SpringMVC

- 文件关系 ——>
- 类关系 ——>

archetypeCatalog

internal

MVC 三层架构设计

三层架构

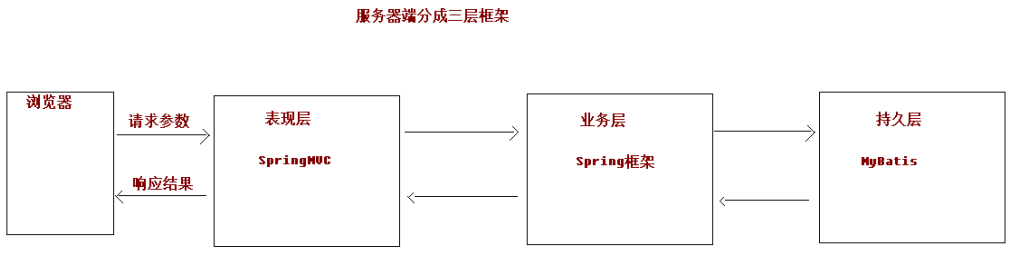
我们的开发架构一般都是基于两种形式：

- C/S 架构，也就是客户端/服务器
- 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(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)：

- **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

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|web层

- 也就是我们常说的 **web层**：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
    - （表现层，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|service层

- 也就是我们常说的 **service 层**：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|dao层

- 也就是我们是常说的 **dao 层**：负责数据持久化；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。
- 包括 **数据层** 和 **数据访问层**：
    - 数据层：即数据库
        - 数据库是对数据进行持久化的载体
    - 数据访问层
        - 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。

MVC设计模型

- MVC全名是**M**odel **V**iew **C**ontroller 模型-视图-控制器，每个部分各司其职。
- 是一种用于设计创建 Web 应用程序 **表现层** 的模式。

Model(模型)

数据模型，**JavaBean**的类

- 通常指的就是我们的 数据模型，一般情况下用于 **封装数据**。

View(视图)：

JSP、HTML...

- 用来**展示数据** 给用户

- 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

应用程序中处理 用户**交互** 的部分。**Servlet**

- 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。

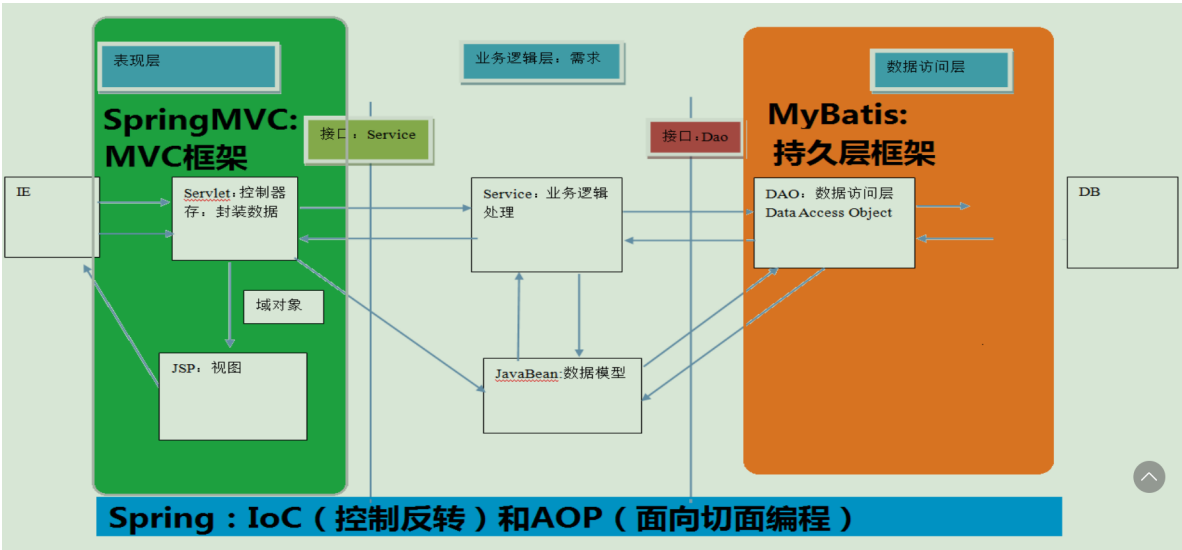
SpringMVC概述

SpringMVC是什么

- 是一种基于 **Java实现** 的 **MVC设计模型** 的 **请求驱动类型** 的 **轻量级 WEB框架**。
    - 属于 SpringFrameWork 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。
- 可插入(可选择)的 MVC 架构。
    - 在使用 Spring进行WEB开发时：可以选择使用Spring的SpringMVC框架或集成其他MVC开发框架，如Struts1(现在一般不用)，Struts2等。
- 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。
- 支持 **RESTful** 编程风格的请求。
- SpringMVC 已经成为目前最主流的 MVC 框架之一
    - 随着 Spring3.0 的发布，全面超越 Struts2，成为最优秀的 MVC 框架。

SpringMVC 与 三层架构

表现层框架。



SpringMVC 的优势

- 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）。
- 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。
- 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。
- 和 Spring 其他框架无缝集成，是其它 Web 框架所不具备的。
- 可适配，通过 HandlerAdapter 可以支持任意的类作为处理器。
- 可定制性，HandlerMapping、ViewResolver 等能够非常简单的定制。
- 功能强大的数据验证、格式化、绑定机制。
- 利用 Spring 提供的 Mock 对象能够非常简单的进行 Web 层单元测试。
- 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。
- 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。

SpringMVC 和 Structs2的优劣势分析

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**。

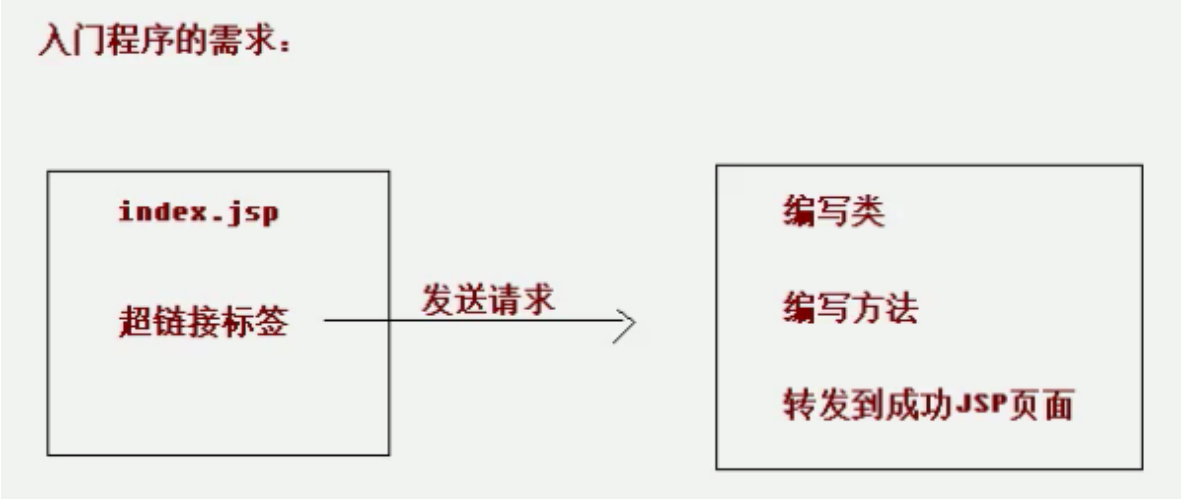
区别

- 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。

快速入门

入门分析

1. index.jsp
2. 控制器类
    - 方法
3. 成功JSP页面



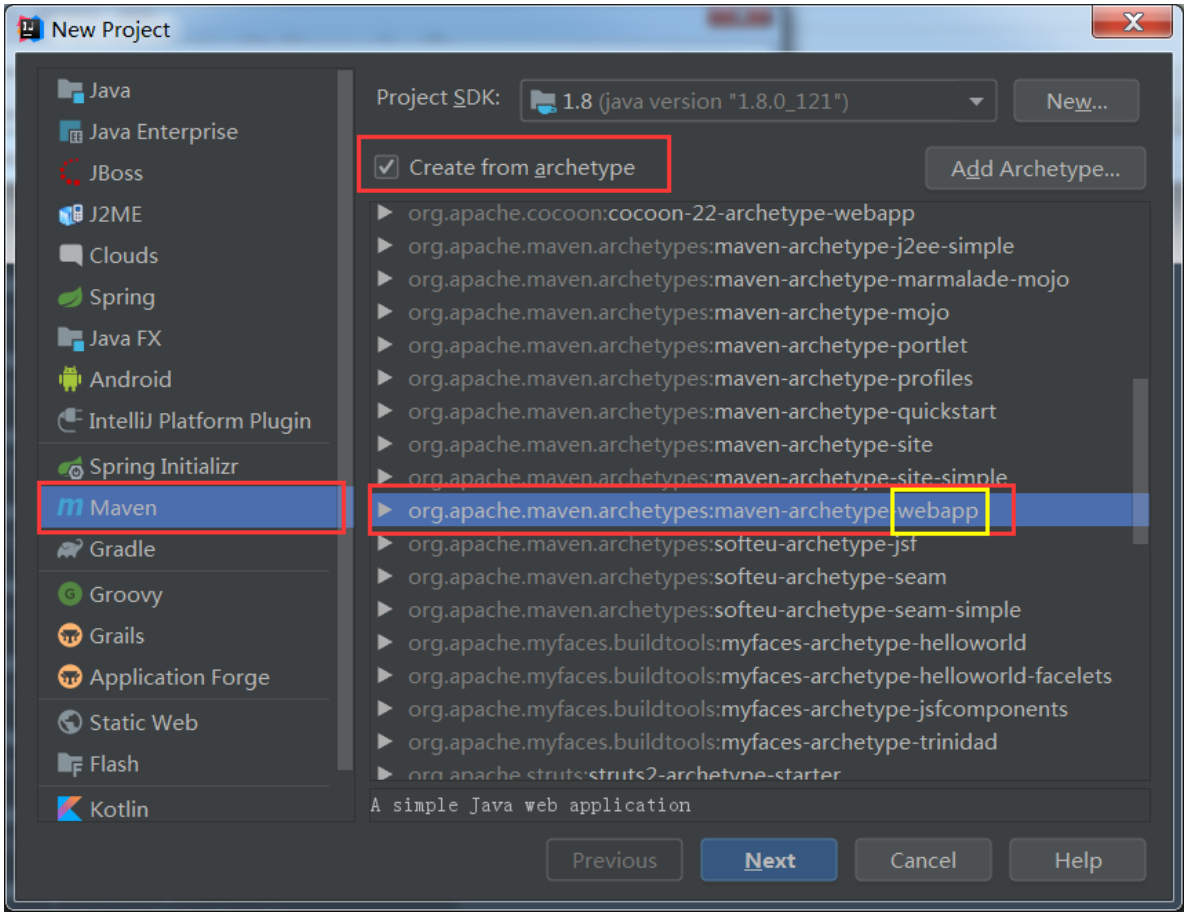
环境搭建

- 配置核心控制器
    - 加载springmvc配置文件
- 创建springmvc的配置文件
    - 开启注解扫描
    - 配置视图解析器
- 服务器部署

