测试用例 USE_CASE11:SpringMVC 接收集合（Map）类型参数</p>
    <form                                                    method="post"
action="${pageContext.request.contextPath}/user/sendMap.action">
        <table>
```

```
                <tr>
                    <td>主键</td>
                    <td>用户名</td>
                    <td>性别</td>
                </tr>
                <tr>
                    <td><input  type="text" name="userMap['key1'].id"/></td>
                    <td><input type="text" name="userMap['key1'].username"/></td>
                    <td><input  type="text" name="userMap['key1'].sex"/></td>
                </tr>
                <tr>
                    <td><input  type="text" name="userMap['key2'].id"/></td>
                    <td><input type="text" name="userMap['key2'].username"/></td>
                    <td><input  type="text" name="userMap['key2'].sex"/></td>
                </tr>
                <tr>
                    <td><input  type="text" name="userMap['key3'].id"/></td>
                    <td><input type="text" name="userMap['key3'].username"/></td>
                    <td><input  type="text" name="userMap['key3'].sex"/></td>
                </tr>

            </table>
            <input type="submit" value="批量修改提交">
        </form>
</fieldset>
```

- pojo

```
ublic class QueryVo {

    private User user;

    private List<User> userList;

    private Map<String,User> userMap;

    public Map<String, User> getUserMap() {
        return userMap;
    }

    public void setUserMap(Map<String, User> userM
        this.userMap = userMap;
```

- 后端 handler

```
/**
```

```
 * 用例：springmvc 绑定 list 集合
 * 重点在于前端参数的取值方式
 * @param modelAndView
 * @return
 */
@RequestMapping("sendList")
public ModelAndView sendList(QueryVo queryVo, ModelAndView modelAndView) {
    Date date = new Date();
    modelAndView.addObject("nowDate",date + "--->>>queryVo:" + queryVo);
    modelAndView.setViewName("result");
    return modelAndView;
}


/**
 * 用例：springmvc 绑定 map 集合
 * 重点在于前端参数的取值方式
 * @param modelAndView
 * @return
 */
@RequestMapping("sendMap")
public ModelAndView sendMap(QueryVo queryVo, ModelAndView modelAndView) {
    Date date = new Date();
    modelAndView.addObject("nowDate",date + "--->>>queryVo:" + queryVo);
    modelAndView.setViewName("result");
    return modelAndView;
}
```

自定义类型转换器

- 自定义类型转换器

```
package com.springmvc.utils;

import org.springframework.core.convert.converter.Converter;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

/**
```

```
 * 自定义类型转换器实现改接口，Converter<S,T>
 *     S：source，转换前的数据类型
 *     T：target，转换后的数据类型
 */
public class DateConverter implements Converter<String,Date> {
    @Override
    public Date convert(String source) {
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd");
        try {
            return simpleDateFormat.parse(source);
        } catch (ParseException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

- 注册自定义类型转换器

```
<!-- 注册自定义类型转换类-->
    <bean id="conversionService" class="org.springframework.context.support.ConversionServiceFactoryBean">
        <property name="converters">
            <set>
                <bean class="com.util.StringToDate"/>
            </set>
        </property>
    </bean>
```

[了解]消息头相关的两个注解

@RequestHeader

```
/**
 * 获取http请求头信息
 * @param userAgent
 */
@RequestMapping("testRequestHeader")
public void testRequestHeader(@RequestHeader("User-Agent") String userAgent) {
    System.out.println(userAgent);
}
```

@CookieValue

```java
/**
 * 获取cookie值
 * @param jsessionid
 */
@RequestMapping("testCookieValue")
public void testCookieValue(@CookieValue("JSESSIONID") String jsessionid) {
    System.out.println(jsessionid);
}
```

[了解]扩展知识

乱码问题处理

- Post 请求乱码，web.xml 中加入过滤器

```xml
<!-- 解决post乱码问题 -->
<filter>
  <filter-name>encoding</filter-name>
  <filter-class>
    org.springframework.web.filter.CharacterEncodingFilter
  </filter-class>
  <!-- 设置编码参是UTF8 -->
  <init-param>
    <param-name>encoding</param-name>
    <param-value>UTF-8</param-value>
  </init-param>
  <init-param>
    <param-name>forceEncoding</param-name>
    <param-value>true</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>encoding</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

- Get 请求乱码

Get 请求乱码需要修改 tomcat 下 server.xml 的配置

```
<Connector URIEncoding="utf-8" connectionTimeout="20000" port="8080"
```

```
protocol="HTTP/1.1" redirectPort="8443"/>
```

[掌握]Controller 方法（Handler）的返回值

- 返回 ModelAndView
- 返回字符串（直接返回逻辑视图名字符串，数据使用 Model 或者 ModelMap 封装）
  ModelAndView = model + view（逻辑视图名）

```
/**Controller 方法返回 String 使用 Model
 * Model 是接口
 * @param model
 * @return
 */
@RequestMapping("gotoResultReturnStrModel")
public String gotoResultReturnStrModel(Model model) {
    model.addAttribute("nowDate",new Date() + "--->>>model");
    return "result";
}


