



Spring MVC 第一天

第1章 SpringMVC 的基本概念

1.1 关于三层架构和 MVC

1.1.1 三层架构

我们的开发架构一般都是基于两种形式，一种是 C/S 架构，也就是客户端/服务器，另一种是 B/S 架构，也就是浏览器服务器。在 JavaEE 开发中，几乎全都是基于 B/S 架构的开发。那么在 B/S 架构中，系统标准的三层架构包括：表现层、业务层、持久层。三层架构在我们的实际开发中使用的非常多，所以我们课程中的案例也都是基于三层架构设计的。

三层架构中，每一层各司其职，接下来我们就说说每层都负责哪些方面：

表现层：

也就是我们常说的 web 层。它负责接收客户端请求，向客户端响应结果，通常客户端使用 http 协议请求 web 层，web 需要接收 http 请求，完成 http 响应。

表现层包括展示层和控制层：控制层负责接收请求，展示层负责结果的展示。

表现层依赖业务层，接收到客户端请求一般会调用业务层进行业务处理，并将处理结果响应给客户端。

表现层的设计一般都使用 MVC 模型。（MVC 是表现层的设计模型，和其他层没有关系）

业务层：

也就是我们常说的 service 层。它负责业务逻辑处理，和我们开发项目的需求息息相关。web 层依赖业务层，但是业务层不依赖 web 层。

业务层在业务处理时可能会依赖持久层，如果要对数据持久化需要保证事务一致性。（也就是我们说的，事务应该放到业务层来控制）

持久层：

也就是我们常说的 dao 层。负责数据持久化，包括数据层即数据库和数据访问层，数据库是对数据进行持久化的载体，数据访问层是业务层和持久层交互的接口，业务层需要通过数据访问层将数据持久化到数据库中。通俗的讲，持久层就是和数据库交互，对数据库表进行增删改查的。

1.1.2 MVC 模型

MVC 全名是 Model View Controller，是模型(model)－视图(view)－控制器(controller)的缩写，是一种用于设计创建 Web 应用程序表现层的模式。MVC 中每个部分各司其职：

Model（模型）：

通常指的就是我们的数据模型。作用一般情况下用于封装数据。

View（视图）：

通常指的就是我们的 jsp 或者 html。作用一般就是展示数据的。

通常视图是依据模型数据创建的。



Controller（控制器）：

是应用程序中处理用户交互的部分。作用一般就是处理程序逻辑的。

它相对于前两个不是很好理解，这里举个例子：

例如：

我们要保存一个用户的信息，该用户信息中包含了姓名，性别，年龄等等。

这时候表单输入要求年龄必须是 1~100 之间的整数。姓名和性别不能为空。并且把数据填充到模型之中。

此时除了 js 的校验之外，服务器端也应该有数据准确性的校验，那么校验就是控制器的该做的。

当校验失败后，由控制器负责把错误页面展示给使用者。

如果校验成功，也是控制器负责把数据填充到模型，并且调用业务层实现完整的业务需求。

1.2 SpringMVC 概述

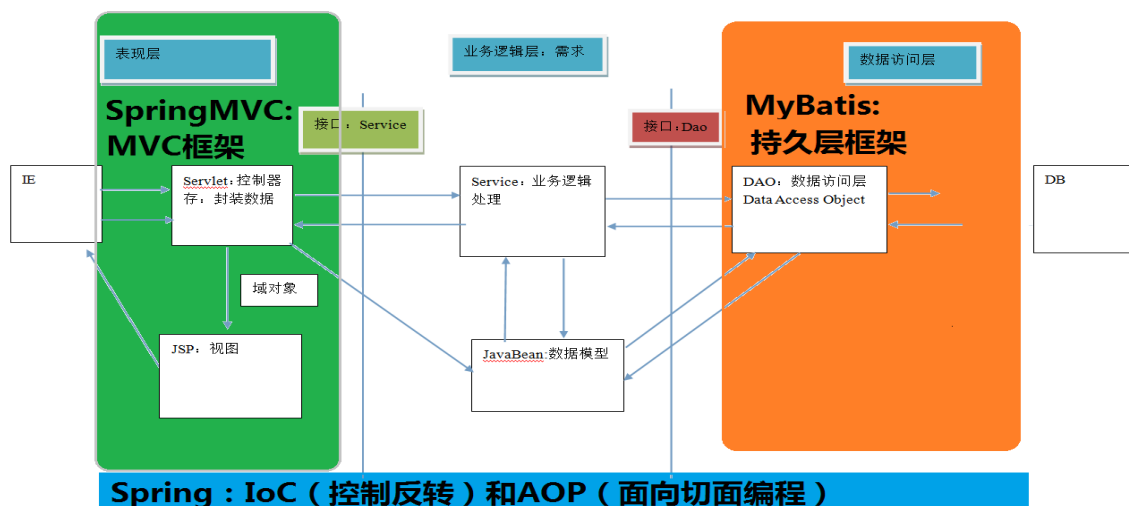
1.2.1 SpringMVC 是什么

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。

1.2.2 SpringMVC 在三层架构的位置





1.2.3 SpringMVC 的优势

1、清晰的角色划分：

前端控制器（DispatcherServlet）

请求到处理器映射（HandlerMapping）

处理器适配器（HandlerAdapter）

视图解析器（ViewResolver）

处理器或页面控制器（Controller）

验证器（Validator）

命令对象（Command 请求参数绑定到的对象就叫命令对象）

表单对象（Form Object 提供给表单展示和提交到的对象就叫表单对象）。

2、分工明确，而且扩展点相当灵活，可以很容易扩展，虽然几乎不需要。

3、由于命令对象就是一个 POJO，无需继承框架特定 API，可以使用命令对象直接作为业务对象。

4、和 Spring 其他框架无缝集成，是其它 Web 框架所不具备的。

5、可适配，通过 HandlerAdapter 可以支持任意的类作为处理器。

6、可定制性，HandlerMapping、ViewResolver 等能够非常简单的定制。

7、功能强大的数据验证、格式化、绑定机制。

8、利用 Spring 提供的 Mock 对象能够非常简单的进行 Web 层单元测试。

9、本地化、主题的解析的支持，使我们更容易进行国际化和主题的切换。

10、强大的 JSP 标签库，使 JSP 编写更容易。

.....还有比如 RESTful 风格的支持、简单的文件上传、约定大于配置的契约式编程支持、基于注解的零配置支持等等。

1.2.4 SpringMVC 和 Struts2 的优略分析

共同点：

它们都是表现层框架，都是基于 MVC 模型编写的。

它们的底层都离不开原始 ServletAPI。

它们处理请求的机制都是一个核心控制器。

区别：

Spring MVC 的入口是 Servlet，而 Struts2 是 Filter

Spring MVC 是基于方法设计的，而 Struts2 是基于类，Struts2 每次执行都会创建一个动作类。所以 Spring MVC 会稍微比 Struts2 快些。

Spring MVC 使用更加简洁，同时还支持 JSR303，处理 ajax 的请求更方便

(JSR303 是一套 JavaBean 参数校验的标准，它定义了很多常用的校验注解，我们可以直接将这些注解加在我们 JavaBean 的属性上面，就可以在需要校验的时候进行校验了。)

Struts2 的 OGNL 表达式使页面的开发效率相比 Spring MVC 更高些，但执行效率并没有比 JSTL 提升，尤其是 struts2 的表单标签，远没有 html 执行效率高。



第2章 SpringMVC 的入门

2.1 SpringMVC 的入门案例

2.1.1 前期准备

下载开发包: <https://spring.io/projects>

其实 spring mvc 的 jar 包就在之前我们的 spring 框架开发包中。

创建一个 javaweb 工程

New Dynamic Web Project

Dynamic Web Project

Name cannot be empty.

Project name:

Project location

☒ Use default location

Location:

Target runtime

Dynamic web module version

Configuration

Hint: Get started quickly by selecting one of the pre-defined project configurations.