创建项目

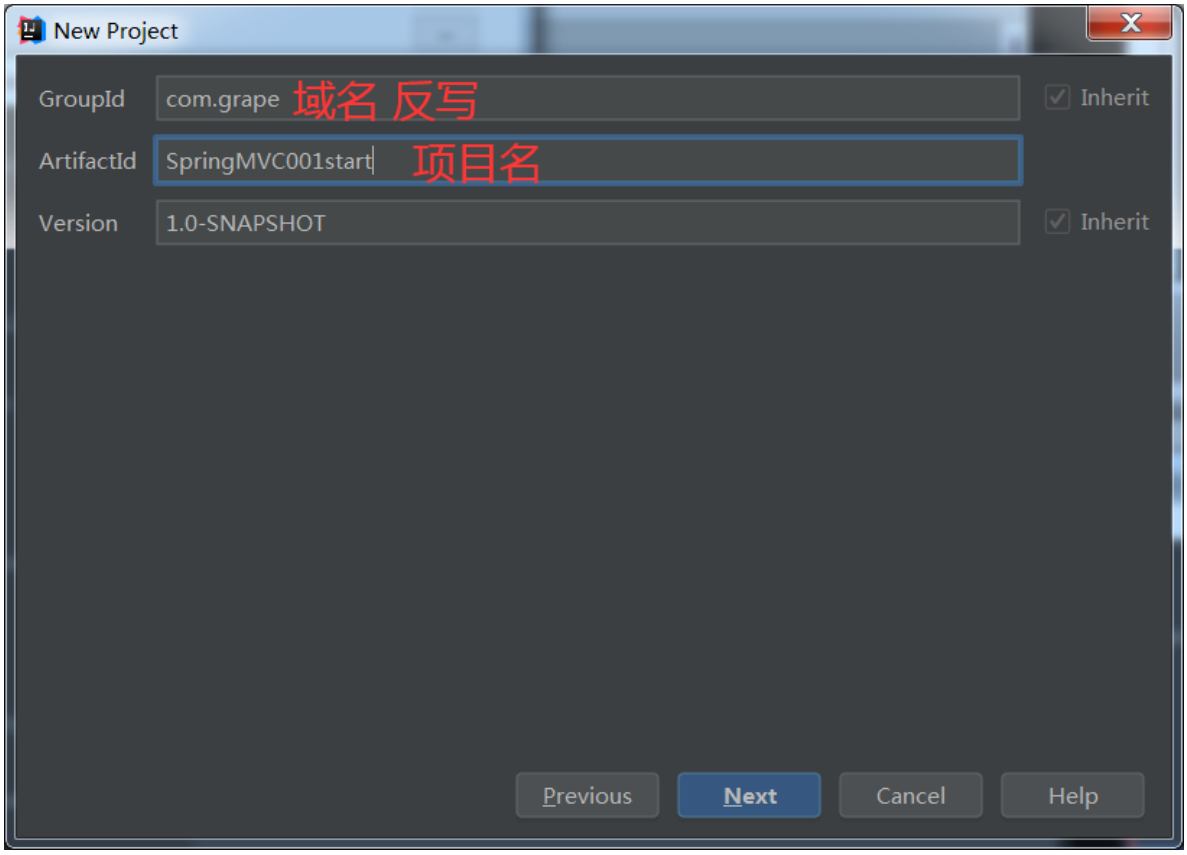
New Project

- Maven
- （Module SDK）
- **骨架构建**：Create from archetype
    - org.apache.maven.archetypes:maven-archetype-**webapp**



New Project

- GroupId
    - 域名反写
- ArtifactId
    - 项目名
- （Version）



New Module

选择Maven仓库：

- Maven home directory
- User settings file
- Local repository

生成pom文件：

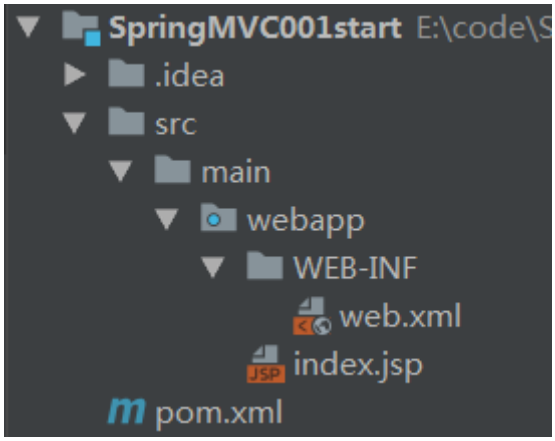
- groupId：公司名
- artifactId：项目名
- version：版本
- archetypeCatalog：解决maven创建项目过慢
    - 默认会去网上下载一些对应的插件，很慢
    - internal

构建好的项目

构建完的项目，目录结构是不全的。

- `src`
    - `main`
        - `webapp`
            - `WEB-INF`
                - web.xml

        - index.jsp
- pom.xml

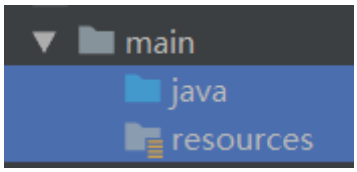


搭建开发环境

- 构建完的项目，目录结构是不全的。
- maven的目录结构应该是固定的。

新建所需文件夹

- main下 新建 ：
    - java
        - 右键：Mark Directory as
        - Sources Root ：源码根目录
    - resources
        - 右键：Mark Directory as
        - Resources Root：资源根目录



pom导入依赖

需要jar包：

- spring-context
- spring-web
- spring-webmvc
- servlet-api
- jsp-api

```
<properties>
    <!-- 版本锁定 -->
    <spring.version>5.0.2.RELEASE</spring.version>
```

```
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>servlet-api</artifactId>
        <version>2.5</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>javax.servlet.jsp</groupId>
        <artifactId>jsp-api</artifactId>
        <version>2.0</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

pom

```
<?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.grape</groupId>
    <artifactId>SpringMVC001start</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>

    <name>SpringMVC001start 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>
```

```
        <!-- 版本锁定 -->
        <spring.version>5.0.2.RELEASE</spring.version>
    </properties>


    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
            <version>2.5</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>javax.servlet.jsp</groupId>
            <artifactId>jsp-api</artifactId>
            <version>2.0</version>
            <scope>provided</scope>
        </dependency>

        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.11</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <finalName>SpringMVC001start</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>3.1.0</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>3.8.0</version>
          </plugin>
          <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.22.1</version>
          </plugin>
          <plugin>
            <artifactId>maven-war-plugin</artifactId>
            <version>3.2.2</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>
```

web.xml配置前端控制器

- 前端控制器，其实就是一个servlet。
    - 加载SpringMVC配置文件
        - servlet/ init-param
        - contextConfigLocation
        - classpath:springmvc.xml
- 类是spring提供好的，打出Dispatcher就会提示出来。
    - 配置好后，正常第一次请求时创建servlet
    - load-on-startup：改为，启动服务器就创建

```
<!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>dispatcherServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <!-- 加载SpringMVC配置文件 -->
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:springmvc.xml</param-value>
    </init-param>
    <!-- 配置servlet启动时加载对象 -->
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
```

```
    <servlet-name>dispatcherServlet</servlet-name>
    <!-- /的意思是，任何请求都会经过这个servlet -->
    <url-pattern>/</url-pattern>
  </servlet-mapping>

</web-app>
```

SpringMVC的配置文件

resources下新建即可。

加入约束：

- mvc
- context
    - 开启注解扫描

- 扫描包
- 视图解析器
- mvc注解支持

```
<?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spring创建容器时要扫描的包 -->
    <context:component-scan base-package="com.learn"></context:component-scan>
    <!-- 配置视图解析器 -->
    <bean id="viewResolver"

class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/pages/"></property>
        <property name="suffix" value=".jsp"></property>
    </bean>
    <!-- 配置spring开启注解mvc的支持 -->
    <mvc:annotation-driven></mvc:annotation-driven>
</beans>
```

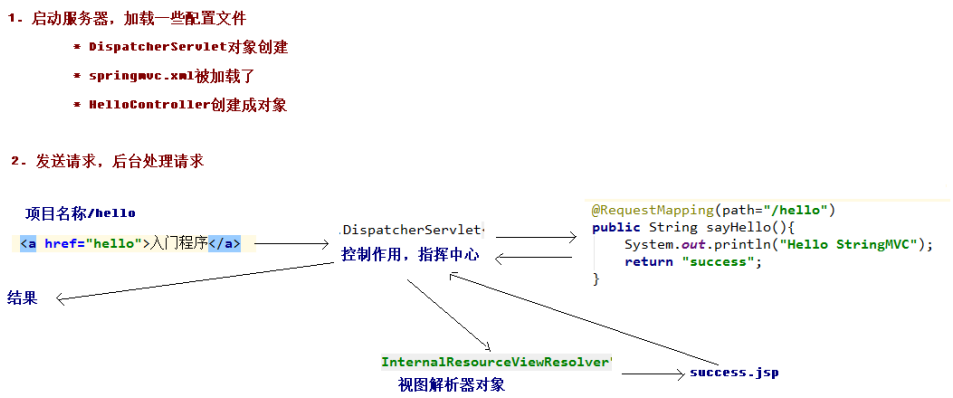
服务器部署

- +个Tomcat Server。

- 部署项目，Artifact下会有个war

程序执行流程

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 DispatherServlet 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。



入门程序

- **index.jsp**
    - 超链接，请求方法
    - 相对路径，从端口号后开始
    - ```
      <a href="/SpringMVC001/hello">入门程序,请求后台方法</a>
      ```
- $web.xml$
    - 配置前端控制器，SpringMVC的入口
    - ：
        - **前端控制器**，DispatcherServlet，Spring提供
    - ：全局初始化参数
        - **加载springmvc配置文件**
        - ：contextConfigLocation
        - ：classpath:springmvc.xml
    - ：
        - 1：**启动服务器就创建**
        - 默认：第一次请求时创建servlet
      ```

        - :
            - /的意思是，**任何请求都会经过这个servlet**

- 控制器类 **HelloController**
    - **@Controller** ：类上
    - **@RequestMapping**：方法上
        - @RequestMapping(path = "/hello")
    - 返回 **字符串**
        - 表示jsp文件的名字
- $springmvc.xml$

> 约束：
>
> - mvc
> - context
>
> 配置：
>
> - 扫描包
>   - <context：component-scan>
>   - base-package
> - 视图解析器：跳转
>   - bean
>   - class：InternalResourceViewResolver
>   - property
>   - prefix：文件所在目录
>   - /WEB-INF/pages/
>   - suffix：文件后缀名
>   - .jsp
> - mvc注解支持
>   - <mvc：annotation-driven/>

- **success.jsp**

java/com.learn.controller

HelloController

- **@Controller** ：类上
- **@RequestMapping**：方法上
    - @RequestMapping(path = "/hello")
- 返回 **字符串**（默认规则）
    - 表示jsp文件的名字

```java
/**
 * 控制器类
 */
@Controller
public class HelloController {

    @RequestMapping(path = "/hello")
    public String sayHello() {
        System.out.println("Hello SpringMVC");
        return "success";
    }
}
```

resources

springmvc.xml

- 约束：
    - mvc
    - context
- 配置：
    - 扫描包
        - `<context: component-scan>`
            - base-package
    - 视图解析器：跳转，找到对应的JSP页面。
        - bean
            - class：InternalResourceViewResolver
            - property
                - prefix：文件所在目录
                    - /WEB-INF/pages/
                - suffix：文件后缀名
                    - .jsp
    - mvc注解支持
        - `<mvc: annotation-driven/>`
            - 作用不只是支持注解
            - 还会配置上各种组件：
                - RequestMappingHandlerMapping（处理映射器）
                - RequestMappingHandlerAdapter（处理适配器）
                - ...

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns: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spring创建容器时要扫描的包 -->
    <context:component-scan base-package="com.learn"></context:component-scan>
    <!-- 配置视图解析器 -->
    <bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/pages/"></property>
        <property name="suffix" value=".jsp"></property>
    </bean>
    <!-- 配置spring开启注解mvc的支持 -->
    <mvc:annotation-driven></mvc:annotation-driven>
</beans>
```

webapp

WEB-INF

pages/success.jsp

success.jsp

WEB-INF下新建对应名字的jsp。

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <h3>入门成功</h3>
</body>
</html>
```

web.xml

-
  - 配置前端控制器，SpringMVC的入口
  - ：
    - **前端控制器**，DispatcherServlet，Spring提供
  - ：全局初始化参数
    - **加载springmvc配置文件**
    - ：contextConfigLocation
    - ：classpath:springmvc.xml

        - ：
            - 1：**启动服务器就创建**
            - 默认：第一次请求时创建servlet
-
        - ：
            - /的意思是，**任何请求都会经过这个servlet**

```
<!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>dispatcherServlet</servlet-name>
    <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <!-- 加载SpringMVC配置文件 -->
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:springmvc.xml</param-value>
    </init-param>
    <!-- 配置servlet启动时加载对象 -->
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>dispatcherServlet</servlet-name>
    <!-- /的意思是，任何请求都会经过这个servlet -->
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

index.jsp

- 自动生成的jsp有问题：没有头。
    - 建议，删掉新建。
- 超链接，请求方法
    - 相对路径，从端口号后开始

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <h3>入门程序</h3>
    <!-- 相对路径写法 localhost:8080可省 -->
    <a href="/SpringMVC001/hello">入门程序,请求后台方法</a>
</body>
</html>
```

<mvc：annotation-driven> 说明

在SpringMVC中，支持MVC注解，实际作用不止于此：

- 使用 <mvc：annotation-driven> 自动加载 RequestMappingHandlerMapping（处理映射器）和 RequestMappingHandlerAdapter（ 处理适配器 ）。三大组件中的两大组件，还缺一个视图解析器我们一般会手动配置
- 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。

就相当于在 SpringMVC.xml中配置了：

