1- 三层架构和MVC

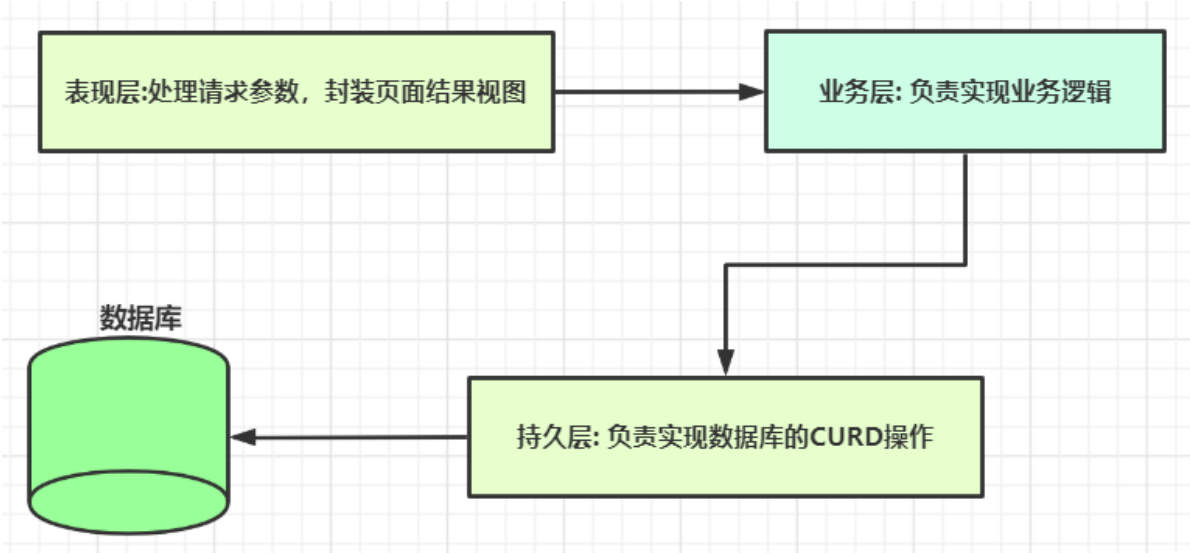
1.1 三层架构

> **企业项目架构中，有两种常见的架构形式**

- C/S架构，即客户端/服务器
- B/S架构，即浏览器/服务器
- 目前在J2EE项目中，几乎都基于B/S架构

> **在B/S架构中，系统标准的三层结构包括**

- 表现层（web）：与请求和响应相关。
- 业务层（service）：与业务需求相关。
- 持久层（dao）：与操作访问数据库相关



1.2 MVC模型

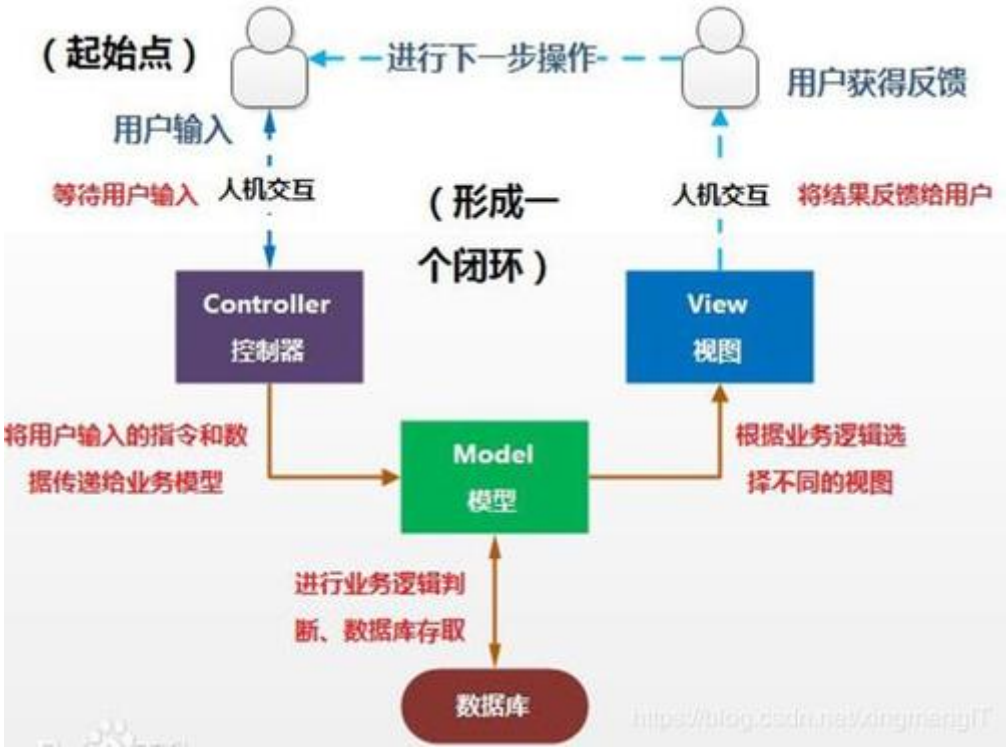
> **MVC是一种表现层的设计模型。**

- Model（模型）：模型指的是数据模型，与封装数据相关的都是模型。比如pojo、vo
- View（视图）：视图指的是展示数据，与页面相关的都是视图。比如html、jsp
- Control（控制器）：控制器指的是用户交互，接收用户请求，响应用户。比如servlet

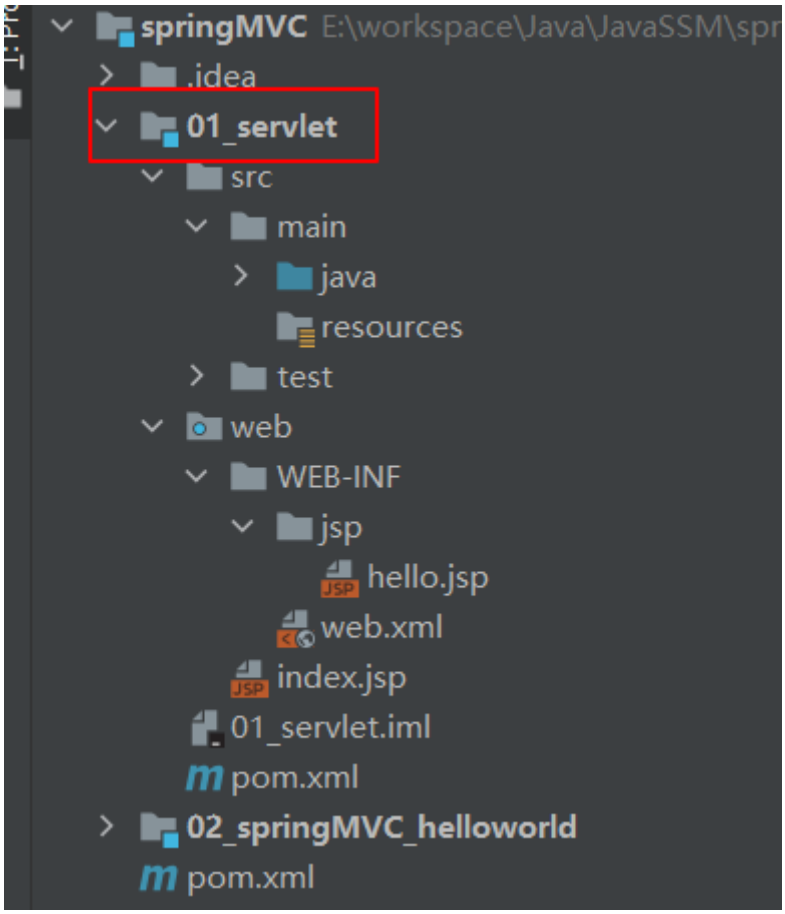
> **作用**

- MVC主要作用是降低了视图与业务逻辑间的双向偶合,MVC不是一种设计模式。
- MVC是一种架构模式。当然不同的MVC存在差异。

最典型的MVC就是JSP + servlet + javabean的模式。



1.3 项目目录



1.3.1 父工程相关依赖

```
<?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>cn.guardwhy</groupId>
<artifactId>springMVC</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>pom</packaging>

<modules>
    <module>01_servlet</module>
</modules>

<!-- 集中定义依赖版本号 -->
<properties>
    <!--spring版本-->
    <spring.version>5.2.9.RELEASE</spring.version>
</properties>

<!--引入依赖-->
<dependencies>
    <!-- Spring -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-beans</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aspects</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context-support</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>${spring.version}</version>
    </dependency>

    <!--JSP-->
    <dependency>
        <groupId>javax.servlet.jsp</groupId>
```

```
            <artifactId>jsp-api</artifactId>
            <version>2.2</version>
        </dependency>
        <!--servlet-->
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>4.0.1</version>
        </dependency>
        <!--jsp-jstl-->
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>jstl</artifactId>
            <version>1.2</version>
        </dependency>

        <!--测试依赖-->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
    </dependencies>

</project>
```

1.3.2 子工程依赖

```
<?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">

    <parent>
        <artifactId>springMVC</artifactId>
        <groupId>cn.guardwhy</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>

    <modelVersion>4.0.0</modelVersion>
    <artifactId>01_servlet</artifactId>
</project>
```

1.3.3 Servlet类

```
package cn.guardwhy.servlet;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class HelloServlet extends HttpServlet {
    @Override
```

```
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        // 1. 取得参数
        String method = req.getParameter("method");
        if(method.equals("add")){
            req.getSession().setAttribute("msg", "执行了add方法");
        }
        if(method.equals("delete")){
            req.getSession().setAttribute("msg","执行了delete方法");
        }

        // 2.视图跳转
        req.getRequestDispatcher("/WEB-INF/jsp/hello.jsp").forward(req, resp);
    }

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

1.3.4 编写Hello.jsp文件

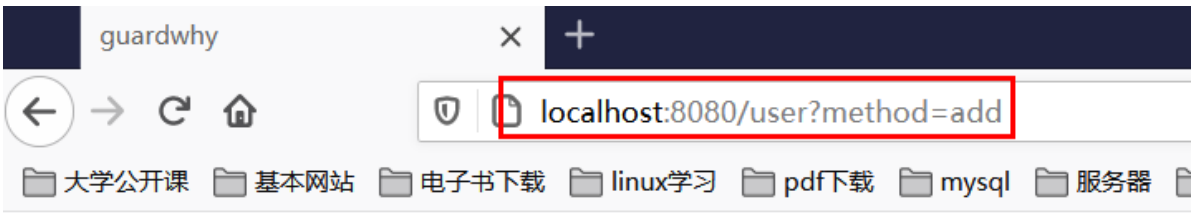
在WEB-INF目录下新建一个jsp的文件夹，新建hello.jsp

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

1.3.5 web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
         version="4.0">
    <servlet>
        <!--注册Servlet-->
        <servlet-name>HelloServlet</servlet-name>
        <servlet-class>cn.guardwhy.servlet.HelloServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>HelloServlet</servlet-name>
        <url-pattern>/user</url-pattern>
    </servlet-mapping>
</web-app>
```

1.3.6 执行结果



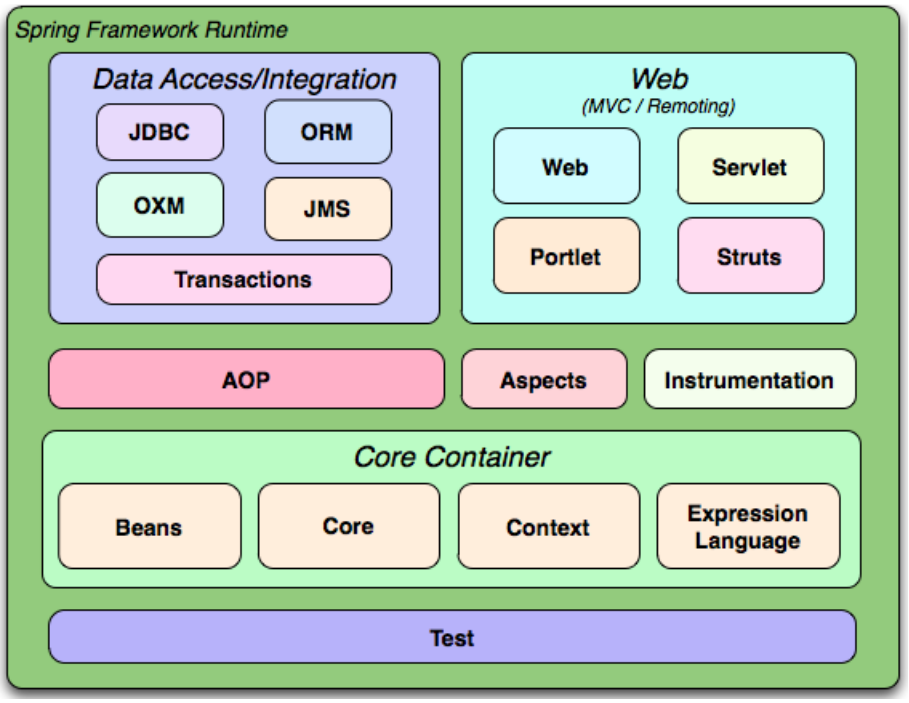
执行了add方法

2- Spring MVC原理

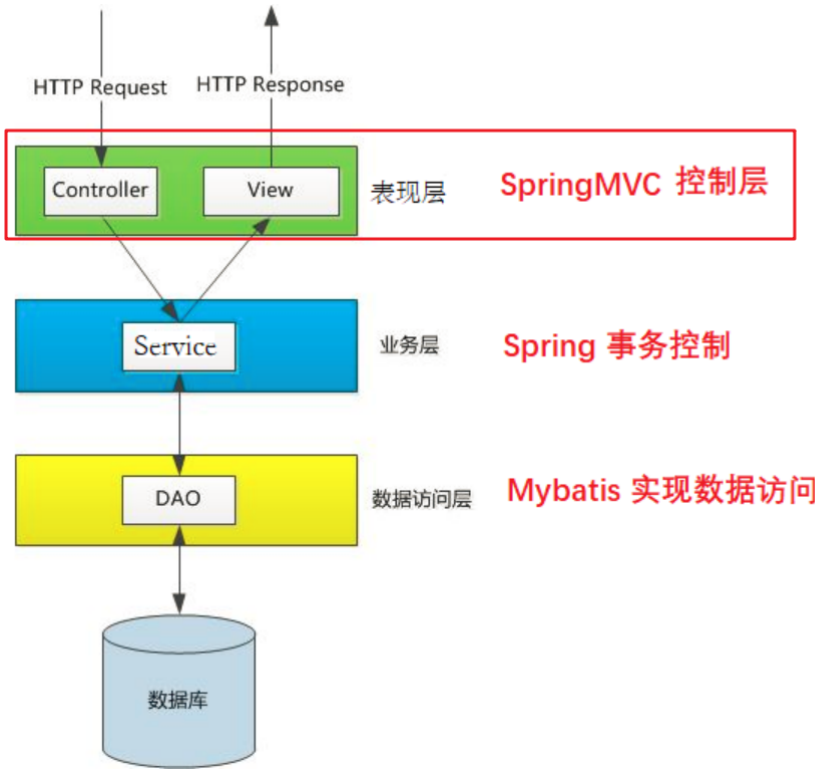
2.1 Spring MVC 基本特点

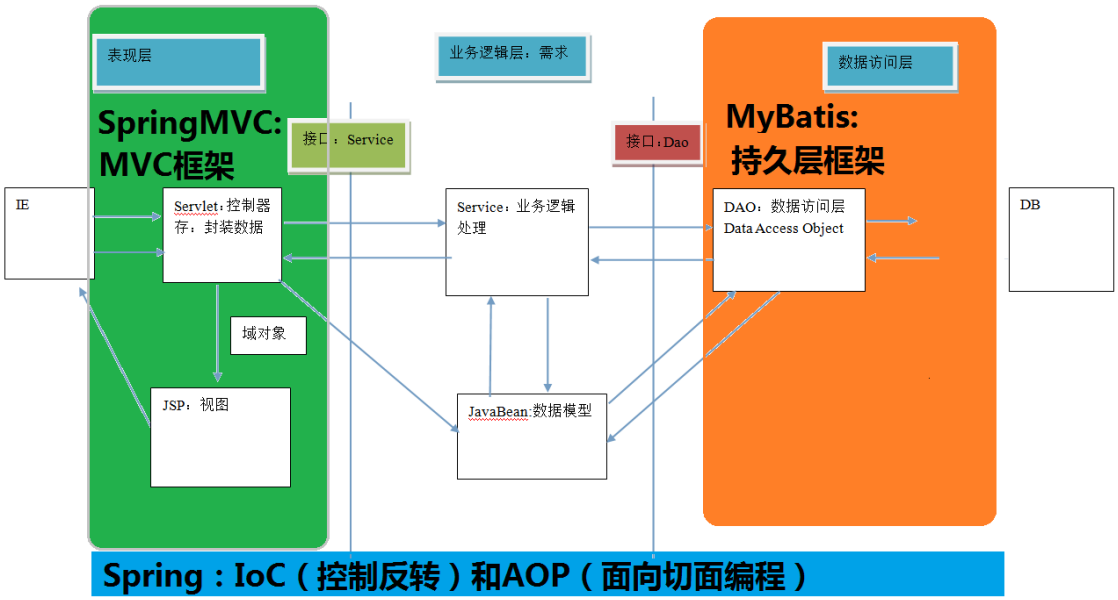
2.1.1 基本概念

- Spring MVC是 Spring体系的轻量级Web MVC框架。
- Spring MVC的核心Controller控制器，用于处理请求，产生响应。
- Spring MVC基于Spring IOC容器运行，所有对象都被IOC管理。



