

[首页 \(/\)](#) / [文章 \(/tutorials\)](#) / [Spring MVC快速入门教程](#)[参考代码 \(/tutorial/spring-mvc-quickstart/repo\)](#)

## Spring MVC快速入门教程

Ricky (/user/luoruici) 发布于 8月7日 10:评论 22791浏览

[java \(/tag/java/tutorials\)](#)[spring-mvc \(/tag/spring-mvc/tutorials\)](#)[springboot \(/tag/springboot/tutorials\)](#)

4

16★

今天给大家介绍一下Spring MVC，让我们学习一下如何利用Spring MVC快速的搭建一个简单的web应用。

更深入地学习Spring MVC，请大家参考 Spring MVC实战入门训练 (<https://course.tianmaying.com/spring-mvc>)。

参考代码请戳右上角，下载下来后可以在Eclipse或者IntelliJ中导入为一个Maven项目。

## 环境准备

- 一个称手的文本编辑器（例如Vim、Emacs、Sublime Text）或者IDE（Eclipse、Idea IntelliJ）
- Java环境（JDK 1.7或以上版本）
- Maven 3.0+（Eclipse和Idea IntelliJ内置，如果使用IDE并且不使用命令行工具可以不安装）

## 一个最简单的Web应用

使用Spring Boot框架可以大大加速Web应用的开发过程，首先在Maven项目依赖中引入 `spring-boot-starter-web`：

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.tianmaying</groupId>
    <artifactId>spring-web-demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>spring-web-demo</name>
    <description>Demo project for Spring WebMvc</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.2.5.RELEASE</version>
        <relativePath/>
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

接下来创建 `src/main/java/com.tmy.Application.java` :

```
package com.tmy;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@RestController
public class Application {

    @RequestMapping("/")
    public String greeting() {
        return "Hello World!";
    }

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

运行应用: `mvn spring-boot:run` 或在IDE中运行 `main()` 方法, 在浏览器中访问 `http://localhost:8080` (`http://localhost:8080`), `Hello World!` 就出现在了页面中。只用了区区十几行Java代码, 一个Hello World应用就可以正确运行了, 那么这段代码究竟做了什么呢? 我们从程序的入口 `SpringApplication.run(Application.class, args);` 开始分析:

1. `SpringApplication` 是Spring Boot框架中描述Spring应用的类, 它的 `run()` 方法会创建一个Spring应用上下文 (Application Context)。另一方面它会扫描当前应用类路径上的依赖, 例如本例中发现 `spring-webmvc` (由 `spring-boot-starter-web` 传递引入) 在类路径中, 那么Spring Boot会判断这是一个Web应用, 并启动一个内嵌的Servlet容器 (默认是Tomcat) 用于处理HTTP请求。
2. Spring WebMvc框架会将Servlet容器里收到的HTTP请求根据路径分发给对应的 `@Controller` 类进行处理, `@RestController` 是一类特殊的 `@Controller`, 它的返回值直接作为HTTP Response的Body部分返回给浏览器。
3. `@RequestMapping` 注解表明该方法处理那些URL对应的HTTP请求, 也就是我们常说的URL路由 (routing), 请求的分发工作是有Spring完成的。例如上面的代码中 `http://localhost:8080/` (`http://localhost:8080/`) 根路径就被路由至 `greeting()` 方法进行处理。如果访问 `http://localhost:8080/hello` (`http://localhost:8080/hello`), 则会出现 404 Not Found 错误, 因为我们并没有编写任何方法来处理 `/hello` 请求。

## 使用 `@Controller` 实现URL路由

现代Web应用往往包括很多页面, 不同的页面也对应着不同的URL。对于不同的URL, 通常需要不同的方法进行处理并返回不同的内容。

### 匹配多个URL

```
@RestController
public class Application {

    @RequestMapping("/")
    public String index() {
        return "Index Page";
    }

    @RequestMapping("/hello")
    public String hello() {
        return "Hello World!";
    }
}
```

`@RequestMapping` 可以注解 `@Controller` 类:

```
@RestController
@RequestMapping("/classPath")
public class Application {
    @RequestMapping("/methodPath")
    public String method() {
        return "mapping url is /classPath/methodPath";
    }
}
```

method 方法匹配的URL是 /classPath/methodPath" 。

提示

可以定义多个 @Controller 将不同URL的处理方法分散在不同的类中。

## URL中的变量——PathVariable

在Web应用中URL通常不是一成不变的，例如微博两个不同用户的个人主页对应两个不同的URL: http://weibo.com/user1 (http://weibo.com/user1) 和 http://weibo.com/user2。(http://weibo.com/user2。) 我们不可能对于每一个用户都编写一个被 @RequestMapping 注解的方法来处理其请求，Spring MVC提供了一套机制来处理这种情况：

```
@RequestMapping("/users/{username}")
public String userProfile(@PathVariable("username") String username) {
    return String.format("user %s", username);
}

@RequestMapping("/posts/{id}")
public String post(@PathVariable("id") int id) {
    return String.format("post %d", id);
}
```

在上述例子中，URL中的变量可以用 {variableName} 来表示，同时在方法的参数中加上 @PathVariable("variableName")，那么当请求被转发给该方法处理时，对应的URL中的变量会被自动赋值给被 @PathVariable 注解的参数（能够自动根据参数类型赋值，例如上例中的 int）。

## 支持HTTP方法

对于HTTP请求除了其URL，还需要注意它的方法（Method）。例如我们在浏览器中访问一个页面通常是GET方法，而表单的提交一般是POST方法。@Controller 中的方法同样需要对其进行区分：