```
<!-- 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.ExceptionHandlerExceptionResolver"></bean>
<bean class="org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver"></bean>
<bean class="org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver"></bean>
<!-- End -->
```

环境搭建

单独使用，maven下 webapp 工程

骨架构建：Create from archetype

- org.apache.maven.archetypes:maven-archetype-**webapp**

文件关系

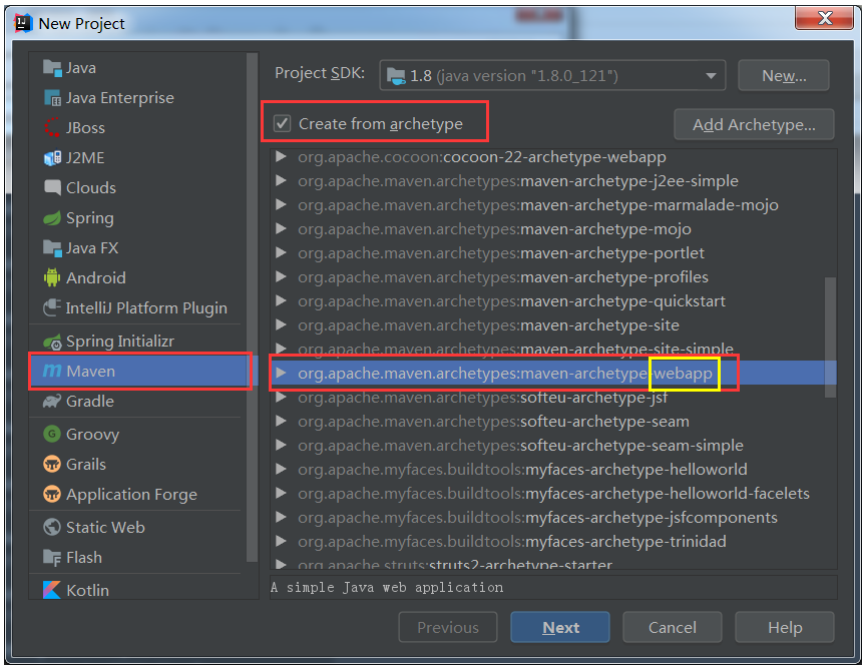
- web.xml
- springmvc.xml

- Controller.java

新建maven-webapp工程

骨架构建：Create from archetype

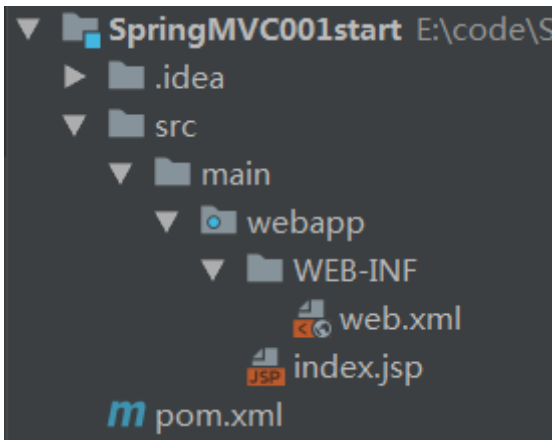
- org.apache.maven.archetypes:maven-archetype-**webapp**



项目结构

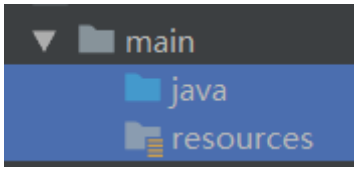
构建完的项目，**目录结构是不全的。**

- `src`
  - `main`
    - `webapp`
      - `WEB-INF`
        - web.xml
      - index.jsp
- pom.xml



main下 新建 ：

- java
    - 右键：Mark Directory as
    - Sources Root ：源码根目录
- resources
    - 右键：Mark Directory as
    - Resources Root：资源根目录



pom文件依赖

- spring-context

- spring-web
- spring-webmvc
- servlet-api
- jsp-api

```
<properties>
    <!-- 版本锁定 -->
    <spring.version>5.0.2.RELEASE</spring.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>servlet-api</artifactId>
        <version>2.5</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
```

```
        <groupId>javax.servlet.jsp</groupId>
        <artifactId>jsp-api</artifactId>
        <version>2.0</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

完整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.grape</groupId>
    <artifactId>SpringMVC001start</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>

    <name>SpringMVC001start 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>
        <!-- 版本锁定 -->
        <spring.version>5.0.2.RELEASE</spring.version>
    </properties>


    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
            <version>2.5</version>
            <scope>provided</scope>
        </dependency>
```

```
        <dependency>
            <groupId>javax.servlet.jsp</groupId>
            <artifactId>jsp-api</artifactId>
            <version>2.0</version>
            <scope>provided</scope>
        </dependency>

        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.11</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <finalName>SpringMVC001start</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>3.1.0</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>3.8.0</version>
            </plugin>
            <plugin>
              <artifactId>maven-surefire-plugin</artifactId>
              <version>2.22.1</version>
            </plugin>
            <plugin>
              <artifactId>maven-war-plugin</artifactId>
              <version>3.2.2</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>
```

JSON

- jackson.jar
- jackson-annotation-xxx.jar
- jackson-databind-xxx.jar
- jackson-core-xxx.jar

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

文件上传

- commons-fileupload-xxx.jar
- commons-io-xxx.jar

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

<dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.4</version>
</dependency>
```

跨服务器

文件上传的必备 jar 包

- jersey-core
- jersey-client

```xml
<!-- 跨服务器 -->
<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>
```

web.xml

位置？命名？

标签？文件约束？

- 最基本的，只需要一个前端控制器
    - 前端控制器，其实就是一个servlet。
        - 加载SpringMVC配置文件
            - servlet/ init-param
            - contextConfigLocation
            - classpath:springmvc.xml
    - 类是spring提供好的，打出Dispatcher就会提示出来。
        - 配置好后，正常第一次请求时创建servlet
        - load-on-startup：改为，启动服务器就创建

前端控制器

```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>dispatcherServlet</servlet-name>
    <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <!-- 加载SpringMVC配置文件 -->
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:springmvc.xml</param-value>
    </init-param>
    <!-- 配置servlet启动时加载对象 -->
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>dispatcherServlet</servlet-name>
    <!-- /的意思是，任何请求都会经过这个servlet -->
    <url-pattern>/</url-pattern>
  </servlet-mapping>
```

```
23
24  </web-app>
```

过滤器

编码过滤器

```
<!-- ****************过滤器***************** -->
<!-- 中文乱码过滤器 -->
<filter>
    <filter-name>characterEncodingFilter</filter-name>
    <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>UTF-8</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>characterEncodingFilter</filter-name>
    <!-- 全过滤 -->
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

springmvc.xml

基本xml

- 扫描包
    - context
- 视图解析器
- mvc注解支持
    - mvc

```
<?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spring创建容器时要扫描的包 -->
    <context:component-scan base-package="com.learn"></context:component-scan>
    <!-- 配置视图解析器 -->
```

```
    <bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/pages/"></property>
        <property name="suffix" value=".jsp"></property>
    </bean>
    <!-- 配置spring开启注解mvc的支持 -->
    <mvc:annotation-driven></mvc:annotation-driven>
</beans>
```

位置

- resources下新建即可。

名称空间&约束

必需 扫描&mvc注解&3组件

- mvc
    - mvc注解支持
    - <mvc：annotation-driven/>
        - 作用不只是支持注解
        - 还会配置上各种组件：
            - RequestMappingHandlerMapping（处理映射器）
            - RequestMappingHandlerAdapter（处理适配器）
            - ...

```
<!-- 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.ExceptionHandlerExceptionResolver"></bean>
<bean class="org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver"></bean>
<bean class="org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver">
</bean>
<!-- End -->
```

- 类型转换器，在这注册

```
<mvc:annotation-driven conversion-service="conversionService">
```

- context
  - 开启注解扫描

```
<?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spring创建容器时要扫描的包 -->
    <context:component-scan base-package="com.learn"></context:component-
scan>
    <!-- 配置视图解析器 -->
    <bean id="viewResolver"

class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/pages/"></property>
        <property name="suffix" value=".jsp"></property>
    </bean>
    <!-- 配置spring开启注解mvc的支持 -->
    <mvc:annotation-driven></mvc:annotation-driven>
</beans>
```

类型转换服务

```
<?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spring创建容器时要扫描的包 -->
    <context:component-scan base-package="com.learn"></context:component-
scan>
    <!-- 配置视图解析器 -->
    <bean id="viewResolver"

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

    <!-- 配置spring开启注解mvc的支持 -->
    <!-- 类型转换器需要在这里声明 -->
    <mvc:annotation-driven conversion-service="conversionService">


    <!-- 设置静态资源不过滤 -->
    <mvc:resources location="/css/" mapping="/css/**"/> <!-- 样式 -->
    <mvc:resources location="/images/" mapping="/images/**"/> <!-- 图片 -->
    <mvc:resources location="/js/" mapping="/js/**"/> <!-- javascript -->

</beans>
```

静态资源不过滤

```
<?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spring创建容器时要扫描的包 -->
    <context:component-scan base-package="com.learn"></context:component-scan>
    <!-- 配置视图解析器 -->
    <bean id="viewResolver"

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

    <!-- 配置spring开启注解mvc的支持 -->
    <!-- 类型转换器需要在这里声明 -->
    <mvc:annotation-driven></mvc:annotation-driven>

    <!-- 设置静态资源不过滤 -->
    <mvc:resources location="/css/" mapping="/css/**"/> <!-- 样式 -->
    <mvc:resources location="/images/" mapping="/images/**"/> <!-- 图片 -->
    <mvc:resources location="/js/" mapping="/js/**"/> <!-- javascript -->

</beans>
```

拦截器

```
<!-- 配置拦截器 -->
<mvc:interceptors>
    <!-- 配置拦截器 -->
    <mvc:interceptor>
        <!-- 要拦截的具体方法 -->
        <mvc:mapping path="/user/*"/>
        <!-- 不拦截的方法 -->
        <!--<mvc:exclude-mapping path=""/>-->
        <!-- 配置拦截器对象 -->
        <bean class="com.learn.interceptor.MyInterceptor1"></bean>
    </mvc:interceptor>

    <!-- 配置拦截器 -->
```

```
    <mvc:interceptor>
        <!-- 要拦截的具体方法 -->
        <mvc:mapping path="/**"/>
        <!-- 不拦截的方法 -->
        <!--<mvc:exclude-mapping path=""/>-->
        <!-- 配置拦截器对象 -->
        <bean class="com.learn.interceptor.MyInterceptor2"></bean>
    </mvc:interceptor>
</mvc:interceptors>
```

文件解析器

```
<!-- 配置文件解析器对象 -->
<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <property name="maxUploadSize" value="10485760"></property>
</bean>
```

异常处理器

