SpringMVC

01.SpringMVC概述

1.1SpringMVC是什么

SpringMVC是一种基于Java的实现MVC设计模型的请求驱动类型的轻量级Web框架，属于 Spring FrameWork 的后续产品，已经融合在Spring Web Flow里面。Spring 框架提供了构建 Web 应用程序的全功能 MVC 模块。使用 Spring 可插入的 MVC 架构，从而在使用Spring进行WEB开发时，可以选择使用 Spring 的 Spring MVC 框架或集成其他MVC开发框架，如Struts1(现在一般不用)，Struts2等。

SpringMVC已经成为目前最主流的 MVC 框架之一，并且随着Spring3.0的发布，全面超越 Struts2，成为最优秀的 MVC 框架。

它通过一套注解，让一个简单的Java类成为处理请求的控制器，而无须实现任何接口。同时它还支持RESTful编程风格的请求。

1.2 MVC

1. MVC全名是Model View Controller 模型视图控制器，每个部分各司其职。
2. Model：数据模型，JavaBean的类，用来进行数据封装。
3. View：指JSP、HTML用来展示数据给用户
4. Controller：用来接收用户的请求，整个流程的控制器。用来进行数据校验等。

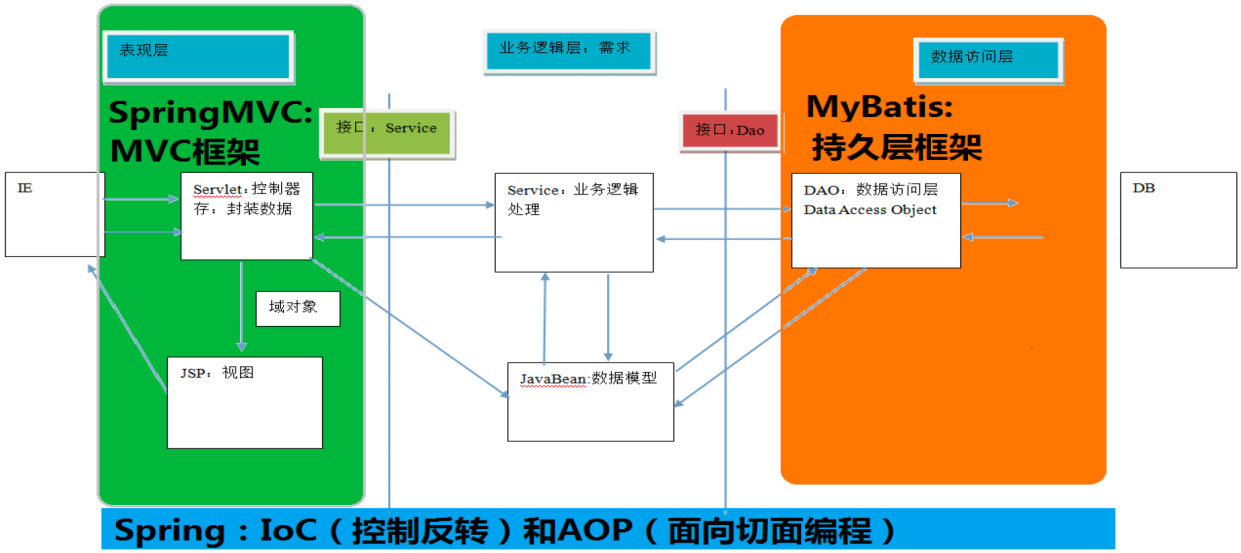
1.3 三层架构

1. 咱们开发服务器端程序，一般都基于两种形式，一种C/S架构程序，一种B/S架构程序
2. 使用Java语言基本上都是开发B/S架构的程序，B/S架构又分成了三层架构
3. 三层架构

   表现层：WEB层，用来和客户端进行数据交互的。表现层一般会采用MVC的设计模型

   业务层：处理公司具体的业务逻辑的

   持久层：用来操作数据库的



02.SpringMVC的入门案例

2.1 环境搭建

1. 创建web项目，引入开发的jar包

```
<!--版本锁定-->
  <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>
    <spring.version>5.1.6.RELEASE</spring.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>${spring.version}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>${spring.version}</version>
    </dependency>
   <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>javax.servlet-api</artifactId>
```

```
      <version>4.0.1</version>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>javax.servlet.jsp</groupId>
      <artifactId>javax.servlet.jsp-api</artifactId>
      <version>2.3.3</version>
      <scope>provided</scope>
    </dependency>

  </dependencies>
```

2. 配置**核心控制器**（DispatcherServlet）

```
 <servlet>
    <servlet-name>dispatcherServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:spring-mvc.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

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

3. 编写springmvc.xml的配置文件

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

    <context:component-scan base-package="com.dgut"/>
```

```
    <!--开启mvc注解-->
    <mvc:annotation-driven/>
</beans>
```

4.编写HelloController控制器类

```
@Controller
public class HelloController {

    @RequestMapping("/test")
    public String test(){
        System.out.println("Hello");
        return "success";
    }
}
```

