SpringMVC

1 概叙

1.1 回顾

- SSM：MyBatis + Spring + SpringMVC （延续了MVC三层架构思想)
- JavaSE：认真学习，老师带，入门快
- JavaWeb：认真学习，老师带，入门快
- SSM框架：研究官方文档，锻炼自学能力，锻炼笔记能力，锻炼项目能力
- SpringMVC + Vue - + SpringBoot + SpringCloud + Linux
- SSM 整合：JavaWeb做项目
- **Spring：IOC（控制反转） 和 AOP（面向切面编程）**
- **SpringMVC：SpringMVC的执行流程 (面试可能会问到)**
- SpringMVC：SSM框架整合！

1.2 MVC框架

1.2.1 MVC

- MVC是**模型（Model）、视图（View）、控制器（Controller）**的简写，是一种软件设计规范
- 是将**业务逻辑、数据、显示**分离的方式来组织代码
- MVC主要作用是降低了**视图与业务逻辑间的双向耦合**
- MVC不是一种设计模式，而是一种架构模式，当然不同的MVC存在差异

简单来说，所谓MVC就是**Model 模型（包括Pojo、Dao和Service，即数据和业务）+ View 视图（JSP/HTML，展示数据）+ Controller 控制器（Servle，获取请求和返回响应）**

1.2.2 MVC结构

Model（模型）：**数据模型，提供要展示的数据，包含数据和行为**，可以认为是领域模型或JavaBean组件（包含数据和行为）。不过一般都分开来写：**数据Dao层（Value Object）和服务层（行为Service）**，也就是模型提供了模型数据查询和模型数据的状态更新等功能，包括数据和业务

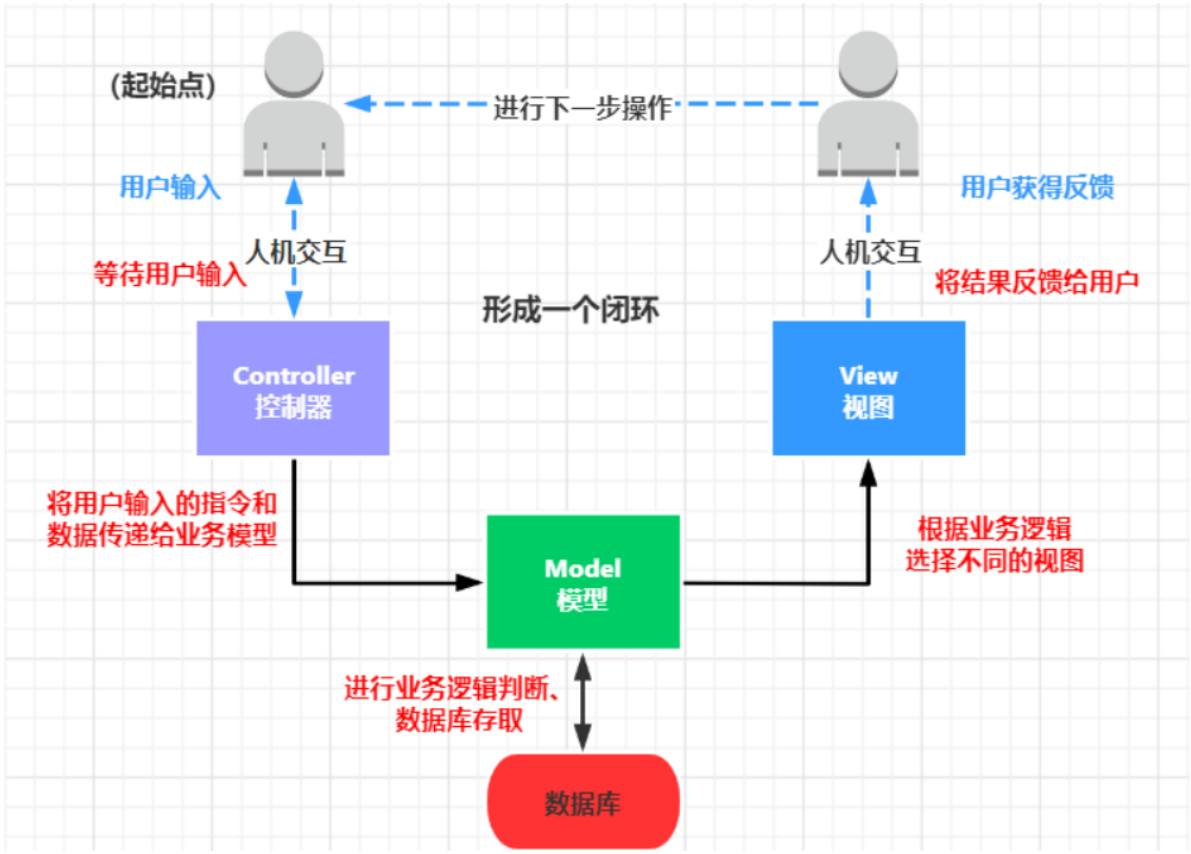
View（视图）：负责进行模型的展示，一般就是我们见到的用户界面，客户想要看到的东西

Controller（控制器）：接收用户请求，委托给模型进行处理（状态改变），处理完毕后把返回的模型数据返回给视图，由视图负责展示，也就是控制器做了个调度员的工作