```
<!-- 配置异常处理器 -->
<bean id="sysExceptionResolver"
class="com.learn.exception.SysExceptionResolver"></bean>
```

Controller

注解

@RequestMapping

作用：

- url —**绑定**—> 方法
- 用于建立请求 URL 和 处理请求方法 之间的对应关系。

位置：

- 类上
- 方法上

属性：

- 【value|path】
    - 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|限定 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。

        - `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。必须包含的请求头
    - 注：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。

@RequestParam

作用：

- 把 请求中指定名称的参数 给 控制器中的 形参 赋值。
- 不使用，是默认同名 赋值。

位置：

- 参数前

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。

@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。
- 注：
    - get 请求方式不适用。

位置：

- 参数前

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。

@PathVariable

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。

位置：

- 参数前

属性：

- value：用于指定 url 中占位符名称。
- required：是否必须提供占位符。

@RequestHeader

作用：

- 用于获取请求消息头。

位置：

- 参数前

属性：

- value：提供消息头名称
- required：是否必须有此消息头

@CookieValue

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。

@ModelAttribute

作用：

- 它可以用于修饰方法和参数。

位置：

- 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。
- 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。

@SessionAttribute

Model

- 一个map
- 向Model里存，它会帮我们存到request域对象中。

作用：

- 用于多次执行控制器方法间的参数共享。

位置：

- 类上

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。

@ResponseBody

作用：

- 把JavaBean对象转换成json字符串
- 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
    - Springmvc 默认用 MappingJacksonHttpMessageConverter 对 json 数据进行转换，需要加入jackson 的包。

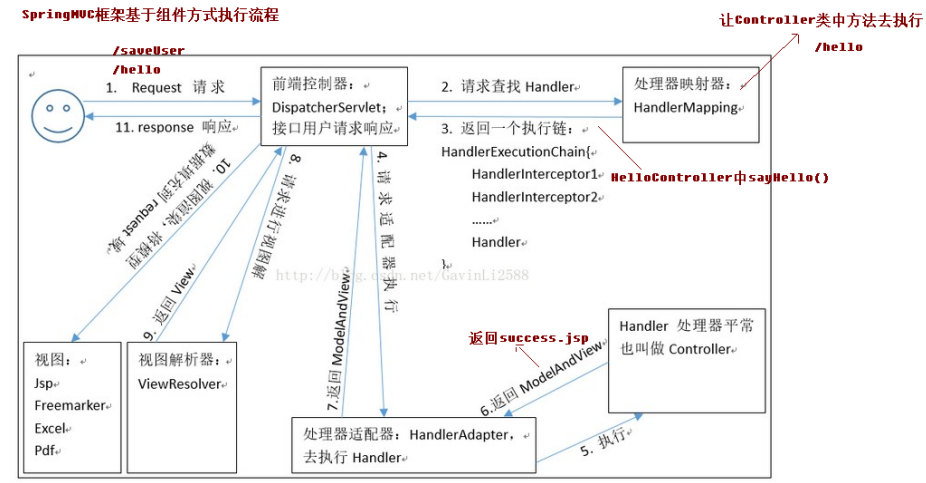
位置：

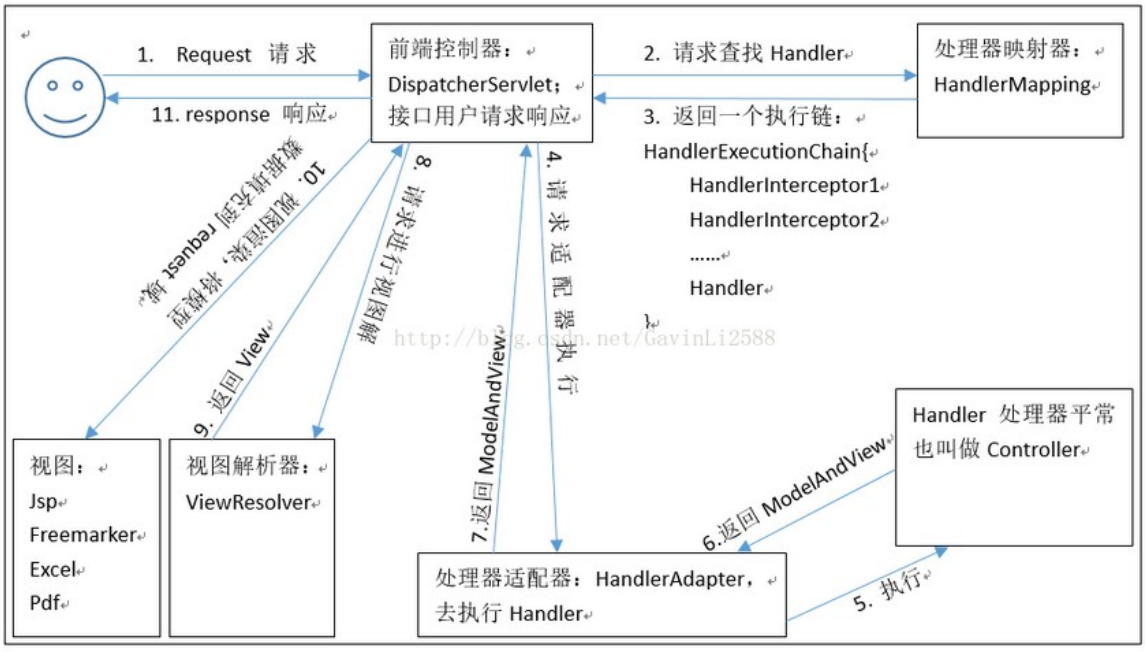
- 返回值前面

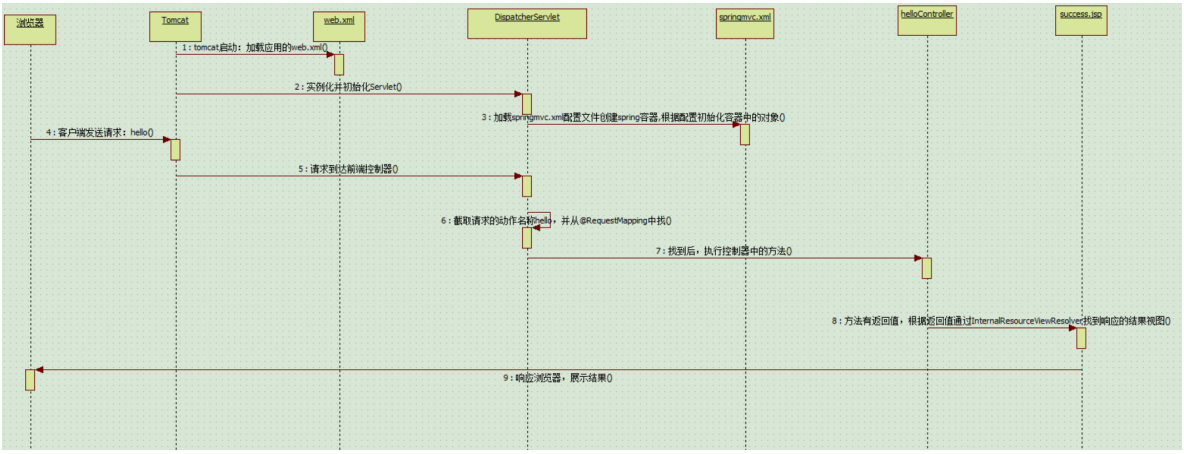
(测试类|使用)

JSP页面

组件&程序执行流程







组件

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。

- **处理器映射器**
- **处理器适配器**
- **视图解析器**

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
- 调用 其它组件 处理用户的请求
- 降低了组件之间的耦合性

HandlerMapping 处理器映射器

- 负责根据用户请求找到 Handler 即处理器(让Controller类中方法区执行/hello)
- 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...

Handler 处理器

- 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。
    - MVC的C，Controller，应用程序中处理 用户**交互** 的部分。**Servlet**
    - SpringMVC编码中实际的Controller，真的是Handler。醉了

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
- (任何Controller|Handler处理器都适配上，去执行方法。处理器Controller|Handler的类转换成适配器，任何的Controller都可以转出适配器，帮我们执行)

View Resolver 视图解析器

- View Resolver 负责将处理结果生成 View 视图：
    - 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

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...
- 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。

注解

@RequestMapping

- url —**绑定**—> 方法

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使我们的 URL 可以按照模块化管理:
        - 账户模块：
            - /account/add
            - /account/update
            - /account/delete
        - 订单模块：
            - /order/add
            - /order/update
            - /order/delete
- 方法上：
    - 请求 URL 的第二级访问目录。

属性

- 【value|path】
    - 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|限定 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。
    - `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。必须包含的请求头
- 注：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。

使用示例

```
@Controller
public class HelloController {
    @RequestMapping(path = "/hello")
    public String sayHello() {
        System.out.println("http://localhost:8080/SpringMVC001/hello");
        return "success";
    }
}

@Controller
@RequestMapping(path = "/user")
public class HelloController {
    @RequestMapping(path = "/hello")
    public String sayHello() {

System.out.println("http://localhost:8080/SpringMVC001/user/hello");
        return "success";
    }
}
```

@RequestParam

- 请求参数 usernameXX=haha
- 参数列表 username
- **不同名，也赋值**

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。
- 不使用，是默认同名 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。

使用示例

anno.jsp

- usernameXX=haha

```
<a href="/SpringMVC001/anno/testRequestParam?
usernameXX=haha">RequestParam</a>
```

AnnoController

- username

```
@RequestMapping("/testRequestParam")
public String testRequestParam(@RequestParam(name="usernameXX") String
username) {
    System.out.println("执行了");
    System.out.println(username);//haha
    return "success";
}

@RequestParam(value="age",required=false)
```

@RequestBody

主要是异步JSON。

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。
- 注：
    - 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。

使用示例

anno.jsp

```
<form action="/SpringMVC001/anno/testRequestBody" method="post">
    用户姓名:<input type="text" name="username"><br>
    用户年龄:<input type="text" name="age"><br>
    <input type="submit" value="提交">
</form>
```

AnnoController

```
@RequestMapping("/testRequestBody")
public String testRequestBody(@RequestBody String body) {//@RequestBody请求体-->body
    System.out.println("执行了");
    System.out.println(body);//username=aaa&age=111
    return "success";
}

@RequestBody(required=false)
```

响应JSON数据

@PathVariable

RESTful风格。

- /SpringMVC001/anno//testPathVariable/10
- @RequestMapping("/testPathVariable/{sid}")
- (@PathVariable(name = "sid") String id)

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。

使用示例

anno.jsp

```
<a href="/SpringMVC001/anno//testPathVariable/10">PathVariable</a>
```

AnnoController

```
@RequestMapping("/testPathVariable/{sid}")
public String testPathVariable(@PathVariable(name = "sid") String id) {
    System.out.println("执行了");
    System.out.println(id);
    return "success";
}
```

@RequestHeader

实际开发中一般不怎么用。

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

使用示例

anno.jsp

```
<a href="/SpringMVC001/anno/testRequestHeader">RequestHeader</a>
```

AnnoController

```
@RequestMapping("/testRequestHeader")
public String testRequestHeader(@RequestHeader(value = "Accept") String
header) {
    System.out.println("执行了");
    System.out.println(header);
    return "success";
}

@RequestHeader(value="Accept-Language", required=false)
```

@CookieValue

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。

使用示例

anno.jsp

```jsp
<a href="/SpringMVC001/anno/testCookieValue">CookieValue</a>
```

AnnoController

```
@RequestMapping("/testCookieValue")
public String testCookieValue(@CookieValue(value = "JSESSIONID") String
cookieValue) {
    System.out.println("执行了");
    System.out.println(cookieValue);
    return "success";
}
```

@ModelAttribute

该注解是 SpringMVC4.3 版本以后新加入的。？？？

使用说明

作用

- 它可以用于修饰方法和参数。

出现位置

- 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。
- 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。

- 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。

使用示例

anno.jsp

```
<form action="/SpringMVC001/anno/testModelAttribute" method="post">
    用户姓名:<input type="text" name="uname"><br>
    用户年龄:<input type="text" name="age"><br>
    <input type="submit" value="提交">
</form>
```

AnnoController

```
@RequestMapping("/testModelAttribute")
public String testModelAttribute(User user) {
    System.out.println("执行了");
    System.out.println(user);
    return "success";
}
@ModelAttribute
public User showUser(String uname) {
    System.out.println("showUser执行了...");
    //通过 uname 查数据库(模拟)
    User user = new User();
    user.setUname(uname);
    user.setAge(20);
    user.setDate(new Date());
    return user;
}
```

anno.jsp

```
<form action="/SpringMVC001/anno/testModelAttribute" method="post">
    用户姓名:<input type="text" name="uname"><br>
    用户年龄:<input type="text" name="age"><br>
    <input type="submit" value="提交">
</form>
```

AnnoController

```
@RequestMapping("/testModelAttribute")
```

```java
public String testModelAttribute(@ModelAttribute("abc") User user) {
    System.out.println("执行了");
    System.out.println(user);
    return "success";
}
@ModelAttribute
public void showUser(String uname, Map<String,User> map) {
    System.out.println("showUser执行了...");
    //通过 uname 查数据库(模拟)
    User user = new User();
    user.setUname(uname);
    user.setAge(20);
    user.setDate(new Date());
    map.put("abc", user);
}
```

@SessionAttribute

session域，会话域。

我们可以通过原生ServletAPI来获取，但这会造成代码耦合。

即插即用是最好的。

- Model
    - 一个map
    - 向Model里存，它会帮我们存到request域对象中。

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。

使用示例

anno.jsp

```jsp
<a href="/SpringMVC001/anno/testSessionAttributes">setSessionAttributes</a>

<a href="/SpringMVC001/anno/getSessionAttributes">getSessionAttributes</a>

<a href="/SpringMVC001/anno/delSessionAttributes">delSessionAttributes</a>
```

success.jsp

注意：开启EL表达式

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

    ${ requestScope.get("msg") }

    ${ requestScope.msg }

    ${ sessionScope }
</body>
</html>
```

AnnoController

```
@RequestMapping("/testSessionAttributes")
public String testSessionAttributes(Model model) {
    System.out.println("执行了");
    //一个map 底层会存储到request对象中
    model.addAttribute("msg", "财富密码");
    return "success";
}
```

```
@Controller
@RequestMapping("/anno")
@SessionAttributes(value = {"msg"})//存入session对象
public class AnnoController {
    //存值
    @RequestMapping("/testSessionAttributes")
    public String testSessionAttributes(Model model) {
        System.out.println("执行了");
        //一个map 底层会存储到request对象中
        model.addAttribute("msg", "财富密码");
        return "success";
    }

    //取值
    @RequestMapping("/getSessionAttributes")
    public String getSessionAttributes(ModelMap modelMap) {//实现类ModelMap
才有方法
        System.out.println("执行了");
        String msg = (String)modelMap.get("msg");
        System.out.println(msg);
        return "success";
```

```
    }

    //清除
    @RequestMapping("/delSessionAttributes")
    public String delSessionAttributes(SessionStatus status) {
        System.out.println("执行了");
        status.setComplete();//清除
        return "success";
    }
}
```

@ResponseBody

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

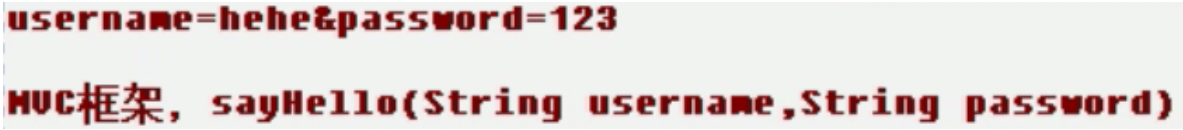
请求参数的绑定

请求参数的绑定：

- 请求会携带请求参数过来，服务端拿到数据的过程
- eg：servlet，request.getParameter()