5. 在WEB-INF目录下创建pages文件夹，编写success.jsp的成功页面

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
你好
</body>
</html>
```

6. 报404错误，原因我们没有配置视图解析器，springmvc不知道**success**是哪个页面

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

2.2 案例执行过程分析

2.2.1执行流程

1. 当启动Tomcat服务器的时候，因为配置了load-on-startup标签，所以会创建DispatcherServlet对象，

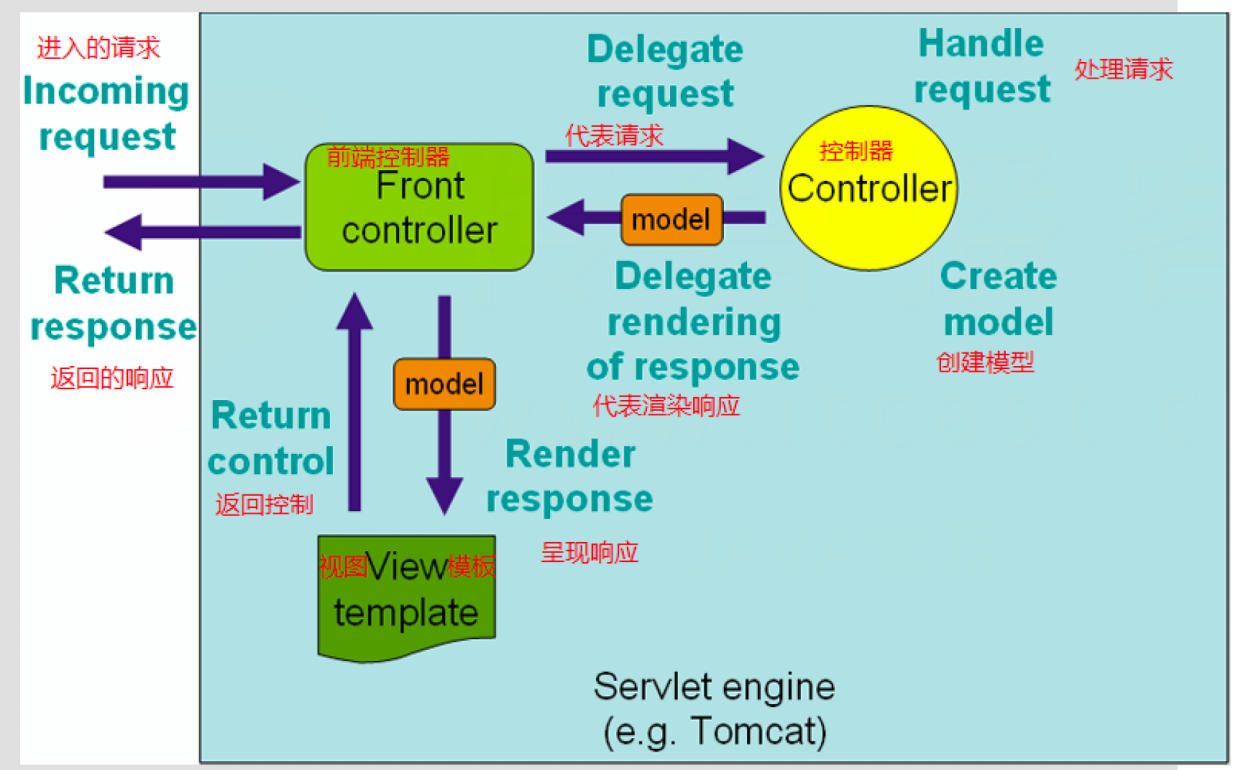
就会加载springmvc.xml配置文件

2. 开启了注解扫描，那么HelloController对象就会被创建
3. 从 **/add** 发送请求，请求会先到达DispatcherServlet核心控制器，根据配置@RequestMapping注解

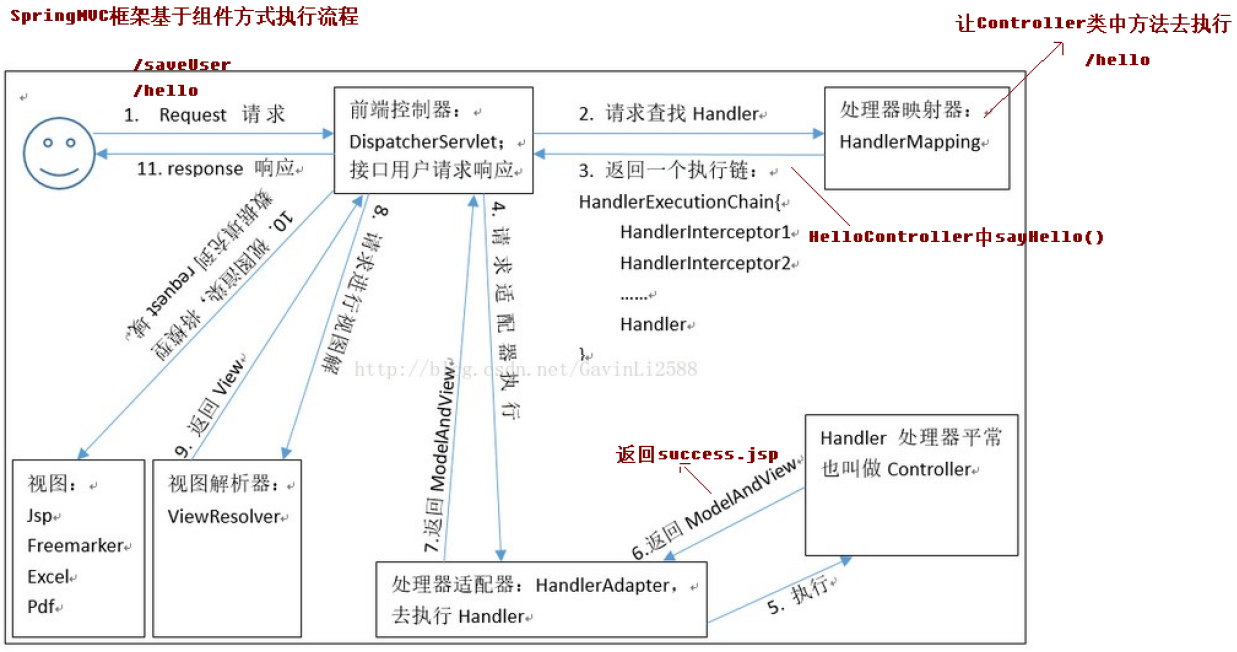
找到执行的具体方法

4. 根据执行方法的返回值，再根据配置的视图解析器，去指定的目录下查找指定名称的JSP文件
5. Tomcat服务器渲染页面，做出响应

2.2.2 **SpringMVC的**请求响应流程



2.3 springMVC组件分析



2.3.1 **DispatcherServlet**：核心控制器 、前端控制器、调度控制器

用户请求到达核心控制器，它就相当于mvc模式中的c，dispatcherServlet是整个流程控制的中心，由它调用其它组件处理用户的请求，dispatcherServlet的存在降低了组件之间的耦合性。

2.3.2 **HandlerMapping**：处理器映射器

HandlerMapping负责根据用户请求找到Handler即处理器，SpringMVC提供了不同的映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

2.3.3 **Handler**：处理器

它就是我们开发中要编写的具体业务控制器。由DispatcherServlet把用户请求转发到Handler。由Handler对具体的用户请求进行处理。

2.3.4 **HandlAdapter**：处理器适配器

通过HandlerAdapter对处理器进行执行，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。

2.3.5 **View Resolver**：视图解析器

View Resolver负责将处理结果生成View视图，View Resolver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，最后对View进行渲染将处理结果通过页面展示给用户。

2.3.6 **<mvc:annotation-driven>**说明

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。

使用<mvc:annotation-driven>自动加载RequestMappingHandlerMapping（处理映射器）和RequestMappingHandlerAdapter（处理适配器），可用在SpringMVC.xml配置文件中使用<mvc:annotation-driven>替代注解处理器和适配器的配置。

2.4 RequestMapping注解

2.4.1 使用说明

