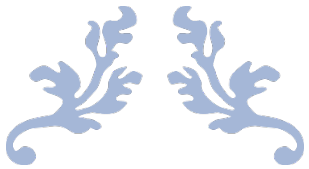


SpringMVC 教程

江南一点雨 · 编著

公众号·江南一点雨

2019-11-1

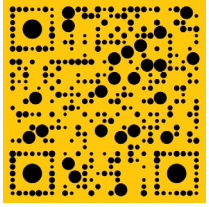
江南一点雨
http://springmvc.javaboy.org

公众号·江南一点雨

扫码关注微信公众号【江南一点雨】，回复 2TB，获取超 2TB 免费 Java 学习资源

公众号·江南一点雨

扫码关注微信公众号【江南一点雨】，回复 2TB，获取超 2TB 免费 Java 学习资源



1. SpringMVC 简介

1.1 Spring Web MVC 是什么

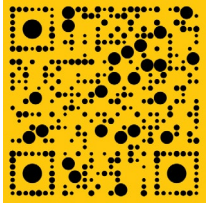
Spring Web MVC 是一种基于 Java 的实现了 Web MVC 设计模式的请求驱动类型的轻量级 Web 框架，即使用了 MVC 架构模式的思想，将 web 层进行职责解耦，基于请求驱动指的就是使用请求-响应模型，框架的目的就是帮助我们简化开发，Spring Web MVC 也是要简化我们日常 Web 开发的。在传统的 Jsp/Servlet 技术体系中，如果要开发接口，一个接口对应一个 Servlet，会导致我们开发出许多 Servlet，使用 SpringMVC 可以有效的简化这一步骤。

Spring Web MVC 也是服务到工作者模式的实现，但进行可优化。前端控制器是 DispatcherServlet；应用控制器可以拆为处理器映射器(Handler Mapping)进行处理器管理和视图解析器(View Resolver)进行视图管理；页面控制器/动作/处理器为 Controller 接口（仅包含 ModelAndView handleRequest(request, response) 方法，也有人称作 Handler）的实现（也可以是任何的 POJO 类）；支持本地化（Locale）解析、主题（Theme）解析及文件上传等；提供了非常灵活的数据验证、格式化和数据绑定机制；提供了强大的约定大于配置（惯例优先原则）的契约式编程支持。

1.2 Spring Web MVC 能帮我们做什么

- 让我们能非常简单的设计出干净的 Web 层和薄薄的 Web 层；
- 进行更简洁的 Web 层的开发；
- 天生与 Spring 框架集成（如 IoC 容器、AOP 等）；
- 提供强大的约定大于配置的契约式编程支持；
- 能简单的进行 Web 层的单元测试；

扫码关注微信公众号【江南一点雨】，回复 2TB，获取超 2TB 免费 Java 学习资源



- 支持灵活的 URL 到页面控制器的映射;
- 非常容易与其他视图技术集成，如 Velocity、FreeMarker 等等，因为模型数据不放在特定的 API 里，而是放在一个 Model 里（Map 数据结构实现，因此很容易被其他框架使用);
- 非常灵活的数据验证、格式化和数据绑定机制，能使用任何对象进行数据绑定，不必实现特定框架的 API;
- 提供一套强大的 JSP 标签库，简化 JSP 开发;
- 支持灵活的本地化、主题等解析;
- 更加简单的异常处理;
- 对静态资源的支持;
- 支持 RESTful 风格

2. HelloWorld

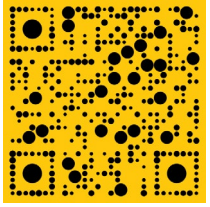
接下来，通过一个简单的例子来感受一下 SpringMVC。

1.利用 Maven 创建一个 web 工程（参考 Maven 教程）。 2.在 pom.xml 文件中，添加 spring-webmvc 的依赖:

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>RELEASE</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>4.0.1</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet.jsp</groupId>
        <artifactId>javax.servlet.jsp-api</artifactId>
        <version>2.3.3</version>
    </dependency>
</dependencies>
```

公众号·江南一点雨

扫码关注微信公众号【江南一点雨】，回复 2TB，获取超 2TB 免费 Java 学习资源



添加了 spring-webmvc 依赖之后，其他的 spring-web、spring-aop、spring-context 等等就全部都加入进来了。

3.准备一个 Controller，即一个处理浏览器请求的接口。

```
public class MyController implements Controller {
    /**
     * 这就是一个请求处理接口
     * @param req 这就是前端发送来的请求
     * @param resp 这就是服务端给前端的响应
     * @return 返回值是一个 ModelAndView，Model 相当于是我们的数据模型，
View 是我们的视图
     * @throws Exception
     */
    public ModelAndView handleRequest(HttpServletRequest req, HttpSer
vletResponse resp) throws Exception {
        ModelAndView mv = new ModelAndView("hello");
        mv.addObject("name", "javaboy");
        return mv;
    }
}
```

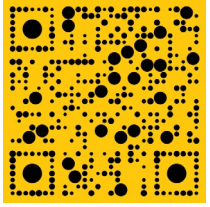
这里我们我们创建出来的 Controller 就是前端请求处理接口。

4.创建视图

这里我们就采用 jsp 作为视图，在 webapp 目录下创建 hello.jsp 文件，内容如下：

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
<h1>hello ${name}!</h1>
</body>
</html>
```

公众号·江南一点雨



5.在 resources 目录下，创建一个名为 spring-servlet.xml 的 springmvc 的配置文件，这里，我们先写一个简单的 demo ，因此可以先不用添加 spring 的配置。

```
<?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/bean
s http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean class="org.javaboy.helloworld.MyController" name="/hello"/>
    <!--这个是处理器映射器，这种方式，请求地址其实就是一个 Bean 的名字，然
后根据这个 bean 的名字查找对应的处理器-->
    <bean class="org.springframework.web.servlet.handler.BeanNameUrlH
andlerMapping" id="handlerMapping">
        <property name="beanName" value="/hello"/>
    </bean>
    <bean class="org.springframework.web.servlet.mvc.SimpleController
HandlerAdapter" id="handlerAdapter"/>

    <!--视图解析器-->
    <bean class="org.springframework.web.servlet.view.InternalResourc
eViewResolver" id="viewResolver">
        <property name="prefix" value="/jsp/"/>
        <property name="suffix" value=".jsp"/>
    </bean>
</beans>
```

6.加载 springmvc 配置文件

在 web 项目启动时，加载 springmvc 配置文件，这个配置是在 web.xml 中完成的。

```
<?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>
```

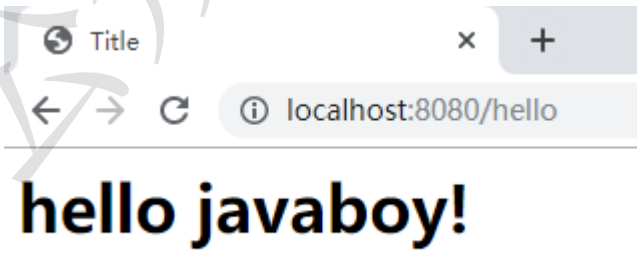
扫码关注微信公众号【江南一点雨】，回复 2TB，获取超 2TB 免费 Java 学习资源

```
        <servlet-name>springmvc</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServ
let</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:spring-servlet.xml</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>springmvc</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

所有请求都将自动拦截下来，拦截下来后，请求交给 DispatcherServlet 去处理，在加载 DispatcherServlet 时，还需要指定配置文件路径。这里有一个默认的规则，如果配置文件放在 webapp/WEB-INF/ 目录下，并且配置文件的名字等于 DispatcherServlet 的名字+ -servlet（即这里的配置文件路径是 webapp/WEB-INF/springmvc-servlet.xml），如果是这样的话，可以不用添加 init-param 参数，即不用手动配置 springmvc 的配置文件，框架会自动加载。

7.配置并启动项目（参考 Maven 教程）

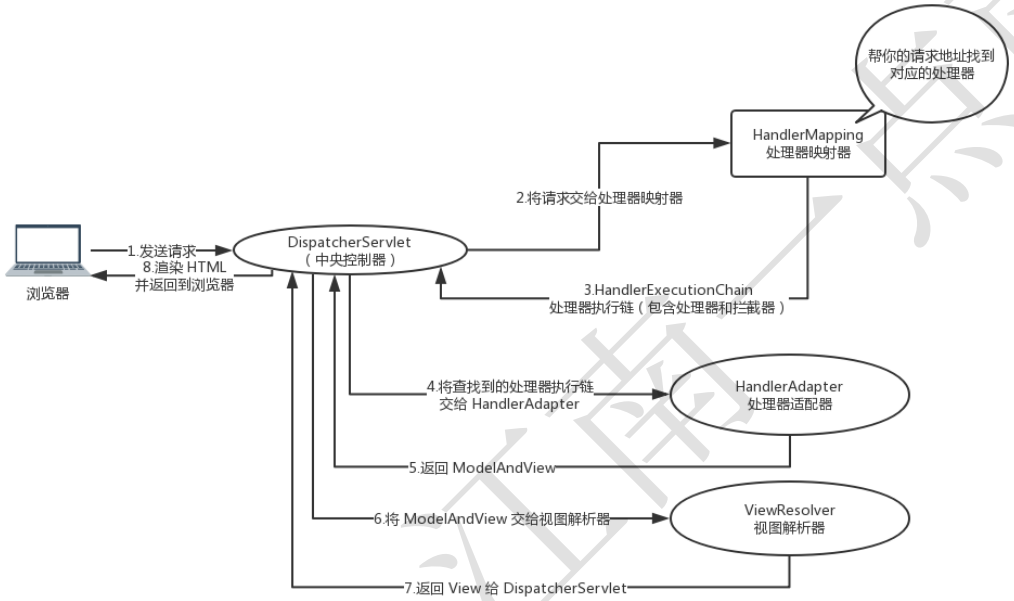
8.项目启动成功后，浏览器输入 http://localhost:8080/hello 就可以看到如下页面：



扫码关注微信公众号【江南一点雨】，回复 2TB，获取超 2TB 免费 Java 学习资源

3. SpringMVC 工作流程

面试时，关于 SpringMVC 的问题，超过 99% 都是这个问题。



4. SpringMVC 中的组件

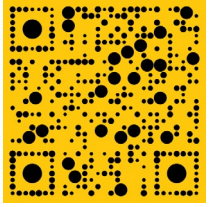
1.DispatcherServlet：前端控制器

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

2.HandlerMapping：处理器映射器

HandlerMapping 负责根据用户请求找到 Handler 即处理器（也就是我们所说的 Controller），SpringMVC 提供了不同的映射器实现不同的映射方式，例

扫码关注微信公众号【江南一点雨】，回复 2TB，获取超 2TB 免费 Java 学习资源



如：配置文件方式，实现接口方式，注解方式等，在实际开发中，我们常用的方式是注解方式。

3.Handler：处理器

Handler 是继 DispatcherServlet 前端控制器的后端控制器，在 DispatcherServlet 的控制下 Handler 对具体的用户请求进行处理。由于 Handler 涉及到具体的用户业务请求，所以一般情况需要程序员根据业务需求开发 Handler。（这里所说的 Handler 就是指我们的 Controller）

4.HandlAdapter：处理器适配器

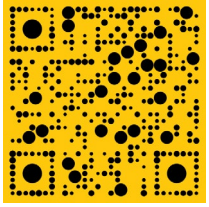
通过 HandlerAdapter 对处理器进行执行，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。

5.ViewResolver：视图解析器

ViewResolver 负责将处理结果生成 View 视图，ViewResolver 首先根据逻辑视图名解析成物理视图名即具体的页面地址，再生成 View 视图对象，最后对 View 进行渲染将处理结果通过页面展示给用户。 SpringMVC 框架提供了很多的 View 视图类型，包括：jstlView、freemarkerView、pdfView 等。一般情况下需要通过页面标签或页面模版技术将模型数据通过页面展示给用户，需要由程序员根据业务需求开发具体的页面。

公众号·江南一点雨

扫码关注微信公众号【江南一点雨】，回复 2TB，获取超 2TB 免费 Java 学习资源



5. DispatcherServlet

5.1 DispatcherServlet 作用

DispatcherServlet 是前端控制器设计模式的实现，提供 Spring Web MVC 的集中访问点，而且负责职责的分派，而且与 Spring IoC 容器无缝集成，从而可以获得 Spring 的所有好处。DispatcherServlet 主要用作职责调度工作，本身主要用于控制流程，主要职责如下：

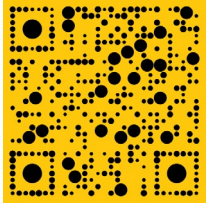
1. 文件上传解析，如果请求类型是 multipart 将通过 MultipartResolver 进行文件上传解析；
2. 通过 HandlerMapping，将请求映射到处理器（返回一个 HandlerExecutionChain，它包括一个处理器、多个 HandlerInterceptor 拦截器）；
3. 通过 HandlerAdapter 支持多种类型的处理器(HandlerExecutionChain 中的处理器)；
4. 通过 ViewResolver 解析逻辑视图名到具体视图实现；
5. 本地化解析；
6. 渲染具体的视图等；
7. 如果执行过程中遇到异常将交给 HandlerExceptionResolver 来解析

5.2 DispathcherServlet 配置详解

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

公众号·江南一点雨

扫码关注微信公众号【江南一点雨】，回复 2TB，获取超 2TB 免费 Java 学习资源



- load-on-startup：表示启动容器时初始化该 Servlet；
- url-pattern：表示哪些请求交给 Spring Web MVC 处理，"/" 是用来定义默认 servlet 映射的。也可以如 `*.html` 表示拦截所有以 html 为扩展名的请求
- contextConfigLocation：表示 SpringMVC 配置文件的路径

其他的参数配置：

参数	描述
contextClass	实现 WebApplicationContext 接口的类，当前的 servlet 用它来创建上下文。如果这个参数没有指定，默认使用 XmlWebApplicationContext。
contextConfigLocation	传给上下文实例（由 contextClass 指定）的字符串，用来指定上下文的位置。这个字符串可以被分成多个字符串（使用逗号作为分隔符）来支持多个上下文（在多上下文的情况下，如果同一个 bean 被定义两次，后面一个优先）。
namespace	WebApplicationContext 命名空间。默认值是[server-name]-servlet。

5.3 Spring 配置

之前的案例中，只有 SpringMVC，没有 Spring，Web 项目也是可以运行的。

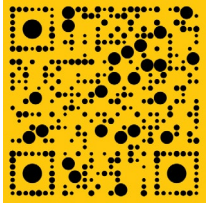
在实际开发中，Spring 和 SpringMVC 是分开配置的，所以我们对上面的项目继续进行完善，添加 Spring 相关配置。

首先，项目添加一个 service 包，提供一个 HelloService 类，如下：