2.1.2 在三层架构中的位置





2.1.3 Springmvc优点

序号	优点	描述
1	清晰的角色划分	前端控制器（DispatcherServlet）、处理器映射器（HandlerMapping）、处理器适配器（HandlerAdapter）、视图解析器（ViewResolver）、后端控制器（Controller）
2	与Spring框架无缝集成	这是其它web框架不具备的
3	可扩展性好	可以很容易扩展，虽然几乎不需要
4	单元测试方便	利用 Spring 提供的 Mock 对象能够非常简单的进行 Web 层单元测试
5.	功能强大	RESTful、数据验证、格式化、本地化、主题等
6	jsp标签库	强大的 JSP 标签库，使 JSP 编写更容易

2.1.4 和Struts2比较

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:

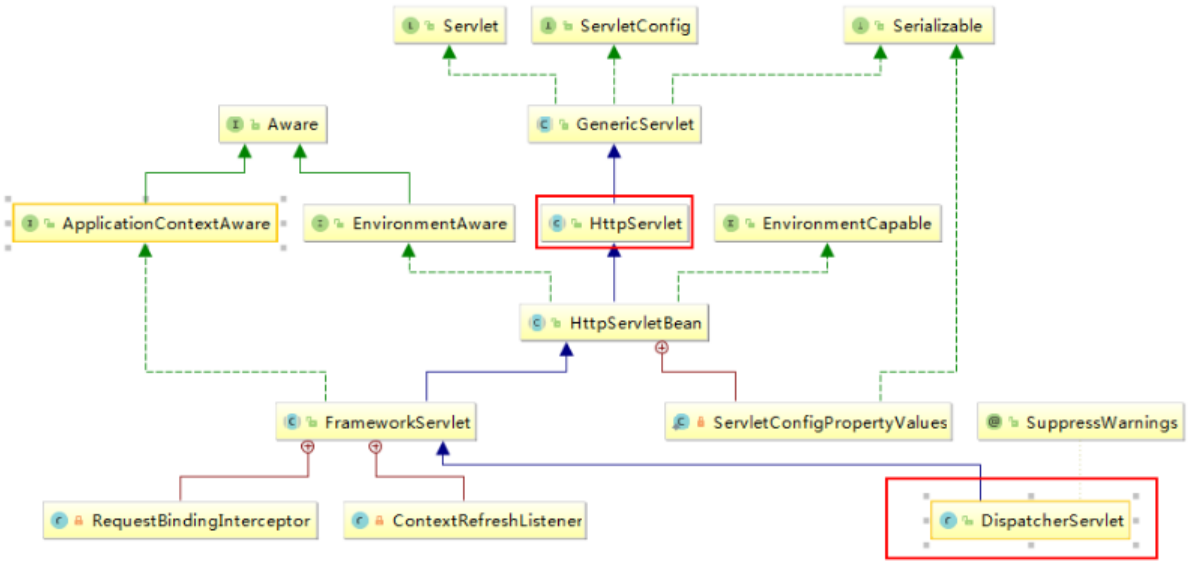
- Spring MVC 的入口是 Servlet, 而 Struts2 是 Filter。
- Spring MVC 是基于方法设计的，而 Struts2 是基于类，Struts2 每次执行都会创建一个动作类。所以 Spring MVC 会稍微比 Struts2 快些。
- Spring MVC 使用更加简洁,同时还支持 JSR303, 处理 ajax 的请求更方便。
- Struts2 的 OGNL 表达式使页面的开发效率相比 Spring MVC 更高些，但执行效率并没有比 JSTL 提升，尤其是 struts2 的表单标签，没有 html 执行效率高。

2.2 执行原理

2.2.1 前端控制器

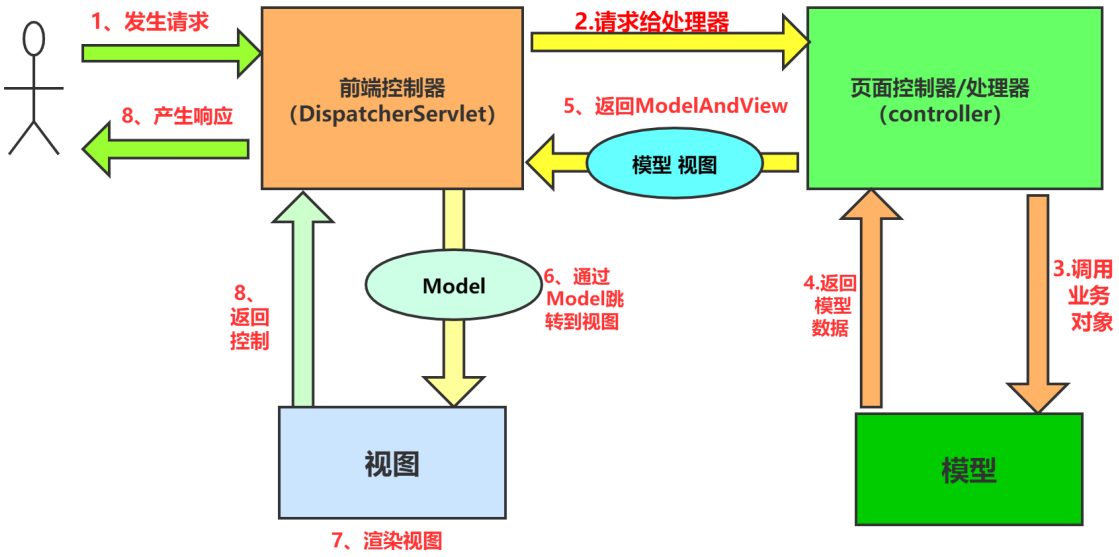
Spring的web框架围绕DispatcherServlet设计。 DispatcherServlet的作用是将请求分发到不同的处理器。

从Spring 2.5开始，使用Java 5或者以上版本的用户可以采用基于注解的controller声明方式。
Spring MVC框架像许多其他MVC框架一样, **以请求为驱动，围绕一个中心Servlet分派请求及提供其他功能，DispatcherServlet是一个实际的Servlet (它继承自HttpServlet 基类)**。



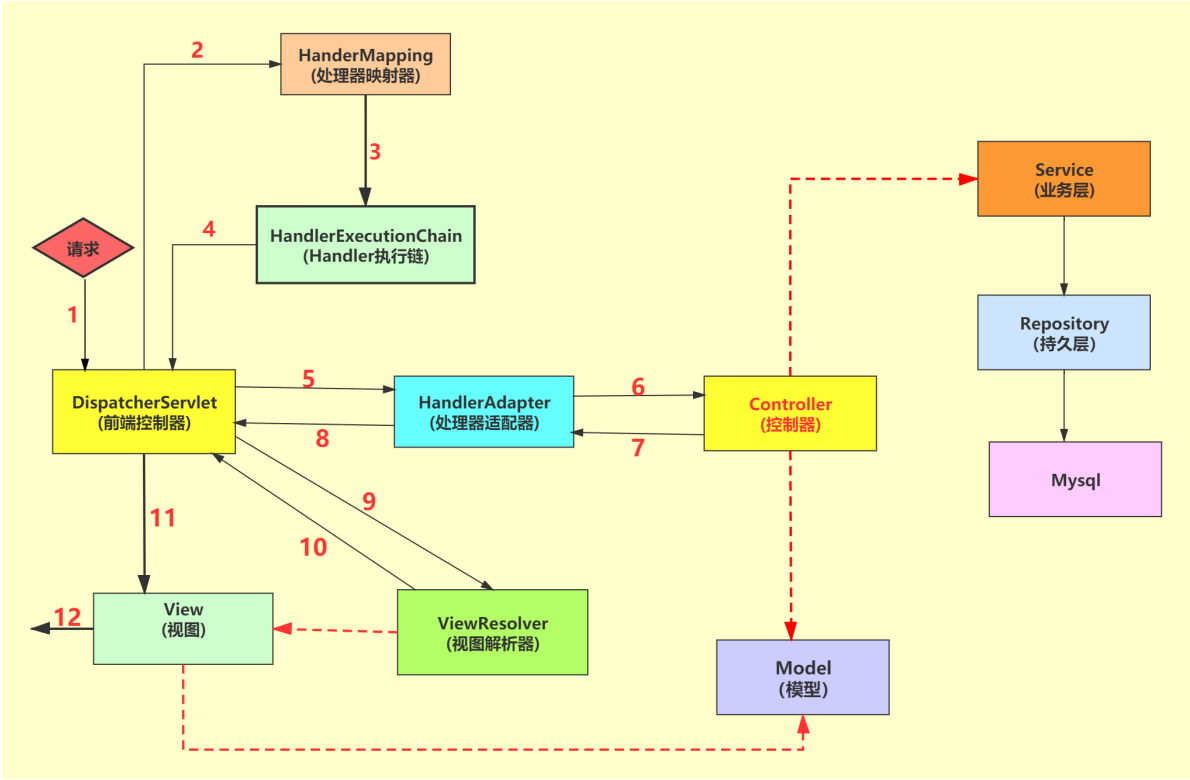
2.2.2 SpringMVC的原理

当发起请求时被前置的控制器拦截到请求，根据请求参数生成代理请求，找到请求对应的实际控制器，控制器处理请求，创建数据模型，访问数据库，将模型响应给中心控制器，控制器使用模型与视图渲染视图结果，将结果返回给中心控制器，再将结果返回给请求者。



2.2.3 SpringMVC执行原理

实线表示SpringMVC框架提供的技术，不需要开发者实现，虚线表示需要开发者实现。



2.2.4 执行流程

1、DispatcherServlet表示前置控制器，是整个SpringMVC的控制中心。用户发出请求，DispatcherServlet接收请求并拦截请求。
假设请求的url为 : http://localhost:8080/SpringMVC/success

```
如上url拆分成三部分：http://localhost:8080服务器域名。
SpringMVC部署在服务器上的web站点。success表示控制器.
通过分析，如上url表示为：请求位于服务器localhost:8080上的SpringMVC站点的hello控制器。
```

2、HandlerMapping为处理器映射。

```
DispatcherServlet调用HandlerMapping,HandlerMapping根据请求url查找Handler。
```

3、HandlerExecutionChain表示**具体的Handler(处理器)**

```
其主要作用是根据url查找控制器，如上url被查找控制器为：success。
```

4、HandlerExecutionChain将解析后的信息传递给DispatcherServlet,如解析控制器映射等。

5、HandlerAdapter表示处理器适配器，其按照特定的规则去执行Handler。

6、Handler让具体的Controller执行。

7、Controller将具体的执行信息返回给HandlerAdapter,如ModelAndView。

8、HandlerAdapter将视图逻辑名或模型传递给DispatcherServlet。

9、DispatcherServlet调用视图解析器(ViewResolver)来解析HandlerAdapter传递的逻辑视图名。

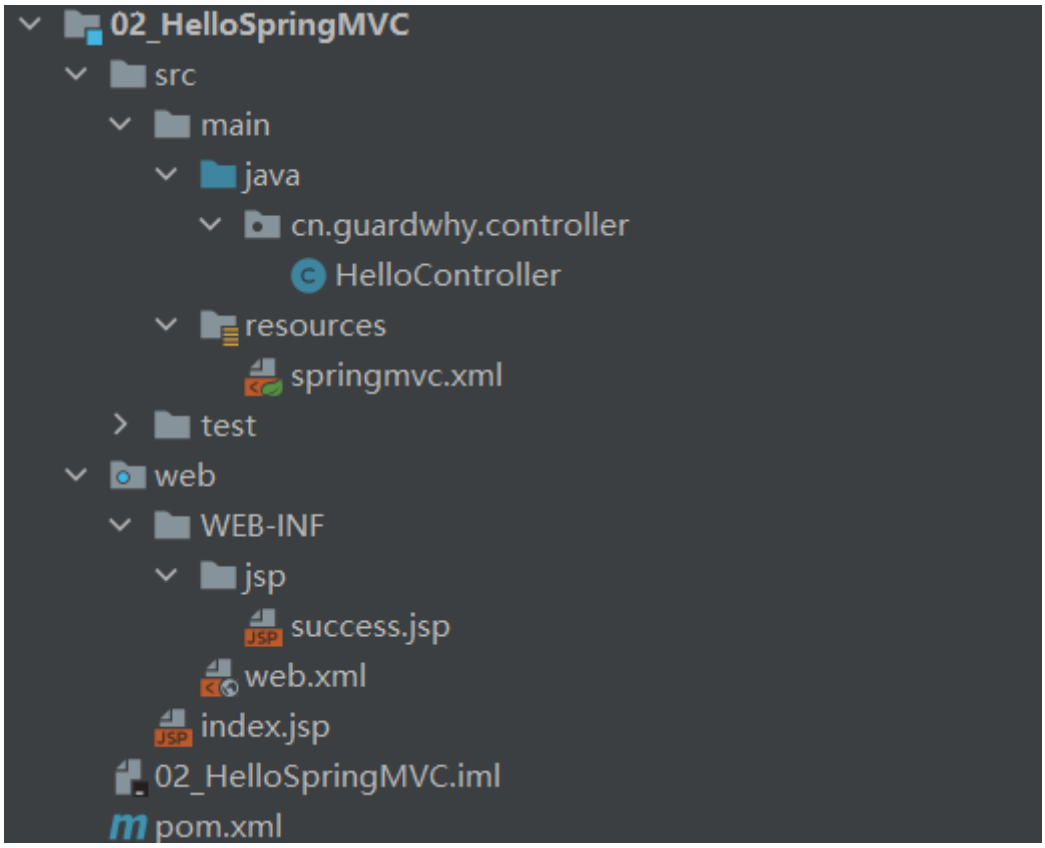
10、视图解析器将解析的逻辑视图名传给DispatcherServlet。

11、DispatcherServlet根据视图解析器解析的视图结果，调用具体的视图。

12、最终视图呈现给用户。

3- 入门案例

3.1 项目目录(原理版)



3.1.1 相关依赖

```
<?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">
    <parent>
        <artifactId>SpringMVC</artifactId>
        <groupId>cn.guardwhy</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>

    <modelVersion>4.0.0</modelVersion>
    <artifactId>01_springMVC_servlet</artifactId>
</project>
```

3.1.2 配置web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
         version="4.0">
```

```
    <!--1.注册DispatcherServlet-->
    <servlet>
        <servlet-name>springmvc</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

        <!--关联一个springmvc的配置文件-->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:springmvc.xml</param-value>
        </init-param>
        <!--启动级别-1-->
        <load-on-startup>1</load-on-startup>
    </servlet>

    <!--/ 匹配所有的请求；（不包括.jsp）-->
    <!--/* 匹配所有的请求；（包括.jsp）-->
    <servlet-mapping>
        <servlet-name>springmvc</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

3.1.3 操作业务Controller

```
package cn.guardwhy.controller;
/**
 * springMVC入门原理案例：需要返回一个ModelAndView，装数据，封视图
 */
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

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

public class HelloController implements Controller {
    @Override
    public ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) throws Exception {
        //1. 创建ModelAndView模型和视图
        ModelAndView mv = new ModelAndView();

        //2. 封装对象，放在ModelAndView中。
        mv.addObject("msg","HelloSpringMVC!");

        //3.封装要跳转的视图，放在ModelAndView中.
        mv.setViewName("success"); // /WEB-INF/jsp/success.jsp
        return mv;
    }
}
```

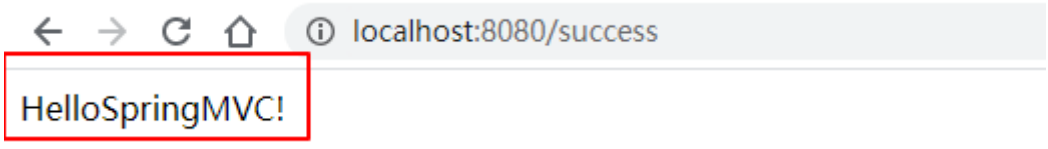
3.1.4 编写SpringMVC的配置文件

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

    <!-- 处理映射器-->
    <bean
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

    <!--处理器适配器-->
    <bean
class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"/>

    <!--视图解析器:DispatcherServlet给他的ModelAndView
        1、获取ModelAndView的数据
        2.解析ModelAndView的视图名字
        3、拼接视图名字,找到对应的视图 /WEB-INF/jsp/success.jsp
        4.将数据渲染到这个视图上。
    -->
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver"
id="InternalResourceViewResolver">
        <!--前缀-->
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <!--后缀-->
        <property name="suffix" value=".jsp"/>
    </bean>