```
//源码：
  @Target({ElementType.METHOD, ElementType.TYPE})
  @Retention(RetentionPolicy.RUNTIME)
  @Documented @Mapping public
  @interface RequestMapping { }
//作用：  用于建立请求URL和处理请求方法之间的对应关系。
```

2.4.2 RequestMapping注解可以作用在方法和类上

1. 作用在类上：第一级的访问目录

   请求URL的第一级访问目录。此处不写的话，就相当于应用的根目录。写的话需要以/开头。

   它出现的目的是为了使我们的URL可以按照模块化管理:

   例如：

   账户模块：

   /**account**/add

   /**account**/update

/**account**/delete

订单模块：

/**order**/add

/**order**/update

/**order**/delete

粗体的部分就是把RequsetMappding写在类上，使我们的URL更加精细。

2. 作用在方法上：第二级的访问目录
3. 细节：路径可以不编写 / 表示应用的根目录开始

2.4.3 RequestMapping的属性

1. path 指定请求路径的url
2. value value属性和path属性是一样的
3. mthod 指定该方法的请求方式
4. params 指定限制请求参数的条件

   例如：

   params = {"accountName"}，表示请求参数必须有accountName

   params = {"moeny!=100"}，表示请求参数中money不能是100。
5. headers 发送的请求中必须包含的请求头

03. 请求参数的绑定

3.1 请求参数的绑定说明

3.1.1 绑定的机制

表单中请求参数都是基于key=value的。
SpringMVC绑定请求参数的过程是通过把表单提交请求参数，作为控制器中方法参数进行绑定的。
例如：

```
<a href="account/findAccount?accountId=10">查询账户</a>
```

中请求参数是： accountId=10

```
/**
* 查询账户
* @return
*/
@RequestMapping("/findAccount")
  public String findAccount(Integer accountId) {
  System.out.println("查询了账户。。。。"+accountId); return "success";
}
```

3.1.2 支持的数据类型：

1. 基本类型参数：

包括基本类型和String类型

2. **POJO**类型参数：

包括实体类，以及关联的实体类

3. 数组和集合类型参数：

包括List结构和Map结构的集合（包括数组）

SpringMVC绑定请求参数是自动实现的，但是要想使用，必须遵循使用要求。

3.1.3 使用要求：

如果是基本类型或者**String**类型：

要求我们的参数名称必须和控制器中方法的形参名称保持一致。(严格区分大小写)

如果是**POJO**类型，或者它的关联对象：

要求表单中参数名称和POJO类的属性名称保持一致。并且控制器方法的参数类型是POJO类型。

如果是集合类型,有两种方式：

第一种：

要求集合类型的请求参数必须在POJO中。在表单中请求参数名称要和POJO中集合属性名称相同。

给List集合中的元素赋值，使用下标。

给Map集合中的元素赋值，使用键值对。

第二种：

接收的请求参数是json格式数据。需要借助一个注解实现。

3.2 基本数据类型和字符串类型

jsp代码：

```
<!-- 基本类型示例 -->
<a href="account/findAccount?accountId=10&accountName=zhangsan">查询账户</a>
```

控制器代码：

```
/** * 查询账户
* @return
* */
  @RequestMapping("/findAccount")
  public String findAccount(Integer accountId,String accountName) {
    System.out.println("查询了账户。。。。"+accountId+","+accountName);
    return "success";
  }
```

3.3 实体类型（JavaBean）

实体类代码：

```
/**
* 账户信息
*/
```

```
public class Account implements Serializable {
  private Integer id;
  private String name;
  private Float money;
  private Address address;
  getters and setters
  }
/**
* 地址的实体类
*/
public class Address implements Serializable
{
  private String provinceName;
  private String cityName;
  //getters and setters
  }

jsp代码:
<!-- pojo类型演示 -->
 <form action="account/saveAccount" method="post">
 账户名称:<input type="text" name="name" ><br/>
 账户金额:<input type="text" name="money" ><br/>
 账户省份:<input type="text" name="address.provinceName" ><br/>
 账户城市:<input type="text" name="address.cityName" ><br/>
 <input type="submit" value="保存"> </form>