/**Controller 方法返回 String 使用 ModelMap
 * ModelMap 是接口 Model 的实现
 * 注意：Model 没有 get 方法，ModelMap 方法有 get 方法，所以你要取值就声明 ModelMap 使用
 * @param model
 * @return
 */
@RequestMapping("gotoResultReturnStrModelMap")
public String gotoResultReturnStrModelMap(ModelMap model) {
    model.addAttribute("nowDate",new Date() + "--->>>modelMap");
    return "result";
}
```

- 返回 void（了解）
  - 直接使用 HttpServletRequest 对象转发请求
  - 直接使用 HttpServletResponse 对象重定向请求
  - 直接使用 HttpServletResponse 输出结果

```
/**Controller 方法返回 void
 * 使用 request 进行请求转发
 * @param model
 * @return
```

```
 */
@RequestMapping("gotoResultReturnVoid")
public void gotoResultReturnVoid(ModelMap model, HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    request.getRequestDispatcher("/WEB-INF/jsp/result.jsp").forward(request,response);
}


/**Controller 方法返回 void
 * 使用 response 重定向
 * @param model
 * @return
 */
@RequestMapping("gotoResultReturnVoidRedirect")
public void gotoResultReturnVoidRedirect(ModelMap model, HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    response.sendRedirect("gotoResult.action");
}


/**Controller 方法返回 void
 * 使用 response 直接输出字符串
 * @param model
 * @return
 */
@RequestMapping("gotoResultReturnVoidWrite")
public void gotoResultReturnVoidWrite(ModelMap model, HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    response.getWriter().write("Hello SpringMVC!");
}
```

- 转发和重定向（返回字符串形式）

```
/**Controller 方法返回 void
 * 使用 request 进行请求转发
 * @param model
 * @return
 */
@RequestMapping("gotoResultReturnVoid2")
public String gotoResultReturnVoid2(ModelMap model, HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    //request.getRequestDispatcher("/WEB-INF/jsp/result.jsp").forward(request,response);
    return "forward:/WEB-INF/jsp/result.jsp";
}
```

```
/**Controller 方法返回 void
 * 使用 response 重定向
 * @param model
 * @return
 */
@RequestMapping("gotoResultReturnVoidRedirect2")
public String gotoResultReturnVoidRedirect2(ModelMap model, HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    //response.sendRedirect("gotoResult.action");
    return "redirect:gotoResult.action";
}
```

[应用]@ModelAttribute@SessionAttributes

- @ModelAttribute
  场景有限
- @SessionAttributes

```
// value配置存入session的数据项名称
/*@SessionAttributes(value = {"field1"})*/
// types配置存入session的数据类型，types和values可以结合一起使用
@SessionAttributes(types = {String.class})
public class UserController {
```

```
    /**
     * 第一个handler：发起一个请求，后端把请求参数写入 session
     * 第二个handler：发起一个请求，取出 session 当中存入的数据
     * 第三个handler：发起一个请求，清空 session
     */
    @RequestMapping("setSessionValue")
    public String setSessionValue(Model model) {
        model.addAttribute("field1","value1");
        model.addAttribute("nowDate",new Date() + "--->>>该请求向 session 写入数据
");
        return "result";
    }


    @RequestMapping("getSessionValue")
    public String getSessionValue(ModelMap model) {
        model.addAttribute("nowDate",new Date() + "--->>>从 session 取出数据： " +
```

```
        model.get("field1"));
            return "result";
        }


        /*
            SessionStatus 提供 setComplete 方法用于清空 session 中通过@SessionAttributes
共享的数据
         */
@RequestMapping("clearSession")
public String clearSession(Model model,SessionStatus sessionStatus) {
        sessionStatus.setComplete();
        model.addAttribute("nowDate",new Date() + "--->>>清空@SessionAttributes 共享
的数据");
        return "result";
}
```

[应用]SpringMVC 对 Restful 风格 URL 的支持

直观而言，是传参风格的一种变化，原来参数跟在？后面，现在参数在 url 路径中

- 什么是 RESTful？

RESTful 就是一个资源定位及资源操作的风格。不是标准也不是协议，只是一种 url 设计风格。基于这个风格设计的软件可以更简洁，更有层次。

资源：互联网所有的事物都可以被抽象为资源 url（只要互联网上的事物可以用一个 url 来表示，那么它就是一个资源）

资源操作：使用 POST、DELETE、PUT、GET 不同方法对同一资源（同一 url）进行操作。
分别对应 添加、 删除、修改、查询

Http 主要的请求方式	
get	主要是想做 select
post	主要是想做 insert
put	主要是想做 update
delete	主要是想做 delete

以上是 http 协议的标准请求方式，当然你用 post 请求也完全可以完成 crud 操作（因为归根结底无非就是把参数传递到后台对应处理即可）

传统方式操作资源

操作啥 （原来 url）？操作谁（传入的参数）

url 中先定义动作，然后传递的参数表明这个动作操作的是哪个对象（数据）

先定位动作，然后定位对象

http://localhost:8080/springmvc02/user/queryUserById.action?id=1 查询

http://localhost:8080/springmvc02/ user /saveUser.action 新增

http://localhost:8080/springmvc02/ user /updateUser.action 更新

http://localhost:8080/springmvc02/ user /deleteUserById.action?id=1 删除

使用 RESTful 操作资源

先定义对象

http://localhost:8080/springmvc02/user/1 操作的对象） 查询,GET

http://localhost:8080/springmvc02/ user 新增,POST

http://localhost:8080/springmvc02/ user 更新,PUT

http://localhost:8080/springmvc02/ user /1 删除,DELETE

- **HiddentHttpMethodFilter 过滤器**

作用：由于浏览器 form 表单只支持 GET 与 POST 请求，而 DELETE、PUT 等 method 并不支持，Spring3.0 之后添加了一个过滤器，可以将浏览器请求改为指定的请求方式，发送给我们的控制器方法，使得支持 GET、POST、PUT 与 DELETE 请求。

- 第一步：在 web.xml 中配置该过滤器