```
@RequestMapping(value = "/login", method = RequestMethod.GET)
public String loginGet() {
    return "Login Page";
}

@RequestMapping(value = "/login", method = RequestMethod.POST)
public String loginPost() {
    return "Login Post Request";
}
```

Spring MVC最新的版本中提供了一种更加简洁的配置HTTP方法的方式，增加了四个标注：

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

在Web应用中常用的HTTP方法有四种：

- PUT方法用来添加的资源
- GET方法用来获取已有的资源
- POST方法用来对资源进行状态转换
- DELETE方法用来删除已有的资源

这四个方法可以对应到CRUD操作（Create、Read、Update和Delete），比如博客的创建操作，按照REST风格设计URL就应该使用PUT方法，读取博客使用GET方法，更新博客使用POST方法，删除博客使用DELETE方法。

每一个Web请求都是属于其中一种，在Spring MVC中如果不特殊指定的话，默认是GET请求。

比如 @RequestMapping("/") 和 @RequestMapping("/hello") 和对应的Web请求是：

- GET /
- GET /hello

实际上 `@RequestMapping("/")` 是 `@RequestMapping("/", method = RequestMethod.GET)` 的简写，即可以通过 `method` 属性，设置请求的HTTP方法。

比如PUT /hello 请求，对应于 `@RequestMapping("/hello", method = RequestMethod.PUT)`

基于新的标注 `@RequestMapping("/hello", method = RequestMethod.PUT)` 可以简写为 `@PutMapping("/hello")`。`@RequestMapping("/hello")` 与 `GetMapping("/hello")` 等价。

## 模板渲染

在之前所有的 `@RequestMapping` 注解的方法中，返回值字符串都被直接传送到浏览器端并显示给用户。但是为了能够呈现更加丰富、美观的页面，我们需要将HTML代码返回给浏览器，浏览器再进行页面的渲染、显示。

一种很直观的方法是在处理请求的方法中，直接返回HTML代码，但是这样做的问题在于——一个复杂的页面HTML代码往往也非常复杂，并且嵌入在Java代码中十分不利于维护。更好的做法是将页面的HTML代码写在模板文件中，渲染后再返回给用户。为了能够进行模板渲染，需要将 `@RestController` 改成 `@Controller`：

```
import org.springframework.ui.Model;

@Controller
public class HelloController {

    @RequestMapping("/hello/{name}")
    public String hello(@PathVariable("name") String name, Model model) {
        model.addAttribute("name", name);
        return "hello"
    }
}
```

在上述例子中，返回值 "hello" 并非直接将字符串返回给浏览器，而是寻找名字为 hello 的模板进行渲染，我们使用 Thymeleaf (<http://www.thymeleaf.org>) 模板引擎进行模板渲染，需要引入依赖：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

接下来需要在默认的模板文件夹 `src/main/resources/templates/` 目录下添加一个模板文件 `hello.html`：

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <title>Getting Started: Serving Web Content</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
  <p th:text="'Hello, ' + ${name} + '!'" />
</body>
</html>
```

`th:text="'Hello, ' + ${name} + '!'"` 也就是将我们之前在 `@Controller` 方法里添加至 `Model` 的属性 `name` 进行渲染，并放入 `<p>` 标签中（因为 `th:text` 是 `<p>` 标签的属性）。模板渲染还有更多的用法，请参考 Thymeleaf 官方文档 (<http://www.thymeleaf.org/doc/tutorials/2.1/usingthymeleaf.html>)。

## 处理静态文件

浏览器页面使用HTML作为描述语言，那么必然也脱离不了CSS以及JavaScript。为了能够浏览器能够正确加载类似 `/css/style.css`，`/js/main.js` 等资源，默认情况下我们只需要在 `src/main/resources/static` 目录下添加 `css/style.css` 和 `js/main.js` 文件后，Spring MVC能够自动将他们发布，通过访问 `/css/style.css`，`/js/main.js` 也就可以正确加载这些资源。

## 文件上传

Spring MVC还能够支持更为复杂的HTTP请求——文件资源。我们在网站中经常遇到上传图片、附件一类的需求，就是通过文件上传技术来实现的。

处理文件的表单和普通表单的唯一区别在于设置 `enctype` ——`multipart`编码方式则需要设置 `enctype` 为 `multipart/form-data`。

```
<form method="post" enctype="multipart/form-data">
  <input type="text" name="title" value="tianmaying">
  <input type="file" name="avatar">
  <input type="submit">
</form>
```

这里我们还设置了 `<input type='text'>` 的默认值为 `tianmaying`。

该表单将会显示为一个文本框、一个文件按钮、一个提交按钮。然后我们选择一个文件：`chrome.png`，点击表单提交后产生的请求可能是这样的：

请求头：

```
POST http://www.example.com HTTP/1.1
Content-Type:multipart/form-data; boundary=----WebKitFormBoundaryrGKCBY7qhFd3TrwA
```

请求体：

```
-----WebKitFormBoundaryrGKCBY7qhFd3TrwA
Content-Disposition: form-data; name="title"

tianmaying
-----WebKitFormBoundaryrGKCBY7qhFd3TrwA
Content-Disposition: form-data; name="avatar"; filename="chrome.png"
Content-Type: image/png

... content of chrome.png ...
-----WebKitFormBoundaryrGKCBY7qhFd3TrwA--
```