  控制器代码:
  /**
  * 保存账户
  */
  @RequestMapping("/saveAccount")
  public String saveAccount(Account account) {
      System.out.println("保存了账户。。。。"+account); return "success";
}
```

3.4 集合属性数据封装

```
//实体类代码
public class User implements Serializable {
    private String username;
    private String password;
    private Integer age;
    private List<Account> accounts;
    private Map<String, Account> accountMap;
    //getters and setters
  }
/**
* 地址的实体类
*/
```

```
public class Address implements Serializable {
  private String provinceName;
  private String cityName;
  //getters and setters
}

jsp:
<form action="account/updateAccount" method="post">
用户名称：<input type="text" name="username" ><br/>
用户密码：<input type="password" name="password" ><br/>
用户年龄：<input type="text" name="age" ><br/>
账户1名称：<input type="text" name="accounts[0].name" ><br/>
账户1金额：<input type="text" name="accounts[0].money" ><br/>
账户2名称：<input type="text" name="accounts[1].name" ><br/>
账户2金额：<input type="text" name="accounts[1].money" ><br/>
账户3名称：<input type="text" name="accountMap['one'].name" ><br/>
账户3金额：<input type="text" name="accountMap['one'].money" ><br/>
账户4名称：<input type="text" name="accountMap['two'].name" ><br/>
账户4金额：<input type="text" name="accountMap['two'].money" ><br/>
<input type="submit" value="保存">
</form>

控制器：
RequestMapping("/saveAccount")
public String saveAccount(Account account){
    System.out.println("查询了账户。。。。"+account);
    return "hello";
}
```

3.5 请求参数中文乱码的解决

1. 然后在 Server > VM options 设置为 -Dfile.encoding=UTF-8 ， 重启tomcat 解决控制台中文乱码
2. 添加过滤器

```
<!-- 配置过滤器，解决中文乱码的问题 -->
<filter>
        <filter-name>CharacterEncodingFilter</filter-name>
        <filter-
class>org.springframework.web.filter.CharacterEncodingFilter</filter-
class>
       <!-- 设置过滤器中的属性值 -->
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
        <!-- 启动过滤器 -->
        <init-param>
```

```
            <param-name>forceEncoding</param-name>
            <param-value>true</param-value>
        </init-param>
    </filter>

  <!-- 过滤所有请求 -->
    <filter-mapping>
        <filter-name>CharacterEncodingFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
```

3.6 自定义类型转换器

```
jsp:
<a href="account/deleteAccount?date=2018-01-01">根据日期删除账户</a>

@RequestMapping("/deleteAccount")
public String deleteAccount(Date date)
{
  System.out.println("删除了账户。。。。"+date);
 return "success";
}
```

```
Failed to convert value of type 'java.lang.String' to required type 'java.util.Date';
```

使用步骤：

1. 定义一个类，实现**Converter**接口，该接口有两个泛型。

```
//interface Converter<S, T> {//S:表示接受的类型，T：表示目标类型
public class StringToDateConverter implements Converter<String, Date> {

    /**
     * 用于把String类型转成日期类型
     */
    @Override
    public Date convert(String source) {
        DateFormat format = null;
        try {
            if (StringUtils.isEmpty(source)) {
                throw new NullPointerException("请输入要转换的日期");
            }
            format = new SimpleDateFormat("yyyy-MM-dd");
            Date date = format.parse(source);
            return date;
        } catch (Exception e) {
```

```
            throw new RuntimeException("输入日期有误");
        }
    }
}
```

2. 在**spring**配置文件中配置类型转换器

```
 <!-- 配置类型转换器工厂 -->
    <bean id="conversionService"
class="org.springframework.context.support.ConversionServiceFactoryBean">
        <!-- 给工厂注入一个新的类型转换器 -->
        <property name="converters">
            <set>
                <!-- 配置自定义类型转换器 -->
                <bean class="com.dgut.converter.StringToDateConverter"/>
            </set>
        </property>
    </bean>
```

3. 在**annotation-driven**标签中引用配置的类型转换服务

```
<!--开启mvc注解-->
<mvc:annotation-driven conversion-service="conversionService"/>
```

3.7使用ServletAPI对象作为方法参数

SpringMVC还支持使用原始ServletAPI对象作为控制器方法的参数。支持原始ServletAPI对象有：

HttpServletRequest

HttpServletResponse

HttpSession

java.security.Principal

Locale

InputStream

OutputStream

Reader

Writer

我们可以把上述对象，直接写在控制的方法参数中使用。

```
<a href="account/testServletAPI">测试访问ServletAPI</a>

@RequestMapping("/testServletAPI")
public String testServletAPI(HttpServletRequest request, HttpServletResponse
response, HttpSession session) {
  System.out.println(request);
  System.out.println(response);
  System.out.println(session);
  return "success";
}
```

04. 常用的注解

4.1 @RequestParam

作用： 把请求中指定名称的参数给控制器中的形参赋值。

属性：

value：请求参数中的名称。

required：请求参数中是否必须提供此参数。默认值：true。表示必须提供，如果不提供将报错。

```
@RequestMapping("/useRequestParam")
public String useRequestParam(@RequestParam("name")String username,
@RequestParam(value="age",required=false)Integer age){
  System.out.println(username+","+age);
  return "success";
}
```

4.2 @RequestBody

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。

```
post请求jsp代码：
<!-- request body注解 -->
<form action="springmvc/useRequestBody" method="post">
  用户名称：<input type="text" name="username" ><br/>
  用户密码：<input type="password" name="password" ><br/>
  用户年龄：<input type="text" name="age" ><br/>
  <input type="submit" value="保存">
 </form>
```

```
 get请求jsp代码：  <a href="springmvc/useRequestBody?body=test">requestBody注解get请求</a>