    <!--注册bean,将类提交给springIOC容器来管理-->
    <bean id="/success" class="cn.guardwhy.controller.HelloController"/>
</beans>
```

3.1.5 跳转的jsp页面

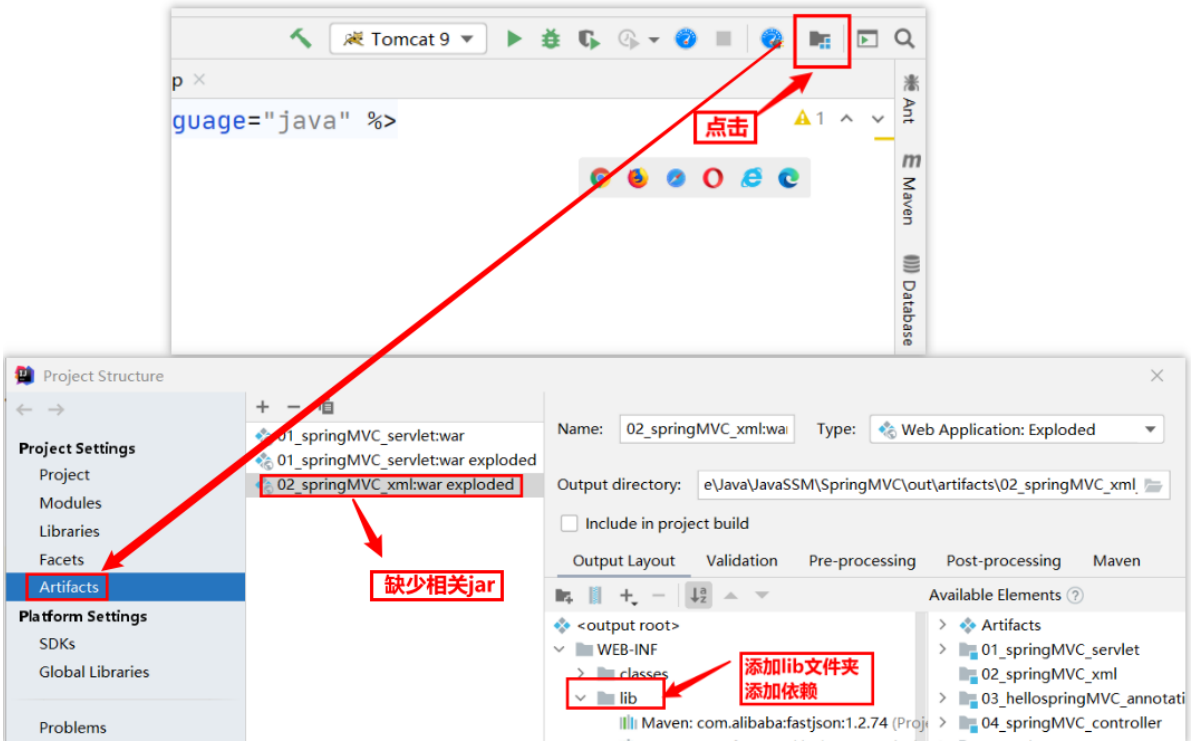
```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>guardwhy</title>
</head>
    <body>
        ${msg}
    </body>
</html>
```

3.1.6 执行结果

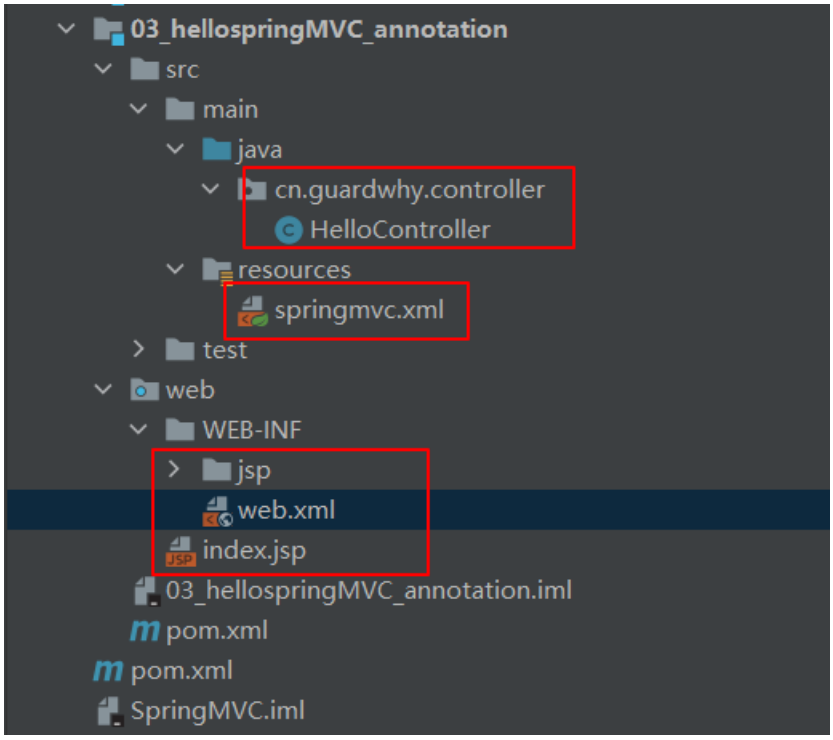


3.1.7 出现错误

访问出现404，项目模块中缺少相关依赖



3.2 项目目录(注解版)



3.2.1 添加父工程相关依赖

```
<build>
    <!--处理资源导出问题-->
    <resources>
        <resource>
            <directory>src/main/java</directory>
```

```
            <includes>
                <include>**/*.properties</include>
                <include>**/*.xml</include>
            </includes>
            <filtering>false</filtering>
        </resource>
        <resource>
            <directory>src/main/resources</directory>
            <includes>
                <include>**/*.properties</include>
                <include>**/*.xml</include>
            </includes>
            <filtering>false</filtering>
        </resource>
    </resources>
</build>
```

3.2.2 配置web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
         version="4.0">

    <!--1.注册DispatcherServlet-->
    <servlet>
        <servlet-name>springmvc</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

        <!--关联一个springmvc的配置文件-->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:springmvc.xml</param-value>
        </init-param>
        <!--启动级别-1-->
        <load-on-startup>1</load-on-startup>
    </servlet>
    <!--所有的请求都会被springmvc拦截-->
    <servlet-mapping>
        <servlet-name>springmvc</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

3.2.3 操作业务Controller

- @Controller是为了让Spring IOC容器初始化时自动扫描到。
- @RequestMapping是为了映射请求路径，这里因为类与方法上都有映射所以访问时应该是/HelloController/success。
- 方法中声明Model类型的参数是为了把Action中的数据带到视图中。
- 方法返回的结果是视图的名称hello，加上配置文件中的前后缀变成WEB-INF/jsp/success.jsp。

```
package cn.guardwhy.controller;
```

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

/**
 * springMVC入门原理案例(注解版本)
 */
@Controller
@RequestMapping("/HelloController")

public class HelloController {
    //真实访问地址 : 项目名/HelloController/success
    @RequestMapping("/success")
    public String sayHello(Model model){
        // 1.向模型中添加属性，可以在页面中取出渲染
        model.addAttribute("msg", "hello, SpringMVC");
        //2. web-inf/jsp/success.jsp
        return "success";
    }
}
```

3.2.4 SpringMVC的配置文件

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


    <!--1.自动扫描包,让指定包下的注解生效,由IOC容器统一管理-->
    <context:component-scan base-package="cn.guardwhy.controller"/>

    <!--2.让Spring MVC处理静态资源-->
    <mvc:default-servlet-handler/>

    <!--3.支持mvc注解驱动
        在spring中一般采用@RequestMapping注解来完成映射关系,要想使@RequestMapping注解生效必须向上下文中注册DefaultAnnotationHandlerMapping
        和一个AnnotationMethodHandlerAdapter实例这两个实例分别在类级别和方法级别处理。
        而annotation-driven配置帮助我们自动完成上述两个实例的注入。
    -->
    <mvc:annotation-driven/>

    <!--4.视图解析器 -->
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver"
id="InternalResourceViewResolver">
        <!--前缀-->
        <property name="prefix" value="/WEB-INF/jsp/"/>
```

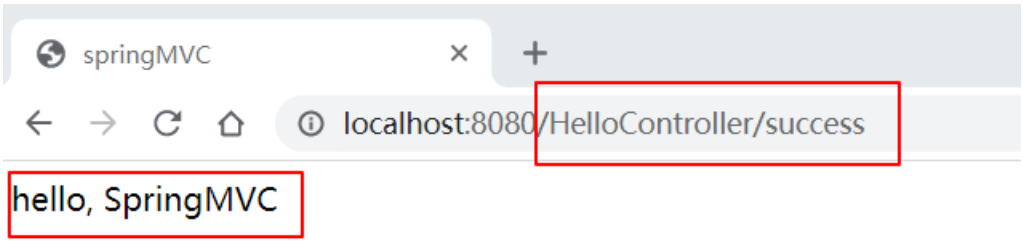
```
        <!--后缀-->
        <property name="suffix" value=".jsp"/>
    </bean>
</beans>
```

3.2.5 创建视图层

在WEB-INF/ jsp目录中创建hello.jsp,视图可以直接取出并展示从Controller带回的信息。可以通过EL表示取出Model中存放的值，或者对象。

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

3.2.6 执行结果

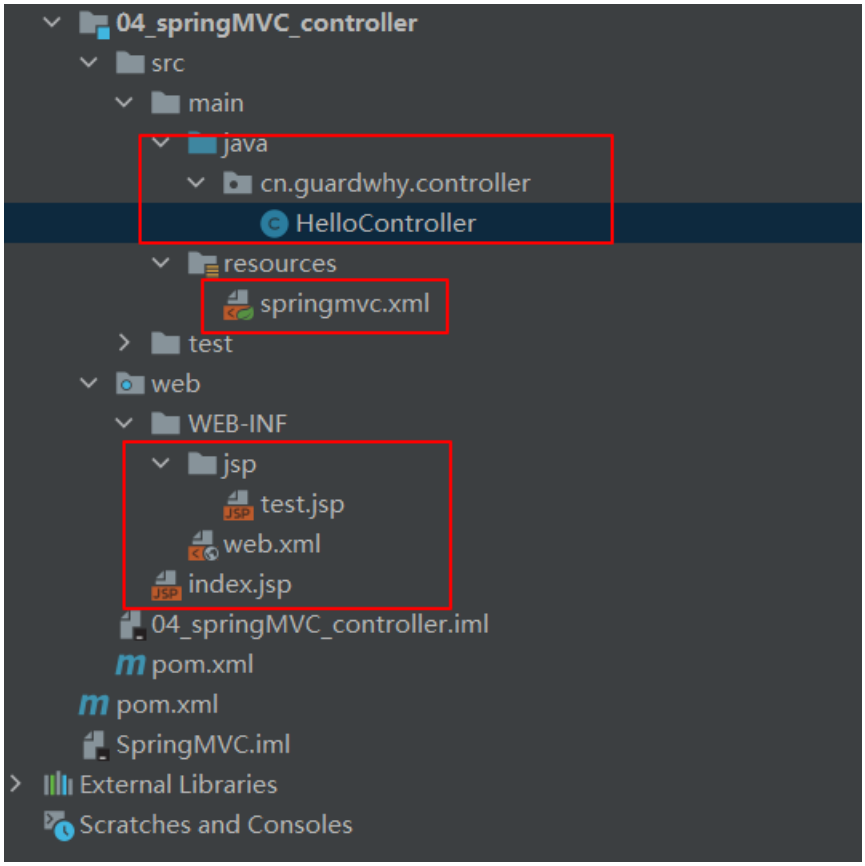


4- 控制器Controller

- 控制器复杂提供访问应用程序的行为，通常通过接口定义或注解定义两种方法实现。
- 控制器负责解析用户的请求并将其转换为一个模型。
- 在Spring MVC中一个控制器类可以包含多个方法，在Spring MVC中，对于Controller的配置方式有很多种

4.1 实现Controller接口

4.1.1项目目录



4.1.2 操作业务Controller

实现Controller接口，从而获得控制器功能

```
package cn.guardwhy.controller;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

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

/**
 * Controller详解
 */
public class TestController1 implements Controller {

    @Override
    public ModelAndView handleRequest(HttpServletRequest httpServletRequest,
                                      HttpServletResponse httpServletResponse)
throws Exception {
        // 1.返回一个模型视图对象
        ModelAndView mv = new ModelAndView();
        // 2.封装对象，放在ModelAndView中。
        mv.addObject("msg", "TestController1");
        // 3. 封装跳转的视图
        mv.setViewName("test");
        return mv;
    }
}
```

4.1.3 SpringMVC的配置文件

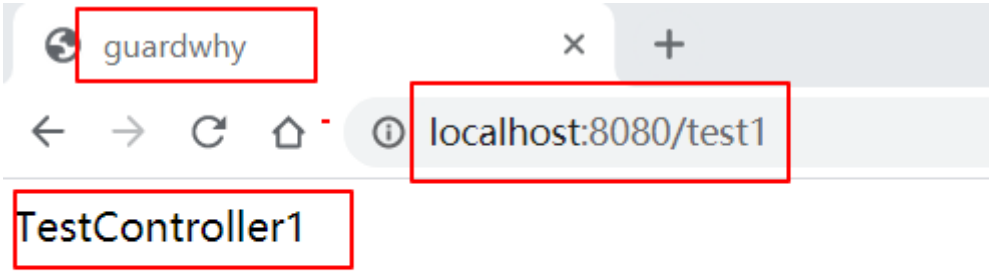
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!--注册bean,将类提交给springIOC容器来管理-->
    <bean id="/test1" class="cn.guardwhy.controller.TestController1"/>

    <!--4.视图解析器 -->
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver"
id="InternalResourceViewResolver">
        <!--前缀-->
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <!--后缀-->
        <property name="suffix" value=".jsp"/>
    </bean>
</beans>
```

4.1.4 创建视图层

在WEB-INF/ jsp目录中创建hello.jsp,视图可以直接取出并展示从Controller带回的信息。可以通过EL表示取出Model中存放的值，或者对象。

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

4.1.5 执行结果



> **注意点**

- 实现接口Controller定义控制器是较老的办法。
- 缺点是：一个控制器中只有一个方法，如果要多个方法则需要定义多个Controller。定义的方式比较麻烦。

4.2 使用注解@Controller

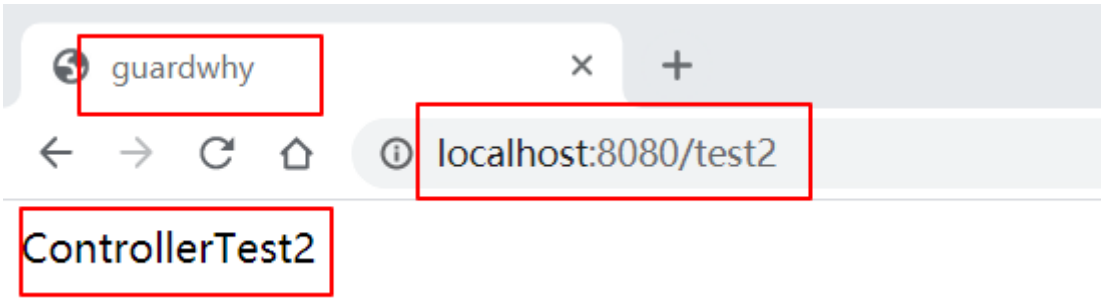
4.2.1 操作业务Controller

```
package cn.guardwhy.controller;

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

/**
 * 注解形式
 */
@Controller // 代表这个类会被Spring接管，被这个注解的类中的所有方法，如果返回值是Spring
            // 并且有具体页面可以跳转，那么就会被视图解析器解析。
public class TestController2 {
    // 映射访问路径
    @RequestMapping("/test2")
    public String Test(Model model){
        // 自动实例化一个Model对象用于向视图中传值
        model.addAttribute("msg", "ControllerTest2");
        return "test"; // 会被拼接成//WEB-INF//jsp//test.jsp
    }
}
```

4.2.2 SpringMVC的配置文件

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

    <!-- 自动扫描指定的包，下面所有注解类交给IOC容器管理 -->
    <context:component-scan base-package="cn.guardwhy.controller"/>

    <!--4.视图解析器 -->
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver"
id="InternalResourceViewResolver">
        <!--前缀-->
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <!--后缀-->
        <property name="suffix" value=".jsp"/>
    </bean>
</beans>
```

4.2.3 执行结果

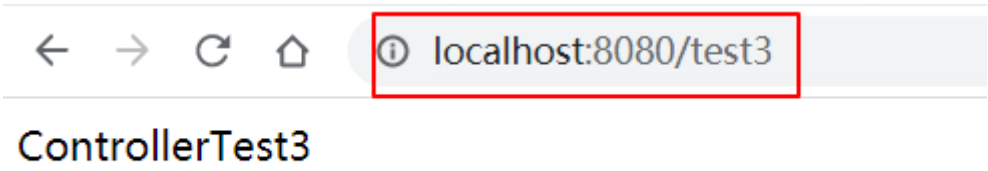


4.3 RequestMapping

@RequestMapping注解用于映射url到控制器类或一个特定的处理程序方法。可用于类或方法上，用于类上，表示类中的所有响应请求的方法都是以该地址作为父路径。

4.3.1 注解在方法上

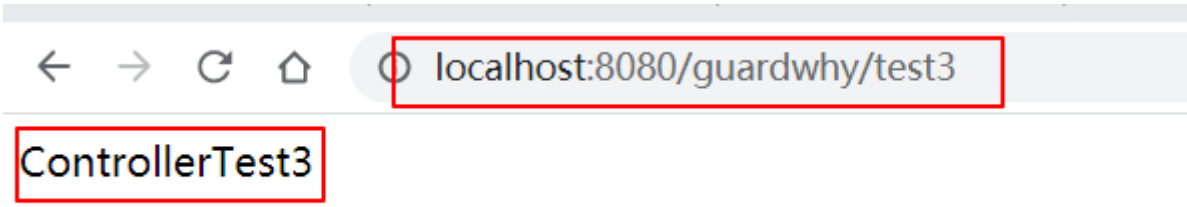
```
package cn.guardwhy.controller;

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

@Controller
public class TestController3 {
    @RequestMapping("/test3")

    public String Test3(Model model){
        // 自动实例化一个Model对象用于向视图中传值
        model.addAttribute("msg", "ControllerTest3");
        return "test";
    }
}
```



4.3.2 同时注解类与方法

```
package cn.guardwhy.controller;