```
<!--HiddenHttpMethodFilter 过滤器可以将 POST 请求转化为 PUT\DELETE 请求-->
<filter>
  <filter-name>methodFilter</filter-name>
  <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>
</filter>
<filter-name>methodFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

- 第二步：请求方式必须使用 post 请求
- 第三步：增加请求参数_method，该参数的取值就是我们需要的请求方式

```
<div>
    <h2>SpringMVC 对 Restful 风格 url 的支持</h2>
    <fieldset>
        <p>测试用例 Use_case11：SpringMVC 对 Restful 风格 url 的支持</p>
        <%--<a
href="${pageContext.request.contextPath}/user/queryUserById.action?id=11"> 点击测试</a>--%>
        <a href="${pageContext.request.contextPath}/user/15.action">rest_get 测试
</a>
        <form                                                    method="post"
action="${pageContext.request.contextPath}/user.action">
            <input   type="text" name="id"/>
            <input type="text" name="username"/>
```

```
                <input type="submit" value="提交 rest_post 请求"/>
            </form>
            <form                                                    method="post"
action="${pageContext.request.contextPath}/user.action">
                <input type="hidden" name="_method" value="PUT"/>
                <input   type="text" name="id"/>
                <input type="text" name="username"/>
                <input type="submit" value="提交 rest_put 请求"/>
            </form>
            <form                                                    method="post"
action="${pageContext.request.contextPath}/user/15.action">
                <input type="hidden" name="_method" value="delete"/>
                <input type="submit" value="提交 rest_delete 请求"/>
            </form>
        </fieldset>
</div>
```

- 第四步：后台接收

```
@RequestMapping(value = "{id}",method = RequestMethod.GET)
public String queryUserById1(@PathVariable("id") Integer id, Model model) {
        System.out.println("======查询： " + id);
        model.addAttribute("nowDate",new Date());
        return "redirect:/default/gotoResult.action";
}
@RequestMapping(value = "{id}",method = RequestMethod.DELETE)
public String queryUserById2(@PathVariable("id") Integer id,Model model) {
        System.out.println("======删除： " + id);
        model.addAttribute("nowDate",new Date());
        return "redirect:/default/gotoResult.action";
}
@RequestMapping(value = "",method = RequestMethod.POST)
public String queryUserById3(User user,Model model) {
        System.out.println("======新增： " + user);
        model.addAttribute("nowDate",new Date());
        return "redirect:/default/gotoResult.action";
}
@RequestMapping(value = "",method = RequestMethod.PUT)
public String queryUserById4(User user,Model model) {
        System.out.println("======更新： " + user);
        model.addAttribute("nowDate",new Date());
        return "redirect:/default/gotoResult.action";
}
```

[应用]Json 数据交互

Json 数据是咱们企业级开发数据交互经常使用的一种方式，它比较轻量级，格式比较清晰（系统间接口调用/前后端调用，json 数据格式都广为使用）

Json 数据交互：前端传递 json 字符串到后台，后台如何能够自动转换为 pojo 对象；后台 return 对象，能否前端直接接收到 json 格式的字符串

- @RequestBody 注解

  作用：用于获取请求体（按照 http 协议进行一个完整的封装，往往都是由请求头+请求体等组成）内容，不适用于 Get 请求方式，只能使用 post 请求方式

```
作用：
    用于获取请求体内容。直接使用得到是 key=value&key=value...结构的数据。
    get 请求方式不适用。
属性：
    required：是否必须有请求体。默认值是:true。当取值为 true 时,get 请求方式会报错。如果取值为 false，get 请求得到是 null。
```