@RequestMapping("/useRequestBody")
public String useRequestBody(@RequestBody(required=false) String body){
  System.out.println(body); return "success";
}
```

4.3 @PathVaribale

作用：

用于绑定url中的占位符。例如：请求url中 /delete/**{id}**，这个**{id}**就是url占位符。

url支持占位符是spring3.0之后加入的。是springmvc支持rest风格URL的一个重要标志。

属性：

value：用于指定url中占位符名称。

required：是否必须提供占位符。

```
jsp代码：  <!-- PathVariable注解 -->
<a href="account/usePathVariable/100">pathVariable注解</a>

 @RequestMapping("/usePathVariable/{id}")
    public String usePathVariable(@PathVariable("id") Integer id) {
        System.out.println(id);
        return "hello";
   }
```

4.3.1 REST风格URL

REST（英文：Representational State Transfer，简称REST）描述了一个架构样式的网络系统，比如 web 应用程序。它首次出现在 2000 年 Roy Fielding 的博士论文中，他是 HTTP 规范的主要编写者之一。在目前主流的三种Web服务交互方案中，REST相比于SOAP（Simple Object Access protocol，简单对象访问协议）以及XML-RPC更加简单明了，无论是对URL的处理还是对Payload的编码，REST都倾向于用更加简单轻量的方法设计和实现。值得注意的是REST并没有一个明确的标准，而更像是一种设计的风格。

1. 请求路径一样，可以根据不同的请求方式去执行后台的不同方法
2. restful风格的URL优点
    1. 结构清晰
    2. 符合标准
    3. 易于理解
    4. 扩展方便

> **restful**的示例：

> /account/1 HTTP GET： 得到 id = 1 的 account
>
> /account/1 HTTP DELETE： 删除 id = 1的 account
>
> /account/1 HTTP PUT： 更新id = 1的 account
>
> /account HTTP POST： 新增 account

4.4 @RequestMapping

@GetMapping @PostMapping @PutMapping ....

用于建立请求URL和处理请求方法之间的对应关系。

属性：

value：用于指定请求的URL。它和path属性的作用是一样的。

method：用于指定请求的方式。

params：用于指定限制请求参数的条件。它支持简单的表达式。要求请求参数的key和value必须和配置的一模一样。

例如：

params = {"accountName"}，表示请求参数必须有accountName

params = {"moeny!=100"}，表示请求参数中money不能是100。

headers：用于指定限制请求消息头的条件。

注意：

以上四个属性只要出现2个或以上时，他们的关系是与的关系。

05.响应数据和结果视图

5.1 返回值分类

5.1.1 字符串

controller方法返回字符串可以指定逻辑视图名，通过视图解析器解析为物理视图地址。

```
//指定逻辑视图名，经过视图解析器解析为jsp物理路径：/WEB-INF/pages/success.jsp
@RequestMapping("/testReturnString")
public String testReturnString() {
  System.out.println("AccountController的testReturnString 方法执行了。。。。");
  return "success";
}
```

5.1.2 void

```
@RequestMapping("/testReturnVoid")
public void testReturnVoid(HttpServletRequest request,HttpServletResponse
response) throws Exception {
}
在controller方法形参上可以定义request和response，使用request或response指定响应结果：
//1、使用request转向页面，如下
  request.getRequestDispatcher("/WEB-INF/pages/success.jsp").forward(request,
response);
//2、通过response页面重定向：
  response.sendRedirect("testRetrunString")
//3、通过response指定响应结果，例如响应json数据：
response.setCharacterEncoding("utf-8");
response.setContentType("application/json;charset=utf-8");
response.getWriter().write("json串");
```

5.1.3 **ModelAndView**

ModelAndView是SpringMVC为我们提供的一个对象，该对象也可以用作控制器方法的返回值。

```
@RequestMapping("/testReturnModelAndView")
public ModelAndView testReturnModelAndView() {
    ModelAndView mv = new ModelAndView();
    mv.addObject("username", "张三");
    mv.setViewName("success");
    return mv;
}

//jsp页面
<body> 执行成功！  ${requestScope.username} </body>
```

注意：

我们在页面上上获取使用的是requestScope.username取的，所以返回ModelAndView类型时，浏览器跳转只能是请求转发。

5.2 转发和重定向

5.2.1 **forward**转发

controller方法在提供了String类型的返回值之后，默认就是请求转发。我们也可以写成：

```
/**
* 转发
* @return
*/ @RequestMapping("/testForward")
public String testForward() {
  System.out.println("AccountController的testForward 方法执行了。。。。");
  return "forward:/WEB-INF/pages/success.jsp";
  }

//需要注意的是，如果用了formward：则路径必须写成实际视图url，不能写逻辑视图。 它相当于
//“request.getRequestDispatcher("url").forward(request,response)”。
//使用请求转发，既可以转发到jsp，也可以转发到其他的控制器方法。
```

5.2.2 **Redirect**重定向

contrller方法提供了一个String类型返回值之后，它需要在返回值里使用:redirect:

```
/**
* 重定向
* @return */
@RequestMapping("/testRedirect") public String testRedirect() {
    System.out.println("AccountController的testRedirect 方法执行了。。。。");
    return "redirect:testReturnModelAndView";
    }
//它相当于“response.sendRedirect(url)”。
//需要注意的是，如果是重定向到jsp页面，则jsp页面不能写在WEB-INF目录中，否则无法找到。
```

5.3 @ResponeBody

作用：

该注解用于将Controller的方法返回的对象，通过HttpMessageConverter接口转换为指定格式的数据如：json,xml等，通过Response响应给客户端

使用@ResponseBody注解实现将controller方法返回对象转换为json响应给客户端。

Springmvc默认用MappingJackson2HttpMessageConverter对json数据进行转换，需要加入jackson的包。

(原理：mvc:annotation-driver/ 因为annotation-driver是被AnnotationDrivenBeanDefinitionParser解析，查看源码)

