



SPRING BOOT 讲义

王瑞



2019-10-25

天汇智码软件技术有限公司——智码堂软件技术培训中心
南京江北新区星火路 15 号智芯科技大厦 1026

目录

1	Spring Boot 简介	4
2	Spring Boot 基础配置	6
	本章概要	6
2.1	开发环境	6
2.1.1	创建 Spring Boot 项目	7
2.1.2	启动器类	13
2.2	Spring MVC	15
2.2.1	创建控制器	15
2.2.2	启动运行	15
2.3	定制 Banner	17
2.4	Web 容器配置	17
2.5	HTTPS 配置	18
2.5.1	生成证书	19
2.5.2	使用证书	20
2.5.3	在浏览器中查看	22
2.6	Jetty 配置	23
2.7	Undertow 配置	24
2.8	Properties 配置	24
2.8.1	配置属性	26
2.9	YAML 配置	28
2.10	Profile	29
3	SpringBoot 整合视图层技术	31
	本章概要	31
3.1	整合 Thymeleaf	32

3.1.1	配置 Thymeleaf.....	32
3.1.2	配置控制器.....	34
3.1.3	创建视图.....	37
3.1.4	运行.....	38
4	Spring Boot 整合 Web 开发.....	40
	本章概要.....	40
4.1	返回 JSON 数据.....	41
4.1.1	默认实现.....	41
4.1.2	自定义 JSON 转换器 Gson.....	45
4.1.1	自定义 JSON 转换器 Fastjson.....	46
4.2	静态资源的访问.....	49
4.2.1	自定义静态资源访问策略.....	50
4.3	文件上传.....	52
4.3.1	多文件上传.....	59
4.4	@ControllerAdvice.....	60
4.4.1	全局异常处理.....	60
4.4.2	添加全局数据.....	62
4.4.3	请求参数预处理.....	63
4.5	自定义错误页面.....	67
4.5.1	静态犯错误页面.....	67
4.5.2	动态错误页面.....	68
4.6	CORS 支持.....	70
4.6.1	@CrossOrigin.....	71
4.6.2	CORS 全局配置.....	73
4.7	配置类与 XML 配置.....	73
4.8	注册拦截器.....	76

4.9	整合 Servlet、Filter、Listener	77
4.10	路径映射	80
4.11	配置 AOP	81
4.12	自定义欢迎页面	84
4.13	自定义 favicon	84
4.14	除去自动配置	86
5	SpringBoot 整合 NoSQL	87
5.1	整合 Redis	87
5.1.1	Redis 安装	88
5.1.2	Redis 配置	88
5.1.3	SpringBoot 整合 Redis	89
5.1.4	共享 Session	93
6	Spring Boot 整合 MQ	98
6.1	原生 JMS 访问 ActiveMQ	98
6.2	Spring Boot 发送消息	106
6.3	SpringBoot 接收消息	107
7	SpringBoot 发送邮件	110

1 Spring Boot 简介

Spring 诞生时是 Java 企业版 (Java Enterprise Edition, JEE, 也称 J2EE) 的轻量级代替品。无需开发重量级的 Enterprise JavaBean (EJB), Spring 为企业级 Java 开发提供了一种相对简单的方法, 通过依赖注入和面向切面编程, 用简单的 Java 对象 (Plain Old Java Object, POJO) 实现了 EJB 的功能。

虽然 Spring 的组件代码是轻量级的, 但它的配置却是重量级的。

第一阶段: xml 配置

在 Spring 1.x 时代, 使用 Spring 开发满眼都是 xml 配置的 Bean, 随着项目的扩大, 我们需要把 xml 配置文件放到不同的配置文件里, 那时需要频繁的在开发的类和配置文件之间进行切换

第二阶段: 注解配置

在 Spring 2.x 时代, 随着 JDK1.5 带来的注解支持, Spring 提供了声明 Bean 的注解 (例如@Component、@Service), 大大减少了配置量。主要使用的方式是应用的基本配置 (如数据库配置) 用 xml, 业务配置用注解

第三阶段: java 配置

Spring 3.0 引入了基于 Java 的配置能力, 这是一种类型安全的可重构配置方式, 可以代替 XML。我们目前刚好处于这个时代, Spring4.x 和 Spring Boot 都推荐使用 Java 配置。

所有这些配置都代表了开发时的损耗。因为在思考 Spring 特性配置和解决业务问题之间需要进行思维切换, 所以写配置挤占了写应用程序逻辑的时间。除此之外, 项目的依赖管理也是件吃力不讨好的事情。决定项目里要用哪些库就已经够让人头痛的了, 你还要知道这些库的哪个版本和其他库不会有冲突, 这难题实在太棘手。并且, 依赖管理也是一种损耗, 添加依赖不是写应用程序代码。一旦选错了依赖的版本, 随之而来的不兼容问题毫无疑问会是生产力杀手。

Spring Boot 让这一切成为了过去。

Spring Boot 简化了基于 Spring 的应用开发, 只需要“run”就能创建一个独立的、生产级别的 Spring 应用。Spring Boot 为 Spring 平台及第三方库提供开箱即用的设置 (提供默认设置), 这样我们就可以简单的开始。多数 Spring Boot 应用只需要很少的 Spring 配置。

我们可以使用 SpringBoot 创建 java 应用, 并使用 `java -jar` 启动它, 或者采用传统的 war 部署方式。

Spring Boot 主要目标是:

为所有 Spring 的开发提供一个从根本上更快的入门体验。

开箱即用，但通过自己设置参数，即可快速摆脱这种方式。

提供了一些大型项目中常见的非功能性特性，如内嵌服务器、安全、指标，健康检测、外部化配置等。

绝对没有代码生成，也无需 XML 配置。

2 Spring Boot 基础配置

本章概要

- @SpringBootApplication
- 定制 Banner
- Web 容器配置
- Properties 配置
- Bean 属性配置
- YAML 配置
- Profile

2.1 开发环境

IDE: SpringToolSuite4 (简称 STS)

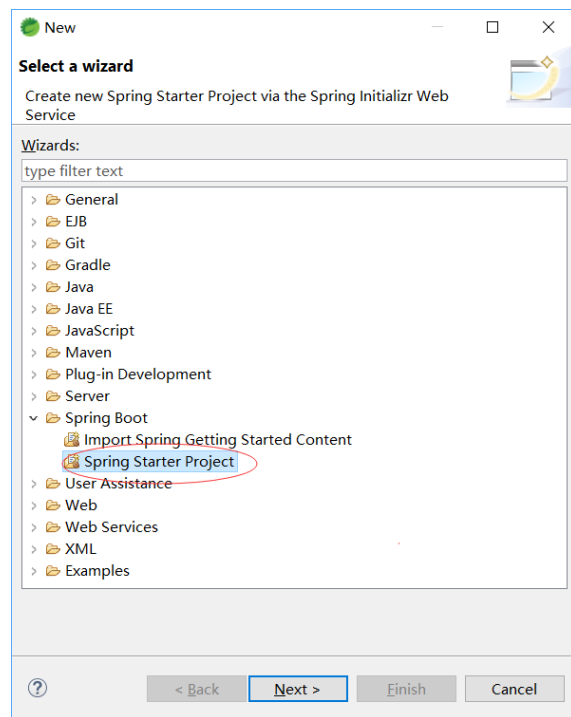
JDK: JDK8

Maven: 3.5.4

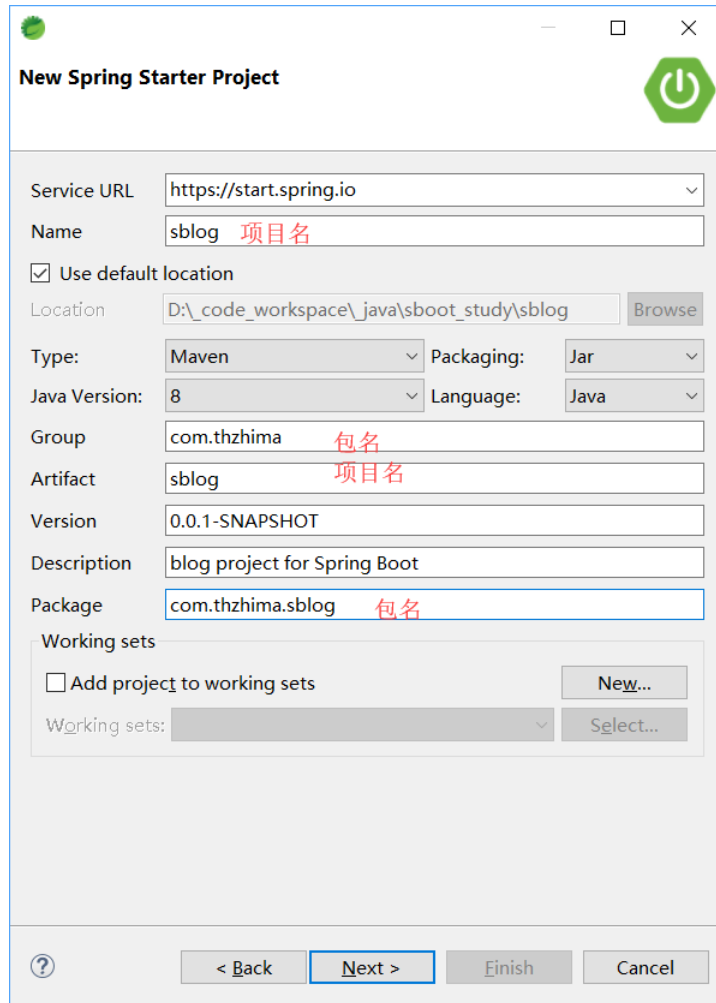
在 STS 中设置 JDK、Maven。

2.1.1 创建 Spring Boot 项目

1) 在 Spring Boot 中选择 Spring Starter Project。



2) 输入项目名、包名等信息。

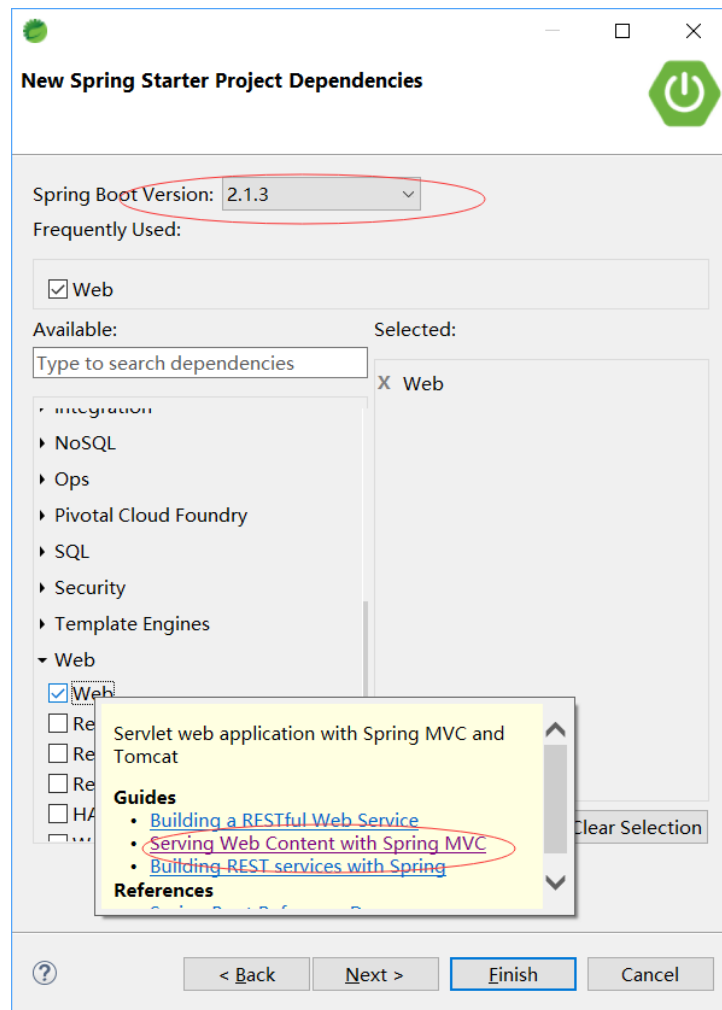


The image shows a 'New Spring Starter Project' dialog box. It contains the following fields and options:

- Service URL:** A dropdown menu with 'https://start.spring.io' selected.
- Name:** A text field containing 'sblog' with a red annotation '项目名' (Project Name).
- Use default location:** A checked checkbox.
- Location:** A text field containing 'D:_code_workspace\java\sboot_study\sblog' with a 'Browse' button.
- Type:** A dropdown menu with 'Maven' selected.
- Packaging:** A dropdown menu with 'Jar' selected.
- Java Version:** A dropdown menu with '8' selected.
- Language:** A dropdown menu with 'Java' selected.
- Group:** A text field containing 'com.thzhima' with a red annotation '包名' (Package Name).
- Artifact:** A text field containing 'sblog' with a red annotation '项目名' (Project Name).
- Version:** A text field containing '0.0.1-SNAPSHOT'.
- Description:** A text field containing 'blog project for Spring Boot'.
- Package:** A text field containing 'com.thzhima.sblog' with a red annotation '包名' (Package Name).
- Working sets:** A section with a checkbox 'Add project to working sets' and a 'New...' button. Below it is a 'Working sets:' dropdown menu and a 'Select...' button.