```
@Service
public class HelloService {
    public String hello(String name) {
        return "hello " + name;
    }
}
```

现在，假设我需要将 HelloService 注入到 Spring 容器中并使用它，这个是属于 Spring 层的 Bean，所以我们一般将除了 Controller 之外的所有 Bean 注

扫码关注微信公众号【江南一点雨】，回复 2TB，获取超 2TB 免费 Java 学习资源



册到 Spring 容器中，而将 Controller 注册到 SpringMVC 容器中，现在，在 resources 目录下添加 applicationContext.xml 作为 spring 的配置：

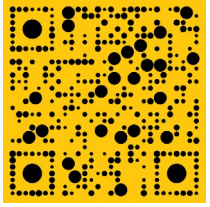
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/bean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

    <context:component-scan base-package="org.javaboy" use-default-fi
lters="true">
        <context:exclude-filter type="annotation" expression="org.spr
ingframework.stereotype.Controller"/>
    </context:component-scan>
</beans>
```

但是，这个配置文件，默认情况下，并不会被自动加载，所以，需要我们在 web.xml 中对其进行配置：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext.xml</param-value>
</context-param>
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderList
ener</listener-class>
</listener>
```

首先通过 context-param 指定 Spring 配置文件的位置，这个配置文件也有一些默认规则，它的配置文件名默认就叫 applicationContext.xml ，并且，如果你将这个配置文件放在 WEB-INF 目录下，那么这里就可以不用指定配置文件位置了，只需要指定监听器就可以了。这段配置是 Spring 集成 Web 环境的

公众号·江南一点雨



通用配置；一般用于加载除 Web 层的 Bean（如 DAO、Service 等），以便于与其他任何 Web 框架集成。

- contextConfigLocation：表示用于加载 Bean 的配置文件；
- contextClass：表示用于加载 Bean 的 ApplicationContext 实现类，默认 WebApplicationContext。

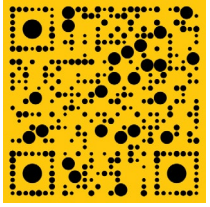
配置完成之后，还需要修改 MyController，在 MyController 中注入 HelloSerivce:

```
@org.springframework.stereotype.Controller("/hello")
public class MyController implements Controller {
    @Autowired
    HelloService helloService;
    /**
     * 这就是一个请求处理接口
     * @param req 这就是前端发送来的请求
     * @param resp 这就是服务端给前端的响应
     * @return 返回值是一个 ModelAndView，Model 相当于是我们的数据模型，
View 是我们的视图
     * @throws Exception
     */
    public ModelAndView handleRequest(HttpServletRequest req, HttpSer
vletResponse resp) throws Exception {
        System.out.println(helloService.hello("javaboy"));
        ModelAndView mv = new ModelAndView("hello");
        mv.addObject("name", "javaboy");
        return mv;
    }
}
```

注意

为了在 SpringMVC 容器中能够扫描到 MyController，这里给 MyController 添加了 @Controller 注解，同时，由于我们目前采用的 HandlerMapping 是 BeanNameUrlHandlerMapping（意味着请求地址就是处理器 Bean 的名字），所以，还需要手动指定 MyController 的名字。

公众号·江南一点雨



最后，修改 SpringMVC 的配置文件，将 Bean 配置为扫描形式：

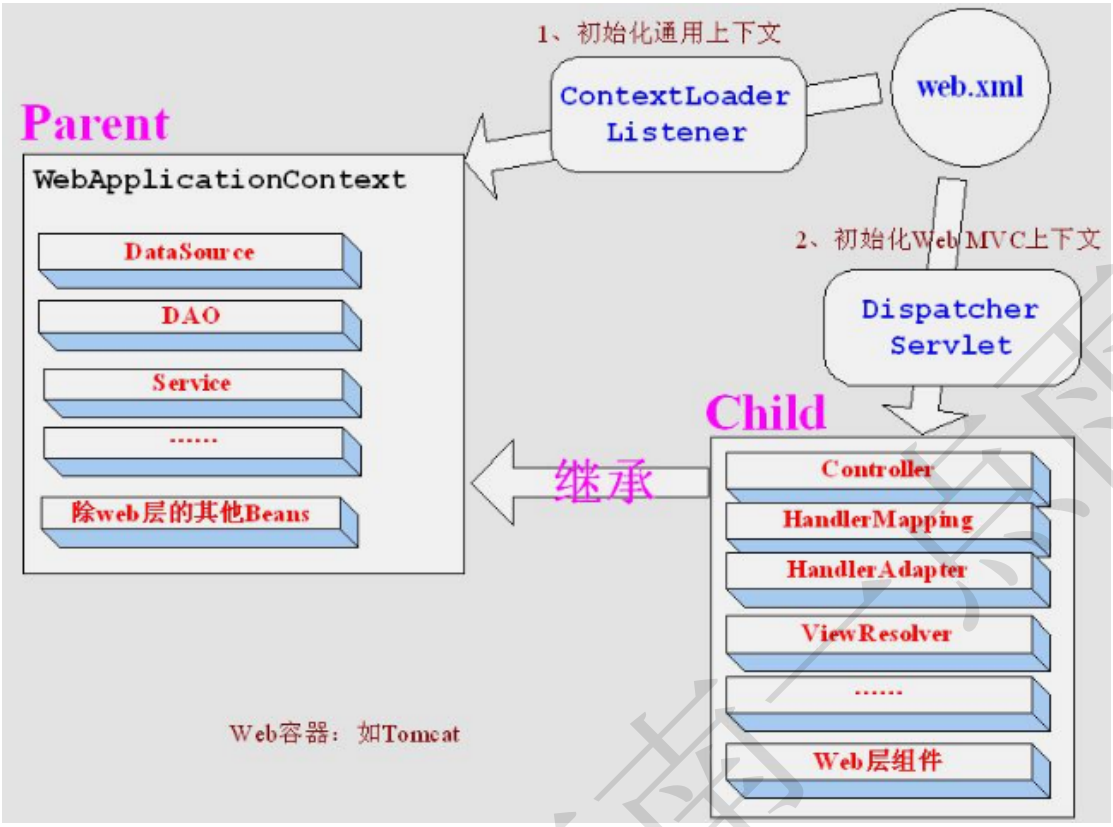
```
<context:component-scan base-package="org.javaboy.helloworld" use-de
fault-filters="false">
    <context:include-filter type="annotation" expression="org.springf
ramework.stereotype.Controller"/>
</context:component-scan>
<!--这个是处理器映射器，这种方式，请求地址其实就是一个 Bean 的名字，然后根
据这个 bean 的名字查找对应的处理器-->
<bean class="org.springframework.web.servlet.handler.BeanNameUrlHand
lerMapping" id="handlerMapping">
    <property name="beanName" value="/hello"/>
</bean>
<bean class="org.springframework.web.servlet.mvc.SimpleControllerHan
dlerAdapter" id="handlerAdapter"/>
<!--视图解析器-->
<bean class="org.springframework.web.servlet.view.InternalResourceVi
ewResolver" id="viewResolver">
    <property name="prefix" value="/jsp/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

配置完成后，再次启动项目，Spring 容器也将会被创建。访问 /hello 接口，HelloService 中的 hello 方法就会自动被调用。

5.4 两个容器

当 Spring 和 SpringMVC 同时出现，我们的项目中将存在两个容器，一个是 Spring 容器，另一个是 SpringMVC 容器，Spring 容器通过 ContextLoaderListener 来加载，SpringMVC 容器则通过 DispatcherServlet 来加载，这两个容器不一样：

公众号：江南一点雨



从图中可以看出：

- ContextLoaderListener 初始化的上下文加载的 Bean 是对于整个应用程序共享的，不管是使用什么表现层技术，一般如 DAO 层、Service 层 Bean；
- DispatcherServlet 初始化的上下文加载的 Bean 是只对 Spring Web MVC 有效的 Bean，如 Controller、HandlerMapping、HandlerAdapter 等等，该初始化上下文应该只加载 Web 相关组件。

1. 为什么不在 Spring 容器中扫描所有 Bean?

这个是不可能的。因为请求达到服务端后，找 DispatcherServlet 去处理，只会去 SpringMVC 容器中找，这就意味着 Controller 必须在 SpringMVC 容器中扫描。

2.为什么不在 SpringMVC 容器中扫描所有 Bean?

这个是可以的，可以在 SpringMVC 容器中扫描所有 Bean。不写在一起，有两个方面的原因：

扫码关注微信公众号【江南一点雨】，回复 2TB，获取超 2TB 免费 Java 学习资源

1. 为了方便配置文件的管理

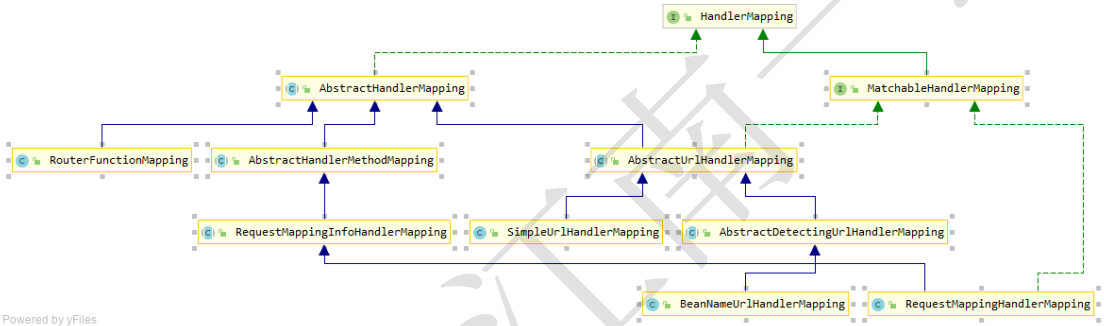
2. 在 Spring+SpringMVC+Hibernate 组合中，实际上也不支持这种写法

6. 处理器详解

6.1 HandlerMapping

注意，下文所说的处理器即我们平时所见到的 Controller

HandlerMapping ，中文译作处理器映射器，在 SpringMVC 中，系统提供了很多 HandlerMapping：



HandlerMapping 是负责根据 request 请求找到对应的 Handler 处理器及 Interceptor 拦截器，将它们封装在 HandlerExecutionChain 对象中返回给前端控制器。

- BeanNameUrlHandlerMapping

BeanNameUrl 处理器映射器，根据请求的 url 与 Spring 容器中定义的 bean 的 name 进行匹配，从而从 Spring 容器中找到 bean 实例，就是说，请求的 Url 地址就是处理器 Bean 的名字。

这个 HandlerMapping 配置如下：

```
<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping" id="handlerMapping">
```

扫码关注微信公众号【江南一点雨】，回复 2TB，获取超 2TB 免费 Java 学习资源

```
    <property name="beanName" value="/hello"/>
</bean>
```

- SimpleUrlHandlerMapping

SimpleUrlHandlerMapping 是 BeanNameUrlHandlerMapping 的增强版本，它可以将 url 和处理器 bean 的 id 进行统一映射配置:

```
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandle
rMapping" id="handlerMapping">
    <property name="mappings">
        <props>
            <prop key="/hello">myController</prop>
            <prop key="/hello2">myController2</prop>
        </props>
    </property>
</bean>
```

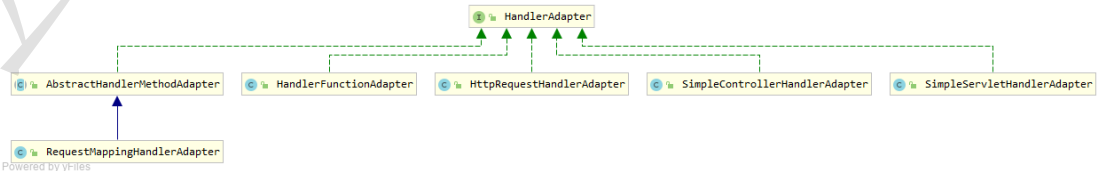
注意，在 props 中，可以配置多个请求路径和处理器实例的映射关系。

6.2 HandlerAdapter

HandlerAdapter，中文译作处理器适配器。

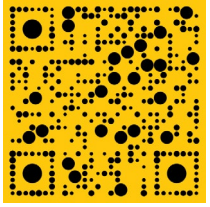
HandlerAdapter 会根据适配器接口对后端控制器进行包装（适配），包装后即可对处理器进行执行，通过扩展处理器适配器可以执行多种类型的处理器，这里使用了适配器设计模式。

在 SpringMVC 中，HandlerAdapter 也有诸多实现类:



- SimpleControllerHandlerAdapter

扫码关注微信公众号【江南一点雨】，回复 2TB，获取超 2TB 免费 Java 学习资源



SimpleControllerHandlerAdapter 简单控制器处理器适配器，所有实现了 org.springframework.web.servlet.mvc.Controller 接口的 Bean 通过此适配器进行适配、执行，也就是说，如果我们开发的接口是通过实现 Controller 接口来完成的（不是通过注解开发的接口），那么 HandlerAdapter 必须是 SimpleControllerHandlerAdapter。

```
<bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter" />
```