```
<dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-databind</artifactId>
      <version>2.9.9.3</version>
    </dependency>
```

```
@RequestMapping("/testResponseJson")
public @ResponseBody Account testResponseJson(@RequestBody Account account) {
  System.out.println("异步请求：  "+account); return account;
  }
```

过滤js等资源

```
  <!--前端控制器哪些静态资源不需要拦截-->
    <mvc:resources mapping="/js/**" location="/js/" />
   <mvc:resources mapping="/img/**" location="/img/" />

//方式二，在web.xml配置
  <servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>*.js</url-pattern>
  </servlet-mapping>
```

06. SpringMVC实现文件上传

6.1 文件上传分析

6.1.1 文件上传的必要前提

1. form表单的**enctype**取值必须是：**multipart/form-data**

(默认值是:application/x-www-form-urlencoded)

enctype:是表单请求正文的类型 2. method属性取值必须是Post

3. 提供一个文件选择域<input type="file" />

6.1.2 文件上传的原理分析

```
当form表单的enctype取值不是默认值后，request.getParameter()将失效。
enctype=”application/x-www-form-urlencoded”时,

form表单的正文内容是：  key=value&key=value&key=value

当form表单的enctype取值为Mutilpart/form-data时，请求正文内容就变成：

每一部分都是MIME类型描述的正文
-----------------------------7de1a433602ac          分界符
Content-Disposition: form-data; name="userName"     协议头
aaa                                                 协议的正文
-----------------------------7de1a433602ac
Content-Disposition: form-data; name="file";
filename="C:\Users\zhy\Desktop\fileupload_demofile\b.txt"
Content-Type: text/plain                                协议的类型（MIME类型）
```

```
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
----------------------------7de1a433602ac--
```

6.1.3 借助第三方组件实现文件上传

使用Commons-fileupload组件实现文件上传，需要导入该组件相应的支撑jar包：Commons-fileupload和commons-io。commons-io 不属于文件上传组件的开发jar文件，但Commons-fileupload 组件从1.1 版本开始，它工作时需要commons-io包的支持。

```
<!-- https://mvnrepository.com/artifact/commons-fileupload/commons-fileupload
-->
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.4</version>
</dependency>
```

6.2 springmvc文件上传实现

1. 添加pom文件

```
<dependency>
      <groupId>commons-fileupload</groupId>
      <artifactId>commons-fileupload</artifactId>
      <version>1.4</version>
    </dependency>

    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>${spring.version}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>${spring.version}</version>
    </dependency>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>javax.servlet-api</artifactId>
```

```
      <version>4.0.1</version>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>javax.servlet.jsp</groupId>
      <artifactId>javax.servlet.jsp-api</artifactId>
      <version>2.3.3</version>
      <scope>provided</scope>
    </dependency>
```

2. 配置下web.xml

```
<servlet>
    <servlet-name>dispatcherServlet</servlet-name>
    <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:spring-mvc.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

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

3. 编写jsp界面

```
<form action="/fileUpload" method="post" enctype="multipart/form-
data">
    名称：<input type="text" name="picname"/><br/>
    图片：<input type="file" name="uploadFile"/><br/>
    <input type="submit" value="上传"/></form>
</body>
```

4. 编写控制器

```
@Controller
public class FileUploadController {

    @RequestMapping(value = "fileUpload", method = RequestMethod.POST)
```

```
    public String fileUpload(String picname, MultipartFile uploadFile,
HttpServletRequest request) throws IOException {
        //定义文件名
        String fileName = "";
        //1.获取文件名
        String uploadFileName = uploadFile.getOriginalFilename();
        //2.截取文件名
        String extendName =
uploadFileName.substring(uploadFileName.lastIndexOf(".") + 1);
        //3.把文件加上随机数，防止文件重复
        String uuid = UUID.randomUUID().toString().replace("-",
"").toUpperCase();
        //4.判断是否输入了文件名
        if (!StringUtils.isEmpty(picname)) {
            fileName = uuid + "_" + picname + "." + extendName;
        } else {
            fileName = uuid + "_" + uploadFileName;
        }
        System.out.println(fileName);

        ServletContext context = request.getServletContext();
        String basePath = context.getRealPath("/uploads");
        //3.解决同一文件夹中文件过多问题
        String datePath = new SimpleDateFormat("yyyy-MM-
dd").format(new Date());
        //4.判断路径是否存在
        File file = new File(basePath + "/" + datePath);
        if (!file.exists()) {
            file.mkdirs();
        }
        uploadFile.transferTo(new File(file, fileName));
        return "success";
    }
}
```

5. 配置文件解析器

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       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">

    <context:component-scan base-package="com.dgut"/>

    <bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolv
er">
        <property name="prefix" value="/WEB-INF//pages/"/>
        <property name="suffix" value=".jsp"/>
    </bean>
    <!-- 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.CommonsMultipartResol
ver">
        <!-- 设置上传文件的最大尺寸为5MB -->
        <property name="maxUploadSize" value="5242880"/>
    </bean>