- Dao和Service
- Servlet：转发和重定向
- JSP/HTML

最典型的MVC就是**JSP + Servlet + JavaBean的模式**

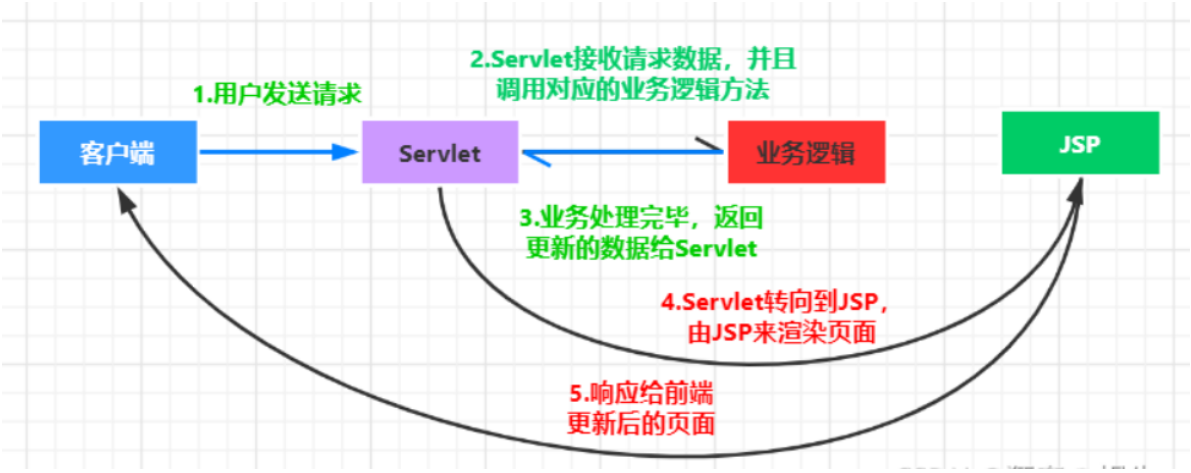
执行流程



1.2.4 知识扩充

> 例如有个User实体类：该实体类有用户名，密码，生日，爱好 ... 等20个字段，而前端需要用户名和密码字段的数据

- 前端：进行数据传输 (传递实体类的数据)
- Pojo：User实体类
- Vo：UserVo（用户视图对象）比如只写两个字段：用户名和密码，然后传给前端 (相当于将实体类进行了细分，本质上还是实体类对象)
- Dto：UserDto (用户数据传输对象) 数据传输目标往往是Dao(数据访问对象)从数据库中检索数据



职责分析：

- Controller：控制器
  - 取得表单数据
  - 调用业务逻辑
  - 转向指定的页面
- Model：模型

    - 业务逻辑
    - 保存数据的状态
- View：视图
    - 显示页面

1.2.5 MVC / MVVM

MVC：M：Model (模型层)； V：View (视图层)； C：Controller (控制层)

MVVM：M：Model (模型层)；V：View (视图层)；VM： ViewModel(视图模型层)，即双向数据绑定

1.3 回顾servlet

总项目中的资源依赖

```
<!-- 导入相应的资源依赖 -->
<dependencies>
    <!-- spring-webmvc资源依赖 -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.2.4.RELEASE</version>
    </dependency>
    <!-- servlet-api资源依赖 -->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>servlet-api</artifactId>
        <version>2.5</version>
    </dependency>
    <!-- jsp-api资源依赖 -->
    <dependency>
        <groupId>javax.servlet.jsp</groupId>
        <artifactId>jsp-api</artifactId>
        <version>2.2</version>
    </dependency>
    <!-- jstl资源依赖 -->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jstl</artifactId>
        <version>1.2</version>
    </dependency>
    <!-- junit单元测试资源依赖 -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

module1中的相关资源jar包

```
<dependencies>
```

```
    <!--servlet-api资源依赖-->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>servlet-api</artifactId>
        <version>2.5</version>
    </dependency>
    <!--jsp-api资源依赖-->
    <dependency>
        <groupId>javax.servlet.jsp</groupId>
        <artifactId>jsp-api</artifactId>
        <version>2.2</version>
    </dependency>
</dependencies>
```

编写控制层Servlet和web.xml配置文件

```
package com.robin.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 {

    /**
     * 1.重写doGet方法
     */
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {

        /**
         * 1.1 获取前端参数
         */
        String method =req.getParameter("method");
        // 判断调用的请求是add方法还是delete方法
        if(method.equals("add")) {
            // 执行add方法
            req.getSession().setAttribute("msg","执行了add方法");
        }
        if(method.equals("delete")) {
            // 执行delete方法
            req.getSession().setAttribute("msg","执行了delete方法");
        }

       /**
        * 1.2 调用业务层
        * 为了省事，具体的业务逻辑就不写了
        */

        /**
         * 1.3 视图转或者重定向
         */
        // 1.3.1 使用转发请求
        req.getRequestDispatcher("/WEB-INF/jsp/test.jsp").forward(req,resp);
```

```
        // 1.3.2 或者使用重定向
//        resp.sendRedirect();
    }

    /**
     * 2.重写doPost方法
     */
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        doPost(req, resp);
    }

}
```

编写web.xml配置文件

```
<?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.配置servlet -->
    <servlet>
        <servlet-name>hello</servlet-name>
        <servlet-class>com.robin.servlet.HelloServlet</servlet-class>
    </servlet>
    <!-- 2.配置servlet-mapping -->
    <servlet-mapping>
        <servlet-name>hello</servlet-name>
        <!-- 请求hello的页面，然后会自动转发给hello的servlet来处理 -->
        <url-pattern>/hello</url-pattern>
    </servlet-mapping>
    <!-- 3.配置session -->
   <session-config>
        <!-- 设置超时时间，超过15分钟就自动关闭会话 -->
        <session-timeout>15</session-timeout>
    </session-config>
    <!-- 4.配置欢迎页面 -->
    <welcome-file-list>
         <!-- 默认欢迎页面设置为index.jsp -->
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

在web文件夹下的WEB-INF文件下，创建一个jsp文件夹，用来存放视图层的相关页面文件

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

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

2. SpringMVC核心

Spring MVC是Spring Framework的一部分，是基于Java实现MVC的轻量级Web框架

Spring MVC的特点，为啥要学习?

1. 轻量级，简单易学
2. 高效 , 基于请求响应的MVC框架
3. 与Spring兼容性好，无缝结合
4. 约定优于配置
5. 功能强大：RESTful、数据验证、格式化、本地化、主题等
6. 简洁灵活

Spring 和 SpringMVC

Spring本质上是个大杂烩，我们可以将SpringMVC中的所有要用到的Bean，注册到Spring的IOC容器中

2.1 DispatcherServlet

Spring的web框架围绕**DispatcherServlet [调度Servlet]** 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形式进行开发，十分简洁； 正因为SpringMVC好 , 简单 , 便捷 , 易学 , 天生和Spring无缝集成(使用SpringIoC和Aop) , 使用约定优于 配置 . 能够进行简单的junit测试 . 支持Restful风格 .异常处理 , 本地化 , 国际化 , 数据验证 , 类型转换 , 拦 截器等等……所以我们要学习 .