At the bottom, there are buttons for '?', '< Back', 'Next >', 'Finish', and 'Cancel'. The 'Next >' button is highlighted with a blue border.

3) 选择 Spring Boot 版本和项目类型为 Spring MVC Web 项目。



4) 点“Finish”从 Spring 下载 Spring Boot 项目框架。

New Spring Starter Project

Site Info

Base Url

https://start.spring.io/starter.zip

Full Url

https://start.spring.io/starter.zip?
name=sblog&groupId=com.thzhima&artifactId=sblog
&version=0.0.1-SNAPSHOT&description=blog
+project+for+Spring
+Boot&packageName=com.thzhima.sblog&type=mav

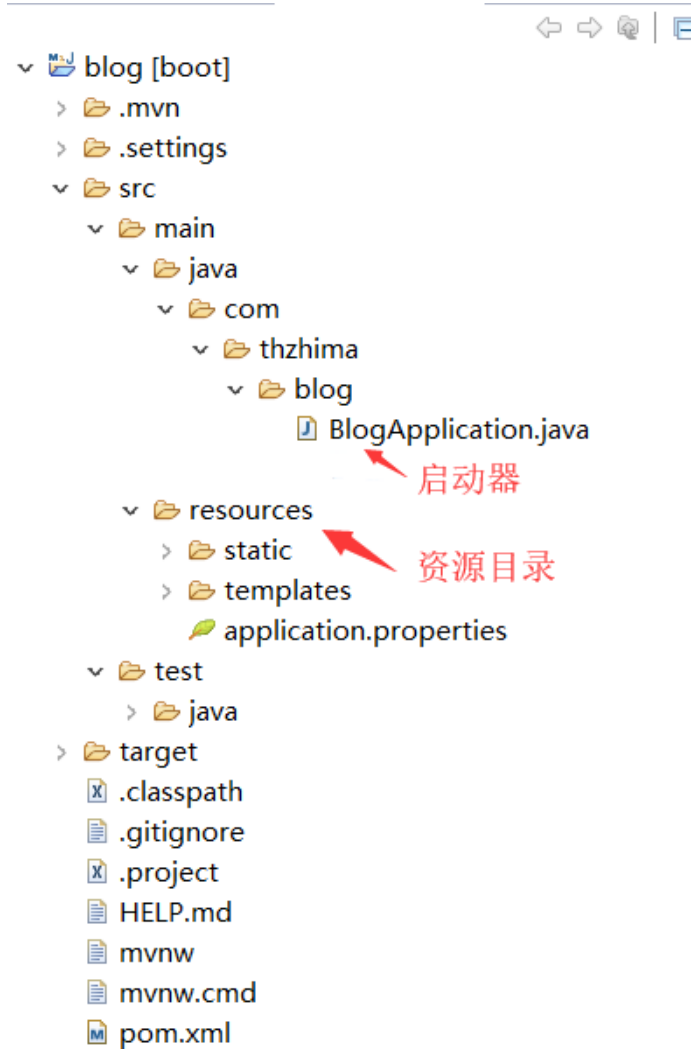
?

< Back

Next >

Finish

Cancel



5) 显示引导页面。可按引导页面内容创建项目内容。

Getting Started · Serving Web Content with Spring MVC

spring by Pivotal. PROJECTS GUIDES BLOG TRAINING & CERTIFICATION

Serving Web Content with Spring MVC

This guide walks you through the process of creating a "hello world" web site with Spring.

What you'll build

You'll build an application that has a static home page, and also will accept HTTP GET requests at:

`http://localhost:8080/greeting`

and respond with a web page displaying HTML. The body of the HTML contains a greeting:

`"Hello, World!"`

You can customize the greeting with an optional `name` parameter in the query string:

`http://localhost:8080/greeting?name=User`

Get the Code

HTTPS SSH Subversion

`https://github.com/spring-guides`

DOWNLOAD ZIP

GO TO REPO

Table of contents

- What you'll build
- What you'll need
- How to complete this guide

2.1.2 启动器类

框架搭建完成后，会生成一个“XxxApplication.java”文件。

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.ComponentScan;

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

该文件有一个注解@SpringBootApplication。这样的类为 SpringBoot 项目启动器。

@SpringBootApplication 是一个复合注解，包括@ComponentScan，和@SpringBootConfiguration，@EnableAutoConfiguration。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
    @Filter(type=FilterType.CUSTOM, classes=TypeExcludeFilter.class),
    @Filter(type=FilterType.CUSTOM, classes=AutoConfigurationExcludeFilter.class)})
public @interface SpringBootApplication {
```

```
.....  
}
```

@SpringBootConfiguration 继承自@Configuration，二者功能也一致，标注当前类是配置类，并会将当前类内声明的一个或多个以@Bean 注解标记的方法的实例纳入到 spring 容器中，并且实例名就是方法名。

@EnableAutoConfiguration 的作用启动自动的配置，@EnableAutoConfiguration 注解的意思就是 Springboot 根据你添加的 jar 包来配置你项目的默认配置，比如根据 spring-boot-starter-web，来判断你的项目是否需要添加了 webmvc 和 tomcat，就会自动的帮你配置 web 项目中所需要的默认配置。

@ComponentScan，扫描当前包及其子包下被@Component，@Controller，@Service，@Repository、@RestController、@Configuration 注解标记的类并纳入到 spring 容器中进行管理。是以前的<context:component-scan>（以前使用在 xml 中使用的标签，用来扫描包配置的平行支持）。

注：Spring Boot 默认只扫描与启动器“BlogApplication.java”同包或子包中的文件。若控制器类在其它包中，需在启动器类中添加@ComponentScan 注解，使用 basePackages 属性指明所要扫描的包。

```
@ComponentScan(basePackages= {"com.thzhima.blog.controller"})
```

```
@SpringBootApplication
```

```
public class BlogApplication {
```

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

```
}
```

2.2 Spring MVC

2.2.1 创建控制器

SpringMVC 中的控制器有两中，一种是用@Controller 声明的，还有一种是用@RestController 声明的。@RestController 相当于@Controller 与 @ResponseBody。返回数据而非页面。

```
@RestController
public class HelloController {

    @RequestMapping("/hello")
    public Map sayHello() {
        Map map = new HashMap();
        map.put("info", "Hello World");
        return map;
    }
}
```

2.2.2 启动运行

运行启动器，控制台输出：


```

  ____  _
 / ___|| | | |
| |___| |_| |
|___|_||_||_|
:: Spring Boot ::      (v2.2.0.M6)

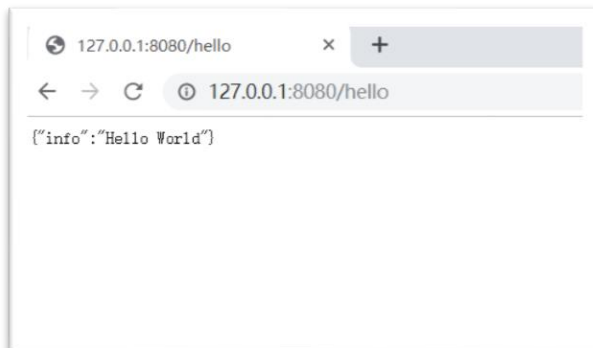
```

```

019-10-03 01:31:10.419 INFO 12456 --- [main] com.thzhima.ch02.Ch02Application : Starting Ch02Application on WR-Extr
019-10-03 01:31:10.424 INFO 12456 --- [main] com.thzhima.ch02.Ch02Application : No active profile set, falling back t
019-10-03 01:31:11.726 INFO 12456 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080
019-10-03 01:31:11.735 INFO 12456 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
019-10-03 01:31:11.736 INFO 12456 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tom
019-10-03 01:31:11.809 INFO 12456 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebAppli
019-10-03 01:31:11.809 INFO 12456 --- [main] o.s.web.context.ContextLoader : Root WebApplicationContext: initiali
019-10-03 01:31:11.949 INFO 12456 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applic
019-10-03 01:31:12.045 WARN 12456 --- [main] ion$DefaultTemplateResolverConfiguration : Cannot find template location: class
019-10-03 01:31:12.100 INFO 12456 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http
019-10-03 01:31:12.102 INFO 12456 --- [main] com.thzhima.ch02.Ch02Application : Started Ch02Application in 1.9894448
019-10-03 01:31:42.977 INFO 12456 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet
019-10-03 01:31:42.977 INFO 12456 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServ
019-10-03 01:31:42.981 INFO 12456 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 4 ms

```

在浏览器打开["http://127.0.0.1:8080/hello"](http://127.0.0.1:8080/hello)，显示：



2.3定制 Banner

Spring Boot 项目在启动时会打印一个 Banner。这个 Banner 是可定制的。在 resources 目录下创建一个 banner.txt 文件，在这个文件中写入的文本将在项目启动时打印出来。

注：可以参考以下网站，生成 Banner 文字

- <http://www.kammerl.de/ascii/AsciiSignature.php>
- <http://patorjk.com/software/taag>
- <http://www.network-science.de/ascii/>

关闭 Banner 输出也是可以的：

```
public static void main(String[] args) {  
    // SpringApplication.run(Ch02Application.class, args);  
    SpringApplicationBuilder builder = new SpringApplicationBuilder(Ch02Application.class);  
    builder.bannerMode(Banner.Mode.OFF);  
    SpringApplication app = builder.build();  
    app.run(args);  
}
```

2.4Web 容器配置

Spring Boot 项目中，可以内置 Tomcat、Jetty、Netty、Undertow 等容器。当在 bom.xml 文件中添加 spring-boot-starter-web 依赖之后，默认会使用 Tomcat 作为 Web 容器。如果要对 Tomcat 做进一步配置，可以在 application.properties 中进行配置：

```
#服务器端口号
```

```
server.port=8088

#当项目出错时，跳转到的页面
server.error.path=/error

#项目URL访问路径，默认是“/”。
server.servlet.context-path=/ch02

# 会话失效时间
server.servlet.session.timeout=30m

# 配置Tomcat请求编码
server.tomcat.uri-encoding=utf-8

# Tomcat最大线程数
server.tomcat.max-threads=500

# 存放Tomcat运行日志和临时文件的目录，若不配置，则默认使用系统的临时目录。
#server.tomcat.basedir=C:\Users\wangrui\tmp
```

2.5 HTTPS 配置

HTTPS 具有很好的安全性，比如微信公众号、小程序等的开发都要使用 HTTPS 来完成。

2.5.1 生成证书

下面我们使用 java 自带的证书管理工具，生成一个数字证书：