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

注意：

文件上传的解析器**id**是固定的，不能起别的名称，否则无法实现请求参数的绑定。（不光是文件，其他字段也将无法绑定）

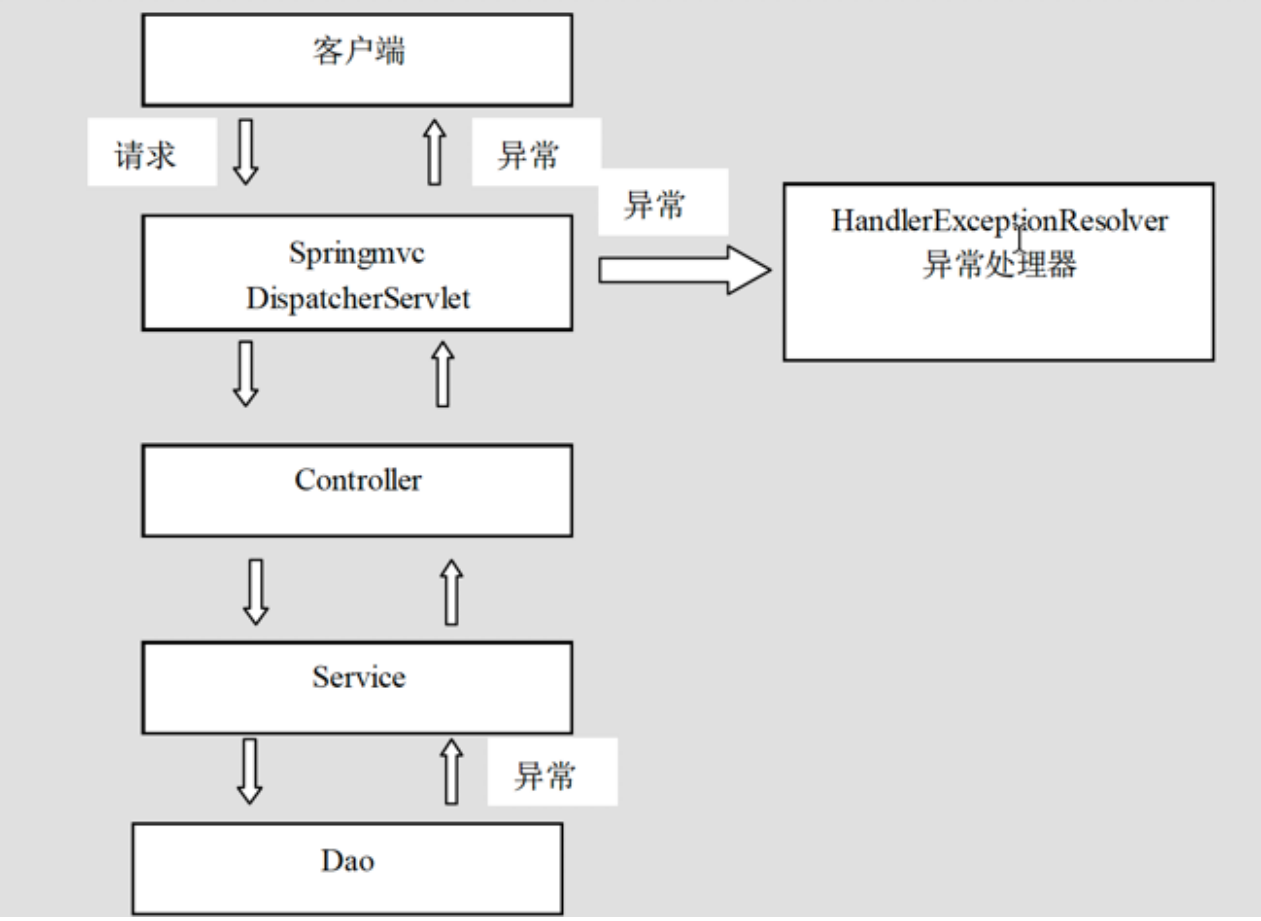
6.3 文件下载

```
 @RequestMapping("/download")
public void download(String name, HttpServletRequest request,
HttpServletResponse response) throws IOException {
    String realPath = request.getServletContext().getRealPath("/upload");
    File file = new File(realPath+File.separator+"/2019-09-16/"+name);
    InputStream fis = new FileInputStream(file);
    response.setHeader("content-disposition","attachment;filename="+name);
    //4.将输入流的数据写出到输出流中
    ServletOutputStream sos = response.getOutputStream();
    byte[] buff = new byte[1024 * 8];
    int len = 0;
    while((len = fis.read(buff)) != -1){
        sos.write(buff,0,len);
    }
    fis.close();
}
```

07. SpringMVC中的异常处理

系统中异常包括两类：预期异常和运行时异常RuntimeException，前者通过捕获异常从而获取异常信息，后者主要通过规范代码开发、测试通过手段减少运行时异常的发生。

系统的dao、service、controller出现都通过throws Exception向上抛出，最后由springmvc前端控制器交由异常处理器进行异常处理，如下图：



7.1 实现步骤

7.1.1 编写异常类

```
public class CustomException extends Exception {
    private String message;

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

    public void setMessage(String message) {
        this.message = message;
    }

    public CustomException(String message) {
```

```
        this.message = message;
    }

    public CustomException() {
    }
}
```

7.1.2 编写异常页面

```
<%@ page contentType="text/html;charset=UTF-8" language="java"
isELIgnored="false" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
执行失败！ ${message }
</body>
</html>
```

7.1.3 自定义异常处理器

```
public class CustomExceptionResolver implements HandlerExceptionResolver {
    @Override
    public ModelAndView resolveException(HttpServletRequest
httpServletRequest, HttpServletResponse httpServletResponse, Object o,
Exception e) {
        e.printStackTrace();
        CustomException customException = null;
        //如果抛出的是系统自定义异常则直接转换
        if (e instanceof CustomException) {
            customException = (CustomException) e;
        } else { //如果抛出的不是系统自定义异常则重新构造一个系统错误异常。
            customException = new CustomException("系统错误，请与系统管理 员联
系！");
        }

        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("message", customException.getMessage());
        modelAndView.setViewName("error");
        return modelAndView;
    }
}
```

7.1.4 配置异常处理器

```
    <bean class="com.dgut.CustomExceptionResolver" id="exceptionResolver"/>
```