  更多的是用于将 JSON 字符串转换为 POJO 对象

  引入 json 相关 jar 坐标

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>2.9.0</version>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.9.0</version>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-annotations</artifactId>
    <version>2.9.0</version>
</dependency>
```

  前端 ajax 传递 json 字符串（post）到后台，后台直接接收为对象

- @ResponseBody 注解

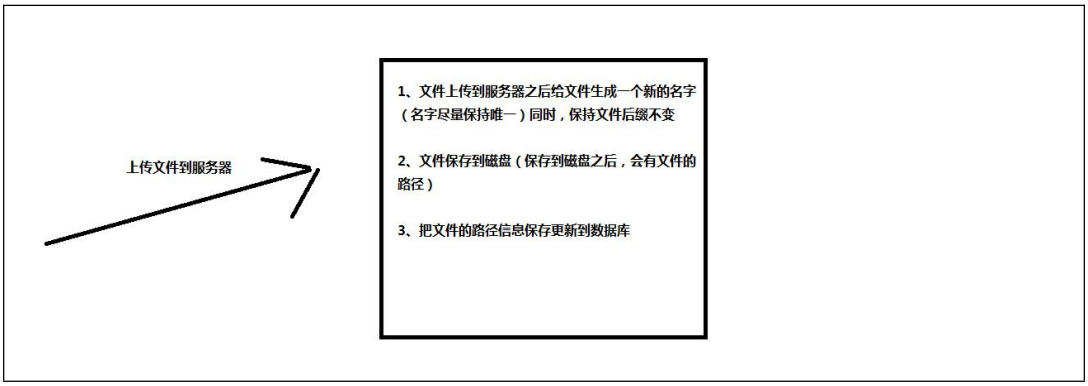
  作用：该注解用于将 `Controller` 的方法返回的对象转换为 json 字符串返回给客户端

  - 代码

```
@RequestMapping("sendJsonStr")
public @ResponseBody User sendJsonStr(@RequestBody User user) {
    user.setAddress("上海");
    System.out.println("---------->>>>sendJsonStr:" + user);
    return user;
```

```
}
$(function () {
    $("#ajaxBtn").bind("click",function () {
        $.ajax({
            url:"${pageContext.request.contextPath}/user/sendJsonStr.action",
            type:"post",
            data:'{"id":1,"username":"zhangsan"}',
            contentType:"application/json;charset=utf-8",
            dataType:"json",
            success:function (data) {
                alert(data);
                alert(data.address);
                console.log(data);
            }
        })
    })
})
```

[应用]SpringMVC 实现文件上传



- 前提条件

```
<fieldset>
    <p>上传图片</p>
    <form method="post" enctype="multipart/form-data" action="user/testUploadFile.action">
        <input type="file" name="uploadFile"/>
        <input type="submit"/>
    </form>
</fieldset>
```

- 引入坐标

```
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.3.1</version>
```

```
</dependency>
```

- 编写 Handler

```
@RequestMapping("testUploadFile")
public String testUploadFile(MultipartFile uploadFile, HttpServletRequest request) throws IOException {
    // 文件原名，如xxx.jpg
    String originalFilename = uploadFile.getOriginalFilename();
    // 获取文件的扩展名,如jpg
    String extendName = originalFilename.substring(originalFilename.lastIndexOf(".") + 1, originalFilename.length());
    String uuid = UUID.randomUUID().toString();
    // 新的文件名字
    String newName = uuid + "." + extendName;
    String realPath = request.getSession().getServletContext().getRealPath("/uploads");
    // 解决文件夹存放文件数量限制，按日期存放
    String datePath = new SimpleDateFormat("yyyy-MM-dd").format(new Date());
    File floder = new File(realPath + "/" + datePath);
    if(!floder.exists()) {
        floder.mkdirs();
    }
    uploadFile.transferTo(new File(floder,newName));
    return "success";
}
```

- 配置文件上传解析器

```
<!-- id 的值是固定的-->
<bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <!-- 设置上传文件最大 5M-->
    <property name="maxUploadSize" value="5242880"/>
</bean>
```

- 跨服务器上传

  分服务器的目的

在实际开发中，我们会有很多处理不同功能的服务器。例如：

应用服务器：负责部署我们的应用

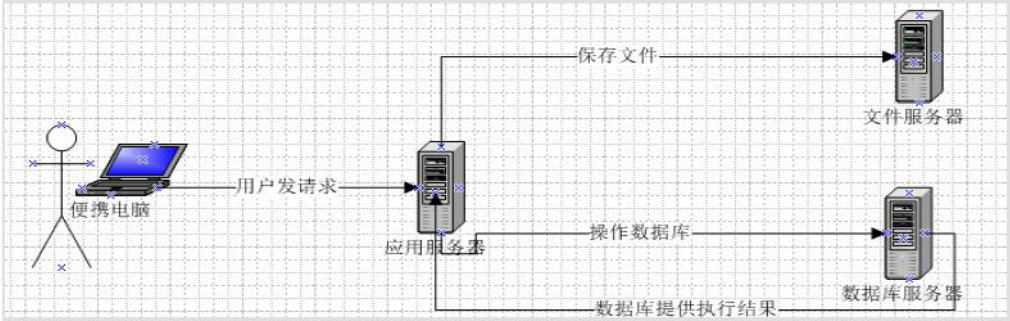
数据库服务器：运行我们的数据库

缓存和消息服务器：负责处理大并发访问的缓存和消息

文件服务器：负责存储用户上传文件的服务器。

（注意：此处说的不是服务器集群）

分服务器处理的目的是让服务器各司其职，从而提高我们项目的运行效率。



- 在文件服务器的 tomcat 配置中加入，允许读写操作