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

@Controller
@RequestMapping("/guardwhy")
public class TestController3 {
    @RequestMapping("/test3")

    public String Test3(Model model){
```

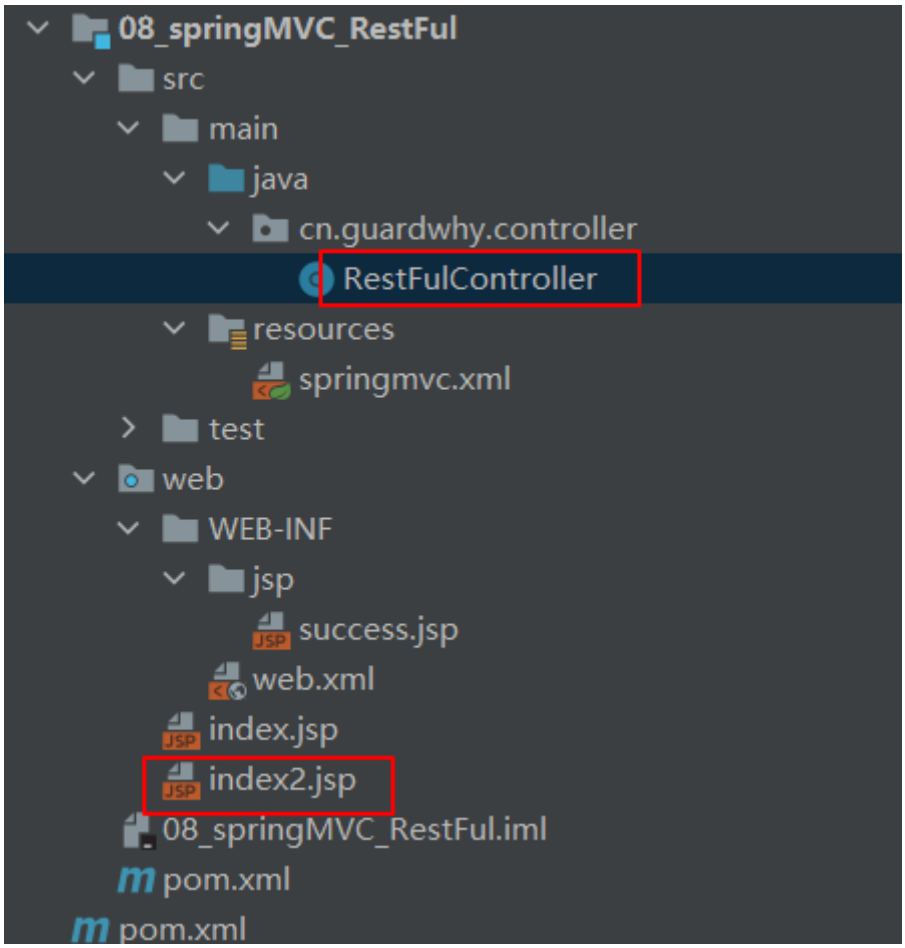
```
        // 自动实例化一个Model对象用于向视图中传值
        model.addAttribute("msg", "ControllerTest3");
        return "test";
    }
}
```



4.4 RestFul风格

Restful就是一个资源定位及资源操作的风格。不是标准也不是协议，只是一种风格。基于这个风格设计的软件可以更简洁，更有层次，更易于实现缓存等机制。

4.4.1 项目目录



4.4.2 基本作用

资源：互联网所有的事物都可以被抽象为资源。

资源操作：使用POST、DELETE、PUT、GET，使用不同方法对资源进行操作。分别对应 添加、 删除、修改、查询。

> **传统方式操作资源**

通过不同的参数来实现不同的效果！！！

- http://localhost:8080/query.stu?id=1 查询，GET
- http://localhost:8080/save.stu 新增，POST
- http://localhost:8080/update.stu 更新，POST
- http://localhost:8080/delete.stu?id=1 删除 POST

> **使用RESTful操作资源**

可以通过不同的请求方式来实现不同的效果！！注意，请求地址一样，但是功能可以不同！！！

- http://localhost:8080/stu/1 查询，GET
- http://localhost:8080/stu 新增，POST
- http://localhost:8080/stu 更新，PUT
- http://localhost:8080/stu/1 删除 DELETE

4.4.3 控制器(Controller)

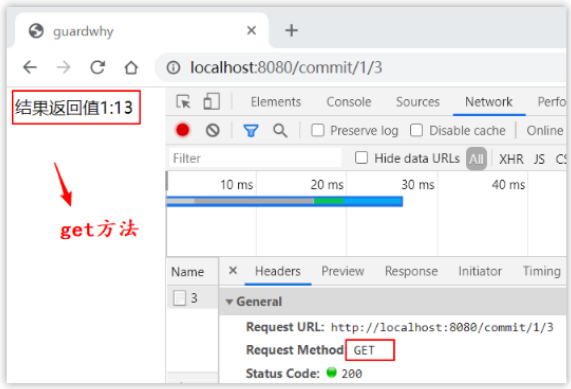
在Spring MVC中可以使用@PathVariable 注解，让方法参数的值对应绑定到一个URL模板变量。

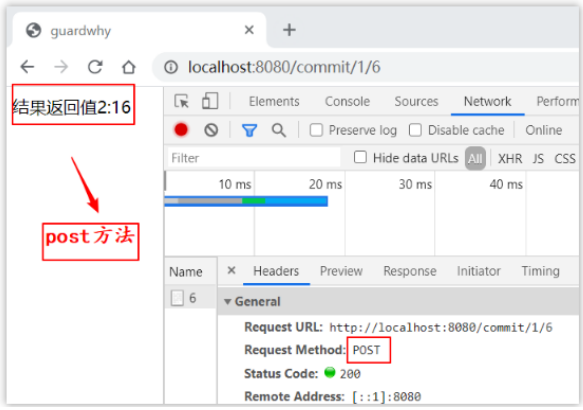
```
package cn.guardwhy.controller;

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

/**
 * RestFul风格入门案例
 */
@Controller
public class RestFulController {
    // 1.访问映射路径
    @RequestMapping(value = "/commit/{a}/{b}", method = RequestMethod.GET)
    public String test1(@PathVariable int a, @PathVariable String b, Model
model){
        // 返回结果集
        String res = a + b;
        model.addAttribute("msg", "结果返回值1:" + res);
        return "success";
    }

    // post方法
    @PostMapping("/commit/{a}/{b}")
    public String test2(@PathVariable int a, @PathVariable String b, Model
model){
        // 返回结果集
        String res = a + b;
        model.addAttribute("msg", "结果返回值2:" + res);
        return "success";
    }
}
```

视图层

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>post提交</title>
</head>
<body>
    <form action="commit/1/6" method="post">
        <input type="submit" value="提交">
    </form>
</body>
</html>
```

4.4.4 执行结果





4.4.5 总结

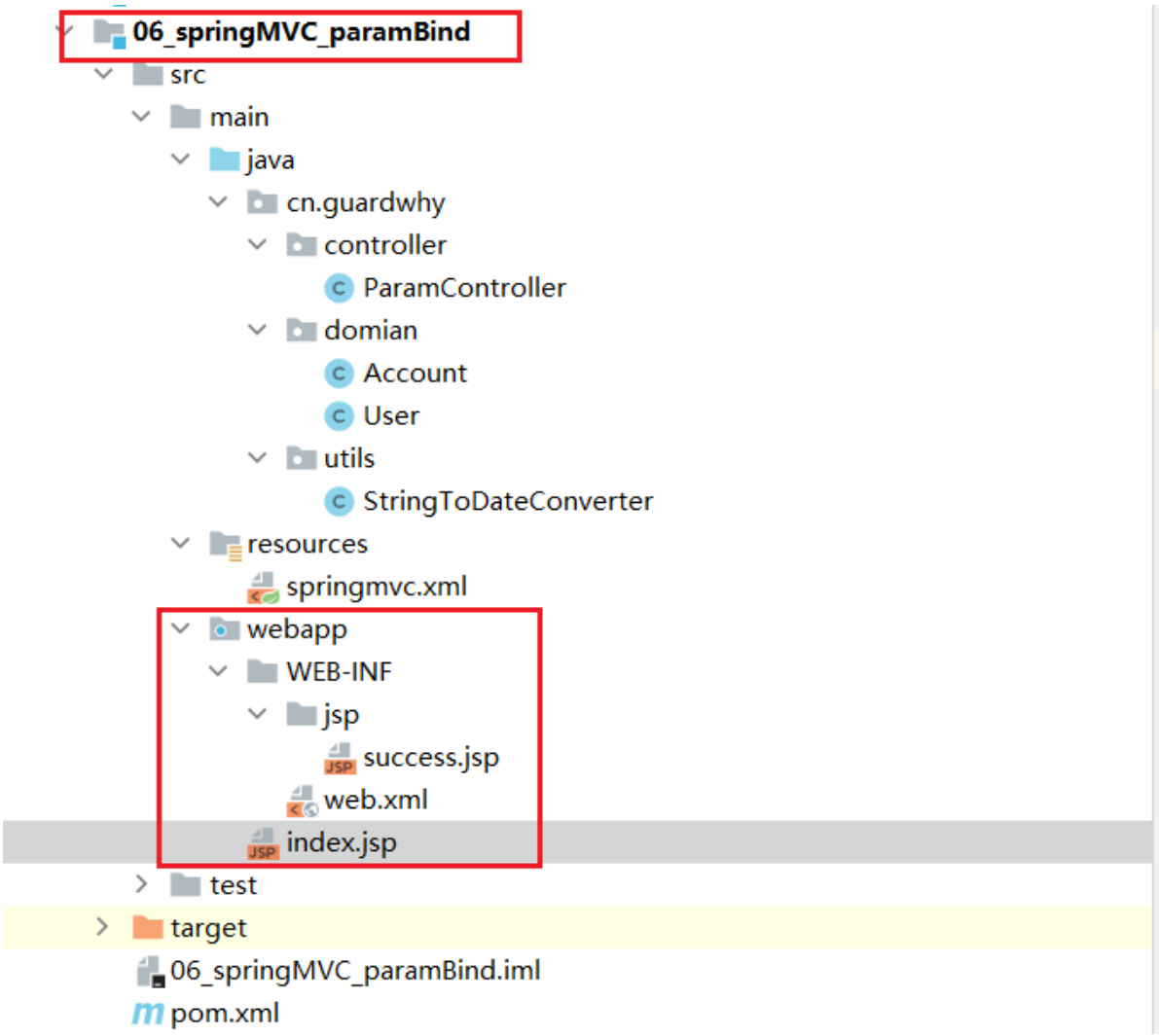
Spring MVC 的 **@RequestMapping** 注解能够处理 HTTP 请求的方法, 比如 GET, PUT, POST, DELETE。

所有的地址栏请求默认都会是 HTTP GET 类型的。
组合注解:

```
@GetMapping
@PostMapping
@PutMapping
@DeleteMapping
@PatchMapping
```

5-Spring MVC 请求参数

5.1 项目目录



参数类型	使用要求
基本类型参数（基本类型和 String 类型）	参数名称必须和控制器中方法的形参名称保持一致。(严格区分大小写)
POJO 类型参数 (实体类，以及关联的实体类)	要求表单中参数名称和 POJO 类的属性名称保持一致。并且控制器方法的参数类型是 POJO 类型。
数组和集合类型参数 (List 结构和 Map结构的集合(包括数组）)	要求集合类型的请求参数必须在 POJO 中。在表单中请求参数名称要和 POJO 中集合属性名称相同。 给 List 集合中的元素赋值，使用下标。给 Map 集合中的元素赋值，使用键值对。 接收的请求参数是 json 格式数据。需要借助一个注解实现。

5.2 基本类型参数

- Controller中的业务方法的参数名称要与请求参数的name一致，参数值会自动映射匹配。
- 并且能自动做类型转换；自动的类型转换是指从String向其他类型的转换。

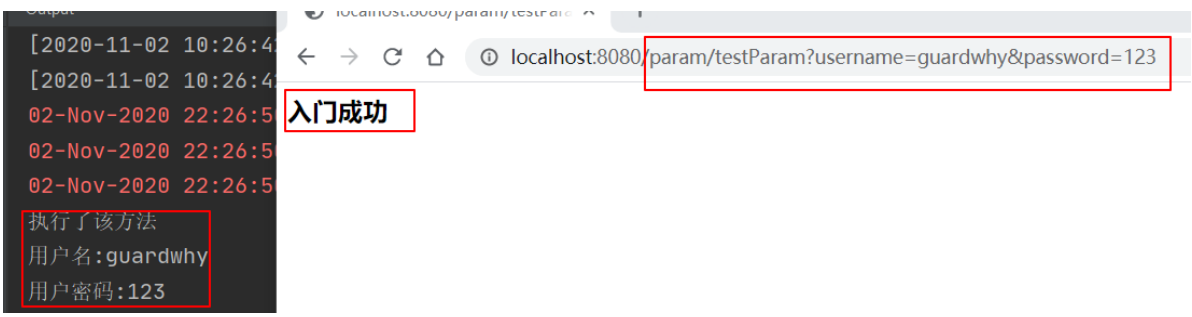
5.2.1 控制层(Controller)

```
package cn.guardwhy.controller;

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

/**
 * 请求参数绑定
*/
@Controller
@RequestMapping("/param")
public class ParamController {
    /**
     * 请求参数绑定入门案例
     * @param username
     * @param password
     * @return
     */
    @RequestMapping("/testParam")
    public String testParam(String username, String password){
        System.out.println("执行了该方法");
        System.out.println("用户名:" + username);
        System.out.println("用户密码:" + password);
        return "success";
    }
}
```

5.2.2创建视图层

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>springMVC参数绑定</title>
</head>
<body>
    <%--请求参数绑定--%>
    <a href="${pageContext.request.contextPath}/param/testParam?
username=guardwhy&password=123">基本类型</a>
</body>
</html>
```

5.2.3 执行结果



5.3 POJO类型实现

Controller中的业务方法参数的POJO属性名与请求参数的name一致，参数值会自动映射匹配。

5.3.1 持久层

Account类

```
package cn.guardwhy.domian;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.io.Serializable;
import java.util.List;
import java.util.Map;

/**
 * 账户类
 */
@NoArgsConstructor
@Data
@AllArgsConstructor
public class Account implements Serializable {
    private String username; //姓名
    private String password; // 密码
    private Double money;   // 金额

    private List<User> list; // 用户list集合
    private Map<String, User> map; // 用户Map集合
}
```

User类

```
package cn.guardwhy.domian;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.io.Serializable;
import java.util.Date;

/**
 * 用户类
 */
@AllArgsConstructor
@Data
@NoArgsConstructor
public class User implements Serializable {
    private String uname;  // 用户name
    private Integer age; // 用户gae
    private Date date; // 用户生日
}
```

5.3.2 Controller

```
package cn.guardwhy.controller;

import cn.guardwhy.domian.Account;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

/**
 * 请求参数绑定
*/
@Controller
@RequestMapping("/param")
public class ParamController {

    /***
     * 将参数绑定的数据封装到javaBean的类中
     * @param account
     * @return
     */
    @RequestMapping("/saveAccount")
    public String saveAccount(Account account){
        System.out.println("执行了该方法");
        System.out.println(account);
        return "success";
    }
}
```

5.3.3 视图层

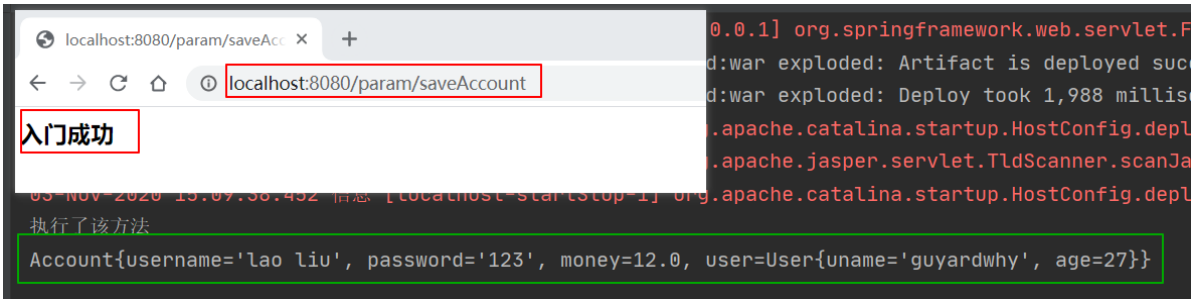
```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>springMVC参数绑定</title>
</head>
<body>
    <%--将数据封装到类中--%>
    <form action="param/saveAccount" method="post">
        姓名: <input type="text" name="username"/><br/>
        密码: <input type="text" name="password"/><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>
</body>
</html>
```

5.3.4 配置web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
         version="4.0">
```

```
    <!--1.注册DispatcherServlet-->
    <servlet>
        <servlet-name>springmvc</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

        <!--关联一个springmvc的配置文件-->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:springmvc.xml</param-value>
        </init-param>
        <!--启动级别-1-->
        <load-on-startup>1</load-on-startup>
    </servlet>
    <!--所有的请求都会被springmvc拦截-->
    <servlet-mapping>
        <servlet-name>springmvc</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

    <!--配置解决中文乱码的过滤器-->
    <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>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

5.3.5 执行结果



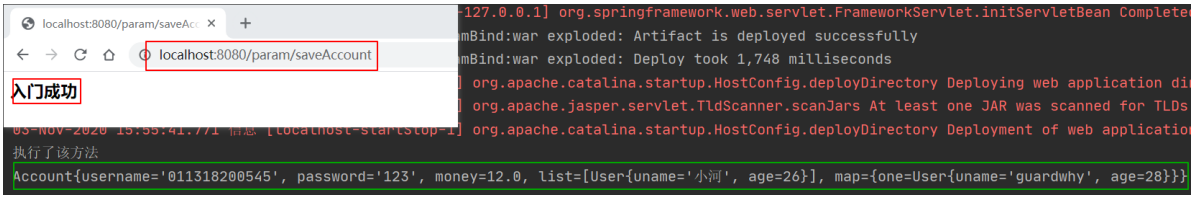
5.4 集合类型实现

获得集合参数时，要将集合参数包装到一个POJO中才可以。

5.4.1 视图层

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>springMVC参数绑定</title>
</head>
<body>
    <%--将数据封装Account类中,类中存在list和map集合--%>
    <form action="param/saveAccount" method="post">
        姓名: <input type="text" name="username"/><br/>
        密码: <input type="text" name="password"/><br/>
        金额: <input type="text" name="money"/><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>
</body>
</html>
```