```
D:\>keytool -genkey -alias tomcathttps -keyalg RSA -keysize 2048 -keystore my.keystore -validity 3650
输入密钥库口令：
再次输入新口令：
您的名字与姓氏是什么？
  [Unknown]: wang rui
您的组织单位名称是什么？
  [Unknown]: thzhima.com
您的组织名称是什么？
  [Unknown]: thzhima.com
您所在的城市或区域名称是什么？
  [Unknown]: NanJing
您所在的省/市/自治区名称是什么？
  [Unknown]: JiangSu
该单位的双字母国家/地区代码是什么？
  [Unknown]: CN
CN=wang rui, OU=thzhima.com, O=thzhima.com, L=NanJing, ST=JiangSu, C=CN是否正确？
[否]: y

输入 <tomcathttps> 的密钥口令
      (如果和密钥库口令相同，按回车):

Warning:
JKS 密钥库使用专用格式。建议使用 "keytool -importkeystore -srckeystore my.keystore -destkeystore my.keystore -deststoretype pkcs12" 迁移到行业标准格式 PKCS12。
```

命令 `keytool` 在 `JAVA_HOME/bin` 目录下。














参数：

- `-genkey` 表示要创建一个密钥。
- `-alias keystore` 的别名。这里别名为 `tomcathttps`。
- `-keyalg` 表示加密算法，这里用了 `RSA`，非对称加密算法。
- `-keysize` 表示密钥的长度
- `-keystore` 表示生成的密钥文件的存放位置，存放在当前目录下的 `my.keystore` 文件中。
- `-validity` 表示密钥的有效时间，这里为 3650 天。

2.5.2 使用证书

设置 Tomcat 使用 HTTPS,分两个步骤：

1. 将生成的 `keystore` 文件复制到项目根目录下。（与 `src` 同级）

- ▼  ch02 [boot]
- >  .mvn
- >  .settings
- >  src
- >  target
-  .classpath
-  .gitignore
-  .project
-  HELP.md
-  mvnw
-  mvnw.cmd
-  my.keystore
-  pom.xml

2. 在 application.properties 文件中加入相应配置:

```
# keystore文件名
server.ssl.key-store=my.keystore

# keystore 密码
server.ssl.key-store-password=kingrui

# 密码别名
server.ssl.key-alias=tomcathttps
```

2.5.3 在浏览器中查看

在浏览器 url 输入“<https://localhost:8888/ch02/hello>”。由于证书是我们自己成生的，不被浏览器认可。可以将证书添加到信任中，或继续访问。



2.6 Jetty 配置

在 spring-boot-starter-web 中默认使用的服务器是 Tomcat，可以在 bom.xml 中进行设置，排除 Tomcat，添加对 Jetty 的依赖。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions>
    <!-- 排除Tomcat -->
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-tomcat</artifactId>
    </exclusion>
  </exclusions>
</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-jetty -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```


2.7 Undertow 配置

与 Jetty 配置类似:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions>
    <!-- 排除Tomcat -->
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-tomcat</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-undertow -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-undertow</artifactId>
</dependency>
```

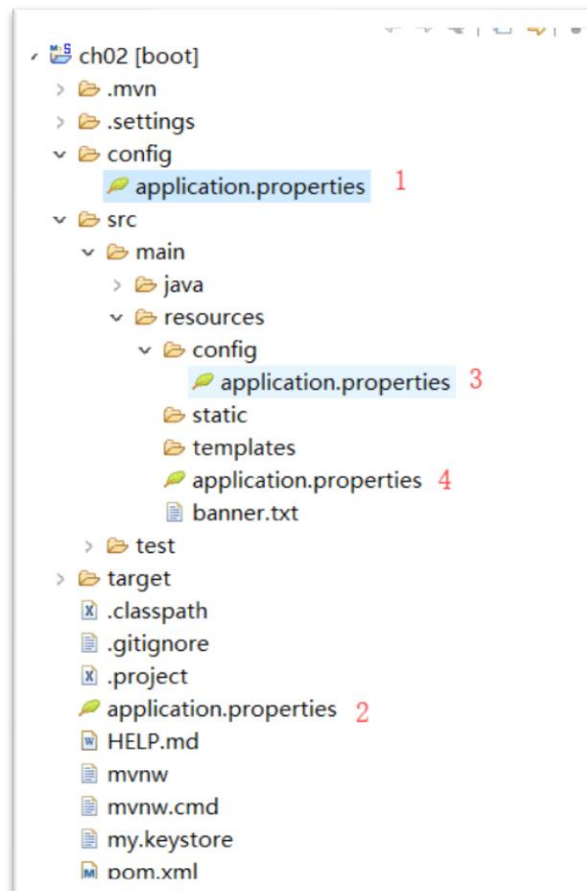
2.8 Properties 配置

SpringBoot 中采用了大量的自动化配置，但是对开发者而言，在实际项目中不可避免会有一些需要自己手动配置。承载这些自定义配置的文件就是

resources 目录下的 application.properties 文件（也可以使用 YAML 配置来替代 application.properties 配置）。

注：Spring Boot 项目中 application.properties 配置文件可以放在 4 个位置下，优先级由高到低。

1. 项目根目录下的 config 目录下
2. 项目根目录下
3. classpath 下的 config 目录下
4. classpath 下（即：src/main/resources 下）



2.8.1 配置属性

为 Java Bean 中的属性进行配置，类似在 spring XML 设置中使用 property 为 bean 的属性赋值。

1) Book 类:

```
@Component
@ConfigurationProperties(prefix = "book")    // 要加载的配置项的前缀。
public class Book {

    private String name;
    private String author;
    private float price;

    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 float getPrice() {  
    return price;  
}  
public void setPrice(float price) {  
    this.price = price;  
}  
  
@Override  
public String toString() {  
    return "Book [name=" + name + ", author=" + author + ", price=" + price + "];"  
}  
}
```

2) application.properties

book.name=Pro Java Programming

book.author=Wang Rui

book.price=101.11

Spring Boot 采用一种宽松的规则来进行属性的绑定，如果 Bean 中的属性名为 authorName，那么在配置文件中可以是：

- **book.author_name**
- **book.author-name**

- `book.authorName`
- `book.authername`

3) 控制器中使用 Book 对象。

```
@RestController
public class MyController {

    @Autowired
    private Book book;

    @RequestMapping("/book")
    public Book aboutBook() {
        return this.book;
    }
}
```

2.9YAML 配置

YAML 是 JSON 的超集，是一种专门用来书写配置文件的语言，可以替代 `application.properties`。在创建一个 Spring Boot 项目时，引入的 `spring-boot-starter-web` 依赖间接地引入了 `snakeyaml` 依赖，`snakeyaml` 会对 YAML 文件进行解析。

在 Spring Boot 项目中，使用 YAML 只需要在 `resources` 目录下创建一个 `application.yml` 文件即可，然后向 `application.yml` 中添加配置：

```
server :  
  port : 8888  
  servlet :  
    context-path : /ch02  
  tomcat :  
    uri-encoding : utf-8  
book :  
  name : pro Java Programming  
  author : wang rui  
  price : 101. 11
```

2.10 Profile

使用 Profile 可以在多个配置文件之间进行切换。在开发过程中，我们可能需要在开发环境与测试环境之间进行切换。比如数据库配置、redis 配置、mongodb 配置等。我们只需要为不同的环境编写不同的配置文件，再指明使用哪个就可以。

示例：有一个是生产环境的配置文件。

1)开发环境：application-dev.properties:

```
server.port=8888
```

2) application.properties 中，指明使用开发环境:

```
server.port=9999
```

profile 设置，当前激活的配置文件。

```
spring.profiles.active=dev
```

这时，启动使用的配置文件为“开发环境”。

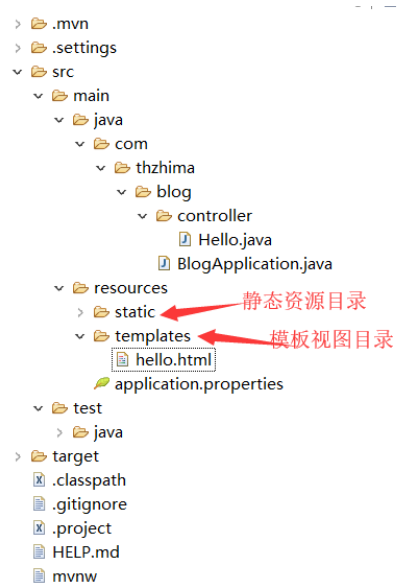
3 SpringBoot 整合视图层技术

本章概要

- 整合 Thymeleaf

在目前的企业级应用开发中，前后端分离是趋势，但是视图层技术还占有一席之地。官方推荐的模板引擎是 Thymeleaf，不过 FreeMarker 也支持，JSP 技术在这里并不推荐使用。

模板视图文件的存放位置：



在 resources/templates 目录下创建 hello.html 文件。该目录是控制器转发的视图目录。其下为模板文件。

3.1 整合 Thymeleaf

Thymeleaf 是用于 Web 和独立环境的现代服务器端 Java 模板引擎。

Thymeleaf 是新一代模板引擎，类似 Velocity、Freemarker 等传统 Java 模板引擎。与传统 Java 模板引擎不同的是，Thymeleaf 支持 HTML 原型，既可以让前端工程师在浏览器中直接打开查看样式，也可以让后端工程师结合真实数据查看显示效果。同时 Spring Boot 提供了 Thymeleaf 自动化配置方案。

在 pom.xml 中添加依赖：

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

3.1.1 配置 Thymeleaf

Spring Boot 为 Thymeleaf 提供了自动化配置类 ThymeleafAutoConfiguration，相关的配置属性在 ThymeleafProperties 类中，部分源码如下：

```
@ConfigurationProperties(prefix = "spring.thymeleaf")
public class ThymeleafProperties {

    private static final Charset DEFAULT_ENCODING = StandardCharsets.UTF_8;
```

```
public static final String DEFAULT_PREFIX = "classpath:/templates/";

public static final String DEFAULT_SUFFIX = ".html";

.....
```

由此配置可以看到，默认模板位置在 classpath:/templates/，默认的后缀名为.html。

如果开发者想对默认的 Thymeleaf 配置参数进行自定义配置，那么可以直接在 application.properties 中进行配置：

是否开启缓存，在开发时设置为False，默认为True。

```
spring.thymeleaf.cache=false
```

检查模板是否存在，默认为true

```
spring.thymeleaf.check-template=true
```

检查模板位置是否存在，默认为true

```
spring.thymeleaf.check-template-location=true
```

模板文件编码

```
spring.thymeleaf.encoding=UTF-8
```

模板文件位置

```
spring.thymeleaf.prefix= classpath:/templates/
```

模板文件后缀

```
spring.thymeleaf.suffix=.html
```

```
# Content-type
```

```
spring.thymeleaf.servlet.content-type=text/html; charset=utf-8
```

3.1.2 配置控制器

创建 Book 实体类，然后在 Controller 中返回 ModelAndView。

Book.java:

```
package com.thzhima.ch03.bean;

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

    public Book(Integer id, String name, String author) {
        super();
        this.id = id;
        this.name = name;
        this.author = author;
    }
}
```

```
}

public Book() {
    super();
    // TODO Auto-generated constructor stub
}

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;
}

public String getAuthor() {
    return author;
}
```

```
public void setAuthor(String author) {  
    this.author = author;  
}  
  
@Override  
public String toString() {  
    return "Book [id=" + id + ", name=" + name + ", author=" + author + "];"  
}  
  
}
```

BookController 控制器:

```
@Controller  
public class BookController {  
  
    @GetMapping("/books") // 相当于@RequestMapping(method = RequestMethod.GET)  
    public ModelAndView books(ModelAndView mv) {  
        List<Book> list = new ArrayList<>();  
  
        Book book1 = new Book(1, "Java Programming", "WangRui");  
        Book book2 = new Book(2, "Python Flask Web开发", "王瑞");  
  
        list.add(book1);  
        list.add(book2);  
        mv.addObject("books", list);  
    }  
}
```

```
mv.setViewName("books");

return mv;
}
```