```
<!--                      relative to the context root. This overrides   -->
<!--                      globalXsltFile[null]                           -->
<!--                                                                     -->
<!--   globalXsltFile     Site wide configuration version of             -->
<!--                      localXsltFile. This argument must either be an -->
<!--                      absolute or relative (to either                -->
<!--                      $CATALINA_BASE/conf or $CATALINA_HOME/conf)    -->
<!--                      path that points to a location below either    -->
<!--                      $CATALINA_BASE/conf (checked first) or         -->
<!--                      $CATALINA_HOME/conf (checked second).[null]    -->
<!--                                                                     -->
<!--   showServerInfo     Should server information be presented in the  -->
<!--                      response sent to clients when directory        -->
<!--                      listings is enabled? [true]                    -->

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

- 主应用引入 jar 坐标

```
<dependency>
   <groupId>com.sun.jersey</groupId>
   <artifactId>jersey-core</artifactId>
   <version>1.18.1</version>
</dependency>
<dependency>
   <groupId>com.sun.jersey</groupId>
```

```
    <artifactId>jersey-client</artifactId>
    <version>1.18.1</version>
</dependency>
```

- Handler 处理程序

```
@RequestMapping("testUploadFileRemote")
public String testUploadFileRemote(MultipartFile uploadFile, HttpServletRequest request) throws IOException {
    // 文件原名，如xxx.jpg
    String originalFilename = uploadFile.getOriginalFilename();
    // 获取文件的扩展名,如jpg
    String extendName = originalFilename.substring(originalFilename.lastIndexOf(".") + 1, originalFilename.length());
    String uuid = UUID.randomUUID().toString();
    // 新的文件名字
    String newName = uuid + "." + extendName;
    String realPath = request.getSession().getServletContext().getRealPath("/uploads");
    // 解决文件夹存放文件数量限制，按日期存放
    String datePath = new SimpleDateFormat("yyyy-MM-dd").format(new Date());
    File floder = new File(realPath + "/" + datePath);
    if(!floder.exists()) {
        floder.mkdirs();
    }

    Client client = Client.create();
    String baseUrl = "http://localhost:8081/fileserver/uploads/";
    WebResource webResource = client.resource(baseUrl+newName);
    String result = webResource.put(String.class,uploadFile.getBytes());
    System.out.println(result);
    return "success";
}
```

注意：文件上传解析器仍然需要配置

[理解]SpringMVC 中的异常处理

Controller—>Service—>Dao 层，异常统一向上抛出，可以自定义全局异常处理器统一处理异常

异常类型：编译异常、运行时异常；运行时异常、预期异常（自定义异常）

- 自定义异常

```
package com.springmvc.exception;
```

```java
public class MyException extends Exception {
    private String message;

    public MyException(String message) {
        this.message = message;
    }

    @Override
    public String getMessage() {
        return message;
    }
}
```

- 自定义异常处理器

```java
package com.springmvc.exception;

import org.springframework.web.servlet.HandlerExceptionResolver;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class MyExceptionResolver implements HandlerExceptionResolver {
    @Override
    public ModelAndView resolveException(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex) {

        String message = null;
        if(ex instanceof MyException) {
            // 自定义异常
            message = ex.getMessage();
        }else {
            // 运行时异常
            message = "系统出现未知异常，请联系管理员";
        }

        // 跳转到一个友好的提示页面
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("exception",message);
        modelAndView.setViewName("exception");
        return modelAndView;
    }
}
```

- 注册自定义异常处理器

```
<bean id="myExceptionResolver"
class="com.springmvc.exception.ExceptionHandler"></bean>
```