- HttpRequestHandlerAdapter

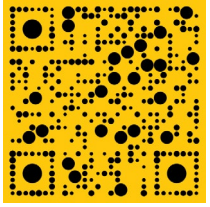
HttpRequestHandlerAdapter，http 请求处理器适配器，所有实现了 org.springframework.web.HttpRequestHandler 接口的 Bean 通过此适配器进行适配、执行。

例如存在如下接口：

```
@Controller
public class MyController2 implements HttpRequestHandler {
    public void handleRequest(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        System.out.println("-----MyController2-----");
    }
}
```

```
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping" id="handlerMapping">
    <property name="mappings">
        <props>
            <prop key="/hello2">myController2</prop>
        </props>
    </property>
</bean>
<bean class="org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter" id="handlerAdapter"/>
```

公众号·江南一点雨



6.3 最佳实践

各种情况都大概了解了，我们看下项目中的具体实践。

- 组件自动扫描

web 开发中，我们基本上不再通过 XML 或者 Java 配置来创建一个 Bean 的实例，而是直接通过组件扫描来实现 Bean 的配置，如果要扫描多个包，多个包之间用 ，隔开即可：

```
<context:component-scan base-package="org.sang"/>
```

- HandlerMapping

正常情况下，我们在项目中使用的是 RequestMappingHandlerMapping，这个是根据处理器中的注解，来匹配请求（即 @RequestMapping 注解中的 url 属性）。因为在上面我们都是通过实现类来开发接口的，相当于还是一个类一个接口，所以，我们可以通过 RequestMappingHandlerMapping 来做处理器映射器，这样我们可以在一个类中开发出多个接口。

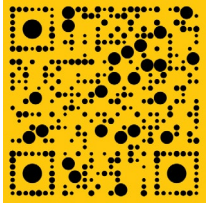
- HandlerAdapter

对于上面提到的通过 @RequestMapping 注解所定义出来的接口方法，这些方法的调用都是要通过 RequestMappingHandlerAdapter 这个适配器来实现。

例如我们开发一个接口：

```
@Controller
public class MyController3 {
    @RequestMapping("/hello3")
    public ModelAndView hello() {
        return new ModelAndView("hello3");
    }
}
```

公众号·江南一点雨



要能够访问到这个接口，我们需要 RequestMappingHandlerMapping 才能定位到需要执行的方法，需要 RequestMappingHandlerAdapter，才能执行定位到的方法，修改 springmvc 的配置文件如下：

```
<?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/bean
s http://www.springframework.org/schema/beans/spring-beans.xsd http:
//www.springframework.org/schema/context https://www.springframewor
k.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="org.javaboy.helloworld"/>

    <bean class="org.springframework.web.servlet.mvc.method.annotatio
n.RequestMappingHandlerMapping" id="handlerMapping"/>
    <bean class="org.springframework.web.servlet.mvc.method.annotatio
n.RequestMappingHandlerAdapter" id="handlerAdapter"/>
    <!--视图解析器-->
    <bean class="org.springframework.web.servlet.view.InternalResourc
eViewResolver" id="viewResolver">
        <property name="prefix" value="/jsp/"/>
        <property name="suffix" value=".jsp"/>
    </bean>
</beans>
```

然后，启动项目，访问 /hello3 接口，就可以看到相应的页面了。

- 继续优化

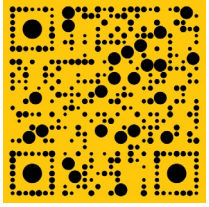
由于开发中，我们常用的是 RequestMappingHandlerMapping 和 RequestMappingHandlerAdapter ，这两个有一个简化的写法，如下：

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

可以用这一行配置，代替 RequestMappingHandlerMapping 和 RequestMappingHandlerAdapter 的两行配置。

公众号·江南一点雨

扫码关注微信公众号【江南一点雨】，回复 2TB，获取超 2TB 免费 Java 学习资源



```
<?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/bean
s http://www.springframework.org/schema/beans/spring-beans.xsd http:
//www.springframework.org/schema/context https://www.springframewor
k.org/schema/context/spring-context.xsd http://www.springframework.o
rg/schema/mvc https://www.springframework.org/schema/mvc/spring-mvc.
xsd">

    <context:component-scan base-package="org.javaboy.helloworld"/>

    <mvc:annotation-driven/>
    <!--视图解析器-->
    <bean class="org.springframework.web.servlet.view.InternalResourc
eViewResolver" id="viewResolver">
        <property name="prefix" value="/jsp/"/>
        <property name="suffix" value=".jsp"/>
    </bean>
</beans>
```

访问效果和上一步的效果一样。这是我们实际开发中，最终配置的形态。

7.1 @RequestMapping

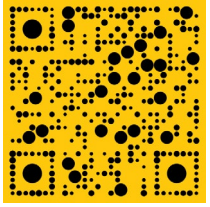
这个注解用来标记一个接口，这算是我们在接口开发中，使用最多的注解之一。

7.1.1 请求 URL

标记请求 URL 很简单，只需要在相应的方法上添加该注解即可：

```
@Controller
public class HelloController {
    @RequestMapping("/hello")
    public ModelAndView hello() {
        return new ModelAndView("hello");
```

公众号·江南一点雨



```
    }
}
```

这里 @RequestMapping("/hello") 表示当请求地址为 /hello 的时候，这个方法会被触发。其中，地址可以是多个，就是可以多个地址映射到同一个方法。

```
@Controller
public class HelloController {
    @RequestMapping({"/hello","/hello2"})
    public ModelAndView hello() {
        return new ModelAndView("hello");
    }
}
```

这个配置，表示 /hello 和 /hello2 都可以访问到该方法。

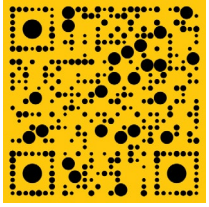
7.1.2 请求窄化

同一个项目中，会存在多个接口，例如订单相关的接口都是 /order/xxx 格式的，用户相关的接口都是 /user/xxx 格式的。为了方便处理，这里的前缀（就是 /order、/user）可以统一在 Controller 上面处理。

```
@Controller
@RequestMapping("/user")
public class HelloController {
    @RequestMapping({"/hello","/hello2"})
    public ModelAndView hello() {
        return new ModelAndView("hello");
    }
}
```

当类上加了 @RequestMapping 注解之后，此时，要想访问到 hello ，地址就应该是 /user/hello 或者 /user/hello2

公众号·江南一点雨



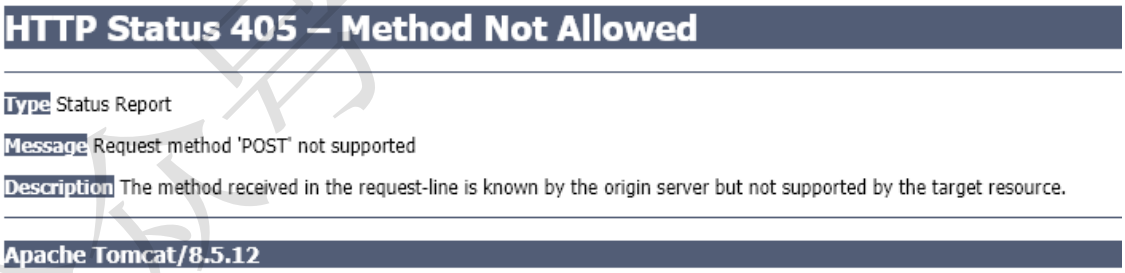
7.1.3 请求方法限定

默认情况下，使用 @RequestMapping 注解定义好的方法，可以被 GET 请求访问到，也可以被 POST 请求访问到，但是 DELETE 请求以及 PUT 请求不可以访问到。

当然，我们也可以指定具体的访问方法：

```
@Controller
@RequestMapping("/user")
public class HelloController {
    @RequestMapping(value = "/hello",method = RequestMethod.GET)
    public ModelAndView hello() {
        return new ModelAndView("hello");
    }
}
```

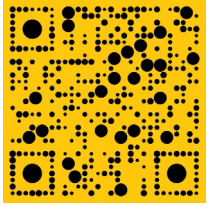
通过 @RequestMapping 注解，指定了该接口只能被 GET 请求访问到，此时，该接口就不可以被 POST 以及请求请求访问到了。强行访问会报如下错误：



当然，限定的方法也可以有多个：

```
@Controller
@RequestMapping("/user")
public class HelloController {
    @RequestMapping(value = "/hello",method = {RequestMethod.GET,Requ
estMethod.POST,RequestMethod.PUT,RequestMethod.DELETE})
    public ModelAndView hello() {
        return new ModelAndView("hello");
```

扫码关注微信公众号【江南一点雨】，回复 2TB，获取超 2TB 免费 Java 学习资源



```
    }
}
```

此时，这个接口就可以被 GET、POST、PUT、以及 DELETE 访问到了。但是，由于 JSP 支支持 GET、POST 以及 HEAD ，所以这个测试，不能使用 JSP 做页面模板。可以讲视图换成其他的，或者返回 JSON，这里就不影响了。

7.2 Controller 方法的返回值

7.2.1 返回 ModelAndView

如果是前后端不分的开发，大部分情况下，我们返回 ModelAndView，即数据模型+视图：

```
@Controller
@RequestMapping("/user")
public class HelloController {
    @RequestMapping("/hello")
    public ModelAndView hello() {
        ModelAndView mv = new ModelAndView("hello");
        mv.addObject("username", "javaboy");
        return mv;
    }
}
```

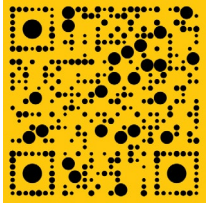
Model 中，放我们的数据，然后在 ModelAndView 中指定视图名称。

7.2.2 返回 Void

没有返回值。没有返回值，并不一定真的没有返回值，只是方法的返回值为 void，我们可以通过其他方式给前端返回。**实际上，这种方式也可以理解为 Servlet 中的那一套方案。**

注意，由于默认的 Maven 项目没有 Servlet，因此这里需要额外添加一个依赖：

扫码关注微信公众号【江南一点雨】，回复 2TB，获取超 2TB 免费 Java 学习资源



```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>4.0.1</version>
</dependency>
```

- 通过 HttpServletRequest 做服务端跳转

```
@RequestMapping("/hello2")
public void hello2(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
    req.getRequestDispatcher("/jsp/hello.jsp").forward(req,resp);//服
务器端跳转
}
```

- 通过 HttpServletResponse 做重定向

```
@RequestMapping("/hello3")
public void hello3(HttpServletRequest req, HttpServletResponse resp)
throws IOException {
    resp.sendRedirect("/hello.jsp");
}
```

也可以自己手动指定响应头去实现重定向：

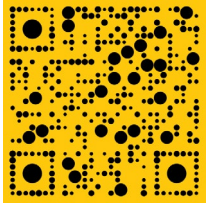
```
@RequestMapping("/hello3")
public void hello3(HttpServletRequest req, HttpServletResponse resp)
throws IOException {
    resp.setStatus(302);
    resp.addHeader("Location", "/jsp/hello.jsp");
}
```

- 通过 HttpServletResponse 给出响应

```
@RequestMapping("/hello4")
public void hello4(HttpServletRequest req, HttpServletResponse resp)
throws IOException {
    resp.setContentType("text/html;charset=utf-8");
    PrintWriter out = resp.getWriter();
    out.write("hello javaboy!");
    out.flush();
    out.close();
}
```

这种方式，既可以返回 JSON，也可以返回普通字符串。

公众号·江南一点雨



7.2.3 返回字符串

- 返回逻辑视图名

前面的 ModelAndView 可以拆分为两部分，Model 和 View，在 SpringMVC 中，Model 我们可以直接在参数中指定，然后返回值是逻辑视图名：

```
@RequestMapping("/hello5")
public String hello5(Model model) {
    model.addAttribute("username", "javaboy");//这是数据模型
    return "hello";//表示去查找一个名为 hello 的视图
}
```

- 服务端跳转

```
@RequestMapping("/hello5")
public String hello5() {
    return "forward:/jsp/hello.jsp";
}
```

forward 后面跟上跳转的路径。

- 客户端跳转

```
@RequestMapping("/hello5")
public String hello5() {
    return "redirect:/user/hello";
}
```

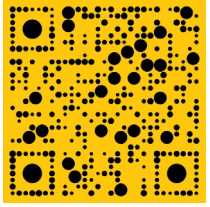
这种，本质上就是浏览器重定向。

- 真的返回一个字符串

上面三个返回的字符串，都是由特殊含义的，如果一定要返回一个字符串，需要额外添加一个注意：@ResponseBody ，这个注解表示当前方法的返回值就是要展示出来返回值，没有特殊含义。

```
@RequestMapping("/hello5")
@ResponseBody
public String hello5() {
```

公众号：江南一点雨



```
    return "redirect:/user/hello";
}
```

上面代码表示就是想返回一段内容为 redirect:/user/hello 的字符串，他没有特殊含义。注意，这里如果单纯的返回一个中文字符串，是会乱码的，可以在 @RequestMapping 中添加 produces 属性来解决：

```
@RequestMapping(value = "/hello5",produces = "text/html;charset=utf-8
")
@ResponseBody
public String hello5() {
    return "Java 语言程序设计";
}
```

7.3 参数绑定

7.3.1 默认支持的参数类型

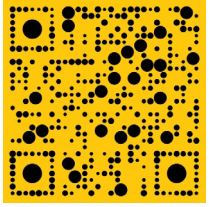
默认支持的参数类型，就是可以直接写在 @RequestMapping 所注解的方法中的参数类型，一共有四类：