2.1.1 源码解析

- 首先DispatcherServlet这个中心控制器继承于FrameworkServlet类

```
public class DispatcherServlet extends FrameworkServlet { ... }
```

- 其次，抽象类FrameworkServlet又继承于HttpServletBean类，同时还实现了一个ApplicationContextAware接口

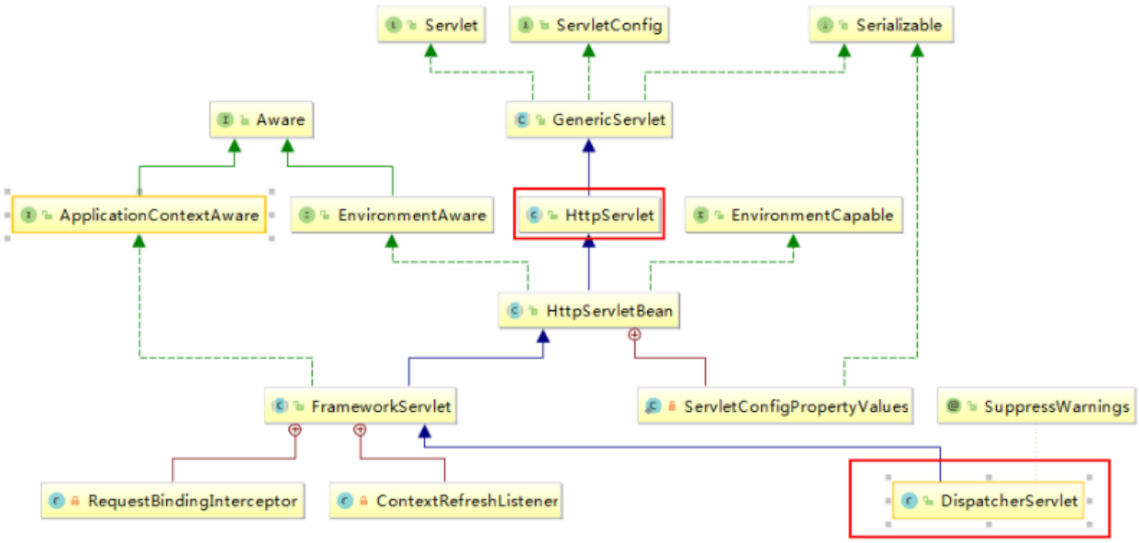
```
public abstract class FrameworkServlet extends HttpServletBean implements
ApplicationContextAware { ... }
```

- 最后，抽象类HttpServletBean继承于HttpServlet基类

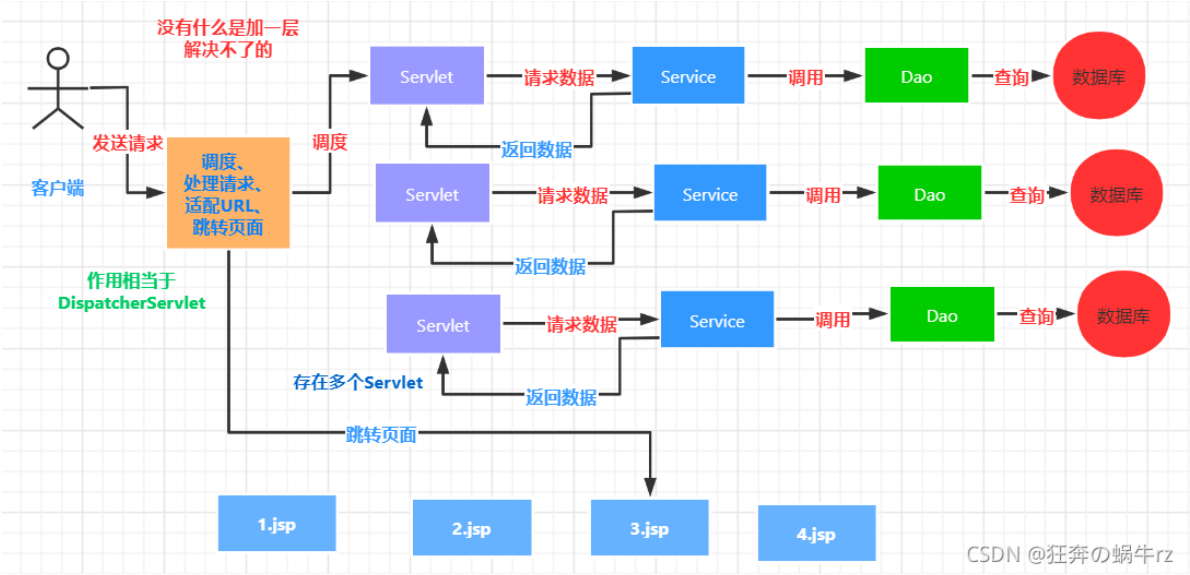
```
public abstract class HttpServletBean extends HttpServlet implements
EnvironmentCapable, EnvironmentAware { ... }
```