[理解]SpringMVC 中拦截器的使用

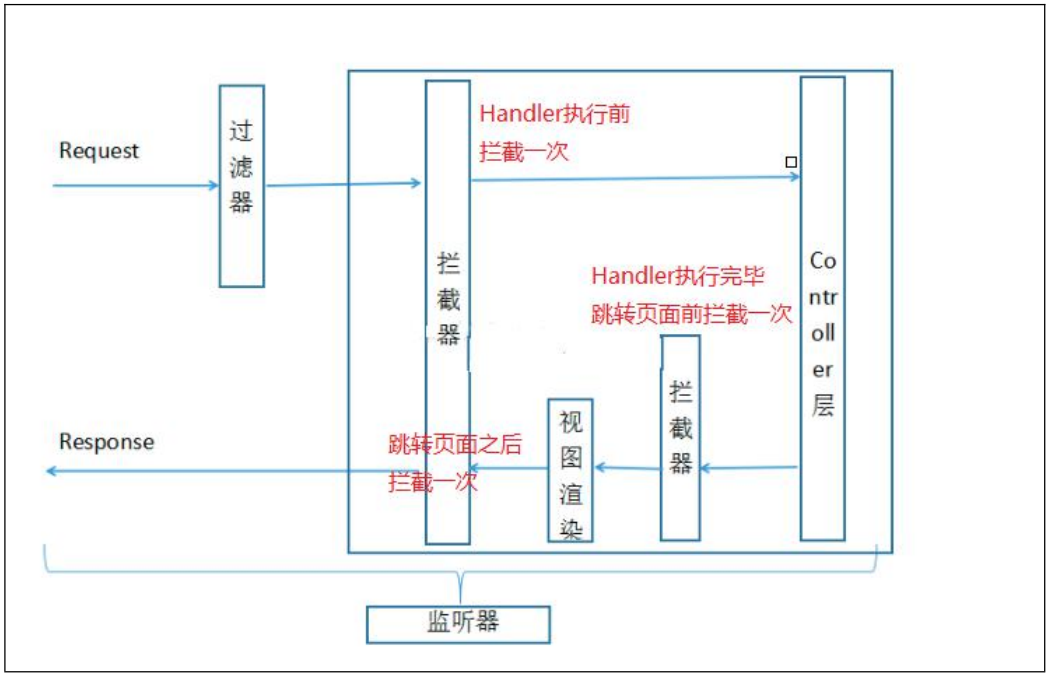
认识拦截器

- Servlet：处理 Request 请求和 Response 响应
- 过滤器（Filter）：对 Request 请求起到过滤的作用，**作用在 Servlet 之前**，如果配置为/*可以对所有的资源访问（servlet、js/css 静态资源等）进行过滤处理
- 监听器（Listener）：实现了 javax.servlet.ServletContextListener 接口的服务器端组件，**它随 Web 应用的启动而启动，只初始化一次，然后会一直运行监视，随 Web 应用的停止而销毁**

  作用一：做一些初始化工作

  作用二：**监听 web 中的特定事件**，比如 HttpSession,ServletRequest 的创建和销毁；变量的创建、销毁和修改等。可以在某些动作前后增加处理，实现监控，比如统计在线人数，利用 HttpSessionLisener 等。
- 拦截器（Interceptor）：**是 SpringMVC、Struts 等表现层框架自己的**，不会拦截 jsp/html/css/image 的访问等，只会**拦截访问的控制器方法（Handler）**。

  从配置的角度也能够总结发现：serlvet、filter、listener 是配置在 web.xml 中的，而 interceptor 是配置在表现层框架自己的配置文件中的
  - 在 Handler 业务逻辑执行之前拦截一次
  - 在 Handler 逻辑执行完毕但未跳转页面之前拦截一次
  - 在跳转页面之后拦截一次



SpringMVC 中自定义拦截器

- 实现 HandlerInterceptor 接口

```
package com.springmvc.interceptor;

import org.springframework.web.method.HandlerMethod;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * SpringMVC 中自定义拦截器实现 HandlerInterceptor 接口
 */
public class MyInterceptor implements HandlerInterceptor {

    /**
     * 之前执行：Handler 逻辑真正执行之前执行
     * @param request
     * @param response
     * @param handler
     * @return 代表是否放行，true 放行，false 中止
     */
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) {
        HandlerMethod handlerMethod = (HandlerMethod)handler;
        System.out.println("===============>>>preHandle0:" + ((HandlerMethod) handler).getMethod().getName());
        return true;
    }

    /**
     * 之中执行：Handler 逻辑真正执行完成但尚未返回页面
     * @param request
     * @param response
     * @param handler
     * @param modelAndView
     */
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) {
```

```
            //modelAndView.addObject("nowDate","123");
            //modelAndView.setViewName("error");
            System.out.println("================>>>postHandle0");
        }


        /**
         * 之后执行：返回页面之后执行
         * @param request
         * @param response
         * @param handler
         * @param ex
         */
        @Override
        public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) {
            System.out.println("================>>>afterCompletion0");
        }
}
```

- 配置拦截器

```
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/**"/>
        <bean class="com.springmvc.interceptor.MyHandlerInterceptor"/>
    </mvc:interceptor>
    <mvc:interceptor>
        <mvc:mapping path="/**"/>
        <bean class="com.springmvc.interceptor.MyHandlerInterceptor1"/>
    </mvc:interceptor>
    <mvc:interceptor>
        <mvc:mapping path="/**"/>
        <bean class="com.springmvc.interceptor.MyHandlerInterceptor2"/>
    </mvc:interceptor>
</mvc:interceptors>
```

- 拦截器链（Interceptor Chain）

```
11 Oct 2018 09:45:07.009 信息 [http-nio-8080
===============>>>preHandle1
===============>>>preHandle2
===============>>>preHandle3
===============>>>postHandle3
===============>>>postHandle2
===============>>>postHandle1
===============>>>afterCompletion3
===============>>>afterCompletion2
===============>>>afterCompletion1
```

- 拦截器链执行时，拦截器链正常流程测试

preHandle 按照拦截器配置顺序执行

postHandle 按照拦截器配置倒序执行

afterCompletion 按照拦截器配置倒序执行

- 拦截器链中断流程测试

  拦截器链中有中断时，整个链中的拦截器的 postHandle 都不会执行

拦截器案例（登录控制）

- 需求
  1. 有一个登录页面，写一个 Handler 用于跳转登录页面
  2. 登录页面有一提交表单的动作。需要在 Controller 中处理
     a) 判断用户名密码是否正确（admin/admin）
     b) 如果正确，向 session 中写入用户信息（写入用户名 username）
     c) 跳转到登录成功页面
  3. 开发拦截器
     a) 拦截用户请求，判断用户是否登录（登录页面跳转请求和登录提交请求不能拦截）
     b) 如果用户已经登录则放行
     c) 如果用户未登录则跳转到登录页面
- 实现
  - login.jsp

```
<%@ page isELIgnored="false" contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>xxx 系统登录</title>

    <style>
        div{
            width:300px;
            height:100px;
            position:absolute;
            top:50%;
            left:50%;
```

```
                margin-top:-50px;
                margin-left:-150px;
            }
        </style>
</head>
<body>
        <div>
            <form                                                    method="post"
action="${pageContext.request.contextPath}/user/login.action">
                <table>
                    <tr>
                        <td>用户名</td>
                        <td><input type="text" name="username"/></td>
                    </tr>
                    <tr>
                        <td>密码</td>
                        <td><input   type="password"   name="password"/><input
type="submit" value="登录"/></td>
                    </tr>
                </table>
            </form>
        </div>
</body>
</html>
```