EAR membership

☐ Add project to an EAR

EAR project name:

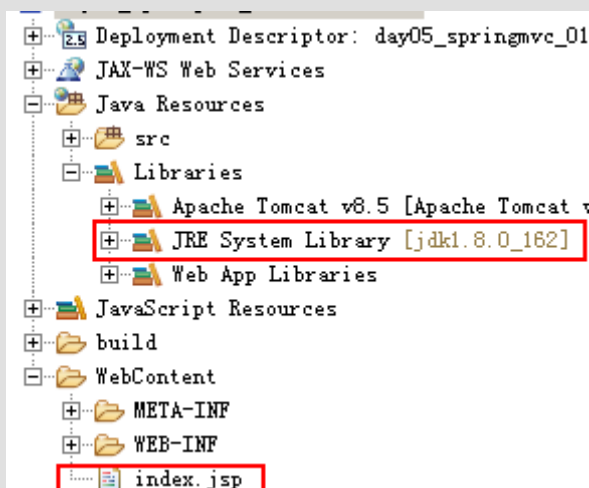
Working sets

☐ Add project to working sets

Working sets:



创建一个 jsp 用于发送请求

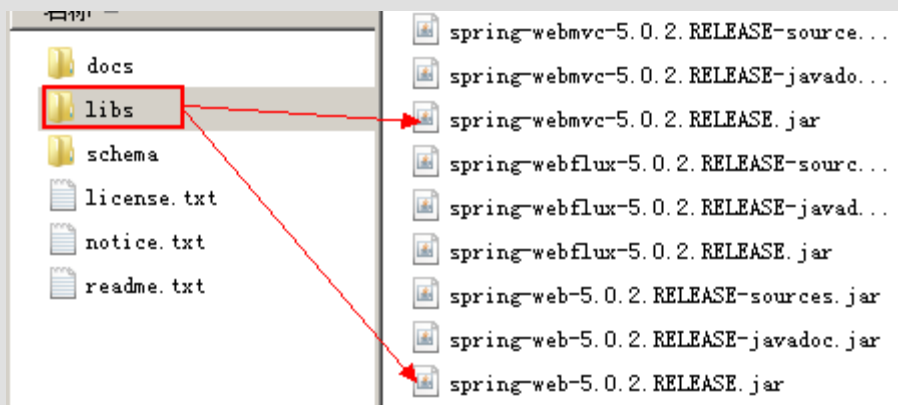


jsp 中的内容:

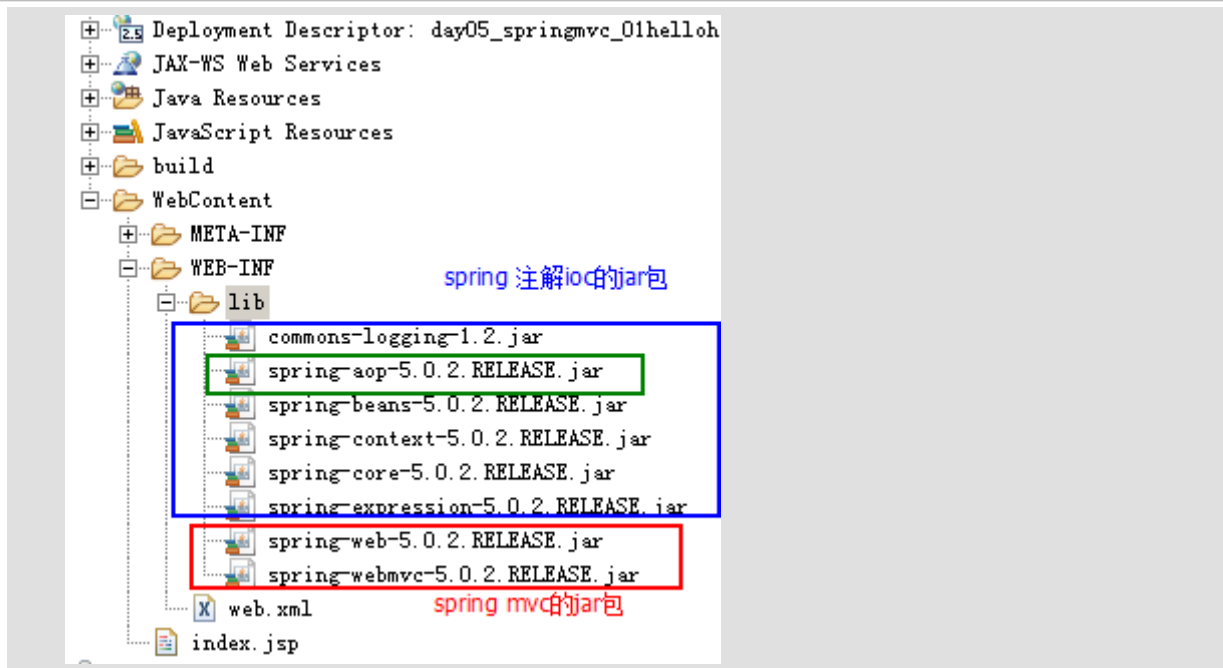
```
<a href="{pageContext.request.contextPath}/hello">SpringMVC 入门案例</a>
<br/>
<a href="hello">SpringMVC 入门案例</a>
```

2.1.2 拷贝 jar 包

spring mvc 的 jar 包就在



除了上面两个 jar 包之外，还需要拷贝 spring 的注解 ioc 所需 jar 包（包括一个 aop 的 jar 包）。



2.1.3 配置核心控制器-一个 Servlet

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  id="WebApp_ID" version="2.5">
  <!-- 配置 spring mvc 的核心控制器 -->
  <servlet>
    <servlet-name>SpringMVCDispatcherServlet</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <!-- 配置初始化参数，用于读取 SpringMVC 的配置文件 -->
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:SpringMVC.xml</param-value>
    </init-param>
    <!-- 配置 servlet 的对象的创建时间点：应用加载时创建。
      取值只能是非 0 正整数，表示启动顺序 -->
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>SpringMVCDispatcherServlet</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
```