DispatcherServlet类确实继承于基类Servlet

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)。**



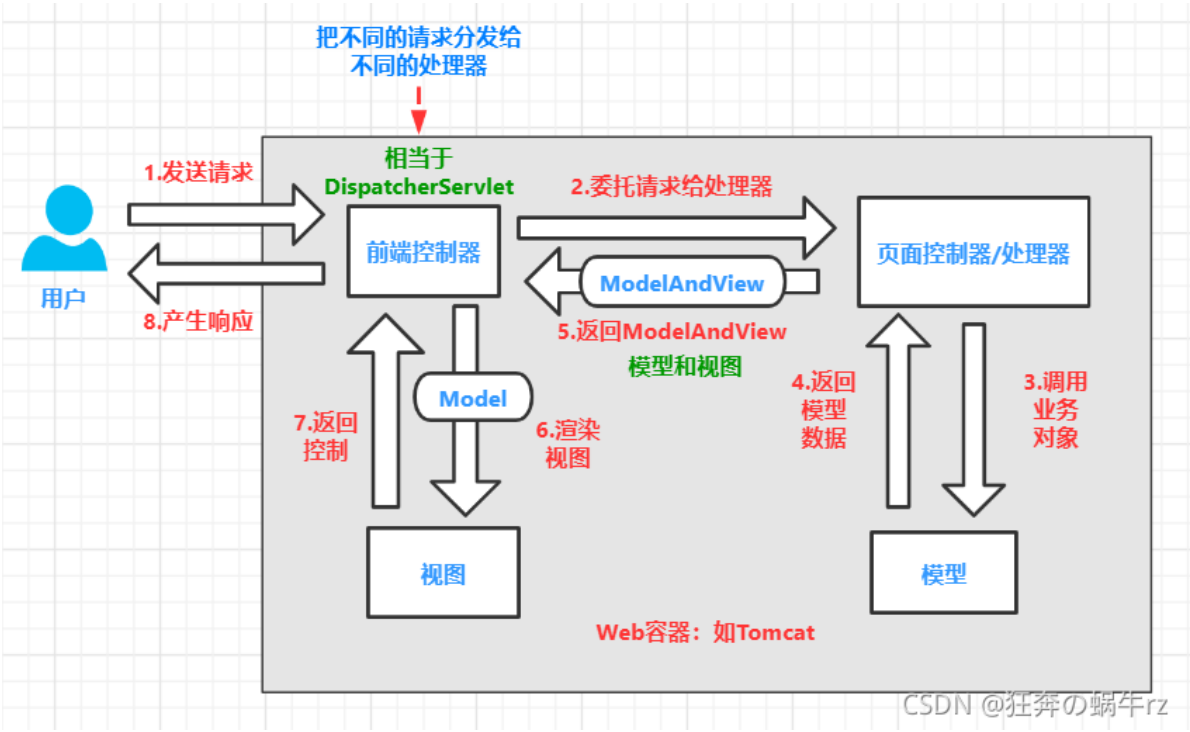
2.1.2 servlet的执行流程



1. **客户端发送请求，请求被Servlet (控制器) 拦截，它主要负责调度、处理请求和适配URL以及跳转页面等 (一个项目中会存在多个Servlet)**
2. **Servlet (控制器) 向Service (服务层) 请求数据，Service (服务层) 调用Dao(数据持久层) 查询数据**
3. **Dao (数据持久层) 通过查询数据库获取数据，将数据交由Service (服务层) 进行处理**
4. **Service (服务层) 将处理的数据返回给Servlet (控制器)，Servlet (控制器) 根据返回的数据信息来跳转对应的jsp页面**

总之，记住一句话，**没有什么是加一层解决不来了的!**

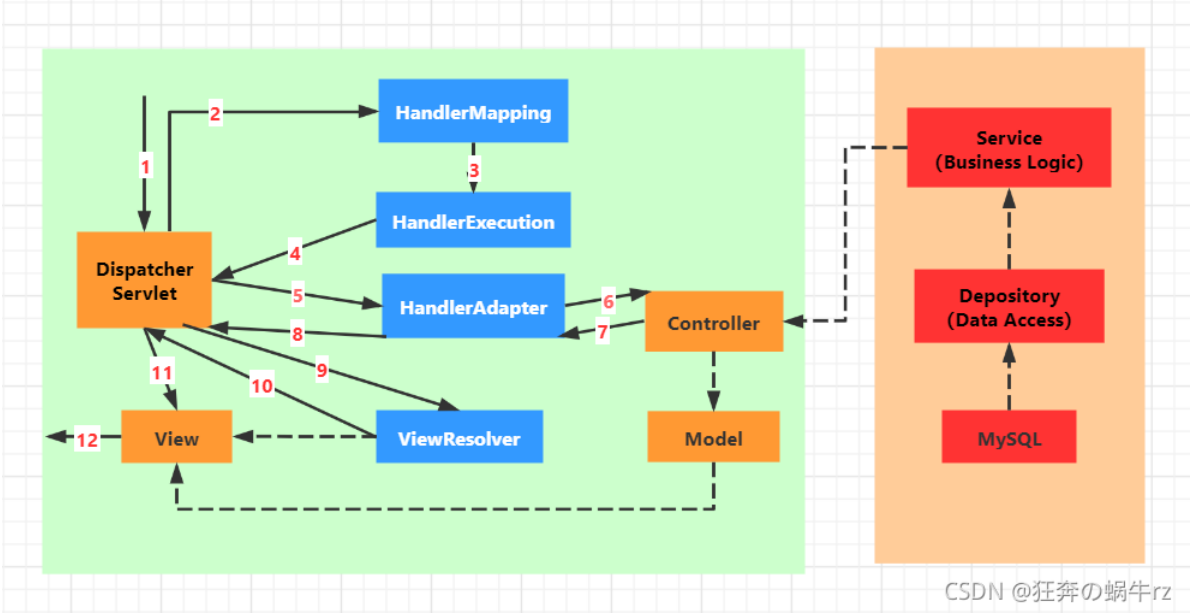
2.2 SpringMVC执行流程



1. 当用户或者客户端发送请求时，被前置的控制器拦截到请求
2. 根据请求参数生成代理请求，找到请求对应的实际控制器
3. 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
4. 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

###

SpringMVC的一个较为完整的流程图，实线表示SpringMVC框架提供的技术，不需要开发者实现，虚线则需要开发者实现



2.2.1 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。

我们假设请求的url为 : http://localhost:8080/SpringMVC/hello 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
- hello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. HandlerExecution表示具体的Handler,其主要作用是根据url查找控制器，如上url被查找控制器为：hello。
4. HandlerExecution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。