这便是个`multipart`编码的表单。`Content-Type` 中还包含了 `boundary` 的定义，它用来分隔请求体中的每个字段。正是这一机制，使得请求体中可以包含二进制文件（当然文件中不能包含 `boundary`）。文件上传正是利用这种机制来完成的。

如果不设置 `<form>` 的 `enctype` 编码，同样可以在表单中设置 `type=file` 类型的输入框，但是请求体和传统的表单一样，这样服务器程序无法获取真正的文件内容。

在服务端，为了支持文件上传我们还需要进行一些配置。

## 控制器逻辑

对于表单中的文本信息输入，我们可以通过 `@RequestParam` 注解获取。对于上传的二进制文件（文本文件同样会转化为 `byte[]` 进行传输），就需要借助Spring提供的 `MultipartFile` 类来获取了：

```
@Controller
public class FileUploadController {

    @PostMapping("/upload")
    @ResponseBody
    public String handleFileUpload(@RequestParam("file") MultipartFile file) {
        byte[] bytes = file.getBytes();

        return "file uploaded successfully."
    }
}
```

通过 `MultipartFile` 的 `getBytes()` 方法即可以得到上传的文件内容（`<form>` 中定义了一个 `type="file"` 的，在这里我们可以将它保存到本地磁盘。另外，在默认的情况下Spring仅仅支持大小为128KB的文件，为了调整它，我们可以修改Spring的配置文件 `src/main/resources/application.properties`：

```
multipart.maxFileSize: 128KB
multipart.maxRequestSize: 128KB
```

修改上述数值即可完成配置。

## HTML表单

HTML中支持文件上传的表单元素仍然是 `<input>`，只不过它的类型是 `file`：

```
<html>
<body>
  <form method="POST" enctype="multipart/form-data" action="/upload">
    File to upload: <input type="file" name="file"><br />
    Name: <input type="text" name="name"><br />
    <input type="submit" value="Upload"> Press here to upload the file!
  </form>
</body>
</html>
```

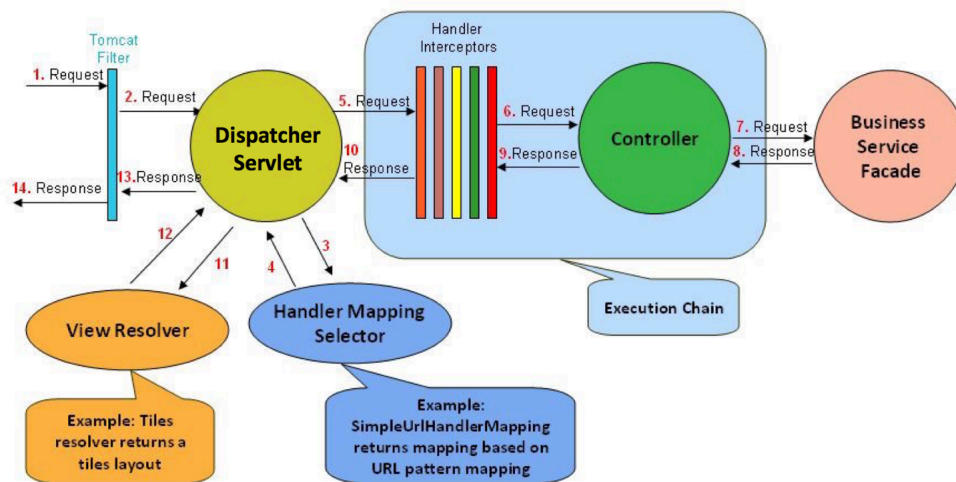
`multipart/form-data` 表单既可以上传文件类型，也可以和普通表单一样提交其他类型的数据，在Spring MVC的 `@RequestMapping` 方法参数中用 `@RequestParam` 标注即可（也可以利用数据绑定机制，绑定一个对象）

## 拦截器Interceptor

Spring MVC框架中的Interceptor，与Servlet API中的Filter十分类似，用于对Web请求进行预处理/后处理。通常情况下这些预处理/后处理逻辑是通用的，可以被应用于所有或多个Web请求，例如：

- 记录Web请求相关日志，可以用于做一些信息监控、统计、分析
- 检查Web请求访问权限，例如发现用户没有登录后，重定向到登录页面
- 打开/关闭数据库连接——预处理时打开，后处理关闭，可以避免在所有业务方法中都编写类似代码，也不会忘记关闭数据库连接

## Spring MVC请求处理流程



天码营

(<http://assets.tianmaying.com/md-image/dbd1017dffce1d08e653d05baa1e4934.png>)

上图是Spring MVC框架处理Web请求的基本流程，请求会经过 `DispatcherServlet` 的分发后，会按顺序经过一系列的 `Interceptor` 并执行其中的预处理方法，在请求返回时同样会执行其中的后处理方法。

在 `DispatcherServlet` 和 `Controller` 之间哪些竖着的彩色细条，是拦截请求进行额外处理的地方，所以命名为**拦截器 (Interceptor)**。

## HandlerInterceptor接口

Spring MVC中拦截器是实现了 `HandlerInterceptor` 接口的Bean：

```
public interface HandlerInterceptor {
    boolean preHandle(HttpServletRequest request,
                      HttpServletResponse response,
                      Object handler) throws Exception;

    void postHandle(HttpServletRequest request,
                    HttpServletResponse response,
                    Object handler, ModelAndView modelAndView) throws Exception;

    void afterCompletion(HttpServletRequest request,
                        HttpServletResponse response,
                        Object handler, Exception ex) throws Exception;
}
```