SpringMVC 绑定请求参数的过程：

- 表单中请求参数都是基于 key=value 的。
- 把表单 提交请求参数，作为控制器中方法参数进行绑定的。
- 底层是反射实现的赋值



- SpringMVC中似乎，想用什么，在参数列表中声明一下即可。

绑定的机制

- 参数列表 直接 接收参数即可。
- "同名" 则 赋值

支持的数据类型

servlet 对象作为方法参数？

基本数据类型 & 字符串

- (表单 name == 参数列表 形参名) 则 赋值

- jsp <a 标签> ? &
    - username=hehe
    - password=123
- 方法 参数列表
    - String username
    - String password

webapp/param.jsp

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

    <%--请求参数绑定--%>
    <a href="/SpringMVC001/param/testParam?username=hehe&password=123">请求参数绑定</a>

</body>
</html>
```

controller/ParamController

```java
@Controller
@RequestMapping("/param")
public class ParamController {

    /**
     * 请求参数绑定入门
     * @return
     */
    @RequestMapping("testParam")
    public String testParam(String username, String password) {
        System.out.println("用户名:" + username);
        System.out.println("密码:" + password);
        return "success";
    }
}
```

实体类型JavaBean

- (表单 name == javaBean 属性名) 则 赋值

- 表单 name
    - username
    - password
    - money
- 方法 参数列表
    - Account account
        - private String username;
        - private String password;
        - private Double money;

webapp/param.jsp

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <%--  请求参数绑定实体类型  --%>
    <form action="/SpringMVC001/param/saveAccount" method="post">
        姓名:<input type="text" name="username"><br><%-- 此处name 须和JavaBean属性相同 --%>
        密码:<input type="text" name="password"><br>
        金额:<input type="text" name="money"><br>
        <input type="submit" value="提交">
    </form>
</body>
</html>
```

domain/Account

```
public class Account implements Serializable {

    private String username;
    private String password;
    private Double money;

    //get set
    //toString
}

```

controller/ParamController

```
@Controller
@RequestMapping("/param")
public class ParamController {
    /**
     * 请求参数绑定
     *      把数据封装到JavaBean的类中
     *          参数列表写个类,完事
```

```
     * @return
     */
    @RequestMapping("/saveAccount")
    public String saveAccount(Account account) {
        System.out.println(account);
        return "success";
    }
}
```

JavaBean中包含引用类型

- (表单 name == javaBean 属性名(.属性名)) 则 赋值

- **表单** name="user.uname"
    - username
    - password
    - money
    - user.uname
    - user.age
- 方法 **参数列表**
    - Account account
        - private String username;
        - private String password;
        - private Double money;
        - private User user;
            - private String uname;
            - private Integer age;

问题：

- get中文参数 没问题，控制台打印ok
- post中文参数 乱码，控制台打印?
- 配置中文乱码过滤器，详见过滤器

webapp/param.jsp

```
<%--  请求参数绑定实体类型  --%>
<form action="/SpringMVC001/param/saveAccount" method="post">
    姓名:<input type="text" name="username"><br><%-- 此处name 须和 JavaBean属性相同 --%>
    密码:<input type="text" name="password"><br>
    金额:<input type="text" name="money"><br>
    用户姓名:<input type="text" name="user.uname"><br>
    用户年龄:<input type="text" name="user.age"><br>
    <input type="submit" value="提交">
</form>
```

domain/User

```java
public class User implements Serializable {
    private String uname;
    private Integer age;

    //get set
    //toString
}
```

domain/Account

```java
public class Account implements Serializable {

    private String username;
    private String password;
    private Double money;

    private User user;

    //get set
    //toString
}
```

controller/ParamController

```java
@Controller
@RequestMapping("/param")
public class ParamController {
    /**
     * 请求参数绑定
     *      把数据封装到JavaBean的类中
     *          参数列表写个类,完事
     * @return
     */
    @RequestMapping("/saveAccount")
    public String saveAccount(Account account) {
        System.out.println(account);
        return "success";
    }
}
```

集合数据类型List、Map...

- **表单** name
    - username
    - password
    - money
    - user.uname
    - user.age
    - list[0].uname
    - list[0].age
    - map['one'].uname
    - map['one'].age

- 方法 **参数列表**
    - Account account
        - private String username;
        - private String password;
        - private Double money;
        - private User user;
            - private String uname;
            - private Integer age;
        - List list
        - Map<String, User> map

webapp/param.jsp

```
<%--  请求参数绑定实体类型  --%>
<form action="/SpringMVC001/param/saveAccount" method="post">
    姓名:<input type="text" name="username"><br><%-- 此处name 须和 JavaBean属性相同 --%>
    密码:<input type="text" name="password"><br>
    金额:<input type="text" name="money"><br>
    用户姓名:<input type="text" name="user.uname"><br>
    用户年龄:<input type="text" name="user.age"><br>

    用户姓名:<input type="text" name="list[0].uname"><br>
    用户年龄:<input type="text" name="list[0].age"><br>
    用户姓名:<input type="text" name="map['one'].uname"><br>
    用户年龄:<input type="text" name="map['one'].age"><br>
    <input type="submit" value="提交">
</form>
```

domain/Account

```
public class Account implements Serializable {

    private String username;
    private String password;
    private Double money;

    private User user;

    private List<User> list;
    private Map<String, User> map;

    //get set
    //toString
}
```

domain/User

```java
public class User implements Serializable {
    private String uname;
    private Integer age;

    //set get
    //toString
}
```

controller/ParamController

```java
@Controller
@RequestMapping("/param")
public class ParamController {
    /**
     * 请求参数绑定
     *       把数据封装到JavaBean的类中
     *           参数列表写个类,完事
     * @return
     */
    @RequestMapping("/saveAccount")
    public String saveAccount(Account account) {
        System.out.println(account);
        return "success";
    }
}
```

自定义类型转换器

- String ——> ?
- 日期：
    - 2000/11/11 默认格式
    - 2000-11-11

- 定义一个类，实现 Converter 接口，该接口有两个泛型
    - S：表示接受的类型
    - T：表示目标类型
- 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。
    - ConversionServiceFactoryBean
        - converters：我们自定义的，注册到这里
- 在 annotation-driven 标签中引用配置的类型转换服务
    - ```xml
      <mvc:annotation-driven conversion-service="conversionService"></mvc:annotation-driven>
      ```
      ```

utils/StringToDateConverter

- 实现Converter<S , T>接口
- 重写convert方法

```
public class StringToDate implements Converter<String, Date> {

    /**
     * source传进来的字符串
     * @param source
     * @return
     */
    @Override
    public Date convert(String source) {
        //判断
        if (source == null) {
            throw new RuntimeException("请您传入数据");
        }
        DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
        //把字符串转换成日期
        try {
            //把字符串转换为日期
            return df.parse(source);
        } catch (Exception e) {
            throw new RuntimeException("数据类型转换出现错误");
        }
    }
}
```

resources/SpringMVC.xml

- bean配置类型转换器
    - property：类中属性Set< ?> converters
        - set
            - bean：注册我们自定义的类型转换器
- <mvc: annotation-driven>中属性
    - conversion-service

```
<!-- 配置类型转换器 -->
<bean id="conversionService"
class="org.springframework.context.support.ConversionServiceFactoryBean">
    <!-- 会提供很多类型转换器,已经帮我们注册好了 -->
    <!-- 注册自定义的类型转换器 -->
    <!-- converters 是该类的属性 -->
    <property name="converters" >
        <set>
            <!-- 在这里注册 -->
            <bean class="com.learn.utils.StringToDate"></bean>
        </set>
    </property>
</bean>
```

```
13
14 <!-- 配置spring开启注解mvc的支持 -->
15 <!-- 类型转换器需要在这里声明 -->
16 <mvc:annotation-driven conversion-service="conversionService">
   </mvc:annotation-driven>
```

获取原生Servlet API

- SpringMVC 还支持使用原始 ServletAPI 对象作为控制器方法的参数
- 参数列表直接写，要什么就行了
- 支持原始 ServletAPI 对象有：
    - HttpServletRequest
    - HttpServletResponse
    - HttpSession
    - java.security.Principal
    - Locale
    - InputStream
    - OutputStream
    - Reader
    - Writer

webapp/param.jsp

```
1 <h3>Servlet 原生API</h3>
2 <a href="/SpringMVC001/param/testServlet">Servlet</a>
```

controller/ParamController

```
1  @RequestMapping("/testServlet")
2  public String testServlet(HttpServletRequest request, HttpServletResponse
   response) {
3      System.out.println("执行了...");
4      System.out.println(request);
5      HttpSession session = request.getSession();
6      System.out.println(session);
7      ServletContext context = session.getServletContext();
8      System.out.println(context);
9
10     System.out.println(response);
11     return "success";
12 }
```

响应数据&结果视图

返回值类型

字符串String

- 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
- ：字符串->逻辑视图—视图解析器—>物理视图地址

使用示例

- 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
    - return "success" 指定 逻辑视图
    - 视图解析器
        - prefix|目录：/WEB-INF/pages/
        - suffix|文件后缀名：.jsp

response.jsp

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

success.jsp

```
<%@ page contentType="text/html;charset=UTF-8" language="java"
isELIgnored="false" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <h3>入门成功</h3>
    ${user.username}
    ${user.password}
</body>
</html>
```

java

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

    @RequestMapping("/testString")
    public String testString(Model model) {//参数列表声明model
        System.out.println("testString方法执行了");
        //模拟从数据库中查询出User对象
```

```
        User user = new User();
        user.setUsername("财富密码");
        user.setPassword("123");
        user.setAge(30);
        //model对象 存入request域
        model.addAttribute("user", user);
        return "success";
    }
}
```

void

- 没有返回值，默认 请求转发路径.jsp：testVoid.jsp

jsp

```
<a href="user/testVoid">testVoid</a>
```

java

```
@RequestMapping("/testVoid")//没有返回值，默认请求转发路径.jsp：testVoid.jsp
public void testVoid(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
    System.out.println("testString方法执行了");
    //请求转发
    //request.getRequestDispatcher("/WEB-
INF/pages/success.jpg").forward(request, response);

    //重定向
    //response.sendRedirect(request.getContextPath() + "/index.jsp");

    //直接进行响应
    response.setCharacterEncoding("UTF-8");
    response.setContentType("text/html;charset=UTF-8");//浏览器打开,解析的编码
    response.getWriter().print("你好");

    return;
}
```

ModelAndView

- ModelAndView：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。
    - **addObject**：添加模型到该对象中。存数据到域对象
        - mv.addObject("user", user);
        - 页面上可以直接用el表达式获取
    - **setViewName**：用预设值逻辑视图名称。
        - mv.setViewName("success");
        - 视图解析器会根据名称前往指定的视图

- 返回 ModelAndView 类型时，浏览器跳转只能是请求转发。

jsp

```
<a href="user/testModelAndView">testModelAndView</a>
```

java

```
@RequestMapping("/testModelAndView")
public ModelAndView testModelAndView() {
    System.out.println("testString方法执行了");
    ModelAndView mv = new ModelAndView();
    //模拟从数据库中查询出User对象
    User user = new User();
    user.setUsername("财富密码");
    user.setPassword("123");
    user.setAge(30);
    //把user对象存储到mv对象中，也会把user对象存入到request对象
    mv.addObject("user", user);
    //跳转到哪个页面
    //通过视图解析器
    mv.setViewName("success");
    return mv;
}
```

响应JSON数据

先需要搭建异步环境。

- @RequestBody
    - 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。
- @ResponseBody
    - 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
    - Springmvc 默认用 MappingJacksonHttpMessageConverter 对 json 数据进行转换，需要加入jackson 的包。
- jackson.jar
- JSON.stringify(data)

环境搭建

导入JQ

- webapp下创建js
- 放入jquery-min.js

- jsp文件 通过script的src引入，干过100遍了

```
<script src="js/jquery.min.js"></script>
```

静态资源不过滤

- 问题：前端控制器DispatcherServlet， 会拦截到所有的资源，导致一个问题就是静态资源（ img、css、js ）也会被拦截到，从而
  不能被使用。
    - / 任何资源都会被拦截到
    - 配置前端控制器 哪些资源 不拦截
    - springmvc.xml中加入
        - mvc:resources
            - mapping：请求映射
        - location：文件所在位置
- 问题：有时候配置完了还是会找不到，可能是加载慢的问题。这时应该去看看工作目录是否同步

```
<!-- 开启SpringMVC框架注解的支持 -->
<mvc:resources mapping="/js/**" location="/js/**"></mvc:resources>
```