2.2.2 简化

1. 用户发送请求，前置控制器 (DispatcherServlet) 进行接收和拦截请求
2. 前置控制器 (DispatcherServlet) 将URL链接发送给处理器映射 (HandlerMapping)
3. 处理器映射 (HandlerMapping) 根据URL链接(或者xml和注解)查找具体的处理器 (Handler)
4. 处理器执行器 (HandlerExecution) 将具体的处理器返回给前置控制器 (DispatcherServlet)
5. 前置控制器(DispatcherServlet) 调用处理器适配器 (HandlerAdapter) 去执行处理器 (Handler)
6. 处理器适配器 (HandlerAdapter) 调用具体的处理器 (Handler) 去执行对应的控制器(Controller)
7. 控制器 (Controller) 返回视图模型 (ModelAndView) 给处理器适配器 (HandlerAdapter)
8. 处理器适配器 (HandlerAdapter) 将视图模型返回前置控制器(DispatcherServlet)
9. 前端控制器(DispatcherServlet) 将视图模型 (或视图逻辑名) 交由视图解析器 (ViewResolver) 进行处理
10. 视图解析器 (VewResolver) 将解析的视图结果返回给前置控制器(DispatcherServlet)
11. 前置控制器 (DispatcherServlet) 进行视图渲染，将视图模型(ModelAndView) 填充到request域
12. 前置控制器 (DispatcherServlet) 向用户响应结果，即将具体的视图 (View) 呈现给用

2.3 HelloSpringMVC

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">

        <!-- 注册DispatcherServlet
             配置DispatcherServlet(前置控制器)，DispatcherServlet主要作用: 这个是
SpringMVC的核心，请求分发器 -->
        <servlet>
            <servlet-name>springmvc</servlet-name>
            <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
            <!-- DispatcherServlet要绑定Spring的配置文件
                 关联一个SpringMVC的配置文件: 【servlet-name】- servlet.xml -->
            <init-param>
                <param-name>contextConfigLocation</param-name>
                <!-- classpath*: 会去所有的包中找，建议使用classpath: -->
                <param-value>classpath:springmvc-servlet.xml</param-value>
            </init-param>
            <!-- 启动级别 -->
            <load-on-startup>1</load-on-startup>
        </servlet>

        <!-- 在SpringMVC中，/ 和 /* 作用不相同:
             /: 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>
```

创建一个com.kuang.controller包，来存放控制层的HelloController类

```
package com.kuang.controller;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
//注意：这里要实现Controller接口，是web.servlet.mvc包下
public class HelloController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) {
        // 获取ModelAndView对象（模型和视图）
        ModelAndView mv = new ModelAndView();
        /**
         * 业务代码
         */
        // 设置返回结果
        String result = "HelloSpringMVC";
        // 封装对象，放在ModelAndView中
        mv.addObject("msg",result);
        // 视图跳转：封装要跳转的视图，放在ModelAndView中
        mv.setViewName("hello"); // 访问路径：/WEB-INF/jsp/hello.jsp
        return mv;
    }

}
```

springmvc-servlet.xml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">

    <!-- 配置一个URL处理器以及一个URL适配器:
         作用是将URL去匹配Spring中有哪一个Controller去处理它 -->

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

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

    <!-- InternalResourceViewResolver：表示视图解析器
         主要处理DispatcherServlet（中心控制器）给它的ModelAndView（模型视图）
        1. 获取了ModelAndView的数据
        2. 解析了ModelAndView的视图名字
        3. 拼接视图，找到对应的视图，/WEB-INF/jsp/hello.jsp
        4. 将数据渲染到这个视图上
         常用的模板引擎包括Thymeleaf和FreeMaker等 -->
    <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信息，将HelloController控制器交给Spring容器管理-->
    <bean id="/hello" class="com.kuang.controller.HelloController"></bean>

</beans>
```

- 在web文件夹中的WEB-INF文件下，创建一个jsp文件夹，用来存放视图层的相关页面文件

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <!-- 展示HellloController控制器中封装的对象信息 -->
    ${msg}
</body>
</html>
```

2.4 注解版helloSpringMVC

实现步骤

1. 新建一个web项目
2. 导入相关Jar包
3. 编写web.xml，注册DispatcherServlet
4. 编写springmvc配置文件
5. 创建对应的控制类, controller
6. 完善前端视图和controller之间的对应
7. 测试运行

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">

    <!-- 配置DispatcherServlet: 表示前置控制器
         主要作用: 这个是SpringMVC的核心，请求分发器 -->
    <servlet>
        <servlet-name>springmvc</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <!-- DispatcherServlet要绑定Spring的配置文件 -->
```

```
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <!-- classpath*:会去所有的包中找，建议使用classpath: -->
            <param-value>classpath:springmvc-servlet.xml</param-value>
        </init-param>
        <!-- 启动级别 -->
        <load-on-startup>1</load-on-startup>
    </servlet>

    <!-- 在SpringMVC中，/ 和 /* 作用不相同
         /:  匹配所有的请求，不会去匹配jsp（推荐使用 /）
         /*: 匹配所有的请求，包括jsp页面 -->
    <servlet-mapping>
        <servlet-name>springmvc</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

</web-app>
```

springmvc-servlet.xml

```
<?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">

    <!-- 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="com.robin.controller"/>

    <!-- 让SpringMVC不处理静态资源: .css .js .html .mp3 .mp4 -->
    <mvc:default-servlet-handler/>
    <!-- 支持mvc注解驱动
         (在Spring中一般采用@RequstMapping注解来完成映射关系，要想让@RequstMapping注解生效，必须向上下文注册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/>

    <!-- 视图解析器 -->
    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"
          id="internalResourceViewResolver">
        <!-- 在视图解析器中我们吧所有的视图都放在/WEB-INF/目录下（这样可以保证视图安全，因为这个目录下的文件，客户端不能直接访问） -->
        <!-- 前缀 -->
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <!-- 后缀 -->
        <property name="suffix" value=".jsp"/>
    </bean>
```

```
</beans>
```