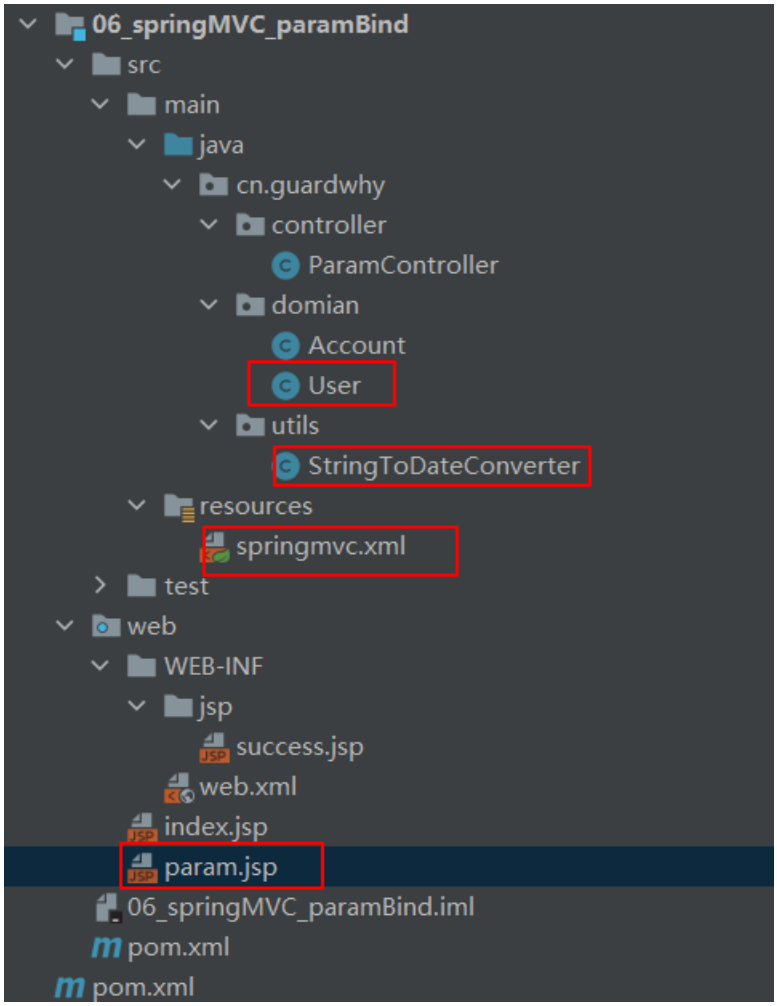
5.4.2 执行结果



5.5 自定义类型转换器实现

SpringMVC 默认已经提供了一些常用的类型转换器；例如：客户端提交的字符串转换成int型进行参数设置，日期格式类型要求为：yyyy/MM/dd 不然的话会报错，对于特有的行为，SpringMVC提供了自定义类型转换器方便开发者自定义处理。

5.5.1 项目目录



5.5.2 持久层

StringToDateConverter

```
package cn.guardwhy.utils;

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

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;
/**
 * 字符串转成日期
 */
public class StringToDateConverter implements Converter<String, Date> {
    /***
     * 传进来字符串
     * @param source
     * @return
     */
    @Override
    public Date convert(String source) {
        // 1.条件判断
        if(source == null){
            throw new RuntimeException("请传入数据");
        }
        DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
        try {
            // 2.将字符串转换日期
```

```
            return df.parse(source);
        } catch (Exception e) {
            throw new RuntimeException("数据类型转换出现错误");
        }
    }
}
```

5.5.3 配置springmvc.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">

    <!--1.自动扫描包,让指定包下的注解生效,由IOC容器统一管理-->
    <context:component-scan base-package="cn.guardwhy"/>

    <!--2.让Spring MVC处理静态资源-->
    <mvc:default-servlet-handler/>

    <!--3.配置自定义类型转换器-->
    <bean id="conversionService"
class="org.springframework.context.support.ConversionServiceFactoryBean">
        <property name="converters">
            <set>
                <bean class="cn.guardwhy.utils.StringToDateConverter"/>
            </set>
        </property>
    </bean>

    <!--4.支持mvc注解驱动-->
    <mvc:annotation-driven conversion-service="conversionService"/>

    <!--4.视图解析器 -->
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver"
id="InternalResourceViewResolver">
        <!--前缀-->
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <!--后缀-->
        <property name="suffix" value=".jsp"/>
    </bean>
</beans>
```

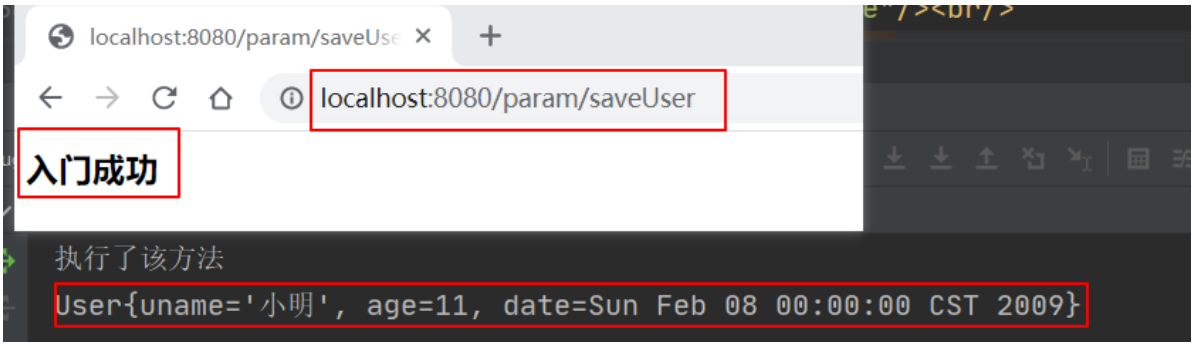
5.5.4 控制层(Controller)

```
package cn.guardwhy.controller;

import cn.guardwhy.domian.Account;
import cn.guardwhy.domian.User;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

/**
 * 请求参数绑定
*/
@Controller
@RequestMapping("/param")
public class ParamController {
    /***
     * 自定义转换器
     * @param user
     * @return
     */
    @RequestMapping("/saveUser")
    public String saveUser(User user){
        System.out.println("执行了该方法");
        System.out.println(user);
        return "success";
    }
}
```

5.5.5 视图层

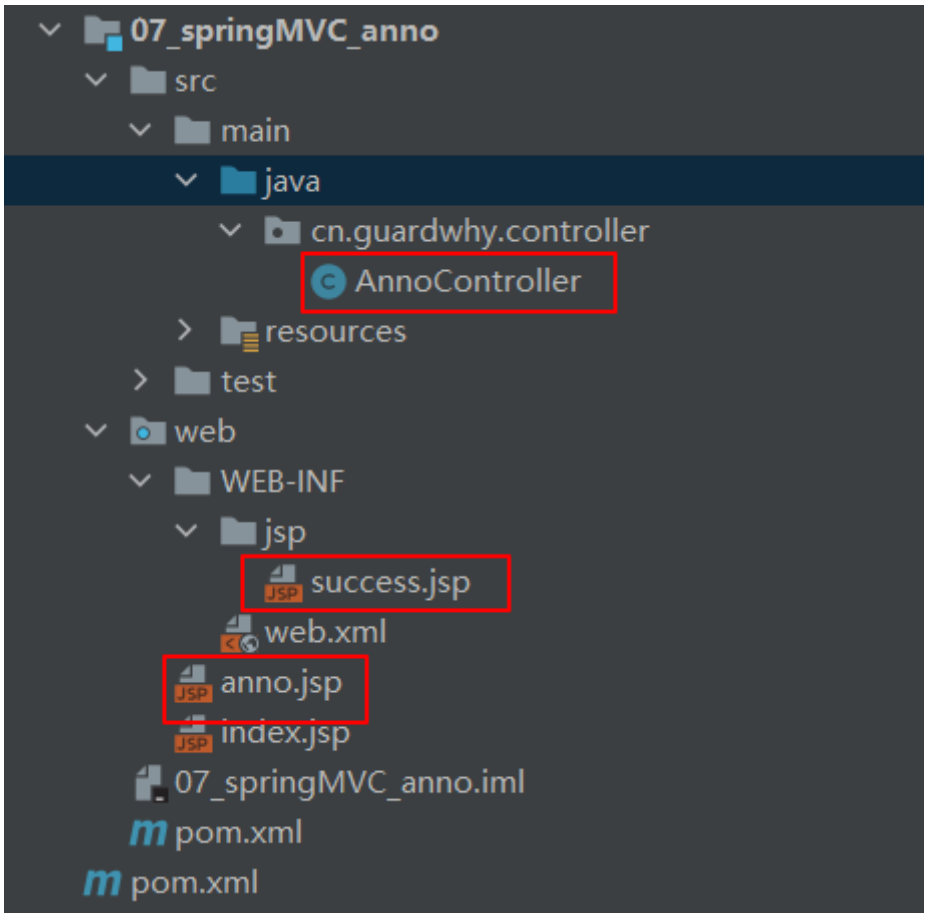
```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>springMVC参数绑定</title>
</head>
<body>
    <%--自定义类型转换器--%>
    <form action="param/saveUser" method="post">
        用户姓名: <input type="text" name="uname"/><br/>
        用户年龄: <input type="text" name="age"/><br/>
        用户生日: <input type="text" name="date"/><br/>
        <input type="submit" value="提交">
    </form>
</body>
</html>
```

5.5.6 执行结果



5.6 常用注解

5.6.1 项目目录



5.6.2 控制层(Controller)

```
package cn.guardwhy.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

/**
 * 常用注解
 */
@Controller
@RequestMapping("/anno")
public class AnnoController {
```

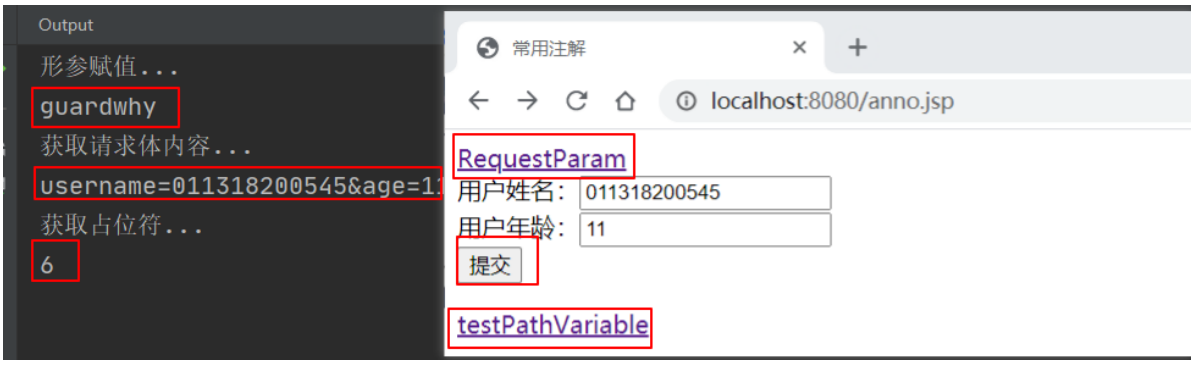
```
    /***
     * 把请求中指定名称的参数给控制器中的形参赋值。
     * @param username
     * @return
     */
    @RequestMapping("/testRequestParam")
    public String testRequestParam(@RequestParam(name = "name") String username)
{
        System.out.println("执行了该方法...");
        System.out.println(username);
        return "success";
    }

    /***
     * 获取请求体的内容
     * @param body
     * @return
     */
    @RequestMapping("/testRequestBody")
    public String testRequestBody(@RequestBody String body){
        System.out.println("获取请求体内容...");
        System.out.println(body);
        return "success";
    }

    /***
     * 用于绑定 url中的占位符
     * @param id
     * @return
     */
    @RequestMapping("/testPathVariable/{sid}")
    public String testPathVariable(@PathVariable(name = "sid") String id){
        System.out.println("获取请求体内容...");
        System.out.println(id);
        return "success";
    }
}
```

5.6.3 视图层

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>常用注解</title>
</head>
<body>
    <%--常用注解--%>
    <a href="anno/testRequestParam?name=guardwhy">RequestParam</a><br/>

    <%--请求体--%>
    <form action="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>

    <%--用于绑定url中的占位符--%>
```

```
    <a href="anno/testPathVariable/6">testPathVariable</a><br/>
</body>
</html>
```

5.6.4 执行结果



6- Spring MVC的响应

6.1 响应方式介绍

> **页面跳转**

- 返回字符串逻辑视图。
- void原始ServletAPI。
- ModelAndView。

> **返回数据**

- 直接返回字符串数据。
- 将对象或集合转为json返回。

6.2 返回字符串视图

直接返回字符串：此种方式会将返回的字符串与视图解析器的前后缀拼接后跳转到指定页面

```
@RequestMapping("/m1")
public String test() {
  return "success";
}
```

6.3 原始ServletAPI

通过request、response对象实现响应。通过设置ServletAPI，不需要视图解析器。

```
@RequestMapping("/m1")
public void test1(HttpServletRequest request, HttpServletResponse response)
throws Exception {
  // 1.通过response直接响应数据
  response.setContentType("text/html;charset=utf-8");
  response.getWriter().write("慕课网");
  ----------------------------------------
  request.setAttribute("username", "imooc");
  // 2.通过request实现转发
```

```
    request.getRequestDispatcher("/WEB-
INF/pages/success.jsp").forward(request,response);

    // 3.通过response实现重定向
    response.sendRedirect(request.getContextPath() + "/index.jsp");
}
```

6.4 转发和重定向

> 请求转发forward

使用返回字符串逻辑视图实现页面的跳转，这种方式其实就是请求转发。如果用了forward：则路径必须写成实际视图url，不能写逻辑视图。

```
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。

```
@RequestMapping("/m1")
public String test1(Model model) {
    model.addAttribute("username", "guardwhy");
    return "forward:/WEB-INF/pages/success.jsp";
}
```

> Redirect重定向

SpringMVC框架会自动拼接，并且将Model中的数据拼接到url地址上。

```
@RequestMapping("/redirect")
public String test2(Model model) {
    model.addAttribute("username", "guardwhy");
    return "redirect:/index1.jsp";
}
```

6.5 ModelAndView

> 方式一

在Controller中方法创建并返回model对象，并且设置视图名称。

```
@RequestMapping("/m1")
public String test1() {
    /*
      Model:模型 作用封装数据
      View: 视图 作用展示数据
    */
    ModelAndView model = new ModelAndView();
    //设置模型数据
    model.addObject("username", "guardwhy");
    //设置视图名称
    model.setViewName("success");
    return model;
}
```

> 方式二

在Controller中方法形参上直接声明model对象，无需在方法中自己创建，在方法中直接使用该对象设置视图，同样可以跳转页面。

```
@RequestMapping("/m2")
public String test2(Model model) {
  /*
    Model:模型 作用封装数据
    View: 视图 作用展示数据
  */
  //设置模型数据
  model.addObject("username", "guardwhy");
  //设置视图名称
  model.setViewName("success");
  return model;
}
```

6.6 跳转结果方式

> 实现Controller接口

设置ModelAndView对象，根据view的名称，和视图解析器跳到指定的页面。
页面：{视图解析器前缀} + viewName +{视图解析器后缀}

```
<?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">

    <!--1.自动扫描包,让指定包下的注解生效,由IOC容器统一管理-->
    <context:component-scan base-package="cn.guardwhy.controller"/>

    <!--2.让Spring MVC处理静态资源-->
    <mvc:default-servlet-handler/>

    <!--4.支持mvc注解驱动-->
    <mvc:annotation-driven/>

    <!--4.视图解析器 -->
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver"
id="InternalResourceViewResolver">
        <!--前缀-->
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <!--后缀-->
        <property name="suffix" value=".jsp"/>
    </bean>

</beans>
```

控制器controller

```
package cn.guardwhy.controller;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

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

/**
 * Controller详解
 */
public class TestController1 implements Controller {

    @Override
    public ModelAndView handleRequest(HttpServletRequest httpServletRequest,
                                      HttpServletResponse httpServletResponse)
throws Exception {
        // 1.返回一个模型视图对象
        ModelAndView mv = new ModelAndView();
        // 2.封装对象，放在ModelAndView中。
        mv.addObject("msg", "TestController1");
        // 3. 封装跳转的视图
        mv.setViewName("test");
        return mv;
    }
}
```

SpringMVC

通过SpringMVC来实现转发和重定向 - 有视图解析器，重定向，不需要视图解析器 。

本质就是重新请求一个新地方嘛，所以注意路径问题，可以重定向到另外一个请求实现 。

```
package cn.guardwhy.controller;

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

/**
 * 转发和重定向
 */
@Controller
public class ResultController02 {
    @RequestMapping(value = "m1/t1")
    public String test1(Model model){
        // 转发方式
        model.addAttribute("{msg}", "ModelTest1");
        return "success";
    }