```
</web-app>
```

2.1.4 创建 spring mvc 的配置文件

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

    <!-- 配置创建 spring 容器要扫描的包 -->
    <context:component-scan base-package="com.itheima"></context:component-scan>

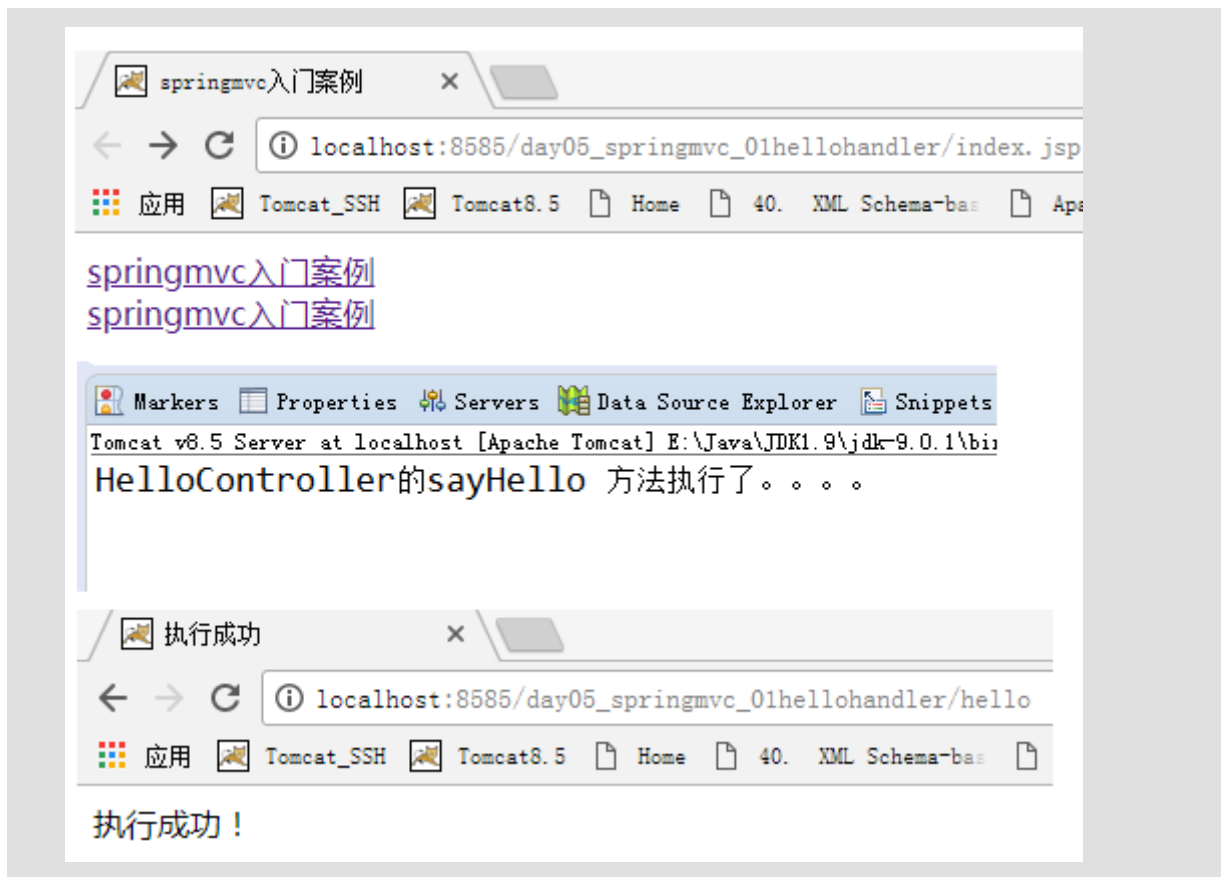
    <!-- 配置视图解析器 -->
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/pages/"></property>
        <property name="suffix" value=".jsp"></property>
    </bean>
</beans>
```

2.1.5 编写控制器并使用注解配置

```
/**
 * spring_mvc 的入门案例
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
@Controller("helloController")
public class HelloController {

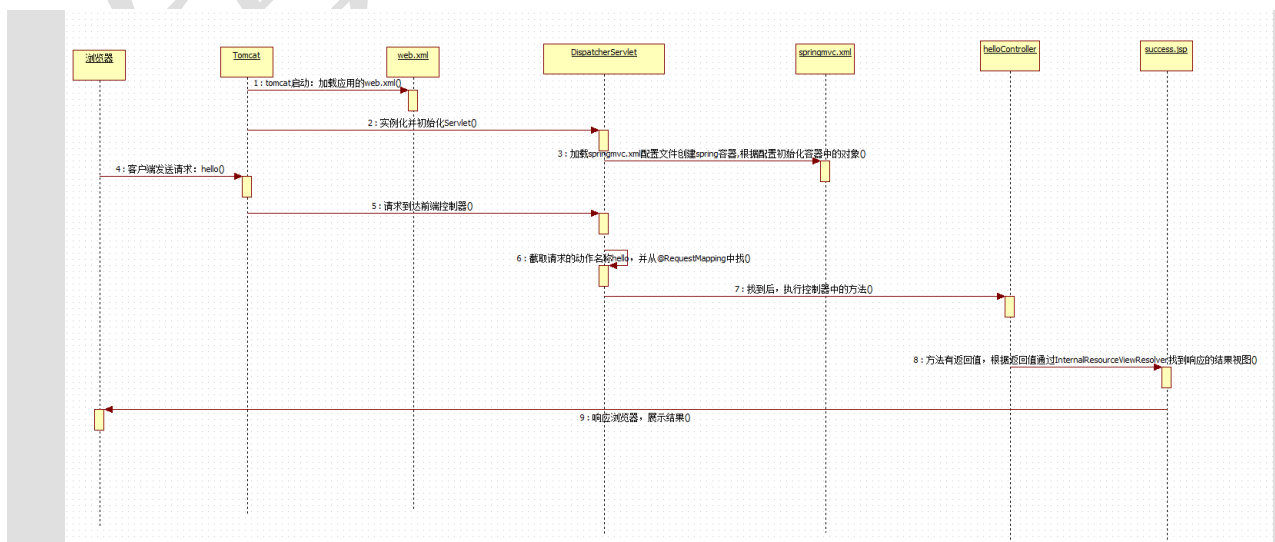
    @RequestMapping("/hello")
    public String sayHello() {
        System.out.println("HelloController 的 sayHello 方法执行了。。。");
        return "success";
    }
}
```

2.1.6 测试



2.2 入门案例的执行过程及原理分析

2.2.1 案例的执行过程



1、服务器启动，应用被加载。读取到 web.xml 中的配置创建 spring 容器并且初始化容器中的对象。



从入门案例中可以看到的是：HelloController 和 InternalResourceViewResolver，但是远不止这些。

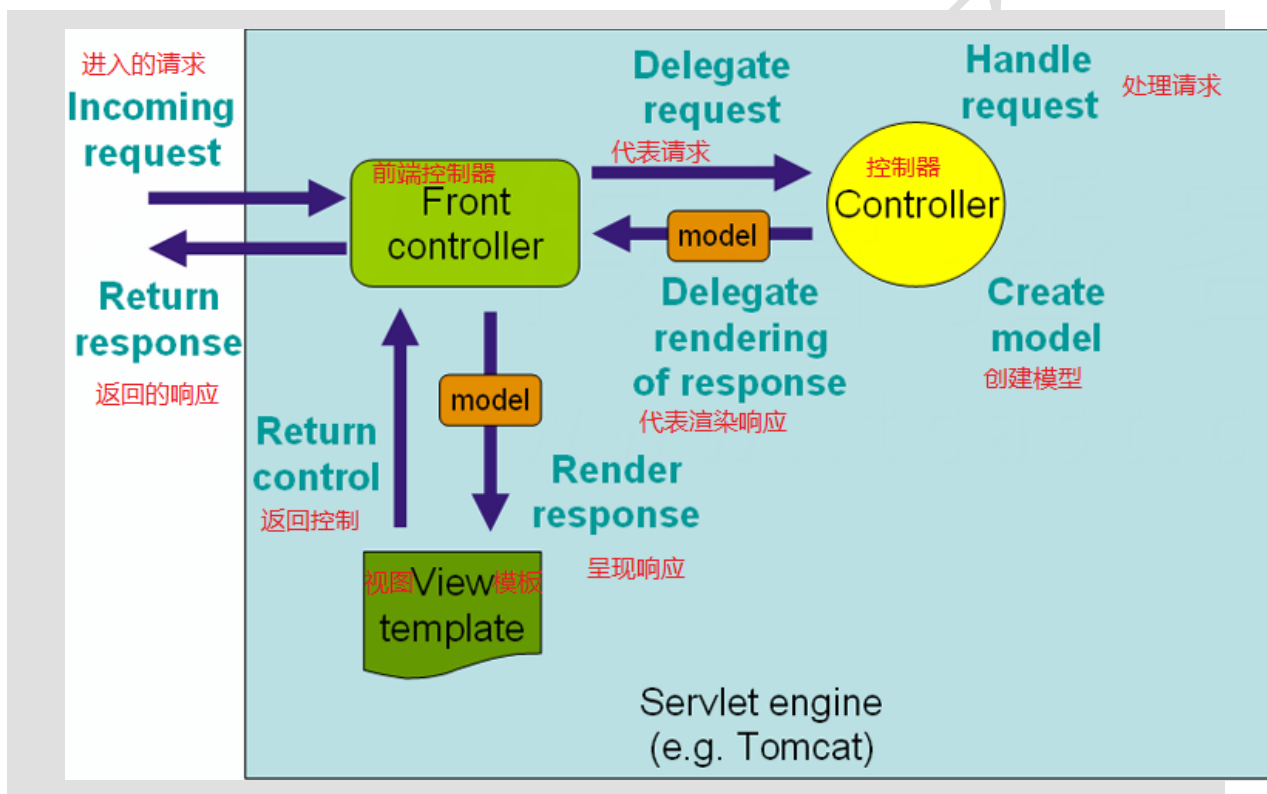
2、浏览器发送请求，被 DispatcherServlet 捕获，该 Servlet 并不处理请求，而是把请求转发出去。转发的路径是根据请求 URL，匹配@RequestMapping 中的内容。