- UserController.java

```
package com.springmvc.controller;

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

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

@Controller
@RequestMapping("user")
public class UserController {

    @RequestMapping("toLogin")
    public String toLogin(Model model) {
        return "login";
    }
```

```
    @RequestMapping("login")
    public String login(Model model, String username, String password,
HttpServletRequest request,HttpSession session) {
        if("admin".equalsIgnoreCase(username) &&
"admin".equalsIgnoreCase(password)) {
            session.setAttribute("username",username);
            return "forward:/default/gotoResult.action";
        }
        return "redirect:toLogin.action";
    }
}
```

- 拦截器

```
package com.springmvc.interceptor;

import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class RequestIntercepter implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        HttpSession httpSession = request.getSession();
        Object username = httpSession.getAttribute("username");
        if(username == null) {
            // 未登录，跳转到登录页面
            response.sendRedirect(request.getContextPath() +
"/user/toLogin.action");
            return false;
        }
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {

    }
```

```
    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {

    }
}
```

```
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/**"/>
        <!--不拦截 user 下的请求-->
        <mvc:exclude-mapping path="/user/**"/>
        <bean class="com.springmvc.interceptor.RequestIntercepter"></bean>
    </mvc:interceptor>
</mvc:interceptors>
```

- web.xml

```
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
  <display-name>Archetype Created Web Application</display-name>

  <servlet>
    <servlet-name>springmvc</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:springmvc.xml</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>springmvc</servlet-name>
    <url-pattern>*.action</url-pattern>
  </servlet-mapping>

  <!--应用启动直接跳转到登录页面-->
  <welcome-file-list>
    <welcome-file>/user/toLogin.action</welcome-file>
  </welcome-file-list>

</web-app>
```

[掌握]SSM 整合

SSM = Spring+SpringMVC+Mybatis

整合策略

SSM = Spring+SpringMVC+Mybatis = （Spring+Mybatis）+SpringMVC
先整合 Spring+Mybatis
然后再整合 SpringMVC

Mybatis 整合 Spring

- 整合目标（目的）
  1. **Mybatis 框架的**数据库连接池以及事务管理都交给 Spring 容器来完成
  2. **Mybatis 框架的** SqlSessionFactory 对象应该放到 Spring 容器中作为单例对象管理
  3. **Mybatis 框架的** Mapper 动态代理对象交给 Spring 管理，我们从 Spring 容器中直接获得 Mapper 的代理对象
- 整合所需 jar 分析
  1. Junit 测试 jar（4.12 版本）
  2. Mybatis 的 jar
  3. Spring 相关 jar（spring-context、spring-test、spring-jdbc、spring-tx、spring-aop、aspectjweaver）
  4. Mybatis/Spring 整合包 jar
     原来这个整合包是由 Spring 提供维护的，后来由 Mybatis 自己维护
     包名是 mybatis-spring 开头的
  5. Mysql 数据库驱动 jar
  6. Dbcp 数据库连接池的 jar
  7. Log4j 日志 jar
- 业务需求：查询商品表的全部数据

- 步骤
  1. Dao 层代码开发
  2. Service 层代码开发
  3. 按照整合目标完成 Mybatis+Spring 框架整合的后续工作

注意：Mybatis 和 Spring 整合之后，在一定程度来说，Mybatis 的全局配置文件不是必须

- 整合后 Pom 坐标

```
<!--junit-->
<dependency>
```

```
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
<!--mybatis-->
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>3.4.5</version>
</dependency>
<!--spring 相关-->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>5.0.2.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-test</artifactId>
  <version>5.0.2.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>5.0.2.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-tx</artifactId>
  <version>5.0.2.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aop</artifactId>
  <version>5.0.2.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjweaver</artifactId>
  <version>1.8.9</version>
</dependency>
<!--mybatis 与 spring 的整合包-->
<dependency>
```

```
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis-spring</artifactId>
  <version>1.3.1</version>
</dependency>
<!--数据库驱动 jar-->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.46</version>
</dependency>
<!--数据库连接池 jar-->
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-dbcp2</artifactId>
  <version>2.2.0</version>
</dependency>
<!--Log4j 日志 jar-->
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.17</version>
</dependency>
```

- 整合后 Mybatis 日志开启，在 SqlMapConfig.xml 文件中配置

```
<settings>
    <setting name="logImpl" value="LOG4J"/>
</settings>
```

- Spring 配置文件
  - applicationContext-dao.xml

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

    <!--数据库连接池配置-->
    <context:property-placeholder location="classpath:db.properties"/>
```

```
    <bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource">
        <property name="driverClassName" value="${jdbc.driver}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>