3.1.3 创建视图

books.html 是我们要创建的对应控制器返回的视图文件:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8">
<title>图书列表</title>
</head>
<body>
    <table border="1">
    <tr>
        <td>图书编号</td>
        <td>书名</td>
        <td>作者</td>
    </tr>
    <tr th:each="book:${books}">
        <td th:text="${book.id}"></td>
```

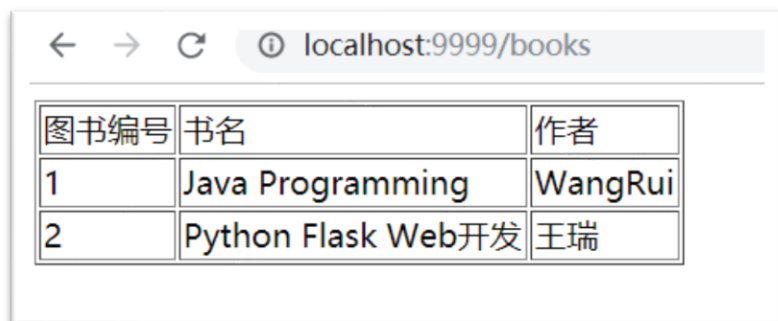
```
<td th:text="${book.name}"></td>
<td th:text="${book.author}"></td>
</tr>
</table>
</body>
</html>
```

注：

安装 thymeleaf Eclipse 插件：<http://www.thymeleaf.org/eclipse-plugin-update-site/>

安装完成之后，在项目右击“Thymeleaf” – “Add Thymeleaf Nature”。

3.1.4 运行



The screenshot shows a web browser window with the address bar displaying 'localhost:9999/books'. The browser content area contains a table with three columns: '图书编号' (Book ID), '书名' (Book Name), and '作者' (Author). The table has two data rows. The first row contains '1', 'Java Programming', and 'WangRui'. The second row contains '2', 'Python Flask Web开发', and '王瑞'.

图书编号	书名	作者
1	Java Programming	WangRui
2	Python Flask Web开发	王瑞

本节主要介绍了 Spring Boot 整合 Thymeleaf。Thymeleaf 具体使用，参考 <https://www.thymeleaf.org>。

4 Spring Boot 整合 Web 开发

本章概要

- 返回 JSON 数据
- 静态资源访问
- 文件上传
- @ControllerAdvice
- 自定义错误页面
- CORS 支持
- 配置类与 XML 配置
- 注册拦截器
- 整合 Servlet、Filter 和 Listener
- 路径映射
- 配置 AOP
- 自定义欢迎页面
- 自定义 favicon
- 除去某自动配置

4.1 返回 JSON 数据

4.1.1 默认实现

JSON 是目前主流的前后端数据传输方式，Spring MVC 中使用消息转换器 `HttpMessageConverter` 对 JSON 的转换提供了很好的支持，在 Spring Boot 中更进一步，对相关配置做了更进一步的简化。

在 Spring Boot 项目中添加了 `spring-boot-starter-web` 依赖，这个依赖中默认加入了 `jackson-annotation` 作为 JSON 处理器，此时不需要添加额外的 JSON 处理器，就可以返回 JSON 数据了。

示例：

创建 Book 实体类：

```
public class Book {  
  
    private String name;  
    private String author;  
  
    @JsonIgnore // 忽略字段  
    private float price;  
  
    @JsonFormat(pattern = "yyyy-MM-dd") // 格式化日期  
    private Date publicationDate;  
  
    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 float getPrice() {
        return price;
    }

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

    public Date getPublicationDate() {
        return publicationDate;
    }
}
```

```
public void setPublicationDate(Date publicationDate) {  
    this.publicationDate = publicationDate;  
}  
  
public Book() {  
    super();  
    // TODO Auto-generated constructor stub  
}  
  
public Book(String name, String author, float price, Date publicationDate) {  
    super();  
    this.name = name;  
    this.author = author;  
    this.price = price;  
    this.publicationDate = publicationDate;  
}  
}
```

创建 BookController 控制器:

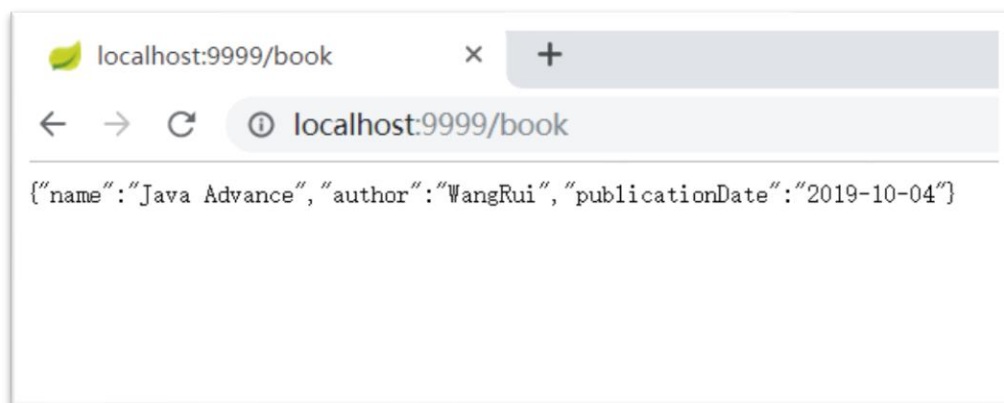
```
@Controller  
public class BookController {  
  
    @GetMapping("/book")  
    @ResponseBody    // 返回JSON
```

```
public Book book() {  
    Book b= new Book("Java Advance","WangRui", 101.11F, new Date());  
    return b;  
}  
  
}
```

如果一个控制器中，所有的方法都返回 JSON 数据，可以将类上的@Controller 更改为@RestController，省略方法上的@ResponseBody 注解。如下：

```
@RestController  
public class BookController {  
  
    @GetMapping("/book")  
    //ResponseBody  
    public Book book() {  
        Book b= new Book("Java Advance","WangRui", 101.11F, new Date());  
        return b;  
    }  
  
}
```

运行结果：



这是 Spring Boot 自带的处理方式。如果采用这种方式，那么对与字段忽略、日期格式化等常见需求都可以通过注解来解决。这是通过 Spring 默认提供的 MappingJackson2HttpMessageConverter 来实现的。当然也可以根据自己的需要自定义 JSON 转换器。

4.1.2 自定义 JSON 转换器 Gson

常用的 JSON 处理器除了 jackson-databind 之外，还有 Gson 和 fastjson。

Gson 是 Google 的一个开源 JSON 解析框架。使用 Gson，需要先除去默认的 jackson-databind，然后加入 Gson 依赖。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions>
    <exclusion>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-databind</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

```
        </exclusion>
    </exclusions>
</dependency>

<!-- https://mvnrepository.com/artifact/com.google.code.gson/gson -->
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
</dependency>
```

字段忽略、日期格式化等常见需求仍可以通过注解来解决。

Gson 使用 GsonHttpMessageConverter 做为数据转换器，在必要的情况下我们可以提供这个 Bean。

4.1.1 自定义 JSON 转换器 Fastjson

fastjson

fastjson 是阿里巴巴的一个开源 JSON 解析框架，是目前 JSON 解析速度最快的开源框架。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <groupId>com.fastxml.jackson.core</groupId>
            <artifactId>jackson-databind</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```



```
        SerializerFeature.PrettyFormat,  
        SerializerFeature.WriteNullListAsEmpty,  
        SerializerFeature.WriteNullStringAsEmpty  
    );
```

```
    return converter;
```

```
}
```

```
}
```

另外：在 application.properties 中加入以下配置，否则中文会乱码。

```
spring.http.encoding.force-response=true
```

返回的结果：

← → ↻ ⓘ localhost:8888/book

```
{ "@type": "com.thzhima.ch04.bean.Book", "author": "王瑞", "name": "Java Advance", "price": 101.11F, "publicationDate": "2019年10月05日" }
```

4.2 静态资源的访问

在 spring-boot-starter-web 中使用 WebMvcAutoConfiguration 类进行默认配置。在该类中 staticLocations 是一个数组，这个数组中保存了静态资源的位置。当我们访问静态资源时，会在这个位置中进行查找。

```
String[] staticLocations = getResourceLocations(this.resourceProperties.getStaticLocations());
```

从这段代码中，可以看到这个数组的值是 getResourceLocations()方法返回的，并输入了 getStaticLocations()的返回值。从以下代码下以面出，该方法的返回值是静态常量中给定的初始值。包含 4 个路径。

```
@ConfigurationProperties(prefix = "spring.resources", ignoreUnknownFields = false)
public class ResourceProperties {

    private static final String[] CLASSPATH_RESOURCE_LOCATIONS = { "classpath:/META-INF/resources/",
        "classpath:/resources/", "classpath:/static/", "classpath:/public/" };

    private String[] staticLocations = CLASSPATH_RESOURCE_LOCATIONS;

    public String[] getStaticLocations() {
        return this.staticLocations;
    }
}
```

getResourceLocations()方法，传入包含这 4 个路径的数组做参数。返回的是 5 个路径的数组：

```
private static final String[] SERVLET_LOCATIONS = { "/" };
static String[] getResourceLocations(String[] staticLocations) {
    String[] locations = new String[staticLocations.length + SERVLET_LOCATIONS.length];
    System.arraycopy(staticLocations, 0, locations, 0, staticLocations.length);
}
```

```
System.arraycopy(SERVLET_LOCATIONS, 0, locations, staticLocations.length, SERVLET_LOCATIONS.length);  
return locations;  
}
```

最终返回的这个数组如下所示。另外，在 ResourceProperties 类中定义了可以在 application.properties 中通过“spring.resources”前缀来定义属性，包含静态资源目录。

```
{ "classpath:/META-INF/resources/",  
  "classpath:/resources/",  
  "classpath:/static/",  
  "classpath:/public/",  
  " /" }
```

优先级：这 5 个路径的优先级从高到低。

我们使用的开发环境，在默认情况下，只给我们建了 classpath:/static/这个目录。对应的是/src/main/resources/static/。

4.2.1 自定义静态资源访问策略

在默认情况下访问静态资源的 url 为"/**"。定义如下。

```
@ConfigurationProperties(prefix = "spring.mvc")  
public class WebMvcProperties {  
  
    /**  
     * Path pattern used for static resources.  
     */
```

```
private String staticPathPattern = "/*";
```

如果默认的静态资源过滤策略不能满足开发需求，也可以自定义静态资源过滤策略，自定义静态资源过滤策略有以下两种方式：

1. 在配置文件中定义

可以在 application.properties 中直接定义过滤规则和静态资源位置，代码如下：

```
spring.mvc.static-path-pattern=/static/**
spring.resource.static-locations=classpath:/static/
```

过滤规则为 /static/**，静态资源位置为 classpath:/static/。

重新启动项目，在浏览器中输入“http://localhost:8080/static/p1.png”，即可看到 classpath:/static/ 目录下的资源。

2. Java 编码定义

也可以通过 Java 编码方式来定义，此时只需要实现 WebMvcConfigurer 接口即可，然后实现该接口的 addResourceHandlers 方法，代码如下：

```
@Configuration
public class MyWebMvcConfig implements WebMvcConfigurer{

    public void addResourceHandlers(ResourceHandlerRegistry reg) {
        ResourceHandlerRegistration regis = reg.addResourceHandler("/static/**");
        regis.addResourceLocations("classpath:/static/");
    }
}
```

4.3 文件上传

Java 中的文件上传一共涉及两个组件，一个是 CommonsMultipartResolver，另一个是 StandardServletMultipartResolver，其中 CommonsMultipartResolver 使用 commons-fileupload 来处理 multipart 请求，而 StandardServletMultipartResolver 则是基于 Servlet 3.0 来处理 multipart 请求的，因为若使用 StandardServletMultipartResolver，则不需要添加额外的 jar 包。Tomcat 7.0 开始就支持 Servlet 3.0 了。

而在 Spring Boot 提供了文件上传自动化配置类 MultipartAutoConfiguration 中，默认也是采用 StandardServletMultipartResolver。

```
@Configuration
@ConditionalOnClass({ Servlet.class, StandardServletMultipartResolver.class, MultipartConfigElement.class })
@ConditionalOnProperty(prefix = "spring.servlet.multipart", name = "enabled", matchIfMissing = true)
@ConditionalOnWebApplication(type = Type.SERVLET)
@EnableConfigurationProperties(MultipartProperties.class)
public class MultipartAutoConfiguration {

    private final MultipartProperties multipartProperties;

    public MultipartAutoConfiguration(MultipartProperties multipartProperties) {
        this.multipartProperties = multipartProperties;
    }