7.1.5 测试

```
@RequestMapping("/testException")
    public String testException() throws CustomException {
        try {
            int a = 1 / 0;
        }catch (Exception e){
            throw new CustomException(e.getMessage());
        }
        return "success";
    }
```

08. 自定义拦截器

8.1 拦截器的作用

> 拦截器的作用
>
> Spring MVC 的处理器拦截器类似于Servlet开发中的过滤器Filter，用于对处理器进行预处理和后处理。
>
> 用户可以自己定义一些拦截器来实现特定的功能。
>
> 谈到拦截器，还要向大家提一个词——拦截器链（Interceptor Chain）。拦截器链就是将拦截器按一定的顺序联结成一条链。在访问被拦截的方法或字段时，拦截器链中的拦截器就会按其之前定义的顺序被调用。
>
> 说到这里，可能大家脑海中有了一个疑问，这不是我们之前学的过滤器吗？是的它和过滤器是有几分相似，但是也有区别，接下来我们就来说说他们的区别：
>
> 过滤器是servlet规范中的一部分，任何java web工程都可以使用。
>
> 拦截器是SpringMVC框架自己的，只有使用了SpringMVC框架的工程才能用。
>
> 过滤器在url-pattern中配置了**/***之后，可以对所有要访问的资源拦截。
>
> 拦截器它是只会拦截访/它也是AOP思想的具体应用。
>
> 我们要想自定义拦截器， 要求必须实现：**HandlerInterceptor**接口。

8.2 自定义拦截器的步骤

8.2.1 编写一个普通类实现**HandlerInterceptor**接口

```
public class MyHandlerInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        System.out.println("MyHandlerInterceptor preHandle");
        return true;
```

```
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
        System.out.println("MyHandlerInterceptor postHandle");
    }

    @Override
    public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex) throws Exception {
        System.out.println("MyHandlerInterceptor afterCompletion");
    }
}
```

8.2.2 配置拦截器

```
<mvc:interceptors>
        <mvc:interceptor>
            <mvc:mapping path="/**"/>
            <bean class="com.dgut.interceptor.MyHandlerInterceptor"/>
        </mvc:interceptor>
    </mvc:interceptors>
 <mvc:annotation-driven/>
```

8.2.3 测试结果

8.3 拦截器细节

8.3.1 拦截器的放行

放行的含义是指，如果有下一个拦截器就执行下一个，如果该拦截器处于拦截器链的最后一个，则执行控制器中的方法。

8.3.2 拦截器中方法的说明

```
/**
* 如何调用：
* 按拦截器定义顺序调用
* 何时调用：
* 只要配置了都会调用
* 有什么用：
* 如果程序员决定该拦截器对请求进行拦截处理后还要调用其他的拦截器，或者是业务处理器去
* 进行处理，则返回true。
* 如果程序员决定不需要再调用其他的组件去处理请求，则返回false。 */
```

```
default boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
  return true;
}

/**
* 如何调用：
* 按拦截器定义逆序调用
* 何时调用：
* 在拦截器链内所有拦截器返成功调用
* 有什么用：
* 在业务处理器处理完请求后，但是DispatcherServlet向客户端返回响应前被调用，
* 在该方法中对用户请求request进行处理。
*/
default void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, @Nullable ModelAndView modelAndView) throws
Exception {
}

/**
* 如何调用：
* 按拦截器定义逆序调用
* 何时调用：
* 只有preHandle返回true才调用
* 有什么用：
* 在 DispatcherServlet 完全处理完请求后被调用，
* 可以在该方法中进行一些资源清理的操作。
*/
default void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, @Nullable Exception ex) throws Exception { } }
```

8.3.3 多个拦截器是按照配置的顺序决定的。

8.4 拦截器案例（验证用户是否登录）

8.4.1 需求

```
1、有一个登录页面，需要写一个controller访问页面

2、登录页面有一提交表单的动作。需要在controller中处理。
  2.1、判断用户名密码是否正确
  2.2、如果正确 向session中写入用户信息
  2.3、返回登录成功。

3、拦截用户请求，判断用户是否登录
  3.1、如果用户已经登录。放行
  3.2、如果用户未登录，跳转到登录页面
```

1. 新建user控制器

```
@Controller
public class UserController {

    @RequestMapping("/")
    public String index(){
        //后台首页
        return "main";
    }
    @RequestMapping("/login")
    public String login(){
      //登录界面
        return "login";
    }
    @RequestMapping("/loginSubmit")
    public String loginSubmit(String username, HttpSession session){
        //向session记录用户身份信息
        session.setAttribute("user",username);
        return "redirect:/";
    }
    @RequestMapping("/logout")
    public String logout(HttpSession session){
        //session过期
        session.invalidate();
        return "redirect:login";
    }
}
```

2. 拦截器代码

```
public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        //如果是登录页面则放行
        if (request.getRequestURI().indexOf("login") >= 0) {
            return true;
        }
        HttpSession session = request.getSession();
        //如果用户已登录也放行
        if (session.getAttribute("user") != null) {
            return true;
        }
        //用户没有登录则跳转到登录页面
        request.getRequestDispatcher("/WEB-
INF/pages/login.jsp").forward(request, response);
        return false;
    }
```

3. jsp界面

登录界面login.jsp

```
<form action="loginSubmit" method="post">
    用户名:<input type="text" name="username">
    <input type="submit">
</form>
```

后台界面main.jsp

```
<%@ page contentType="text/html;charset=UTF-8" language="java"
isELIgnored="false" %>
<html>
<head>
    <title>h</title>
</head>
<body>
这里是后台首页
${sessionScope.get("user")}
<a href="logout">退出登录</a>
</body>
</html>
```