    @RequestMapping(value = "m1/t2")
    public String test2(Model model){
        // 转发方式
        model.addAttribute("{msg}", "ModelTest1");
```

```
        return "redirect:/index2.jsp";
    }
}
```

7 - JSON

7.1 JSON基础

> JSON基本概念

- JSON(JavaScript Object Notation, JS 对象标记) 是一种轻量级的数据交换格式。
- 采用完全独立于编程语言的文本格式来存储和表示数据。简洁和清晰的层次结构使得 JSON 成为理想的数据交换语言。

注意点

在 JavaScript 语言中，一切都是对象。因此，任何JavaScript 支持的类型都可以通过 JSON 来表示。

```
- 对象表示为键值对，数据由逗号分隔。
- 花括号保存对象
- 方括号保存数组
```

JSON 键值对是用来保存 JavaScript 对象的一种方式和 JavaScript 对象的写法也大同小异。

键/值对组合中的键名写在前面并用双引号 "" 包裹，使用冒号 : 分隔，然后紧接着值：

```
<script type="text/javascript">
    // 1.编写一个JS对象
    var user = {
        name: "guardwhy",
        age: 26,
        sex: "男"
    };

    // 2.将js对象转换成JSON对象
    var json = JSON.stringify(user);
    console.log(json);

    // 输出结果
    console.log("======");

    // 3.将JSON对象转换成JavaScript对象
    var obj = JSON.parse(json);
    console.log(obj);
</script>
```

> JSON 和 JS 区别

JSON 是 JavaScript 对象的字符串表示法，它使用文本表示一个 JS 对象的信息，本质是一个字符串。

```
var obj = {a: 'guard', b: 'why'};   //这是一个对象，注意键名也是可以使用引号包裹的
var json = '{"a": "guard", "b": "why"}';    //这是一个 JSON 字符串，本质是一个字符串
```

> JSON 和 JS互转

要实现从JSON字符串转换为JavaScript 对象，使用 JSON.parse() 方法：

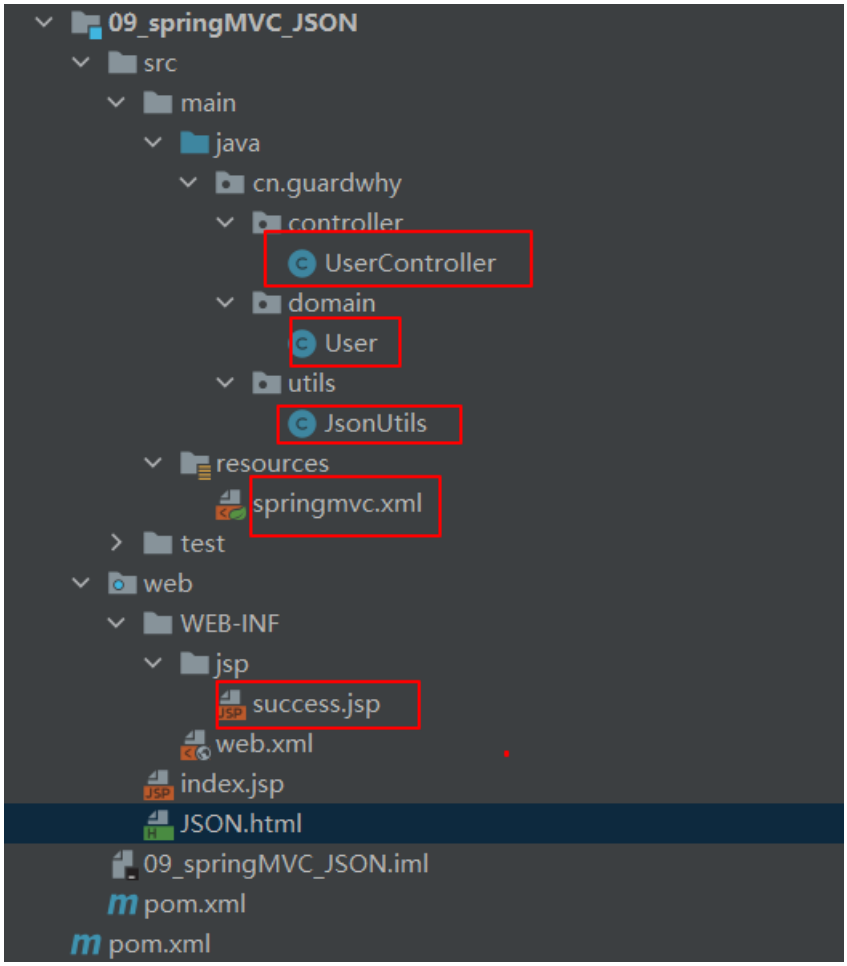
```
var obj = JSON.parse('{"a": "guard", "b": "why"}');  //结果是 {a: 'guard', b: 'why'}
```

要实现从JavaScript 对象转换为JSON字符串，使用 JSON.stringify() 方法：

```
var json = JSON.stringify({a: 'Hello', b: 'World'}); //结果是 '{"a": "Hello", "b": "World"}'
```

7.2 项目实现

7.2.1 项目目录



> **相关依赖**

```
<!--导入JSON包-->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.11.3</version>
</dependency>

<!--导入fastjson-->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.2.74</version>
</dependency>
```

```
<!--导入lombok-->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.16</version>
</dependency>
```

> 配置web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
         version="4.0">

    <!--1.注册DispatcherServlet-->
    <servlet>
        <servlet-name>springmvc</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

        <!--关联一个springmvc的配置文件-->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:springmvc.xml</param-value>
        </init-param>
        <!--启动级别-1-->
        <load-on-startup>1</load-on-startup>
    </servlet>
    <!--所有的请求都会被springmvc拦截-->
    <servlet-mapping>
        <servlet-name>springmvc</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
    <!--配置解决中文乱码的过滤器-->
    <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>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

> 配置springmvc.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">

    <!--1.自动扫描包,让指定包下的注解生效,由IOC容器统一管理-->
    <context:component-scan base-package="cn.guardwhy"/>

    <!--2.让Spring MVC处理静态资源-->
    <mvc:default-servlet-handler/>

    <!--3.支持mvc注解驱动-->
    <mvc:annotation-driven>
        <!--JSON格式乱码处理方式-->
        <mvc:message-converters register-defaults="true">
            <bean
class="org.springframework.http.converter.StringHttpMessageConverter">
                <constructor-arg value="UTF-8"/>
            </bean>
            <bean
class="org.springframework.http.converter.json.MappingJackson2HttpMessageConvert
er">
                <property name="objectMapper">
                    <bean
class="org.springframework.http.converter.json.Jackson2ObjectMapperFactoryBean">
                        <property name="failOnEmptyBeans" value="false"/>
                    </bean>
                </property>
            </bean>
        </mvc:message-converters>
    </mvc:annotation-driven>
    <!--4.视图解析器 -->
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver"
id="InternalResourceViewResolver">
        <!--前缀-->
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <!--后缀-->
        <property name="suffix" value=".jsp"/>
    </bean>
</beans>
```

7.2.2 持久层

User类

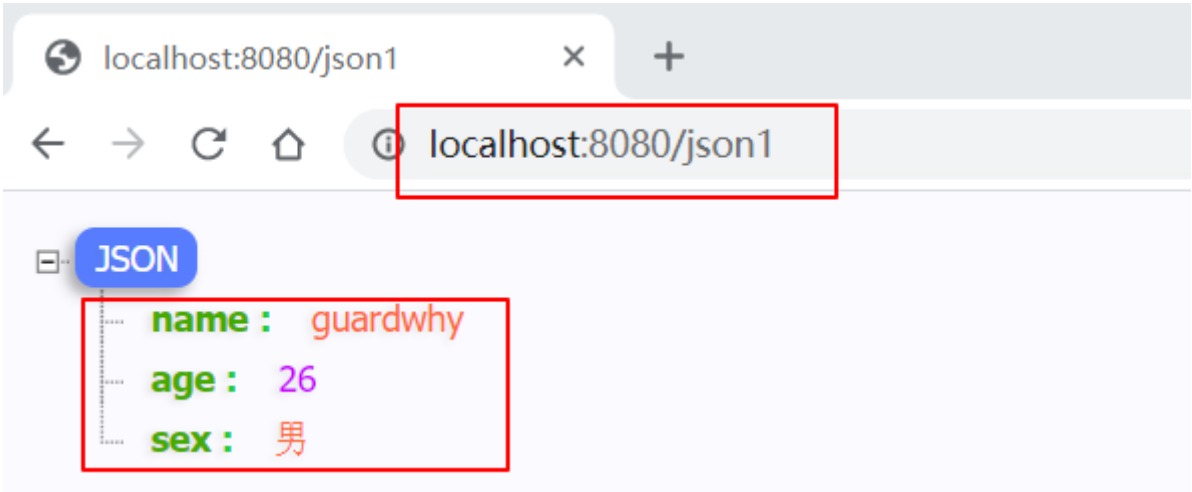
```
package cn.guardwhy.domain;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
```

```java
//需要导入lombok
@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {
    private String name; // 用户name
    private int age; // 用户年龄
    private String sex; // 性别
}
```

7.2.3 控制层(Controller)

```java
// @Controller
@RestController // 不走视图解析器
public class UserController {
    /***
     * 返回JSON字符串
     * @return
     */
    @RequestMapping("/json1")
    @ResponseBody   // 该注解不会走视图解析器,会直接返回一个字符串
    public String json1() throws JsonProcessingException {
        ObjectMapper mapper = new ObjectMapper();
        // 1.创建一个user对象
        User user = new User("guardwhy", 26, "男");
        //2. 将Java对象转换为json字符串
        String str = mapper.writeValueAsString(user);
        // 3.返回JSON字符串
        return str;
  }
}
```

执行结果



7.2.4 工具类

```java
package cn.guardwhy.utils;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.SerializationFeature;
```

```java
import java.text.SimpleDateFormat;

/**
 * JSON工具类
 */
public class JsonUtils {

    public static String getJson(Object object){
        return getJson(object, "yyyy-MM-dd HH:mm:ss");
    }

    public static String getJson(Object object, String dateFormat){
        // 1.创建mapper对象
        ObjectMapper mapper = new ObjectMapper();
        // 2.不使用时间差方式
        mapper.configure(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS, false);
        // 3.定义自定义日期格式对象
        SimpleDateFormat sdf = new SimpleDateFormat(dateFormat);
        // 4.指定日期格式
        mapper.setDateFormat(sdf);

        try {
            // 5.返回该值
            return mapper.writeValueAsString(object);
        } catch (JsonProcessingException e) {
            e.printStackTrace();
        }

        return null;
    }
}
```

7.2.5 集合输出(JOSN)

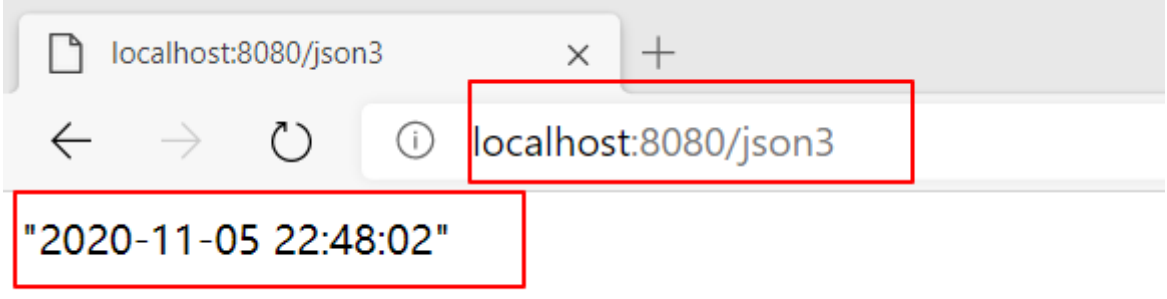
```java
/***
 * 集合输出
 * @return
 * @throws JsonProcessingException
*/
@RequestMapping("/json2")
@ResponseBody
public String json2() throws JsonProcessingException {
    // 1.创建list集合
    List<User> list1 = new ArrayList<>();
    // 2.给User对象添加属性
    User user1 = new User("kobe", 41, "男");
    User user2 = new User("James", 35, "男");
    User user3 = new User("Curry", 10, "男");
    // 3.将对象添加到集合中
    list1.add(user1);
    list1.add(user2);
    list1.add(user3);

    // 4.返回集合
    return new ObjectMapper().writeValueAsString(list1);
}
```

执行结果:

7.2.6 输出时间对象

```
/***
* 输出时间对象
* @return
* @throws JsonProcessingException
*/
@RequestMapping("/json3")
@ResponseBody
public String json3() throws JsonProcessingException {
    // 1.创建时间对象
    Date date = new Date();
    // 2.得到json时间对象
    String json = JsonUtils.getJson(date);
    // 3.返回JSON时间对象
    return json;
}
```

执行结果:



7.3 FastJson

fastjson.jar是阿里开发的一款专门用于Java开发的包，可以方便的实现json对象与JavaBean对象的转换，实现JavaBean对象与json字符串的转换，实现json对象与json字符串的转换。

7.3.1 fastjson主要的类

> JSONObject 代表 json 对象

- JSONObject实现了Map接口, 猜想 JSONObject底层操作是由Map实现的。
- JSONObject对应json对象，通过各种形式的get()方法可以获取json对象中的数据。
- 也可利用诸如size()，isEmpty()等方法获取"键：值"对的个数和判断是否为空。其本质是通过实现Map接口并调用接口中的方法完成的。

> JSONArray 代表 json 对象数组

内部是有List接口中的方法来完成操作的。

> JSON 代表 JSONObject和JSONArray的转化

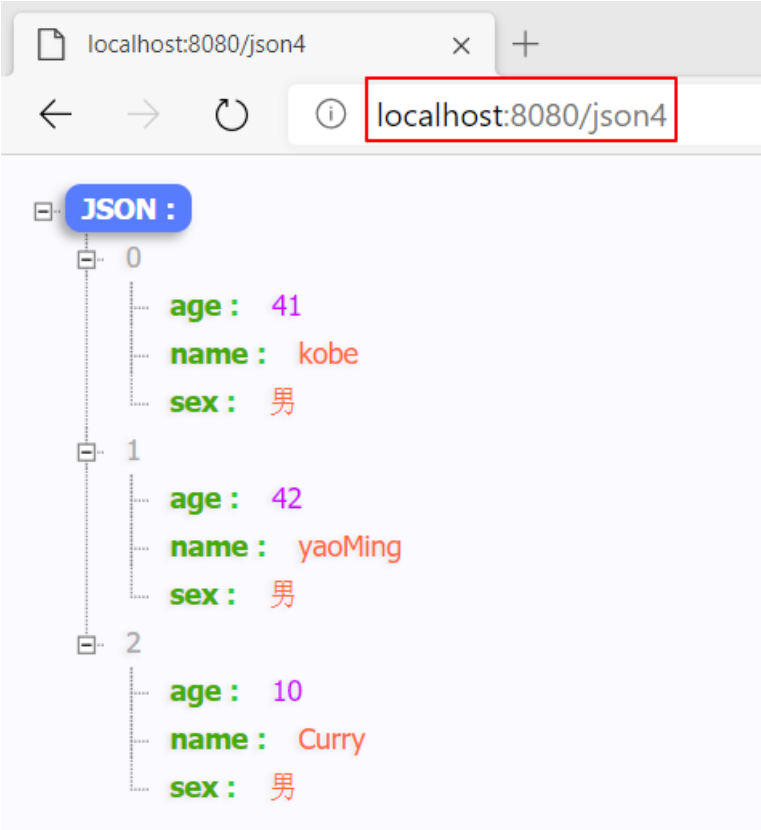
JSON类源码分析与使用。
仔细观察这些方法，主要是实现json对象，json对象数组，javabean对象，json字符串之间的相互转化。

7.3.2 FastJson(Controller)

```
/***
* fastjson
* @return
* @throws JsonProcessingException
*/
@RequestMapping("/json4")
@ResponseBody
public String json4() throws JsonProcessingException {
    // 1.创建list2集合
    List<User> list2 = new ArrayList<>();
    // 2.给User对象添加属性
    User user1 = new User("kobe", 41, "男");
    User user2 = new User("yaoMing", 42, "男");
    User user3 = new User("Curry", 10, "男");
    // 3.将对象添加到集合中
    list2.add(user1);
    list2.add(user2);
    list2.add(user3);
    // 4.将集合对象转成JSON字符串
    String str = JSON.toJSONString(list2);

    // return "str";

    System.out.println("*******Java对象 转 JSON字符串*******");
    String str1 = JSON.toJSONString(list2);
    System.out.println("JSON.toJSONString(list)==>"+str1);
    String str2 = JSON.toJSONString(user1);
    System.out.println("JSON.toJSONString(user1)==>"+str2);

    System.out.println("\n****** JSON字符串 转 Java对象*******");
    User jp_user1=JSON.parseObject(str2,User.class);
```

```
    System.out.println("JSON.parseObject(str2,User.class)==>"+jp_user1);

    System.out.println("\n****** Java对象 转 JSON对象 ******");
    JSONObject jsonObject1 = (JSONObject) JSON.toJSON(user2);
    System.out.println("(JSONObject)
JSON.toJSON(user2)==>"+jsonObject1.getString("name"));

    System.out.println("\n****** JSON对象 转 Java对象 ******");
    User to_java_user = JSON.toJavaObject(jsonObject1, User.class);
    System.out.println("JSON.toJavaObject(jsonObject1,
User.class)==>"+to_java_user);

    return "success";
    }
}
```