3、匹配到了后，执行对应方法。该方法有一个返回值。

4、根据方法的返回值，借助 InternalResourceViewResolver 找到对应的结果视图。

5、渲染结果视图，响应浏览器。

2.2.2 SpringMVC 的请求响应流程



2.3 入门案例中涉及的组件

2.3.1 DispatcherServlet: 前端控制器

用户请求到达前端控制器，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 HandlerAdapter：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 View：视图

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

一般情况下需要通过页面标签或页面模版技术将模型数据通过页面展示给用户，需要由程序员根据业务需求开发具体的页面。

2.3.7 <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。

它就相当于在 xml 中配置了：

```
<!-- 上面的标签相当于 如下配置 -->
```

```
<!-- Begin -->
```

```
<!-- HandlerMapping -->
```

```
<bean
```



```
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"></bean>
    <bean
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"></bean>
    <!-- HandlerAdapter -->
    <bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"></bean>
    <bean
class="org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter"></bean>
    <bean
class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"></bean>
    <!-- HadnlerExceptionResolvers -->
    <bean
class="org.springframework.web.servlet.mvc.method.annotation.ExceptionHandlerExceptionHandlerResolver"></bean>
    <bean
class="org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionHandlerResolver"></bean>
    <bean
class="org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionHandlerResolver"></bean>
    <!-- End -->
```

注意：

一般开发中，我们都需要写上此标签（虽然从入门案例中看，我们不写也行，随着课程的深入，该标签还有具体的使用场景）。

明确：

我们只需要编写处理具体业务的控制器以及视图。

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。



出现位置:

类上:

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
```

红色的部分就是把 RequestMapping 写在类上，使我们的 URL 更加精细。

方法上:

请求 URL 的第二级访问目录。

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 = {"money!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。

2.4.2 使用示例

2.4.2.1 出现位置的示例:

控制器代码:

```
/**  
 * RequestMapping 注解出现的位置  
 * @author 黑马程序员  
 * @Company http://www.ithiema.com  
 * @Version 1.0  
 */  
  
@Controller("accountController")  
@RequestMapping("/account")  
public class AccountController {
```



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

jsp 中的代码:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>requestmapping 的使用</title>
    </head>
    <body>
        <!-- 第一种访问方式 -->
        <a href="${pageContext.request.contextPath}/account/findAccount">
            查询账户
        </a>
        <br/>
        <!-- 第二种访问方式 -->
        <a href="account/findAccount">查询账户</a>
    </body>
</html>
```

注意:

当我们使用此种方式配置时，在jsp中第二种写法时，不要在访问URL前面加/，否则无法找到资源。

2.4.2.2 method 属性的示例:

控制器代码:

```
/**
 * 保存账户
 * @return
 */
@RequestMapping(value="/saveAccount",method=RequestMethod.POST)
public String saveAccount() {
    System.out.println("保存了账户");
    return "success";
}
```



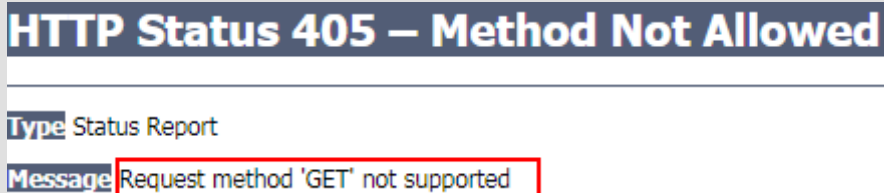
jsp 代码:

<!-- 请求方式的示例 -->

```
<a href="account/saveAccount">保存账户，get 请求</a>
<br/>
<form action="account/saveAccount" method="post">
    <input type="submit" value="保存账户，post 请求">
</form>
```

注意:

当使用 get 请求时，提示错误信息是 405，信息是方法不支持 get 方式请求



2.4.2.3 params 属性的示例:

控制器的代码:

```
/**
 * 删除账户
 * @return
 */
@RequestMapping(value="/removeAccount",params= {"accountName","money>100"})
public String removeAccount() {
    System.out.println("删除了账户");
    return "success";
}
```

jsp 中的代码:

<!-- 请求参数的示例 -->

```
<a href="account/removeAccount?accountName=aaa&money>100">删除账户，金额 100</a>
<br/>
<a href="account/removeAccount?accountName=aaa&money>150">删除账户，金额 150</a>
```

注意:

当我们点击第一个超链接时，可以访问成功。

当我们点击第二个超链接时，无法访问。如下图:

HTTP Status 400 – Bad Request

Type Status Report

Message Parameter conditions "accountName, money>100" not met for actual request parameters: accountName={aaa}, money>150={}



第3章 请求参数的绑定

3.1 绑定说明

3.1.1 绑定的机制

我们都知道，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：

基本类型参数：

包括基本类型和 String 类型

POJO 类型参数：

包括实体类，以及关联的实体类

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。

注意：

它还可以实现一些数据类型自动转换。内置转换器全都在：

org.springframework.core.convert.support 包下。有：

```
java.lang.Boolean -> java.lang.String : ObjectToStringConverter
java.lang.Character -> java.lang.Number : CharacterToNumberFactory
java.lang.Character -> java.lang.String : ObjectToStringConverter
java.lang.Enum -> java.lang.String : EnumToStringConverter
java.lang.Number -> java.lang.Character : NumberToCharacterConverter
java.lang.Number -> java.lang.Number : NumberToNumberConverterFactory
java.lang.Number -> java.lang.String : ObjectToStringConverter
java.lang.String -> java.lang.Boolean : StringToBooleanConverter
java.lang.String -> java.lang.Character : StringToCharacterConverter
java.lang.String -> java.lang.Enum : StringToEnumConverterFactory
java.lang.String -> java.lang.Number : StringToNumberConverterFactory
java.lang.String -> java.util.Locale : StringToLocaleConverter
java.lang.String -> java.util.Properties : StringToPropertiesConverter
java.lang.String -> java.util.UUID : StringToUUIDConverter
java.util.Locale -> java.lang.String : ObjectToStringConverter
java.util.Properties -> java.lang.String : PropertiesToStringConverter
java.util.UUID -> java.lang.String : ObjectToStringConverter
.....
```

如遇特殊类型转换要求，需要我们自己编写自定义类型转换器。

3.1.4 使用示例

3.1.4.1 基本类型和 String 类型作为参数

jsp 代码：

<!-- 基本类型示例 -->

查询账户

控制器代码：

/**

* 查询账户

* @return

*/

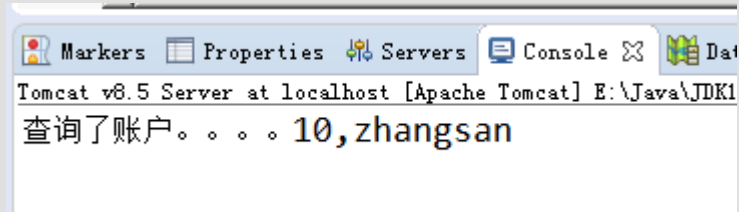
@RequestMapping("/findAccount")

public String findAccount(Integer accountId,String accountName) {



```
System.out.println("查询了账户。。。"+accountId+", "+accountName);  
return "success";  
}
```

运行结果:



3.1.4.2 POJO 类型作为参数

实体类代码:

```
/**  
 * 账户信息  
 * @author 黑马程序员  
 * @Company http://www.ithiema.com  
 * @Version 1.0  
 */  
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  
 * @author 黑马程序员  
 * @Company http://www.ithiema.com  
 * @Version 1.0  
 */  
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>

```

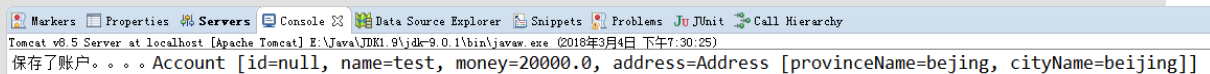
控制器代码:

```

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

```

运行结果:



Tomcat v6.5 Server at localhost [Apache Tomcat] E:\Java\jdk1.9.0_1\bin\javaw.exe (2018年3月4日 下午7:30:25)
保存了账户。。。 Account [id=null, name=test, money=20000.0, address=Address [provinceName=beijing, cityName=beijing]]

3.1.4.3 POJO 类中包含集合类型参数

实体类代码:

```

/**
 * 用户实体类
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
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

    @Override
    public String toString() {
        return "User [username=" + username + ", password=" + password + ", age="
+ age + ",\n accounts=" + accounts
        + ",\n accountMap=" + accountMap + " ]";
    }
}

```

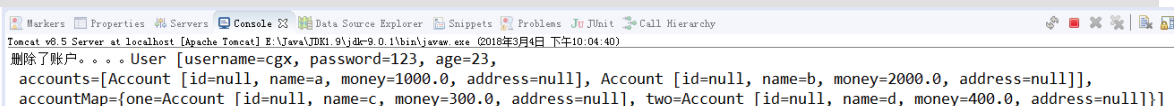


```
}  
}  
  
jsp 代码:  
<!-- POJO 类包含集合类型演示 -->  
<form action="/account/updateAccount" method="post">  
    用户名称: <input type="text" name="username" ><br/>  
    用户密码: <input type="password" name="password" ><br/>  
    用户年龄: <input type="text" name="age" ><br/>  
    账户 1 名称: <input type="text" name="accounts[0].name" ><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>
```

控制器代码:

```
/**  
 * 更新账户  
 * @return  
 */  
@RequestMapping("/updateAccount")  
public String updateAccount(User user) {  
    System.out.println("更新了账户。。。" + user);  
    return "success";  
}
```

运行结果:



Tomcat v8.5 Server at localhost [Apache Tomcat/8.5.9] [Java/1.8.0_101] [bin/javaw.exe] (2018年3月4日 下午10:04:40)
删除了账户。。。 User [username=cgx, password=123, age=23,
accounts=[Account [id=null, name=a, money=1000.0, address=null], Account [id=null, name=b, money=2000.0, address=null]],
accountMap={one=Account [id=null, name=c, money=300.0, address=null], two=Account [id=null, name=d, money=400.0, address=null]]}

3.1.4.4 请求参数乱码问题

post 请求方式:

在 web.xml 中配置一个过滤器

<!-- 配置 springMVC 编码过滤器 -->

```
<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>
在 springmvc 的配置文件中可以配置，静态资源不过滤：
<!-- location 表示路径，mapping 表示文件，**表示该目录下的文件以及子目录的文件 -->
<mvc:resources location="/css/" mapping="/css/**"/>
<mvc:resources location="/images/" mapping="/images/**"/>
<mvc:resources location="/scripts/" mapping="/javascript/**"/>

```

get 请求方式:

tomcat 对 GET 和 POST 请求处理方式是不同的，GET 请求的编码问题，要改 tomcat 的 server.xml 配置文件，如下：

```

<Connector connectionTimeout="20000" port="8080"
    protocol="HTTP/1.1" redirectPort="8443"/>

```

改为：

```

<Connector connectionTimeout="20000" port="8080"
    protocol="HTTP/1.1" redirectPort="8443"
    useBodyEncodingForURI="true"/>

```

如果遇到 ajax 请求仍然乱码，请把：

```
useBodyEncodingForURI="true"改为 URIEncoding="UTF-8"
```

即可。

3.2 特殊情况

3.2.1 自定义类型转换器

3.2.1.1 使用场景：

jsp 代码：



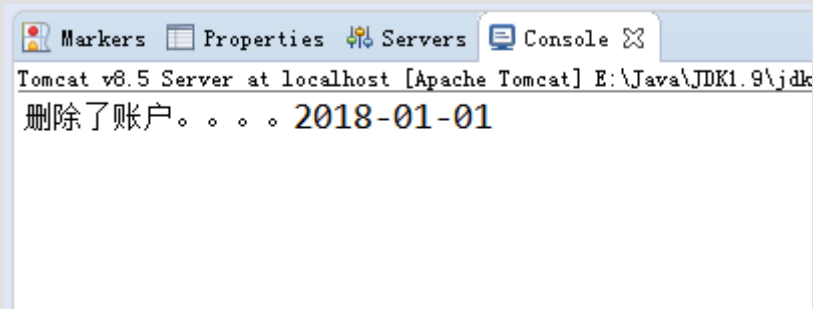
<!-- 特殊情况之：类型转换问题 -->

根据日期删除账户

控制器代码：

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

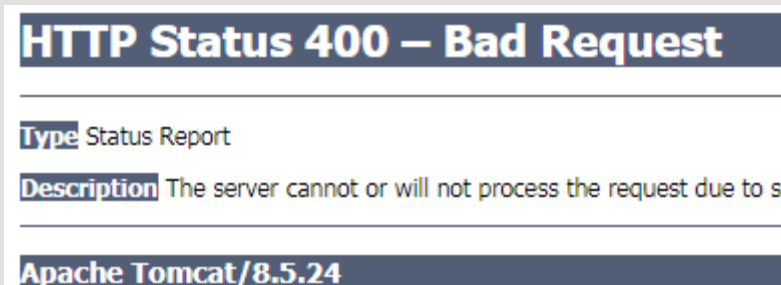
运行结果：



当我们把控制器中方法参数的类型改为 **Date** 时：

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

运行结果：



异常提示：

Failed to bind request element:

[org.springframework.web.method.annotation.MethodArgumentTypeMismatchException](#):

Failed to convert value of type 'java.lang.String' to required type



```
'java.util.Date'; nested exception is  
    org.springframework.core.convert.ConversionFailedException:  
    Failed to convert from type [java.lang.String] to type [java.util.Date] for  
    value '2018-01-01'; nested exception is java.lang.IllegalArgumentException
```

3.2.1.2 使用步骤

第一步：定义一个类，实现 `Converter` 接口，该接口有两个泛型。

```
public interface Converter<S, T> { //S:表示接受的类型, T: 表示目标类型  
    /**  
     * 实现类型转换的方法  
     */  
    @Nullable  
    T convert(S source);  
}  
/**  
 * 自定义类型转换器  
 * @author 黑马程序员  
 * @Company http://www.ithiema.com  
 * @Version 1.0  
 */  
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");  
        }  
    }  
}
```

第二步：在 `spring` 配置文件中配置类型转换器。

`spring` 配置类型转换器的机制是，将自定义的转换器注册到类型转换服务中去。

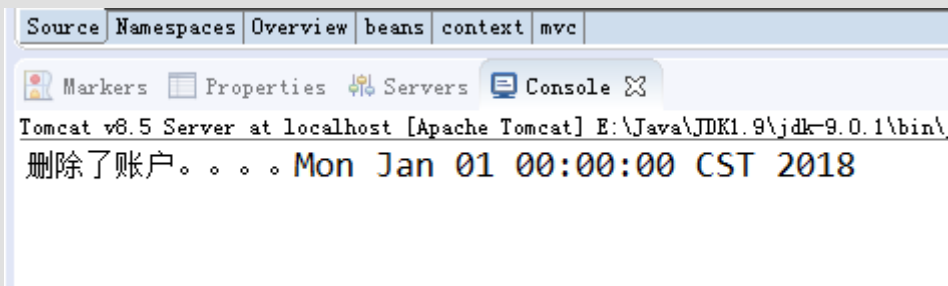


```
<!-- 配置类型转换器工厂 -->
<bean id="converterService"
      class="org.springframework.context.support.ConversionServiceFactoryBean">
  <!-- 给工厂注入一个新的类型转换器 -->
  <property name="converters">
    <array>
      <!-- 配置自定义类型转换器 -->
      <bean class="com.itheima.web.converter.StringToDateConverter"/>
    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

第三步：在 **annotation-driven** 标签中引用配置的类型转换服务