controller

```
package com.robin.controller;

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

// 使用@Controller注解，将该类注册为控制器，交由Spring的IOC容器统一管理
@Controller
// 使用@RequestMapping注解，设置请求映射：作用域是类或者方法
@RequestMapping("/hello")
public class HelloController {

    // 真实访问地址：localhost:8080/hello/h1
    @RequestMapping("/h1")
    public String hello1(Model model) {
        // 封装数据：向模型中添加属性msg与值，可以在JSP页面中取出并渲染
        model.addAttribute("msg","Hello, SpringMVCAnnotation!");
        // 视图解析器处理视图名称（注意：这里的"hello"是指jsp页面的名称）
        return "hello";
    }

    /**
     * 设置多个请求映射，用来跳转不同的页面
     */
    // 真实访问地址：localhost:8080/hello/h2
    @RequestMapping("/h2")
    public String hello2(Model model) {
        // 封装数据
        model.addAttribute("msg","Hello, SpringMVC!");
        // 视图解析器处理视图名称
        return "hello";
    }

    // 真实访问地址：localhost:8080/hello/h3
    @RequestMapping("/h3")
    public String hello3(Model model) {
        // 封装数据
        model.addAttribute("msg","Hello, Spring!");
        // 视图解析器处理视图名称
        return "hello";
    }

}
```

3. Controller + RequestMapping

3.1 概念

- Controller (即控制器)，它主要负责提供**访问应用程序的行为和解析用户的请求，并将其转换为一个模型**，通常通过接口定义或注解定义这两种方式来实现
- 在Spring MVC中一个控制器类可以包含多个方法，并且对于 Controller的配置方式 包含多种，包括接口定义和注解定义

3.2 Controller接口

handleRequest方法，用来处理请求并返回一个ModelAndView（模型和视图）对象

```
// 实现该接口的类获得控制器功能
public interface Controller {

    // 处理请求并返回一个模型与视图对象
    ModelAndView handleRequest(HttpServletRequest var1, HttpServletResponse
var2) throws Exception;

}
```

3.3 两种实现方法

1. 接口定义

```
// 只要实现了Controller接口的类，说明这就是一个控制器了
public class HelloController implements Controller {
    /**
     * 用于处理请求并返回一个模型与视图对象
     * @param: httpServletRequest http请求
     * @param: httpServletResponse http响应
     * @return ModelAndView 视图模型对象
     */
    public ModelAndView handleRequest(HttpServletRequest httpServletRequest,
HttpServletResponse httpServletResponse) throws Exception {
        // 获取ModelAndView(模型视图)对象
        ModelAndView mv = new ModelAndView();
        // 封装数据到ModelAndView对象中
        mv.addObject("msg","Hello,Controller!");
        // 设置跳转视图，存入到ModelAndView对象中
        mv.setViewName("hello");
        // 返回视图模型对象给视图解析器
        return mv;
    }
}
```

对应的springmvc-servlet.xml

```
 <!-- 注册控制器HelloController的Bean信息，交由Spring的IOC容器进行管理；
        该bean标签中的包含两个属性class和name，其中"class"对应处理请求的控制类，
而"name"对应请求路径-->
<bean class="com.kuang.controller.HelloController" name="/hello"/>
```

- 与使用注解相比，配置文件和控制层代码还是略过繁琐
- 每个控制层只能控制一个跳转页面，页面无法复用

2. 注解实现

- @Controller注解类型用于声明Spring类的实例是一个控制器（在讲IOC时还提到了另外3个注解）；
- Spring可以使用扫描机制来找到应用程序中所有基于注解的控制器类，为了保证Spring能找到你的控制器，需要在配置文件中声明组件扫描。

```
@Component：注册为组件
@Repository：注册为Dao持久层
@Service：注册为Service服务层
@Controller：注册为Controller控制层
```

```
// 使用@Controller注解，将其注册为控制层，由Spring的IOC容器统一管理
@Controller
public class HelloController2 {

    /**
     * 使用@Controller注解注解的类中的所有方法，
     * 如果返回类型是String，并且有具体的页面可以跳转，那么就会被视图解析器解析
     */

    // 使用@RequestMapping注解，设置请求映射，其作用域是类或者方法
    @RequestMapping("/h1")
    public String hello(Model model) {
        // 封装数据：向模型中添加属性msg与其值，可以在JSP页面取出并且渲染
        model.addAttribute("msg","Hello,Controller!");
        // 会被视图解析器处理，注意：这里的hello对应的是JSP页面的名字
        return "hello";
    }

    // 使用@RequestMapping注解，设置请求映射，其作用域是类或者方法
    @RequestMapping("/h2")
    public String hello2(Model model) {
        // 封装数据：向模型中添加属性msg与其值，可以在JSP页面取出并且渲染
        model.addAttribute("msg","Hello,World!");
        // 会被视图解析器处理(注意：这里的hello对应的是JSP页面的名字)
        return "hello";
    }

}
```

对应的springmvc-servlet.xml

```
<!-- 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="com.kuang.controller"/>
<!-- 设置default-servlet-handler用来防止静态资源被过滤 -->
<mvc:default-servlet-handler/>
<!-- 设置mvc的注解驱动，让@RequstMapping注解生效以及引入
DefaultAnnotationHandlerMapping和AnnotationMethodHandlerAdapter实例;
使用annotation-driven会帮助我们自动完成上述两个实例的注入 -->
<mvc:annotation-driven/>
```

可以发现，我们的两个请求都指向一个视图，但是页面结果却是不一样的，从这里可以看出视图是被**复用**的，因此**控制器与视图之间是弱耦合关系**

3.4 RequestMapping

在方法上使用

上述方法的默认实现是**WEB-INF/jsp/hello.jsp**

```
/**
     * 在hello()方法上使用@RequestMapping，设置请求映射路径
     * 而真实访问路径为：http://localhost:8080/项目名称/hello
     */
    @RequestMapping("/hello")
    public String hello(Model model) {
        // 封装数据：向模型中添加属性msg与其值，进行视图渲染
        model.addAttribute("msg","Hello,RequestMapping!");
        // 返回视图逻辑名，让视图解析器进行解析(注意：这里的hello是要跳转的视图逻辑名)
        return "hello";
    }
```