执行结果



8- Ajax

8.1 什么是ajax

异步JavaScript和XML (Asynchronous Javascript And XML)

特点:

1、异步的访问方式，使用到的技术：JavaScript和XML。

2、JavaScript：用于后台请求的发送和响应数据的接收，(以前提交表单或在浏览器上输入地址)。

3、XML：用于封装服务器发送的大量的数据，因为XML无关的数据太多，而且解析比较麻烦。所以目前几乎不再使用了，使用JSON格式来代替。

8.2 同步和异步区别

> **同步方式**

浏览器与服务器是串行的操作，浏览器发工作的时候，服务器没有处理数据的。

服务器在工作的时候，浏览器只能等待。以前使用JSP开发的方式都是同步的方式。

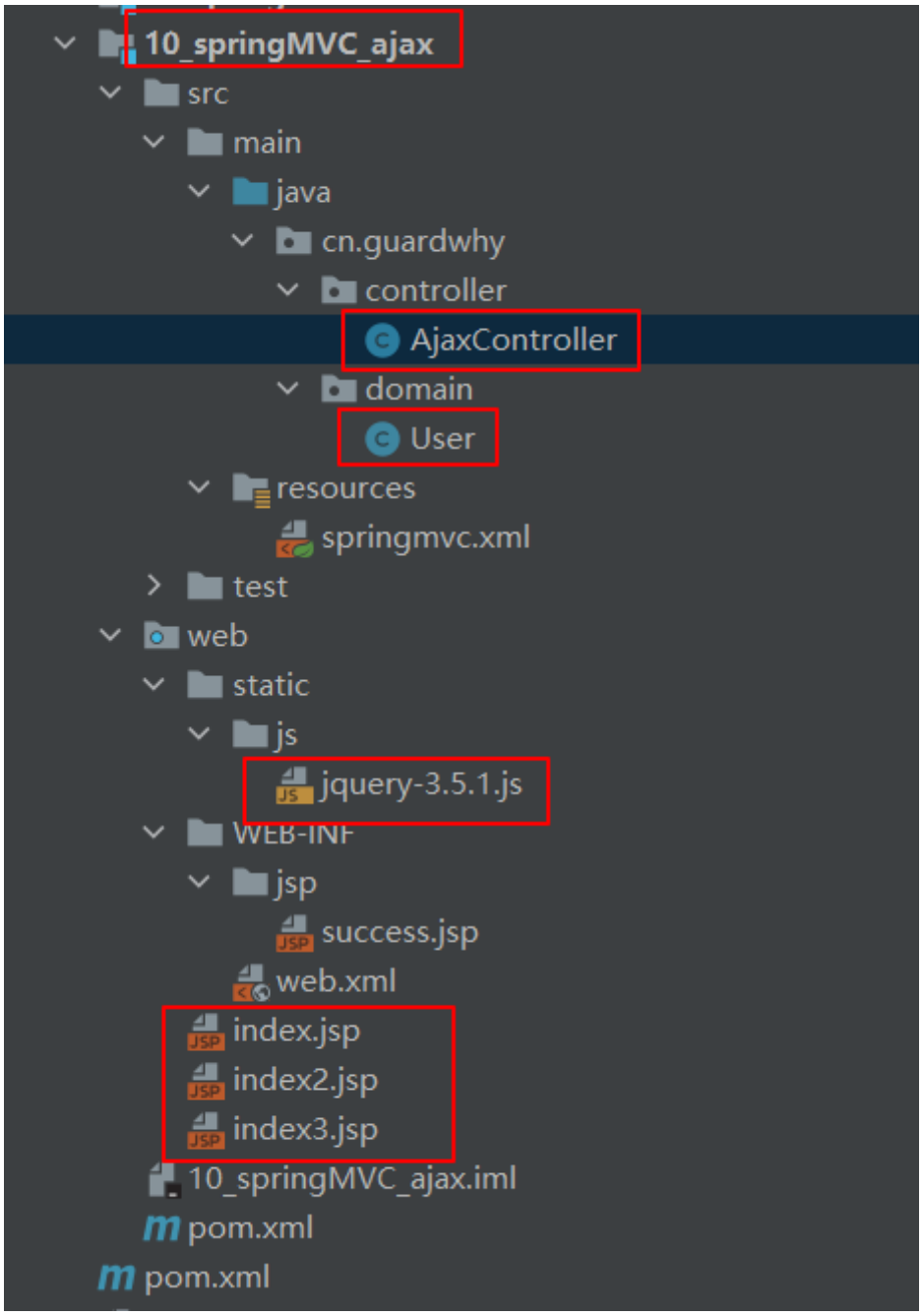
缺点：执行效率低，用户体验差。

> **异步方式**

以后逐渐会使用异步的开发，浏览器与服务器是并行工作的。很大企业开发中，是同步和异步并存的方式。

优点：执行效率高，用户体验更好。

8.3 项目目录



> **配置web.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
         version="4.0">

    <!--1.注册DispatcherServlet-->
    <servlet>
        <servlet-name>springmvc</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

        <!--关联一个springmvc的配置文件-->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:springmvc.xml</param-value>
        </init-param>
        <!--启动级别-1-->
        <load-on-startup>1</load-on-startup>
    </servlet>
    <!--所有的请求都会被springmvc拦截-->
    <servlet-mapping>
        <servlet-name>springmvc</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
    <!--配置解决中文乱码的过滤器-->
    <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>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

> **配置springMVC.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">
```

```
    <!--1.自动扫描包,让指定包下的注解生效,由IOC容器统一管理-->
    <context:component-scan base-package="cn.guardwhy"/>

    <!--2.让Spring MVC处理静态资源-->
    <mvc:default-servlet-handler/>

    <!--3.支持mvc注解驱动-->
    <mvc:annotation-driven>
        <!--JSON格式乱码处理方式-->
        <mvc:message-converters register-defaults="true">
            <bean
class="org.springframework.http.converter.StringHttpMessageConverter">
                <constructor-arg value="UTF-8"/>
            </bean>
            <bean
class="org.springframework.http.converter.json.MappingJackson2HttpMessageConvert
er">
                <property name="objectMapper">
                    <bean
class="org.springframework.http.converter.json.Jackson2ObjectMapperFactoryBean">
                        <property name="failOnEmptyBeans" value="false"/>
                    </bean>
                </property>
            </bean>
        </mvc:message-converters>
    </mvc:annotation-driven>

    <!--4.视图解析器 -->
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver"
id="InternalResourceViewResolver">
        <!--前缀-->
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <!--后缀-->
        <property name="suffix" value=".jsp"/>
    </bean>

</beans>
```

8.4 控制器Controller1

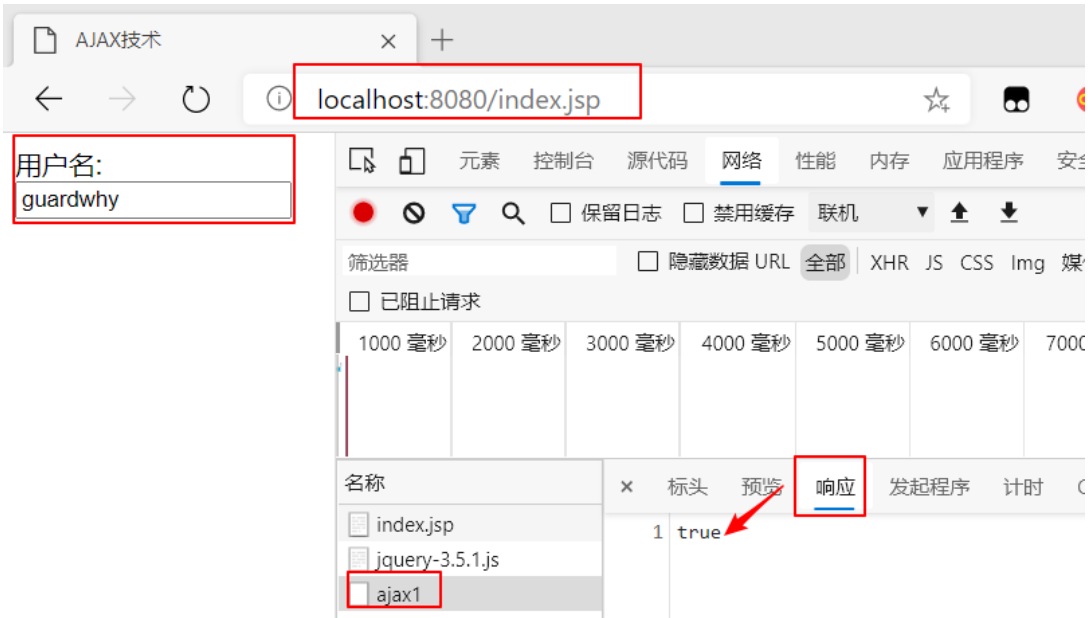
```
@RestController
public class AjaxController {
    /***
     * 判断用户名是否存在
     * @param name
     * @param response
     * @throws IOException
     */
    @RequestMapping("/ajax1")
    public void ajax1(String name, HttpServletResponse response) throws
IOException {
        // 1.条件判断
        if("guardwhy".equals(name)){
            response.getWriter().print("true");
        }else {
            response.getWriter().print("false");
```

```
        }
    }
```

> 视图层

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
  <head>
    <title>AJAX技术</title>
    <%--引入jquery--%>
    <script src="${pageContext.request.contextPath}/static/js/jquery-3.5.1.js">
</script>

    <script type="text/javascript">
        function ajax1(){
          $.post({
            url:"${pageContext.request.contextPath}/ajax1",
            data:{'name':$("#username").val()},
            success: function (data, status){
              console.log(data);
              console.log(status);
            }
          });
        }
    </script>
  </head>
  <body>
      <%--onblur:失去焦点触发事件--%>
      用户名:<input type="text" id="username" onblur="ajax1()"/>
  </body>
</html>
```

执行结果



8.5 控制器Controller2

```
/***
* 返回集合数据
* @return
*/
@RequestMapping("/ajax2")
public List<User> ajax2() {
    // 1.创建list集合
    List<User> list = new ArrayList<User>();
    // 2.将对象添加到集合中
    list.add(new User("kobe", 41, "男"));
    list.add(new User("James", 35, "男"));
    list.add(new User("Curry", 10, "男"));

    // 3.将集合转换成JSON格式返回
    return list;
}
```

> **视图层**

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>异步请求获取集合元素</title>
</head>
<body>
    <input type="button" id="btn" value="获取集合数据"/>
    <table width="100%" align="center">
        <tr>
            <td>姓名</td>
            <td>年龄</td>
            <td>性别</td>
        </tr>

        <tbody id="content"></tbody>
    </table>

    <%--引入jquery--%>
    <script src="${pageContext.request.contextPath}/static/js/jquery-3.5.1.js">
</script>

    <script type="text/javascript">
       $(function (){
           $("#btn").click(function (){
               $.post("${pageContext.request.contextPath}/ajax2", function
(data){
                   console.log(data);
                   let html= "";
                   for (let i = 0; i < data.length; i++) {
                       html += "<tr>" +
                           "<td>" + data[i].name + "</td>" +
                           "<td>" + data[i].age + "</td>" +
                           "<td>" + data[i].sex + "</td>" +
                           "</tr>"
                   }
```

```
                $("#content").html(html);
            });
        })
    })
</script>
</body>
</html>
```

执行结果

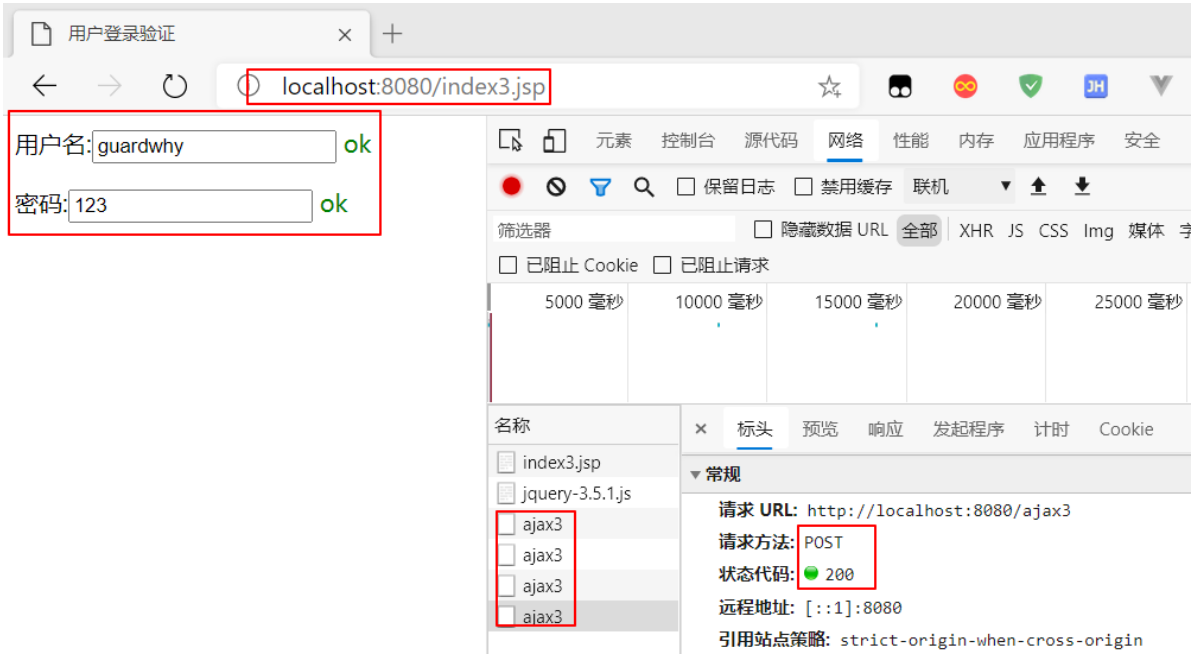
8.6 控制器Controller3

```
/***
* 用户名密码是否正确案例
* @param name
* @param pwd
* @return
*/
@RequestMapping("/ajax3")
public String ajax3(String name, String pwd){
    // 1.定义str字符串
    String str = "";
    // 2.判断用户名是否正确
    if(name != null){
       if("guardwhy".equals(name)){
           str = "ok";
        }else {
           str = "用户名错误";
          }
     }
    // 3.判断密码是否正确
    if(pwd != null){
        if("123".equals(pwd)){
            str = "ok";
         }else {
              str = "密码错误";
         }
      }

   // 4.将str转换成JSON格式返回
  return str;
}
```

> 视图层

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>用户名和密码判断</title>
    <%--引入Jquery--%>
    <script src="${pageContext.request.contextPath}/static/js/jquery-3.5.1.js">
</script>
    <script type="text/javascript">
        function fun1(){
            $.post({
                url:"${pageContext.request.contextPath}/ajax3",
                data:{'name':$("#name").val()},
                success:function (data){
                    if(data.toString()=='ok'){
                        $("#userInfo").css("color","green");
                    }else {
                        $("#userInfo").css("color","red");
                    }
                    $("#userInfo").html(data);
                }
            });
        }

        function fun2(){
            $.post({
                url:"${pageContext.request.contextPath}/ajax3",
                data:{'pwd':$("#pwd").val()},
                success:function (data){
                    if(data.toString()=='ok'){
                        $("#pwdInfo").css("color","green");
                    }else {
                        $("#pwdInfo").css("color","red");
                    }
                    $("#pwdInfo").html(data);
                }
            });
        }
    </script>
</head>
<body>
    <p>
        用户名:<input type="text" id="name" onblur="fun1()">
        <span id="userInfo"></span>
    </p>
    <p>
        密码:<input type="text" id="pwd" onblur="fun2()">
        <span id="pwdInfo"></span>
    </p>
</body>
</html>
```