- `preHandle()`：预处理回调方法，若方法返回值为 `true`，请求继续（调用下一个拦截器或处理器方法）；若方法返回值为 `false`，请求处理流程中断，不会继续调用其他的拦截器或处理器方法，此时需要通过 `response` 产生响应；
- `postHandle()`：后处理回调方法，实现处理器的后处理（但在渲染视图之前），此时可以通过 `ModelAndView` 对模型数据进行处理或对视图进行处理
- `afterCompletion()`：整个请求处理完毕回调方法，即在视图渲染完毕时调用

`HandlerInterceptor` 有三个方法需要实现，但大部分时候可能只需要实现其中的一个方法，`HandlerInterceptorAdapter` 是一个实现了 `HandlerInterceptor` 的抽象类，它的三个实现方法都为空实现（或者返回 `true`），继承该抽象类后可以仅仅实现其中的一个方法：

```
public class Interceptor extends HandlerInterceptorAdapter {

    public boolean preHandle(HttpServletRequest request,
                            HttpServletResponse response,
                            Object handler) throws Exception {
        // 在controller方法调用前打印信息
        System.out.println("This is interceptor.");
        // 返回true，将请求继续传递（传递到下一个拦截器，没有其它拦截器了，则传递给Controller）
        return true;
    }
}
```

## 配置Interceptor

定义 `HandlerInterceptor` 后，需要创建 `WebMvcConfigurerAdapter` 在MVC配置中将它们应用于特定的URL中。一般一个拦截器都是拦截特定的某一部分请求，这些请求通过URL模型来指定。

下面是一个配置的例子：

```
@Configuration
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new LocaleInterceptor());
        registry.addInterceptor(new ThemeInterceptor()).addPathPatterns("/**").excludePathPatterns(
            registry.addInterceptor(new SecurityInterceptor()).addPathPatterns("/secure/*");
    }
}
```

## @ModelAttribute

### 方法使用@ModelAttribute标注

`@ModelAttribute` 标注可被应用在方法或方法参数上。

标注在方法上的 `@ModelAttribute` 说明方法是用于添加一个或多个属性到model上。这样的方法能接受与 `@RequestMapping` 标注相同的参数类型，只不过不能被映射到具体的请求上。

在同一个控制器中，标注了 `@ModelAttribute` 的方法实际上会在 `@RequestMapping` 方法之前被调用。

以下是示例：

环境准备

一个最简单的Web应用

使用@Controller实现URL路由

模板渲染

处理静态文件

文件上传

拦截器Interceptor

Spring MVC请求处理流程

HandlerInterceptor接口

配置Interceptor

@ModelAttribute

异常处理

进一步阅读

Python入门基础课程  
(<https://course.tianmaying.com/basic>)

将Python的入门基础知识贯穿在简单易懂的实例中，代码闯关，名师指导，同学帮助，逐步深入，帮助你快...

Java入门基础教程  
(<https://course.tianmaying.com/basic>)

将Java的入门基础知识贯穿在简单易懂的实例中，写代码闯关，名师指导，逐步深入，帮助你快速进入Jav...

打造功能完整的博客系统：

```
// Add one attribute
// The return value of the method is added to the model under the name "account"
// You can customize the name via @ModelAttribute("myAccount")

@ModelAttribute
public Account addAccount(@RequestParam String number) {
    return accountManager.findAccount(number);
}

// Add multiple attributes

@ModelAttribute
public void populateModel(@RequestParam String number, Model model) {
    model.addAttribute(accountManager.findAccount(number));
    // add more ...
}
```

`@ModelAttribute` 方法通常被用来填充一些公共需要的属性或数据，比如一个下拉列表所预设的几种状态，或者宠物的几种类型，或者去取得一个HTML表单渲染所需要的命令对象，比如 `Account` 等。

`@ModelAttribute` 标注方法有两种风格：

- 在第一种写法中，方法通过返回值的方式默认地将添加一个属性；
- 在第二种写法中，方法接收一个 `Model` 对象，然后可以向其中添加任意数量的属性。

可以在根据需要，在两种风格中选择合适的一种。

一个控制器可以拥有多个 `@ModelAttribute` 方法。同个控制器内的所有这些方法，都会在 `@RequestMapping` 方法之前被调用。

`@ModelAttribute` 方法也可以定义在 `@ControllerAdvice` 标注的类中，并且这些 `@ModelAttribute` 可以同时许多控制器生效。

属性名没有被显式指定的时候又当如何呢？在这种情况下，框架将根据属性的类型给予一个默认名称。举个例子，若方法返回一个 `Account` 类型的对象，则默认的属性名为“account”。可以通过设置 `@ModelAttribute` 标注的值来改变默认值。当向 `Model` 中直接添加属性时，请使用合适的重载方法 `addAttribute(..)` -即带或不带属性名的方法。

`@ModelAttribute` 标注也可以被用在 `@RequestMapping` 方法上。这种情况下，`@RequestMapping` 方法的返回值将会被解释为model的一个属性，而非一个视图名，此时视图名将以视图命名约定来方式来确定。