```
<?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spring创建容器时要扫描的包 -->
    <context:component-scan base-package="com.learn"></context:component-scan>
    <!-- 配置视图解析器 -->
    <bean id="viewResolver"

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

    <!-- 配置spring开启注解mvc的支持 -->
    <!-- 类型转换器需要在这里声明 -->
    <mvc:annotation-driven></mvc:annotation-driven>

    <!-- 设置静态资源不过滤 -->
```

```
    <mvc:resources location="/css/" mapping="/css/**"/> <!-- 样式 -->
    <mvc:resources location="/images/" mapping="/images/**"/> <!-- 图片 -->
    <mvc:resources location="/js/" mapping="/js/**"/> <!-- javascript -->

</beans>
```

接收JSON字符串

jsp

```
<script src="js/jquery.min.js"></script>
<script>
    //加载页面,绑定单击事件
    $(function () {
        $("#btn").click(function () {
            //发送ajax请求
            $.ajax(
                {
                    //编写json格式,设置属性和值
                    url:"user/testAjax",
                    contentType:"application/json;charset=UTF-8",
                    data:'{"username":"hehe", "password":"123","age":30}',
                    dataType:"json",
                    type:"post",
                    success:function (data) {
                        //data:服务器端响应的数据,这里是json

                    }
                }
            );
        });
    });
</script>

<body>
    <button id="btn">发送AJAX请求</button>
</body>
```

java

- @RequestBody
- 接收字符串

```
@RequestMapping("/testAjax")
public void testAjax(@RequestBody String body) {//@RequestBody请求体-->body
    System.out.println("testAjax方法执行了");
    System.out.println(body);
}
```

串->JSON->串

https://www.cnblogs.com/zhujiabin/p/5122950.html

以前，一直以为在SpringMVC环境中，@RequestBody接收的是一个Json对象，一直在调试代码都没有成功，后来发现，其实 @RequestBody接收的是一个Json对象的字符串，而不是一个Json对象。然而在ajax请求往往传的都是Json对象，后来发现用 JSON.stringify(data)的方式就能将对象变成字符串。同时ajax请求的时候也要指定dataType: "json",contentType:"application/json" 这样就可以轻易的将一个对象或者List传到Java端，使用@RequestBody即可绑定对象或者List。

依赖 pom

- jackson-annotation-xxx.jar
- jackson-databind-xxx.jar
- jackson-core-xxx.jar

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

jsp

- data不能传 JSON对象，必须传字符串。
- JSON.stringify(data) ：将对象转为字符串

```
<script>
    //加载页面,绑定单击事件
    $(function () {
        $("#btn").click(function () {
            //发送ajax请求
            $.ajax(
                {
                    //编写json格式,设置属性和值
                    url:"user/testAjax",
                    contentType:"application/json;charset=UTF-8",
                    data:'{"username":"hehe", "password":"123","age":30}',
                    // data:{"username":"hehe", "password":"123","age":30},
不是字符串就报错
                    dataType:"json",
                    type:"post",
                    success:function (data) {
                        //data:服务器端响应的数据,这里是json
                        alert(data);
                        alert(data.username);
                        alert(data.password);
```

```
                        alert(data.age);
                    }
                }
            );
        });
    });
</script>
```

java

- @RequestBody
    - 把json的字符串转换成JavaBean的对象
- @ResponseBody
    - 把JavaBean对象转换成json字符串

```
@RequestMapping("/testAjax")
public @ResponseBody User testAjax(@RequestBody User user) {
    System.out.println("testAjax方法执行了");
    //客户端 发送ajax请求 传的是 json字符串
    //后端把 json字符串 封装到 对象user中 @RequestBody
    System.out.println(user);

    //模拟查询数据库
    user.setUsername("hhhh");
    user.setAge(40);
    //做响应,返回的是对象,但jsp那里接收的是JSON。
    //@ResponseBody转成JSON串,?可是我在jsp页面接收到的是Object
    return user;
}
```

转发&重定向

- 关键字来表示转发和重定向。

jsp

```
<a href="user/testForwardOrRedirect">testForwardOrRedirect</a>
```

java

```java
@RequestMapping("/testForwardOrRedirect")
public String testForwardOrRedirect(Model model) {
    System.out.println("testForwardOrRedirect方法执行了");

    //请求的转发,不能使用视图解析器
    //        return "forward:/WEB-INF/pages/success.jsp";

    //重定向.请求不到上面的
    return "redirect:/index.jsp";
}
```

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。
- 请求的转发,不能使用视图解析器；
    - 用了 forward ：则路径必须写成实际视图 url，不能写逻辑视图。
- 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。

```java
return "forward:/WEB-INF/pages/success.jsp";
```

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:
- 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。

```java
return "redirect:/index.jsp";
```

? 这里重定向请求不到success.jsp?我该研究一下

过滤器

- 配置：web.xml
- filter
    - filter-name
    - filter-class
    - init-param
        - param-name
        - param-value
- filter-mapping

    - filter-name
    - filter-mapping

编码过滤器

- 解决中文乱码问题。
- post乱码：以前Servlet时，都是CharacterEncoding设置。
    - 猜测，这里过滤是全加了这句

- filter
    - filter-name
    - filter-class：CharacterEncodingFilter
    - init-param
        - param-name：encoding，CharacterEncodingFilter类中属性
        - param-value：UTF-8
- filter-mapping
    - filter-name
    - filter-mapping：/*

```
<!-- ****************过滤器**************** -->
<!-- 中文乱码过滤器 -->
<filter>
    <filter-name>characterEncodingFilter</filter-name>
    <filter-
class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>UTF-8</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>characterEncodingFilter</filter-name>
    <!-- 全过滤 -->
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

servlet解决

```
? ? ?
response.setCharacterEncoding("utf-8");
response.setContentType("application/json;charset=utf-8");
```

HiddentHttpMethodFilter

表单只能 使用get | post。

修改请求 方法。

使用太麻烦，以后用WebClient(WebService)使用静态方法发送请求。

拦截器

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。
- 它也是 AOP 思想的具体应用。
- 要想自定义拦截器， 要求必须实现：HandlerInterceptor 接口。

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。

过滤器？拦截器 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 **访问的控制器** 方法
    - 如果访问的是 jsp，html,css,image 或者 js 是不会进行拦截的。

springmvc.xml里配一下，完事。没有多余依赖配置

细节

拦截器方法说明

preHandle

- controller方法执行前，进行拦截的方法
- return true放行
    - 执行下一个拦截器，如果没有拦截器，执行controller中的方法。
- return false拦截
    - 不会执行controller中的方法。
- 可以使用转发或者重定向直接跳转到指定的页面

postHandle

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。

- controller方法执行后执行的方法，在JSP视图执行前
- 可以使用request或者response跳转到指定的页面
- 如果指定了跳转的页面，那么controller方法跳转的页面将不会显示

afterCompletion

只有 preHandle 返回 true 才调用

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

- 在JSP执行后执行
- request或者response不能再跳转页面了

拦截器的作用路径

拦截器配置

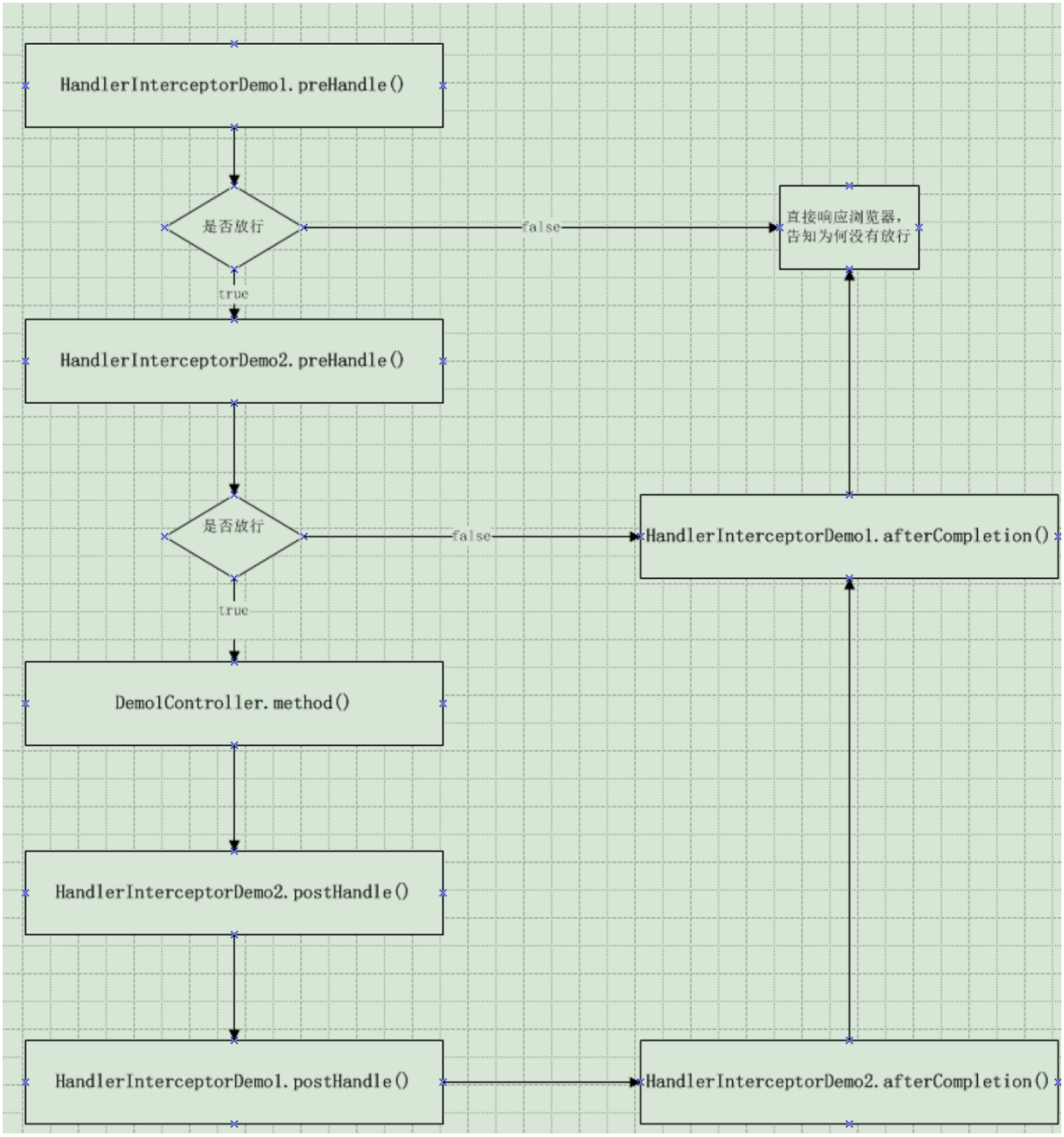
```
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/**" /><!--  用于指定对拦截的 url -->
        <mvc:exclude-mapping path=""/><!--  用于指定排除的 url-->
        <bean id="handlerInterceptorDemo1"
        class="com.learn.interceptor.HandlerInterceptorDemo1"></bean>
    </mvc:interceptor>
</mvc:interceptors>
```

多个拦截器

执行顺序

多个拦截器是按照配置的顺序决定的。

```
拦截器1：preHandle拦截器拦截了
拦截器2：preHandle拦截器拦截了
控制器中的方法执行了
拦截器2：postHandle方法执行了
拦截器1：postHandle方法执行了
拦截器2：afterCompletion方法执行了
拦截器1：afterCompletion方法执行了
```



配置

```
<!-- 配置拦截器 -->
<mvc:interceptors>
    <mvc:interceptor>
        <!-- 哪些方法进行拦截 -->
        <mvc:mapping path="/user/*"/>
        <!-- 哪些方法不进行拦截
            <mvc:exclude-mapping path=""/>
            -->
        <!-- 注册拦截器对象 -->
        <bean class="cn.itcast.demo1.MyInterceptor1"/>
    </mvc:interceptor>
    <mvc:interceptor>
        <!-- 哪些方法进行拦截 -->
        <mvc:mapping path="/**"/>
        <!-- 注册拦截器对象 -->
        <bean class="cn.itcast.demo1.MyInterceptor2"/>
    </mvc:interceptor>
</mvc:interceptors>
```

依赖pom

pom

最简单的，跟Exception一样

```
<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>
    <!-- 版本锁定 -->
    <spring.version>5.0.2.RELEASE</spring.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>servlet-api</artifactId>
        <version>2.5</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>javax.servlet.jsp</groupId>
        <artifactId>jsp-api</artifactId>
        <version>2.0</version>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.11</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

实现

controller

```
@Controller
@RequestMapping("/user")
public class UserController {
    @RequestMapping("/testInterceptor")
    public String testInterceptor() {
        System.out.println("testInterceptor执行了...");
        return "success";
    }
}
```

success.jsp

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <h3>执行成功</h3>
    <% System.out.println("success.jsp执行了..."); %>
</body>
</html>
```

index.jsp

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <a href="user/testInterceptor">拦截器</a>
</body>
</html>
```

自定义拦截器

- 编写一个普通类实现 HandlerInterceptor 接口

自定义拦截器1