- HttpServletRequest
- HttpServletResponse
- HttpSession
- Model/ModelMap

这几个例子可以参考上一小节。

在请求的方法中，默认的参数就是这几个，如果在方法中，刚好需要这几个参数，那么就可以把这几个参数加入到方法中。

公众号·江南一点雨



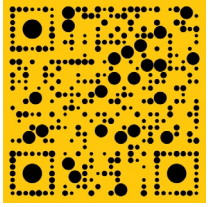
7.3.2 简单数据类型

Integer、Boolean、Double 等等简单数据类型也都是支持的。例如添加一本书：

首先，在 /jsp/ 目录下创建 add book.jsp 作为图书添加页面：

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
<form action="/doAdd" method="post">
    <table>
        <tr>
            <td>书名：</td>
            <td><input type="text" name="name"></td>
        </tr>
        <tr>
            <td>作者：</td>
            <td><input type="text" name="author"></td>
        </tr>
        <tr>
            <td>价格：</td>
            <td><input type="text" name="price"></td>
        </tr>
        <tr>
            <td>是否上架：</td>
            <td>
                <input type="radio" value="true" name="ispublic">是
                <input type="radio" value="false" name="ispublic">否
            </td>
        </tr>
        <tr>
           <td colspan="2">
               <input type="submit" value="添加">
           </td>
        </tr>
    </table>
</form>
```

公众号·江南一点雨



```
</body>
</html>
```

创建控制器，控制器提供两个功能，一个是访问 jsp 页面，另一个是提供添加接口：

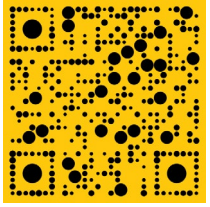
```
@Controller
public class BookController {
    @RequestMapping("/book")
    public String addBook() {
        return "addbook";
    }

    @RequestMapping(value = "/doAdd",method = RequestMethod.POST)
    @ResponseBody
    public void doAdd(String name,String author,Double price,Boolean ispublic) {
        System.out.println(name);
        System.out.println(author);
        System.out.println(price);
        System.out.println(ispublic);
    }
}
```

注意，由于 doAdd 方法确实不想返回任何值，所以需要给该方法添加 @ResponseBody 注解，表示这个方法到此为止，不用再去查找相关视图了。

另外， POST 请求传上来的中文会乱码，所以，我们在 web.xml 中再额外添加一个编码过滤器：

```
<filter>
    <filter-name>encoding</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>
    <init-param>
        <param-name>forceRequestEncoding</param-name>
```

扫码关注微信公众号【江南一点雨】，回复 2TB，获取超 2TB 免费 Java 学习资源



```
        <param-value>true</param-value>
    </init-param>
    <init-param>
        <param-name>forceResponseEncoding</param-name>
        <param-value>true</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>encoding</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

最后，浏览器中输入 http://localhost:8080/book ，就可以执行添加操作，服务端会打印出来相应的日志。

在上面的绑定中，有一个要求，表单中字段的 name 属性要和接口中的变量名一一对应，才能映射成功，否则服务端接收不到前端传来的数据。有一些特殊情况，我们的服务端的接口变量名可能和前端不一致，这个时候我们可以通过 @RequestParam 注解来解决。

- @RequestParam

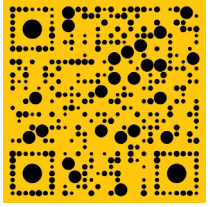
这个注解的的功能主要有三方面：

1. 给变量取别名
2. 设置变量是否必填
3. 给变量设置默认值

如下：

```
@RequestMapping(value = "/doAdd",method = RequestMethod.POST)
@ResponseBody
public void doAdd(@RequestParam("name") String bookname, String autho
r, Double price, Boolean ispublic) {
    System.out.println(bookname);
    System.out.println(author);
    System.out.println(price);
```

公众号·江南一点雨



```
    System.out.println(ispublic);
}
```

注解中的“name”表示给 bookname 这个变量取的别名，也就是说，bookname 将接收前端传来的 name 这个变量的值。在这个注解中，还可以添加 required 属性和 defaultValue 属性，如下：

```
@RequestMapping(value = "/doAdd",method = RequestMethod.POST)
@ResponseBody
public void doAdd(@RequestParam(value = "name",required = true,defaul
tValue = "三国演义") String bookname, String author, Double price, Boo
lean ispublic) {
    System.out.println(bookname);
    System.out.println(author);
    System.out.println(price);
    System.out.println(ispublic);
}
```

required 属性默认为 true，即只要添加了 @RequestParam 注解，这个参数默认就是必填的，如果不填，请求无法提交，会报 400 错误，如果这个参数不是必填项，可以手动把 required 属性设置为 false。但是，如果同时设置了 defaultValue，这个时候，前端不传该参数到后端，即使 required 属性为 true，它也不会报错。

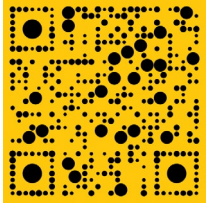
7.3.3 实体类

参数除了是简单数据类型之外，也可以是实体类。实际上，在开发中，大部分情况下，都是实体类。

还是上面的例子，我们改用一个 Book 对象来接收前端传来的数据：

```
public class Book {
    private String name;
    private String author;
    private Double price;
```

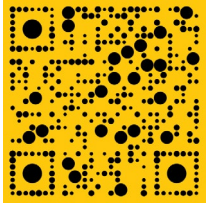
公众号：江南一点雨



```
    private Boolean ispublic;

    @Override
    public String toString() {
        return "Book{" +
                "name='" + name + '\'' +
                ", author='" + author + '\'' +
                ", price=" + price +
                ", ispublic=" + ispublic +
                '}';
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    public Double getPrice() {
        return price;
    }

    public void setPrice(Double price) {
        this.price = price;
    }

    public Boolean getIspublic() {
        return ispublic;
    }

    public void setIspublic(Boolean ispublic) {
        this.ispublic = ispublic;
    }
}
```

公众号·江南一点雨

扫码关注微信公众号【江南一点雨】，回复 2TB，获取超 2TB 免费 Java 学习资源



服务端接收数据方式如下：

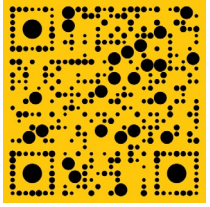
```
@RequestMapping(value = "/doAdd",method = RequestMethod.POST)
@ResponseBody
public void doAdd(Book book) {
    System.out.println(book);
}
```

前端页面传值的时候和上面的一样，只需要写属性名就可以了，不需要写 book 对象名。

当然，对象中可能还有对象。例如如下对象：

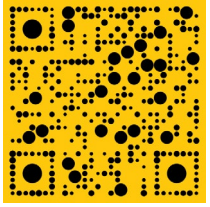
```
public class Book {
    private String name;
    private Double price;
    private Boolean ispublic;
    private Author author;

    public void setAuthor(Author author) {
        this.author = author;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Book{" +
                "name='" + name + '\'' +
                ", price=" + price +
                ", ispublic=" + ispublic +
                ", author=" + author +
                '}';
    }

    public Double getPrice() {
```

公众号·江南一点雨



```
        return price;
    }

    public void setPrice(Double price) {
        this.price = price;
    }

    public Boolean getIspublic() {
        return ispublic;
    }

    public void setIspublic(Boolean ispublic) {
        this.ispublic = ispublic;
    }
}
public class Author {
    private String name;
    private Integer age;

    @Override
    public String toString() {
        return "Author{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

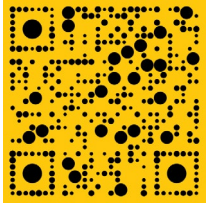
公众号·江南一点雨



Book 对象中，有一个 Author 属性，如何给 Author 属性传值呢？前端写法如下：

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
<form action="/doAdd" method="post">
    <table>
        <tr>
            <td>书名：</td>
            <td><input type="text" name="name"></td>
        </tr>
        <tr>
            <td>作者姓名：</td>
            <td><input type="text" name="author.name"></td>
        </tr>
        <tr>
            <td>作者年龄：</td>
            <td><input type="text" name="author.age"></td>
        </tr>
        <tr>
            <td>价格：</td>
            <td><input type="text" name="price"></td>
        </tr>
        <tr>
            <td>是否上架：</td>
            <td>
                <input type="radio" value="true" name="ispublic">是
                <input type="radio" value="false" name="ispublic">否
            </td>
        </tr>
        <tr>
           <td colspan="2">
               <input type="submit" value="添加">
           </td>
        </tr>
    </table>
</form>
```

公众号·江南一点雨



```
</body>
</html>
```

这样在后端直接用 Book 对象就可以接收到所有数据了。

7.3.4 自定义参数绑定

前面的转换，都是系统自动转换的，这种转换仅限于基本数据类型。特殊的数据类型，系统无法自动转换，例如日期。例如前端传一个日期到后端，后端不是用字符串接收，而是使用一个 Date 对象接收，这个时候就会出现参数类型转换失败。这个时候，需要我们手动定义参数类型转换器，将日期字符串手动转为一个 Date 对象。

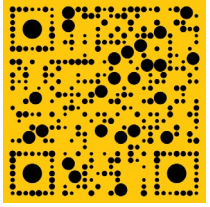
```
@Component
public class DateConverter implements Converter<String, Date> {
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
    public Date convert(String source) {
        try {
            return sdf.parse(source);
        } catch (ParseException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

在自定义的参数类型转换器中，将一个 String 转为 Date 对象，同时，将这个转换器注册为一个 Bean。

接下来，在 SpringMVC 的配置文件中，配置该 Bean，使之生效。

```
<mvc:annotation-driven conversion-service="conversionService"/>
<bean class="org.springframework.format.support.FormattingConversion
ServiceFactoryBean" id="conversionService">
    <property name="converters">
        <set>
            <ref bean="dateConverter"/>
```

公众号·江南一点雨



```
        </set>
    </property>
</bean>
```

配置完成后，在服务端就可以接收前端传来的日期参数了。

7.3.5 集合类的参数

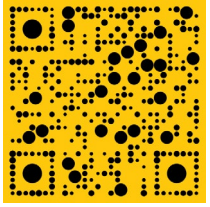
- String 数组

String 数组可以直接用数组去接收，前端传递的时候，数组的传递其实就多相同的 key，这种一般用在 checkbox 中较多。

例如前端增加兴趣爱好一项：

```
<form action="/doAdd" method="post">
    <table>
        <tr>
            <td>书名：</td>
            <td><input type="text" name="name"></td>
        </tr>
        <tr>
            <td>作者姓名：</td>
            <td><input type="text" name="author.name"></td>
        </tr>
        <tr>
            <td>作者年龄：</td>
            <td><input type="text" name="author.age"></td>
        </tr>
        <tr>
            <td>出生日期：</td>
            <td><input type="date" name="author.birthday"></td>
        </tr>
        <tr>
            <td>兴趣爱好：</td>
            <td>
                <input type="checkbox" name="favorites" value="足球">
足球
                <input type="checkbox" name="favorites" value="篮球">
篮球
                <input type="checkbox" name="favorites" value="乒乓球
">乒乓球
```

公众号·江南一点雨



```
            </td>
        </tr>
        <tr>
            <td>价格：</td>
            <td><input type="text" name="price"></td>
        </tr>
        <tr>
            <td>是否上架：</td>
            <td>
                <input type="radio" value="true" name="ispublic">是
                <input type="radio" value="false" name="ispublic">否
            </td>
        </tr>
        <tr>
           <td colspan="2">
               <input type="submit" value="添加">
           </td>
        </tr>
    </table>
</form>
```

在服务端用一个数组去接收 favorites 对象：

```
@RequestMapping(value = "/doAdd",method = RequestMethod.POST)
@ResponseBody
public void doAdd(Book book,String[] favorites) {
    System.out.println(Arrays.toString(favorites));
    System.out.println(book);
}
```

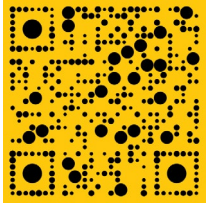
注意，前端传来的数组对象，服务端不可以使用 List 集合去接收。

- List 集合

如果需要使用 List 集合接收前端传来的数据，List 集合本身需要放在一个封装对象中，这个时候，List 中，可以是基本数据类型，也可以是对象。例如有一个班级类，班级里边有学生，学生有多个：

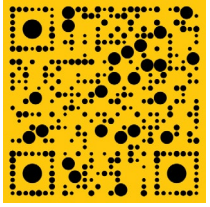
```
public class MyClass {
    private Integer id;
    private List<Student> students;
```

公众号·江南一点雨



```
    @Override
    public String toString() {
        return "MyClass{" +
                "id=" + id +
                ", students=" + students +
                '}';
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public List<Student> getStudents() {
        return students;
    }

    public void setStudents(List<Student> students) {
        this.students = students;
    }
}
public class Student {
    private Integer id;
    private String name;

    @Override
    public String toString() {
        return "Student{" +
                "id=" + id +
                ", name='" + name + '\'' +
                '}';
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }
```

公众号·江南一点雨



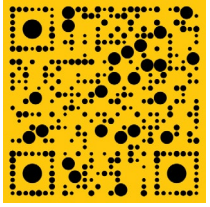
```
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

添加班级的时候，可以传递多个 Student，前端页面写法如下：

```
<form action="/addclass" method="post">
    <table>
        <tr>
            <td>班级编号：</td>
            <td><input type="text" name="id"></td>
        </tr>
        <tr>
            <td>学生编号：</td>
            <td><input type="text" name="students[0].id"></td>
        </tr>
        <tr>
            <td>学生姓名：</td>
            <td><input type="text" name="students[0].name"></td>
        </tr>
        <tr>
            <td>学生编号：</td>
            <td><input type="text" name="students[1].id"></td>
        </tr>
        <tr>
            <td>学生姓名：</td>
            <td><input type="text" name="students[1].name"></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="提交">
            </td>
        </tr>
    </table>
</form>
```

服务端直接接收数据即可：

公众号·江南一点雨



```
@RequestMapping("/addclass")
@ResponseBody
public void addClass(MyClass myClass) {
    System.out.println(myClass);
}
```