    <!--配置 SqlSessionFactory-->
    <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="dataSource"/>
        <property name="typeAliasesPackage" value="com.ssm.pojo"/>
        <property name="configLocation" value="classpath:SqlMapConfig.xml"/>
    </bean>

    <!--Spring 管理 Mapper 动态代理对象-->
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <property name="basePackage" value="com.ssm.mapper"/>
        <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"/>
    </bean>
</beans>
```

- applicationContext-service.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx.xsd
">
    <context:component-scan base-package="com.ssm.service"/>
    <!--配置 Spring 事务-->
    <bean                                                id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>
    <tx:annotation-driven transaction-manager="transactionManager"/>
```

```
</beans>
```

- Mybatis 配置文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <!--日志配置，指定为 LOG4J 输出-->
    <settings>
        <setting name="logImpl" value="LOG4J"/>
    </settings>
</configuration>
```

- 测试程序

```
import com.ssm.pojo.Item;
import com.ssm.service.ItemService;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import java.util.List;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations                                    =
{"classpath:applicationContext-dao.xml","classpath:applicationContext-service.x
ml"})
public class MainTest {

    @Autowired
    private ItemService itemService;

    @Test
    public void testSpringMybatis() throws Exception {
        List<Item> itemList = itemService.queryItemList();
        if(itemList != null && itemList.size() > 0) {
            for (int i = 0; i < itemList.size(); i++) {
                Item item =    itemList.get(i);
                System.out.println(item);
            }
        }
    }
```

```
}
```

Mybatis 整合 Spring 后整合 SpringMVC

- 整合思路

  在已有工程基础上，开发 SpringMVC 的入门案例即可

- pom 坐标

```
<!--SpringMVC-->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>5.0.2.RELEASE</version>
</dependency>
<!--jsp-api&servlet-api-->
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>jsp-api</artifactId>
  <version>2.0</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>servlet-api</artifactId>
  <version>2.5</version>
  <scope>provided</scope>
</dependency>
<!--页面使用 jstl 表达式-->
<dependency>
  <groupId>jstl</groupId>
  <artifactId>jstl</artifactId>
  <version>1.2</version>
</dependency>
<dependency>
  <groupId>taglibs</groupId>
  <artifactId>standard</artifactId>
  <version>1.1.2</version>
</dependency>
```

- jsp 页面

```
<%@ page isELIgnored="false" contentType="text/html;charset=UTF-8" language="java" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

```
<html>
<head>
    <title>商品列表页</title>
</head>
<body>
<table border="1" width="100%">
    <tr>
        <td>序号</td>
        <td>商品名称</td>
        <td>商品价格</td>
        <td>商品详情</td>
    </tr>
    <c:forEach items="${itemList}" var="item" varStatus="status">
        <tr>
            <td>${status.count}</td>
            <td>${item.name}</td>
            <td>${item.price}</td>
            <td>${item.detail}</td>
        </tr>
    </c:forEach>
</table>

</body>
</html>
```

- web.xml

```
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
  <display-name>Archetype Created Web Application</display-name>

  <!--配置 Spring 配置文件-->
  <context-param>
    <param-name>contextConfigLocation</param-name>

<param-value>classpath:applicationContext-service.xml,classpath:applicationContext-d
ao.xml</param-value>
  </context-param>

  <!--配置编码过滤器-->
  <filter>
    <filter-name>encoding</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>encoding</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

  <!--配置 Spring 启动监听器-->
  <listener>

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

  <!--SpringMVC 配置-->
  <servlet>
    <servlet-name>springmvc</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:springmvc.xml</param-value>
    </init-param>
  </servlet>

  <servlet-mapping>
    <servlet-name>springmvc</servlet-name>
    <url-pattern>*.action</url-pattern>
  </servlet-mapping>
</web-app>
```

- springmvc.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:context="http://www.springframework.org/schema/context"
       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

http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.x
sd
        http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc.xsd
">

    <!--配置 Controller 扫描-->
<!--
    注意事项：Controller 层的扫描需要配置在 SpringMVC 中，不要配置在
Spring 的扫描中
            让 Spring 只扫描 service
-->
<context:component-scan base-package="com.springmvc.controller"/>


<!--注册组件-->
<mvc:annotation-driven/>


<!--视图解析器配置（视图路径的前后缀配置）-->
<bean
class="org.springframework.web.servlet.view.InternalResourceView
Resolver">
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
</bean>

    <mvc:interceptors>
        <mvc:interceptor>
            <mvc:mapping path="/**"/>
            <!--不拦截 user 下的请求-->
            <mvc:exclude-mapping path="/user/**"/>
            <bean class="com.ssm.interceptor.RequestIntercepter"></bean>
        </mvc:interceptor>
    </mvc:interceptors>
</beans>
```

- ItemController.java

```java
package com.ssm.controller;

import com.ssm.service.ItemService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequestMapping("item")
public class ItemController {

    @Autowired
    private ItemService itemService;

    @RequestMapping("itemList")
    public String itemList(Model model) throws Exception {
        model.addAttribute("itemList",itemService.queryItemList());
        return "itemList";
    }
}
```

- 最后,我们是可以把我们今天第一部分的登录控制添加进来的(略)