    @Bean
    @ConditionalOnMissingBean({ MultipartConfigElement.class, CommonsMultipartResolver.class })
    public MultipartConfigElement multipartConfigElement() {
        return this.multipartProperties.createMultipartConfig();
    }
}
```

```
@Bean(name = DispatcherServlet.MULTIPART_RESOLVER_BEAN_NAME)
@ConditionalOnMissingBean(MultipartResolver.class)
public StandardServletMultipartResolver multipartResolver() {
    StandardServletMultipartResolver multipartResolver = new StandardServletMultipartResolver();
    multipartResolver.setResolveLazily(this.multipartProperties.isResolveLazily());
    return multipartResolver;
}
}
```

根据这里的配置可以看出, 如果开发者没有提供 MultipartResolver, 那么默认采用的 MultipartResolver 就是 StandardServletMultipartResolver。因此, 在 Spring Boot 中上传文件甚至可以做到零配置。下面来看具体上传过程。

1) upload.html, 为了能直接访问该文件, 该文件放在静态资源目录中。

- ▼ src
 - ▼ main
 - ▼ java
 - ▼ com
 - ▼ thzhima
 - ▼ ch04
 - ▼ bean
 - ▼ controllers
 - BookController.java
 - FileUploadController.java
 - Ch041Application.java
 - Config.java
 - MyWebMvcConfig.java
 - ▼ resources
 - ▼ static
 - 1.png
 - upload.html
 - templates
 - application.properties

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>上传文件</title>
</head>
<body>
  <form action="/upload" method="post" enctype="multipart/form-data">
```

```
        选择一个文件: <input type="file" name="photo" />
        <input type="submit" value="上传"/>
    </form>
</body>
</html>
```

2) FileUploadController 控制器

```
@RestController
public class FileUploadController {

    @PostMapping("/upload")
    public String upload(MultipartFile photo, HttpServletRequest request) {
        String msg = "ok";

        // 获取Tomcat运行目录，这个目录在一个临时文件夹中。在生产环境下，应指定一个目录。
        //String basePath = request.getServletContext().getRealPath("/");
        String basePath = "/photos/"; // 指定根目录下的photos目录，为静态资源文件存储目录
        File dir = new File(basePath, "upload");// photos/upload为上传文件目录
        if(!dir.exists()) {// 创建上传文件的目录，
            dir.mkdirs();
            System.out.println("mkdir:::::::::::::" + dir.getAbsolutePath());
        }
        String fileName = photo.getOriginalFilename(); // 取文件的原始名字
        System.out.println("name: "+photo.getName());
        try {
```



```
        photo.transferTo(new File(dir, fileName)); // 保存文件

    } catch (IllegalStateException | IOException e) {
        msg = "fail";
        e.printStackTrace();
    }

    return msg;
}

}
```

上传的文件，如果想能访问，要设置这个目录是静态资源目录：

```
@Configuration
public class MyWebMvcConfig implements WebMvcConfigurer{

    public void addResourceHandlers(ResourceHandlerRegistry reg) {
        // 静态资源的访问路径，也可以配置为"/static/**"，以/static/aa.jpg来访问图片。
        ResourceHandlerRegistration regis = reg.addResourceHandler("/**");

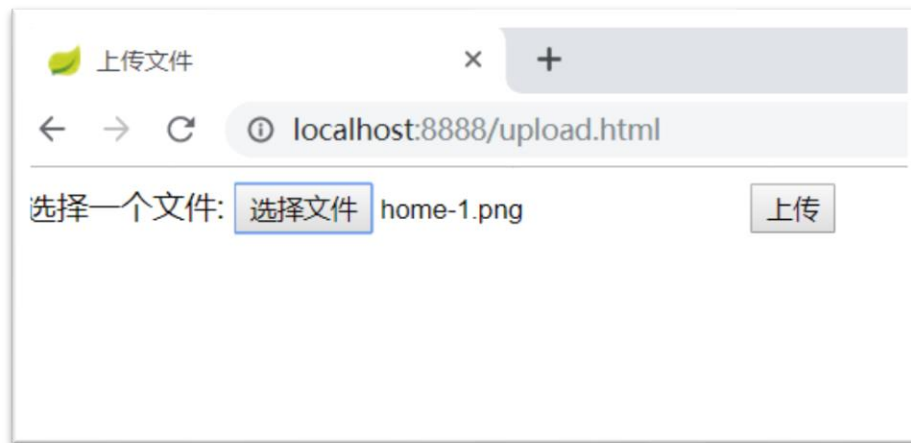
        // 这里设置了"resources/static"和"/photos/"根目录是资源目录。
        regis.addResourceLocations("classpath:/static/", "file:/photos/");
    }
}
```

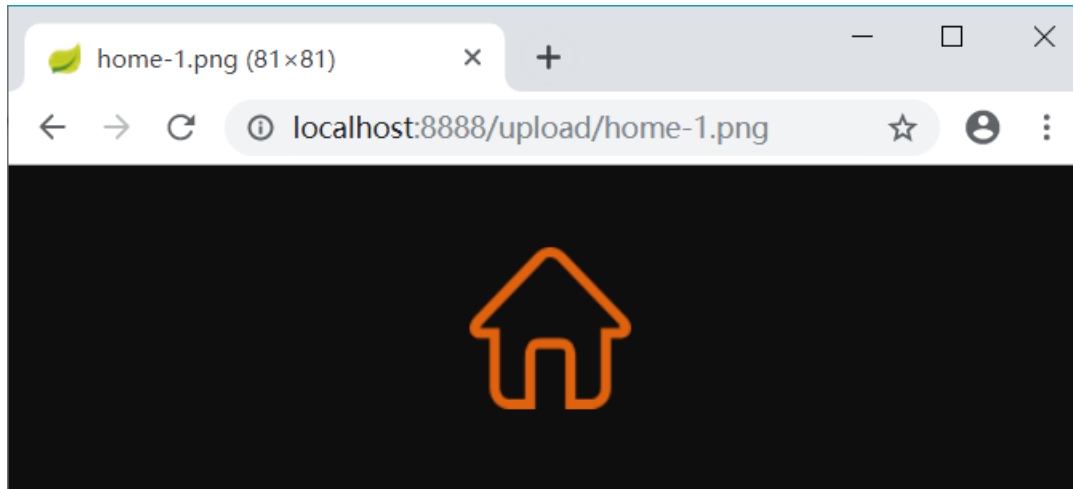
等效的配置，也可以使用配置文件在 `application.properties` 中进行配置：

#静态资源访问路径

```
spring.mvc.static-path-pattern=/**  
#静态资源映射路径  
spring.resources.static-locations=classpath:/static/,file:/photos/
```

运行：





如果开发者要对图片上传的细节进行配置，也是允许的，配置如下：

```
# 是否开启文件上传，默认为true。  
spring.servlet.multipart.enabled=true  
  
# 文件写入磁盘的阈值，默认为0  
spring.servlet.multipart.file-size-threshold=1024  
  
# 上传文件的最大大小，默认1MB。  
spring.servlet.multipart.max-file-size=10MB  
  
# 多文件上传时，文件的总大小，默认为10MB。  
spring.servlet.multipart.max-request-size=30MB
```

```
# 文件是否延迟解析，默认为false.  
spring.servlet.multipart.resolve-lazily=false
```

4.3.1 多文件上传

HTML 部分：

```
<!DOCTYPE html>  
<html>  
<head>  
<meta charset="UTF-8">  
<title>上传文件</title>  
</head>  
<body>  
    <form action="/upload" method="post" enctype="multipart/form-data">  
        选择一个文件: <input type="file" name="photo" multiple/>  
        <input type="submit" value="上传"/>  
    </form>  
</body>  
</html>
```

控制器部分：

```
@PostMapping("/uploads")
```

```
public String upload(MultipartFile[] photos, HttpServletRequest request) {  
    .....  
}
```

4.4@ControllerAdvice

@ControllerAdvice 是@Controller 的增强版。其实就是在控制器上加入切面处理。一般搭配@ExceptionHandler、@ModelAttribute 以及@InitBinder 使用。

4.4.1 全局异常处理

@ControllerAdvice 最常见的使用场景就是全局异常处理。可以通过@ControllerAdvice 结合@ExceptionHandler 定义全局异常捕获机制，如下所示：

```
@ControllerAdvice  
public class MyExceptionHandler {  
  
    @ExceptionHandler(MaxUploadSizeExceededException.class)  
    public void uploadException(MaxUploadSizeExceededException ex, HttpServletResponse resp) throws IOException {  
        resp.setContentType("text/html; charset=utf-8");  
        PrintWriter out = resp.getWriter();  
        out.print("上传文件大小超过限制");  
        out.flush();  
    }  
}
```

```
        out.close();
    }
}
```

如果要处理所有类型的异常只要将 `MaxUploadSizeExceededException` 改为 `Exception`。方法的入参可以是异常对象、`HttpServletRequest`、`HttpServletResponse`、`Model` 等。返回值可以是 JSON、`ModelAndView`、一个逻辑视图名等。

如返回一个视图：

```
@ExceptionHandler(Exception.class)
public ModelAndView except(Exception ex) {
    ModelAndView mv = new ModelAndView();
    mv.setViewName("error");
    mv.addObject("msg", "发生了错误。");
    return mv;
}
```

模板页面 `error.html`:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8">
<title>消息</title>
</head>
<body>
    网站建设中。
    <div th:text="${msg}"></div>
</body>
</html>
```

4.4.2 添加全局数据

@ControllerAdvice 是一个全局数据处理组件，因此也可以在@ControllerAdvice 中配置全局数据，使用@ModelAttribute 注解进行配置，代码如下：

```
@ControllerAdvice
public class GlobalConfig {

    @ModelAttribute(name="userInfo")
    public Map getUserInfo() {
        Map map = new HashMap();
        map.put("userName", "WangRui");
        map.put("gender", "Male");
        return map;
    }
}
```

这个“userInfo”为 key 的数据是一个 Map 类型。会加到所有的 springMVC 的 ModelAndView 对象的 Model 对象中。成为 Model 对象的一个属性。
在控制器中，获取 Model 对象中这个数据。

```
@GetMapping("/user")
@ResponseBody
public Map getUserInfo(Model m) {
    Map mp = m.asMap();

    Map info = new HashMap();
    info.putAll(mp);
}
```

```
    return info;  
}
```

4.4.3 请求参数预处理

@ControllerAdvice 结合 @InitBinder 还能实现请求参数预处理，即将表单中的数据绑定到实体类上进行一些额外处理。
例如有两个实体类 Book 和 Author，代码如下（两个类中都定义了 name 属性）：

```
public class Book {  
  
    private String name;  
    private String author;  
    .....  
}  
  
public class Author {  
  
    private String name;  
    private int age;  
    .....  
}
```

页面，要输入 book 和 Author 信息发送到控制器。

书名	<input type="text" value="Java"/>	作者	<input type="text" value="Wang"/>
作者	<input type="text" value="LinXin"/>	年龄	<input type="text" value="12"/>
<input type="button" value="提交"/>			

为了在控制器中，能正确的为“模型”赋值。做如下修改（为输入参数添加@ModelAttribute 注解）：

```
@RequestMapping("/bookInfo")
public String putBookInfo(@ModelAttribute("b") Book book,@ModelAttribute("a") Author author) {
    System.out.println(book);
    System.out.println(author);
    return "ok";
}
```

在 html 页面，做如下改动（为 name 添加前缀）：

```
<form action="/bookInfo" >
  <table>
    <tr>
      <td>书名</td>

      <td><input type="text" name="b.name"/></td>

    <td>作者</td>
    <td><input type="text" name="author"/></td>
  </tr>
```

```
<tr>
  <td>作者</td>

  <td><input type="text" name="a.name" /></td>

  <td>年龄</td>
  <td><input type="text" name="age"/></td>
</tr>
<tr>
  <td colspan="4">
    <input type="submit" />
  </td>
</tr>
</table>
</form>
```

设置请求参数预处理:

```
@ControllerAdvice
public class GlobalConfig {

    @InitBinder("b")
    public void init(WebDataBinder binder) {
        binder.setFieldDefaultPrefix("b.");
    }

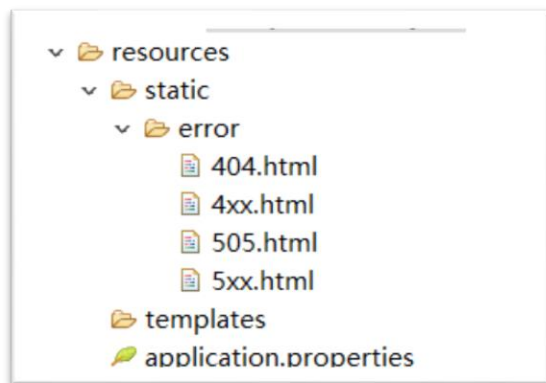
    @InitBinder("a")
    // 对应控制器中参数设置了@ModelAttribute("a")的入参
    // 对应控制器中参数设置了@ModelAttribute("b")的入参
    // HTML中以b.开头的的数据
```

```
public void init2(WebDataBinder binder) {  
    binder.setFieldDefaultPrefix("a.");           // HTML中以a.开头的数据  
  
}  
}
```

4.5 自定义错误页面

上一节介绍了使用@ControllerAdvice 和@ExceptionHandler 处理全局异常。但是这种异常是应用级别的异常，有一些容器级别的异常处理不了，例如 Filter 中抛出的异常，使用@ControllerAdvice 定义的全局异常处理机制就无法处理。这正是本节要介绍的。

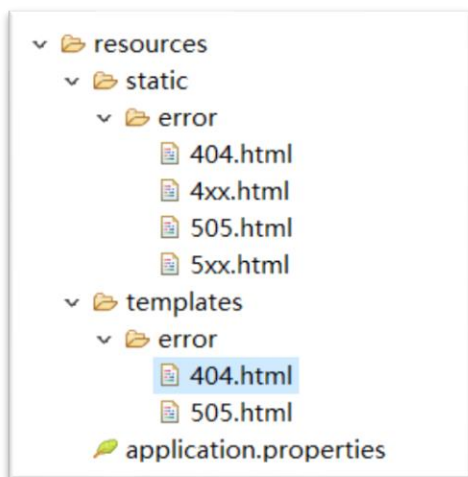
4.5.1 静态犯错误页面



我们在 static 目录下创建 error 目录，在该目录中创建 4xx.html 和 5xx.html 文件，当发生这一类错误时，将会显示这些页面。也可以根据具体的错误号建立不同的文件，如：400.html、404.html、500.html、505.html。有具体的错误号的文件的优先级大于 4xx 或 5xx 的文件。

4.5.2 动态错误页面

我们可以在 templates 目录下，建立如下 Error 模板页面。



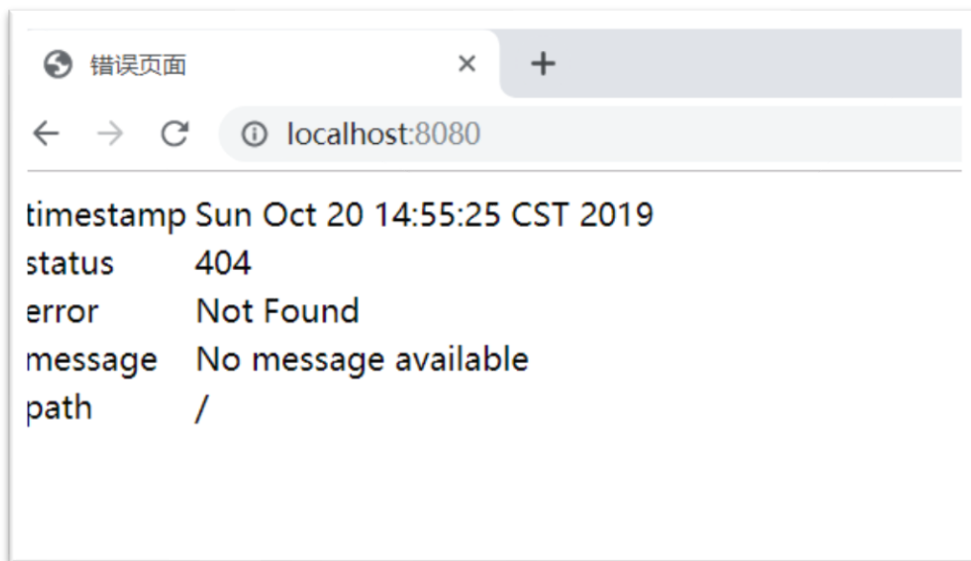
在页面中取得 DefaultErrorAttributes 中的参数。当有动态错误页面时，它的优先级高于静态页面。

以 404.html 为例，页面内容如下：（需 thymeleaf 模板支持）

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8">
<title>错误页面</title>
```

```
</head>
<body>
  <table>
    <tr>
      <td>timestamp</td>
      <td th:text="${timestamp}"></td>
    </tr>
    <tr>
      <td>status</td>
      <td th:text="${status}"></td>
    </tr>
    <tr>
      <td>error</td>
      <td th:text="${error}"></td>
    </tr>
    <tr>
      <td>message</td>
      <td th:text="${message}"></td>
    </tr>
    <tr>
      <td>path</td>
      <td th:text="${path}"></td>
    </tr>
  </table>
</body>
</html>
```

当访问一个不存在的网页，返回的内容如下：



4.6CORS 支持

CORS (Cross-Origin Resource Sharing) 是由 W3C 制定的一种跨域资源共享技术标准，其目的是解决前端的跨域请求。在常用的跨域请求方案中，最常见的是 JSONP，但是 JSONP 只支持 Get 请求，这是一个很大的缺陷。而 CORS 则支持多种 HTTP 请求。

4.6.1 @CrossOrigin

SpringBoot 使用@CrossOrigin 注解来为响应添加 Access-Control-Allow-Origin 响应头信息。

1) Controller 代码:

```
@RestController
public class MyController {

    @PostMapping("/book/add")
    // maxAge表示探测请求的有效期, 单位为分钟 (put\delete方法有探测行为)
    @CrossOrigin(origins = "http://192.168.50.48", maxAge=1800, allowedHeaders = "**")
    public String addBook(String name) {
        return "receive:"+name;
    }

    @DeleteMapping("/book/{id}")
    @CrossOrigin(origins = "http://192.168.50.48", maxAge = 1800, allowedHeaders = "**")
    public String del(@PathVariable int id) {
        return "delete:"+id;
    }

}
```

2) Html 代码

post.html:

```
<script>
    $.post("http://192.168.50.193:8080/book/add",{"name":"SpringBoot ADV"},function(data, status, xhr){
        alert(data);
    });
</script>
```

del.html

```
<script>
    $.ajax({
        url:"http://192.168.50.193:8080/book/123",
        method:"DELETE",
        success:function(data, status , xhr){
            alert(data);
        }
    });
</script>
```

HTML 页面，运行在 Nginx 服务器下：

```
server{
    listen 80 default_server;
    listen [::]:80 default_server;
    root /var/www/html;
    .....
}
```

4.6.2 CORS 全局配置

这种配置方式是细粒度的了，可以控制到每一个方法上。当然也可以采用全局配置,如下：

全局配置需自定义实现 WebMvcConfigure 接口，然后实现接口中 addCorsMapping 方法。

```
@Configuration
public class MyWebMvcConfiguration implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        CorsRegistration cr = registry.addMapping("/book/**");
        cr.allowedHeaders("*");
        cr.allowedMethods("post", "delete", "get", "put");
        cr.maxAge(1800);
        cr.allowedOrigins("http://192.168.50.48");
    }
}
```

4.7 配置类与 XML 配置

Spring Boot 推荐使用 Java 来完成相关配置工作。在项目中，不建议将所有配置放在一个配置类中，可以根据不同的需求提供不同的配置类，例如专

门处理 Spring Security 的配置类、提供 Bean 的配置类、Spring MVC 的配置类。这些配置类上都需要添加@Configuration 注解。@ComponentScan 注解会扫描所有 Spring 组件。

Spring Boot 中并不推荐使用 XML 配置，建议尽量用 Java 配置代替 XML 配置。如果想使用 XML 配置，只需要在 resources 目录下提供配置文件，然后通过@ImportResource 加载配置文件。例如：

Bean 定义：book.java

```
public class Book implements Serializable{

    private String name;
    private String author;
    private float price;
    .....
}
```

在 resources 目录下建立 beans.xml 文件：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="javaBook" class="com.profound.ch04.Book">
        <property name="name" value="Java"></property>
        <property name="author" value="WangRui"></property>
        <property name="price" value="34.5"></property>
    </bean>
```

```
</beans>
```

创建 Beans 配置类：

```
@Configuration
@ImportResource("classpath:beans.xml")
public class Beans {

}
```

在控制器中直接导入 Book。

```
@RestController
public class MyController {

    @Autowired
    private Book book;

    @GetMapping("/book")
    public Book getBook() {
        return book;
    }

}
```

4.8 注册拦截器

首先创建一个拦截器类，该类需实现 `HandlerInterceptor` 接口。

```
public class CharSetInterceptor implements HandlerInterceptor{

    @Override    // 该方法在handle()方法之前运行。
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
        throws Exception {
        System.out.println("-----CharSetInterceptor-----");
        request.setCharacterEncoding("utf-8");
        return true;
    }

    @Override    // 该方法在handle()方法之后运行。
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,
        ModelAndView modelAndView) throws Exception {

    }

    @Override    // 该方法在请求完成（页面已经返回）之后运行。
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)
        throws Exception {

    }
}
```

```
}
```

配置拦截器:

```
@Configuration
public class MyWebMvcConfiguration implements WebMvcConfigurer {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        InterceptorRegistration ir = registry.addInterceptor(new CharSetInterceptor());
        ir.addPathPatterns("/*");
    }
}
```

4.9整合 Servlet、Filter、Listener

一般情况下、使用 Spring MVC 这些框架之后，基本上告别了 Servlet、Filter、Listener，但是有时可能还会使用到。

Servlet，一个返回随机数图片的 Servlet:

```
@WebServlet("/imgcode")
public class ImgServlet extends HttpServlet{

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
```

```
BufferedImage img = new BufferedImage(100, 50, BufferedImage.TYPE_3BYTE_BGR);
Graphics g = img.getGraphics();

g.setColor(Color.WHITE);
g.fillRect(0, 0, 100, 50);

int ran = (int)(Math.random()*10000);
g.setColor(Color.RED);
g.drawString(ran+"", 20, 30);

resp.setContentType("image/jpeg");

ImageIO.write(img, "jpg", resp.getOutputStream());
}
}
```

过滤器，过滤器中注入 Spring 管理的 Bean 对象。

```
@WebFilter("/*")
public class AutoLoginFilter implements Filter{

    @Autowired
    private Book javaBook;

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
```

```
        throws IOException, ServletException {
        System.out.println(this.javaBook);
        chain.doFilter(request, response);
    }
}
```

监听器: ServletContextListener (注: SessionListen 没有成功, 可能不支持了。有待确认)

```
@WebListener
public class NewUserListener implements ServletContextListener {

    @Autowired
    private Book javaBook;

    @Override
    public void contextInitialized(ServletContextEvent sce) {
        System.out.println("=====NewUserListener=====");
        System.out.println(this.javaBook);
    }

    @Override
    public void contextDestroyed(ServletContextEvent sce) {
        System.out.println("=====NewUserListener=====");
    }
}
```


在项目的入口类上添加@ServletComponentScan 注解，实现对 Servlet、Filter、Listener 的支持。

```
@SpringBootApplication
@ServletComponentScan
public class Ch042Application {

    public static void main(String[] args) {
        SpringApplication.run(Ch042Application.class, args);
    }