```
<!-- 引用自定义类型转换器 -->
<mvc:annotation-driven
  conversion-service="converterService"/>
```

运行结果：



3.2.2 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。

部分示例代码：

jsp 代码：

```
<!-- 原始 ServletAPI 作为控制器参数 -->
<a href="account/testServletAPI">测试访问 ServletAPI</a>
```

控制器中的代码：

```
/**
 * 测试访问 testServletAPI
```



```
* @return
*/
@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";
}
```

执行结果:

```
Tomcat v8.5 Server at localhost [Apache Tomcat] E:\Java\JDK1.9\jdk-9.0.1\bin\javaw.exe (2018年3月4
org.apache.catalina.connector.RequestFacade@5606d01a
org.apache.catalina.connector.ResponseFacade@49519f1d
org.apache.catalina.session.StandardSessionFacade@21592227
```

第4章 常用注解

4.1 RequestParam

4.1.1 使用说明

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。

4.1.2 使用示例

jsp 中的代码:

```
<!-- requestParams 注解的使用 -->
```

```
<a href="springmvc/useRequestParam?name=test">RequestParam 注解</a>
```

控制器中的代码:

```
/**
```

```
 * requestParams 注解的使用
```

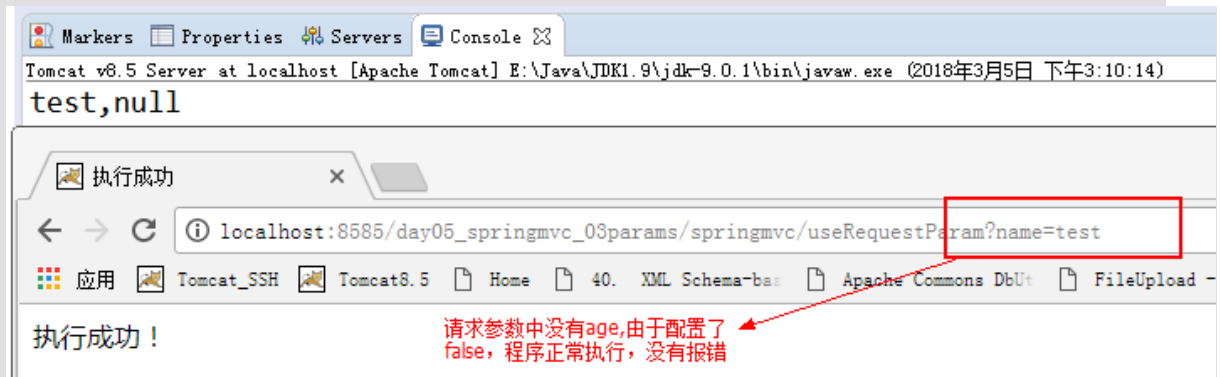
```
 * @param username
```

```
 * @return
```




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

运行结果:



4.2RequestBody

4.2.1 使用说明

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。

4.2.2 使用示例

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:

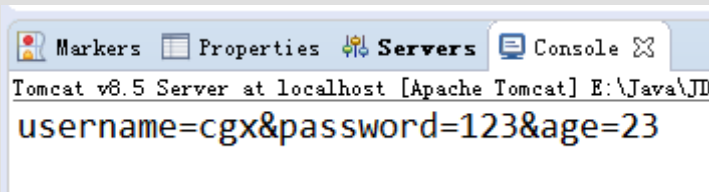
```
<a href="springmvc/useRequestBody?body=test">requestBody 注解 get 请求</a>
```

控制器代码:

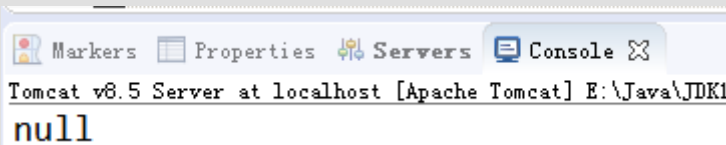


```
/**
 * RequestBody 注解
 * @param user
 * @return
 */
@RequestMapping("/useRequestBody")
public String useRequestBody(@RequestBody(required=false) String body){
    System.out.println(body);
    return "success";
}
```

post 请求运行结果:



get 请求运行结果:



4.3 PathVariable

4.3.1 使用说明

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。

4.3.2 使用示例

jsp 代码:

```
<!-- PathVariable 注解 -->
```

```
<a href="springmvc/usePathVariable/100">pathVariable 注解</a>
```

控制器代码:

```
/**
```

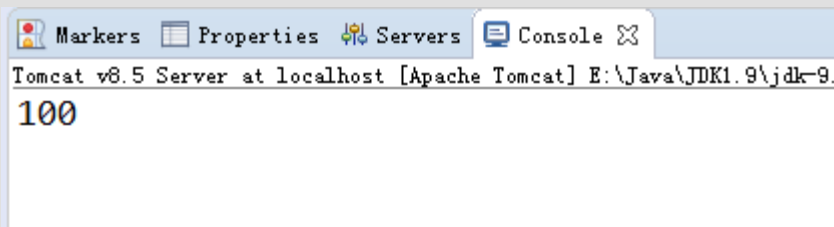
```
 * PathVariable 注解
```

```
 * @param user
```



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

运行结果:



4.3.3 REST 风格 URL

什么是 rest:

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。

它本身并没有什么实用性, 其核心价值在于如何设计出符合 REST 风格的网络接口。

restful 的优点

它结构清晰、符合标准、易于理解、扩展方便, 所以正得到越来越多网站的采用。

restful 的特性:

资源 (Resources): 网络上的一个实体, 或者说是网络上的一个具体信息。

它可以是一段文本、一张图片、一首歌曲、一种服务, 总之就是一个具体的存在。可以用一个 URI (统一资源定位符) 指向它, 每种资源对应一个特定的 URI。要

获取这个资源, 访问它的 URI 就可以, 因此 **URI** 即为每一个资源的独一无二的识别符。

表现层 (Representation): 把资源具体呈现出来的形式, 叫做它的表现层 (Representation)。

比如, 文本可以用 txt 格式表现, 也可以用 HTML 格式、XML 格式、JSON 格式表现, 甚至可以采用二进制格式。

状态转化 (State Transfer): 每发出一个请求, 就代表了客户端和服务器的一个交互过程。

HTTP 协议, 是一个无状态协议, 即所有的状态都保存在服务器端。因此, 如果客户端想要操作服务器, 必须通过某种手段, 让服务器端发生“状态转化” (State Transfer)。而这种转化是建立在表现层之上的, 所以就是“表现层状态转化”。具体说, 就是 HTTP 协议里面, 四个表示操作方式的动词: **GET**、**POST**、**PUT**、**DELETE**。它们分别对应四种基本操作: **GET** 用来获取资源, **POST** 用来新建资源, **PUT** 用来更新资源, **DELETE** 用来删除资源。

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

4.3.4 基于 HiddentHttpMethodFilter 的示例

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 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。

使用方法:

第一步：在 web.xml 中配置该过滤器。

第二步：请求方式必须使用 post 请求。

第三步：按照要求提供_method 请求参数，该参数的取值就是我们需要的请求方式。

源码分析:

```
/** Default method parameter: {@code _method} */
public static final String DEFAULT_METHOD_PARAM = "_method";

private String methodParam = DEFAULT_METHOD_PARAM;

/**
 * Set the parameter name to look for HTTP methods.
 * @see #DEFAULT_METHOD_PARAM
 */
public void setMethodParam(String methodParam) {
    Assert.hasText(methodParam, "'methodParam' must not be empty");
    this.methodParam = methodParam;
}

@Override
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
    throws ServletException, IOException {

    HttpServletRequest requestToUse = request;

    if ("POST".equals(request.getMethod()) && request.getAttribute(WebUtils.ERROR_EXCEPTION_ATTRIBUTE) == null) {
        String paramValue = request.getParameter(this.methodParam);
        if (StringUtils.hasLength(paramValue)) {
            requestToUse = new HttpMethodRequestWrapper(request, paramValue);
        }
    }

    filterChain.doFilter(requestToUse, response);
}
```

jsp 中示例代码:

```
<!-- 保存 -->
<form action="springmvc/testRestPOST" method="post">
    用户名: <input type="text" name="username"><br/>
    <!-- <input type="hidden" name="_method" value="POST"> -->
    <input type="submit" value="保存">
</form>
<hr/>
<!-- 更新 -->
<form action="springmvc/testRestPUT/1" method="post">
    用户名: <input type="text" name="username"><br/>
    <input type="hidden" name="_method" value="PUT">
    <input type="submit" value="更新">
</form>
```



```
<hr/>
<!-- 删除 -->
<form action="springmvc/testRestDELETE/1" method="post">
    <input type="hidden" name="_method" value="DELETE">
    <input type="submit" value="删除">
</form>
<hr/>
<!-- 查询一个 -->
<form action="springmvc/testRestGET/1" method="post">
    <input type="hidden" name="_method" value="GET">
    <input type="submit" value="查询">
</form>
```

控制器中示例代码:

```
/**
 * post 请求: 保存
 * @param username
 * @return
 */
@RequestMapping(value="/testRestPOST",method=RequestMethod.POST)
public String testRestfulURLPOST(User user){
    System.out.println("rest post "+user);
    return "success";
}

/**
 * put 请求: 更新
 * @param username
 * @return
 */
@RequestMapping(value="/testRestPUT/{id}",method=RequestMethod.PUT)
public String testRestfulURLPUT(@PathVariable("id") Integer id,User user){
    System.out.println("rest put "+id+", "+user);
    return "success";
}

/**
 * post 请求: 删除
 * @param username
 * @return
 */
@RequestMapping(value="/testRestDELETE/{id}",method=RequestMethod.DELETE)
public String testRestfulURLDELETE(@PathVariable("id") Integer id){
    System.out.println("rest delete "+id);
}
```



```
        return "success";
    }

    /**
     * post 请求: 查询
     * @param username
     * @return
     */
    @RequestMapping(value="/testRestGET/{id}",method=RequestMethod.GET)
    public String testRestfulURLGET(@PathVariable("id") Integer id){
        System.out.println("rest get "+id);
        return "success";
    }
}
```

运行结果:

```
Tomcat v8.5 Server at localhost [Apache Tomcat] E:\Java\jdk1.9\jdk-9.0.1\bin\javaw.exe (2018年3月5日 下午5:23:4)
rest post User [username=test save, password=null, age=null,
accounts=null,
accountMap=null]
rest put 1, User [username=test update, password=null, age=null,
accounts=null,
accountMap=null]
rest delete 1
rest get 1
```

4.4 RequestHeader

4.4.1 使用说明

作用:

用于获取请求消息头。

属性:

value: 提供消息头名称

required: 是否必须有此消息头

注:

在实际开发中一般不怎么用。



4.4.2 使用示例

jsp 中代码:

```
<!-- RequestHeader 注解 -->
```

```
<a href="springmvc/useRequestHeader">获取请求消息头</a>
```

控制器中代码:

```
/**
```

```
 * RequestHeader 注解
```

```
 * @param user
```

```
 * @return
```

```
 */
```

```
@RequestMapping("/useRequestHeader")
```

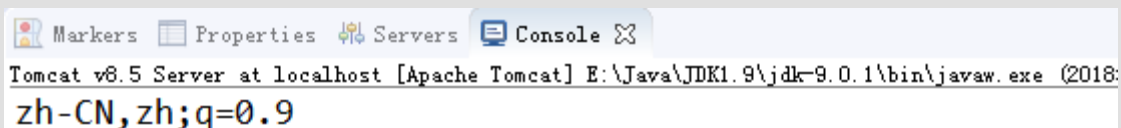
```
public String useRequestHeader(@RequestHeader(value="Accept-Language",  
                                         required=false)String requestHeader){
```

```
    System.out.println(requestHeader);
```

```
    return "success";
```

```
}
```

运行结果:



```
Tomcat v8.5 Server at localhost [Apache Tomcat] E:\Java\JDK1.9\jdk-9.0.1\bin\javaw.exe (2018:  
zh-CN,zh;q=0.9
```

4.5 CookieValue

4.5.1 使用说明

作用:

用于把指定 cookie 名称的值传入控制器方法参数。

属性:

value: 指定 cookie 的名称。

required: 是否必须有此 cookie。

4.5.2 使用示例

jsp 中的代码:

```
<!-- CookieValue 注解 -->
```

```
<a href="springmvc/useCookieValue">绑定 cookie 的值</a>
```

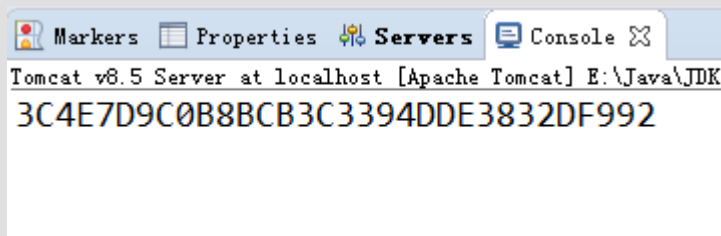
控制器中的代码:

```
/**
```



```
* Cookie 注解注解
* @param user
* @return
*/
@RequestMapping("/useCookieValue")
public String useCookieValue(@CookieValue(value="JSESSIONID",required=false)
String cookieValue){
    System.out.println(cookieValue);
    return "success";
}
```

运行结果:



4.6ModelAttribute

4.6.1 使用说明

作用:

该注解是 SpringMVC4.3 版本以后新加入的。它可以用于修饰方法和参数。

出现在方法上，表示当前方法会在控制器的方法执行之前，先执行。它可以修饰没有返回值的方法，也可以修饰有具体返回值的方法。

出现在参数上，获取指定的数据给参数赋值。

属性:

value: 用于获取数据的 key。key 可以是 POJO 的属性名称，也可以是 map 结构的 key。

应用场景:

当表单提交数据不是完整的实体类数据时，保证没有提交数据的字段使用数据库对象原来的数据。

例如:

我们在编辑一个用户时，用户有一个创建信息字段，该字段的值是不允许被修改的。在提交表单数据是肯定没有此字段的内容，一旦更新会把该字段内容置为 null，此时就可以使用此注解解决问题。

4.6.2 使用示例

4.6.2.1 基于 POJO 属性的基本使用:

jps 代码:

```
<!-- ModelAttribute 注解的基本使用 -->
```




```
<a href="springmvc/testModelAttribute?username=test">测试 modelattribute</a>
```

控制器代码:

```
/**
 * 被 ModelAttribute 修饰的方法
 * @param user
 */
@RequestMapping("/testModelAttribute")
public void showModel(User user) {
    System.out.println("执行了 showModel 方法"+user.getUsername());
}