## 方法参数使用@ModelAttribute标注

`@ModelAttribute` 标注既可以被用在方法上，也可以被用在方法参数上。

标注在方法参数上的 `@ModelAttribute` 说明了该方法参数的值将由model中取得。如果model中找不到，那么该参数会先被实例化，然后被添加到model中。在model中存在以后，请求中所有名称匹配的参数都会填充到该参数中。

这在 Spring MVC (<https://www.tianmaying.com/tutorial/spring-mvc-quickstart>) 中被称为数据绑定，一个非常有用的特性，我们不用每次都手动从表格数据中转换这些字段数据。

```
@RequestMapping(path = "/owners/{ownerId}/pets/{petId}/edit", method = RequestMethod.POST)
public String processSubmit(@ModelAttribute Pet pet) { }
```

上面的代码为例，这个 `Pet` 类型的实例可能来自哪里呢？有几种可能：

- 它可能因为 `@SessionAttributes` 标注的使用已经存在于model中
- 它可能因为在同个控制器中使用了 `@ModelAttribute` 方法已经存在于model中——正如上一小节所叙述的
- 它可能是由URI模板变量和类型转换中取得的（下面会详细讲解）
- 它可能是调用了自身的默认构造器被实例化出来的

`@ModelAttribute` 方法常用于从数据库中取一个属性值，该值可能通过 `@SessionAttributes` 标注在请求中间传递。在一些情况下，使用URI模板变量和类型转换的方式来取得一个属性是更方便的方式。这里有个例子：

```
@RequestMapping(path = "/accounts/{account}", method = RequestMethod.PUT)
public String save(@ModelAttribute("account") Account account) {

}
```

这个例子中，`model` 属性的名称（“account”）与URI模板变量的名称相匹配。如果配置了一个可以将 `String` 类型的账户值转换成 `Account` 类型实例的转换器 `Converter<String, Account>`，那么上面这段代码就可以工作的很好，而不需要再额外写一个 `@ModelAttribute` 方法。

Spring MVC实战入门  
(<https://course.tianmaying.com/mvc>)

Spring MVC实战入门训练，以一个博客系统为例，将Spring MVC的核心知识融入到实战当中...

Servlet/JSP实战教程：搭建博客系统  
(<https://course.tianmaying.com/and-jsp>)

可能是最简单易学的Servlet和JSP开发入门教程，结合开发实例，深入浅出地介绍学习Servlet...

反馈意见 ^



下一步就是数据的绑定。WebDataBinder 类能将请求参数——包括字符串的查询参数和表单字段等——通过名称匹配到model的属性上。成功匹配的字段在需要的时候会进行一次类型转换（从String类型到目标字段的类型），然后被填充到model对应的属性中。

进行了数据绑定后，则可能会出现一些错误，比如没有提供必须的字段、类型转换过程的错误等。若想检查这些错误，可以在标注了 @ModelAttribute 的参数紧跟着声明一个 BindingResult 参数：

```
@RequestMapping(path = "/owners/{ownerId}/pets/{petId}/edit", method = RequestMethod.POST)
public String processSubmit(@ModelAttribute("pet") Pet pet, BindingResult result) {
    if (result.hasErrors()) {
        return "petForm";
    }

    // ...
}
```

拿到 BindingResult 参数后，可以检查是否有错误，可以通过Spring的 <errors> 表单标签来在同一个表单上显示错误信息。

BindingResult 被用于记录数据绑定过程的错误，因此除了数据绑定外，还可以把该对象传给自己定制的验证器来调用验证。这使得数据绑定过程和验证过程出现的错误可以被搜集到一起，然后一并返回给用户：

```
@RequestMapping(path = "/owners/{ownerId}/pets/{petId}/edit", method = RequestMethod.POST)
public String processSubmit(@ModelAttribute("pet") Pet pet, BindingResult result) {

    new PetValidator().validate(pet, result);
    if (result.hasErrors()) {
        return "petForm";
    }

    // ...
}
```

又或者可以通过添加一个JSR-303规范的 @Valid 标注，这样验证器会自动被调用。

```
@RequestMapping(path = "/owners/{ownerId}/pets/{petId}/edit", method = RequestMethod.POST)
public String processSubmit(@Valid @ModelAttribute("pet") Pet pet, BindingResult result) {

    if (result.hasErrors()) {
        return "petForm";
    }

    // ...
}
```

## 异常处理

Spring MVC框架提供了多种机制用来处理异常，初次接触可能会对他们用法以及适用的场景感到困惑。现在以一个简单例子来解释这些异常处理的机制。

假设现在我们开发了一个博客应用，其中最重要的资源就是文章（Post），应用中的URL设计如下：

- 获取文章列表：GET /posts/
- 添加一篇文章：POST /posts/
- 获取一篇文章：GET /posts/{id}
- 更新一篇文章：PUT /posts/{id}
- 删除一篇文章：DELETE /posts/{id}

这是非常标准的复合RESTful风格的URL设计，在Spring MVC实现的应用过程中，相应也会有5个对应的用 @RequestMapping 注解的方法来处理相应的URL请求。在处理某一篇文章的请求中（获取、更新、删除），无疑需要做这样一个判断——请求URL中的文章id是否存在于系统中，如果不存在需要返回 404 Not Found 。

## 使用HTTP状态码

在默认情况下，Spring MVC处理Web请求时如果发现存在没有应用代码捕获的异常，那么会返回HTTP 500（Internal Server Error）错误。但是如果该异常是我们自己定义的并且使用 @ResponseStatus 注解进行修饰，那么Spring MVC则会返回指定的HTTP状态码：

```
@ResponseStatus(value = HttpStatus.NOT_FOUND, reason = "No Such Post")//404 Not Found
public class PostNotFoundException extends RuntimeException {
}
```