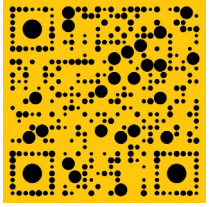
- Map

相对于实体类而言，Map 是一种比较灵活的方案，但是，Map 可维护性比较差，因此一般不推荐使用。

例如给上面的班级类添加其他属性信息：

```
public class MyClass {
    private Integer id;
    private List<Student> students;
    private Map<String, Object> info;

    @Override
    public String toString() {
        return "MyClass{" +
                "id=" + id +
                ", students=" + students +
                ", info=" + info +
                '}';
    }

    public Map<String, Object> getInfo() {
        return info;
    }

    public void setInfo(Map<String, Object> info) {
        this.info = info;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }
```

公众号·江南一点雨

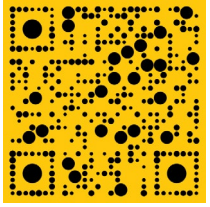


```
    public List<Student> getStudents() {
        return students;
    }

    public void setStudents(List<Student> students) {
        this.students = students;
    }
}
```

在前端，通过如下方式给 info 这个 Map 赋值。

```
<form action="/addclass" method="post">
    <table>
        <tr>
            <td>班级编号：</td>
            <td><input type="text" name="id"></td>
        </tr>
        <tr>
            <td>班级名称：</td>
            <td><input type="text" name="info['name']"></td>
        </tr>
        <tr>
            <td>班级位置：</td>
            <td><input type="text" name="info['pos']"></td>
        </tr>
        <tr>
            <td>学生编号：</td>
            <td><input type="text" name="students[0].id"></td>
        </tr>
        <tr>
            <td>学生姓名：</td>
            <td><input type="text" name="students[0].name"></td>
        </tr>
        <tr>
            <td>学生编号：</td>
            <td><input type="text" name="students[1].id"></td>
        </tr>
        <tr>
            <td>学生姓名：</td>
            <td><input type="text" name="students[1].name"></td>
        </tr>
        <tr>
            <td colspan="2">
```

公众号·江南一点雨



```
                <input type="submit" value="提交">
            </td>
        </tr>
    </table>
</form>
```

8. 文件上传

SpringMVC 中对文件上传做了封装，我们可以更加方便的实现文件上传。从 Spring3.1 开始，对于文件上传，提供了两个处理器：

- CommonsMultipartResolver
- StandardServletMultipartResolver

第一个处理器兼容性较好，可以兼容 Servlet3.0 之前的版本，但是它依赖了 commons-fileupload 这个第三方工具，所以如果使用这个，一定要添加 commons-fileupload 依赖。

第二个处理器兼容性较差，它适用于 Servlet3.0 之后的版本，它不依赖第三方工具，使用它，可以直接做文件上传。

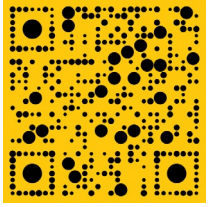
8.1 CommonsMultipartResolver

使用 CommonsMultipartResolver 做文件上传，需要首先添加 commons-fileupload 依赖，如下：

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

然后，在 SpringMVC 的配置文件中，配置 MultipartResolver：

公众号·江南一点雨



```
<bean class="org.springframework.web.multipart.commons.CommonsMultip
artResolver" id="multipartResolver"/>
```

注意，这个 Bean 一定要有 id，并且 id 必须是 multipartResolver

接下来，创建 jsp 页面：

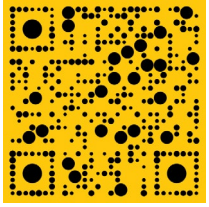
```
<form action="/upload" method="post" enctype="multipart/form-data">
    <input type="file" name="file">
    <input type="submit" value="上传">
</form>
```

注意文件上传请求是 POST 请求，enctype 一定是 multipart/form-data

然后，开发文件上传接口：

```
@Controller
public class FileUploadController {
    SimpleDateFormat sdf = new SimpleDateFormat("/yyyy/MM/dd/");

    @RequestMapping("/upload")
    @ResponseBody
    public String upload(MultipartFile file, HttpServletRequest req)
{
        String format = sdf.format(new Date());
        String realPath = req.getServletContext().getRealPath("/img")
 + format;
        File folder = new File(realPath);
        if (!folder.exists()) {
            folder.mkdirs();
        }
        String oldName = file.getOriginalFilename();
        String newName = UUID.randomUUID().toString() + oldName.subst
ring(oldName.lastIndexOf("."));
        try {
            file.transferTo(new File(folder, newName));
            String url = req.getScheme() + "://" + req.getServerName()
 + ":" + req.getServerPort() + "/img" + format + newName;
            return url;
        } catch (IOException e) {
            e.printStackTrace();
        }
```

公众号·江南一点雨



```
        return "failed";
    }
}
```

这个文件上传方法中，一共做了四件事：

1. 解决文件保存路径，这里是保存在项目运行目录下的 img 目录下，然后利用日期继续宁分类
2. 处理文件名问题，使用 UUID 做新的文件名，用来代替旧的文件名，可以有效防止文件名冲突
3. 保存文件
4. 生成文件访问路径

这里还有一个小问题，在 SpringMVC 中，静态资源默认都是被自动拦截的，无法访问，意味着上传成功的图片无法访问，因此，还需要我们在 SpringMVC 的配置文件中，再添加如下配置：

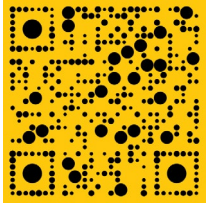
```
<mvc:resources mapping="/**" location="/"/>
```

完成之后，就可以访问 jsp 页面，做文件上传了。

当然，默认的配置不一定满足我们的需求，我们还可以自己手动配置文件上传大小等：

```
<bean class="org.springframework.web.multipart.commons.CommonsMultipartResolver" id="multipartResolver">
    <!--默认的编码-->
    <property name="defaultEncoding" value="UTF-8"/>
    <!--上传的总文件大小-->
    <property name="maxUploadSize" value="1048576"/>
    <!--上传的单个文件大小-->
    <property name="maxUploadSizePerFile" value="1048576"/>
    <!--内存中最大的数据量，超过这个数据量，数据就要开始往硬盘中写了-->
    <property name="maxInMemorySize" value="4096"/>
    <!--临时目录，超过 maxInMemorySize 配置的大小后，数据开始往临时目录写，等全部上传完成后，再将数据合并到正式的文件上传目录-->
    <property name="uploadTempDir" value="file:///E:\\tmp"/>
</bean>
```

公众号·江南一点雨



8.2 StandardServletMultipartResolver

这种文件上传方式，不需要依赖第三方 jar（主要是不需要添加 commons-fileupload 这个依赖），但是也不支持 Servlet3.0 之前的版本。

使用 StandardServletMultipartResolver ，那我们首先在 SpringMVC 的配置文件中，配置这个 Bean：

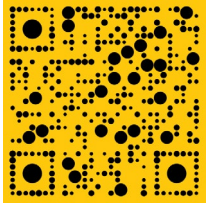
```
<bean class="org.springframework.web.multipart.support.StandardServletMultipartResolver" id="multipartResolver">
</bean>
```

注意，这里 Bean 的名字依然叫 multipartResolver

配置完成后，注意，这个 Bean 无法直接配置上传文件大小等限制。需要在 web.xml 中进行配置（这里，即使不需要限制文件上传大小，也需要在 web.xml 中配置 multipart-config）：

```
<servlet>
    <servlet-name>springmvc</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring-servlet.xml</param-value>
    </init-param>
    <multipart-config>
        <!--文件保存的临时目录，这个目录系统不会主动创建-->
        <location>E:\\temp</location>
        <!--上传的单个文件大小-->
        <max-file-size>1048576</max-file-size>
        <!--上传的总文件大小-->
        <max-request-size>1048576</max-request-size>
        <!--这个就是内存中保存的文件最大大小-->
        <file-size-threshold>4096</file-size-threshold>
    </multipart-config>
</servlet>
```

公众号·江南一点雨



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

配置完成后，就可以测试文件上传了，测试方式和上面一样。

8.3 多文件上传

多文件上传分为两种，一种是 key 相同的文件，另一种是 key 不同的文件。

8.3.1 key 相同的文件

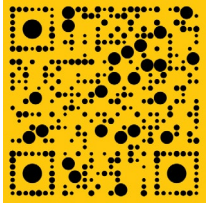
这种上传，前端页面一般如下：

```
<form action="/upload2" method="post" enctype="multipart/form-data">
    <input type="file" name="files" multiple>
    <input type="submit" value="上传">
</form>
```

主要是 input 节点中多了 multiple 属性。后端用一个数组来接收文件即可：

```
@RequestMapping("/upload2")
@ResponseBody
public void upload2(MultipartFile[] files, HttpServletRequest req) {
    String format = sdf.format(new Date());
    String realPath = req.getServletContext().getRealPath("/img") + f
ormat;
    File folder = new File(realPath);
    if (!folder.exists()) {
        folder.mkdirs();
    }
    try {
        for (MultipartFile file : files) {
            String oldName = file.getOriginalFilename();
            String newName = UUID.randomUUID().toString() + oldName.su
bstring(oldName.lastIndexOf("."));
            file.transferTo(new File(folder, newName));
            String url = req.getScheme() + "://" + req.getServerName()
 + ":" + req.getServerPort() + "/img" + format + newName;
            System.out.println(url);
```

公众号·江南一点雨



```
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

8.3.2 key 不同的文件

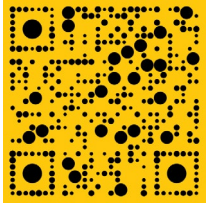
key 不同的，一般前端定义如下：

```
<form action="/upload3" method="post" enctype="multipart/form-data">
    <input type="file" name="file1">
    <input type="file" name="file2">
    <input type="submit" value="上传">
</form>
```

这种，在后端用不同的变量来接收就行了：

```
@RequestMapping("/upload3")
@ResponseBody
public void upload3(MultipartFile file1, MultipartFile file2, HttpSer
vletRequest req) {
    String format = sdf.format(new Date());
    String realPath = req.getServletContext().getRealPath("/img") + f
ormat;
    File folder = new File(realPath);
    if (!folder.exists()) {
        folder.mkdirs();
    }
    try {
        String oldName = file1.getOriginalFilename();
        String newName = UUID.randomUUID().toString() + oldName.subst
ring(oldName.lastIndexOf("."));
        file1.transferTo(new File(folder, newName));
        String url1 = req.getScheme() + "://" + req.getServerName() +
":" + req.getServerPort() + "/img" + format + newName;
        System.out.println(url1);
        String oldName2 = file2.getOriginalFilename();
        String newName2 = UUID.randomUUID().toString() + oldName2.sub
string(oldName2.lastIndexOf("."));
        file2.transferTo(new File(folder, newName2));
        String url2 = req.getScheme() + "://" + req.getServerName() +
":" + req.getServerPort() + "/img" + format + newName2;
```

公众号·江南一点雨



```
        System.out.println(url2);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

9. 全局异常处理

项目中，可能会抛出多个异常，我们不可以直接将异常的堆栈信息展示给用户，有两个原因：

1. 用户体验不好
2. 非常不安全

所以，针对异常，我们可以自定义异常处理，SpringMVC 中，针对全局异常也提供了相应的解决方案，主要是通过 @ControllerAdvice 和 @ExceptionHandler 两个注解来处理的。

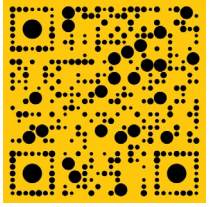
以第八节的文件上传大小超出限制为例，自定义异常，只需要提供一个异常处理类即可：

```
@ControllerAdvice//表示这是一个增强版的 Controller，主要用来做全局数据处理
public class MyException {
    @ExceptionHandler(Exception.class)
    public ModelAndView fileuploadException(Exception e) {
        ModelAndView error = new ModelAndView("error");
        error.addObject("error", e.getMessage());
        return error;
    }
}
```

在这里：

- @ControllerAdvice 表示这是一个增强版的 Controller，主要用来做全局数据处理

公众号·江南一点雨



- @ExceptionHandler 表示这是一个异常处理方法，这个注解的参数，表示需要拦截的异常，参数为 Exception 表示拦截所有异常，这里也可以具体到某一个异常，如果具体到某一个异常，那么发生了其他异常则不会被拦截到。
- 异常方法的定义，和 Controller 中方法的定义一样，可以返回 ModelAndview，也可以返回 String 或者 void

例如如下代码，指挥拦截文件上传异常，其他异常和它没关系，不会进入到自定义异常处理的方法中来。

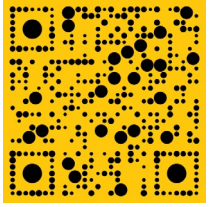
```
@ControllerAdvice//表示这是一个增强版的 Controller，主要用来做全局数据处理
public class MyException {
    @ExceptionHandler(MaxUploadSizeExceededException.class)
    public ModelAndView fileuploadException(MaxUploadSizeExceededException e) {
        ModelAndView error = new ModelAndView("error");
        error.addObject("error", e.getMessage());
        return error;
    }
}
```

10. 服务端数据校验

B/S 系统中对 http 请求数据的校验多数在客户端进行，这也是出于简单及用户体验性上考虑，但是在一些安全性要求高的系统中服务端校验是不可缺少的，实际上，几乎所有的系统，凡是涉及到数据校验，都需要在服务端进行二次校验。为什么要在服务端进行二次校验呢？这需要理解客户端校验和服务端校验各自的目的。

1. 客户端校验，我们主要是为了提高用户体验，例如用户输入一个邮箱地址，要校验这个邮箱地址是否合法，没有必要发送到服务端进行校验，直接在前端用 js 进行校验即可。但是大家需要明白的是，前端校验无法代替后端校验，前端校验可以有效的提高用户体验，但是无法确保数据完整性，因为在 B/S 架构中，用户可以方便的拿到请求地址，然后直接发送请求，传递非法参数。
2. 服务端校验，虽然用户体验不好，但是可以有效的保证数据安全与完整性。
3. 综上，实际项目中，两个一起用。