执行结果



9- 拦截器

9.1 拦截器(interceptor)概述

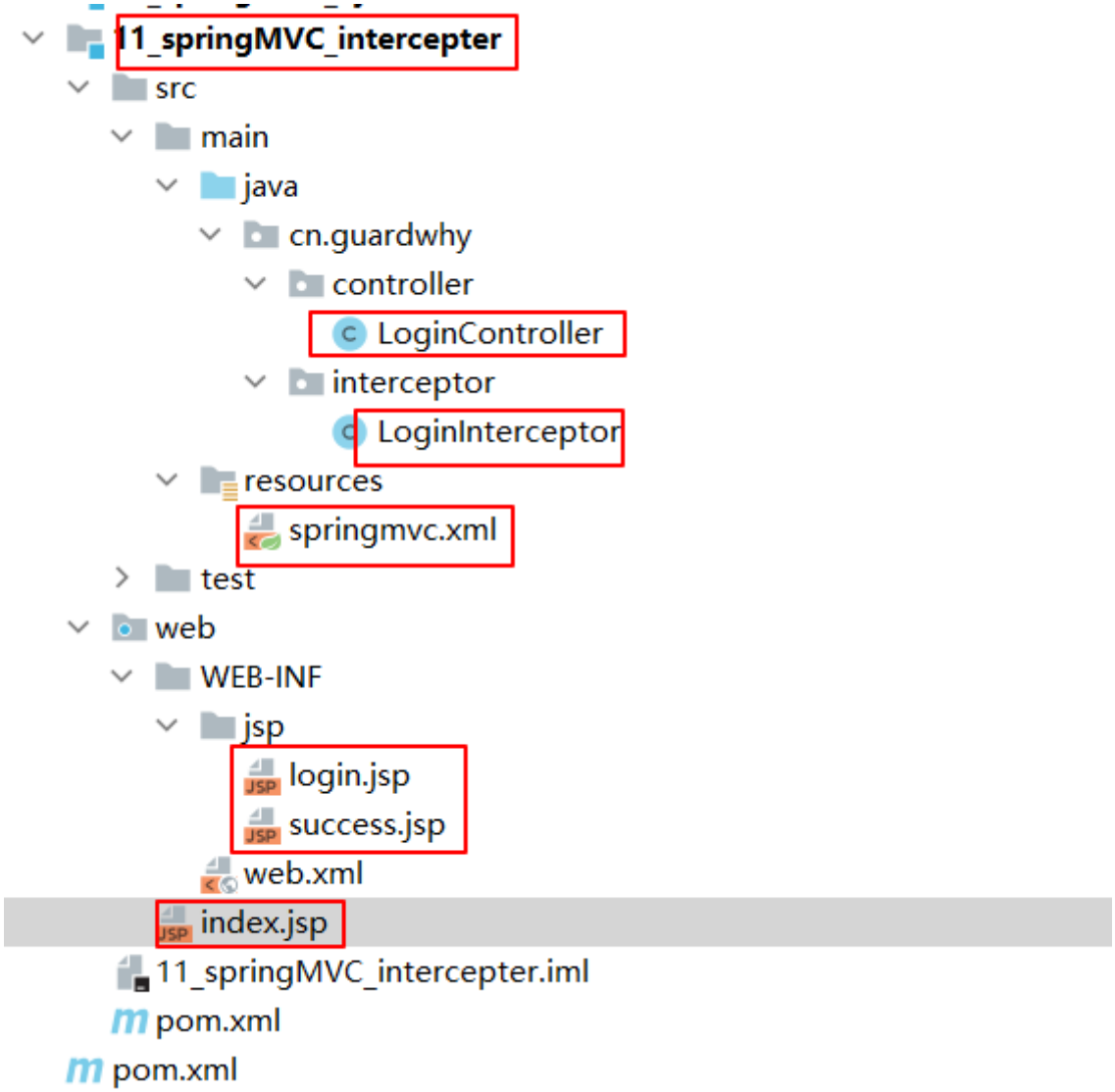
Spring MVC 的拦截器类似于 Servlet 开发中的过滤器 Filter，用于对处理器进行预处理和后处理。将拦截器按一定的顺序联结成一条链，这条链称为拦截器链。

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。拦截器也是AOP思想的具体实现。

9.2 过滤器与拦截器的区别

区别	过滤器	拦截器
使用范围	是 servlet 规范中的一部分，任何 Java Web 工程都可以使用	是 SpringMVC 框架自己的，只有使用了 SpringMVC 框架的工程才能用
拦截范围	在 url-pattern 中配置了/*之后，可以对所有要访问的资源拦截	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

9.3 项目目录



> 实现思路

- 有一个登陆页面，需要写一个controller访问页面。登陆页面有一提交表单的动作，需要在controller中处理。
- 判断用户名密码是否正确。如果正确，向session中写入用户信息。返回登陆成功。
- 拦截用户请求，判断用户是否登陆。如果用户已经登陆。放行， 如果用户未登陆，跳转到登陆页面。

9.4 首页(index.jsp)

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
  <head>
    <title>首页</title>
  </head>
  <body>
    <h3><a href="${pageContext.request.contextPath}/user/jumpLogin">登录页面</a>
</h3>
    <h3><a href="${pageContext.request.contextPath}/user/success">首页</a> </h3>
  </body>
</html>
```

9.5 登录页面(login.jsp)

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>登录页面</title>
</head>
<body>
    <%--在WEB-INF下的所有页面或者资源，只能通过controller或者servlet进行访问--%>
    <h3>登录页面</h3>
    <form action="${pageContext.request.contextPath}/user/login" method="post">
        用户名:<input type="text" name="username"/>
        密码:<input type="text" name="password"/>
        <input type="submit" value="提交">
    </form>
</body>
</html>
```

9.6 控制器(Controller)

```
package cn.guardwhy.controller;

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

import javax.servlet.http.HttpSession;

@Controller
@RequestMapping("/user")
public class LoginController {
    /***
     * 登录成功页面
     * @param username
     * @param pwd
     * @param session
     * @return
     */
    @RequestMapping("/login")
    public String login(String username, String pwd, HttpSession session, Model
model){
        // 1.1 像session记录用户身份信息
        System.out.println("接收前端===" + username);
        // 1.2 将用户的登录信息存放在session中
        session.setAttribute("userLoginInfo", username);
        // 1.3 添加username传给页面
        model.addAttribute("username", username);
        // 1.3 返回该页面
        return "success";
    }

    /***
     * 直接跳转到登录页面
     * @return
     */
    @RequestMapping("/jumpLogin")
```

```
    public String login(){
        // 返回登录页面
        return "login";
    }

    /***
     * 直接跳转到登录成功页面
     * @return
     */
    @RequestMapping("/success")
    public String success(){
        // 直接跳转到登录成功页面
        return "success";
    }

    /***
     * 退出登陆页面
     * @return
     */
    @RequestMapping("/logout")
    public String logout(HttpSession session){
        // session 过期
        session.removeAttribute("userLoginInfo");
        // 返回登录页面
        return "login";
    }
}
```

9.6 拦截器

```
package cn.guardwhy.interceptor;

import org.springframework.web.servlet.HandlerInterceptor;

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

/**
 * 用户登录拦截器
 */
public class LoginInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        // 1.创建session对象
        HttpSession session = request.getSession();

        // 2.提交登录操作会放行
        if(request.getRequestURI().contains("login")){
            return true;
        }
        // 3.第一次登入,没有session，所以放行
        if(session.getAttribute("userLoginInfo") != null){
            return true;
        }
        // 4.登录页面会放行
```

```
        if(request.getRequestURI().contains("jumpLogin")){
            return true;
        }

        // 5.判断什么情况下没有登录
        request.getRequestDispatcher("/WEB-INF/jsp/login.jsp").forward(request,
response);
        return false;
    }
}
```

9.7 配置springmvc.xml

在Springmvc的配置文件中注册拦截器

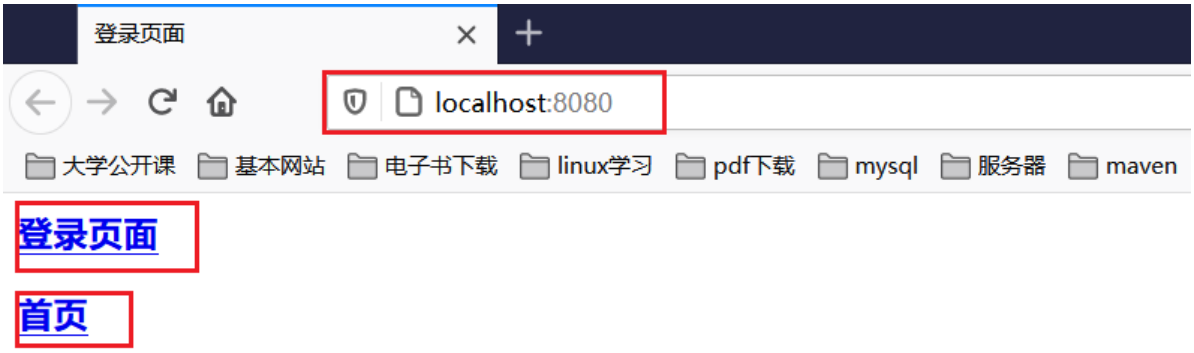
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="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

    <!--1.自动扫描包,让指定包下的注解生效,由IOC容器统一管理-->
    <context:component-scan base-package="cn.guardwhy"/>

    <!--2.让Spring MVC处理静态资源-->
    <mvc:default-servlet-handler/>
    <!--3.支持mvc注解驱动-->
    <mvc:annotation-driven>
        <!--JSON格式乱码处理方式-->
        <mvc:message-converters register-defaults="true">
            <bean
class="org.springframework.http.converter.StringHttpMessageConverter">
                <constructor-arg value="UTF-8"/>
            </bean>
            <bean
class="org.springframework.http.converter.json.MappingJackson2HttpMessageConvert
er">
                <property name="objectMapper">
                    <bean
class="org.springframework.http.converter.json.Jackson2ObjectMapperFactoryBean">
                        <property name="failOnEmptyBeans" value="false"/>
                    </bean>
                </property>
            </bean>
        </mvc:message-converters>
    </mvc:annotation-driven>

    <!--4.视图解析器 -->
```

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

    <!--5.配置拦截器-->
    <mvc:interceptors>
        <mvc:interceptor>
            <mvc:mapping path="/user/**"/>
            <bean class="cn.guardwhy.interceptor.LoginInterceptor"/>
        </mvc:interceptor>
    </mvc:interceptors>
</beans>
```

9.8 登录成功页面(success.jsp)

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>登录成功页面</title>
</head>
<body>
    <h3>恭喜你，登入成功</h3>
    <p>${username}</p>

    <%--注销--%>
    <a href=${pageContext.request.contextPath}/user/logout>注销</a>
</body>
</html>
```

> 执行结果

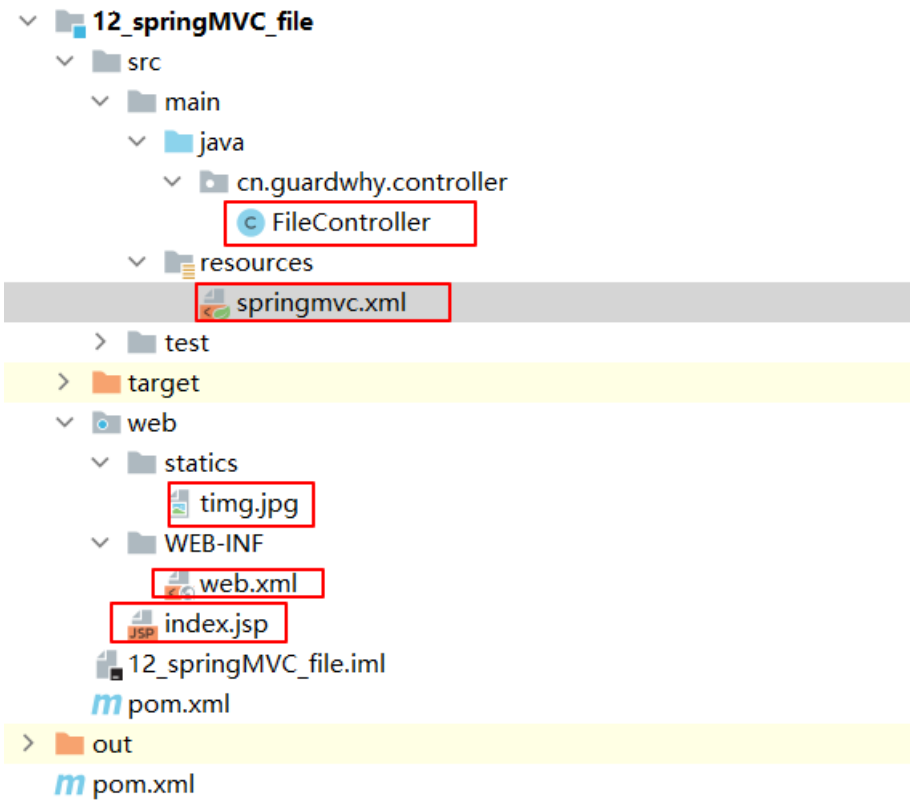


10- 文件上传和下载

文件上传是项目开发中最常见的功能之一 ,springMVC 可以很好的支持文件上传，但是SpringMVC上下文中默认没有装配MultipartResolver，因此默认情况下其不能处理文件上传工作。

如果想使用Spring的文件上传功能，则需要在上下文中配置MultipartResolver。

10.1 项目目录



10.2 导入相关依赖

导入文件上传的jar包，commons-fileupload ， Maven会自动帮我们导入他的依赖包 commons-io包；

```
<!--导入文件上传-->
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.3.3</version>
</dependency>
<!--servlet-->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>4.0.1</version>
</dependency>
```

10.3 配置springmvc.xml

这个bena的id必须为：multipartResolver，否则上传文件会报400的错误！！！

```
<!--文件上传配置-->
<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <!-- 请求的编码格式，必须和jSP的pageEncoding属性一致，以便正确读取表单的内容，默认为
ISO-8859-1 -->
    <property name="defaultEncoding" value="utf-8"/>
    <!-- 上传文件大小上限，单位为字节（10485760=10M） -->
    <property name="maxUploadSize" value="10485760"/>
    <property name="maxInMemorySize" value="40960"/>
</bean>
```

CommonsMultipartFile的常用方法：

- String getOriginalFilename()：获取上传文件的原名。
- InputStream getInputStream()：获取文件流。
- void transferTo(File dest)：将上传文件保存到一个目录文件中。

10.4 前端页面

为了能上传文件，必须将表单的method设置为POST，并将enctype设置为multipart/form-data。
只有在这样的情况下，浏览器才会把用户选择的文件以二进制数据发送给服务器。

> **enctype属性说明**

- application/x-www=form-urlencoded：默认方式，只处理表单域中的 value 属性值，采用这种编码方式的表单会将表单域中的值处理成 URL 编码方式。
- multipart/form-data：这种编码方式会以二进制流的方式来处理表单数据，这种编码方式会把文件域指定文件的内容也封装到请求参数中，

  不会对字符编码。

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
  <head>
    <title>下载与上传文件</title>
  </head>
  <body>
    <%--上传--%>
    <form action="${pageContext.request.contextPath}/upload2"
enctype="multipart/form-data" method="post">
      <input type="file" name="file"/>
      <input type="submit" value="upload">
    </form>

    <%--下载图片--%>
    <a href="${pageContext.request.contextPath}/statics/timg.jpg">下载图片</a>
  </body>
</html>
```

注意事项

- 一旦设置了enctype为multipart/form-data，浏览器即会采用二进制流的方式来处理表单数据，而对于文件上传的处理则涉及在服务器端解析原始的HTTP响应。
- Spring MVC为文件上传提供了直接的支持，这种支持是用即插即用的MultipartResolver实现的。
- Spring MVC使用Apache Commons FileUpload技术实现了一个MultipartResolver实现类：CommonsMultipartResolver。

- 因此，SpringMVC的文件上传还需要依赖Apache Commons FileUpload的组件。

10.5控制器(Controller)

```
package cn.guardwhy.controller;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.multipart.commons.CommonsMultipartFile;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.*;
import java.net.URLEncoder;

@RestController
public class FileController {
    /***
     * 上传文件(方式1)
     * @param file
     * @param request
     * @return
     * @throws IOException
     */
    @RequestMapping("/upload")
    public String fileUpload(@RequestParam("file") CommonsMultipartFile file,
                             HttpServletRequest request) throws IOException {

        //1.获取文件名 : file.getOriginalFilename();
        String uploadFileName = file.getOriginalFilename();
        //2.如果文件名为空，直接回到首页
        if ("".equals(uploadFileName)) {
            return "redirect:/index.jsp";
        }
        System.out.println("上传文件名 : " + uploadFileName);
        //3. 上传路径保存设置
        String path = request.getServletContext().getRealPath("/upload");
        //4. 如果路径不存在，创建一个
        File realPath = new File(path);
        if (!realPath.exists()) {
            realPath.mkdir();
        }
        System.out.println("上传文件保存地址：" + realPath);
        //5. 文件输入流
        InputStream is = file.getInputStream();
        //6. 文件输出流
        OutputStream os = new FileOutputStream(new File(realPath,
uploadFileName));
        //7. 读取写出
        int len = 0;
        byte[] buffer = new byte[1024];
        while ((len = is.read(buffer)) != -1) {
            os.write(buffer, 0, len);
            os.flush();
        }
        // 8.关闭资源
```

```
        os.close();
        is.close();
        // 9.重定向到index页面
        return "redirect:/index.jsp";
    }

    /***
     * 采用file.Transto 来保存上传的文件
     * @param file
     * @param request
     * @return
     * @throws IOException
     */
    @RequestMapping("/upload2")
    public String fileUpload2(@RequestParam("file") CommonsMultipartFile file,
                              HttpServletRequest request) throws IOException {
        //1.上传路径保存设置
        String path = request.getServletContext().getRealPath("/upload");
        File realPath = new File(path);
        if (!realPath.exists()) {
            realPath.mkdir();
        }
        //2. 上传文件地址
        System.out.println("上传文件保存地址：" + realPath);
        //3. 通过CommonsMultipartFile的方法直接写文件
        file.transferTo(new File(realPath + "/" + file.getOriginalFilename()));
        // 4.重定向到index页面
        return "redirect:/index.jsp";
    }

    /***
     * 下载资源
     * @param response
     * @param request
     * @return
     * @throws Exception
     */
    @RequestMapping(value = "/download")
    public String downloads(HttpServletResponse response,
                            HttpServletRequest request) throws Exception {
        //1. 要下载的图片地址
        String path = request.getServletContext().getRealPath("/upload");
        String fileName = "timg.jpg";
        //2.设置response 响应头
        response.reset();
        //3.字符编码
        response.setCharacterEncoding("UTF-8");
        //4.二进制传输数据
        response.setContentType("multipart/form-data");
        //5.设置响应头
        response.setHeader("Content-Disposition", "attachment;fileName=" +
URLEncoder.encode(fileName, "UTF-8"));
        File file = new File(path, fileName);
        //6.读取文件--输入流
        InputStream input = new FileInputStream(file);
        //7. 写出文件--输出流
        OutputStream out = response.getOutputStream();
        byte[] buff = new byte[1024];
```

```
        int index = 0;
        //8. 执行 写出操作
        while ((index = input.read(buff)) != -1) {
            out.write(buff, 0, index);
            out.flush();
        }
        // 9.关闭资源
        out.close();
        input.close();
        return "ok";
    }
}
```

10.6 执行结果