在 Controller 中可以这样使用它:

```
@RequestMapping(value = "/posts/{id}", method = RequestMethod.GET)
public String showPost(@PathVariable("id") long id, Model model) {
    Post post = postService.get(id);
    if (post == null) throw new PostNotFoundException("post not found");
    model.addAttribute("post", post);
    return "postDetail";
}
```

这样如果我们访问了一个不存在的文章, 那么Spring MVC会根据抛出的 `PostNotFoundException` 上的注解值返回一个HTTP 404 Not Found给浏览器。

## 最佳实践

上述场景中, 除了获取一篇文章的请求, 还有更新和删除一篇文章的方法中都需要判断文章id是否存在。在每一个方法中都加上 `if (post == null) throw new PostNotFoundException("post not found");` 是一种解决方案, 但如果有10个、20个包含 `/posts/{id}` 的方法, 虽然只有一行代码但让他们重复10次、20次也是非常不优雅的。

为了解决这个问题, 可以将这个逻辑放在Service中实现:

```
@Service
public class PostService {

    @Autowired
    private PostRepository postRepository;

    public Post get(long id) {
        return postRepository.findById(id)
            .orElseThrow(() -> new PostNotFoundException("post not found"));
    }
}
```

这里 `PostRepository` 继承了 `JpaRepository`, 可以定义 `findById` 方法返回一个 `Optional<Post>` —如果不存在则Opt...

这样在所有的 Controller 方法中, 只需要正常获取文章即可, 所有的异常处理都交给了Spring MVC。

## 在 Controller 中处理异常

Controller 中的方法除了可以用于处理Web请求, 还能够用于处理异常处理——为它们加上 `@ExceptionHandler` 即可:

```

@Controller
public class ExceptionHandlingController {

    // @RequestHandler methods
    ...

    // Exception handling methods

    // Convert a predefined exception to an HTTP Status code
    @ResponseStatus(value=HttpStatus.CONFLICT, reason="Data integrity violation") // 409
    @ExceptionHandler(DataIntegrityViolationException.class)
    public void conflict() {
        // Nothing to do
    }

    // Specify the name of a specific view that will be used to display the error:
    @ExceptionHandler({SQLException.class, DataAccessException.class})
    public String databaseError() {
        // Nothing to do. Returns the logical view name of an error page, passed to
        // the view-resolver(s) in usual way.
        // Note that the exception is _not_ available to this view (it is not added to
        // the model) but see "Extending ExceptionHandlerExceptionHandlerResolver" below.
        return "databaseError";
    }

    // Total control - setup a model and return the view name yourself. Or consider
    // subclassing ExceptionHandlerExceptionHandlerResolver (see below).
    @ExceptionHandler(Exception.class)
    public ModelAndView handleError(HttpServletRequest req, Exception exception) {
        logger.error("Request: " + req.getRequestURL() + " raised " + exception);

        ModelAndView mav = new ModelAndView();
        mav.addObject("exception", exception);
        mav.addObject("url", req.getRequestURL());
        mav.setViewName("error");
        return mav;
    }
}

```

首先需要明确的一点是，在 Controller 方法中的 @ExceptionHandler 方法只能够处理同一个 Controller 中抛出的异常。这些方法上同时也可以继续使用 @ResponseStatus 注解用于返回指定的HTTP状态码，但同时还能够支持更加丰富的异常处理：

- 渲染特定的视图页面
- 使用 ModelAndView 返回更多的业务信息

大多数网站都会使用一个特定的页面来响应这些异常，而不是直接返回一个HTTP状态码或者显示Java异常调用栈。当然异常信息对于开发人员是非常有用的，如果想要在视图中直接看到它们可以这样渲染模板（以JSP为例）：

```

<h1>Error Page</h1>
<p>Application has encountered an error. Please contact support on ...</p>

<!--
Failed URL: ${url}
Exception: ${exception.message}
<c:forEach items="${exception.stackTrace}" var="ste">    ${ste}
</c:forEach>
-->

```

## 全局异常处理

@ControllerAdvice (<http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#mvc-ann-controller-advice>) 提供了和上一节一样的异常处理能力，但是可以被应用于Spring应用上下文中的所有 @Controller：

```

@ControllerAdvice
class GlobalControllerExceptionHandler {
    @ResponseStatus(HttpStatus.CONFLICT) // 409
    @ExceptionHandler(DataIntegrityViolationException.class)
    public void handleConflict() {
        // Nothing to do
    }
}

```

Spring MVC默认对于没有捕获也没有被 `@ResponseStatus` 以及 `@ExceptionHandler` 声明的异常，会直接返回500，这显然并不友好，可以在 `@ControllerAdvice` 中对其进行处理（例如返回一个友好的错误页面，引导用户返回正确的位置或者提交错误信息）：

```
@ControllerAdvice
class GlobalExceptionHandler {
    public static final String DEFAULT_ERROR_VIEW = "error";

    @ExceptionHandler(value = Exception.class)
    public ModelAndView defaultErrorHandler(HttpServletRequest req, Exception e) throws Exception {
        // If the exception is annotated with @ResponseStatus rethrow it and let
        // the framework handle it - like the OrderNotFoundException example
        // at the start of this post.
        // AnnotationUtils is a Spring Framework utility class.
        if (AnnotationUtils.findAnnotation(e.getClass(), ResponseStatus.class) != null)
            throw e;

        // Otherwise setup and send the user to a default error-view.
        ModelAndView mav = new ModelAndView();
        mav.addObject("exception", e);
        mav.addObject("url", req.getRequestURL());
        mav.setViewName(DEFAULT_ERROR_VIEW);
        return mav;
    }
}
```