/**
 * 接收请求的方法
 * @param user
 * @return
 */
@RequestMapping("/testModelAttribute")
public String testModelAttribute(User user) {
    System.out.println("执行了控制器的方法"+user.getUsername());
    return "success";
}
```

运行结果:

Tomcat v8.5 Server at localhost [Apache Tomcat] E:\Java\...
执行了showModel方法test
执行了控制器的方法test

4.6.2.2 基于 Map 的应用场景示例 1: ModelAttribute 修饰方法带返回值

需求:

修改用户信息, 要求用户的密码不能修改

jsp 的代码:

```
<!-- 修改用户信息 -->
<form action="springmvc/updateUser" method="post">
    用户名称: <input type="text" name="username" ><br/>
    用户年龄: <input type="text" name="age" ><br/>
    <input type="submit" value="保存">
</form>
```

控制的代码:

```
/**
 * 查询数据库中用户信息
 * @param user
 */
```

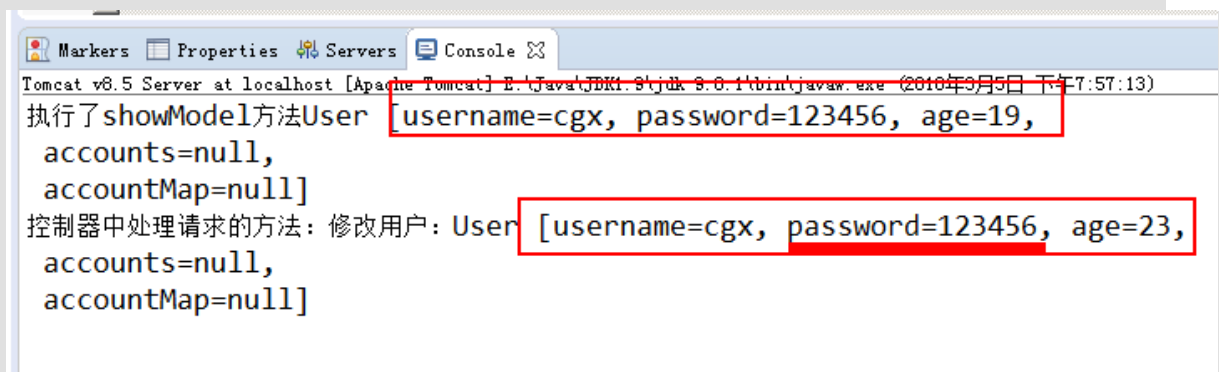


```
@ModelAttribute
public User showModel(String username) {
    //模拟去数据库查询
    User abc = findUserByName(username);
    System.out.println("执行了 showModel 方法"+abc);
    return abc;
}

/**
 * 模拟修改用户方法
 * @param user
 * @return
 */
@RequestMapping("/updateUser")
public String testModelAttribute(User user) {
    System.out.println("控制器中处理请求的方法: 修改用户: "+user);
    return "success";
}

/**
 * 模拟去数据库查询
 * @param username
 * @return
 */
private User findUserByName(String username) {
    User user = new User();
    user.setUsername(username);
    user.setAge(19);
    user.setPassword("123456");
    return user;
}
```

运行结果:





4.6.2.3 基于 Map 的应用场景示例 1: ModelAttribute 修饰方法不带返回值

需求:

修改用户信息, 要求用户的密码不能修改

jsp 中的代码:

<!-- 修改用户信息 -->

```
<form action="springmvc/updateUser" method="post">
    用户名称: <input type="text" name="username" ><br/>
    用户年龄: <input type="text" name="age" ><br/>
    <input type="submit" value="保存">
</form>
```

控制器中的代码:

```
/**
 * 查询数据库中用户信息
 * @param user
 */
@ModelAttribute
public void showModel(String username, Map<String, User> map) {
    //模拟去数据库查询
    User user = findUserByName(username);
    System.out.println("执行了 showModel 方法"+user);
    map.put("abc", user);
}

/**
 * 模拟修改用户方法
 * @param user
 * @return
 */
@RequestMapping("/updateUser")
public String testModelAttribute(@ModelAttribute("abc") User user) {
    System.out.println("控制器中处理请求的方法: 修改用户: "+user);
    return "success";
}

/**
 * 模拟去数据库查询
 * @param username
 * @return
 */
private User findUserByName(String username) {
    User user = new User();
    user.setUsername(username);
}
```



```
user.setAge(19);  
user.setPassword("123456");  
return user;  
}
```

运行结果:

Tomcat v8.5 Server at localhost [Apache Tomcat/8.5.90-jdk-9.0.1\bin\java.exe (2018年3月5日 下午8:01:21)]
执行了showModel方法User [username=test, password=123456, age=19,
accounts=null,
accountMap=null]
控制器中处理请求的方法: 修改用户: User [username=test, password=123456, age=23,
accounts=null,
accountMap=null]

4.7 SessionAttribute

4.7.1 使用说明

作用:

用于多次执行控制器方法间的参数共享。

属性:

value: 用于指定存入的属性名称

type: 用于指定存入的数据类型。

4.7.2 使用示例

jsp 中的代码:

```
<!-- SessionAttribute 注解的使用 -->  
<a href="springmvc/testPut">存入 SessionAttribute</a>  
<hr/>  
<a href="springmvc/testGet">取出 SessionAttribute</a>  
<hr/>  
<a href="springmvc/testClean">清除 SessionAttribute</a>
```

控制器中的代码:

```
/**  
 * SessionAttribute 注解的使用  
 * @author 黑马程序员  
 * @Company http://www.ithiema.com  
 * @Version 1.0  
 */  
@Controller("sessionAttributeController")
```



```
@RequestMapping("/springmvc")
@SessionAttributes(value = {"username", "password"}, types = {Integer.class})
public class SessionAttributeController {

    /**
     * 把数据存入 SessionAttribute
     * @param model
     * @return
     * Model 是 spring 提供的一个接口，该接口有一个实现类 ExtendedModelMap
     * 该类继承了 ModelMap，而 ModelMap 就是 LinkedHashMap 子类
     */
    @RequestMapping("/testPut")
    public String testPut(Model model) {
        model.addAttribute("username", "泰斯特");
        model.addAttribute("password", "123456");
        model.addAttribute("age", 31);
        //跳转之前将数据保存到 username、password 和 age 中，因为注解@SessionAttribute 中有
        return "success";
    }

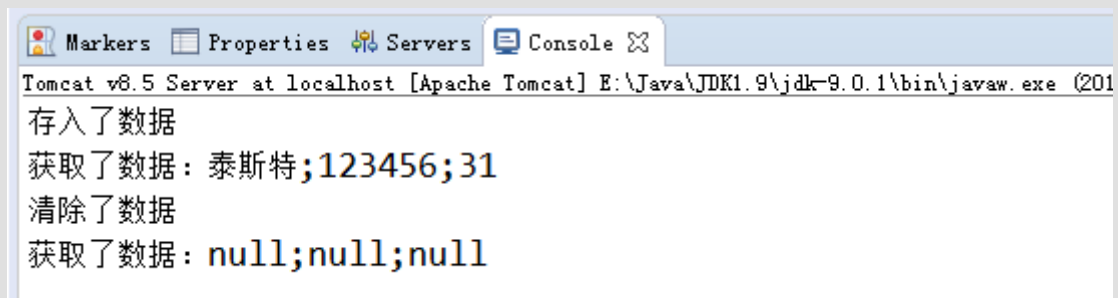
    @RequestMapping("/testGet")
    public String testGet(ModelMap model) {

        System.out.println(model.get("username") + ";" + model.get("password") + ";" + model.get("age"));

        return "success";
    }

    @RequestMapping("/testClean")
    public String complete(SessionStatus sessionStatus) {
        sessionStatus.setComplete();
        return "success";
    }
}
```

运行结果：



```
Tomcat v8.5 Server at localhost [Apache Tomcat] E:\Java\JDK1.9\jdk-9.0.1\bin\javaw.exe (201...
存入了数据
获取了数据：泰斯特;123456;31
清除了数据
获取了数据：null;null;null
```