公众号·江南一点雨



Spring 支持 JSR-303 验证框架，JSR-303 是 JAVA EE 6 中的一项子规范，叫做 Bean Validation，官方参考实现是 Hibernate Validator（与 Hibernate ORM 没有关系），JSR-303 用于对 Java Bean 中的字段的值进行验证。

10.1 普通校验

普通校验，是这里最基本的用法。

首先，我们需要加入校验需要的依赖：

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.1.0.Final</version>
</dependency>
```

接下来，在 SpringMVC 的配置文件中配置校验的 Bean：

```
<bean class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean" id="validatorFactoryBean">
    <property name="providerClass" value="org.hibernate.validator.HibernateValidator"/>
</bean>
<mvc:annotation-driven validator="validatorFactoryBean"/>
```

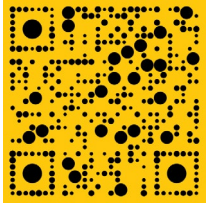
配置时，提供一个 LocalValidatorFactoryBean 的实例，然后 Bean 的校验使用 HibernateValidator。

这样，配置就算完成了。

接下来，我们提供一个添加学生的页面：

```
<form action="/addstudent" method="post">
    <table>
        <tr>
            <td>学生编号：</td>
            <td><input type="text" name="id"></td>
```

公众号·江南一点雨

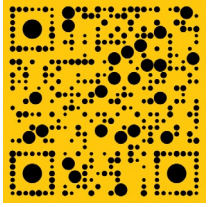


```
        </tr>
        <tr>
            <td>学生姓名：</td>
            <td><input type="text" name="name"></td>
        </tr>
        <tr>
            <td>学生邮箱：</td>
            <td><input type="text" name="email"></td>
        </tr>
        <tr>
            <td>学生年龄：</td>
            <td><input type="text" name="age"></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="提交">
            </td>
        </tr>
    </table>
</form>
```

在这里需要提交的数据中，假设学生编号不能为空，学生姓名长度不能超过 10 且不能为空，邮箱地址要合法，年龄不能超过 150。那么在定义实体类的时候，就可以加入这个判断条件了。

```
public class Student {
    @NotNull
    private Integer id;
    @NotNull
    @Size(min = 2,max = 10)
    private String name;
    @Email
    private String email;
    @Max(150)
    private Integer age;

    public String getEmail() {
        return email;
    }

    @Override
    public String toString() {
```

公众号：江南一点雨



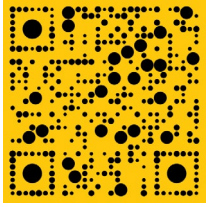
```
        return "Student{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", email='" + email + '\'' +
                ", age=" + age +
                '}';
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

在这里:

- @NotNull 表示这个字段不能为空
- @Size 中描述了这个字符串长度的限制
- @Email 表示这个字段的值必须是一个邮箱地址

公众号·江南一点雨

扫码关注微信公众号【江南一点雨】，回复 2TB，获取超 2TB 免费 Java 学习资源



- @Max 表示这个字段的最大值

定义完成后，接下来，在 Controller 中定义接口：

```
@Controller
public class StudentController {
    @RequestMapping("/addstudent")
    @ResponseBody
    public void addStudent(@Validated Student student, BindingResult
result) {
        if (result != null) {
            //校验未通过，获取所有的异常信息并展示出来
            List<ObjectError> allErrors = result.getAllErrors();
            for (ObjectError allError : allErrors) {
                System.out.println(allError.getObjectName()+":"+allErr
or.getDefaultMessage());
            }
        }
    }
}
```

在这里：

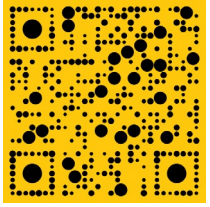
- @Validated 表示 Student 中定义的校验规则将会生效
- BindingResult 表示出错信息，如果这个变量不为空，表示有错误，否则校验通过。

接下来就可以启动项目了。访问 jsp 页面，然后添加 Student，查看校验规则是否生效。

默认情况下，打印出来的错误信息时系统默认的错误信息，这个错误信息，我们也可以自定义。自定义方式如下：

由于 properties 文件中的中文会乱码，所以需要我们先修改一下 IDEA 配置，点 File-->Settings->Editor-->File Encodings，如下：

公众号·江南一点雨



然后定义错误提示文本，在 resources 目录下新建一个 MyMessage.properties 文件，内容如下：

```
student.id.notnull=id 不能为空
student.name.notnull=name 不能为空
student.name.length=name 最小长度为 2 ， 最大长度为 10
student.email.error=email 地址非法
student.age.error=年龄不能超过 150
```

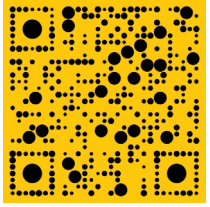
接下来，在 SpringMVC 配置中，加载这个配置文件：

```
<bean class="org.springframework.validation.beanvalidation.LocalVali
datorFactoryBean" id="validatorFactoryBean">
    <property name="providerClass" value="org.hibernate.validator.Hib
ernateValidator"/>
    <property name="validationMessageSource" ref="bundleMessageSource
"/>
</bean>
<bean class="org.springframework.context.support.ReloadableResourceB
undleMessageSource" id="bundleMessageSource">
    <property name="basenames">
        <list>
            <value>classpath:MyMessage</value>
        </list>
    </property>
    <property name="defaultEncoding" value="UTF-8"/>
    <property name="cacheSeconds" value="300"/>
</bean>
<mvc:annotation-driven validator="validatorFactoryBean"/>
```

最后，在实体类上的注解中，加上校验出错时的信息：

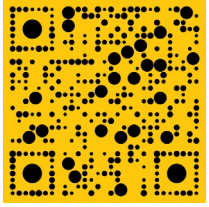
```
public class Student {
    @NotNull(message = "{student.id.notnull}")
    private Integer id;
    @NotNull(message = "{student.name.notnull}")
    @Size(min = 2,max = 10,message = "{student.name.length}")
    private String name;
    @Email(message = "{student.email.error}")
    private String email;
    @Max(value = 150,message = "{student.age.error}")
    private Integer age;
```

公众号·江南一点雨



```
    public String getEmail() {
        return email;
    }

    @Override
    public String toString() {
        return "Student{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", email='" + email + '\'' +
                ", age=" + age +
                '}';
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

公众号·江南一点雨



配置完成后，如果校验再出错，就会展示我们自己的出错信息了。

10.2 分组校验

由于校验规则都是定义在实体类上面的，但是，在不同的数据提交环境下，校验规则可能不一样。例如，用户的 id 是自增长的，添加的时候，可以不用传递用户 id，但是修改的时候则必须传递用户 id，这种情况下，就需要使用分组校验。

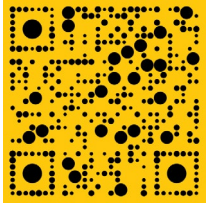
分组校验，首先需要定义校验组，所谓的校验组，其实就是空接口：

```
public interface ValidationGroup1 {
}
public interface ValidationGroup2 {
}
```

然后，在实体类中，指定每一个校验规则所属的组：

```
public class Student {
    @NotNull(message = "{student.id.notnull}",groups = ValidationGrou
p1.class)
    private Integer id;
    @NotNull(message = "{student.name.notnull}",groups = {ValidationG
roup1.class, ValidationGroup2.class})
    @Size(min = 2,max = 10,message = "{student.name.length}",groups =
 {ValidationGroup1.class, ValidationGroup2.class})
    private String name;
    @Email(message = "{student.email.error}",groups = {ValidationGrou
p1.class, ValidationGroup2.class})
    private String email;
    @Max(value = 150,message = "{student.age.error}",groups = {Valida
tionGroup2.class})
    private Integer age;

    public String getEmail() {
        return email;
    }
```

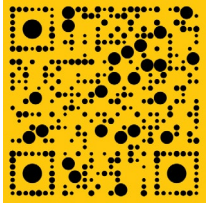
公众号·江南一点雨



```
    @Override
    public String toString() {
        return "Student{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", email='" + email + '\'' +
                ", age=" + age +
                '}';
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

在 group 中指定每一个校验规则所属的组，一个规则可以属于一个组，也可以属于多个组。

公众号·江南一点雨



最后，在接收参数的地方，指定校验组：

```
@Controller
public class StudentController {
    @RequestMapping("/addstudent")
    @ResponseBody
    public void addStudent(@Validated(ValidationGroup2.class) Student
 student, BindingResult result) {
        if (result != null) {
            //校验未通过，获取所有的异常信息并展示出来
            List<ObjectError> allErrors = result.getAllErrors();
            for (ObjectError allError : allErrors) {
                System.out.println(allError.getObjectName()+":"+allErr
or.getDefaultMessage());
            }
        }
    }
}
```

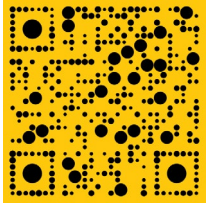
配置完成后，属于 ValidationGroup2 这个组的校验规则，才会生效。

10.3 校验注解

校验注解，主要有如下几种：

- @Null 被注解的元素必须为 null
- @NotNull 被注解的元素必须不为 null
- @AssertTrue 被注解的元素必须为 true
- @AssertFalse 被注解的元素必须为 false
- @Min(value) 被注解的元素必须是一个数字，其值必须大于等于指定的最小值
- @Max(value) 被注解的元素必须是一个数字，其值必须小于等于指定的最大值
- @DecimalMin(value) 被注解的元素必须是一个数字，其值必须大于等于指定的最小值
- @DecimalMax(value) 被注解的元素必须是一个数字，其值必须小于等于指定的最大值
- @Size(max=, min=) 被注解的元素的大小必须在指定的范围内

公众号·江南一点雨



- @Digits (integer, fraction) 被注解的元素必须是一个数字，其值必须在可接受的范围内
- @Past 被注解的元素必须是一个过去的日期
- @Future 被注解的元素必须是一个将来的日期
- @Pattern(regex=,flag=) 被注解的元素必须符合指定的正则表达式
- @NotBlank(message =) 验证字符串非 null，且长度必须大于 0
- @Email 被注解的元素必须是电子邮箱地址
- @Length(min=,max=) 被注解的字符串的大小必须在指定的范围内
- @NotEmpty 被注解的字符串的必须非空
- @Range(min=,max=,message=) 被注解的元素必须在合适的范围内

11.1 数据回显基本用法

数据回显就是当用户数据提交失败时，自动填充好已经输入的数据。一般来说，如果使用 Ajax 来做数据提交，基本上是没有数据回显这个需求的，但是如果是通过表单做数据提交，那么数据回显就非常有必要了。

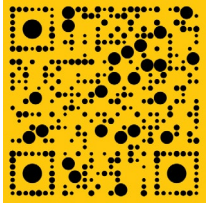
11.1.1 简单数据类型

简单数据类型，实际上框架在这里没有提供任何形式的支持，就是我们自己手动配置。我们继续在第 10 小节的例子上演示 Demo。加入提交的 Student 数据不符合要求，那么重新回到添加 Student 页面，并且预设之前已经填好的数据。

首先我们先来改造一下 student.jsp 页面：

```
<form action="/addstudent" method="post">
    <table>
        <tr>
            <td>学生编号：</td>
            <td><input type="text" name="id" value="${id}"></td>
```

公众号·江南一点雨



```
        </tr>
        <tr>
            <td>学生姓名：</td>
            <td><input type="text" name="name" value="${name}"></td>
        </tr>
        <tr>
            <td>学生邮箱：</td>
            <td><input type="text" name="email" value="${email}"></td>
        </tr>
        <tr>
            <td>学生年龄：</td>
            <td><input type="text" name="age" value="${age}"></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="提交">
            </td>
        </tr>
    </table>
</form>
```

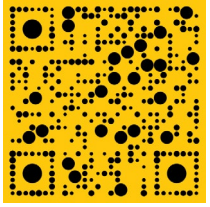
在接收数据时，使用简单数据类型去接收：

```
@RequestMapping("/addstudent")
public String addStudent2(Integer id, String name, String email, Inte
ger age, Model model) {
    model.addAttribute("id", id);
    model.addAttribute("name", name);
    model.addAttribute("email", email);
    model.addAttribute("age", age);
    return "student";
}
```

这种方式，相当于框架没有做任何工作，就是我们手动做数据回显的。此时访问页面，服务端会再次定位到该页面，而且数据已经预填好。

公众号：江南一点雨

扫码关注微信公众号【江南一点雨】，回复 2TB，获取超 2TB 免费 Java 学习资源



11.1.2 实体类

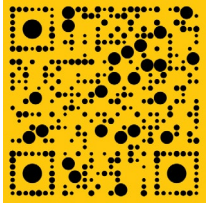
上面这种简单数据类型的回显，实际上非常麻烦，因为需要开发者在服务端一个一个手动设置。如果使用对象的话，就没有这么麻烦了，因为 SpringMVC 在页面跳转时，会自动将对象填充进返回的数据中。

此时，首先修改一下 student.jsp 页面：

```
<form action="/addstudent" method="post">
    <table>
        <tr>
            <td>学生编号：</td>
            <td><input type="text" name="id" value="${student.id}"></t
d>
        </tr>
        <tr>
            <td>学生姓名：</td>
            <td><input type="text" name="name" value="${student.name}
"></td>
        </tr>
        <tr>
            <td>学生邮箱：</td>
            <td><input type="text" name="email" value="${student.emai
l}"></td>
        </tr>
        <tr>
            <td>学生年龄：</td>
            <td><input type="text" name="age" value="${student.age}"><
/td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="提交">
            </td>
        </tr>
    </table>
</form>
```