## 总结

Spring在异常处理方面提供了一如既往的强大特性和支持，那么在应用开发中我们应该如何使用这些方法呢？以下提供一些经验性的准则：

- 不要在 `@Controller` 中自己进行异常处理逻辑。即使它只是一个Controller相关的特定异常，在 `@Controller` 中添加一个 `@ExceptionHandler` 方法处理。
- 对于自定义的异常，可以考虑对其加上 `@ResponseStatus` 注解
- 使用 `@ControllerAdvice` 处理通用异常（例如资源不存在、资源存在冲突等）

## 进一步阅读

- Spring MVC实战入门训练 (<https://course.tianmaying.com/spring-mvc>)
- 开发问答网站：基于MyBatis和Spring MVC (<https://course.tianmaying.com/spring-mvc-mybatis-qa>)
- Spring MVC DispatcherServlet详解 (<https://www.tianmaying.com/tutorial/spring-mvc-DispatcherServlet>)
- 基于Spring MVC的 Websocket在线聊天室 (<https://www.tianmaying.com/tutorial/websocket-chatroom>)
- 整合Spring Data JPA与Spring MVC: 分页和排序 (<https://www.tianmaying.com/tutorial/spring-jpa-page-sort>)
- 整合Spring Data JPA与Spring MVC: 使用@Query标注自定义查询语句 (<https://www.tianmaying.com/tutorial/spring-jpa-query>)
- 基于Spring和Spring MVC实现可跨域访问的REST服务 (<https://www.tianmaying.com/tutorial/cross-origin-rest-service>)
- Spring MVC拦截器 (<https://www.tianmaying.com/tutorial/spring-mvc-interceptor>)
- Spring MVC中使用Thymeleaf模板引擎 (<https://www.tianmaying.com/tutorial/spring-mvc-thymeleaf>)

### 版权声明

本文由Ricky (/user/luoruici)创作，转载需署名作者且注明文章出处

### 参考代码

要获取本文的参考代码，请访问：<https://www.tianmaying.com/tutorial/spring-mvc-quickstart/repo> (/tutorial/spring-mvc-quickstart/repo)

相关文章

基于Spring的QQ第三方登录实现 (/tutorial/OAuth-login-...)  
基于Spring的微信第三方登录实现 (/tutorial/OAuth-login-...)  
基于Spring的新浪微博第三方登录实现 (/tutorial/OAuth-l...  
基于Spring提供支持不同设备的页面 (/tutorial/content-fo...  
Spring Boot应用开发初探与示例 (/tutorial/spring-boot-i...  
基于Spring Boot为关系型数据库构建REST访问接口 (/tu...  
部署Spring Boot应用 (/tutorial/deploy-spring-boot-appli...  
测试Spring MVC应用 (/tutorial/spring-mvc-testing)  
在Docker容器中运行Spring Boot应用 (/tutorial/spring-b...  
Spring Boot — 开发新一代Spring Java应用 (/tutorial/s...

其他文章

Spring的AOP原理 (/tutorial/spring-aop)  
LeetCode题解 #5 Longest Palindromic Substring (/tutor...  
LeetCode题解 #22 Generate Parentheses (/tutorial/LC22)  
LeetCode题解 #1 Two Sum (/tutorial/LC1)  
环信首席架构师梁宇鹏谈编程与艺术 (/tutorial/techmuse...  
基于Spring的新浪微博第三方登录实现 (/tutorial/OAuth-l...  
Spring MVC中创建URL路由 (/tutorial/spring-mvc-routing)  
MySQL索引创建、效率测试 (/tutorial/mysql)  
C++类的静态和常量成员 (/tutorial/cpp-static-and-const...  
Ubuntu 14.04 搭建Nexus Maven 私服 (/tutorial/maven)


评论

向作者提问 (/qas/create?respondent=luoruici&tutorial=8ab3eda84f02a10f14f03fe82010035)


登录发表评论 登录

(http://v.t.sina.com.cn/share/share.php?title=%E3%80%90Spring+MVC%E5%BF%AB%E9%80%9F%E5%AE%9E%E6%88%98%E5%BC%80%E5%8F%91%E6%+%3Cp%3E%E4%BB%8A%E5%A4%A9%E7%BB%99%E5%/mvc%22%3ESpring+MVC%E5%AE%9E%E6%88%98%E5%8Eboot-starter-web%2Fcode%3E%EF%BC%9A%3C%2Fp%3E%3Cp%38%22%3E%26gt%3EByinghttps%3A%2F%2Fwww.tianmaying.com%2F%2Fspring-mvc-mvc-quickstart&app\_id=1502829444)

注册 (/account/register?next=%2Ftutorial%2Fspring-mvc-quickstart)

 David (/user/david) 于 1月4日 (/tutorial/spring-mvc-quickstart/comments/1333) @nethub (/user/nethub)