```
public class MyInterceptor1 implements HandlerInterceptor {
    /**
     * 预处理 Controller方法执行前
     * return true 放行，执行下一个拦截器，如果没有，执行Controller中的方法
     * return false 不放行
     * @param request
     * @param response
     * @param handler
```

```
     * @return
     * @throws Exception
     */
    @Override
    public boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception {
        System.out.println("MyInterceptor1执行了");
        return true;
    }

}
```

自定义拦截器2

```
public class MyInterceptor1 implements HandlerInterceptor {
    /**
     * 预处理 Controller方法执行前
     * return true 放行，执行下一个拦截器，如果没有，执行Controller中的方法
     * return false 不放行
     * @param request
     * @param response
     * @param handler
     * @return
     * @throws Exception
     */
    @Override
    public boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception {
        System.out.println("MyInterceptor1执行了");
        //不放行,请求转发
        //这里加/ http://localhost:8080/user/testInterceptor
        //不加/   /user/WEB-INF/pages/error.jsp
        request.getRequestDispatcher("WEB-
INF/pages/error.jsp").forward(request, response);
        return false;
    }
}
```

自定义拦截器3

```
public class MyInterceptor1 implements HandlerInterceptor {
    /**
     * 预处理 Controller方法执行前
     * return true 放行，执行下一个拦截器，如果没有，执行Controller中的方法
     * return false 不放行
     * @param request
     * @param response
     * @param handler
     * @return
     * @throws Exception
     */
    @Override
```

```
13      public boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception {
14          System.out.println("MyInterceptor1执行了  前");
15          //不放行,请求转发
16          //这里加/ http://localhost:8080/user/testInterceptor
17          //不加/   /user/WEB-INF/pages/error.jsp
18          request.getRequestDispatcher("WEB-
INF/pages/error.jsp").forward(request, response);
19          return false;
20      }
21
22      /**
23       * 后处理方法 Controller方法执行后,seccess.jsp执行之前
24       * @param request
25       * @param response
26       * @param handler
27       * @param modelAndView
28       * @throws Exception
29       */
30      @Override
31      public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
32          System.out.println("MyInterceptor1执行了  后");
33      }
34
35      /**
36       * success.jsp页面执行后,该方法会执行
37       * @param request
38       * @param response
39       * @param handler
40       * @param ex
41       * @throws Exception
42       */
43      @Override
44      public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex) throws
Exception {
45          System.out.println("MyInterceptor1执行了  最后");
46      }
47  }
```

配置拦截器

springmvc.xml

配置拦截器

```
<!-- 配置拦截器 -->
<mvc:interceptors>
    <!-- 配置拦截器 -->
    <mvc:interceptor>
        <!-- 要拦截的具体方法 -->
        <mvc:mapping path="/user/*"/>
        <!-- 不拦截的方法 -->
        <!--            <mvc:exclude-mapping path=""/>-->
        <!-- 配置拦截器对象 -->
        <bean class="com.learn.interceptor.MyInterceptor1"></bean>
    </mvc:interceptor>
</mvc:interceptors>
```

```
<?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spring创建容器时要扫描的包 -->
    <context:component-scan base-package="com.learn"></context:component-scan>
    <!-- 配置视图解析器 -->
    <bean id="viewResolver"

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

    <mvc:resources mapping="/js/" location="/js/**"/>

    <!-- 配置拦截器 -->
    <mvc:interceptors>
        <!-- 配置拦截器 -->
        <mvc:interceptor>
            <!-- 要拦截的具体方法 -->
            <mvc:mapping path="/user/*"/>
            <!-- 不拦截的方法 -->
<!--            <mvc:exclude-mapping path=""/>-->
            <!-- 配置拦截器对象 -->
            <bean class="com.learn.interceptor.MyInterceptor1"></bean>
        </mvc:interceptor>
    </mvc:interceptors>


```

```
    <!-- 配置spring开启注解mvc的支持 -->
    <!-- 类型转换器需要在这里声明 -->
    <mvc:annotation-driven></mvc:annotation-driven>

    <!-- 设置静态资源不过滤 -->
<!--    <mvc:resources location="/css/" mapping="/css/**"/> &lt;!&ndash; 样式 &ndash;&gt;-->
<!--    <mvc:resources location="/images/" mapping="/images/**"/> &lt;!&ndash; 图片 &ndash;&gt;-->
    <!-- javascript -->

</beans>
```

多个拦截器

拦截器

```
public class MyInterceptor2 implements HandlerInterceptor {
    /**
     * 预处理 Controller方法执行前
     * return true 放行, 执行下一个拦截器, 如果没有, 执行Controller中的方法
     * return false 不放行
     * @param request
     * @param response
     * @param handler
     * @return
     * @throws Exception
     */
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        System.out.println("》》》》》MyInterceptor2执行了  前");
        //不放行,请求转发
        //这里加/ http://localhost:8080/user/testInterceptor
        //不加/   /user/WEB-INF/pages/error.jsp
        request.getRequestDispatcher("WEB-INF/pages/error.jsp").forward(request, response);
        return false;
    }

    /**
     * 后处理方法 Controller方法执行后,seccess.jsp执行之前
     * @param request
     * @param response
     * @param handler
     * @param modelAndView
     * @throws Exception
     */
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {
        System.out.println("》》》》》MyInterceptor2执行了  后");
    }

```

```
    /**
     * success.jsp页面执行后,该方法会执行
     * @param request
     * @param response
     * @param handler
     * @param ex
     * @throws Exception
     */
    @Override
    public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex) throws
Exception {
        System.out.println("》》》》》MyInterceptor2执行了  最后");
    }
}
```

springmvc.xml

配置第二个拦截器。

```
<!-- 配置拦截器 -->
<mvc:interceptors>
    <!-- 配置拦截器 -->
    <mvc:interceptor>
        <!-- 要拦截的具体方法 -->
        <mvc:mapping path="/user/*"/>
        <!-- 不拦截的方法 -->
        <!--<mvc:exclude-mapping path=""/>-->
        <!-- 配置拦截器对象 -->
        <bean class="com.learn.interceptor.MyInterceptor1"></bean>
    </mvc:interceptor>

    <!-- 配置拦截器 -->
    <mvc:interceptor>
        <!-- 要拦截的具体方法 -->
        <mvc:mapping path="/**"/>
        <!-- 不拦截的方法 -->
        <!--<mvc:exclude-mapping path=""/>-->
        <!-- 配置拦截器对象 -->
        <bean class="com.learn.interceptor.MyInterceptor2"></bean>
    </mvc:interceptor>
</mvc:interceptors>
```

文件上传

文件上传的必要前提

- 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
- method 属性取值必须是 Post
- 提供一个文件选择域 `<input type="file" />`

文件上传原理分析

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）

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

第三方组件依赖

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-xxx.jar
- commons-io-xxx.jar

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。

传统方式

E:\apache-tomcat-8.5.50\webapps\ROOT\uploads

传统方式的文件上传，指的是：

- 我们上传的文件和访问的应用存在于同一台服务器上。
- 并且上传完成之后，浏览器可能跳转。

- DiskFileItemFactory
- ServletFileUpload
- FileItem

依赖pom

加入依赖

- commons-fileupload-xxx.jar
- commons-io-xxx.jar

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

<dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.4</version>
</dependency>
```

完整版

```
<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>
    <!-- 版本锁定 -->
    <spring.version>5.0.2.RELEASE</spring.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>servlet-api</artifactId>
        <version>2.5</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>javax.servlet.jsp</groupId>
```

```
        <artifactId>jsp-api</artifactId>
        <version>2.0</version>
        <scope>provided</scope>
    </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-core</artifactId>
        <version>2.9.0</version>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-annotations</artifactId>
        <version>2.9.0</version>
    </dependency>

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

    <dependency>
        <groupId>commons-io</groupId>
        <artifactId>commons-io</artifactId>
        <version>2.4</version>
    </dependency>

    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.11</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

代码编写

index.jsp

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <h3>文件上传</h3>

    <form action="/user/fileupload1" method="post" enctype="multipart/form-data">
        选择文件:<input type="file" name="upload"><br>
```

```
        <input type="submit" value="上传">
    </form>
</body>
</html>
```

success.jsp

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <h3>上传文件成功</h3>
</body>
</html>
```

java

```
@RequestMapping("/fileupload1")
public String fileupload1(HttpServletRequest request) throws Exception {
    System.out.println("文件上传");

    //使用fileupload组件完成组件上传
    //上传的位置
    String path =
request.getSession().getServletContext().getRealPath("/uploads/");
    System.out.println(path);
    //判断该路径是否存在
    File file = new File(path);
    if (!file.exists()) {
        //创建该文件夹
        file.mkdirs();
    }

    //解析request对象,获取上传文件项
    DiskFileItemFactory factory = new DiskFileItemFactory();
    ServletFileUpload upload = new ServletFileUpload(factory);
    //解析request
    List<FileItem> fileItems = upload.parseRequest(request);
    //遍历
    for (FileItem item : fileItems) {
        //进行判断,当前item对象是否是上传文件项
        if (item.isFormField()) {
            //普通表单项
        } else {
            //上传文件项
            //获取到上传文件的名称
            String filename = item.getName();
            //把文件名 设置成唯一值
            String uuid = UUID.randomUUID().toString().replace("-", "");
```

```
            filename = uuid + "_" + filename;
            //完成文件上传
            item.write(new File(path, filename));
            //删除临时文件
            item.delete();
        }
    }

    return "success";
}
```

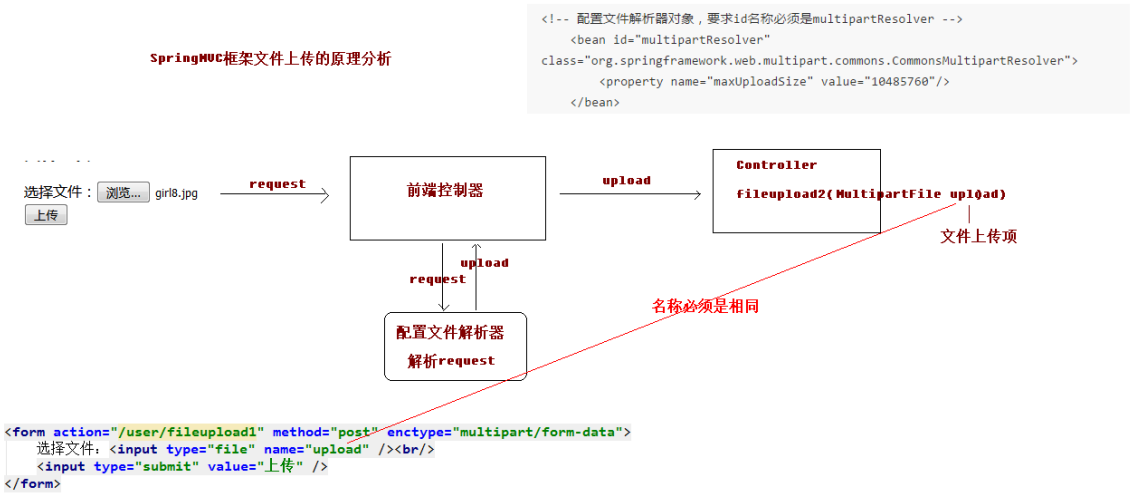
```
@RequestMapping(value="/fileupload")
public String fileupload(HttpServletRequest request) throws Exception {
    // 先获取到要上传的文件目录
    String path =
request.getSession().getServletContext().getRealPath("/uploads");
    // 创建File对象，一会向该路径下上传文件
    File file = new File(path);
    // 判断路径是否存在，如果不存在，创建该路径
    if(!file.exists()) {
        file.mkdirs();
    }
    // 创建磁盘文件项工厂
    DiskFileItemFactory factory = new DiskFileItemFactory();
    ServletFileUpload fileUpload = new ServletFileUpload(factory);
    // 解析request对象
    List<FileItem> list = fileUpload.parseRequest(request);
    // 遍历
    for (FileItem fileItem : list) {
        // 判断文件项是普通字段，还是上传的文件
        if(fileItem.isFormField()) {

        } else {
            // 上传文件项
            // 获取到上传文件的名称
            String filename = fileItem.getName();
            // 上传文件
            fileItem.write(new File(file, filename));
            // 删除临时文件
            fileItem.delete();
        }
    }
    return "success";
}
```

SpringMVC上传文件

E:\apache-tomcat-8.5.50\webapps\ROOT\uploads

- 文件解析器：CommonsMultipartResolver



- MultipartFile

springmvc.jsp

文件解析器

```
<!-- 配置文件解析器对象 -->
<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <property name="maxUploadSize" value="10485760"></property>
</bean>
```

index.jsp