公众号·江南一点雨

扫码关注微信公众号【江南一点雨】，回复 2TB，获取超 2TB 免费 Java 学习资源



注意，在预填数据中，多了一个 student. 前缀。这 student 就是服务端接收数据的变量名，服务端的变量名和这里的 student 要保持一直。服务端定义如下：

```
@RequestMapping("/addstudent")
public String addStudent(@Validated(ValidationGroup2.class) Student s
tudent, BindingResult result) {
    if (result != null) {
        //校验未通过，获取所有的异常信息并展示出来
        List<ObjectError> allErrors = result.getAllErrors();
        for (ObjectError allError : allErrors) {
            System.out.println(allError.getObjectName()+":"+allError.
getDefaultMessage());
        }
        return "student";
    }
    return "hello";
}
```

注意，服务端什么都不用做，就说要返回的页面就行了，student 这个变量会被自动填充到返回的 Model 中。变量名就是填充时候的 key。如果想自定义这个 key，可以在参数中写出来 Model，然后手动加入 Student 对象，就像简单数据类型回显那样。

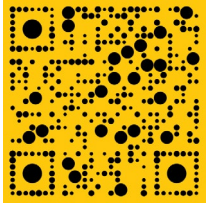
另一种定义回显变量别名的方式，就是使用 @ModelAttribute 注解。

11.2 @ModelAttribute

@ModelAttribute 这个注解，主要有两方面的功能：

1. 在数据回显时，给变量定义别名
2. 定义全局数据

公众号·江南一点雨



11.2.1 定义别名

在数据回显时，给变量定义别名，非常容易，直接加这个注解即可：

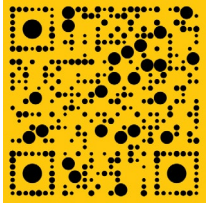
```
@RequestMapping("/addstudent")
public String addStudent(@ModelAttribute("s") @Validated(ValidationG
roup2.class) Student student, BindingResult result) {
    if (result != null) {
        //校验未通过，获取所有的异常信息并展示出来
        List<ObjectError> allErrors = result.getAllErrors();
        for (ObjectError allError : allErrors) {
            System.out.println(allError.getObjectName()+":"+allError.
getDefaultMessage());
        }
        return "student";
    }
    return "hello";
}
```

这样定义完成后，在前端再次访问回显的变量时，变量名称就不是 student

了，而是 s：

```
<form action="/addstudent" method="post">
    <table>
        <tr>
            <td>学生编号：</td>
            <td><input type="text" name="id" value="${s.id}"></td>
        </tr>
        <tr>
            <td>学生姓名：</td>
            <td><input type="text" name="name" value="${s.name}"></td>
        </tr>
        <tr>
            <td>学生邮箱：</td>
            <td><input type="text" name="email" value="${s.email}"></t
d>
        </tr>
        <tr>
            <td>学生年龄：</td>
            <td><input type="text" name="age" value="${s.age}"></td>
        </tr>
        <tr>
```

公众号·江南一点雨



```
            <td colspan="2">
                <input type="submit" value="提交">
            </td>
        </tr>
    </table>
</form>
```

11.2.2 定义全局数据

假设有一个 Controller 中有很多方法，每个方法都会返回数据给前端，但是每个方法返回给前端的数据又不太一样，虽然不太一样，但是没有方法的返回值又有一些公共的部分。可以将这些公共的部分提取出来单独封装成一个方法，用 @ModelAttribute 注解来标记。

例如在一个 Controller 中 ，添加如下代码:

```
@ModelAttribute("info")
public Map<String,Object> info() {
    Map<String, Object> map = new HashMap<>();
    map.put("username", "javaboy");
    map.put("address", "www.javaboy.org");
    return map;
}
```

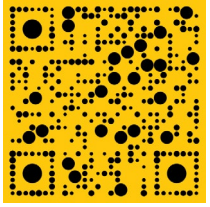
当用户访问当前 Controller 中的任意一个方法，在返回数据时，都会将添加了 @ModelAttribute 注解的方法的返回值，一起返回给前端。@ModelAttribute 注解中的 info 表示返回数据的 key。

12.1 返回 JSON

目前主流的 JSON 处理工具主要有三种:

- jackson
- gson
- fastjson

公众号·江南一点雨



在 SpringMVC 中，对 jackson 和 gson 都提供了相应的支持，就是如果使用这两个作为 JSON 转换器，只需要添加对应的依赖就可以了，返回的对象和返回的集合、Map 等都会自动转为 JSON，但是，如果使用 fastjson，除了添加相应的依赖之外，还需要自己手动配置 HttpMessageConverter 转换器。其实前两个也是使用 HttpMessageConverter 转换器，但是是 SpringMVC 自动提供的，SpringMVC 没有给 fastjson 提供相应的转换器。

12.1.1 jackson

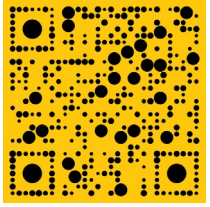
jackson 是一个使用比较多，时间也比较长的 JSON 处理工具，在 SpringMVC 中使用 jackson ，只需要添加 jackson 的依赖即可：

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.10.1</version>
</dependency>
```

依赖添加成功后，凡是在接口中直接返回的对象，集合等等，都会自动转为 JSON。如下：

```
public class Book {
    private Integer id;
    private String name;
    private String author;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getAuthor() {
```

公众号·江南一点雨



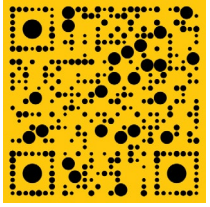
```
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }
}
@RequestMapping("/book")
@ResponseBody
public Book getBookById() {
    Book book = new Book();
    book.setId(1);
    book.setName("三国演义");
    book.setAuthor("罗贯中");
    return book;
}
```

这里返回一个对象，但是在前端接收到的则是一个 JSON 字符串，这个对象会通过 HttpMessageConverter 自动转为 JSON 字符串。

如果想返回一个 JSON 数组，写法如下：

```
@RequestMapping("/books")
@ResponseBody
public List<Book> getAllBooks() {
    List<Book> list = new ArrayList<Book>();
    for (int i = 0; i < 10; i++) {
        Book book = new Book();
        book.setId(i);
        book.setName("三国演义:" + i);
        book.setAuthor("罗贯中:" + i);
        list.add(book);
    }
```

公众号：江南一点雨



```
    return list;
}
```

添加了 jackson ，就能够自动返回 JSON，这个依赖于一个名为 HttpMessageConverter 的类，这本身是一个接口，从名字上就可以看出，它的作用是 Http 消息转换器，既然是消息转换器，它提供了两方面的功能：

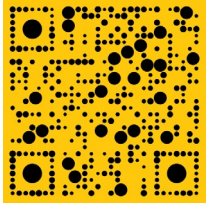
1. 将返回的对象转为 JSON
2. 将前端提交上来的 JSON 转为对象

但是，HttpMessageConverter 只是一个接口，由各个 JSON 工具提供相应的实现，在 jackson 中，实现的名字叫做 MappingJackson2HttpMessageConverter，而这个东西的初始化，则由 SpringMVC 来完成。除非自己有一些自定义配置的需求，否则一般来说不需要自己提供 MappingJackson2HttpMessageConverter。

举一个简单的应用场景，例如每一本书，都有一个出版日期，修改 Book 类如下：

```
public class Book {
    private Integer id;
    private String name;
    private String author;
    private Date publish;


    public Date getPublish() {
        return publish;
    }

    public void setPublish(Date publish) {
        this.publish = publish;
    }

    public String getName() {
```

公众号·江南一点雨



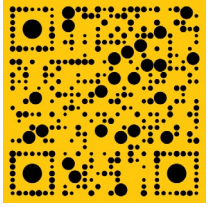
```
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }
}
```

然后在构造 Book 时添加日期属性：

```
@RequestMapping("/book")
@ResponseBody
public Book getBookById() {
    Book book = new Book();
    book.setId(1);
    book.setName("三国演义");
    book.setAuthor("罗贯中");
    book.setPublish(new Date());
    return book;
}
```

访问 /book 接口，返回的 json 格式如下：

公众号·江南一点雨



如果我们想自己定制返回日期的格式，简单的办法，可以通过添加注解来实现：

```
public class Book {
    private Integer id;
    private String name;
    private String author;
    @JsonFormat(pattern = "yyyy-MM-dd",timezone = "Asia/Shanghai")
    private Date publish;
```

注意这里一定要设置时区。

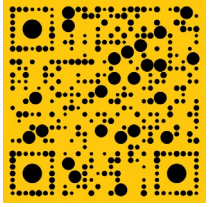
这样，就可以定制返回的日期格式了。

但是，这种方式有一个弊端，这个注解可以加在属性上，也可以加在类上，也就说，最大可以作用到一个类中的所有日期属性上。如果项目中有很多实体类都需要做日期格式化，使用这种方式就比较麻烦了，这个时候，我们可以自己提供一个 jackson 的 HttpMesageConverter 实例，在这个实例中，自己去配置相关属性，这里的配置将是一个全局配置。

在 SpringMVC 配置文件中，添加如下配置：

```
<mvc:annotation-driven>
    <mvc:message-converters>
        <ref bean="httpMessageConverter"/>
    </mvc:message-converters>
</mvc:annotation-driven>
<bean class="org.springframework.http.converter.json.MappingJackson2
HttpMessageConverter" id="httpMessageConverter">
    <property name="objectMapper">
        <bean class="com.fasterxml.jackson.databind.ObjectMapper">
            <property name="dateFormat">
                <bean class="java.text.SimpleDateFormat">
                    <constructor-arg name="pattern" value="yyyy-MM-dd
HH:mm:ss"/>
                </bean>
```

公众号：江南一点雨



```
            </property>
            <property name="timeZone" value="Asia/Shanghai"/>
        </bean>
    </property>
</bean>
```

添加完成后，去掉 Book 实体类中日期格式化的注解，再进行测试，结果如下：

12.1.2 gson

gson 是 Google 推出的一个 JSON 解析器，主要在 Android 开发中使用较多，不过，Web 开发中也是支持这个的，而且 SpringMVC 还针对 Gson 提供了相关的自动化配置，以致我们在项目中只要添加 gson 依赖，就可以直接使用 gson 来做 JSON 解析了。

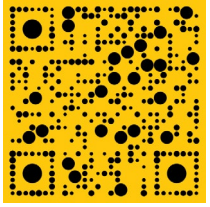
```
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.8.6</version>
</dependency>
```

如果项目中，同时存在 jackson 和 gson 的话，那么默认使用的是 jackson，为社么呢？在 org.springframework.http.converter.support.AllEncompassingFormHttpMessageConverter 类的构造方法中，加载顺序就是先加载 jackson 的 HttpMessageConverter，后加载 gson 的 HttpMessageConverter。

加完依赖之后，就可以直接返回 JSON 字符串了。使用 Gson 时，如果想做自定义配置，则需要自定义 HttpMessageConverter。

```
<mvc:annotation-driven>
    <mvc:message-converters>
```

公众号·江南一点雨



```
        <ref bean="httpMessageConverter"/>
    </mvc:message-converters>
</mvc:annotation-driven>
<bean class="org.springframework.http.converter.json.GsonHttpMessage
Converter" id="httpMessageConverter">
    <property name="gson">
        <bean class="com.google.gson.Gson" factory-bean="gsonBuilder"
 factory-method="create"/>
    </property>
</bean>
<bean class="com.google.gson.GsonBuilder" id="gsonBuilder">
    <property name="dateFormat" value="yyyy-MM-dd"/>
</bean>
```

12.1.3 fastjson

fastjson 号称最快的 JSON 解析器，但是也是这三个中 BUG 最多的一个。在 SpringMVC 并没针对 fastjson 提供相应的 HttpMessageConverter，所以，fastjson 在使用时，一定要自己手动配置 HttpMessageConverter（前面两个如果没有特殊需要，直接添加依赖就可以了）。

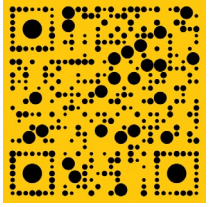
使用 fastjson，我们首先添加 fastjson 依赖：

```
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.2.60</version>
</dependency>
```

然后在 SpringMVC 的配置文件中配置 HttpMessageConverter：

```
<mvc:annotation-driven>
    <mvc:message-converters>
        <ref bean="httpMessageConverter"/>
    </mvc:message-converters>
</mvc:annotation-driven>
<bean class="com.alibaba.fastjson.support.spring.FastJsonHttpMessage
Converter" id="httpMessageConverter">
    <property name="fastJsonConfig">
```

公众号·江南一点雨



```
        <bean class="com.alibaba.fastjson.support.config.FastJsonConf
ig">
            <property name="dateFormat" value="yyyy-MM-dd"/>
        </bean>
    </property>
</bean>
```

fastjson 默认中文乱码，添加如下配置解决：

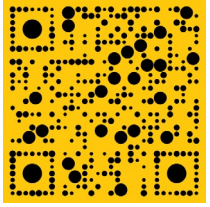
```
<mvc:annotation-driven>
    <mvc:message-converters>
        <ref bean="httpMessageConverter"/>
    </mvc:message-converters>
</mvc:annotation-driven>
<bean class="com.alibaba.fastjson.support.spring.FastJsonHttpMessage
Converter" id="httpMessageConverter">
    <property name="fastJsonConfig">
        <bean class="com.alibaba.fastjson.support.config.FastJsonConf
ig">
            <property name="dateFormat" value="yyyy-MM-dd"/>
        </bean>
    </property>
    <property name="supportedMediaTypes">
        <list>
            <value>application/json;charset=utf-8</value>
        </list>
    </property>
</bean>
```

12.2 接收 JSON

浏览器传来的参数，可以是 key/value 形式的，也可以是一个 JSON 字符串。在 Jsp/Servlet 中，我们接收 key/value 形式的参数，一般是通过 getParameter 方法。如果客户端商户惨的是 JSON 数据，我们可以通过如下格式进行解析：

```
@RequestMapping("/addbook2")
@ResponseBody
public void addBook2(HttpServletRequest req) throws IOException {
    ObjectMapper om = new ObjectMapper();
```

公众号·江南一点雨