谢谢提醒，或者给 Application 加上 @ComponentScan 标注，指定需要扫描的包。  
参考代码中是放在一个包里的，见 SpringMvcQuickstartApplication (https://www.tianmaying.com/tutorial/spring-mvc-quickstart/repo/blob/master/src/main/java/tmy/demo/SpringMvcQuickstartApplication.java)


 nethub (/user/nethub) 于 1月4日 (/tutorial/spring-mvc-quickstart/comments/1332) (/user/nethub) 老师，这个教程可能需要稍微修改下。

因为直接运行会报这个错误：  

Your ApplicationContext is unlikely to start due to a @ComponentScan of the default package

原因是： application.java 文件不能直接放在main/java文件夹下，必须要建一个包把他放进去

 weixiny (/user/weixiny) 于 12月7日 (/tutorial/spring-mvc-quickstart/comments/1276) (/user/weixiny) 非常好的教程，感谢！ @David (/user/david)

 David (/user/david) 于 11月23日 (/tutorial/spring-mvc-quickstart/comments/1187) (/user/david) @cser (/user/webmaster)

看Url中包含的变量名，应该是想实现类似Github代码托管：  
管： https://github.com/pinterest/mysql\_utils/blob/master/lib/backup.py  
(https://github.com/pinterest/mysql\_utils/blob/master/lib/backup.py)  
restOfUri变量中间可能包含slash(/)  
第一种办法使用 \*\* 来匹配不定长的Url：

```
@RequestMapping("/{repoName}/**")
public Map searchWithSearchTerm(HttpServletRequest request, @PathVariable("repoName") String
// Don't repeat a pattern
String pattern = (String)
    request.getAttribute(HandlerMapping.BEST_MATCHING_PATTERN_ATTRIBUTE);

String restOfUrl = new AntPathMatcher().extractPathWithinPattern(pattern,
    request.getServletPath());
...
}
```

第二种方法:

```
@RequestMapping(value =("/{repoName}/{path:.+}", method = RequestMethod.GET)
public Object path(@PathVariable String repoName,
    @PathVariable String path)
```

{path:.+} 表示匹配任意字符, 但是由于Spring MVC默认将/作为Url分隔符, 所以需要更改一下配置:

```
@Configuration
@EnableWebMvc
public class WebMvc extends WebMvcConfigurerAdapter {

    @Override
    public void configurePathMatch(PathMatchConfigurer configurer) {
        UrlPathHelper urlPathHelper = new UrlPathHelper();
        urlPathHelper.setUrlDecode(false);
        configurer.setUrlPathHelper(urlPathHelper);
    }
}
```



cser (/user/webmaster) 于 11月23日 (/tutorial/spring-mvc-quickstart/comments/56)  
 (/user/webmaster) @RequestMapping 中获取 @PathVariable 时, 如何获取一个完整的路径? 举例:

/ {repoName}/{restOfUrl} 应该可以解析 /spring-boot/src/main/java/org/springframework, 其中

```
repoName = spring-boot
restOfUrl = src/main/java/org/springframework
```

其中 restOfUrl 中可能会包含路径分隔符 /



Toder (/user/tiangie) 于 11月21日 (/tutorial/spring-mvc-quickstart/comments/1172)  
 (/user/tiangie) @cser (/user/webmaster) Spring的依赖注入使得我们的代码非常容易进行单元测试

—— @Controller, @Service, @Entity 等注解标注的类基本都是POJO(plain old Java object), 也就是说很少依赖于Spring容器本身的API。我们可以非常容易地使用JUnit (<http://junit.org/>) 或 TestNG (<http://testng.org/>) 编写测试代码。另一方面, 对于三层架构的Spring Web应用 (Controller, Service, DAO), 使用Mock/活Stub方法也能够更好的来测试我们的代码逻辑。例如Service层代码的单元测试中, 依赖的DAO (或Repository)对象都是根据应用测试需求Mock出来的, 而不需要真正去访问数据库。

参考这篇文章 [测试Spring MVC应用 \(https://www.tianmaying.com/tutorial/spring-mvc-testing\)](https://www.tianmaying.com/tutorial/spring-mvc-testing)



cser (/user/webmaster) 于 11月20日 (/tutorial/spring-mvc-quickstart/comments/1090)  
 (/user/webmaster) Spring MVC的应用测试有什么建议? @David (/user/david)



David (/user/david) 于 9月2日 (/tutorial/spring-mvc-quickstart/comments/246)  
 (/user/david) Spring MVC最新的版本中提供了一种更加简洁的配置HTTP方法的方式, 增加了四个标注:

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

对应于Web应用中常用的HTTP方法有四种:

- PUT方法用来添加的资源
- GET方法用来获取已有的资源
- POST方法用来对资源进行状态转换
- DELETE方法用来删除已有的资源



Ricky (/user/luoruici) 于 8月21日 (/tutorial/spring-mvc-quickstart/comments/768)  
(/user/luoruici) 已添加



David (/user/david) 于 8月21日 (/tutorial/spring-mvc-quickstart/comments/767)  
(/user/david) 能把代码贴上来吗？参考代码是空的啊！

Copyright Tianmaying © 2016 | 京ICP备15008133号-1 | TMY-EDU | 231216939 (<http://shang.qq.com/wpa/qunwpa?idkey=6cbe9f8835aa1aa6c62f4bceb585d415b9a7f710c4e7f0b20e424905cf3f7e27>) | 天码营 (<http://weibo.com/u/5623427244>) | 服务条款 (/terms-of-service) | 官方博客 (/blog) | 分享 (/shares) | TCoder (/coders) | 代码 (/snippets) | 友情链接 (/links)