}
```

4.10 路径映射

一般情况下，使用了页面模板后，用户需要通过控制器才能访问页面。有一些页面需要在控制器中加载数据，然后渲染，才能显示出来；还有一些页面在控制器中不需要加载数据，只是完成简单的跳转，对于这样的页面，可以直接设置路径映射，提高访问速度。

- resources
 - static
 - templates
 - error
 - protocol.html

templates 目录下 protocol.html 是没有动态数据的纯静态页面。

设置路径映射如下：

```
@Configuration
public class MyWebMvcConfiguration implements WebMvcConfigurer {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/protocol").setViewName("protocol");
    }
}
```

4.11 配置 AOP

Spring Boot 在 Spring 的基础上对 AOP 配置提供了自动化解决方案 spring-boot-starter-aop。

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

定义 Service 类：

```
@Service
```

```
public class UserService {  
  
    public String login() {  
        System.out.println("-----用户登录-----");  
//        int i = 1/0;  
        return "---Wang-----";  
    }  
}
```

附加逻辑类:

```
@Component  
@Aspect  
public class MonitorAdvice {  
  
    @Pointcut("execution(* com.profound.ch04.service.*Service.*(..))") // 使用注解定义一个切点  
    public void pc() {}  
  
    @Around("pc()") // 使用切点。（这里直接写切点表达式也是同样的）  
    public void provessTime(ProceedingJoinPoint pjp) throws Throwable {  
        long start = System.currentTimeMillis();  
        System.out.println("-----开始计时-----");  
        pjp.proceed();  
        System.out.println("-----用时" + (System.currentTimeMillis()-start) + "毫秒-----");  
    }  
}
```

```
@Before("pc()")
public void start(JoinPoint jp) {
    System.out.println(jp.getSignature().getName());
    System.out.println("将开始执行");
}

@After("pc()")
public void logging(JoinPoint jp) {
    System.out.println("jp:"+jp.getSignature().getName());
    System.out.println("===== end =====");
}

@AfterReturning(pointcut = "pc()", returning = "result" ) // 获取返回值
public void getReturn(JoinPoint jp, Object result) {
    System.out.println(jp.getSignature().getName());
    System.out.println("return : " + result);
}

@AfterThrowing(pointcut="pc()", throwing = "ex" ) // 获取异常对象
public void afterExec(JoinPoint jp, Exception ex) {
    System.out.println(jp.getSignature().getName());
    System.out.println(ex.getMessage());
}

}
```

4.12 自定义欢迎页面

Spring Boot 项目在启动后，首先会去静态资源路径下查找 index.html 作为首页，若查找不到，则会去找动态 index.html 文件。

因此，如想使用静态 index.html 文件，做为项目首页，需创建在 resources/static/index.html 文件。如使用动态页面，需创建 resource/templates/index.html 文件。

当然，动态页面，还需要加载数据，可以定义控制器映射到"/index"。

```
@RequestMapping("/")
public String index(Map m) {
    m.put("msg", "hello world");
    return "index";
}
```

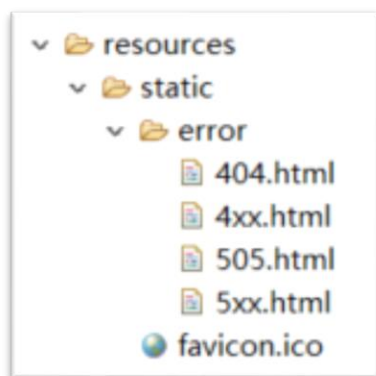
4.13 自定义 favicon

favicon 是浏览器选项卡左边的图标。可以在静态资源目录或类路径下。

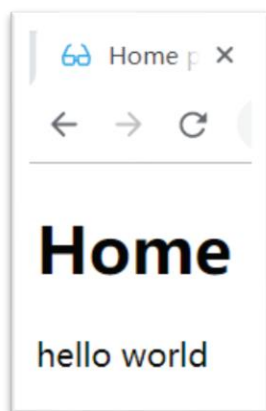
图标如下所示：



保存在静态资源目录下。



浏览器显示如下图：



4.14 除去自动配置

Spring Boot 中提供了大量的自动化配置类，如 `ErrorMvcAutoConfiguration`、`ThymeleafAutoConfiguration`、`FreeMarkAutoConfiguration`、`MultipartAutoConfiguration` 等。在 Spring Boot 入口类上有一个 `@SpringBootApplication` 注解，该注解是一个组合注解，由 `@SpringBootConfiguration`、`@EnableAutoConfiguration` 以及 `@ComponentScan` 组成，其中 `@EnableAutoConfiguration` 注解开启自动化配置，相关配置类就会被使用，如果不想启用某个配置，可以如下方式关闭：

方式 1，修改注解参数：

```
@SpringBootApplication
@EnableAutoConfiguration(exclude = {ErrorMvcAutoConfiguration.class}) // 去除错误页面的自动配置
public class Ch042Application {

    public static void main(String[] args) {
        SpringApplication.run(Ch042Application.class, args);
    }

}
```

方式 2，在 SpringBoot 配置文件中修改：

```
spring:
  autoconfigure:
    exclude:
      - org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration
```

5 SpringBoot 整合 NoSQL

本章概要

整合 Redis

Session 共享

NoSQL 是指非关系型数据库，非关系型数据库和关系型数据库两者之间存在许多显著的不同点，其中最重要的是 NoSQL 不使用 SQL 作为查询语言。其数据存储不需要固定的表格式，一般都有水平可扩展性的特征。NoSQL 主要分如下几种不同的类型：

- Key/Value 键值存储。这种数据库存储通常都是无数据结构的，一般被当做字符串或者二进制数据，但是数据加载速度快，典型的使用场景是处理高并发或者日志系统等，这一类的数据库的 Redis、Tokyo Cabinet 等。
- 列存储数据库。列存储数据库功能相对局限，但是查找速度快，容易进行分布式扩展，一般用于分布式文件系统中，这一类的数据库有 HBase、Cassandra 等。
- 文档型数据库。和 Key/Value 键值存储类似，文档型数据库也没有严格的数据格式。不需要创建表结构，数据格式更加灵活，一般可用在 Web 应用中，这一类数据库有 MongoDB、CouchDB 等。
- 图形数据库。专注构建关系图谱，例如社交网络，推荐系统等，这类数据库有 Neo4J、DEX 等。

5.1 整合 Redis

redis 是一个 key-value 存储系统。和 Memcached 类似，它支持存储的 value 类型相对更多，包括 string(字符串)、list(链表)、set(集合)、zset(sorted set --有序集合)和 hash (哈希类型)。这些数据类型都支持 push/pop、add/remove 及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的。在此基础上，redis 支持各种不同方式的排序。与 memcached 一样，为了保证效率，数据都是缓存在内存中。区别的是 redis 会周期性的把更新的数据写

入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了 master-slave(主从)同步。

Redis 是一个高性能的 key-value 数据库。redis 的出现，很大程度补偿了 memcached 这类 key/value 存储的不足，在部分场合可以对关系数据库起到很好的补充作用。它提供了 Java，C/C++，C#，PHP，JavaScript，Perl，Object-C，Python，Ruby，Erlang 等客户端，使用很方便。

5.1.1 Redis 安装

- 1) 从 <https://redis.io> 下载 redis-5.0.5.tar.gz。
- 2) 解压：tar -zxvf redis-5.0.5.tar.gz 。解压后的文件夹为：redis-5.0.5
- 3) 进入目录：cd redis-5.0.5
- 4) 编译：在 redis-5.0.5 目录下输入 make
- 5) 启动服务：进入 src 目录，执行./redis-server。
- 6) 打开客户端工具测试：进入 src 目录，执行./redis-cli

5.1.2 Redis 配置

在 redis-5.0.5 目录下有 redis.conf 文件，这是 Redis 数据库配置文件。

修改配置如下：

```
# 设置为守护线程，在后台运行
daemonize    yes

# 默认只能从本机访问，注释掉，可以从任何电脑访问。
# bind 127.0.0.1
```

```
# 设置连接到数据库的密码
```

```
requirepass 123456
```

```
# 关闭保护模式
```

```
protected-mode no
```

如果要指明使用这个配置文件在启动 Redis 数据库时命令如下：`./redis-server ../redis.conf`。

客户端登录命令使用密码：`./redis-cli -a 123456` 或

```
./redis-cli
```

```
127.0.0.1:6379> auth 123456
```

5.1.3 SpringBoot 整合 Redis

1) Redis 的 Java 客户端有 Jedis、JRedis、Lettuce 等。要想使用 Redis 需添加 Redis 依赖：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-pool2</artifactId>
</dependency>
```

SpringBoot2 开始默认使用 Lettuce。这已经包含在 `spring-boot-starter-data-redis` 依赖中。

3) 配置 Redis

在 application.yml 或 application.properties 中设置如下（本例使用 yml）：

```
server:
  port: 9090

spring:
  redis:
    host: 192.168.50.29 # Redis 主机地址
    port: 6379 # Redis 端口号
    password: 123456 # Redis 密码
    database: 0 # Redis 数据库编号
    lettuce:
      pool:
        max-active: 8 # Lettuce 连接池最大连接数
        max-idle: 8 # 最大空闲连接数
        max-wait: -1ms # 最大等待时间，默认为-1，表示没有限制。
        min-idle: 0 # 最小空闲连接数。
```

4) 创建实体类

注意：实体类一定要实现 Serializable 接口。否则在将对象序列化，写到数据库时会出错。

```
public class Book implements Serializable{
    private String name;
    private String author;
    private float price;
    .....省略 get\set\构造和 toString 方法
```

```
}
```

5) 创建 Controller

```
@RestController
public class MyController {

    @Autowired
    private StringRedisTemplate strRedisTemplste;

    @Autowired
    private RedisTemplate redisTemplate;

    @GetMapping("/test")
    public void test() {
        ValueOperations<String, String> sop = strRedisTemplste.opsForValue();
        sop.set("name", "WangRui");
        String name = sop.get("name");
        System.out.println(name);

        ValueOperations op = redisTemplate.opsForValue();
        Book book = new Book("Python", "WangRui", 10.4F);
        op.set("Python", book);
        System.out.println(op.get("Python"));
    }
}
```

```
}
```

运行结果:

WangRui

Book [name=Python, author=WangRui, price=10.4]

通过 redis-cli 查看数据库中的数据:

```
root@debian-vm:/home/wangrui# redis-cli -a 123456
```

```
127.0.0.1:6379> keys *
```

```
1) "name"
```

```
2) "\xac\xed\x00\x05t\x00\x06Python"
```

```
127.0.0.1:6379>
```

```
127.0.0.1:6379> get "\xac\xed\x00\x05t\x00\x06python"
```

```
"\xac\xed\x00\x05sr\x00\x1ccom.profound.sbootredis.Book\xfc\x3&\xbf\xdc\xe4\x05\x02\x00\x03F\x00\x05priceL\x00\x06authort\x00\x12Ljava/lang/
String;L\x00\x04nameq\x00~\x00\x01xpA&ft\x00aWangRuit\x00\x06Python"
```

发现多了很多“\xac\xed\x00\x05t\x00”这样的东西。无论是 Key 中还是 Value 中。这是因为 Spring 提供的 RedisTemplate 在底层的实现上的原因。如果想去掉这些“\xac\xed\x00\x05t\x00”我们也可以自定义这个 RedisTemplate。如下:

```
@Configuration
```

```
@AutoConfigureBefore(value = RedisAutoConfiguration.class) // SpringBoot在 RedisAutoConfiguration中提供了默认的RedisTemplate。
```

```
public class RedisTemplateConfigruation {
```

```
    @Bean(name = "redisTemplate") // 要指定这个Bean的名字, 才可以替换SpringBoot默认提供的RedisTemplate。
```

```
    public RedisTemplate<String, Serializable> createRedisTemplate(RedisConnectionFactory factory) {
```

```
        RedisTemplate<String, Serializable> tmp = new RedisTemplate<>();
```

```
        tmp.setConnectionFactory(factory);
```