```
    Book book = om.readValue(req.getInputStream(), Book.class);
    System.out.println(book);
}
```

但是这种解析方式有点麻烦，在 SpringMVC 中，我们可以通过一个注解来快速的将一个 JSON 字符串转为一个对象：

```
@RequestMapping("/addbook3")
@ResponseBody
public void addBook3(@RequestBody Book book) {
    System.out.println(book);
}
```

这样就可以直接收到前端传来的 JSON 字符串了。这也是 HttpMessageConverter 提供的第二个功能。

13. RESTful

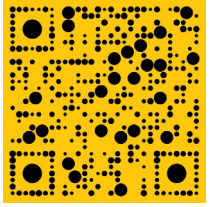
本小节选自外部博客，原文链接：

https://www.ruanyifeng.com/blog/2011/09/restful.html

越来越多的人开始意识到，网站即软件，而且是一种新型的软件。这种"互联网软件"采用客户端/服务器模式，建立在分布式体系上，通过互联网通信，具有高延时（high latency）、高并发等特点。网站开发，完全可以采用软件开发的模式。但是传统上，软件和网络是两个不同的领域，很少有交集；软件开发主要针对单机环境，网络则主要研究系统之间的通信。互联网的兴起，使得这两个领域开始融合，现在我们必须考虑，如何开发在互联网环境中使用的软件。

RESTful 架构，就是目前最流行的一种互联网软件架构。它结构清晰、符合标准、易于理解、扩展方便，所以正得到越来越多网站的采用。

公众号·江南一点雨



但是，到底什么是 RESTful 架构，并不是一个容易说清楚的问题。下面，我就谈谈我理解的 RESTful 架构。、

RESTful 它不是一个具体的架构，不是一个软件，不是一个框架，而是一种规范。在移动互联网兴起之前，我们都很少提及 RESTful，主要是因为用的少，移动互联网兴起后，RESTful 得到了非常广泛的应用，因为在移动互联网兴起之后，我们再开发后端应用，就不仅仅只是开发一个网站了，还对应了多个前端（Android、iOS、HTML5 等等），这个时候，我们在设计后端接口是，就需要考虑接口的形式，格式，参数的传递等等诸多问题了。

13.1 起源

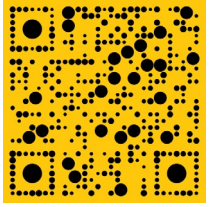
REST 这个词，是 Roy Thomas Fielding 在他 2000 年的博士论文中提出的。

Fielding 是一个非常重要的人，他是 HTTP 协议（1.0 版和 1.1 版）的主要设计者、Apache 服务器软件的作者之一、Apache 基金会的第一任主席。所以，他的这篇论文一经发表，就引起了关注，并且立即对互联网开发产生了深远的影响。

他这样介绍论文的写作目的：

"本文研究计算机科学两大前沿----软件和网络----的交叉点。长期以来，软件研究主要关注软件设计的分类、设计方法的演化，很少客观地评估不同的设计选择对系统行为的影响。而相反地，网络研究主要关注系统之间通信行为的细节、如何改进特定通信机制的表现，常常忽视了一个事实，那就是改变应用程序的互动风格比改变互动协议，对整体表现有更大的影响。我这篇文章的写作

扫码关注微信公众号【江南一点雨】，回复 2TB，获取超 2TB 免费 Java 学习资源



目的，就是想在符合架构原理的前提下，理解和评估以网络为基础的应用软件的架构设计，得到一个功能强、性能好、适宜通信的架构。"

13.2 名称

Fielding 将他对互联网软件的架构原则，定名为 REST，即 Representational State Transfer 的缩写。我对这个词组的翻译是"表现层状态转化"。

如果一个架构符合 REST 原则，就称它为 RESTful 架构。

要理解 RESTful 架构，最好的方法就是去理解 Representational State Transfer 这个词组到底是什么意思，它的每一个词代表了什么涵义。如果你把这个名称搞懂了，也就不难体会 REST 是一种什么样的设计。

13.3 资源（Resources）

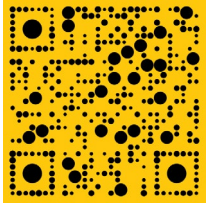
REST 的名称"表现层状态转化"中，省略了主语。"表现层"其实指的是"资源"（Resources）的"表现层"。

所谓"资源"，就是网络上的一个实体，或者说是网络上的一个具体信息。它可以是一段文本、一张图片、一首歌曲、一种服务，总之就是一个具体的实在。你可以用一个 URI （统一资源定位符）指向它，每种资源对应一个特定的 URI。要获取这个资源，访问它的 URI 就可以，因此 URI 就成了每一个资源的地址或独一无二的识别符。

所谓"上网"，就是与互联网上一系列的"资源"互动，调用它的 URI。

在 RESTful 风格的应用中，每一个 URI 都代表了一个资源。

扫码关注微信公众号【江南一点雨】，回复 2TB，获取超 2TB 免费 Java 学习资源



13.4 表现层（Representation）

"资源"是一种信息实体，它可以有多种外在表现形式。我们把"资源"具体呈现出来的形式，叫做它的"表现层"（Representation）。

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

URI 只代表资源的实体，不代表它的形式。严格地说，有些网址最后的 ".html" 后缀名是不必要的，因为这个后缀名表示格式，属于 "表现层" 范畴，而 URI 应该只代表"资源"的位置。它的具体表现形式，应该在 HTTP 请求的头信息中用 Accept 和 Content-Type 字段指定，这两个字段才是对"表现层"的描述。

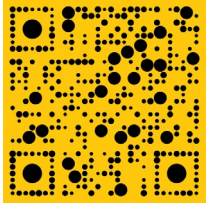
13.5 状态转化（State Transfer）

访问一个网站，就代表了客户端和服务器的一个互动过程。在这个过程中，势必涉及到数据和状态的变化。

互联网通信协议 HTTP 协议，是一个无状态协议。这意味着，所有的状态都保存在服务器端。因此，如果客户端想要操作服务器，必须通过某种手段，让服务器端发生"状态转化"（State Transfer）。而这种转化是建立在表现层之上的，所以就是"表现层状态转化"。

客户端用到的手段，只能是 HTTP 协议。具体来说，就是 HTTP 协议里面，四个表示操作方式的动词：GET、POST、PUT、DELETE。它们分别对应四种基本操作：

扫码关注微信公众号【江南一点雨】，回复 2TB，获取超 2TB 免费 Java 学习资源



- GET 用来获取资源
- POST 用来新建资源（也可以用于更新资源）
- PUT 用来更新资源
- DELETE 用来删除资源

13.6 综述

综合上面的解释，我们总结一下什么是 RESTful 架构：

- 每一个 URI 代表一种资源；
- 客户端和服务器之间，传递这种资源的某种表现层；
- 客户端通过四个 HTTP 动词，对服务器端资源进行操作，实现"表现层状态转化"。

13.7 误区

RESTful 架构有一些典型的设计误区。

最常见的一种设计错误，就是 URI 包含动词。因为"资源"表示一种实体，所以应该是名词，URI 不应该有动词，动词应该放在 HTTP 协议中。

举例来说，某个 URI 是 /posts/show/1，其中 show 是动词，这个 URI 就设计错了，正确的写法应该是 /posts/1，然后用 GET 方法表示 show。

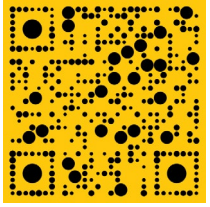
如果某些动作是 HTTP 动词表示不了的，你就应该把动作做成一种资源。比如网上汇款，从账户 1 向账户 2 汇款 500 元，错误的 URI 是：

- POST /accounts/1/transfer/500/to/2

正确的写法是把动词 transfer 改成名词 transaction，资源不能是动词，但是可以是一种服务：

公众号·江南一点雨

扫码关注微信公众号【江南一点雨】，回复 2TB，获取超 2TB 免费 Java 学习资源



```
POST /transaction HTTP/1.1
Host: 127.0.0.1
from=1&to=2&amount=500.00
```

另一个设计误区，就是在 URI 中加入版本号：

- http://www.example.com/app/1.0/foo
- http://www.example.com/app/1.1/foo
- http://www.example.com/app/2.0/foo

因为不同的版本，可以理解成同一种资源的不同表现形式，所以应该采用同一个 URI。版本号可以在 HTTP 请求头信息的 Accept 字段中进行区分（参见 Versioning REST Services）：

```
Accept: vnd.example-com.foo+json; version=1.0
Accept: vnd.example-com.foo+json; version=1.1
Accept: vnd.example-com.foo+json; version=2.0
```

13.8 SpringMVC 的支持

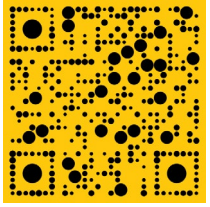
SpringMVC 对 RESTful 提供了非常全面的支持，主要有如下几个注解：

- @RestController

这个注解是一个组合注解：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Controller
@ResponseBody
public @interface RestController {

    /**
     * The value may indicate a suggestion for a logical component nam
e,
     * to be turned into a Spring bean in case of an autodetected comp
onent.
     * @return the suggested component name, if any (or empty String o
```

公众号·江南一点雨



```
therwise)
     * @since 4.0.1
     */
    @AliasFor(annotation = Controller.class)
    String value() default "";

}
```

一般，直接用 @RestController 来标记 Controller，可以不使用 @Controller。

请求方法中，提供了常见的请求方法：

- @PostMapping
- @GetMapping
- @PutMapping
- @DeleteMapping

另外还有一个提取请求地址中的参数的注解 @PathVariable：

```
@GetMapping("/book/{id}")//http://localhost:8080/book/2
public Book getBookById(@PathVariable Integer id) {
    Book book = new Book();
    book.setId(id);
    return book;
}
```

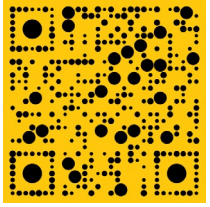
参数 2 将被传递到 id 这个变量上。

14. 静态资源访问

在 SpringMVC 中，静态资源，默认都是被拦截的，例如 html、js、css、jpg、png、txt、pdf 等等，都是无法直接访问的。因为所有请求都被拦截了，所以，针对静态资源，我们要做额外处理，处理方式很简单，直接在 SpringMVC 的配置文件中，添加如下内容：

```
<mvc:resources mapping="/static/html/**" location="/static/html/"/>
```

公众号·江南一点雨



mapping 表示映射规则，也是拦截规则，就是说，如果请求地址是 /static/html 这样的格式的话，那么对应的资源就去 /static/html/ 这个目录下查找。

在映射路径的定义中，最后是两个 *，这是一种 Ant 风格的路径匹配符号，一共有三个通配符：

通配符	含义
**	匹配多层路径
*	匹配一层路径
?	匹配任意单个字符

一个比较原始的配置方式可能如下：

```
<mvc:resources mapping="/static/html/**" location="/static/html/"/>
<mvc:resources mapping="/static/js/**" location="/static/js/"/>
<mvc:resources mapping="/static/css/**" location="/static/css/"/>
```

但是，由于 ** 可以表示多级路径，所以，以上配置，我们可以进行简化：

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

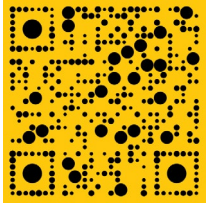
15. 拦截器

SpringMVC 中的拦截器，相当于 Jsp/Servlet 中的过滤器，只不过拦截器的功能更为强大。

拦截器的定义非常容易：

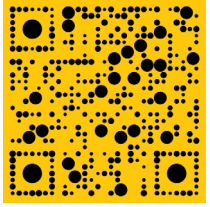
```
@Component
public class MyInterceptor1 implements HandlerInterceptor {
    /**
     * 这个是请求预处理的方法，只有当这个方法返回值为 true 的时候，后面的方法才会执行
     * @param request
     * @param response
     * @param handler
```

扫码关注微信公众号【江南一点雨】，回复 2TB，获取超 2TB 免费 Java 学习资源



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

    public void postHandle(HttpServletRequest request, HttpServletRes
ponse response, Object handler, ModelAndView modelAndView) throws Exc
eption {
        System.out.println("MyInterceptor1:postHandle");

    }

    public void afterCompletion(HttpServletRequest request, HttpServl
etResponse response, Object handler, Exception ex) throws Exception {
        System.out.println("MyInterceptor1:afterCompletion");

    }
}
@Component
public class MyInterceptor2 implements HandlerInterceptor {
    /**
     * 这个是请求预处理的方法，只有当这个方法返回值为 true 的时候，后面的方
法才会执行
     * @param request
     * @param response
     * @param handler
     * @return
     * @throws Exception
     */
    public boolean preHandle(HttpServletRequest request, HttpServletR
esponse response, Object handler) throws Exception {
        System.out.println("MyInterceptor2:preHandle");
        return true;
    }

    public void postHandle(HttpServletRequest request, HttpServletRes
ponse response, Object handler, ModelAndView modelAndView) throws Exc
eption {
        System.out.println("MyInterceptor2:postHandle");
```

公众号·江南一点雨



```
    }

    public void afterCompletion(HttpServletRequest request, HttpServl
etResponse response, Object handler, Exception ex) throws Exception {
        System.out.println("MyInterceptor2:afterCompletion");

    }
}
```

拦截器定义好之后，需要在 SpringMVC 的配置文件中进行配置：

```
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/**"/>
        <ref bean="myInterceptor1"/>
    </mvc:interceptor>
    <mvc:interceptor>
        <mvc:mapping path="/**"/>
        <ref bean="myInterceptor2"/>
    </mvc:interceptor>
</mvc:interceptors>
```

如果存在多个拦截器，拦截规则如下：

- preHandle 按拦截器定义顺序调用
- postHandler 按拦截器定义逆序调用
- afterCompletion 按拦截器定义逆序调用
- postHandler 在拦截器链内所有拦截器返成功调用
- afterCompletion 只有 preHandle 返回 true 才调用

公众号：江南一点雨