/hello 访问

```
/**
     * 使用@RequestMapping，设置请求映射路径 (注意：直接在方法上把路径补全)
     * 真实访问路径为：http://localhost:8080/项目名称/user/hello
     */
    @RequestMapping("/user/hello3")
    public String hello3(Model model) {
        // 封装数据：向模型中添加属性msg与其值，进行视图渲染
        model.addAttribute("msg","Hello,User!");
        // 返回视图逻辑名，让视图解析器进行解析 (注意：由于路径发生了改变，因此要跳转的视图逻
辑名也要相应变化)
        return "hello/user";
    }
```

- 将hello.jsp的存放路径修改为WEB-INF/jsp/user/hello.jsp，即在jsp文件下创建一个user，将hello.jsp放入其中
- /user.hello3访问

类加方法一起使用

```
// 使用@Controller注解，将该类注册为Controller接口，交由Spring的IOC容器统一管理
@Controller
// 在控制类上使用@RequestMapping: 相当于在URL链接上再加一个父目录进行区分
@RequestMapping("/user") // 先访问类上的/user
public class HelloController3 {

    // 真实访问路径变为了: http://localhost:8080/项目名称/user/hello2
    @RequestMapping("/hello2") //再访问/hello2
    public String hello2(Model model) {
        // 封装数据：向模型中添加属性msg与值，进行视图渲染
        model.addAttribute("msg","Hello,RequestMapping2!");
        // 返回视图逻辑名，让视图解析器进行解析（注意：由于路径发生了改变，因此要跳转的视图逻辑名也要相应变化）
        return "user/hello";
    }

}
```

/user/hello2访问

- **只在方法上使用@RequestMapping注解，代码编写更加简便，只用在每个方法前设置映射关系，而且方法中的页面的路径选择更加宽泛**
- **在类和方法上同时使用@RequestMapping注解，逻辑更加清晰，但页面的调用范围也受到了限制，即只能调用在类上的映射关系中设置的父目录下的页面**

4. Restful 风格

4.1 概括

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更加简洁，更有层次，更易于实现缓存等机制。

它主要用于实现资源操作 (互联网所有的事物都可以被抽象为资源)，即使用不同方法对资源进行操作。例如使用的请求方式为POST、DELETE、PUT和GET，分别对应添加、删除、修改和查询操作

4.2 对比

传统方式

```
http://localhost:8080/item/queryItem.action?id=1: 查询（对应GET请求）
http://localhost:8080/item/saveItem.action: 新增（对应POST请求）
http://localhost:8080/item/queryItem.action?id=1: 更新（对应POST请求）
http://localhost:8080/item/deleteItem.action?id=1: 删除（（对应POST或者GET请求））
```

传统方式操作资源：通过不同的参数来实现不同的效果，方法单一，例如使用post和get请求

Restful

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！**

```
http://localhost:8080/item/1: 查询（对应GET请求）
http://localhost:8080/item: 新增（对应POST请求）
http://localhost:8080/item: 更新（对应PUT请求）
http://localhost:8080/item/1: 删除（对应DELETE请求）
```

4.3 常规使用 vs Restful -Code

```
// 使用@Controller注解，将该类注册为Controller层，交由Spring的IOC容器统一管理
@Controller
public class RestfulController {

     /**
      * 使用默认方式的进行访问
      * 使用@RequestMapping注解，设置请求映射的访问路径
      * 其真实访问路径为：http://localhost:8888/add?a=1&b=2
      */
    @RequestMapping("/add")
    public String test(int a, int b, Model model) {
        // 设置一个结果
        int result = a + b;
        // 封装数据：向模型中添加属性msg与其值，进行视图渲染
        model.addAttribute("msg","结果为"+result);
        // 返回视图逻辑名，交由视图解析器进行处理
        return "user/hello";
    }

}
```

- 在add后面传递a和b的参数值才可以实现功能，即URL链接为：**localhsot:8080/add?a=1&b=2**

Restful

```
// 使用@Controller注解，将该类注册为Controller层，交由Spring的IOC容器统一管理
@Controller
public class RestfulController {

     /**
      * 使用RestFul风格进行访问
      * 使用@RequestMapping注解，设置请求映射的访问路径
      * 其真实访问路径为：http://localhost:8888/add/a/b
      * 而使用默认方式的访问路径则为：http://localhost:8888/add?a=1&b=2
      */
    @RequestMapping("/add/{a}/{b}")
    // 使用@PathVariable注解，让方法参数的值对应绑定到一个URL模板变量上
    public String test(@PathVariable int a, @PathVariable int b, Model model) {
```

```
        // 设置一个结果
        int result = a + b;
        // 封装数据：向模型中添加属性msg与其值，进行视图渲染
        model.addAttribute("msg","结果为"+result);
        // 返回视图逻辑名，交由视图解析器进行处理
        return "user/hello";
    }

     /**
     * 使用@RequestMapping注解，设置映射请求路径
     * 使用"{}"将参数a和b括起来，参数之间使用"/"分开
     */
    @RequestMapping("/add/{a}/{b}")
    // 使用@PathVariable注解，将方法值对应绑定到URL模板变量上
    public String test2(@PathVariable String a, @PathVariable String b,Model
model) {
        // 设置一个结果
        String result = a + b;
        // 封装数据：向模型中添加属性msg与其值，进行视图渲染
        model.addAttribute("msg","结果为"+result);
        // 返回视图逻辑名，交由视图解析器进行处理
        return "user/hello";
    }
}
```

- **使用默认方式的访问地址为：http://localhost:8888/add?a=1&b=2，这种方式虽然看起来比较清楚，但容易暴露源代码中变量信息，安全性不够高**
- **使用RestFul风格的访问地址为：http://localhost:8888/add/a/b，这种方式比较简洁高效，虽然只看URL链接不清楚在干嘛，但是安全性却相对较高**