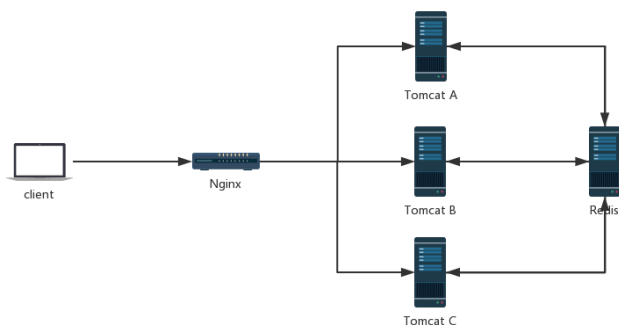
```
StringRedisSerializer strSerializer = new StringRedisSerializer();
tmp.setKeySerializer(strSerializer);
RedisSerializer<Object> valueSerialize = new GenericJackson2JsonRedisSerializer();
tmp.setValueSerializer(valueSerialize);
return tmp;
}
}
```

测试使用自定义的 RedisTemplate，在 Redis 数据库中写入内容如下：

```
127.0.0.1:6379> keys Python
1) "Python"
127.0.0.1:6379>
127.0.0.1:6379> get Python
"{\"@class\":\"com.profound.sbootredis.Book\",\"name\":\"Python\",\"author\":\"WangRui\",\"price\":10.4}"
127.0.0.1:6379>
```

5.1.4 共享 Session

当一个服务器在性能上不能满足用户需求的情况下。多台服务器组成集群处理用户的请求是常用的方式。



我们以一个 Nginx 作为反射代理服务器，在其后有多个服务器。这些服务器共享 Session。

应用场景：

当所有 Tomcat 需要往 Session 中写数据时，都往 Redis 中写，当所有 Tomcat 需要读数据时，都从 Redis 中读。这样，不同的服务就可以使用相同的 Session 数据了。

这样的方案，可以由开发者手动实现，即手动往 Redis 中存储数据，手动从 Redis 中读取数据，相当于使用一些 Redis 客户端工具来实现这样的功能，毫无疑问，手动实现工作量还是蛮大的。

一个简化的方案就是使用 Spring Session 来实现这一功能，Spring Session 就是使用 Spring 中的代理过滤器，将所有的 Session 操作拦截下来，自动的将数据 同步到 Redis 中，或者自动的从 Redis 中读取数据。

对于开发者来说，所有关于 Session 同步的操作都是透明的，开发者使用 Spring Session，一旦配置完成后，具体的用法就像使用一个普通的 Session 一样。

1) 添加 Spring Session 依赖：

```
<dependency>
  <groupId>org.springframework.session</groupId>
  <artifactId>spring-session-data-redis</artifactId>
</dependency>
```

2) 在 application.yml 中添加 SpringSession 存储（非必须）

```
server:
  port: 9090

spring:
  redis:
    host: 192.168.50.48
    port: 6379
    password: 123456
    database: 0
    lettuce:
      pool:
        max-active: 8
        max-idle: 8
        max-wait: -1ms
        min-idle: 0

    session:
      store-type: redis
```

3)

4) Nginx

```
upstream profound.com{
    server 127.0.0.1:8088 weight=1;
    server 127.0.0.1:8089 weight=1;
}
```

设置集群。这个网址不必真实存在
weight 这置这个服务器的权重，性能好的可定义高些。


```
server{
    listen 8080;
    server_name localhost;
    location /{
        proxy_pass http://profound.com;
        proxy_redirect default;
    }
}
```

5) Controller 进行测试

```
@RestController
public class SessionTestController {

    @Value("${server.port}")
    private int port;

    @GetMapping("/save")
    public Map save(String msg, HttpSession session) {
        Map map = new HashMap();
        map.put("port", this.port);
        map.put("msg", msg);

        session.setAttribute("msg", msg);
    }
}
```

```
        return map;
    }

    @GetMapping("/get")
    public Map get(HttpSession session) {
        Map map = new HashMap();
        Object msg = session.getAttribute("msg");

        map.put("port", this.port);
        map.put("msg", msg);

        return map;
    }
}
```

6) 测试

测试赋值: <http://localhost:8080/save?msg=hello>

返回: {"port":8080, "msg":"hello"}

测试取值: <http://localhost:8080/get>

返回: {"port":8088, "msg":"hello"}

测试取值: <http://localhost:8080/get>

返回: {"port":8089, "msg":"hello"}

6 Spring Boot 整合 MQ

本章概要

原生 JMS API

使用 JmsTemplate

6.1 原生 JMS 访问 ActiveMQ

Java 中定义了标准的 JMS API，通过具体 JMS 的实现，可以实现相应的功能。本例以 ActiveMQ 为消息队列中间件的实现。

示例 1：发送 P2P 消息。本示例中，一个子线程运行接收消息。主线程从键盘输入消息，发送 10 次。在到 10 次后程序结束。

```
public class MessageUtil {  
  
    private static QueueConnectionFactory factory;  
  
    static {  
        factory = new ActiveMQConnectionFactory("user", "user", "tcp://192.168.50.48:61616");  
    }  
  
    public static void send(String msg) throws JMSException {  
        Session session = null;  
        QueueConnection connection = null;  
        ;  
    }  
}
```

```
try {
    connection = factory.createQueueConnection();
    connection.start(); // 这句可不能少。
    session = connection.createQueueSession(true, QueueSession.AUTO_ACKNOWLEDGE);
    Queue msgQueue = session.createQueue("msgQueue");

    MessageProducer producer = session.createProducer(msgQueue);
    producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT); // 没有监听会失效的消息

    Message message = session.createTextMessage(msg);

    producer.send(message);

    session.commit();
} catch (JMSException ex) {

    ex.printStackTrace();
} finally {
    connection.close();
}

}

public static void receive() throws JMSException {
    QueueConnection connection = null;
    Session session = null;
    try {
```

```
connection = factory.createQueueConnection();
connection.start();
session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
Queue msgQueue = session.createQueue("msgQueue");
MessageConsumer consumer = session.createConsumer(msgQueue);

Message msg = consumer.receive();
if(msg!=null) {
    TextMessage txtMsg = (TextMessage) msg;
    System.out.println(txtMsg.getText());
}

session.commit();
} catch (JMSEException e) {
    e.printStackTrace();
} finally {
    connection.close();
}
}

public static void main(String[] args) throws NamingException, JMSEException {

    Thread t = new Thread() {
        public void run() {
```

```
        try {
            while (true) {
                System.out.println("----will receive-----");
                MessageUtil.receive();
            }
        } catch (JMSEException e) {
            e.printStackTrace();
        } finally {
            System.out.println("-----thread end-----");
        }

    }

};

t.setDaemon(true);

t.start();

// -----
Scanner sc = new Scanner(System.in);
for (int i = 0; i < 10; i++) {
    if (sc.hasNext()) {
        String msg = sc.next();
        send(msg);
    }
}
```

```
    }  
    sc.close();  
  
}  
}
```

示例 2：发布与监听订阅的消息。生产者发布消息到一个“主题”，消费者监听“主题”。消费者先启动了监听。

```
public class PublishMQ {  
  
    private static ConnectionFactory factory;  
  
    static {  
        factory = new ActiveMQConnectionFactory("admin", "admin", "tcp://127.0.0.1:61616");  
    }  
  
    // 发布消息  
    public static void publish(String msg) throws JMSException {  
        Connection conn = null;  
        Session session = null;  
        MessageProducer producer = null;  
  
        try {  
            conn = factory.createConnection();  
            conn.start();
```

```
        session = conn.createSession(true, Session.AUTO_ACKNOWLEDGE);

        Topic topic = session.createTopic("about1902");

        producer = session.createProducer(topic);

        TextMessage message = session.createTextMessage(msg);

        producer.send(message);

        session.commit();
    } catch (JMSEException e) {
        session.rollback();
        e.printStackTrace();
    } finally {
        producer.close();
        session.close();
        conn.close();
    }
}

// 监听消息
public static void listenMsg() throws JMSEException {
```



```
try {
    final Connection conn = factory.createConnection();
    conn.start();

    final Session session = conn.createSession(true, Session.AUTO_ACKNOWLEDGE);
    Topic topic = session.createTopic("about1902");
    final MessageConsumer consumer = session.createConsumer(topic);

    consumer.setMessageListener(new MessageListener() {

        @Override
        public void onMessage(Message message) {
            TextMessage msg=(TextMessage) message;
            String txt;
            try {
                txt = msg.getText();
                System.out.println("收到:" + txt);
                session.commit();
            } catch (JMSException e) {

                e.printStackTrace();
            } finally {

                try {
                    consumer.close(); // 在收到消息后，如果这些资源都不释放，将一直监听下去。程序不会退出。
                } catch (JMSException e) {
                    e.printStackTrace();
                }
            }
        }
    });
}
```

```
        session.close();
        conn.close();
    } catch (JMSEException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}

});

} catch (JMSEException e) {

    e.printStackTrace();
} finally {

}

}

public static void main(String[] args) throws JMSEException {
    listenMsg();

    publish("hello world!!!!");
```

```
}
```

```
}
```

6.2 Spring Boot 发送消息

配置 ActiveMQ 服务

```
# activeMQ
spring.activemq.broker-url=tcp://127.0.0.1:61616
spring.activemq.user=user
spring.activemq.password=user
spring.activemq.packages.trust-all=true
```

SpringBoot 实现：Spring 管理 Queue 对象。使用 Spring 提供的 JmsTemplate 对象，发送消息。

```
@Configuration
public class MyConfiguration {

    @Bean
    public Queue queue() {
        return new ActiveMQQueue("emailMQ");
    }
}
```

```
@Service
public class UserService {

    @Autowired
    private JmsTemplate jmsTemplate;

    @Autowired
    private Queue emailQueue;

    public void sendToMQ(User user) {
        this.jmsTemplate.convertAndSend(emailQueue, user);
    }
}
```

6.3 SpringBoot 接收消息

Spring Boot 接收消息，是在另一个 Spring Boot 项目中。该项目也实现为一个 Spring Boot Web 项目（也可以不是 Web 项目，这里做为 Web 项目的目的，是让该应用启动后不会立即退出）。

application.properties 配置 ActiveMQ 服务：

```
spring.activemq.broker-url=tcp://127.0.0.1:61616
spring.activemq.user=user
spring.activemq.password=user
spring.activemq.packages.trust-all=true

server.port=9999
```

监听消息的代码：在启动器中直接配置了@JmsListener 指定对某队列的监听程序，对收到的消息进行处理。（监听处理也可以写到一个@Configuration 类中。）

```
@SpringBootApplication
public class MailserverApplication {

    @Autowired
    private JavaMailSender javaMailSender;

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

    @JmsListener(destination = "emailQueue")
    public void receive(Message msg) throws JMSEException {
        System.out.println("receive:" + msg);
        ActiveMQObjectMessage m = (ActiveMQObjectMessage) msg;
        User u = (User) m.getObject();
        System.out.println(u);
    }
}
```

}

7 SpringBoot 发送邮件

Java Mail 提供了发送邮件的功能，但是比较繁琐。Spring 提供了 JavaMailSender 工具类，可以方便快捷的发送邮件。

在 application.properties 中配置邮件服务器：

```
spring.mail.host=smtp.qq.com
spring.mail.port=465
spring.mail.username=392977758@qq.com #你的登录邮箱的账户名
spring.mail.password= #QQ 邮件给的授权码
spring.mail.default-encoding=utf-8
spring.mail.properties.mail.smtp.socketFactory.class=javax.net.ssl.SSLSocketFactory
```

Java 代码实现：

```
@Service
public class Mailserver {

    @Autowired
    private JavaMailSender javaMailSender;

    public void sendMail(String msg) {

        // 设置消息
        SimpleMailMessage mail = new SimpleMailMessage();
        mail.setTo("392977758@qq.com");
        mail.setFrom("392977758@qq.com"); // 发件人，必须与登录邮件配置中一致，否则出错。
    }
}
```

```
        mail.setSubject("regist active");  
        mail.setText(msg);  
        mail.setSentDate(new Date());  
  
        this.javaMailSender.send(mail); //发送邮件  
    }  
  
}
```