```
<h3>SpringMVC方式文件上传</h3>
<form action="/user/fileupload2" method="post" enctype="multipart/form-data">
    选择文件:<input type="file" name="upload"><br>
    <input type="submit" value="上传">
</form>
```

java

```
@RequestMapping("/fileupload2")
public String fileupload2(HttpServletRequest request, MultipartFile upload)
throws Exception {
    System.out.println("SpringMVC文件上传");

    //使用fileupload组件完成组件上传
    //上传的位置
    String path =
request.getSession().getServletContext().getRealPath("/uploads/");
    System.out.println(path);
    //判断该路径是否存在
    File file = new File(path);
    if (!file.exists()) {
        //创建该文件夹
        file.mkdirs();
```

```
    }

    //上传文件项
    //获取到上传文件的名称
    String filename = upload.getOriginalFilename();
    //把文件名 设置成唯一值
    String uuid = UUID.randomUUID().toString().replace("-", "");
    filename = uuid + "_" + filename;
    //完成文件上传
    upload.transferTo(new File(path, filename));

    return "success";
}
```

```
@RequestMapping(value="/fileupload2")
public String fileupload2(HttpServletRequest request,MultipartFile upload)
throws Exception {
    System.out.println("SpringMVC方式的文件上传...");
    // 先获取到要上传的文件目录
    String path =
request.getSession().getServletContext().getRealPath("/uploads");
    // 创建File对象，一会向该路径下上传文件
    File file = new File(path);
    // 判断路径是否存在，如果不存在，创建该路径
    if(!file.exists()) {
        file.mkdirs();
    }
    // 获取到上传文件的名称
    String filename = upload.getOriginalFilename();
    String uuid = UUID.randomUUID().toString().replaceAll("-",
"").toUpperCase();
    // 把文件的名称唯一化
    filename = uuid+"_"+filename;
    // 上传文件
    upload.transferTo(new File(file,filename));
    return "success";
}
```

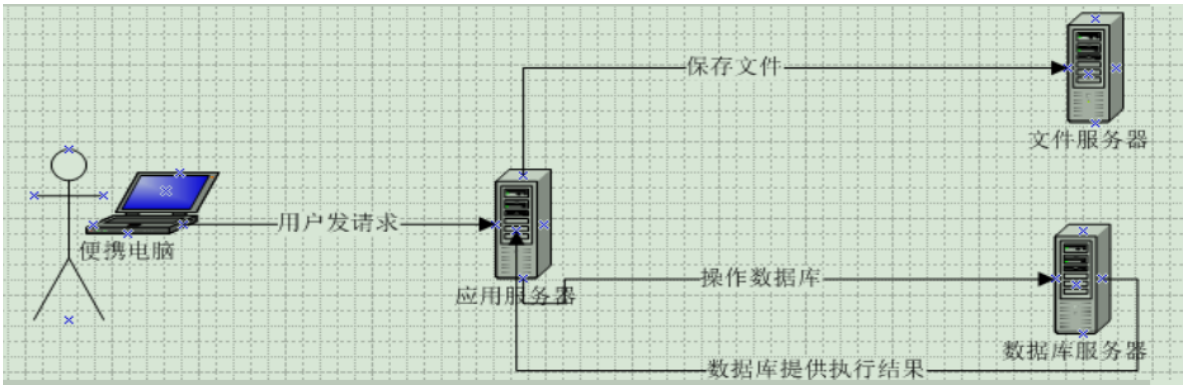
跨服务器

E:\code\SpringMVC003fileserver\target\SpringMVC003fileserver\uploads

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。



- MultipartFile
- Client
- WebResource

环境搭建

新建 空项目服务器

新建个项目，配个tomcat就行了。

充当文件服务器，我们这次往这里上传。

Tomcat web.xml

- 接收文件的目标服务器支持写入操作。

```
<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>
    <!-- 允许写入 -->
    <init-param>
        <param-name>readonly</param-name>
        <param-value>false</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

原项目依赖pom

文件上传的必备 jar 包

- jersey-core

- jersey-client

```xml
<!-- 跨服务器 -->
<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>
```

代码编写

jsp

```jsp
<h3>跨服务器文件上传</h3>
<form action="/user/fileupload3" method="post" enctype="multipart/form-data">
    选择文件：<input type="file" name="upload" /><br/>
    <input type="submit" value="上传" />
</form>
```

java

```java
@RequestMapping("/fileupload3")
public String fileupload3(MultipartFile upload) throws Exception {
    System.out.println("跨服务器 文件上传");

    //定义上传服务器的路径
    String path = "http://localhost:9090/file/uploads/";

    //上传文件项
    //获取到上传文件的名称
    String filename = upload.getOriginalFilename();
    //把文件名 设置成唯一值
    String uuid = UUID.randomUUID().toString().replace("-", "");
    filename = uuid + "_" + filename;

    //完成文件上传 跨服务器上传
    //创建客户端
    Client client = Client.create();
    //和图片服务器进行连接
    WebResource resource = client.resource(path + filename);
    //上传文件
    resource.put(upload.getBytes());

    return "success";
}
```

```java
@RequestMapping(value="/fileupload3")
public String fileupload3(MultipartFile upload) throws Exception {
    System.out.println("SpringMVC跨服务器方式的文件上传...");
    // 定义图片服务器的请求路径
    String path =
"http://localhost:9090/day02_springmvc5_02image/uploads/";
    // 获取到上传文件的名称
    String filename = upload.getOriginalFilename();
    String uuid = UUID.randomUUID().toString().replaceAll("-",
"").toUpperCase();
    // 把文件的名称唯一化
    filename = uuid+"_"+filename;
    // 向图片服务器上传文件
    // 创建客户端对象
    Client client = Client.create();
    // 连接图片服务器
    WebResource webResource = client.resource(path+filename);
    // 上传文件
    webResource.put(upload.getBytes());
    return "success";
}
```

问题

- 服务器不允许写入405
    - tomcat服务器默认是不可写操作
    - Tomcat安装目录，web.xml

```xml
<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>
    <!-- 允许写入 -->
    <init-param>
        <param-name>readonly</param-name>
        <param-value>false</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

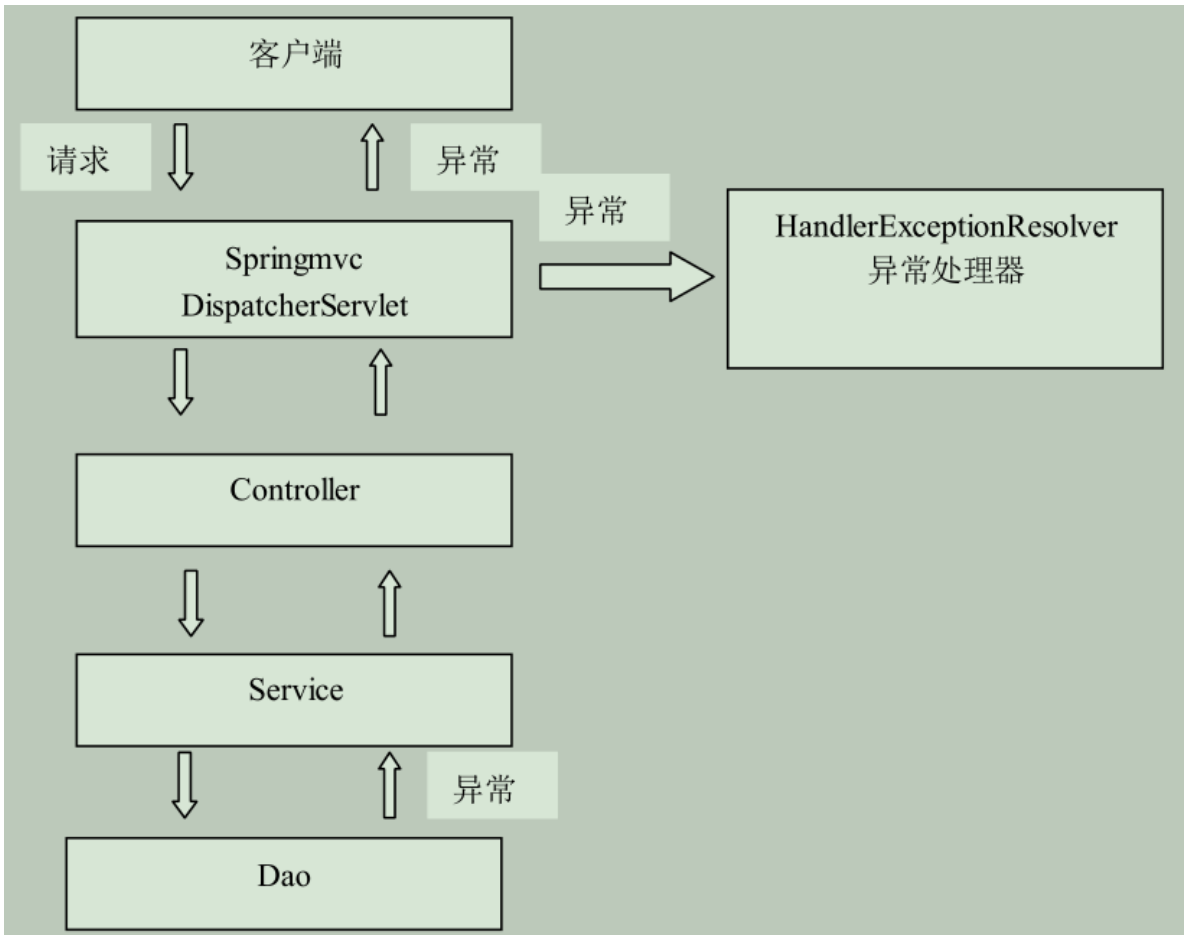
- 409 response status of 409 Conflict
    - 路径错误，即当前访问的路径下没有相关文件

异常处理

系统中异常包括两类：

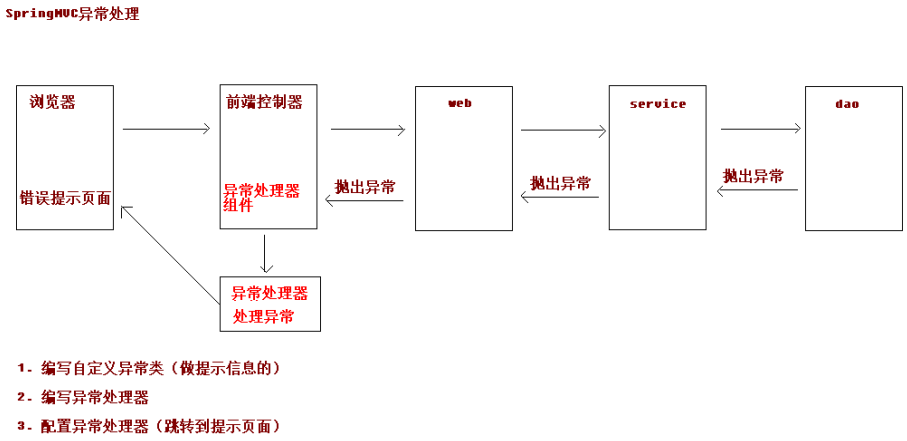
- 预期异常
    - 通过捕获异常从而获取异常信息
- 运行时异常 RuntimeException
    - 主要通过规范代码开发、测试通过手段减少运行时异常的发生

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



原理分析

异常处理器



实现步骤

springmvc.xml配置一下完事

环境搭建

最简单的

pom

```
<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>
    <!-- 版本锁定 -->
    <spring.version>5.0.2.RELEASE</spring.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>servlet-api</artifactId>
        <version>2.5</version>
```

```
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>javax.servlet.jsp</groupId>
        <artifactId>jsp-api</artifactId>
        <version>2.0</version>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.11</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

实现

自定义异常类

```
public class SysException extends Exception {
    //存储提示信息
    private String message;

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

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

    public SysException(String message) {
        this.message = message;
    }
}
```

自定义异常处理器

```
public class SysExceptionResolver implements HandlerExceptionResolver{

    /**
     *
     * @param httpServletRequest
     * @param httpServletResponse
     * @param o
     * @param ex
     * @return
     */
    @Override
```

```
    public ModelAndView resolveException(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o, Exception ex) {
        ex.printStackTrace();
        //获取到异常对象
        SysException e = null;
        if (ex instanceof SysException) {
            e = (SysException) ex;
        } else {
            e = new SysException("说点啥好呢 系统正在维护吧");
        }
        //创建ModelAndView对象,跳到error.jsp
        ModelAndView mv = new ModelAndView();
        //errorMsg
        mv.addObject("errorMsg", e.getMessage());
        mv.setViewName("error");
        return mv;
    }
}
```

springmvc.xml

- 配置自定义的异常处理器
    - 就是普通的bean

```
<!-- 配置异常处理器 -->
<bean id="sysExceptionResolver" class="com.learn.exception.SysExceptionResolver"></bean>
```

error.jsp

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

案例DEMO

EOF