4.4 RequestMapping 源码

```
package org.springframework.web.bind.annotation;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.springframework.core.annotation.AliasFor;

/**
 * 该注解用于将web请求映射到具有灵活方法签名的请求处理类中的方法
 *
 * 注意：该注解可以使用在类和方法级别上.
 * 在大多数情况，在方法级别，应用程序更喜欢使用特定于HTTP方法的变体之一：
 * @GetMapping，@PostMapping，@PutMapping，@DeleteMapping，@PatchMapping.
 *
 * 注意：若使用controller(控制器)接口(例如AOP动态代理)，确保持你所有的映射注解都是一致的：
 * 例如 @RequestMapping(请求映射)和@SessionAttributes(会话属性)在控制器接口上，而不是实
现类上
 * @see GetMapping
 * @see PostMapping
 * @see PutMapping
```

```
 * @see DeleteMapping
 * @see PatchMapping
 */
// 使用@Target注解，设置作用目标，元素类型为类和方法上
@Target({ElementType.TYPE, ElementType.METHOD})
// 使用@Retention注解，设置保留政策为运行时策略
@Retention(RetentionPolicy.RUNTIME)
// 使用@Documented注解，该注解将由javadoc和类型工具记录，并且成为注解元素公共API的一部分
@Documented
// 使用@Mapping注解，表示该注解为Web映射注解
@Mapping
public @interface RequestMapping {

    /**
     * 主要作用：给这个映射分配一个名字，它支持在类级别和方法级别使用
     * 若使用两种级别，派生的组合名将使用"#"分隔符连接
     * @see
org.springframework.web.servlet.mvc.method.annotation.MvcUriComponentsBuilder(Mvc
的Uri组件构筑器)
     * @see
org.springframework.web.servlet.handler.HandlerMethodMappingNamingStrategy(处理器
方法映射命名策略)
     */
    String name() default "";

    /**
     * 主映射由该注解表达，这是path(路径)的别名，同样支持类级别和方法级别
     * 例如，使用@RequestMapping("/foo")等同于@RequestMapping(path="/foo")
     * 若使用类级别，所有的方法级别映射包含该主映射，为特定的处理器方法缩小范围
     * 注意：若一个处理器方法没有显式映射到任何路径，它将被有效地映射到一个空路径
     */
    @AliasFor("path")
    String[] value() default {};

    /**
     * 路径映射URI(例如"/profile")，也支持"Ant-style"路径格式(例如"/profile/**")
     * 在方法级别，在类级别表示的主映射中支持相对路径(例如"edit")，路径映射URIs包含占位符(例
如"/${profile_path}")，支持类级别和方法级别
     * 若使用类级别，所有的方法级别映射包含该主映射，为指定的处理器方法缩小范围
     * 注意：若一个处理器方法没有显式映射到任何路径，它将被有效地映射到一个空路径
     * @since 4.2
     */
    @AliasFor("value")
    String[] path() default {};

    /**
     * HTTP请求要方法映射到的方法，缩小主映射：
     * GET(查询)，POST(增加)，HEAD，OPTIONS，PUT(修改)，PATCH，DELETE(删除)，TRACE
     * 支持在类型级别和方法级别使用
     * 若使用类型级别，所有的方法级别映射包含这个HTTP方法限制
     */
    RequestMethod[] method() default {};

    /**
     * 映射请求的参数，缩小主映射范围
     */
    String[] params() default {};
```

```
    /**
     * 映射请求的头部，缩小主映射范围
     * @see org.springframework.http.MediaType(媒介类型)
     */
    String[] headers() default {};


    /**
     * 根据映射处理程序可以使用的媒体类型缩小主映射
     * 由一个或者多个媒介类型组成，其中一个必须与请求"Content-Type"(内容类型)头相匹配
     * @see org.springframework.http.MediaType(媒介类型)
     * @see javax.servlet.http.HttpServletRequest(HttpServlet的请
求)#getContentType()(获取内容类型)
     */
    String[] consumes() default {};

    /**
     * 通过映射处理器可以生成媒介类型来缩小主映射.
     * 由一个或多个媒介类型组成，必须根据请求的"可接受"媒介类型通过内容协商选择其中之一
     * @see org.springframework.http.MediaType(媒介类型)
     */
    String[] produces() default {};

}
```

使用method属性来使用RequestMapping

- "name"属性：为该映射起一个名字，而并不代表该映射的具体路径
- "value"属性：表示该映射的具体路径，但不清楚在哪些情况下指定路径
- "path"属性：也表示该映射的具体路径，作用与value相同

```
// 使用@Controller注解，将该类注册为Controller层，交由Spring的IOC容器统一管理
@Controller
public class RestfulController {

    // 使用@RequestMapping注解，设置映射请求路径
     @RequestMapping(value = "/add/{a}/{b}",method = RequestMethod.GET)
    // 使用@PathVariable注解，将方法值对应绑定到URL模板变量上
    public String test4(@PathVariable int a, @PathVariable String b,Model model)
{
        // 设置一个结果
        String result = a + b;
        // 封装数据：向模型中添加属性msg与其值，进行视图渲染
        model.addAttribute("msg","结果1为"+result);
        // 返回视图逻辑名，交由视图解析器进行处理
        return "user/hello";
    }

}
```

- SpringMVC的@RequstMapping注解能够**请求处理HTTP请求的方法**，比如GET、PUT、POST/DELETE以及PATCH
- **所有的地址栏请求默认都是HTTP GET类型的**

4.5 方法注解

```
@GetMapping: 对应get请求，查询操作
@PostMapping: 对应post请求，添加操作
@PutMapping: 对应put请求，修改操作
@DeleteMapping: 对应delete请求，删除操作
@PatchMapping: 对应patct请求，多条数据修改和删除操作
```

@GetMapping是一个组合注解，相当于@RequstMapping(method=ReustMethod.GET)，平常使用的会比较多

```
// 使用@PostMapping注解，设置映射请求路径，请求方式为post请求
@PostMapping("/add/{a}/{b}")
// 使用@PathVariable注解，让方法值对应绑定到URL模板变量上
```

如果同时使用了@GetMapping注解和RequestMapping注解， 会出现模棱两可的映射问题

- 同时使用@RequestMapping和@PostMapping注解，控制器调用的是@RequestMapping注解的方法
- 地址栏请求的默认方式是GET请求